Абстракция данных и решение задач на C++

СТЕНЫ И ЗЕРКАЛА



ТРЕТЬЕ ИЗДАНИЕ

Фрэнк М. Каррано
Джанет Дж. Причард

По мотивам книги "Стены и зеркала" Пола Хелмана и Роберта Веррoффа

ВИЛЬЯМС

ADDISON WESLEY

АБСТРАКЦИЯ ДАННЫХ И РЕШЕНИЕ ЗАДАЧ НА C++

СТЕНЫ И ЗЕРКАЛА

Третье издание

DATA ABSTRACTION AND PROBLEM SOLVING WITH C++

WALLS AND MIRRORS

Third Edition

Frank M. Carrano
University of Rhode Island

Janet J. Prichard
Bryant College



ADDISON-WESLEY PUBLISHING COMPANY

Boston • San Francisco • New York
London • Toronto • Sydney • Tokyo • Singapore • Madrid
Mexico City • Munich • Paris • Cape Town • Hong Kong • Montreal

АБСТРАКЦИЯ ДАННЫХ И РЕШЕНИЕ ЗАДАЧ НА C++

СТЕНЫ И ЗЕРКАЛА

Третье издание

Фрэнк М. Каррано
Университет Род Айленд

Джанет Дж. Причард
Брайант колледж



Москва • Санкт-Петербург • Киев
2003

ББК 32.973.26-018.2.75
К26
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *А.В. Слепцов*

Перевод с английского и редакция канд. физ.-мат. наук *Д.А. Клюшина*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Каррано Ф.М., **Причард** Дж.Дж.

К26 Абстракция данных и решение задач на C++. Стены и зеркала, 3-е издание. : Пер. с англ. — М.: Издательский дом "Вильямс", 2003. — 848 с.: ил. — Парал. тит. англ.

ISBN 5-8459-0389-0 (рус.)

Книга представляет собой классический учебник для высшей школы, содержащий глубокое изложение вопросов, связанных с абстракцией и структурами данных, а также их реализацией на языке C++. Помимо предоставления прочных основ методов абстракции данных, в ней особо подчеркивается различие между спецификацией и реализацией, что является принципиально важным в объектно-ориентированном подходе. В книге подробно обсуждаются ключевые понятия объектно-ориентированного программирования, включая инкапсуляцию, наследование и полиморфизм, однако в центре внимания всегда находится именно абстракция данных, а не синтаксические конструкции языка C++.

Книга будет полезна всем, кто заинтересован в глубоком изучении важнейших аспектов ООП и полном освоении соответствующих возможностей языка C++.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Pearson Education, Inc, Copyright © 2002

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2003

ISBN 5-8459-0389-0 (рус)
ISBN 0-2017-4119-9 (англ.)

© Издательский дом "Вильямс", 2003
© Pearson Education, Inc., 2002

Оглавление

Предисловие 13

ЧАСТЬ I. МЕТОДЫ РЕШЕНИЯ ЗАДАЧ 23

Глава 1. Принципы программирования и разработки программного обеспечения 24

Глава 2. Рекурсия: зеркала 69

Глава 3. Абстракция данных: стены 123

Глава 4. Связанные списки 169

Глава 5. Рекурсивный метод решения задач 236

ЧАСТЬ II. РЕШЕНИЕ ЗАДАЧ С ПОМОЩЬЮ АБСТРАКТНЫХ ТИПОВ ДАННЫХ 267

Глава 6. Стеки 268

Глава 7. Очереди 319

Глава 8. Особенности языка C++ 358

Глава 9. Эффективность алгоритмов и сортировка 408

Глава 10. Деревья 455

Глава 11. Таблицы и очереди с приоритетами 535

Глава 12. Эффективные реализации таблиц 579

Глава 13. Графы 645

Глава 14. Методы работы с внешними запоминающими устройствами 681

Приложение А. Основы языка C++ 719

Приложение Б. ASCII-коды символов 788

Приложение В. Заголовочные файлы и стандартные функции в языке C++ 790

Приложение Г. Метод математической индукции 795

Приложение Д. Стандартные шаблонные классы 800

Приложение Е. Операторы языка C++ 803

Словарь терминов 806

Ответы на вопросы для самопроверки 825

Предметный указатель 844

Содержание

Предисловие 13
Обращение к студентам 13
Метод изложения 14
Необходимые условия 14
Гибкость 14
Абстракция данных 15
Решение задач 16
Приложения 16
Новый и переработанный материал 16
Обзор 17
Методические особенности 17
Организация 18
Вспомогательные материалы 19
Пишите нам 19
Благодарности 19

ЧАСТЬ I. МЕТОДЫ РЕШЕНИЯ ЗАДАЧ 23

Глава 1. Принципы программирования и разработки программного обеспечения 24
Решение задач и разработка программного обеспечения 25
Решение задачи 25
Жизненный цикл программного обеспечения 26
Хорошее решение задачи 34
Модульный подход 36
Абстракция и сокрытие информации 36
Объектно-ориентированное проектирование 38
Проектирование "сверху вниз" 40
Общие принципы проектирования 41
Моделирование объектно-ориентированных проектов с помощью языка UML 42
Преимущества объектно-ориентированного подхода 44
Краткий обзор основных понятий программирования 45
Модульность 45
Модифицируемость 47
Легкость использования 49
Надежное программирование 50
Стиль 55
Отладка 61
Резюме 63
Предупреждения 64
Вопросы для самопроверки 64
Упражнения 65
Задачи по программированию 67

Глава 2. Рекурсия: зеркала 69
Рекурсивные решения 70
Рекурсивная функция, возвращающая значение: факториал числа n 73
Рекурсивные функции, не возвращающие никаких значений: обратная запись строки 79
Перечислимые предметы 90
Размножающиеся кролики (последовательность Фибоначчи) 90
Организация парада 92
Дилемма мистера Спока (выбор k из n предметов) 94
Поиск элемента в массиве 96
Поиск наибольшего элемента в массиве 97
Бинарный поиск 98
Поиск k-го наименьшего элемента массива 102
Организация данных 105
Ханойские башни 105
Рекурсия и эффективность 109
Резюме 114
Предупреждения 115
Вопросы для самопроверки 115
Упражнения 116
Задания по программированию 122
Глава 3. Абстракция данных: стены 123
Абстрактные типы данных 124
Спецификации абстрактных типов данных 129
Абстрактный список 130
Абстрактный упорядоченный список 135
Разработка абстрактных типов данных 136
Аксиомы 140
Реализация абстрактных типов данных 142
Классы языка C++ 143
Пространства имен 152
Реализация абстрактного списка в виде массива 154
Исключительные ситуации в языке C++ 159
Реализация абстрактного списка с учетом исключительных ситуаций 160
Резюме 162
Предупреждения 163
Вопросы для самопроверки 164
Упражнения 165
Задания по программированию 167
Глава 4. Связанные списки 169
Предварительные замечания 170
Указатели 170
Динамические массивы 176
Связанные списки, основанные на указателях 179
Работа со связанными списками 181
Вывод на экран содержания связанного списка 181
Удаление указанного узла из связанного списка 183
Вставка узла в указанную позицию связанного списка 185
Реализация абстрактного списка, основанная на указателях 190
Реализации списка в виде массива и на основе указателей 197
Запись связанных списков в файл и считывание их из файла 199
Передача связанного списка в качестве аргумента функции 202

Рекурсивная обработка связанных списков 203
Объекты, хранящиеся в узлах списка 208
Разновидности связанных списков 209
Кольцевые связанные списки 209
Фиктивные головные узлы 211
Дважды связанные списки 211
Приложение: инвентарная ведомость 214
Стандартная библиотека шаблонов языка C++ 219
Контейнеры 220
Итераторы 221
Шаблонный класс list из библиотеки STL 222
Резюме 224
Предупреждения 226
Вопросы для самопроверки 227
Упражнения 229
Задания по программированию 232
Глава 5. Рекурсивный метод решения задач 236
Поиск с возвратом 237
Задача о восьми ферзях 237
Определение языков 241
Основы грамматики 242
Два простых языка 243
Алгебраические выражения 246
Связь между рекурсией и математической индукцией 254
Правильность рекурсивной функции для вычисления факториала 255
Количество ходов при решении задачи о ханойских башнях 256
Резюме 257
Предупреждения 258
Вопросы для самопроверки 258
Упражнения 258
Задания по программированию 261
ЧАСТЬ II. РЕШЕНИЕ ЗАДАЧ С ПОМОЩЬЮ АБСТРАКТНЫХ ТИПОВ ДАННЫХ 267
Глава 6. Стеки 268
Абстрактный стек 269
Разработка абстрактных типов данных в процессе решения задачи 269
Простые примеры использования абстрактного стека 274
Проверка баланса фигурных скобок 275
Распознавание строк языка 277
Реализации абстрактного стека 278
Реализация абстрактного стека в виде массива 279
Реализация абстрактного стека в виде связанного списка 282
Реализация стека в виде абстрактного списка 285
Сравнение реализаций 288
Класс stack из стандартной библиотеки шаблонов 289
Приложение: алгебраические выражения 290
Вычисление постфиксных выражений 291
Преобразование инфиксного выражения в постфиксное 292
Приложение: поиск 295
Итеративное решение с помощью стеков 297
Рекурсивное решение 304
Взаимосвязь между стеками и рекурсией 307

Резюме 308
Предупреждения 309
Вопросы для самопроверки 309
Упражнения 310
Задания по программированию 313
Глава 7. Очереди 319
Абстрактная очередь 320
Некоторые применения абстрактной очереди 322
Считывание строки символов 322
Распознавание палиндромов 323
Реализация абстрактной очереди 324
Реализация очереди в виде связанного списка 325
Реализация очереди в виде массива 330
Реализация очереди с помощью абстрактного списка 335
Шаблонный класс queue из библиотеки STL 337
Сравнение реализаций 340
Абстрактные типы данных, основанные на позиционном принципе 341
Приложение: моделирование 342
Резюме 351
Предупреждения 351
Вопросы для самопроверки 351
Упражнения 352
Задания по программированию 354
Глава 8. Особенности языка C++ 358
Еще раз о наследовании 359
Открытое, закрытое и защищенное наследование 365
Отношения “является”, “содержит” и “подобен” 366
Виртуальные функции и позднее связывание 368
Абстрактные базовые классы 373
Дружественные функции и классы 377
Новая реализация абстрактного и упорядоченного списка 380
Реализации абстрактного упорядоченного списка на основе абстрактного списка 382
Шаблонные классы 386
Перегруженные операторы 392
Итераторы 395
Реализация абстрактного списка с помощью итераторов 397
Резюме 401
Предупреждения 402
Вопросы для самопроверки 402
Упражнения 403
Задания по программированию 406
Глава 9. Эффективность алгоритмов и сортировка 408
Измерение эффективности алгоритмов 409
Быстродействие алгоритмов 410
Степень роста временных затрат 411
Оценка порядка величины и обозначение O-большое 413
Перспективы 417
Эффективность алгоритмов поиска 419
Алгоритмы сортировки и их эффективность 420
Сортировка методом пузырька 424
Сортировка методом вставок 426

Сортировка слиянием 428
Быстрая сортировка 433
Поразрядная сортировка 444
Сравнение алгоритмов сортировки 446
Резюме 447
Предупреждения 447
Вопросы для самопроверки 448
Упражнения 449
Задания по программированию 452
Глава 10. Деревья 455
Терминология 457
Абстрактное бинарное дерево 463
Обход бинарного дерева 467
Способы представления бинарного дерева 470
Реализация абстрактного бинарного дерева в виде связанного списка 474
Абстрактное бинарное дерево поиска 488
Алгоритмы, реализующие операции над абстрактным бинарным деревом поиска 492
Реализация абстрактного бинарного дерева поиска с помощью указателей 506
Эффективность операций над бинарными деревьями поиска 514
Древовидная сортировка 518
Запись бинарного дерева поиска в файл 519
Деревья общего вида 522
Резюме 524
Предупреждения 525
Вопросы для самопроверки 525
Упражнения 527
Задания по программированию 532
Глава 11. Таблицы и очереди с приоритетами 535
Абстрактная таблица 536
Выбор способа реализации 541
Реализация абстрактной таблицы в виде упорядоченного массива 548
Реализация абстрактной таблицы в виде бинарного дерева поиска 552
Абстрактная очередь с приоритетами: вариант абстрактной таблицы 555
Кучи 558
Реализация абстрактной очереди с приоритетами в виде кучи 567
Пирамидальная сортировка 569
Резюме 573
Предупреждения 574
Вопросы для самопроверки 574
Упражнения 575
Задания по программированию 577
Глава 12. Эффективные реализации таблиц 579
Сбалансированные деревья поиска 580
2-3 деревья 581
2-3-4 деревья 599
Красно-черные деревья 607
AVL-деревья 611
Хэширование 615
Функции хэширования 619
Разрешение конфликтов 621

Эффективность хэширования 629
Чем отличается хорошая функция хэширования 632
Обход таблицы: неэффективная операция при хэшировании 634
Одновременное применение нескольких структур данных 635
Резюме 640
Предупреждения 640
Вопросы для самопроверки 641
Упражнения 641
Задания по программированию 644
Глава 13. Графы 645
Терминология 646
Графы как абстрактные типы данных 649
Реализация графов 650
Алгоритмы обхода графа 653
Поиск в глубину 654
Поиск в ширину 656
Применения графов 657
Топологическая сортировка 657
Остовные деревья 661
Минимальные остовные деревья 665
Кратчайшие пути 668
Простые цепи 672
Некоторые трудные задачи 674
Резюме 676
Предупреждения 676
Вопросы для самопроверки 676
Упражнения 677
Задания по программированию 680
Глава 14. Методы работы с внешними запоминающими устройствами 681
Внешние запоминающие устройства 682
Сортировка данных во внешнем файле 685
Внешние таблицы 692
Индексирование внешнего файла 694
Внешнее хэширование 698
B-деревья 701
Алгоритмы обхода 711
Множественная индексация 713
Резюме 714
Предупреждения 715
Вопросы для самопроверки 715
Упражнения 716
Задания по программированию 718
Приложение A. Основы языка C++ 719
Основные конструкции языка 720
Комментарии 721
Идентификаторы и ключевые слова 721
Основные типы данных 721
Переменные 722
Литеральные константы 723
Именованные константы 724
Перечисления 724
Оператор typedef 725

Присваивания и выражения 725
Входной и выходной потоки 730
Ввод 730
Вывод 731
Флаги формата и манипуляторы 732
Функции 734
Стандартные функции 737
Условные операторы 737
Оператор if 737
Оператор switch 738
Операторы цикла 740
Оператор while 740
Оператор for 741
Оператор do 743
Массивы 743
Одномерные массивы 743
Многомерные массивы 745
Массивы массивов 747
Строки 748
Строки языка C++ 749
Строки языка C 750
Структуры 753
Структуры внутри других структур 755
Массивы структур 755
Исключительные ситуации 755
Перехват исключительных ситуаций 756
Генерирование исключительных ситуаций 760
Работа с файлами 762
Текстовые файлы 763
Бинарные файлы 773
Библиотеки 774
Предотвращение дублирования заголовочных файлов 775
Сравнение с языком Java 775
Резюме 780
Предупреждения 782
Вопросы для самопроверки 783
Упражнения 785
Задания по программированию 786
Приложение Б. ASCII-коды символов и ключевые слова языка C++ 788
Приложение В. Заголовочные файлы и стандартные функции в языке C++ 790
Приложение Г. Метод математической индукции 795
Вопросы для самопроверки 798
Упражнения 799
Приложение Д. Стандартные шаблонные классы 800
Приложение Е. Операторы языка C++ 803
Словарь терминов 806
Ответы на вопросы для самопроверки 825
Предметный указатель 844

Предисловие

Перед вами — книга "Абстракция данных и решение задач на C++: стены и зеркала". В ней отражен наш опыт преподавания объектно-ориентированной абстракции данных и эволюция, которой подвергся язык C++ в последнее время.

Книга написана по мотивам бестселлера Пауля Хелмана (Paul Helman) и Роберта Вероффа (Robert Veroff) *Intermediate Problem Solving and Data Structures: Walls and Mirrors*. Она посвящена тем же проблемам, так же организована, а ее техническое и литературное содержание, примеры, рисунки и упражнения созданы по образцу оригинала. Профессоры Хелман и Верофф предложили очень точную аналогию — стены и зеркала. Эта концепция облегчает изложение материала и позволяет лучше преподавать компьютерные науки.

Ориентируясь на абстракцию данных и другие средства решения задач, книга представляет собой учебник по компьютерным наукам для второго курса. Учитывая динамичное развитие этой отрасли знаний и весьма разнообразные учебные планы, принятые в разных университетах, мы включили в нее сжатое изложение материала, который может стать основой для других курсов. Например, нашу книгу можно использовать в качестве учебника и по структурам данных, и по программированию. Наша цель осталась прежней — изложить студентам основы абстракции данных, объектно-ориентированного программирования и других современных методов решения задач.

Обращение к студентам

Предыдущие издания этой книги прочли уже тысячи студентов. Стены и зеркала, упоминаемые в названии, представляют собой два основных метода решения задач. Абстракция данных изолирует и скрывает детали реализации модуля от остальной части программы, так же как стены изолируют и скрывают вас от соседей. Рекурсия — это способ сведения исходной задачи к решению задач того же типа, но имеющих меньшие размеры, так же как зеркала уменьшают изображение при каждом новом отражении.

Книга написана именно для студентов. Мы прекрасно помним, как сами были студентами, и теперь, будучи преподавателями, особенно ценим ясное изложение. Мы стремились сделать нашу книгу как можно понятнее. Чтобы облегчить процесс обучения и подготовки к экзаменам, мы разместили на полях пометки (в русском издании они размещены внутри врезок. — *Прим. ред.*), включили в главы резюме, вопросы для самопроверки с ответами, а также словарь терминов. В качестве справочника по языку C++ можно использовать приложение, приведенное в конце книги, а также информацию, помещенную в Приложениях Б и Е. Обратите внимание на характерные черты нашего учебника, изложенные в разделе "Методические особенности".

В ходе изложения мы предполагали, что читатели уже знакомы с основами языка C++. Те, кто впервые сталкивается с этим языком, могут изучить его, обратившись к приложению А. Для понимания материала, изложенного в книге, достаточно знать следующие темы: условные операторы `is` и `switch`, операторы цикла `for`, `while` и `do`, функции и способы передачи аргументов, массивы, строки, структуры и файлы. Классы в языке C++ описываются в главах 1, 3 и 8, поэтому их предварительного изучения не требуется. Кроме того, мы не предпо-

лагали, что студенты должны быть знакомы с рекурсивными функциями, поэтому включили их описание в главы 2 и 5.

Все фрагменты программ, приведенные в книге, пригодны к использованию. В конце предисловия указан адрес, откуда можно получить соответствующие файлы.

Метод изложения

Основной акцент в данном издании книги делается на абстракции данных и структурах данных. В ней тщательно учтены все преимущества и недостатки языка C++. Осталось только изложить методические принципы, которые позволяют усвоить материал начинающим студентам.

Необходимые условия

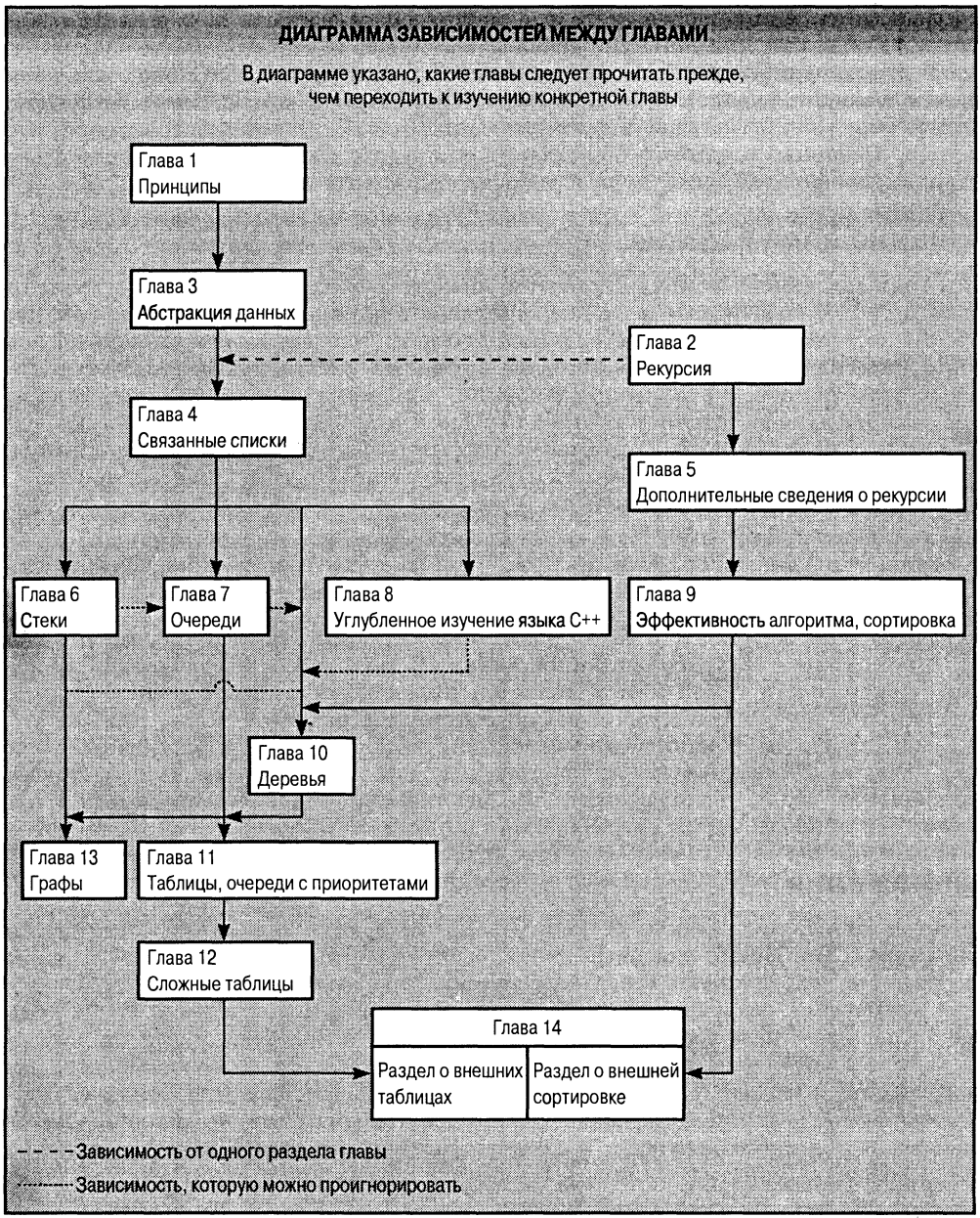
Мы предполагаем, что читатели либо уже знают язык C+, либо владеют другим языком программирования и могут прибегнуть к помощи преподавателя для перехода на язык C++, используя информацию, изложенную в Приложении А. Книга содержит формальное описание классов, поэтому предварительные знания по этой теме не требуются. Кроме того, в ней изложены основные принципы объектно-ориентированного программирования, а также темы, посвященные наследованию, виртуальным функциям и шаблонным классам. Эти вопросы тесно переплетаются с реализациями абстрактных типов данных (АТД) в виде классов, причем акцент делается именно на абстракции, а не на особенностях языка C++. Весь материал изложен в контексте объектно-ориентированного программирования. Подразумевается, что в дальнейшем студенты перейдут к изучению объектно-ориентированного проектирования и принципов разработки программного обеспечения, поэтому в центре внимания постоянно находится абстракция данных. Кроме того, в книге содержится краткое введение в универсальный язык моделирования (Universal Modeling Language — UML).

Гибкость

Книга построена таким образом, что ее можно использовать как основу разных курсов по программированию. Темы и порядок их изложения можно выбирать по своему усмотрению. Взаимные зависимости между главами изображены на диаграмме.

В первой части книги мы изложили необходимый минимум знаний. Три из этих глав посвящены подробному изложению вопросов, связанных с абстракцией данных и рекурсией. Обе эти темы очень важны, поэтому существует много точек зрения, какую из них следует излагать первой. Хотя в данном издании рекурсия описывается раньше абстракции данных, преподаватели могут менять порядок изложения по своему усмотрению.

Порядок глав во второй части книги также можно менять. Например, можно сначала изложить материал, содержащийся в главе 8, и лишь затем переходить к описанию стеков (глава 6). Способы оценки сложности алгоритмов и методы сортировки (глава 9) можно рассматривать после главы 5. Понятие дерева можно вводить до очередей, а понятие графа — до таблиц. Хэширование, сбалансированные деревья поиска или очереди с приоритетами можно описывать до таблиц, причем в любом порядке. Кроме того, методы работы с внешними запоминающими устройствами (глава 14) можно излагать раньше, чем в книге. Например, методы внешней сортировки можно описать сразу после алгоритма сортировки слиянием (глава 9).



Абстракция данных

Вопросы разработки и применения абстрактных типов данных "пронизывают" всю книгу. В ней содержится несколько примеров, позволяющих проиллюстрировать методы разработки АТД как части решения задачи. Сначала всегда указывается спецификация абстрактного типа данных, как на естественном языке, так и на псевдокоде. Затем на примере простых приложений иллюстрируется его

использование. Лишь после этого рассматриваются вопросы его реализации. В центре внимания постоянно находится различие между абстрактным типом данных и структурой данных. Инкапсуляция и классы в языке C++ вводятся уже в первых главах. Студенты имеют возможность увидеть, как с помощью классов можно скрыть реализацию структуры данных от клиента абстрактного типа данных. Основными темами обсуждения являются абстрактные списки, стеки, очереди, деревья, таблицы, кучи и очереди с приоритетами.

Решение задач

Книга предназначена помочь студентам соединить в одно целое методы решения задач и способы программирования, придавая одинаковую важность обоим процессам, которые, собственно, и составляют инструментарий специалиста по компьютерным наукам. Изучение методов, которыми специалисты пользуются при разработке, анализе и реализации решения, так же важно, как и устройство алгоритма. Здесь недостаточно простого перечисления рецептов.

В книге на конкретных примерах рассматриваются аналитические методы разработки программ. Абстракция, последовательное уточнение алгоритмов и структур данных, а также рекурсия — вот средства, позволяющие решить задачи, приведенные в этой книге.

Указатели и связанные списки вводятся уже в первых главах. Они используются при разработке структур данных. Кроме того, книга содержит элементарное изложение способов оценки сложности алгоритмов. Это позволяет, сначала на неформальном уровне, а затем более точно, оценить преимущества и недостатки реализации абстрактных типов данных в виде массивов и связанных списков. Центральной темой книги является поиск компромиссов между разными возможными решениями задач и реализациями абстрактных типов данных.

Стиль программирования, документация, включая пред- и постусловия, способы отладки и инварианты циклов представляют собой важную часть методологии решения задач, используемой для реализации типов и верификации программ. Эти вопросы также затрагиваются в книге.

Приложения

Основные темы, изложенные в книге, сопровождаются описанием классических приложений. Например, бинарный поиск, быстрая сортировка и алгоритм сортировки слиянием представляют собой классические примеры, на которых иллюстрируются применение рекурсии и способы оценки сложности алгоритмов. Такие темы, как сбалансированные деревья поиска, хэширование и индексация файлов позволяют углубить изложение методов поиска. Методы поиска и сортировки вновь рассматриваются в контексте работы с внешними запоминающими устройствами.

Алгоритмы распознавания и вычисления алгебраических выражений сначала вводятся в контексте рекурсии, а позднее рассматриваются в качестве приложений, в которых применяются стеки. В качестве других приложений укажем задачу о восьми ферзях, которая иллюстрирует понятие отката; очередь, позволяющую осуществлять событийно-ориентированное моделирование; а также поиск вершин и обход графа, представляющие собой важные применения стеков и очередей.

Новый и переработанный материал

В данной книге сохранен подход и философия второго издания. Абстракция данных и программирование рассматриваются как с общих точек зрения, так и в контексте языка C++. В ходе подготовки данного издания каждое предложение,

пример, заметка на полях и рисунок были тщательно проверены. Чтобы упростить изложение, в тексте и рисунках сделано много изменений и добавлений. Кроме того, некоторые фрагменты были просто удалены. Все программы были переработаны, чтобы учесть новейшие изменения языка C++.

Все главы и приложения были переработаны. Список основных изменений приводится ниже.

- Спецификации всех операций над абстрактными типами данных теперь используют систему обозначений языка UML. Это позволяет более ясно и точно указывать предназначение и тип данных, используемых в качестве параметров.
- В главе 1 расширено описание методов объектно-ориентированного проектирования и включено описание языка UML. Имена идентификаторов изменены, чтобы учесть соглашения, ставшие общепринятыми. Это облегчит работу студентов, изучавших язык Java, а также тех, кто будет изучать этот язык в дальнейшем.
- В главе 3 после введения классов кратко рассматривается наследование. Кроме того, описываются пространства имен и исключительные ситуации, предусмотренные в языке C++. Хотя абстрактные типы данных по-прежнему используют булевы переменные в качестве индикатора ошибки, в дальнейшем для этого применяются исключительные ситуации.
- В главу 4 включен новый раздел, посвященный стандартной библиотеке шаблонов языка C++ (STL). Вводится понятие шаблонного класса, контейнера и итератора. В главе 8 эта тема излагается более подробно. В главе 4 также рассматривается класс list из стандартной библиотеки STL. По ходу изложения в книге упоминаются и другие классы из библиотеки SDL. При желании их описание можно пропускать или откладывать.
- В главе 6 описан стандартный класс stack из библиотеки STL.
- В главе 7 описан стандартный класс queue из библиотеки STL.
- В главе 8 содержится более глубокое обсуждение наследования и шаблонных классов. Кроме того, в ней описаны дружественные классы и итераторы.
- Приложение A содержит обновленное изложение основ языка C++, в которое добавлено описание исключительных ситуаций. Приложение B содержит обновленный список заголовочных файлов, предусмотренных в языке C++. Приложение Д является совершенно новым. В нем приведены описания стандартных шаблонных классов *list*, *stack* и *queue* из библиотеки STL.

Обзор

Методические принципы и организация книги позволяют максимально облегчить процесс обучения, предоставляя преподавателям свободу выбора тем и способа их изложения в рамках конкретного курса.

Методические особенности

Цель книги — помочь студентам не только освоить материал, но и применить его в дальнейшей работе. Она характеризуется следующими особенностями.

- Каждая глава содержит введение, в котором кратко анонсируется ее содержание.

- Основные понятия заключены в рамку.
- Практически каждый абзац сопровождается пометкой на полях (в русском издании эти пометки выделены с помощью врезок. — *Прим. ред.*) .
- Каждая глава содержит резюме.
- В конце каждой главы приводятся предостережения о распространенных ошибках и заблуждениях.
- Каждая глава сопровождается вопросами для самопроверки с ответами.
- Каждая глава содержит упражнения и задания по программированию.
- Спецификации всех основных абстрактных типов приводятся как на естественном языке, так и с помощью псевдокода, а также на языке UML.
- В книге приведены определения классов на языке C++ для всех абстрактных типов.
- Классы и абстрактные типы иллюстрируются примерами.
- Книга содержит приложения, в которых изложены основы языка C++.
- В конце книги помещен словарь основных терминов.

Организация

Книга состоит из двух частей. Как правило, главы 1–11 образуют ядро курса, излагаемого на протяжении одного семестра. Главы 1 и 2 носят обзорный характер. Значение глав 11–14 зависит от характера курса.

Часть I. Методы решения задач. В главе 1 освещаются основные проблемы программирования и разработки программного обеспечения. Здесь приводится новое введение в язык UML. В следующей главе описывается рекурсия. Способность мыслить рекурсивно является весьма полезной для специалистов по компьютерным наукам. Часто она позволяет лучше понять природу задачи. В этой главе рекурсия рассматривается очень подробно. В дальнейшем она обсуждается в главе 5 и применяется практически во всех главах. Приведенные примеры варьируются от простых рекурсивных определений до рекурсивных алгоритмов, применяемых при распознавании выражений, поиске и сортировке.

В главе 3 излагаются принципы абстракции данных, а также детально описываются абстрактные типы данных (АТД). После обсуждения понятия спецификации и способов ее применения для описания абстрактных типов данных в этой главе рассматриваются классы языка C++, которые применяются для реализации АТД. В главе кратко описываются наследование, пространства имен и исключительные ситуации. В главе 4 обсуждаются указатели и связанные списки, а также их роль в реализации абстрактных типов данных. Кроме того, в этой главе описываются шаблонные классы, стандартная библиотека шаблонов языка C++ (STL), контейнеры и итераторы.

Порядок изложения тем, затронутых в части I, можно выбирать в зависимости от уровня подготовки студентов.

Часть II. Решение задач с помощью абстрактных типов данных. В этой части продолжается исследование абстракции данных как метода решения задач. Впервые описываются основные абстрактные типы данных, а именно: стек, очередь, бинарное дерево, бинарное дерево поиска, таблица, куча и очередь с приоритетами. Указанные типы реализуются в виде классов. Применение абстрактных типов данных иллюстрируется примерами. Проводится сравнение реализаций каждого АТД.

Глава 8 содержит более глубокое описание классов, наследования, шаблонных классов и итераторов. В этой главе вводятся дружественные классы и виртуальные функции. Глава 9 посвящена формализации понятия эффективности алгоритма путем использования обозначений О-большое. В этой главе проводится анализ эффективности нескольких алгоритмов поиска и сортировки, включая рекурсивную сортировку слиянием и быструю сортировку.

Часть II также содержит более сложные темы, например, описание сбалансированных деревьев поиска (2-3, 2-3-4, красно-черных и AVL-деревьев) и хэширования. Эти темы тесно связаны с анализом реализаций абстрактной таблицы.

В заключение рассматриваются методы хранения данных на внешних запоминающих устройствах. Описывается модифицированный метод сортировки слиянием, а также внешнее хэширование и индексы B-деревьев. Эти алгоритмы поиска представляют собой обобщение схем внутреннего хэширования и 2-3 деревьев, описанных ранее.

Вспомогательные материалы

Студенты и преподаватели могут получить вспомогательные материалы через Internet.

- *Исходные тексты программ.* Все классы, функции и программы, приведенные к книге, читатели могут получить на сайте `www.aw.com/cssupport`.
- *Ошибки.* Мы старались не делать ошибок, но полностью их избежать не удалось. Список обнаруженных ошибок, обновляемый по мере надобности, размещен на сайте `www.aw.com/cssupport`.

Пишите нам

Эта книга еще не закончена. Ваши комментарии, предложения и исправления будут с благодарностью приняты. Контактировать с нами можно либо непосредственно по адресам

```
carrano@acm.org
```

и

```
prichard@bryant.edu
```

либо через издательство
Computer Science Editorial Office
Addison-Wesley
75 Arlington Street
Boston, MA 02116

Благодарности

Предложения, полученные нами от рецензентов, оказали на книгу весьма благотворное влияние. Перечислим их в алфавитном порядке.

Вики Аллан (Vicki H. Allan) — *государственный университет Юты* (Utah State University)

Дон Бэйли (Don Bailey) — *университет Карлтона* (Carleton University)

Себастьян Элбаум (Sebastian Elbaum) — *университет Небраски*, г. Линкольн (University of Nebraska, Lincoln)

Мэтью Эветт (Matthew Evett) — *университет Западного Мичигана* (Eastern Michigan University)

Сьюзан Гейч (Susan Gauch) — *университет Канзаса* (University of Kansas)

Мартин Гранье (Martin Granier) — *университет Западного Вашингтона* (Western Washington University)

Джуди Хэнкинс (Judy Hankins) — *государственный университет Среднего Теннесси* (Middle Tennessee State University)

Джон Гарнетт-старший (Sr. Joan Harnett) — *колледж Манхэттена* (Manhattan College)

Том Ирби (Tom Irby) — *университет Северного Техаса* (University of North Texas)

Эдвин Дж. Кэй (Edwin J. Kay) — *университет Лехай* (Lehigh University)

Дэвид Нэффин (David Naffin) — *колледж Фуллертона* (Fullerton College)

Поль Нэйгин (Paul Nagin) — *университет Нофстра* (Hofstra University)

Бина Рамамурти (Bina Ramamurthy) — *университет SUNY в г. Буффало* (SUNY at Buffalo)

Дуайт Тьюнистра (Dwight Tunistra)

Карен ван Хойтен (Karen VanHouten) — *университет Айдахо* (University of Idaho)

Кэти Йерион (Kathie Yerion) — *университет Гонзага* (Gonzaga University)

Мы особенно благодарны людям, создавшим эту книгу. Наши редакторы в издательстве Addison-Wesley, Сьюзан Хартман (Susan Hartman) и Кэтрин Арутунян (Katherine Harutunian), оказали нам неоценимую помощь. Эта книга не была бы напечатана во время, если бы не наш менеджер проекта Дэниэл Райш (Daniel Rausch) из компании Argosy Publishing. Выражаем ему благодарность за поддержку.

Хотим выразить благодарность литературному редактору Ребекке Пеппер (Rebecca Pepper), сгладившей многие острые углы. Мы также благодарны Пэту Матани (Pat Mantani), Михаэлю Хитшу (Michael Hitsch), Джине Хэйген (Gina Hagen), Джэроду Гиббонсу (Jarrod Gibbons), Мишелю Ренда (Michelle Renda) и Джо Ветере (Joe Vetere), внесшим большой вклад в производство этой книги.

Мы хотели бы поблагодарить много других замечательных людей. Вспомним их поименно: Дуг Маккрейди (Doug McCreadie), Майкл Хэйден (Michael Hayden), Сара Хэйден (Sarah Hayden), Эндрю Хэйден (Andrew Hayden), Альберт Причард (Albert Prichard), Тэд Эммотт (Ted Emmott), Мэйбет Конвэй (Maybeth Conway), Лорэйн Берьюб (Lorraine Berube), Мардж Вайт (Marge White), Джеймс Ковальски (James Kowalski), Жерар Боде (Gerard Baudet), Джоан Пэкхэм (Joan Peckham), Эд Ламанья (Ed Lamagna), Виктор Фэй-Вольф (Victor Fay-Wolfe), Бала Равикумар (Bala Ravikumar), Лиза ди Пилиппо (Lisa DiPippo), Жан-Ив Эрве (Jean-Yves Hervé), Хэл Рекордс (Hal Records), Уолли Вуд (Wally Wood), Элен Лавалли (Elaine Lavallee), Кен Соуза (Ken Sousa), Салли Лоуренс (Sally Lawrence), Лайен Данн (Lianne Dunn), Гейл Армстронг (Gail Armstrong), Том Мэннинг (Tom Manning), Джим Лабонт (Jim Labonte), Джим Эбрю (Jim Abreu) и Билл Хардинг (Bill Harding).

Хотим также упомянуть многочисленных людей, внесших свой вклад в создание предыдущих изданий нашей книги. Все их замечания были весьма полезными и приняты нами с благодарностью. Вот их имена в алфавитном порядке. Карл Абрахамсон (Karl Abrahamson), Стефен Алберг (Stephen Alberg), Рональд Алферез (Ronald Alferez), Вики Аллан (Vicki Allan), Джихад Альмахайни (Jihad

Almahayni), Джеймс Эймес (James Ames), Клод В. Андерсон (Claude W. Anderson), Эндрю Аззинаро (Andrew Azzinaro), Тони Бэйчинг (Tony Baiching), Дон Бэйли (Don Bailey), Н. Дуйат Барнетт (N. Dwight Barnette), Джек Байдлер (Jack Beidler), Вольфганг В. Байн (Wolfgang W. Bein), Сто Белл (Sto Bell), Дэвид Берард (David Berard), Джон Блэк (John Black), Ричард Боттинг (Richard Botting), Вольфин Брамли (Wolfin Brumley), Филип Кэрриган (Philip Carrigan), Сефен Клэмидж (Stephen Clamage), Майкл Клэнси (Michael Clancy), Дэвид Клейтон (David Clayton), Майкл Клерон (Michael Cleron), Крис Константино (Chris Constantino), Шон Купер (Shaun Cooper), Чарльз Дено (Charles Denault), Винсент Дж. ди Пилиппо (Vincent J. DiPippo), Сьюзан Дорней (Suzanne Dorney), Коллин Данн (Colleen Dunn), Карл Экберг (Carl Eckberg), Карла Штайнбрюгге Фант (Karla Steinbrugge Fant), Джин Фольтц (Jean Foltz), Сьюзан Гейч (Susan Gauch), Маргарэт Хейфен (Marguerite Hafen), Рэндли Рейл (Randy Hale), Джордж Хэймер (George Hamer), Джуди Хэнкинс (Judy Hankins), Лайза Хеллерштайн (Lisa Hellerstein), Мэри Лу Хайнс (Mary Lou Hines), Джек Ходжес (Jack Hodges), Стефани Хорощак (Stephanie Horoschak), Лили Хоу (Lily Hou), Джон Хаббард (John Hubbard), Крис Йенсен (Kris Jensen), Томас Джадсон (Thomas Judson), Лаура Кенни (Laura Kenney), Роджер Кинг (Roger King), Ладислав Когут (Ladislav Kohout), Джим Лабонт (Jim LaBonte), Джин Лэйк (Jean Lake), Януш Ласки (Janusz Laski), Кэти Лебланк (Cathie LeBlanc), Урбан Лежен (Urban LeJeune), Джон М. Лайнбергер (John M. Linebarger), Кен Лорд (Ken Lord), Поль Лукер (Paul Luker), Маниша Манде (Manisha Mande), Пьер-Арно де Манеф (Pierre-Arnoul de Marneffe), Джон Марсалья (John Marsaglia), Джейн Уоллэс Майо (Jane Wallace Mayo), Марк Маккормик (Mark McCormick), Дэн Маккракен (Dan McCracken), Вивьен Макдугал (Vivian McDougal), Ширли Макгуайр (Shirley McGuire), Сью Медейрос (Sue Medeiros), Джим Миллер (Jim Miller), Гай Миллс (Guy Mills), Рамин Мохаммади (Rameen Mohammadi), Клев Моулер (Cleve Moler), Нараян Мурти (Narayan Murthy), Поль Нэйгин (Paul Nagin), Рейно Ниеми (Rayno Niemi), Джон О'Доннелл (John O'Donnell), Эндрю Олдройд (Andrew Oldroyd), Лэри Олсен (Larry Olsen), Реймонд Л. Пэйден (Raymond L. Paden), Рой Паргас (Roy Pargas), Бренда К. Паркер (Brenda C. Parker), Тадейш Ф. Павлицки (Thaddeus F. Pawlicki), Кэйт Зирс (Keith Pierce), Лукаш Пруски (Lucasz Pruski), Джордж Б. Пэрди (George B. Purdy), Дэвид Рэдфорд (David Radford), Стив Рэйтринг (Steve Ratering), Стюарт Реджис (Stuart Regis), Дж. Д. Робертсон (J. D. Robertson), Роберт А. Росси (Robert A. Rossi), Джон Роув (John Rowe), Майкл Е. Рапп (Michael E. Rupp), Шэрон Салветер (Sharon Salveter), Чарльз Саксон (Charles Saxon), Чандра Секхаран (Chandra Sekharan), Линда Шапиро (Linda Shapiro), Юджин Шенг (Yujian Sheng), Мэри Шилдс (Mary Shields), Ронни Смит (Ronnie Smith), Карл Спикола (Carl Spicola), Ричард Снодграсс (Richard Snodgrass), Нейл Снайдер (Neil Snyder), Крис Спаннабел (Chris Spannabel), Поль Спиракис (Paul Spirakis), Клинтон Стэйли (Clinton Staley), Мэтт Штальман (Matt Stallman), Марк Стеглик (Mark Stehlick), Бенджамин Т. Шомп (Benjamin T. Schomp), Хэрриет Тэйлор (Harriet Taylor), Дэвид Тиге (David Teague), Дэвид Тетро (David Tetreault), Джон Тэрнер (John Turner), Сьюзан Уоллес (Susan Wallace), Джеймс Е. Уоррен (James E. Warren), Джерри Вельтман (Jerry Weltman), Нэнси Виганд (Nancy Wiegand), Говард Вильямс (Howard Williams), Брэд Уилсон (Brad Wilson), Джеймс Вирт (James Wirth), Салих Юрттас (Salih Yurttas) и Алан Заринг (Alan Zaring).

Спасибо всем!
F.M.C
J.J.P.

I

Методы решения задач

Принципы программирования и разработки программного обеспечения

В этой главе...

Решение задач и разработка программного обеспечения
- Решение задачи
- Жизненный цикл программного обеспечения
- Хорошее решение задачи

Модульный подход
- Абстракция и сокрытие информации
- Объектно-ориентированное проектирование
- Проектирование "сверху вниз"
- Общие принципы проектирования
- Моделирование объектно-ориентированных проектов с помощью языка UML
- Преимущества объектно-ориентированного подхода

Краткий обзор основных понятий программирования
- Модульность
- Модифицируемость
- Легкость использования
- Надежное программирование
- Стиль
- Отладка

Резюме
Предупреждения
Вопросы для самопроверки
Упражнения
Задачи по программированию

Введение. В этой главе излагаются фундаментальные принципы, лежащие в основе решения больших и сложных задач. В ней излагаются основные принципы программирования, а также показано, что тщательно продуманные и хорошо описанные программы являются экономически эффективными. Глава содержит краткое описание алгоритмов и абстракции данных. Демонстрируется связь этих понятий с главной темой книги, а именно способами решения задач и методами программирования. В последующих главах акцент будет сделан на способах организации и обработки данных. Тем не менее нужно ясно понимать, что при решении любых задач необходимо твердо придерживаться основных принципов, изложенных в данной главе.

Решение задач и разработка программного обеспечения

Кодирование без предварительного проектирования увеличивает время отладки

С чего вы начинали, создавая свою последнюю программу? Многие начинающие программисты, прочитав постановку задачи, сразу же начинают писать код. Очевидно, они стремятся к тому, чтобы их программы работали, причем, по возможности, правильно. С этой целью они запускают свои программы, исследуют сообщения об ошибках, вставляют пропущенные точки с запятыми, изменяют логику, удаляют точки с запятыми, молятся и подвергают свои программы другим издевательствам, пока те не заработают правильно. Большую часть времени такие программисты затрачивают на вылавливание синтаксических ошибок и проверку логики работы программы. Очевидно, сейчас, когда вы уже написали свою первую программу, ваши программистские навыки намного улучшились, однако готовы ли вы создать на самом деле большую программу, используя те способы, которые мы описали только что? Может быть и готовы, однако лучше поступать иначе.

Технологии программирования облегчают разработку программ

Поймите, над разработкой очень больших программных проектов трудятся команды программистов, а не одиночки. Для командной работы нужен подробный план, четкая организация и полное взаимопонимание. Бессистемный подход к программированию здесь совершенно неприемлем и экономически неэффективен. К счастью, применение **технологий программирования** (software engineering) позволяет облегчить разработку компьютерных программ.

В книгах, предназначенных для начинающих программистов, основное внимание обычно уделяется приемам программирования. В нашей книге рассматривается более широкий круг вопросов, связанных с решением задач. Сначала мы рассмотрим весь процесс решения задачи и различные способы достижения результата.

Решение задачи

Термин **решение задачи** (solving problem) охватывает все этапы, начиная с постановки задачи и заканчивая разработкой компьютерной программы для ее решения. Этот процесс состоит из многих этапов — раскрытие смысла задачи, разработка концептуального решения, реализация решения в виде компьютерной программы.

Решение состоит из алгоритмов и способов хранения данных

Что именно называется решением? Обычно **решение** (solution) состоит из двух компонентов: алгоритма и способов хранения данных. **Алгоритм** (algorithm) — это пошаговое описание метода решения задачи за конечный отрезок времени. Алгоритмы часто работают со структурами данных. Например, алгоритм может вносить новые данные в структуру, удалять их оттуда либо просматривать.

Возможно, такое описание решения создает ложное впечатление, что вся сложность заключается в разработке подходящего алгоритма, а способы хранения данных играют вспомогательную роль. Это далеко от истины. Для решения задачи нужно не просто хранить данные, но и организовывать их таким образом, чтобы ускорить выполнение алгоритма. Фактически большая часть книги посвящена именно способам организации данных в различных структурах.

Для решения задач можно применять методы, описанные в этой главе. Более детально они изложены в последующих главах.

Жизненный цикл программного обеспечения

Разработка хорошего программного обеспечения должна учитывать долгий и продолжительный процесс, называемый **жизненным циклом программного обеспечения** (software's life cycle). Этот процесс начинается с первоначальной идеи, включает в себя написание и отладку программ и продолжается многие годы, в течение которых в исходное программное обеспечение вносятся изменения и улучшения. На рис. 1.1 показаны девять этапов жизненного цикла программного обеспечения в виде сегментов водяного колеса.[1] Это означает, что этапы представляют собой части некоторого умозрительного круга, а не простого линейного списка. Хотя все начинается с постановки задачи, обычно переход от одного этапа к другому не бывает последовательным. Например, тестирование программы может предполагать внесение изменений как в постановку задачи, так и сам проект. Кроме того, обратите внимание, что все девять сегментов расположены вокруг документирования, расположенного в центре круга. Документирование программы не является отдельным этапом ее жизненного цикла, как можно было бы подумать, а сопровождает ее на протяжении всей жизни.

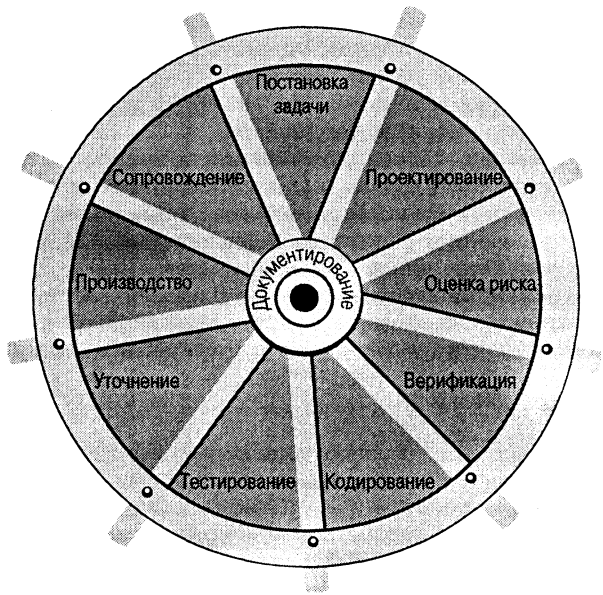


Рис. 1.1. Жизненный цикл программного обеспечения в виде вращающегося водяного колеса

[1] Благодарю Реймонда Падена (Raymond L. Paden) за подсказанную аллегорию.

На рисунке изображены этапы жизненного цикла типичного программного обеспечения. Несмотря на то что все они важны, в книге обсуждаются только некоторые из них.

Этап 1. Постановка задачи. Получив задание, мы должны ясно представлять все его аспекты. Часто люди, формулирующие задачи, не являются программистами, поэтому исходная постановка задачи может быть неточной. Следовательно, на первом этапе в ходе тесного общения программисты и непрограммисты должны совместными усилиями уточнить и детализировать исходную задачу.

Постановка задачи должна быть точной и подробной

Вот вопросы, на которые следует ответить. Каковы входные данные? Какие данные считаются корректными, а какие — нет? Для кого предназначено программное обеспечение? Какой пользовательский интерфейс следует применить? Какие сообщения об ошибках следует предусмотреть? Какие ограничения накладываются на программу? Существуют ли особые ситуации? В каком виде следует представлять выходные данные? Какая документация должна сопровождать программу? Какие усовершенствования программного обеспечения предусмотрены в будущем?

Макетные программы позволяют прояснить постановку задачи

Для полного взаимопонимания между заказчиками и исполнителями можно написать **макетные программы** (prototype programs), имитирующие поведение отдельных частей создаваемого программного обеспечения. Например, простая — пусть даже не эффективная — программа может демонстрировать предполагаемый пользовательский интерфейс. Лучше выявить все подводные камни либо изменить подход к решению задачи на этом этапе, а не в процессе программирования или при эксплуатации программного обеспечения.

Возможно, прежде ваш работодатель сам формулировал спецификации программы за вас. Скорее всего, не все аспекты этого описания были вам понятны, и вы нуждались в разъяснениях, но, вероятнее всего, у вас нет практики создания собственных спецификаций программы.

Слабо связанные модули являются независимыми

Этап 2. Разработка. Завершив этап постановки задачи, мы переходим к ее решению. Многие люди, разрабатывающие программы среднего размера и сложности, считают, что с целой программой справиться трудно. Лучше всего упростить процесс решения задачи, разбив большую задачу на несколько маленьких, которыми было бы легче управлять. В результате программа будет состоять из нескольких **модулей** (modules), представляющих собой самостоятельные единицы кода. Модуль может содержать одну или несколько функций, а также другие блоки кода. Следует стремиться к тому, чтобы модули были как можно более независимыми, или **слабо связанными** (loosely coupled) друг с другом. Разумеется, это не относится к их **интерфейсам** (interfaces), представляющим собой механизм их взаимодействия. Умозрительно модули можно считать изолированными друг от друга.

Узкоспециализированные модули предназначены для решения общей точно определенной задачи

Каждый модуль должен выполнять свою, точно определенную задачу. Следовательно, он должен быть **узкоспециализированным** (highly cohesive). Таким образом, **модульность (modularity)** — это свойство программ, состоящих из слабо связанных и узко специализированных модулей.

Указывайте предназначение каждого модуля, условия его применения, а также входные и выходные данные

На этапе проектирования важно точно указывать не только предназначение каждого модуля, но и **поток данных** (data flow) между модулями. Например, разрабатывая модуль, нужно ответить на следующие вопросы. Какие

данные доступны данному модулю во время его выполнения? В каких условиях можно выполнять данный модуль? Какие действия выполняет модуль и как изменяются данные после завершения его работы? Таким образом, нужно детально сформулировать предположения, а также входные и выходные данные для каждого модуля.

Например, если при разработке программы потребовалось упорядочить массив целых чисел, можно написать следующую спецификацию функции сортировки.

- Функция получает на вход *num* целых чисел, где *num* > 0.
- Функция возвращает упорядоченный массив, состоящий из целых чисел.

Спецификации — это контракт

Эту спецификацию можно рассматривать как **контракт** (contract) между вашей функцией и вызывающим ее модулем.

Если вы разрабатываете программу самостоятельно, этот контракт поможет систематически разбить исходную задачу на более мелкие части. Если над проектом работает команда программистов, контракт поможет разделить ответственность между ними. Программист, разрабатывающий функцию сортировки, должен выполнять контракт. Контракт законченной функции сортировки сообщает остальным программистам, как ее вызывать и какие результаты она должна возвращать.

Спецификация модуля не должна описывать метод решения задачи

Однако следует особо подчеркнуть, что контракт модуля не связывает его с конкретным методом решения задачи. Делать в другой части программы какие-либо предположения, касающиеся этого метода, не следует. Тогда, например, если в дальнейшем вы перепишете свою функцию и примените другой алгоритм сортировки, вносить изменения в остальной код не потребуется вообще. Если новая функция выполняет условия старого контракта, о других модулях можно не заботиться.

Спецификации функции состоят из точных пред- и постусловий

Все вышеизложенное не должно быть для вас новостью. Хотя до сих пор вы могли не использовать в своей речи слово "контракт", его концепция должна быть вам ясна. Формулируя **предусловие** (precondition) и **постусловие** (postcondition) функции, вы пишете ее контракт, состоящий из условий, которые должны выполняться перед ее вызовом и после завершения ее работы, соответственно. Например, псевдокод функции сортировки, придерживающейся приведенного выше контракта, выглядит следующим образом.[2]

Черновой набросок спецификаций

```
sort(anArray, num)
// Сортировка массива.
// Предусловие: переменная anArray является массивом,
// состоящим из num целых чисел; num > 0.
// Постусловие: целые числа в массиве anArray упорядочены.
```

На самом деле в данном случае этих пред- и постусловий недостаточно. Например, в каком порядке упорядочен массив: возрастающем или убывающем? Насколько большим может быть число *num*? Реализуя эту функцию, вы могли предполагать, что массив упорядочивается в возрастающем порядке, а число *num* не должно превышать 100. Представьте себе трудности, с которыми столкнется человек, который попытается применить функцию *sort* для сортировки 500 чисел в убывающем порядке. Этот пользователь ничего не знает о ваших предположениях, пока вы ясно не укажете их в пред- и постусловиях.

[2] Псевдокоды в книге набраны курсивом.

```
sort(anArray, num)
// Сортировка массива в возрастающем
// порядке.
// Предусловие: переменная anArray является массивом,
// состоящим из num целых чисел; 1 <= num <= MAX_ARRAY,
// где MAX_ARRAY — это глобальная константа, задающая
// максимальный размер массива anArray.
// Постусловие: anArray[0] <= anArray[1] <= ... <=
// anArray[num-1]; число num не изменяется.
```

Пересмотренная спецификация

В предусловии описываются входные аргументы функции, указываются все глобальные именованные константы, использующиеся в ней, и перечисляются все ограничения, которые накладываются функцией. Аналогично, в постусловии описываются результаты работы функции — либо возвращаемое функцией значение — и все последствия ее работы.

Документация должна быть точной

Новички стремятся приуменьшить значение точной документации, особенно когда они одновременно являются и разработчиками, и программистами, и пользователями программы. Если вы разработали функцию `sort`, но не указали условия ее контракта, вспомните ли вы о них при ее реализации? А через неделю? Что лучше освежает память —код на языке C++ или пред- и постусловия, сформулированные простым языком? При увеличении размера программы важность хорошей документации возрастает, независимо от того, в одиночку вы пишете программу или в команде.

Использование компонентов существующего программного обеспечения в собственном проекте

Не следует пренебрегать возможностью применения готовых модулей, решающих вашу задачу. Возможности повторного использования кода, предоставляемые языком C++, обычно реализуются в виде компилируемых библиотек. Это означает, что вы не всегда будете иметь доступ к исходному коду функции. Библиотеки представляют собой яркий пример коллекции готовых компонентов программного обеспечения. Например, вы знаете, как использовать стандартную функцию `sqrt`, содержащуюся в математической библиотеке языка C++ (*math.h*), однако не можете увидеть ее исходный текст. Если же функции `sqrt` передать число с плавающей точкой или соответствующее выражение, она извлечет из него квадратный корень и вернет его в вызывающий модуль. Функцию `sort` можно применять, ничего не зная о деталях ее реализации. Более того, она вообще может быть написана на другом языке! Функцию `sqrt` можно применять вслепую, *поскольку нам известна ее спецификация.*

Итак, если в прошлом вы не задерживались на этапе разработки программы, вам следует немедленно отказаться от этой привычки! Результатом этого этапа должно быть модульное решение, которое легко выразить с помощью конструкций конкретного языка программирования. Уделив должное внимание этому вопросу, вы сэкономите время, необходимое для написания и отладки вашей программы.

Позднее мы еще вернемся к обсуждению модульной структуры программ.

Некоторые, но не все, проблемы можно предсказывать и предотвращать

Этап 3. Оценка риска. Создание программного обеспечения сопряжено с риском. Некоторые проблемы присущи всем проектам, а некоторые характерны лишь для определенных разработок. Кое-какие из них можно предвидеть, в то время как другие остаются в тени. Они могут влиять на график и стоимость выполнения работ, экономические успехи и даже на жизнь и здоровье людей. Некоторые опасности можно предотвратить или смягчить, а некоторые — нет. Для идентификации, оценки и

предотвращения опасностей, возникающих при разработке программного обеспечения, существуют специальные методы. Вы познакомитесь с ними при освоении более сложного курса программирования. Результат оценки риска влияет на все этапы жизненного цикла программного обеспечения.

Этап 4. Верификация. Для проверки правильности алгоритмов существуют формальные методы. Хотя полностью эта задача еще не решена, стоит напомнить о некоторых аспектах процесса верификации программ.

Диагностическое утверждение (assertion) — это формальное высказывание, описывающее конкретные условия, которые должны выполняться в определенной точке программы. Пред- и постусловия представляют собой пример простых утверждений об условиях, которые должны выполняться в начале и в конце функции. **Инвариант** (invariant) — это условие, которое всегда должно быть истинным в конкретной точке алгоритма. **Инвариант цикла** (loop invariant) — это условие, которое должно выполняться до и после каждого выполнения цикла, являющегося частью алгоритма. Как мы убедимся в дальнейшем, инварианты цикла оказываются полезными для создания правильных циклов. Используя инварианты, легче обнаруживать ошибки, следовательно, сокращается время отладки и тестирования программы. Короче говоря, инварианты позволяют сэкономить время.

Правильность некоторых алгоритмов можно доказать

Доказательство правильности алгоритма напоминает доказательство теоремы в геометрии. Например, чтобы доказать, что функция работает правильно, нужно начать с проверки ее предусловия, аналогичного аксиомам и предположениям в геометрии, и продемонстрировать, что шаги алгоритма в итоге приводят к выполнению постусловия. Для этого нужно проверить каждый шаг алгоритма и показать, что из диагностического утверждения, относящегося к моменту времени, предшествующему выполнению конкретного шага, следует диагностическое утверждение, относящееся к моменту времени после выполнения этого шага.

Доказав корректность отдельных операторов, можно доказать правильность последовательности операторов, затем функций, и в итоге — всей программы. Допустим, мы доказали, что если диагностическое утверждение A_1 истинно и выполняется оператор S_1, то утверждение A_2 также истинно. Кроме того, предположим, что утверждение A_2 и оператор S_2 приводят к выполнению утверждения A_3. Отсюда следует, что если утверждение A_1 истинно, то выполнение операторов S_1 и S_2 приводит к истинности утверждения A_3. Продолжая в том же духе, в конце концов можно доказать правильность программы в целом.

Очевидно, что если в процессе верификации программы обнаружилась ошибка, алгоритм можно исправить, а постановку задачи немного изменить. Таким образом, используя инварианты, можно доказать, что ошибка содержалась не в коде, а *в самом алгоритме*. В результате время, затраченное на отладку программы, существенно сократится.

С помощью формальных методов можно доказать правильность разных конструкций, в частности операторов *if*, циклов и операторов присваивания. Для проверки правильности итерационных алгоритмов широко используются инварианты циклов. Например, мы докажем, что приведенный ниже цикл вычисляет сумму первых *n* элементов массива *item*.

```
// Вычисляет сумму элементов item[0], item[1], ...,
// item[n-1] для любого n>=1.
int sum = 0;
int j = 0;
```

```
while (j < n)
{
  sum += item[j];
  ++;
} // конец оператора while
```

Перед началом этого цикла значения переменной *sum* и *j* равны 0. После первого выполнения цикла значение переменных *sum* равно *item[0]*, а значение переменной *j* равно 1. Итак, можно сформулировать инвариант данного цикла.

Инвариант цикла

Значение переменной *sum* равно сумме элементов от *item[0]* до *item[j-1]*.

Инвариант правильного цикла должен выполняться в следующих точках.

- После каждого шага инициализации переменных, но до начала выполнения цикла.
- Перед каждым повторением цикла.
- После каждого повторения цикла.
- После завершения цикла.

В предыдущем примере перечисленные точки находятся в следующих местах программы.

```
// Вычисляет сумму элементов item[0], item[1], ...,
// item[n-1] для любого n>=1.
                  <- здесь должен выполняться инвариант
int sum = 0;
int j = 0;
while (j < n)
{                 <- здесь должен выполняться инвариант
sum += item[j];
  ++;
                  <- здесь должен выполняться инвариант
} // конец оператора while
                  <- здесь должен выполняться инвариант
```

Эти рассуждения можно применять при доказательстве правильности итерационного алгоритма. В нашем примере нужно доказать, что инвариант выполняется в каждой из следующих четырех точек.

Шаги, которые следует выполнить для доказательства правильности алгоритма

1. **Инвариант должен быть истинным изначально**, до начала первой итерации. В предыдущем примере инвариант утверждает, что значение переменной *sum* равно сумме элементов от *item[0]* до *item[-1]*. Это утверждение истинно, поскольку в этом диапазоне индексов элементов нет.
2. **Выполнение цикла должно сохранять инвариант.** Это означает, что если перед каждой итерацией цикла инвариант является истинным, нужно показать, что он остается истинным и после ее выполнения. В нашем примере цикл добавляет элемент *item[j]* к переменной *sum*, а затем увеличивает значение переменной *j* на единицу. Таким образом, после выполнения цикла к переменной *sum* добавляется последний элемент, т.е. *item[j-1]*. Таким образом, после выполнения цикла инвариант остается истинным.
3. **Из выполнения инварианта должна следовать правильность алгоритма.** Нужно показать, что если после завершения цикла инвариант остается истинным, то алгоритм является корректным. В предыдущем примере по за-

вершении цикла переменная *j* содержит значение *n*, следовательно, инвариант цикла остается истинным: переменная *sum* содержит сумму элементов от *item[0]* до *item[n-1]*, что и требовалось доказать.

4. **Цикл должен завершиться.** Нужно доказать, что цикл завершится после выполнения конечного количества итераций. В нашем примере переменная *j* сначала равна 0, а затем при каждой итерации увеличивается на 1. Таким образом, в конце концов переменная *j* станет равной числу *n* при любом *n*>=1. Этот факт и оператор *while* гарантирует, что цикл в конце концов завершится.

Инварианты можно применять не только для доказательства правильности цикла, но и для доказательства его неправильности. Например, допустим, что в предыдущем примере в операторе *while* вместо условия *j<=n* поставлено условие *j<n*. Шаги 1 и 2 в доказательстве правильности программы остаются без изменения, а вот шаг 3 изменится: по завершении цикла переменная *j* будет содержать число *n+1*, и, поскольку инвариант цикла должен быть истинным, переменная *sum* станет содержать сумму элементов от *item[0]* до *item[n]*. Поскольку при этом мы получаем неверное решение задачи, цикл следует признать неправильным.

Обратите внимание на очевидную связь между описанным выше процессом доказательства и **математической индукцией** (mathematical induction).[3] Доказательство истинности инварианта в начальный момент называется **базисом индукции** (base case). Оно аналогично доказательству, что некоторое свойство выполняется для натурального числа 0. Доказательство истинности инварианта на каждой итерации цикла называется **шагом индукции** (induction step). Он аналогичен доказательству утверждения, что если некоторое свойство выполняется для произвольного натурального числа *k*, то оно выполняется и для числа *k+1*. После выполнения четырех шагов, перечисленных выше, мы приходим к выводу, что инвариант является истинным после каждой итерации цикла, точно так же, как, следуя принципу математической индукции, можно доказать, что некоторое свойство выполняется для любого натурального числа.

Идентификация вариантов цикла позволяет конструировать правильные циклы. Инвариант нужно формулировать в виде комментария либо перед циклом, либо в его начале. Например, в предыдущем фрагменте программы следует поместить такой комментарий.

Формулируйте инварианты цикла в своих программах

```
// Инвариант: 0 <= j <= n и
// sum = item[0] + ... + item[j-1]
while (j < n)
...
```

В приведенном ниже примере нужно подтвердить, что инварианты двух не связанных друг с другом циклов являются корректными. Напомним, что каждый инвариант должен быть истинным как до, так и после каждой итерации цикла, включая последнюю итерацию. Кроме того, инвариант цикла *for* легче понять, если этот цикл временно преобразовать в эквивалентный цикл *while*.

Пример инвариантов цикла

```
// Вычисляет n! для целого числа n>=0
int f = 1;
// Инвариант: f == (j-1)!
for (int j = 1; j <= n; ++j)
   f *= j;
```

[3] Принцип математической индукции изложен в Приложении Г.

```
// Вычисляет приближенное значение функции e^x
// для действительного числа x
double t = 1.0;
double s = 1.0;
int k = 1;
// Инвариант: t == x^(k-1)/(k-1)! и
// s == 1+x+x^2/2!+...+x^(k-1)/(k-1)!
while (k <= n)
{ t *= x/k;
  s += t;
  ++k;
} // конец цикла while
```

Этап 5. Кодирование. Кодирование заключается в переводе алгоритма на конкретный язык программирования с последующим исправлением синтаксических ошибок. Вполне вероятно, именно кодирование многие считают собственно программированием. И все же следует понимать, что кодирование — это не самое главное, это лишь один из этапов жизненного цикла программного обеспечения

Кодирование - это относительно небольшая часть жизненного цикла программного обеспечения

Этап 6. Тестирование. На этапе тестирования нужно выявить и исправить как можно больше логических ошибок. Для этого можно прибегнуть к проверке отдельных функций, применяя их к выбранным данным и сравнивая с заранее известным результатом. Если входные данные изменяются в каком-то диапазоне, обязательно проверьте их крайние значения. Например, если входное значение *n* может изменяться от 1 до 10, обязательно протестируйте программу при значениях 1 и 10. Кроме того, проверьте, как работает программа, если в нее ввести заведомо неверные данные, и может ли она обнаруживать такие ошибки. Попробуйте ввести в программу случайно выбранные данные, а затем примените ее для реального набора данных. Тестирование — это и наука, и искусство одновременно.

Разработайте набор тестовых данных для проверки вашей программы

Этап 7. Уточнение решения. Результатом выполнения этапов 1–6 является работающая программа, которую интенсивно тестировали и отлаживали. Если программа действительно решает поставленную задачу, возникает вопрос: зачем уточнять решение?

Лучше всего решать задачу при наиболее простых предположениях, постепенно усложняя программу. Например, можно предположить, что входные данные имеют определенный формат и являются правильными. Создав простейший вариант, можно дополнять его более сложными процедурами ввода и вывода данных, оснащать дополнительными возможностями и средствами для обнаружения ошибок.

Разрабатывайте программу при упрощающих предположениях, постепенно усложняя ее

Таким образом, если вы применяете подход "от простого — к сложному", этап уточнения решения становится необходимым. Разумеется, окончательное уточнение решения не должно приводить к полному пересмотру программы. Каждое уточнение решения является довольно очевидным, особенно если программа имеет модульную структуру. Фактически постепенное уточнение решения представляет собой основное преимущество модульного подхода к разработке программ! Кроме того, после каждой, даже простейшей, модификации программы, ее нужно снова тщательно протестировать.

Измененную программу следует протестировать снова

Как видим, этапы жизненного цикла программного обеспечения не изолированы друг от друга и не следуют один за другим. Сделав реалистичные упрощающие предположения в самом начале процесса разработки программы, вы должны точно предвидеть, как учесть их в дальнейшем. Тестирование программы может вынудить внести в программу изменения, однако модифицированную программу придется снова тестировать.

Этап 8. Производство. После завершения разработки программного продукта он распространяется среди пользователей, инсталлируется на их компьютерах и применяется.

Этап 9. Сопровождение. Поддержка программы не имеет ничего общего с обслуживанием автомобиля. Программное обеспечение не изнашивается, если за ним не ухаживать. Однако пользователи ваших программ могут обнаружить ошибки, оставшиеся незамеченными при тестировании. Кроме того, со временем программное обеспечение нужно совершенствовать, добавляя в него новые функциональные возможности или модифицируя его компоненты. Авторы программ занимаются этим довольно редко, тем важнее становится наличие хорошей документации.

Сопровождение программного обеспечения заключается в исправлении ошибок, обнаруженных пользователем, и его усовершенствовании

Необходимо ли точно следовать описанным выше этапам в реальной работе? Конечно да! Этапы 1–7 — это компоненты процесса решения задачи. Используя эту стратегию, сначала нужно разработать и реализовать решение (этапы 1–6), основываясь на некоторых первоначальных упрощающих предположениях. В результате вы получите хорошо организованную программу, решающую несколько упрощенную задачу. На последнем этапе эта программа усложняется и должна полностью соответствовать исходной постановке задачи.

Хорошее решение задачи

Перед тем как приступить к изучению методов решения задач, следует вначале убедиться, что овладение этими приемами действительно приводит к хорошим результатам. Очевидно, что применение этих методов позволяет получить хорошее решение задачи. Тогда возникает более существенный вопрос: а что считается хорошим решением? Попробуем на него ответить.

Поскольку окончательное решение задачи выражается в виде компьютерной программы, рассмотрим, какими свойствами обладает хорошая компьютерная программа. По-видимому, программа создается для решения конкретной задачи. Решение этой задачи имеет реальную и вполне ощутимую **стоимость** (cost). В нее входят ресурсы компьютера (время вычислений и память), потребленные программой, неудобства, с которыми сталкиваются пользователи программы, и последствия, к которым приводит ее неправильная работа.

Однако это еще не все. Эти факторы относятся лишь к одному из этапов жизненного цикла программы — этапу ее поддержки. Стремясь ответить на вопрос, насколько хорошее решение получено вами, нужно рассмотреть все этапы разработки программы. Каждый из этих этапов также имеет свои затраты. Общая стоимость решения должна учитывать объем рабочего времени, затраченного программистами, которые его разрабатывали, уточняли, кодировали, отлаживали и тестировали. Кроме того, необходимо учесть стоимость поддержки, модификации и усовершенствования программы.

Таким образом, вычисляя общую стоимость решения, нужно принимать во внимание разнообразные факторы. Встав на такую многомерную точку зрения, можно сформулировать следующий критерий.

- Решение считается хорошим, если его общая стоимость минимальна.

Многомерная точка зрения на стоимость решения

Интересно проследить, как изменялась относительная важность разных компонентов в ходе эволюции программирования. Вначале доля стоимости работы компьютера по сравнению со стоимостью работы программистов была чрезвычайно высока. Кроме того, программы разрабатывались для решения очень специфичных, узко поставленных задач. Если постановка задачи изменялась, создавалась новая программа. Поддержка программ во внимание не принималась, их читабельность не имела никакого значения. Программу обычно использовал только один человек, ее автор. Как следствие, программистов не интересовало, удобно ли работать с программой. Интерфейс программы не считался важным фактором.

В такой среде программирования все перевешивала стоимость компьютерных ресурсов. Если две программы решали одну и ту же задачу, лучшей считалась та, которая работала быстрее и занимала меньший объем памяти. Как все изменилось с тех пор! Сейчас стоимость компьютерного времени резко снизилась, и время, затраченное разработчиками и программистами, стало более значительным фактором, влияющим на общую стоимость решения задачи. Другим следствием падения стоимости вычислений стало широкое использование компьютеров в разных сферах деятельности человека, многие из которых не связаны с наукой. Люди, работающие на компьютерах, часто не имеют специального опыта и знаний, необходимых для работы с программами. Следовательно, программы должны быть легкими в эксплуатации.

Программы должны быть хорошо организованными и сопровождаться подробной документацией

В настоящее время программы становятся все более сложными и большими. Часто они настолько велики, что в их разработку и эксплуатацию вовлекается много людей. Хорошая структура и документация в этих условиях приобретают чрезвычайно важное значение. Чем более важную задачу решают программы, тем серьезнее последствия их неправильной работы. Таким образом, людям нужны хорошо организованные программы и способы их формальной верификации. Люди не хотят рисковать, используя программы, с которыми могут работать лишь их авторы.

Как видим, развитие технологии привело к тому, что в настоящее время самое эффективное решение не всегда является наилучшим. Если две программы решают одну и ту же задачу, то лучшей из них не обязательно является та, которая быстрее работает. Программисты, стремящиеся использовать любую возможность, чтобы сэкономить несколько миллисекунд вычислений, отстали от жизни. В настоящее время, создавая программы, нужно ориентироваться не только на компьютеры, но и на людей, которые будут их использовать.

Эффективность — лишь один из многих аспектов, влияющих на стоимость решения

В то же время, не следует считать, что эффективность решения больше не имеет значения. Во многих ситуациях она очень важна. Просто нужно иметь в виду, что эффективность решения — это всего лишь один из многих факторов, которые следует учитывать. Если два решения обладают примерно одинаковой эффективностью, на сцене появляются другие аспекты, влияющие на выбор. Однако, если решения *значительно* отличаются по эффективности, этот факт может перекрыть остальные соображения. Выбирая или разрабатывая методы решения задачи, следует иметь это в виду. Выбор компонентов решения — алгоритмов и способов хранения данных — влияет на эффективность решения больше, чем непосредственное кодирование.

В книге последовательно отстаивается многомерная точка зрения на стоимость решения. В сегодняшних условиях эта точка зрения вполне разумна, и нам кажется, что в ближайшие годы это положение вещей не изменится.

Модульный подход

Мы уже убедились, насколько важно сопровождать каждый модуль точным описанием пред- и постусловий, но как разбить программу на эти модули? Решению именно этой проблемы и посвящена вся книга. В этом разделе мы рассмотрим два важных способа проектирования. Оба эти способа используют абстракцию, поэтому начнем с определения этого понятия.

Абстракция и сокрытие информации

Каждый модуль, из которого состоит решение задачи, начинается строками, в которых указано, для чего он предназначен, но не написано, как именно он работает. Ни один модуль не может "знать", как работает другой модуль, — в лучшем случае, он может знать, лишь для решения какой задачи предназначены другие модули.

Указывайте, что делает модуль, но не описывайте, как он это делает

Например, если в какой-то части программы данные должны упорядочиваться, то в одном из модулей выполняется алгоритм сортировки (рис. 1.2). Другие модули знают о том, что здесь выполняется сортировка данных, но не знают, как именно она осуществляется. Таким образом, разные части решения изолируются друг от друга.

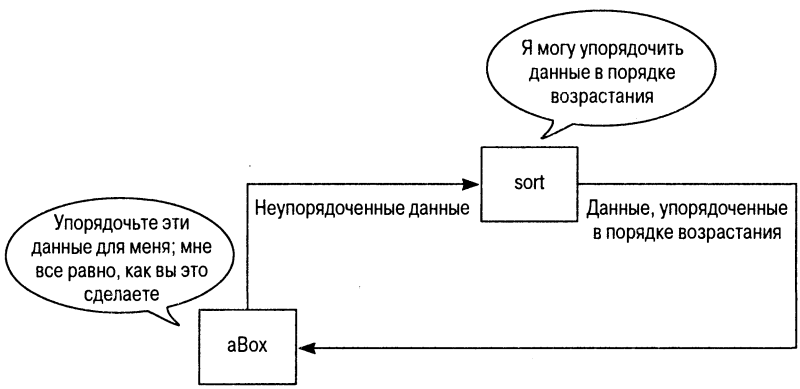


Рис. 1.2. Детали алгоритма сортировки скрыты от других частей программы

Спецификация каждого модуля создается до его реализации

Абстракция (abstraction) отделяет предназначение модуля от его реализации. Модульность и абстракция дополняют друг друга. Модульный подход позволяет разделить решение задачи на блоки; абстракция определяет содержание модуля *до* его реализации на конкретном языке программирования.

В спецификациях не указывается, как именно реализован модуль

Например, в спецификации модуля указывается, какие условия должны выполняться и что именно в нем происходит. Такие спецификации облегчают решение задачи, позволяя сосредоточиться только на функциональных возможностях высокого уровня, не вникая в детали их реализации. Кроме того, эти принципы позволяют модифицировать части решения независимо друг от друга. Например, можно ли изменить алгоритм сортировки, приведенный выше, не затрагивая остальную часть решения?

В ходе решения задачи содержание каждого модуля постепенно уточняется, воплощаясь в итоге в виде функций на языке C++. Предназначение функции следует отделять от ее реализации. Этот процесс называется **функциональной** (или **процедурной**) **абстракцией** (functional, or procedural abstraction). Готовую функцию можно применять, не вникая в детали реализации алгоритма, поскольку для использования достаточно знать ее предназначение и описание аргументов. Если функция сопровождается соответствующей документацией, ее можно использовать, зная лишь объявление и первичное описание, реализацию можно не изучать.

> Указывайте, что делает функция, но не описывайте, как она это делает

Функциональная абстракция играет важную роль в командных проектах. В таких ситуациях участники проектов должны применять функции, разработанные другими программистами, не вникая в детали их алгоритмов. Неужели можно применять функцию, не зная ее кода? Но ведь именно так вы и поступаете, вызывая функцию `sqrt` из математической библиотеки языка C++.

Рассмотрим теперь совокупность данных и набор операций над ними. В этом наборе могут быть операции добавления данных в совокупность, удаления их оттуда или операции поиска. **Абстракция данных** (data abstraction) сосредоточивает внимание на предназначении операций, а не на деталях их выполнения. Другие модули программы будут "знать", *что именно* делает та или иная операция, но не смогут узнать, *как* при этом хранятся данные или *как* именно выполняется данная операция.

> Указывайте, что именно вы хотите сделать с данными, но не описывайте, как это нужно сделать

В предыдущих примерах мы использовали массив. А что, собственно, он собой представляет? В книге приведено много иллюстраций, посвященных массивам. Они не могут точно соответствовать их машинной реализации, а могут лишь отдаленно напоминать о ней. Дело в том, что нам не важно, что именно представляет собой массив, т.е. как он реализован. Мы и без этого можем его использовать. Несмотря на то что разные операционные системы реализуют массивы по-разному, программисту это безразлично. Например, независимо от реализации массива `years`, число 1492 всегда можно записать в ячейку массива с номером `index`, используя следующий оператор.

```
years[index] = 1492;
```

Позднее, это значение можно вывести на экран, воспользовавшись оператором

```
cout << years[index] << endl;
```

Таким образом, мы вполне способны использовать массив, ничего не зная о способе его реализации, точно так же, как функцию `sqrt` мы можем вызывать, не зная, как она извлекает квадратный корень из своего аргумента.

Большая часть книги посвящена абстракции данных. Чтобы научить вас думать о данных абстрактно — т.е. фокусировать внимание на операциях с данными, а не на деталях их реализации, — нужно дать определение **абстрактного типа данных**, или **АТД** (abstract data type). АТД — это совокупность данных *и* множество операторов над ними. Операции АТД можно применять, если известны их спецификации, при этом не обязательно знать детали их реализации или способы хранения данных.

Для реализации АТД можно использовать **структуру данных** (data structure), представляющую собой конструкцию, определенную в языке программирования для хранения совокупности данных. Например, данные можно хранить в массивах целых чисел, объектов или массивах массивов.

> АТД — это не синоним структуры данных

В процессе решения задачи абстрактные типы данных помогают реализовывать алгоритм, а алгоритмы диктуют выбор абстрактного типа данных. Разработка алгоритма и АТД должны быть связаны друг с другом. Глобальный алгоритм, предназначенный для решения задачи, предполагает выполнение последовательности операций над данными, что, в свою очередь, приводит к определению АТД и алгоритмов, выполняющих эти операции. Однако процедуру решения задачи можно выполнять и в обратном порядке. Вид применяемого АТД может диктовать выбор стратегии глобального алгоритма решения задачи. Таким образом, зная, какие операции над данными выполнять легко, а какие — трудно, можно существенно повысить эффективность решения задачи.

Разработка алгоритма и АТД должны быть связаны друг с другом

Возможно, вы уже догадались, что обычно трудно четко отделить проблемы, связанные с алгоритмами, от проблем, связанных со структурами данных. Часто невозможно понять, благодаря чему достигается эффективность программы: остроумному алгоритму или удачному выбору структуры данных.

Сокрытие информации. Как видим, абстракция вынуждает создавать функциональные спецификации для каждого модуля, делая его **открытым** (public) для внешнего мира. Однако она позволяет идентифицировать детали, которые должны быть скрыты от публичного обозрения, — т.е. быть **закрытыми** (private). Принцип **сокрытия информации** (information hiding) гарантирует, что такие детали будут не только скрыты внутри модуля, но и ни один другой модуль не будет даже подозревать об их существовании.

Все модули и АТД иногда нужно скрывать

Принцип сокрытия информации ограничивает способы работы с функциями и данными. Пользователь модуля не должен интересоваться деталями его реализации. Разработчик модуля не должен заботиться о способах его использования.

Объектно-ориентированное проектирование

Один из способов модульного решения задачи — идентификация **объектов** (objects), объединяющих в единое целое данные и операции над ними. В результате такого **объектно-ориентированного подхода** (object-oriented approach) к модульному решению задачи возникает совокупность объектов, обладающих определенным поведением.

Объекты инкапсулируют данные и операции

Не зная этого, вы уже встречались с объектами. Будильник, разбудивший вас сегодня утром, **инкапсулирует** (encapsulates) время и операции, например "звонок". Инкапсулировать — значит "упаковывать" или "вкладывать". Таким образом, инкапсуляци — это способ сокрытия внутренних деталей. Функции инкапсулируют действия, объекты инкапсулируют данные вместе с действиями. Когда вы хотите, чтобы будильник зазвенел, вы не знаете, как он это сделает. Вы увидите лишь результат этой операции.

Инкапсуляция скрывает внутренние детали

Допустим, мы хотим написать программу, выводящую на экран циферблат часов. Для простоты предположим, что это электронные часы без будильника, как показано на рис. 1.3. Начать решение задачи можно с идентификации объектов.

Для идентификации объектов существуют несколько способов, но все они не идеальны. Один из простых способов[4] основан на распознавании имен существительных и глаголов, входящих в описание задачи. Имена существительные можно считать объектами, действия которых обозначаются глаголами. Тогда поставленную выше задачу можно описать следующим образом.

[4] Этот метод не слишком надежен. Спефикация задачи может состоять как из имен существительных, так и глаголов. Так, например, слово "звонок" иногда может означать как существительное, так и глагол. В этом случае идентифицировать объекты и операции будет непросто.

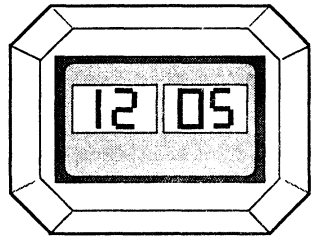


Рис. 1.3. Электронные часы

Спецификации программы для вывода на экран циферблата электронных часов

Программа имитирует работу электронных часов, показывающих часы и минуты. Цифровые индикаторы часов и минут позволяют отображать числа от 1 до 12 и от 0 до 60 соответственно. Время задается с помощью установок индикаторов часов и минут, причем программа должна постоянно обновлять их показания.

Даже не имея детального описания задачи, можно идентифицировать по крайней мере один объект — сами часы. Эти часы выполняют следующие операции.

- *Установка времени*
- *Изменение времени*
- *Вывод показаний на экран*

Индикаторы часов и минут также являются объектами, причем они очень похожи. Каждый из них выполняет следующие операции.

- *Установка значения*
- *Изменение значения*
- *Вывод значения на экран*

Объект — это экземпляр класса

Фактически оба индикатора представляют собой один и тот же тип объекта. Множество объектов, имеющих один и тот же тип, называется **классом** (class). Таким образом, нам нужно указать не конкретный объект, а класс объектов: класс часов и класс индикаторов. Объект, обозначающий часы, представляет собой **экземпляр** (instance) класса часов. Он состоит из двух объектов, представляющих собой экземпляры класса индикаторов.

Классы определяют данные и операции над объектами. Отдельные элементы данных, определенных в классе, называются **данными-членами** (data members), **полями данных** (data fields) или **атрибутами** (attributes). Операции, заданные в классе, называются **методами** (methods) или **функциями-членами** (member functions).

Инкапсуляция будет рассмотрена в главе 3. В частности, там будут определены классы языка C++. В последующих главах мы изучим различные абстрактные типы данных и их реализации в виде классов. Основное внимание будет уделено абстракции данных и инкапсуляции. Такой подход к программированию называется **объектным** (object based).

Объектно-ориентированное программирование (object-oriented programming), или **ООП**, дополняет инкапсуляцию двумя новыми принципами.

ОСНОВНЫЕ ПОНЯТИЯ

Три принципа объектно-ориентированного программирования

1. Инкапсуляция: объекты объединяют данные и операции.
2. Наследование: классы могут наследовать свойства других классов.
3. Полиморфизм: объекты могут выбирать подходящие операции во время выполнения программы.

Классы могут **наследовать** (inherit) свойства других классов. Например, определив класс часов, мы можем разработать класс будильников, наследующий свойства часов, добавив новые операции, свойственные будильникам. Это можно сделать быстро, поскольку класс часов уже разработан. Таким образом, **наследование** (inheritance) позволяет повторно использовать классы, определенные ранее (возможно, для других, но похожих целей), выполняя соответствующие модификации.

Наследование может поставить компилятор в затруднительное положение, поскольку он не сможет определить, какую операцию следует выполнить в конкретной ситуации. Однако полиморфизм (polymorphism) — буквально означающий *изменчивость форм* — позволяет выбрать нужную операцию уже на этапе выполнения программы. Таким образом, результат выполнения конкретной операции зависит от объектов, к которым она применяется.

Перегруженный оператор имеет несколько значений

Например, если в программе используется оператор +, операндами которого являются числа, то выполняется сложение чисел, но если к строкам применяется **перегруженный** (overloaded) оператор +, то выполняется их конкатенация. Хотя в данном случае компилятор может сам определить правильный смысл оператора +, полиморфизм допускает ситуации, когда смысл операции уточняется лишь на этапе выполнения программы.

Наследование и полиморфизм обсуждаются в главе 8.

Проектирование "сверху вниз"

Обычно объектно-ориентированный подход приводит к модульному решению задач, основываясь лишь на анализе данных. При разработке алгоритма для конкретной функции или в ситуациях, когда на первое место выходит алгоритм, а не данные, с которыми он работает, модульное решение можно получить с помощью **проектирования "сверху вниз"** (top-down design). В то время как с помощью объектно-ориентированного подхода можно идентифицировать данные, основываясь на именах существительных, использованных в описании задачи, проектирование "сверху вниз" основано на анализе глаголов.

Структурная схема иллюстрирует отношения между модулями

Стратегия проектирования "сверху вниз" основана на последовательном понижении уровня детализации задачи. Рассмотрим простой пример. Допустим, что нам нужно вычислить среднюю экзаменационную оценку. На рис. 1.4 показана **структурная схема** (structire chart), иллюстрирующая иерархию модулей и взаимодействие между ними. Во-первых, для каждого модуля указывается лишь описание его *предназначения*, лишенное каких-либо деталей. Каждый модуль разбивается на несколько более мелких модулей. В результате возникает иерархия модулей. Каждый модуль уточняется его наследником, решающим более мелкую задачу и содержащим больше информации о *способе* решения задачи, чем его предшественник. Процесс уточнения продолжается, пока модули не окажутся достаточно простыми для представления их в виде функций на языке C++ и изолированных фрагментов кода, решающих очень маленькие, независимые друг от друга задачи.

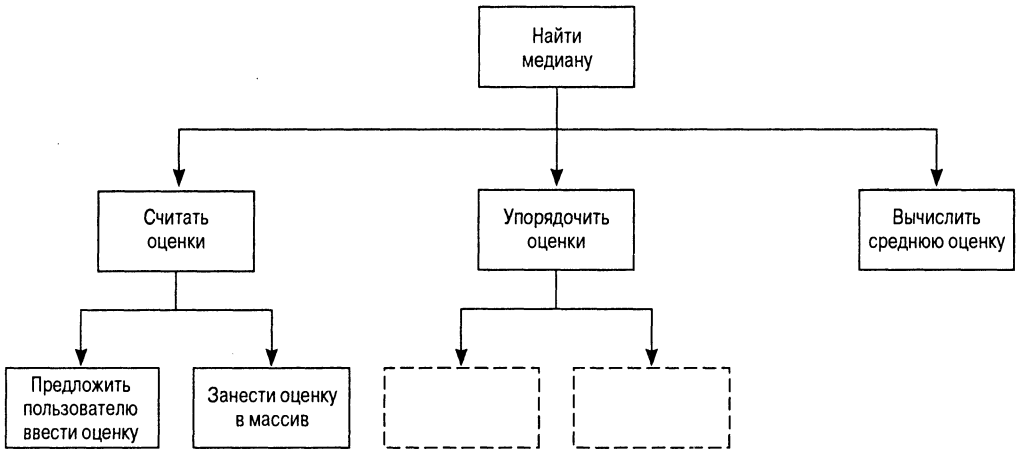


Рис. 1.4. Структурная схема, иллюстрирующая иерархию модулей

Обратите внимание, что на рис. 1.4 задача разбивается на три независимые подзадачи.

Решение состоит из независимых подзадач

- *Считать экзаменационные оценки*
- *Упорядочить оценки*
- *Определить "среднюю" оценку*

Если три эти задачи решаются тремя разными модулями, то, вызывая их, можно найти среднюю оценку, независимо от способов их реализации.

Разработка каждого модуля начинается с разбиения его на подзадачи. Например, задачу считывания оценок можно уточнить с помощью двух модулей.

Подзадачи

- *Предложить пользователю ввести оценку*
- *Записать оценку в массив*

Каждый из этих модулей можно уточнить аналогичным способом. В итоге мы получим псевдокод алгоритма, решающего поставленную задачу.

Общие принципы проектирования

Обычно при решении задачи используются объектно-ориентированное проектирование (ООП), проектирование "сверху вниз" (ПСВ), абстракция и сокрытие информации. Подход, ведущий к модульному решению задачи, описывается следующими принципами проектирования.

ОСНОВНЫЕ ПОНЯТИЯ

Принципы проектирования

1. Для получения модульного решения одновременно используйте объектно-ориентированное проектирование и подход "сверху вниз". Таким образом, абстрактные типы данных и алгоритмы нужно разрабатывать параллельно.
2. Для решения задач обработки данных используйте объектно-ориентированное проектирование.
3. Для разработки алгоритмов используйте подход "сверху вниз".
4. Если главными в решении задачи являются алгоритмы, а не данные, применяйте проектирование "сверху вниз".

5. При разработке абстрактных типов данных и алгоритмов акцентируйте внимание на вопросе *что*, а не *как*.
6. Старайтесь применять готовые компоненты программного обеспечения.

Моделирование объектно-ориентированных проектов с помощью языка UML

Универсальный язык моделирования (UML — Unified Modeling Language) используется для описания объектно-ориентированных проектов. Этот язык содержит спецификации диаграмм и текстовых описаний. Диаграммы особенно полезны для общего описания проектов, включая спецификации классов, и разных способов взаимодействия между ними. Обычно программа состоит из многих классов, поэтому возможность описывать взаимодействия между ними представляет собой ценное свойство языка UML.

В этом разделе мы рассмотрим лишь спецификации классов, поэтому он содержит только диаграммы классов и связанные с ними синтаксические конструкции. В диаграмме класса указывается его имя, данные-члены и операции. На рис. 1.5 показана диаграмма класса `Clock`, описанного выше. Верхний раздел диаграммы содержит имя класса. Средний раздел содержит данные-члены, а нижний — операции класса. Обратите внимание, что диаграмма носит довольно общий характер; она не диктует выбор фактической реализации класса. Это типичное представление концептуальной модели класса, не зависящее от выбора языка его реализации.

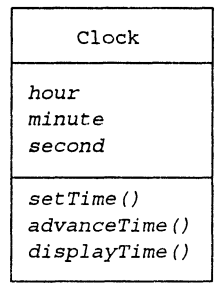


Рис. 1.5. Диаграмма класса Clock на языке UML

Наряду с диаграммами классов язык UML позволяет создавать текстовые описания для представления данных-членов и операций, выполняемых в классе. Эти записи можно включать в диаграммы классов, однако это усложняет диаграммы, снижая степень их общности. В данном разделе мы будем использовать именно текстовые описания классов, поскольку они позволяют создавать более полные спецификации, чем диаграммы.

Синтаксис описания данных-членов на языке UML имеет следующий вид.

модификатор_доступа имя: тип = значение_по_умолчанию

Здесь использованы следующие обозначения.

- Модификатор доступа принимает значение + (`public`) или - (`private`). Третье возможное значение — символ # (`protected`). Эту возможность мы обсудим в главе 8.
- Элемент *имя* означает имя атрибута.
- Элемент *тип* означает тип атрибута.
- Элемент *значение_по_умолчанию* задает начальное значение атрибута.

Как показывает диаграмма класса, нужно задать хотя бы имя класса. Элемент *значение_по_умолчанию* используется лишь в тех ситуациях, когда значение атрибута задается по умолчанию. В некоторых случаях нужно избегать явного указания типа атрибута, отложив решение этого вопроса до этапа реализации. В дальнейшем мы будем использовать следующие названия распространенных типов аргументов: `integer` — для целочисленных значений, `float` — для значений с плавающей точкой, `boolean` — для булевых значений и `string` — для строковых значений. Обратите внимание, что эти имена не совпадают с соответствующими названиями типов данных в языке C++, поскольку текстовое описание класса не должно зависеть от языка его реализации.

Вот как выглядит текстовое описание атрибутов класса `Clock`, показанного на рис. 1.5.

- *-hour: integer*
- *-minute: integer*
- *-second: integer*

Следуя принципу сокрытия информации, данные-члены `hours`, `minute` и `second` объявлены закрытыми.

Синтаксические конструкции языка UML, предназначенные для описания операций, выглядят немного сложнее.

модификатор_доступа имя(список_параметров):
тип_возвращаемого_значения (строка_свойств)

Здесь использованы следующие обозначения.

- Модификатор доступа принимает те же значения, что и в предыдущем случае.
- Элемент *имя* означает имя операции.
- Элемент *список_параметров* содержит параметры, разделенные запятой. Синтаксическая конструкция для описания параметров выглядит следующим образом.

  направление имя: тип = значение_по_умолчанию.
- Здесь элемент *направление* используется для индикации ввода (`in`), вывода (`out`) или ввода-вывода (`inout`) параметра.
- Элемент `name` является именем параметра.
- Элемент `type` задает тип параметра.
- Элемент *значение_по_умолчанию* задает значение, которое следует присвоить параметру, если соответствующий аргумент пропущен.
- Элемент *тип_возвращаемого_значения* задает тип значения, возвращаемого операцией; если операция не возвращает никакого значения, место этого элемента остается пустым.
- Элемент *строка_свойств* перечисляет свойства операции.

Как и для атрибутов, в диаграммах классов нужно указывать хотя бы *имя* операции. Иногда в диаграмму включается *список_параметров*, если это позволяет лучше понять функциональные возможности класса.

Строка_свойств может содержать множество разнообразных значений, однако нас будет интересовать лишь свойство `query`. Это свойство позволяет идентифицировать операции, которые не имеют права модифицировать данные, содержащиеся в классе.

Текстовое описание операций, предусмотренных в классе `Clock`, имеет следующий вид.

- *+setTime(in hr: integer, in min: integer, in sec: integer)*
- *–advanceTime( )*
- *+displayTime( ) (query)*

Здесь операции `setTime` и `displayTime` определены открытыми, а операция `advanceTime` — закрытой. Функция `displayTime` имеет свойство `query`, означающее, что она не изменяет никаких данных. Эта функция лишь выводит данные на экран.

Преимущества объектно-ориентированного подхода

При использовании объектно-ориентированного подхода (ООП) время, затрачиваемое на проектирование программы, увеличивается. Кроме того, решение, к которому приводит этот подход, обычно носит более общий характер, чем это необходимо. Однако дополнительные усилия, потраченные на ООП, обычно компенсируются.

Используя объектно-ориентированное проектирование при решении задач, необходимо идентифицировать возникающие классы. При этом указывается предназначение каждого класса и способ его взаимодействия с другими классами. Таким образом, возникает спецификация каждого класса, в которой указываются его данные и операции. Затем центр внимания перемещается на детали реализации каждого класса, используя подход "сверху вниз" для разработки операций. Классы легче реализовывать по отдельности.

Реализовав класс, необходимо провести его двойное тестирование. Во-первых, нужно проверить операции класса. Для этого обычно создают небольшие программы, вызывающие разные операции и проверяющие результаты в соответствии с их спецификациями. Проверив каждый класс, нужно провести тестирование взаимодействий между классами, возникающих при решении задачи.

Семейство связанных классов

При идентификации классов, возникающих при решении задачи, часто обнаруживаются семейства связанных друг с другом классов. Этот этап занимает много времени, особенно если классы разрабатываются с нуля. Реализовав один класс (называемый **предком** (ancestor)), можно ускорить создание новых классов (**потомков** (descendant)), поскольку потомки могут наследовать данные и операции предка.

Повторное использование классов

Например, как указывалось выше, определив класс часов, можно разработать класс будильников, наследующий свойства часов, но обладающий дополнительными особенностями. На реализацию класса будильников пришлось бы затратить намного больше времени, если бы класс часов не был разработан раньше. Ранее реализованные классы можно применять в новых программах либо без изменения, либо с модификациями, которые включают в себя новые классы, производные от существующих. Повторное использование классов позволяет сократить время, затрачиваемое на объектно-ориентированное проектирование.

Наследование облегчает эксплуатацию и верификацию программ

Объектно-ориентированное программирование оказывает положительное влияние и на другие этапы жизненного цикла программного обеспечения, в частности на эксплуатацию и верификацию программ. Для изменения всей цепочки потомков достаточно модифицировать их предка. Если бы не было наследования, изменения пришлось бы вносить в каждый класс иерархии. Кроме того, программу можно обогатить новыми свойствами, добавляя но-

вых потомков. Это никак не влияет на их предков, и, следовательно, не порождает новых ошибок в остальной части программы. Можно даже добавлять потомков, которые изменяют поведение их предка, даже если он был написан и скомпилирован очень давно.

Краткий обзор основных понятий программирования

Будем считать хорошим наиболее эффективное решение задачи. Тогда возникают вопросы: чем отличается хорошее решение от плохого и как сконструировать хорошее решение? В этом разделе мы попробуем кратко подытожить ответы на эти очень трудные вопросы.

Темы, которые здесь обсуждались, вам должны быть знакомы. Однако новички обычно не придают этим вопросам большого значения. Освоив первый курс программирования, многие студенты считают достаточным, если программа "просто работает". Последующее обсуждение должно помочь читателям понять, насколько важны эти вопросы.

Люди тоже читают программы

Одно из наиболее распространенных заблуждений гласит: программы предназначены только для компьютеров. Как следствие, новички считают, что их программы могут "понимать" только компьютеры — ведь это они их компилируют, выполняют и выдают результаты их работы! Однако и другие люди тоже часто вынуждены читать и модифицировать программы. Обычно в программистской среде над программой работают несколько человек. Один программист может написать программу, которую другие люди будут использовать вместе со своими программами, а через несколько лет совсем другие люди станут ее модифицировать. Следовательно, очень важно, чтобы программы можно было легко читать и понимать.

Программист должен постоянно помнить о шести принципах программирования.

ОСНОВНЫЕ ПОНЯТИЯ

Шесть принципов программирования

1. Модульность.
2. Модифицируемость.
3. Легкость использования.
4. Безопасность.
5. Стиль.
6. Отладка.

Модульность

На протяжении всей книги мы будем постоянно напоминать, что на каждом этапе решения задачи необходимо придерживаться принципа модульности, начиная с самого начала. В предыдущих разделах мы уже указывали, что задачи становятся все больше и сложнее. Модульность позволяет понизить уровень сложности программы. Благотворное влияние модульности проявляется в следующих аспектах программирования.

Модульность облегчает программирование

- **Конструирование программ.** Единственное различие между маленькой модульной программой и большой модульной про-

граммой заключается в количестве модулей, из которых они состоят. Поскольку модули не зависят друг от друга, создание большой модульной программы не очень отличается от написания многих маленьких независимых модульных программ, хотя взаимодействие между модулями может быть весьма сложным. Работа над большой цельной программой напоминает одновременную работу со множеством маленьких взаимосвязанных программ. Кроме того, модульность позволяет применить командный способ программирования, при котором несколько программистов работают независимо друг от друга, а затем объединяют свои модули в одну программу.

- **Отладка программ.** Отладка большой программы может оказаться практически невыполнимой задачей. Представьте себе, что вы набрали 10 000 строк и наконец-то приступили к их компиляции. Ничего не может быть приятнее! Теперь представьте, что при выполнении программы среди нескольких сотен строк вывода вы обнаружили неверное число. Пройдет несколько дней, пока вы продеретесь сквозь переплетения операторов и узнаете причину этой ошибки, которая может оказаться вполне невинной.

  Модульность позволяет изолировать ошибки

  Большое преимущество модульного подхода заключается в том, что задача отладки большой программы сводится к отладке множества маленьких подпрограмм. Начиная кодировать модуль, вы должны быть уверены, что все остальные модули закодированы правильно. Следовательно, закончив программирование модуля, вы должны внимательно проверить его, как отдельно, так и вместе с другими модулями, вызывая его с фактическими аргументами, тщательно подобранными для выявления всех возможных недостатков. Если это тестирование проведено подобающим образом, можно быть уверенным, что любая обнаруженная ошибка содержится только в модуле, который кодировался последним. *Модульность позволяет изолировать ошибки.*

  Теоретически можно применять формальные методы проверки программ. Модульные программы хорошо поддаются такой верификации.

- **Чтение программ.** Человек, читающий большую программу, чувствует себя заблудившимся в глухом лесу. Модульный подход не только позволяет программистам справляться со сложностями, возникающими при решении задачи, но и помогает читателям программы понять, как она работает. Модульную программу легко отследить, поскольку читатель хорошо представляет себе, что происходит, не вдаваясь в детали кода. Для того чтобы разобраться в хорошо написанной функции, достаточно лишь прочитать ее имя, начальные комментарии и имена функций, которые вызываются внутри нее. Читатели программы должны вникать в тонкости кода, только если они хотят понять детали выполняемых операций. Читабельность программ обсуждается в разделе, посвященном стилю программирования.

  Модульные программы легко читать

- **Модификация программ.** Модифицируемость — это тема следующего раздела, однако она тесно связана с модульностью программы, поэтому о ней стоит вспомнить. Небольшое изменение в требованиях, предъявляемых к программе, должно приводить к небольшому изменению ее кода. Если это не так, значит, программа плохо написана и, в частности, не обладает свойством модульности. Чтобы учесть небольшие изменения в исходных требо-

  Модульность изолирует изменения

ваниях, в модульной программе обычно достаточно изменить лишь несколько модулей, особенно если модули не зависят друг от друга (т.е. слабо связаны) и каждый модуль выполняет отдельную точно поставленную задачу (т.е. высоко координирован).

Вносить изменения в программу нужно постепенно. При модульном подходе большие изменения разбиваются на множество маленьких и относительно простых модификаций в изолированных частях программы. *Модульность изолирует модификации.*

- **Исключение избыточного кода.** Другое преимущество модульного подхода проявляется в идентификации вычислений, которые в программе выполняются несколько раз. Такие вычисления следует реализовывать в виде функций. В этом случае код, предназначенный для таких вычислений, в программе встречается только один раз, повышая ее читабельность и модифицируемость. В следующем разделе мы продемонстрируем это на конкретном примере.

Модульность исключает избыточность

Модифицируемость

Представьте себе, что спецификация программы через какое-то время изменилась. Обычно люди не вполне отчетливо представляют себе, чего они хотят от программы, постепенно уточняя ее спецификацию. В этом разделе указаны три способа, позволяющие облегчить изменение программы: использование функций, именованных констант и операторов `typedef`.

Функции. Допустим, что в некую библиотеку входит большая программа для ведения каталога книг. В некоторых точках программа выводит на экран информацию о заказанной книге. В каждой из этих точек программа может вызывать оператор `cout`, для того чтобы вывести на экран номер, фамилию автора и название книги. Этот оператор можно заменить вызовом функции `displayBook`, которая выводит ту же самую информацию.

Функции облегчают модификацию программ

Функции позволяют не только исключить избыточный код, но и облегчают модификацию программ. Например, чтобы изменить формат вывода, достаточно изменить реализацию функции `displayBook`, а не вносить исправления в многочисленные операторы `cout`, как это предполагалось в исходном варианте. Если бы функции не было, пришлось бы вносить изменения в каждой точке программы, где на экран выводится информация о книгах. Найти каждую такую точку было бы достаточно трудно и, вероятно, некоторые из них остались бы не измененными. Этот простой пример наглядно демонстрирует преимущества использования функций.

В качестве другой иллюстрации напомним пример, рассмотренный нами ранее, в котором упорядочивались данные. Разрабатывая алгоритм сортировки в виде отдельного модуля и реализуя его в виде функции, можно сделать программу легко модифицируемой. Например, если алгоритм сортировки окажется слишком медленным, можно просто заменить соответствующую функцию, оставив неизменной остальную часть программы. Нужно лишь "вырезать" старую функцию и "вставить" новую. Если бы сортировка была интегрирована в программу, понадобилась бы довольно сложная хирургическая операция.

В общем, будьте готовы переписать вашу программу, чтобы учесть небольшие изменения в ее спецификации. Обычно хорошо организованные программы модифицируются легко: поскольку каждый ее модуль решает определенную часть общей задачи, небольшое изменение в постановке задачи влияет лишь на отдельные модули.

Именованные константы. Для облегчения модификации программы можно применять именованные константы. Например, если на размер массива, используемого в программе, накладываются ограничения, исправить его довольно сложно. Допустим, что программа использует массив для обработки экзаменационных оценок по компьютерным наукам. В момент написания программы курс компьютерных наук слушали 202 студента, поэтому массив был объявлен следующим образом.

Именованные константы облегчают модификацию программ

```
int scores[202];
```

Программа обрабатывает массив несколькими способами. Например, она считывает оценки, записывает их и усредняет. Псевдокод решения каждой из этих задач содержит примерно такую конструкцию.

```
for (index = 0 through 201)
   Обработка оценок
```

Если количество студентов изменится, нужно не только изменить объявление массива *scores*, но и модифицировать каждый цикл, чтобы учесть новый размер массива. Кроме того, размер массива может влиять на другие операторы в программе. Здесь 202, а там 201 — что изменять?

Однако можно применить именованную константу.

```
const int NUMBER_OF_MAJORS = 202;
```

Тогда массив можно объявить следующим образом.

```
int scores[NUMBER_OF_MAJORS];
```

Псевдокод соответствующих циклов примет такой вид.

```
for (index = 0 through NUMBER_OF_MAJORS-1)
   Обработка оценок
```

В выражениях, которые включают в себя размер массива, нужно использовать именованную константу *NUMBER_OF_MAJORS* (например, *NUMBER_OF_MAJORS-1*). Тогда размер массива можно изменить, изменив всего лишь определение именованной константы и скомпилировав программу снова.

Оператор typedef. Допустим, что ваша программа выполняет вычисления с переменными, имеющими тип *float*, и вдруг обнаружилось, что точности типа *float* недостаточно. Например, для того чтобы изменить объявление типа *float* на объявление типа *long double*, придется пройтись по всем объявлениям и в каждом из них сделать соответствующее изменение.

Операторы typedef облегчают модификацию программ

Для того чтобы облегчить процесс изменений, используется оператор *typedef*, который переименовывает существующий тип. Например, оператор

```
typedef float RealType;
```

объявляет тип *RealType* синонимом типа *float*, что позволяет использовать их с одинаковым успехом. Если в предыдущей программе все переменные типа *float* объявить как переменные типа *RealType*, то программу будет легко модифицировать и читать. Для того чтобы изменить точность вычислений, нужно просто изменить оператор *typedef*.

```
typedef long double RealType;
```

Легкость использования

Разрабатывая интерфейс программы, нужно думать о людях, которые будут с ней работать. Пользователи часто вводят в программу входные данные и анализируют полученные результаты. При этом следует учитывать следующие очевидные особенности.

- В интерактивной среде ввод данных должен быть простым и ясным. Например, приглашение "?" невозможно сравнить с предложением "Пожалуйста, введите номер вашего банковского счета." Никогда не следует рассчитывать, что пользователи интуитивно догадаются, какого ответа ждет от них программа.

Приглашение к вводу данных

- Программа всегда должна выводить эхо входных данных. Если программа считывает данные, неважно, с клавиатуры или из файла, она должна выводить их на экран. Это необходимо по двум причинам. Во-первых, это позволяет пользователям контролировать входные данные, предотвращая опечатки и ошибки. Эта проверка особенно полезна в интерактивном режиме. Во-вторых, выходные данные более осмысленны и самоочевидны, если они содержат исходные данные, введенные пользователем.

Эхо ввода

- Вывод должен быть хорошо размеченным и понятным. Рассмотрим в качестве примера следующий набор выходных данных.

Разметка вывода

```
18:00 6 1
Джонс, К. 223 2234.00 1088.19 Н, О Смит, Т. 111
110.23 З, Харрис, У. 44 44000.000 22222.22
```

- Эти данные намного легче интерпретировать, если вывести их в следующем виде.

```
Счета вкладчиков по состоянию на 18:00 1 июня

Состояние счета: Н - новый, О- общий, З - закрыт

Имя          Номер    Снятие      Вклады      Состояние
Джонс, К.    223      $ 2234.00   $ 1088.19   Н, О
Смит, Т.     111      $  110.23   ---------   З
Харрис, У.    44      $44000.00   $22222.22   ------
```

Это лишь самые общие характеристики хорошего пользовательского интерфейса. В зависимости от более тонких моментов, программы классифицируются от просто пригодных к работе до дружелюбных к пользователю. Обычно студенты стремятся игнорировать необходимость разработки хорошего пользовательского интерфейса. Однако, посвятив этому немного дополнительного времени, они могут обнаружить существенную разницу между хорошей программой и программой, которая просто решает задачу. Рассмотрим, например, программу, предлагающую пользователю ввести строку данных в некотором фиксированном формате, где элементы ввода разделяются только одним пробелом. Свободный формат ввода, допускающий несколько пробелов между данными, был бы более удобен для пользователя. Для создания цикла, игнорирующего пробелы, нужно затратить совсем немного времени, так зачем же навязывать пользователю фиксированный формат? Кроме того, разработав такой интерфейс однажды, вы можете затем постоянно использовать его в своих программах и библиотеках, а пользователь никогда не будет беспокоиться о формате входных данных.

Хороший пользовательский интерфейс имеет большое значение

Надежное программирование

Надежная программа всегда работает безотказно, независимо от способов ее применения. К сожалению, эта цель является практически недостижимой. Намного реальнее ограничить возможности неправильного обращения с программой и предотвратить эти ошибки.

Проверка ошибок при вводе данных

Мы рассмотрим два вида ошибок. Первая разновидность — это *ошибки при вводе данных*. Допустим, например, что программа ожидает ввода неотрицательного числа, а на вход поступает число –12. Обнаружив такую ошибку, программа не должна вычислять неверный результат или прекращать работу, выдав непонятное сообщение об ошибке. Вместо этого надежная программа должна вывести на экран сообщение, имеющее приблизительно следующее содержание.

```
-12 — неправильное количество детей.
Пожалуйста, повторите ввод.
```

Проверка логики программы

Вторая разновидность ошибок — *семантические*, т.е. *ошибки в логике программы*. Хотя они тесно связаны с процессом отладки, который будет обсуждаться в конце этой главы, обнаружение семантических ошибок является этапом безопасного программирования. Внешне совершенно правильные программы в некоторых ситуациях начинают вести себя непредсказуемо, даже если введенные данные были абсолютно корректными. Например, программист мог не предусмотреть реакцию программы на конкретные данные, даже если во всем остальном ее логика безупречна. Кроме того, модифицируя часть программы, авторы часто нарушают предположения, которые должны выполняться в отношении остальных ее частей. Программа должна быть организована так, чтобы семантические ошибки такого рода не возникали. Она должна постоянно контролировать себя, обнаруживая отклонения и неверные результаты.

Предотвращение неверного ввода. Допустим, что мы должны вычислить статистические показатели, касающиеся людей, чей годовой доход колеблется от $10000 до $100000. Суммы округляются до тысяч: $10000, $11000 и т.д. Исходные данные хранятся в файле, состоящем из одной или нескольких строк, имеющих следующий вид.

```
G  N
```

Здесь *N* — это количество людей, попадающих в группу с доходом *G* тысяч долларов в год. Если эти данные записывали несколько разных людей, то в файле могут оказаться несколько записей, относящихся к одному и тому же числу *G*. После ввода данных программа должна суммировать их и записать количество людей, соответствующее каждой величине *G*. В этом контексте совершенно ясно, что *G* — это целое число, изменяющееся в диапазоне от 10 до 100 включительно, а *N* — неотрицательное целое число.

Чтобы продемонстрировать, как можно предотвратить ввод неверных данных, рассмотрим функцию, предназначенную для ввода чисел при решении поставленной выше задачи. Первый вариант этой функции иллюстрирует, насколько программа оказывается далекой от идеала. В конце концов, нам все же удастся приблизить функцию ввода данных к желательному эталону.

Первый вариант функции выглядит следующим образом.

```
const int LOW_END = 10;
// Нижняя граница доходов
const int HIGH_END = 10;   // Верхняя граница доходов
const int TABLE_SIZE = HIGH_END - LOW_END + 1;
typedef int TableType[TABLE_SIZE];

int index(int group)
// Возвращает индекс массива, соответствующий номеру группы.
{
   return group - LOW_END;
} // Конец функции index

void readData(TableType incomeData)
// - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Считывает и организовывает статистические данные о доходах.
// Предусловие: вызываемый модуль выдает инструкции и
// предлагает пользователю ввести данные. Входные данные
// не должны содержать ошибки. Каждая строка имеет вид G N,
// где N — количество людей, чей годовой доход равен
// G тысяч долларов, причем LOW_END <= G <= HIGH_END.
// Ввод данных завершается после считывания строки,
// в которой числа G и N равны нулю.
// Постусловие: число incomeData[G-LOW_END] равно общему
// количеству людей, чей доход равен G тысяч долларов для
// каждого считанного значения G. Считанные значения
// выводятся на экран.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   int group, number; // Входные значения

   // Очищаем массив
   for (group = LOW_END; group <= HIGH_END; ++group)
      incomeData[index(group)] = 0;

   for (cin >> group >> number;
      (group != 0) || (number != 0);
      cin >> group >> number)
   {  // Инвариант: переменные group и number не равны нулю
      cout << "Количество людей в группе" << group <<
         " равно " << number << ".\n";
      incomeData[index(group)] += number;
   } // Конец цикла for
} // Конец функции readData
```

Эта функция порождает несколько проблем. Если входная строка содержит неожиданные данные, программа не сможет на них адекватно среагировать. Рассмотрим две конкретные возможности.

- Первое целое число, которое функция присваивает переменной *group*, выходит за пределы допустимого диапазона (от *LOW_END* до *HIGH_END*). В этом случае обращение к элементу массива *income[index(group)]* становится некорректным.
- Второе целое число, которое функция присваивает переменной *number*, является отрицательным. Несмотря на то что отрицательное значение переменной *number* лишено смысла, так как количество людей в группе не может быть меньше нуля, функция добавит его в массив. Таким образом, массив *incomeData* будет содержать неверные данные.

После считывания данных нужно проверить, лежит ли значение переменной *group* в допустимом диапазоне (от *LOW_END* до *HIGH_END*) и является ли переменная *number* положительной. Если это не так, необходимо обработать ошибку ввода.

Проверка неправильных входных данных

Вместо проверки переменной *number* можно было бы проверить, положителен ли элемент *incomeData[index(group)]* после добавления к нему числа *number*. Однако такой подход неэффективен. Во-первых, добавить отрицательное число к элементу массива *incomeData* можно так, что сам он не станет отрицательным. Например, если число *number* равно −4000, а соответствующий элемент массива *incomeData* равен 10000, то их сумма будет равна 6000. Следовательно, факт, что число *number* отрицательно, останется незамеченным. Это приведет к неправильной работе всей остальной программы.

При обнаружении неправильных входных данных возможны несколько сценариев. Во-первых, функция может установить соответствующий признак ошибки и прекратить работу. Во-вторых, функция может установить соответствующий признак ошибки, проигнорировать ее и продолжить работу. Какой из этих сценариев выбрать, зависит от конкретной ситуации.

Функция *readData*, приведенная ниже, универсальна и максимально облегчает модифицируемость программы, в которой она используется. Обнаружив ошибку при вводе данных, она задает ее признак, игнорирует неправильную строку и продолжает работу. Установив признак ошибки, функция предоставляет вызывающему модулю возможность самому принять решение — прекратить работу или продолжить выполнение программы. Таким образом, эту функцию можно применять в разных контекстах, легко модифицируя реакцию на обнаружение ошибки.

Надежная функция

```
bool readData(TableType incomeData)
// - - - - - - - - - - - - - - - - - - - -
// Считывает и организовывает статистические данные.
// Предусловие: вызываемый модуль выдает инструкции и
// предлагает пользователю ввести данные. Каждая строка
// содержит два целых числа в виде G N, где N — количество
// людей, чей годовой доход равен G тысяч долларов, причем
// LOW_END <= G <= HIGH_END. Ввод данных завершается после
// считывания строки, в которой числа G и N равны нулю.
// Постусловие: число incomeData[G-LOW_END] равно общему
// количеству людей, чей доход равен G тысяч долларов для
// каждого считанного значения G. Считанные значения
// выводятся на экран. Если числа G или N неверны
// (не равны нулю и G < LOW_END, G > HIGH_END или N < 0),
// функция игнорирует строку ввода, задает возвращаемое
// значение равным false и продолжает работу.
// Решение о продолжении выполнения программы принимает
// вызывающий модуль. Если входные данные не содержат ошибок,
// функция возвращает значение true.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   int group, number; // Входные значения
   bool dataCorrect = true; // Ошибок пока нет

   for (group = LOW_END; group <= HIGH_END; ++group)
      incomeData[index(group)] = 0;
```

```
    for (cin >> group >> number;
         (group != 0) || (number != 0);
         cin >> group >> number)
    {    // Инвариант: переменные group и number не равны нулю
         cout << "Количество людей в группе" << group <<
            " равно " << number << ".\n";

         if ((group >= LOW_end) && (group <= HIGH_END) &&
            (number >= 0))
            // Входные данные корректны -- добавляем их в счетчик
            incomeData[index(group)] += number;

         else
            // Ошибка при вводе данных:
            // устанавливаем признак ошибки, игнорируя строку
            dataCorrect = false;
    } // Конец цикла for
    return dataCorrect;
} // Конец функции readData
```

Хотя в большинстве случаев эта функция работает отлично, все же она еще недостаточно надежна. Что произойдет, если входная строка будет содержать лишь одно целое число? А если числа в этой строке окажутся нецелыми? Функция была бы более надежной, если бы она считывала данные посимвольно, конвертируя их в целое число и проверяя конец строки. Чаще всего это было бы небольшим излишеством. Однако если люди, вводящие данные, часто делают ошибки, набирая нецелые числа, функцию ввода можно было бы легко изменить, поскольку она реализована в виде изолированного модуля. В любом случае в комментариях, сопровождающих текст функции, нужно формулировать все предположения о входных данных и указывать, как функция реагирует на неправильный ввод.

Предотвращение семантических ошибок. Рассмотрим теперь вторую разновидность: семантические ошибки. Их иногда не удается выловить на этапе отладки и легко внести, модифицируя программу.

К сожалению, сама программа не может сообщить, что в ней кроется ошибка. (Неправильная программа не знает, что она неправильная.) Однако в программу можно включить проверку определенных условий, которые должны выполняться, если она работает правильно. Как уже указывалось, эти условия называются инвариантами.

Функции должны проверять свои инварианты

В качестве простого инварианта рассмотрим предыдущий пример. *Все целые числа в массиве `incomeData` должны быть неотрицательными.* Хотя выше мы сказали, что проверка элементов массива `incomeData` неэффективна при анализе числа `number`, ее можно использовать в качестве *дополнительного условия*. Например, если функция `readData` обнаружила, что какой-то элемент массива `incomeData` выходит за пределы допустимого диапазона, это является сигналом о потенциальных проблемах.

Функции должны проверять предусловия

Еще один способ повышения надежности программы заключается в проверке предусловий функций. Рассмотрим, например, функцию `factorial`, вычисляющую факториал целого числа.

```
int factorial(int n)
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вычисляет факториал целого числа.
// Предусловие: n >= 0.
// Постусловие: если n > 0, возвращает n * (n-1) * ... * 1;
// если n = 0, возвращает число 1.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   int fact = 1;

   for (int i = n; i > 1; --i)
      fact *= i;

   return fact;
} // Конец функции factorial
```

Комментарии, помещенные в начале этой функции, содержат предусловие — информацию о сделанных предположения. *Это нужно делать всегда.* Значение, возвращаемое функцией, будет правильным, только если выполняется ее предусловие. Если число *n* меньше нуля, функция вернет неверный результат — число 1.

В контексте программы, частью которой является эта функция, предположение, что число *n* никогда не бывает отрицательным, может иметь определенный смысл. Таким образом, если остальная часть программы работает правильно, она будет вызывать функцию *factorial* только для корректных значений *n*. Ирония заключается в том, что именно это рассуждение обосновывает проверку значения переменной *n* внутри функции *factorial*: если оно отрицательно, значит, в программе есть ошибка.

Для проверки числа *n* в функции *factorial* есть еще одна причина: функция должна быть корректной вне контекста программы. Это значит, что если вы используете эту функцию в другой программе, она также должна предупреждать вас об ошибке, если значение переменой *n* отрицательно.

Функция должна проверять значения своих аргументов

Желательно, чтобы проверка была более строгой и не сводилась лишь к формулировке предусловия. Это означает, что *функция должна настаивать на выполнении сделанных предположений и, по возможности, проверять, соответствуют ли ее аргументы этим предположениям.*

В нашем примере функция *factorial* могла бы проверять значение переменной *n*, и, если оно оказалось отрицательным, возвращать значение 0, поскольку факториал никогда не равен нулю.

Кроме того, функция *factorial* может прекращать выполнение программы, если ее аргумент меньше нуля. Во многих языках программирования, включая язык C++, существует механизм для обработки ошибок, называемый **исключительной ситуацией** (exception). Модуль подает сигнал о возникшей ошибке, **генерируя** (throwing) исключительную ситуацию. Модуль может реагировать на исключительную ситуацию, возбужденную другим модулем, **перехватывая** (catching) ее и выполняя код, предназначенный для обработки ошибок. Более подробно исключительные ситуации рассматриваются в главе 3.

В языке C++ есть удобный макрос *assert(expr)*, который выводит информативное сообщение и прекращает выполнение программы, если выражение *expr* равно нулю. Макрос *assert* можно применять как для обнаружения ошибок, так и для проверки предусловий.

Обработка ошибок обсуждается в следующем разделе.

Стиль

В этом разделе рассматриваются восемь вопросов, касающихся стиля программирования.

ОСНОВНЫЕ ПОНЯТИЯ

Восемь вопросов, касающихся стиля программирования

1. Широкое использование функций.
2. Использование закрытых данных-членов.
3. Избегание глобальных переменных в функциях.
4. Правильное применение аргументов, передаваемых по ссылке.
5. Правильное применение функций.
6. Обработка ошибок.
7. Читабельность.
8. Документация.

Разумеется, изложенные ниже высказывания отражают личные предпочтения авторов. Возможны и другие мнения о том, что считать хорошим стилем программирования.

Функции лишними не бывают

Широкое использование функций. Функциями трудно злоупотребить. Если в программе есть несколько фрагментов идентичного кода, их следует оформить в виде функции. Однако это не единственная причина, по которой следует применять функции.

Несмотря на то что программы, не содержащие функций, выполняются быстрее программ, состоящих из многих функций, они не становятся от этого эффективнее. Использование функций повышает эффективность, если принимать во внимание стоимость человеческого труда, вложенного в создание программы. Мы уже убедились в преимуществах модульного подхода к конструированию программ. Кроме того, компиляторы могут сокращать время, затрачиваемое на вызовы функций, заменяя их подставляемыми операторами, выполняющими то же самое задание.

Использование закрытых данных-членов. Каждый объект состоит из функций, представляющих собой операции, которые должны над ним выполняться. Помимо этого, объект содержит данные-члены, в которых записана надлежащая информация. Точное представление этих полей следует скрывать от модулей, использующих данный объект, объявляя данные-члены закрытыми. Это полностью соответствует принципу сокрытия информации. Детали реализации объекта должны быть скрыты от постороннего взгляда, а для взаимодействия с внешним миром следует предусмотреть функции, осуществляющие получение и передачу информации. Даже если единственными операциями над данными-членами объекта являются операции извлечения и изменения, в объекте следует предусмотреть функцию, называемую **методом доступа** (accessor), возвращающую значение поля, и функцию, называемую **модифицирующим методом** (mutator), задающую значение поля. Например, объект `Person` может предоставлять доступ к своему полю `theName` через функцию `getName`, а для изменения имени использовать функцию `setName`.

Не используйте глобальные переменные

Избегание глобальных переменных в функциях. Одно из основных преимуществ функций заключается в том, что они реализуются в виде

отдельных модулей. Если в функции используются глобальные переменные, эта изолированность нарушается. Это приводит к появлению **побочного эффекта** (side effect). Таким образом, применение глобальных переменных внутри функций нарушает изоляцию ошибок и модификаций.

Для передачи параметров в функцию используйте передачу по значению

Правильное применение аргументов, передаваемых по ссылке. Функции взаимодействуют с остальными частями программы с помощью своих аргументов. **Передача параметров по значению** (value arguments) используется по умолчанию, если после типа формального аргумента не поставлен знак &. При этом любые изменения, происходящие над формальными аргументами внутри функции, никак не отражаются на фактических параметрах, передаваемых ей вызывающим модулем. Взаимодействие между вызывающей программой и функцией является односторонним. Поскольку ограничения, накладываемые на одностороннее взаимодействие, основаны на понятии изолированного модуля, передачу параметров по значению следует применять при малейшей возможности.

Для возврата значения из функции используйте передачу по ссылке

В каких случаях применяется **передача параметров по ссылке** (reference arguments) ? Очевидно, это необходимо, когда функция должна возвращать в вызывающую программу несколько значений одновременно. Допустим, что функция имеет аргумент *x*, значение которого не изменяется. Естественно применить для этого аргумента передачу по значению. Однако это вынуждает функцию при вызове копировать значение фактического аргумента *x* во временную локальную переменную. Это практически незаметно, если переменная *x* невелика, но при копировании больших объектов накладные затраты могут оказаться значительными. В то же время, если бы переменная *x* передавалась по ссылке, то никакие копии создавать не пришлось бы, и затраты компьютерных ресурсов оказались бы существенно ниже.

Если копирование аргумента нежелательно, вместо передачи по значению используйте константный аргумент, передаваемый по ссылке

С передачей аргумента по ссылке связана одна проблема: она искажает информацию о взаимодействии функции с остальной частью программы. Передачу параметра по ссылке принято использовать для возврата результата в вызывающий модуль. Если этот механизм применяется для передачи аргументов, то читабельность программы снижается, и возрастает вероятность появления ошибок при ее модификациях. Аналогичная ситуация возникает, когда в программе есть переменная, которая никогда не меняет своего значения. В этом случае логичнее использовать константу. Проблема решается просто: перед объявлением формального аргумента следует указать ключевое слово `const`, которое предотвратит изменение соответствующего фактического параметра.

Правильное использование функций. Как известно, **функции, вычисляющие значение** (valued function), возвращают результат своей работы с помощью оператора `return`, а **пустые функции** (void function) не возвращают ничего. Функции, возвращающие значения, позволяют программистам создавать новые *выражения* (expressions). Каждый раз, когда нужно вычислить некое значение, можно вызвать соответствующую функцию, определенную пользователем, если она не предусмотрена в самом языке. Это полностью совпадает с понятием математической функции. Таким образом, *функции, вычисляющие значения, должны возвращать единственный результат*. Вообразите себе функцию `2*x`, изменяющую значения еще пяти переменных! Разумеется, функция, возвращающая значение, не должна иметь побочных эффектов.

Итак, перечислим то, чего не должна делать функция, возвращающая значение.

- Использовать передачу аргументов по ссылке. Если аргументы нужно передать по ссылке, следует применять пустые функции.

Функции, возвращающие значения, не должны иметь побочных эффектов

- Выполнять ввод или вывод данных.

Вполне возможно, что в определенных ситуациях эти правила можно нарушать, особенно программируя на языке C++. Например, эти требования не относятся к обработке ошибок, описанной в следующем разделе. Фактически многие стандартные функции языка C++ возвращают значения и одновременно изменяют свои аргументы, при этом возвращаемое значение служит признаком ее успешного или безуспешного выполнения.

В случае обнаружения ошибки функции должны возвращать значение или генерировать исключительную ситуацию, но не выводить сообщение об ошибке

Обработка ошибок. Надежная программа должна проверять как ошибки ввода, так и семантические ошибки, а также правильно на них реагировать. Функция должна проверять лишь определенные виды ошибок, например, неверный ввод или неправильные значения аргументов. В зависимости от контекста, реакция функции может варьироваться от игнорирования неверных данных до прекращения работы программы. В рассмотренной выше программе функция `readData` возвращала в вызывающий модуль булево значение, отмечая, что при вводе данных была обнаружена ошибка. Выбор соответствующей реакции предоставлялся вызывающему модулю. В общем случае функции должны либо возвращать признак ошибки, либо генерировать исключительную ситуацию, но не выводить на экран сообщение об ошибке.

Читабельность. Для того чтобы программу было легко читать и отслеживать, она должна иметь хорошую структуру, правильно выбранные идентификаторы, содержать нужное количество пустых строк и сопровождаться подробной документацией. Следует избегать изощренных программистских трюков, которые экономят компьютерное время за счет времени, затрачиваемого людьми на их анализ. Далее в книге мы неоднократно проиллюстрируем это утверждение примерами.

Имена идентификаторов должны быть понятны и самоочевидны. Они должны отличаться от ключевых слов, таких как `int` и др. В книге приняты следующие соглашения.

Стиль идентификаторов

- Ключевые слова набираются строчными буквами и полужирным шрифтом.
- Имена стандартных функций набираются строчными буквами.
- В идентификаторах, определенных пользователем, могут использоваться как строчные, так и прописные буквы.
  - Классы называются именами существительными, причем каждое слово начинается с прописной буквы.
  - Имена функций внутри классов являются глаголами. Первое слово начинается со строчной буквы, остальные — с прописной.
  - Переменные начинаются со строчной буквы, а следующие слова, образующие их имена, — с прописной.
  - Типы данных, объявленные в операторе `typedef`, а также имена структур и перечислений, начинаются с прописных букв.
  - Имена констант и счетчиков целиком состоят из прописных букв, а для разделения слов используют символ подчеркивания.

- Остальные соглашения об именах носят методический характер.
  - Имена типов, объявленных в операторе *typedef*, заканчиваются словом *Type*.
  - Имена исключительных ситуаций заканчиваются словом *Exception*.

Два соглашения методического характера

Для повышения читабельности программ следует использовать свободный стиль форматирования текста. Программа должна быть написана так, чтобы ее модули сразу бросались в глаза. Каждая функция должна отделяться от остального текста пустой строкой. Внутри функций и главного модуля отдельные блоки кода также следует перемежать пустыми строками, облегчая чтение программы. Обычно (но не всегда) под блоками понимается некая управляющая структура, например цикл *while* или оператор *if*.

Есть несколько хороших стилей свободного форматирования текста программы. Рассмотрим четыре наиболее важных из них.

- Блоки должны четко отделяться друг от друга.
- Форматирование должно быть последовательным: идентичные конструкции должны выглядеть одинаково.
- Стиль форматирования должен учитывать проблему **дрейфа вправо** (rightward drift), которая заключается в том, что вложенные блоки наезжают на правое поле страницы.
- В составных операторах открывающие и закрывающие фигурные скобки должны быть выровнены.

Принципы свободного форматирования

```
{
 <оператор1>
 <оператор2>
 ...
 <операторn>
}
```

Остальные элементы форматирования текста — дело личного вкуса программистов. Ниже приводится краткий обзор стиля, который применяется на протяжении всей книги.

- Операторы цикла *for* или *while*, тело которых состоит только из одного оператора, записываются так.

Стиль форматирования, принятый в книге

```
while (выражение)
   оператор
```

- Если они состоят из нескольких операторов, применяется такая запись.

```
while (выражение)
{
   операторы
} // Конец цикла while
```

- Оператор *do*, выполняющий одно действие, имеет следующий вид.

```
do
   оператор
while (выражение);
```

- Если он состоит из нескольких операторов, то применяется такая запись.

```
do
{
   операторы
} while (выражение);
```

- Оператор *if*, выполняющий одно действие, имеет следующий вид.

```
if (выражение)
   оператор1
else
   оператор2
```

- Если он состоит из нескольких операторов, то применяется такая запись.

```
if (выражение)
{
   операторы
}
else
{
   операторы
} // Конец оператора if
```

- В одном конкретном случае, когда три и более операторов *if* вложены друг в друга, лучше применять другой стиль. Например, две формы записи, приведенной ниже, совершенно эквивалентны по смыслу. В стиле с отступами эта запись выглядит следующим образом.

```
if (условие1)
   действие1
else if (условие2)
        действие2
    else if (условие3)
             действие3
```

- В стиле без отступов она выглядит иначе.

```
if (условие1)
   действие1
else if (условие2)
   действие2
else if (условие3)
   действие3
```

- Второй стиль лучше отражает природу этой конструкции, которая напоминает обобщенный оператор *switch*.

```
case условие1 : действие1; break;
case условие2 : действие2; break;
case условие3 : действие3; break;
```

- Фигурные скобки повышают читабельность программы, даже если этого не требуют синтаксические правила. Например, в приведенной ниже конструкции фигурные скобки использовать не обязательно, поскольку тело оператора *if* состоит лишь из одного оператора. Однако эти скобки четче очерчивают область видимости оператора *while*.

```
while (выражение)
{
   if (условие1)
      оператор2
   else
      оператор2
} // Конец оператора while
```

Документация. Программа должна сопровождаться подробной документацией, чтобы ее можно было легко читать, использовать и модифицировать. Сейчас используются многие стили документирования программ. Их выбор зависит от конкретной программы и личных предпочтений. Однако при документировании программы нужно придерживаться следующих общих принципов.

ОСНОВНЫЕ ПОНЯТИЯ

Основные характеристики программной документации

1. Комментарии в начале программы должны содержать следующие пункты.
   1.1. Предназначение программы.
   1.2. Автор и дата создания.
   1.3. Описание ввода и вывода.
   1.4. Описание способа применения программы.
   1.5. Предположения об ожидаемых типах данных.
   1.6. Перечисление возможных исключительных ситуаций.
   1.7. Краткое описание основных классов.
2. В комментариях, помещенных в начале каждого класса, указывается его предназначение и описываются данные, содержащиеся в нем (константы и переменные).
3. В комментариях, помещенных в начале каждой функции, указывается ее предназначение, предусловия, постусловия и вызываемые функции.
4. Комментарии, размещенные в теле каждой функции, должны пояснять ее основные свойства и особенности логики.

Не забывайте о людях, которые будут читать ваши комментарии

Начинающие программисты стремятся приуменьшить роль хорошей документации, поскольку компьютер не умеет читать комментарии. Однако вы должны понять, что люди тоже читают программы. Ваши комментарии должны быть достаточно ясными и понятными всем, кто будет применять написанную вами функцию или модифицировать ее. Таким образом, одни комментарии должны быть предназначены в первую очередь людям, которые будут работать с вашей функцией, а другие — тем, кто ее будет изменять. Эти виды комментариев следует четко различать.

Документацию нужно создавать в процессе разработки программы

Начинающие программисты обычно создают документацию в самом конце. Однако документировать программу нужно одновременно с

ее разработкой. Поскольку разработка большой программы может занять длительный период времени, факты, которые казались очевидными, через несколько недель будут забыты. Чтобы этого не случилось, каждый шаг разработки программы нужно сопровождать документацией.

Отладка

Как бы тщательно вы ни писали программу, она будет содержать ошибки, которые необходимо выявить и исправить. К счастью, модульные, ясные и хорошо документированные программы обычно успешно поддаются отладке. Методы предупреждения отказов, предназначенные для обнаружения ошибок и выдачи сообщений при их обнаружении, также могут оказать неоценимую помощь.

Многие студенты понятия не имеют, что делать с ошибками, обнаруженными в их программах. Они просто не умеют систематически отслеживать ошибки. Без систематического подхода обнаружить маленькую ошибку в большой программе было бы практически невозможно.

Основная трудность, подстерегающая программистов на этапе отладки программы, заключается в том, что они часто выдают желаемое за действительное. Например, получив при выполнении программы сообщение, гласящее, что в строке 1098 содержится ошибка, студент может заявить: "Это невозможно. Оператор, находящийся в строке 1098, даже не выполнялся, поскольку он находится в разделе `else`, а я уверен, что этот раздел не выполнялся." Однако одного протеста мало. Нужно либо отследить выполнение программы, используя доступные средства отладки, либо добавить операторы вывода, для того чтобы продемонстрировать, какая из частей оператора `if` была выполнена. Для этого нужно верифицировать значение выражения, входящего в оператор `if`. Если это выражение равно 0, а вы ожидали, что оно будет равно 1, то придется разбираться, отчего это произошло.

Для обнаружения логических ошибок используйте средства наблюдения или временные операторы cout

Как обнаружить точку, в которой программа работает неправильно? Обычно среда программирования предоставляет возможность отслеживать выполнение программы либо с помощью пошаговой трассировки операторов, либо путем установки **точек прерывания** (breakpoints), в которых выполнение программы должно быть временно приостановлено. Значение конкретных переменных можно выявить либо с помощью **средств наблюдения** (watches), либо вставив в определенные точки программы временные операторы вывода. Основное предназначение отладки — сообщать вам, что происходит при выполнении программы. Это может звучать слишком приземленно, но главная задача программиста при отладке — эффективно использовать предоставленные ему возможности. Помимо всего прочего, отладка не сводится к простой установке точек прерывания, настройке средств наблюдения и вставке операторов вывода с последующим анализом поступающей случайной информацией.

Систематически проверяйте логику программы

Основная идея отладки заключается в систематической локализации точек, вызывающих проблемы. Из логики программы следует, что в разных точках программы должны выполняться определенные условия. (Напомним, что эти условия называются инвариантами.) Если результат работы программы отличается от ожидаемого (ведь вы сформулировали инварианты, не так ли?), фиксируется ошибка. Для того чтобы исправить ее, сначала нужно найти, в какой точке условия отличаются от инварианта. Вставив точки прерывания, настроив средства наблюдения либо вставив операторы промежуточного вывода в стратегически важных местах программы, — например, на входе и выходе из циклов и функций, — вы систематически изолируете ошибку.

Этот способ отладки позволяет найти точку, до которой ошибка не проявляется, и точку, после которой результаты становятся неверными. Между эти точками и находится ошибка. Допустим, что до вызова функции F_1 все шло прекрасно, а при вызове функции F_2 произошла ошибка. Это позволяет нам ограничить область поиска ошибки этими двумя точками. Последовательно сужая эту область, мы обнаружим несколько операторов, в которых может содержаться ошибка. Ошибке просто некуда деваться, и рано или поздно мы ее найдем.

Умение выбирать места для установки точек прерывания и промежуточных операторов вывода, настраивать средства наблюдения и анализировать поступающую информацию частично достигается путем логических размышлений, а частично приобретается с опытом. Укажем несколько основных принципов отладки.

Отладка функций. Следует проверять значения аргументов в начале и конце функции, используя средства наблюдения или промежуточные операторы вывода *cout*. В идеале перед использованием в программе каждая из функций должна быть отлажена отдельно.

Отладка циклов. Необходимо проверять значения ключевых переменных в начале и конце цикла, отмеченных комментариями.

```
// Проверка значений переменных start и stop перед входом в цикл
for (index = start; index <= stop; ++index)
{
   // Проверка значений переменных index и key
   // в начале итерации
   ...
   // Проверка значений переменных index и key
   // в конце итерации
}  // Конец цикла for
// Проверка значений переменных start и stop перед выходом из цикла
```

Отладка операторов *if*. Непосредственно перед выполнением оператора *if* необходимо проверить значения переменных, входящих в условное выражение. Для проверки ветвей оператора *if* можно использовать точки прерывания либо операторы промежуточного вывода, как указано в комментариях.

```
// Проверка переменных, входящих в выражения,
// перед выполнением оператора if
if (выражение)
{
   cout << "Условие выполняется (значение выражения равно 1).";
   ...
}

else
{
   cout << "Условие не выполняется (значение выражения равно 0).";
   ...
} // Конец оператора if
```

Использование операторов cout. Иногда операторы *cout* оказываются более удобными, чем средства наблюдения. Такие операторы могут выводить на экран информацию не только о значении переменной, но и о месте программы, где они приобретают эти значения. Обозначить точки программы можно с помощью комментариев.

```
// Это точка A.
cout << "В точке A функции computeResults:\n"
     << "x=" << x << ", y=" << y << endl;
```

Напомним, что после отладки программы эти операторы можно удалить или закомментировать.

Использование специальных функций для вывода отладочной информации. Часто приходится отслеживать значения массивов или других, более сложных структур данных. Для того чтобы отладочная информация выводилась на экран в удобном для чтения виде, необходимо создавать специальные функции для вывода отладочной информации (dump functions). Отдельный оператор, вызывающий специальную отладочную функцию, можно легко перемещать по программе, локализуя ошибку. Время, затраченное на разработку таких функций, окупается сторицей, поскольку их можно применять для отладки разных частей программы.

Надеемся, что нам удалось показать вам, насколько важно *эффективно использовать диагностические средства отладки.* Даже самые опытные программисты не могут избежать отладки. Таким образом, программист-профессионал должен быть хорошим отладчиком.

Резюме

1. Технология программирования — это область компьютерных наук, изучающая способы разработки компьютерных программ.
2. Жизненный цикл программного обеспечения состоит из нескольких этапов: постановки задачи, разработки алгоритма, оценки риска, верификации алгоритма, кодирования программ, тестирования программ, уточнения решения, использования программного обеспечения и поддержки программного обеспечения.
3. Инвариант цикла — это свойство алгоритма, которое должно выполняться до и после каждой итерации цикла. Инварианты цикла позволяют разрабатывать итерационные алгоритмы и проверять их правильность.
4. Оценивая качество решения, необходимо учитывать большое количество факторов: правильность решения, его эффективность, время, затраченное на его разработку, легкость использования, стоимость модификации и усовершенствования.
5. Сочетание объектно-ориентированного подхода и проектирования "сверху вниз" приводит к модульному решению. Данные и операции над ними инкапсулируются в классах. Классы можно идентифицировать, анализируя имена существительные, употребленные при постановке задачи. Алгоритмы следует разбивать на независимые подзадачи, постепенно уточняя их. В любом случае применяется абстракция данных, т.е. внимание сосредоточивается на том, что делает модуль, а не на том, как он это делает.
6. Язык UML — это язык моделирования, используемый для описания объектно-ориентированных проектов. Он предоставляет функциональные возможности для описания данных и операций и применяет диаграммы для выявления отношений между классами.
7. Стремитесь к тому, чтобы окончательное решение можно было легко модифицировать. Обычно модульные программы модифицируются легко, поскольку изменения в одном модуле не затрагивают остальных. Программы не должны зависеть от конкретной реализации своих модулей.
8. Функции должны быть как можно более независимыми и выполнять одну точно поставленную задачу.

9. Функции всегда должны сопровождаться комментариями, которые помещаются в их начало, формулируют их предназначение, а также предусловие, которое должно выполняться в начале модуля, и постусловие, которое должно выполняться в конце модуля.
10. Программа должна быть максимально надежной. Например, программа должна иметь защиту от ошибок при вводе данных и логических ошибок. С помощью проверки инвариантов — условий, которые должны выполняться в определенных точках программы, — можно отслеживать правильность выполнения программы.
11. Эффективное использование доступных диагностических средств — ключ к успешной отладке программ. Для проверки значений переменных в ключевых точках следует применять средства наблюдения и операторы промежуточного вывода *cout*. Их нужно размещать в начале и в конце функций и циклов, а также внутри ветвей условных операторов.

Предупреждения

1. Программа должна содержать средства, защищающие ее от ошибок. Надежные программы всегда проверяют правильность входных данных и сообщают об ошибках. Ошибка ввода не должна приводить к прекращению работы программы, пока не поступит сообщение, в чем именно заключается ошибка. Надежные программы должны распознавать логические ошибки. Например, во многих ситуациях функция должна проверять, правильные ли значения принимают ее аргументы.
2. Перечисленные ниже советы позволяют писать более правильные программы за более короткое время.
   - Пишите точные спецификации программы
   - Используйте модульный подход
   - Формулируйте пред- и постусловия каждой функции до начала ее реализации
   - Используйте осмысленные идентификаторы и последовательный стиль оформления программы
   - Пишите комментарии, включая диагностические утверждения и инварианты

Вопросы для самопроверки

Ответы на вопросы для самопроверки приведены в конце книги.

1. Сформулируйте инвариант следующего цикла.

```
int index = 0;
int sum = item[0];

while (index < 0)
{
   ++index;
   sum += item[index];
} // Конец цикла while
```

2. Напишите спецификацию, используя язык UML, для функции, вычисляющей сумму первых пяти положительных целых чисел из массива, в котором хранятся *n* произвольных целых чисел.

Упражнения

1. Стоимость вещи, которую вы хотите купить, выражается в долларах и центах. Вы платите наличными, отдавая клерку *d* долларов и *c* центов. Напишите спецификацию функции, вычисляющей сдачу, если она вам полагается. Обязательно укажите ее предназначение, пред- и постусловия, а также приведите описание ее аргументов.
2. Дата состоит из месяца, дня и года. Часто она выражается целым числом. Например, для даты 4 июля 1776 количество месяцев равно 7, дней — 4 и лет — 1776.
   - Напишите спецификации для функции, получающей на вход произвольную дату. Обязательно укажите ее предназначение, пред- и постусловия, а также приведите описание ее аргументов.
   - Напишите реализацию этой функции на языке C++. Включите в нее комментарии, позволяющие эксплуатировать ее в будущем.
3. Проанализируйте следующую программу, которая считывает и записывает идентификационный номер, возраст, оклад (в тысячах долларов) и имя каждого сотрудника, входящего в группу. Как ее можно улучшить? Некоторые моменты очевидны, но другие нужно обнаружить. Придерживайтесь принципов, изложенных в этой главе.

```
#include <iostream.h>

int main()
 {
   int x1, x2, x3, i;
   char name[8];

   for (cin >> x1 >> x2 >> x3; x1 != 0;
        cin >> x1 >> x2 >> x3)
     {
        for (i = 0; i < 8; ++i)
        cin >> name[i];

          cout << x1 << x2 <, x3 << endl;
          for (i = 0; i < 8; ++i)
          cin >> name[i];
          cout << endl;
     } // Конец wикла for
        return 0;
}  // Конец функции main
```

4. В этой главе подчеркивалась важность широкого использования проверок. Почему приведенная выше функция работает неправильно? Как этого избежать?

```
double compute(double x)
{
   return sqrt(x)/cos(x);
}  // Конец функции compute
```

5. Напишите инвариант цикла для функции *factorial*, описанной в разделе "Надежное программирование".
6. Напишите функцию, описанную в упражнении 2, и сформулируйте инварианты циклов.
7. Используя инварианты циклов, продемонстрируйте, что алгоритм, описанный в упражнении 1, правильно вычисляет сумму *item[0]+item[1]+...+item[n]*.
8. В следующей программе предполагается, что функция **floor** вычисляет целую часть квадратного корня из числа *x*. (Целая часть числа *n* равна наибольшему целому числу, не превышающему числа *n*.)

```
#include <iostream.h>
// Вычисляет и выводит значение floor(sqrt(x)) для x>=0.
int main()
{
   int x; // Входное значение

   // Инициализация
   int result = 0; // Будет хранить результат вычислений
   int temp1 = 1;
   int temp2 = 1;

   cin >> x; // Считываем входное значение

   // Вычисляем целую часть
   while (temp1 < x)
   {
      ++result;
      temp2 += 2;
      temp1 += temp2;
   } // Конец цикла while

   cout << "Целая часть квадратного корня из числа
        << x << "равна" << result << endl;
   return 0;
} // Конец функции main
```

Эта программа содержит ошибку.

- Чему равен ответ при x = 64?
- Запустите программу и удалите ошибку. Опишите ваши действия.
- Как сделать эту программу более дружелюбной и надежной?

9. Допустим, что в результате серьезной ошибки программа прервала свою работу в точке, расположенной глубоко внутри вложенных вызовов, циклов *while* и операторов *if*. Напишите диагностическую функцию, которая распознает ошибочные значения аргументов (некий вид мнемонического перечисления), выводит на экран сообщение об ошибке и прекращает работу программы.

Задачи по программированию

1. Опишите программу, которая вводит информацию о сотрудниках в массив структур, упорядочивает их в соответствии с идентификационными номерами, выводит на экран упорядоченный массив и вычисляет разнообразные статистические показатели. Напишите ее полную спецификацию на языке UML и разработайте модульное решение. Какие функции можно идентифицировать при разработке решения? Напишите их спецификации, а также пред- и постусловия.
2. Напишите программу, сортирующую и вычисляющую игральные карты при игре в бридж. На вход программы поступает поток, состоящий из пар символов, обозначающих игральные карты. Например, поток

   ```
   2C QD TC AD 6C 3D TD 3H 5H 7H AS JH KH
   ```

3. представляет собой двойку треф, бубновую даму, десятку треф, бубнового туза и т.п. Каждая пара состоит из ранга и масти, где рангами являются символы A, 2, ..., 9, T, J, Q или K, а мастью C, D, H или S. Можно предполагать, что каждая строка ввода состоит из описания ровно 13 карт и не содержит ошибок.
4. Для каждой строки ввода формируется набор, состоящий из 13 карт. Каждый набор выводится на экран в порядке возрастания рангов и мастей (самой старшей картой является туз). Затем вычисляется оценка набора, основанная на обычных правилах игры в бридж.

   Туз стоит 4 балла

   Король стоит 3 балла

   Дама стоит 2 балла

   Валет стоит один балл

   Пустышки (нет карт в наборе) стоят 3 балла

   Одиночки (одна карта в наборе) стоят 2 балла

   Дублеты (две карты в наборе) стоят 1 балл

   Длинные масти, состоящие более чем из 5 карт, стоят 1 балл за каждую карту, ранг которой превышает 5.

   Например, для приведенной выше строки программа выдаст следующий результат

   ```
   CLUBS          10   6    2
   DIAMONDS       A    Q   10   3
   HEARTS         K    J    7   5   3
   SPADES         A
   Баллы = 16
   ```

   Это объясняется тем, что в наборе содержатся два туза, один король, один валет, одна одиночка, дублетов нет и длинных мастей тоже нет. (Одиночный туз пик засчитывается дважды: как туз и как одиночка.)

   Факультативное задание: сделайте вашу программу как можно более гибкой. Попробуйте снять многочисленные ограничения на ввод данных.
5. Напишите программу, имитирующую работу калькулятора с очень длинными числами (длина которых намного превышает размер типа *long*). Этот калькулятор должен выполнять только операции сложения и умножения.

Строка ввода должна иметь такой вид

*num*1 *op* *num*2

и выводить на экран результат:

$$\begin{array}{r} num_1 \\ op\ num_2 \\ \hline num_3 \end{array}$$

Здесь num_1 и num_2 — это целые неотрицательные числа, символ *op* обозначает операцию + или *, а num_3 — это целое число, являющееся результатом вычислений.

Тщательно разработайте программу, используя перечисленные ниже компоненты.

- Структура данных для представления больших чисел, например, массив цифр, из которых состоит число.
- Функция для считывания числа. Незначащие нули следует игнорировать. Не забывайте, что нуль — это обычная цифра.
- Функция для записи числа. Незначащие нули записывать не следует, но все значащие нули должны быть выведены на печать, даже если число равно 0.
- Функция для сложения двух чисел.
- Функция для умножения двух чисел.

Кроме того, необходимо сделать следующее.

- Поверить переполнение (*чи*сла, количество цифр которых превышает константу `MAX_SIZE`).
- Разработать хороший пользовательский интерфейс.

Факультативное задание: учтите знаки чисел и напишите функцию для их вычитания.

Рекурсия: зеркала

В этой главе ...

Рекурсивные решения
- Рекурсивная функция, возвращающая значение: факториал числа n
- Рекурсивные функции, не возвращающие никаких значений. обратная запись строки

Перечислимые предметы
- Размножающиеся кролики (последовательность Фибоначчи)
- Организация парада
- Дилемма мистера Спока (выбор k из n предметов)

Поиск элемента в массиве
- Поиск наибольшего элемента в массиве
- Бинарный поиск
- Поиск k-го наименьшего элемента массива

Организация данных
- Ханойские башни

Рекурсия и эффективность

Резюме

Предупреждения

Вопросы для самопроверки

Упражнения

Задания по программированию

Введение. В главе описывается рекурсия — один из самых мощных методов решения задач. Предполагается, что большинство читателей практически ничего не знают о нем. Те, кто знаком с рекурсией, могут лишь бегло просмотреть изложенный материал.

Рекурсивный образ мышления демонстрируется на примере решения нескольких относительно простых задач, включая вычисления, поиск и организацию данных. Рекурсия рассматривается как с теоретической, так и практической точек зрения. Изучаются методы анализа рекурсивных процессов, позволяющие отслеживать и отлаживать рекурсивные функции.

Некоторые рекурсивные решения намного элегантнее и лаконичнее своих итеративных аналогов. Например, классическая задача о ханойских башнях довольно сложна, однако она имеет чрезвычайно простое рекурсивное решение. С другой стороны, некоторые рекурсивные функции крайне неэффективны и применять их в своих приложениях не следует.

Применение рекурсии для решения более сложных задач рассматривается в главе 5. Рекурсия играет важную роль во многих приложениях, рассмотренных в книге.

Рекурсивные решения

Рекурсия сводит задачу к решению более мелких идентичных задач

Рекурсия — чрезвычайно мощный метод решения задач. Часто задачи, которые, на первый взгляд, выглядят довольно сложными, имеют простое рекурсивное решение. Как и при проектировании "сверху вниз", рекурсия разбивает задачу на несколько более мелких задач. Особенность заключается в том, что эти небольшие задачи должны быть *совершенно идентичны* исходной задаче — так сказать, быть ее зеркальным отражением.

Если поставить одно перед другим два зеркала, вы увидите несколько своих зеркальных отражений, расположенных один за другим и уменьшающихся в размерах. Рекурсия напоминает эти зеркальные отражения. (Идеальным примером рекурсии являются матрешки. — *Прим. ред.*) Это означает, что рекурсия позволяет свести исходную задачу к решению ее более мелких копий. Последовательно уменьшая размеры этих задач, в конце концов процесс рекурсии приводит либо к очевидному, либо к уже известному решению, используя которое можно легко получить решение исходной задачи.

Допустим, например, что для решения задачи P_1 нужно решить задачу P_2, которая представляет собой уменьшенную копию задачи P_1. Кроме того, будем считать, что для решения задачи P_2 нужно решить задачу P_3, которая представляет собой уменьшенную копию задачи P_3. Зная решение задачи P_3 (предположим, что оно тривиально), можно решить задачу P_2. В свою очередь, используя найденное решение задачи P_2, легко решить исходную задачу P_1.

Некоторые рекурсивные решения неэффективны и непрактичны

На первых порах рекурсия производит сильное впечатление, но, как мы увидим далее, реальной альтернативой рекурсии является метод **итераций** (iteration), основанный на использовании циклов. Следует отдавать себе отчет, что не все рекурсивные решения лучше итеративных. Фактически некоторые рекурсивные решения не практичны, поскольку они не эффективны. И все же рекурсия позволяет находить простые и элегантные решения очень сложных задач.

Сложные задачи могут иметь простые рекурсивные решения

В качестве иллюстрации рассмотрим поиск слова в словаре. Допустим, нам нужно найти слово "vademecum". Будем считать, что поиск начинается с начала словаря, причем слова перебираются один за другим, пока не встретится искомое. Именно в этом и заключается **последовательный поиск** (sequential search). Очевидно, что этот способ решения задачи не слишком эффективен.

Ускорить процесс можно с помощью **бинарного поиска** (binary search). Он очень напоминает способ, которым обычно пользуются при работе со словарем. Словарь открывается приблизительно посередине, а затем читатель определяет, в какой половине словаря содержится искомое слово. Попробуем формализовать этот процесс в виде псевдокода.

Бинарный поиск слова в словаре

```
// Рекурсивный бинарный поиск в словаре
   if (словарь состоит из одной страницы)
      Ищем слово на этой странице
   else
   {
      Открываем словарь посередине
      Определяем, какая половина словаря содержит искомое слово
      if (слово содержится в первой половине)
         Выполняем поиск слова в первой половине словаря
      else
         Выполняем поиск слова во второй половине словаря
   }
```

Это решение все еще не вполне понятно. Как искать слово на странице? Как найти середину словаря? Если середина найдена, как определить, в какой половине содержится искомое слово? Ответы на эти вопросы не сложны, но попытка на них ответить уведет нас далеко в сторону.

Базовая задача имеет заранее известное решение

Стратегия поиска слова, примененная выше, позволяет свести задачу к поиску слова лишь в половине словаря, как показано на рис. 2.1. Отметим два важных момента. Во-первых, разделив словарь пополам, мы уменьшили сложность задачи вдвое: для поиска в половине словаря можно применять ту же стратегию, что и для всего словаря в целом. Во-вторых, один их пунктов решения отличается от всех остальных: последовательно деля части словаря пополам, мы обнаружим страницу, на которой содержится искомое слово. В этот момент задача становится достаточной простой, поэтому можно применять простой перебор оставшихся слов. Эта задача называется **базовой** (base case), или **базисом** (basis), или **вырожденной задачей** (degenerative case).

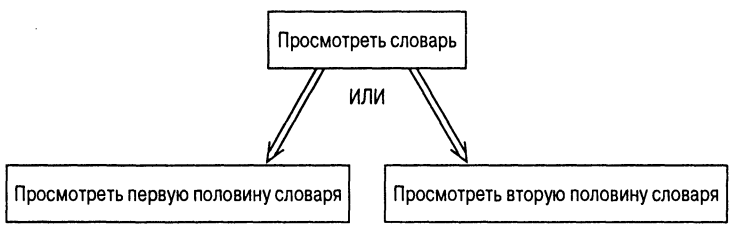


Рис. 2.1. Рекурсивное решение задачи о поиске слова в словаре

Бинарный поиск основан на принципе "разделяй и властвуй"

Описанная стратегия основана на принципе **"разделяй и властвуй"** (divide and conqeuer). Сначала словарь разделяется на две половины, а затем каждая из них обрабатывается исходным алгоритмом. Повторяя этот процесс неоднократно, можно, в конце концов, прийти к базовой задаче. Как мы увидим в дальнейшем, эта стратегия присуща многим рекурсивным алгоритмам.

Рассмотрим более строгую формулировку описанного выше решения задачи.

```
search(in aDictionary:Dictionary, in word:string)

   if (словарь aDictionary состоит из одной страницы)
      Ищем слово на этой странице
   else
   {
      Открываем словарь aDictionary посередине
      Определяем, какая половина словаря содержит искомое слово
      if (слово word содержится в первой половине aDictionary)
         search(первая половина словаря aDictionary, word)
      else
         search(вторая половина словаря aDictionary, word)
    }
```

Запишем процесс решения задачи в виде функции, обратив внимание на следующие особенности.

1. Функция вызывает саму себя, т.е функция *search* вызывает функцию *search*. Именно это свойство делает функцию рекурсивной. Стратегия решения задачи заключается в последовательном делении словаря *aDictionary* пополам, определении, какая из половин содержит слово *word*, и применении той же самой стратегии к соответствующей половине.

   Рекурсивная функция вызывает саму себя

2. Каждый вызов функции *search*, выполненный внутри функции *search*, уменьшает размер словаря вдвое. Таким образом, функция *search* решает идентичные задачи, размер которых постоянно уменьшается вдвое.

   Каждый рекурсивный вызов решает идентичную задачу меньшего размера

3. Одна из подзадач решается иначе, чем другие. Когда размер словаря *aDictionary* сокращается до одной страницы, используется другой метод: прямой перебор слов, содержащихся на странице. Поиск слова на странице является базисом исходной задачи. На этом этапе рекурсивные вызовы функции прекращаются, и задача решается с помощью простого перебора.

   Проверка базисных условий позволяет остановить процесс рекурсии

4. Способ, применяемый для последовательного уменьшения размера задач, гарантирует достижение базиса.

   В итоге одна из задач должна оказаться базовой

Эти свойства характерны для любого рекурсивного решения. Хотя не все рекурсивные методы в точности соответствуют описанным выше принципам, совпадений все же больше, чем отличий. Разрабатывая рекурсивное решение следует учитывать следующие четыре вопроса.

ОСНОВНЫЕ ПОНЯТИЯ

Четыре вопроса о рекурсивном решении

1. Как свести исходную задачу к идентичным задачам меньшего размера?
2. Как уменьшать размер задачи при каждом рекурсивном вызове?
3. Какая задача является базовой?
4. Можно ли достичь базиса, постоянно уменьшая размер исходной задачи?

Рассмотрим две относительно простые задачи: вычисление факториала и обратную запись строки. Решения этих задач иллюстрируют различия между рекурсивными функциями, возвращающими некоторые значения, и рекурсивными функциями, не возвращающими никаких значений (*void* functions).

Рекурсивная функция, возвращающая значение: факториал числа n

Рассмотрим рекурсивное вычисление факториала целого числа *n*. Эта задача является хорошей иллюстрацией, поскольку ее легко понять и она хорошо укладывается в общую схему рекурсии, описанную выше. Однако, поскольку эта задача имеет простое и эффективное итеративное решение, на практике рекурсивное вычисление факториала не применяется.

> Не применяйте рекурсию для решения задач, имеющих простое и эффективное итеративное решение

Для начала рассмотрим итеративное определение функции *factorial*(*n*) (более широко используется обозначение n!)

factorial(*n*) = *n**(*n*-1) *(*n*-2)... *1
для любого целого числа *n*>0
factorial(0) = 1

> Итеративное определение факториала

Факториал для отрицательных чисел не определен. Основываясь на этом определении, легко написать итеративную функцию, вычисляющую факториал.

Для рекурсивного определения функции *factorial*(*n*) нужно выразить ее через факториал меньшего числа. Для этого достаточно учесть, что факториал числа *n* равен факториалу числа *n*–1, умноженному на число *n*. Таким образом, приходим к следующему определению.

factorial(*n*) = *n**[(*n*-1) *(*n*-2)... *1]
= *n***factorial*(*n*-1)

> Рекуррентное отношение

Определение факториала числа *n* через факториал числа *n*–1 представляет собой пример **рекуррентного отношения** (recurrent relation). Отсюда следует, что факториал числа *n*–1 можно выразить через факториал числа *n*–2 и т.д. Этот процесс аналогичен поиску слова в словаре, описанному выше.

В определении факториала не хватает ключевой детали: базовой задачи. Как и прежде, нам необходимо сформулировать задачу, которая отличается от других, иначе процесс рекурсии никогда не закончится. Базисом функции, вычисляющей факториал, является значение *factorial*(0), равное 1. Поскольку исходное значение *n* больше или равно нулю и каждый вызов функции *factorial* уменьшает его на 1, мы всегда можем свести исходную задачу к базовой.

Таким образом, полное рекурсивное определение факториала выглядит следующим образом.

> Рекурсивное определение факториала

$$factorial(n) = \begin{cases} 1, \text{если } n = 0, \\ n * factorial(n-1), \text{если } n > 0. \end{cases}$$

Применим это определение для вычисления *factorial*(4). Поскольку 4 > 0, из рекурсивного определения следует, что

factorial(4) = 4 * *factorial*(3).

Аналогично,

factorial(3) = 3 * *factorial*(2),
factorial(2) = 2 * *factorial*(1),
factorial(1) = 1 * *factorial*(0).

Мы достигли базиса, и, по определению,

factorial(0) = 1.

На этом рекурсия заканчивается, а мы все еще не знаем, чему равно значение *factorial*(4). Теперь нужно выполнить обратный ход:

- поскольку *factorial*(0) = 1, *factorial*(1) = 1 * 1 = 1,
- поскольку *factorial*(1) = 1, *factorial*(2) = 2 * 1 = 2,
- поскольку *factorial*(2) = 2, *factorial*(3) = 3 * 2 = 6,
- поскольку *factorial*(3) = 6, *factorial*(4) = 4 * 6 = 24.

Рекурсия — это процесс разбиения исходной задачи на подзадачи, которые можно решать с другом. Например, если вам нужно вычислить значение *factorial*(4), сначала следует проверить, нельзя ли получить ответ сразу. Немедленно решается лишь базовая задача (*factorial*(0) = 1), но это не позволяет непосредственно вычислить значение *factorial*(4). Однако если ваш друг уже вычислил значение *factorial*(3), то значение *factorial*(4) можно вычислить, умножив число *factorial*(3) на 4. Таким образом, вам остается лишь выполнить операцию умножения, а ваш друг должен вычислить значение *factorial*(3).

Теперь уже ваш друг должен применить для вычисления числа *factorial*(3) тот же способ, которым вы пользовались при вычислении значения *factorial*(4). Ваш друг придет к выводу, что задача вычисления значения *factorial*(3) не является базовой, и попросит другого приятеля вычислить значение *factorial*(2). Зная это значение, ваш друг сможет вычислить значение *factorial*(3), а вы, в свою очередь, получив от него этот число, сможете вычислить значение *factorial*(4).

Обратите внимание, что рекурсивное вычисление значения *factorial*(4) приводит к тому же ответу, что и итеративное вычисление 4 * 3 * 2 * 1 = 24. Для доказательства, что эти определения факториала эквивалентны, используется **математическая индукция** (Приложение Г). Тесная взаимосвязь рекурсии и математической индукции обсуждается в главе 5.

Рекурсивное определение функции, вычисляющей факториал, иллюстрирует две особенности: 1) значение *factorial*(*n*) можно определить *интуитивно*, через значение *factorial*(*n-1*); 2) значение *factorial*(*n*) можно вычислить *механически*, последовательно применяя определение факториала. Даже в этом простом примере для применения рекурсивного определения пришлось выполнить много работы, которую можно было бы поручить компьютеру.

Рекурсивное определение функции, вычисляющей факториал, легко реализовать на языке C++.

```
int fact(int n)
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вычисляет факториал неотрицательного целого числа.
// Предусловие: число n должно быть неотрицательным.
// Постусловие: возвращает факториал числа n;
// само число n не изменяется.
{
   if ( n == 0)
      return 1;
   else
      return n * fact(n-1);
} // Конец функции fact
```

На рис. 2.2 показан процесс вычисления значения *fact(3)*, если в программе используется следующий оператор.

```
cout << fact(3);
```

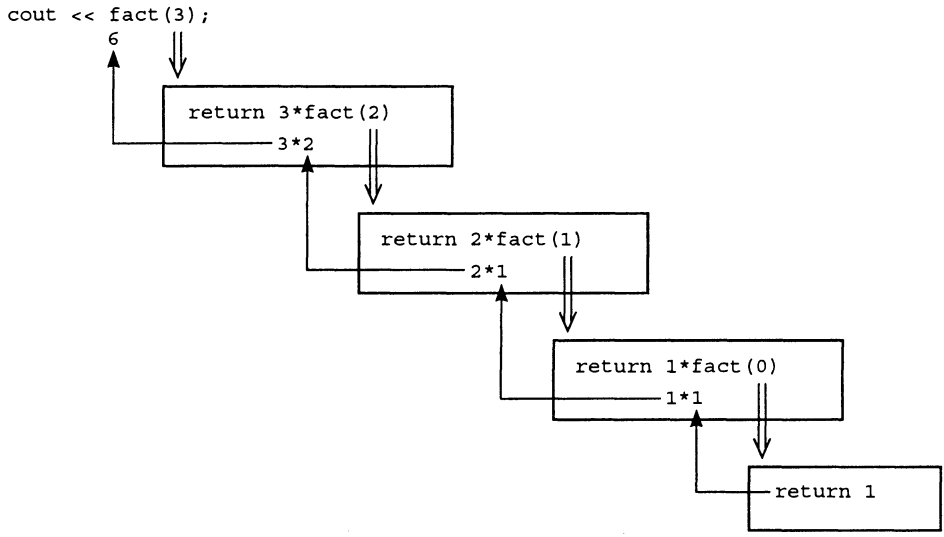


Рис. 2.2. Вычисление значения fact(3)

Эта функция полностью соответствует четырем критериям рекурсивного решения, сформулированным ранее.

Функция fact соответствует четырем критериям рекурсивного решения

1. Функция `fact` вызывает саму себя.
2. При каждом рекурсивном вызове функции `fact` значение ее аргумента *уменьшается* на 1.
3. Факториал нуля функция вычисляет иначе, чем остальные факториалы. Для этого она не генерирует рекурсивный вызов. Вместо этого она использует заранее известный факт, что значение `fact(0)` равно 1. Таким образом, базис рекурсии достигается, когда значение `n` становится равным 0.
4. Учитывая, что значение `n` неотрицательно, п. 2 гарантирует, что в процессе вычислений *базис обязательно достигается*.

Интуитивно ясно, что функция `fact` реализует рекурсивное определение факториала. Рассмотрим теперь механизм выполнения рекурсивной функции. Его логика проста, за исключением, возможно, условного выражения из раздела `else`. Это выражение имеет следующие последствия.

1. Вычисляется каждый операнд вида `n * fact(n-1)`.
2. Второй операнд — `fact(n-1)` — вычисляется с помощью вызова функции `fact`. Хотя этот вызов является рекурсивным (функция `fact` вызывает саму себя), в нем нет ничего особенного. Мысленно подставьте на место рекурсивного вызова не функцию `fact`, а другую функцию, например `abs`. Принцип его действия тот же самый: просто вычисляется значение функции.

Теоретически вычисление рекурсивной функции не должно составлять особого труда. Однако на практике она быстро выходит из-под контроля. Для систематического исследования анализа рекурсивных функций используется **метод блок-схем** (box method). Этот метод можно применять как для изучения самой рекурсии, так и для отладки рекурсивных функций. И все же такой механический подход к рекурсии не может заменить собой ее интуитивное понимание. Метод блок-схем иллюстрирует типичную реализацию рекурсии с помощью компиляторов.

Анализируя описание метода блок-схем, приведенное ниже, обратите внимание, что каждый блок соответствует отдельной **записи активации** (activation record), которая обычно применяется компиляторами для реализации вызова функции. Более подробно этот вопрос обсуждается в главе 6.

Для каждого вызова функции создается запись активации

Метод блок-схем. Проиллюстрируем этот метод на примере функции *fact*. Как мы увидим в следующем разделе, для функций, не возвращающих никаких значений (*void*-функции), этот метод упрощается.

1. Пометим каждый рекурсивный вызов в теле рекурсивной функции. В отдельной функции могут встретиться несколько рекурсивных вызовов, поэтому важно уметь отличать их друг от друга. Эти метки позволяют правильно определить место, в которое мы должны вернуться после выполнения вызова функции. Например, пометим выражение *fact(n-1)* внутри тела функции буквой A.

Пометим каждый рекурсивный вызов в теле функции

```
if (n == 0)
   return 1;
else
   return n * fact(n-1);
                   A
```

   Мы будем возвращаться в точку A после каждого рекурсивного вызова, подставлять вычисленное значение *fact(n-1)* и продолжать выполнение программы, вычисляя выражение *n * fact(n-1)*.

2. Каждый вызов функции на протяжении ее выполнения будем представлять в виде нового блока, в котором описывается **локальное окружение** (local environment) функции. Точнее говоря, каждый блок содержит следующие элементы.

Каждый раз при вызове функции новый блок описывает ее локальное окружение

   2.1. Формальные аргументы функции, передаваемые по значению.

   2.2. Значения локальных переменных функции.

   2.3. Ячейку, в которой хранится значение, возвращаемое при каждом рекурсивном вызове из текущего блока. Метка этой ячейки должна соответствовать метке, указанной в п. 1.

   2.4. Значение самой функции.

   Когда блок создается впервые, нам известны лишь значения входных аргументов. Значения остальных элементов уточняются по мере вычисления функции. Например, можно создать блок для вызова *fact(3)*, изображенный на рис. 2.3. (В следующих примерах мы увидим, что аргументы, передаваемые по ссылке, должны обрабатываться иначе, в отличие от аргументов, передаваемых по значению, и локальных переменных.)

3. Нарисуем стрелку от оператора, инициировавшего рекурсивный процесс, в первый блок. Затем, создав новый блок после рекурсивного вызова, как указано в п. 2, нарисуем стрелку из блока, выполнявшего вызов, во вновь созданный блок. Пометим каждую стрелку соответствующей меткой рекурсивного вызова (из п. 1). Эта метка точно указывает место, в которое мы вернемся после выполнения очередного рекурсивного вызова. Например, на рис. 2.4 показаны первые два блока, порожденные вызовом функции *fact* в операторе *cout<<fact(3)*.

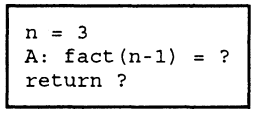


Рис. 2.3. Блок для вызова fact(3)

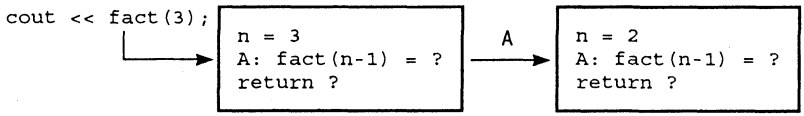
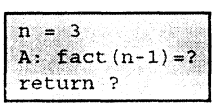


Рис. 2.4. Начало выполнения метода блок-схем

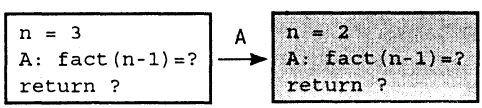
4. После создания нового блока и стрелок, описанных в пп. 2 и 3, начинается выполнение тела функции. Каждая ссылка на элемент локального окружения соответствует определенному значению в текущем блоке, независимо от того, каким образом сгенерирован этот блок.
5. При выходе из функции текущий блок вычеркивается, и управление перемещается по стрелке в блок, вызвавший рекурсивную функцию. Этот блок становится текущим, а метка соответствующей стрелки точно указывает место, с которого должно продолжаться выполнение функции. Подставляем значение, возвращенное только что выполненной функцией, в соответствующий элемент текущего блока.

На рис. 2.5 показана полная трассировка вызова `fact(3)` с помощью метода блок-схем. В диаграммах, изображенных на этом рисунке, текущим всегда является блок, последним встречающийся на пути, указанном стрелками. Текущие блоки закрашиваются, а вычеркиваемые — выделяются пунктиром.

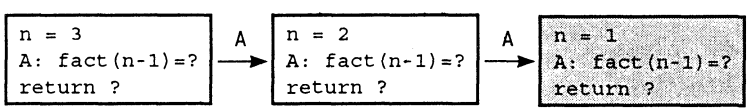
Выполнен первоначальный вызов, начинается выполнение метода `fact`:



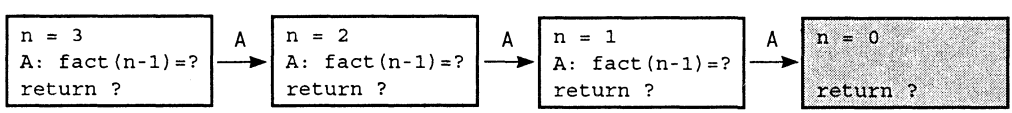
В точке A выполнен рекурсивный вызов, начинается новое выполнение метода `fact`:



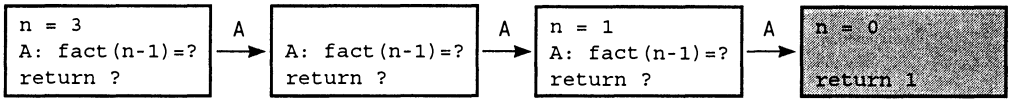
В точке A выполнен рекурсивный вызов, начинается новое выполнение метода `fact`:



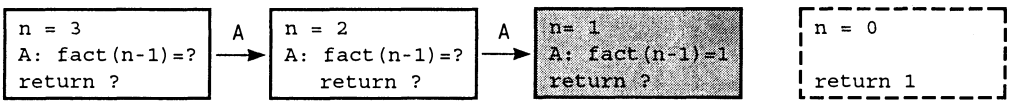
В точке A выполнен рекурсивный вызов, начинается новое выполнение метода `fact`:



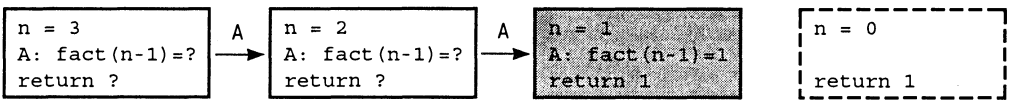
Это — базовая задача, выполнение функции `fact` завершается:



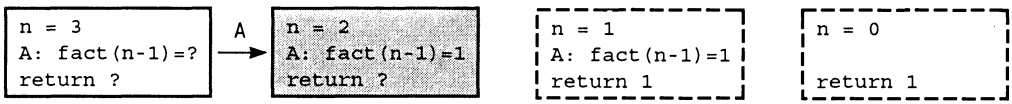
Метод возвращает вычисленное значение в вызывающий блок, который продолжает работу:



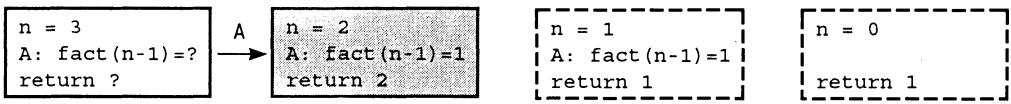
Текущее выполнение функции `fact` завершено:



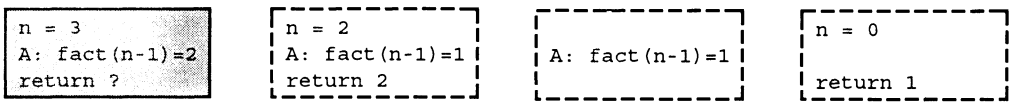
Метод возвращает вычисленное значение в вызывающий блок, который продолжает работу:



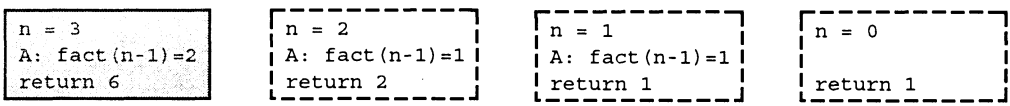
Текущее выполнение функции `fact` завершено:



Метод возвращает вычисленное значение в вызывающий блок, который продолжает работу:



Текущее выполнение функции `fact` завершено:



Результатом исходного вызова является число 6.

Рис. 2.5. Блоки, возникающие при трассировке вызова fact(3)

Инварианты. Инварианты рекурсивных функций имеют не меньшую важность, чем инварианты итеративных функций, причем часто они проще своих итеративных аналогов. Рассмотрим, например, рекурсивную функцию *fact*.

```
int fact(int n)
// Предусловие: число n должно быть неотрицательным.
// Постусловие: возвращает факториал числа n.
{
   if (n == 0)
       return 1;
   else  // Инвариант: n>0, поэтому n-1>=0;
         // Следовательно, вызов fact(n-1) возвращает (n-1)!
       return n * fact(n-1); // n * (n-1)! = n!
} // Конец функции fact
```

Если предусловие выполняется, постусловие рекурсивного вызова также должно выполняться

Предусловие функции требует, чтобы значение аргумента *n* было неотрицательным. В момент рекурсивного вызова *fact(n-1)* аргумент *n* положителен, поэтому число *n-1* является неотрицательным. Поскольку рекурсивный вызов удовлетворяет предусловию функции *fact*, следует ожидать, что вызов *fact(n-1)* вернет факториал числа *n-1*. Следовательно, число *n * fact(n-1)!* является факториалом числа *n*. Для формального доказательства, что вызов *fact(n)* возвращает факториал числа *n* в главе 5 применяется метод математической индукции.

Нарушение предусловия функции fact приводит к бесконечной рекурсии

Если предусловие функции *fact* нарушается, функция может работать неправильно. Это означает, что если вызывающий модуль передаст функции отрицательное значение, возникнет бесконечная цепочка рекурсивных вызовов, которая прервется, лишь полностью исчерпав системные ресурсы, поскольку функция не сможет достичь базиса рекурсии. Например, вызов *fact(-4)* может сгенерировать вызов *fact(-5)*, тот в свою очередь — вызов *fact(-6)* и так до бесконечности.

В идеале функция должна предотвращать такие ситуации, проверяя знак аргумента. Если *n<0*, функция может вернуть либо 0, либо признак ошибки. Методы предотвращения ошибок обсуждаются в главе 1 в разделах "Надежное программирование" и "Стиль".

Рекурсивные функции, не возвращающие никаких значений: обратная запись строки

Рассмотрим теперь более сложную задачу. Дана строка символов, требуется записать ее в обратном порядке. Например, строку "cat" нужно записать в виде "tac". Для создания рекурсивного решения нужно вспомнить четыре вопроса, сформулированных в подразделе "Основные понятия" на стр. 72.

Обратную запись строки, состоящей из *n* символов можно свести к идентичной задаче для строки, состоящей из *n*-1 символов. Таким образом, мы сможем на каждом рекурсивном шаге уменьшать длину строки на 1. Последовательное уменьшение длины строки должно закончиться на базовой задаче, в которой длина строки становится настолько малой, что записать ее в обратном порядке не составляет никакого труда. Примером очень короткой строки является пустая строка, длина которой равна 0. Следовательно, в качестве базовой можно выбрать следующую задачу.

Записать пустую строку в обратном порядке

Базовая задача

Для решения этой задачи ничего не нужно делать — проще не бывает! (Как альтернативу можно использовать в качестве базиса строку, состоящую из одного символа.)

Как записать в обратном порядке строку, состоящую из *n* символов, если эта задача уже решена для строки, длина которой равна *n*-1? Ситуация аналогична вычислению факториала числа *n*, когда известен факториал числа *n*-1. Однако в данном случае все немного сложнее.

Как записать в обратном порядке строку, состоящую из n символов, если известно, как это можно сделать для строки, состоящей из n-1 символа

Очевидно, нам подходит не всякая строка, имеющая длину *n*-1. Например, обратная запись строки "груша" (длина строки равна 5) не имеет ничего общего с обратной записью строки "дыня" (длина строки равна 4). Задача меньшего размера должна точно подходить для решения исходной задачи.

Искомая строка, состоящая из *n*-1 символа, должна быть подстрокой (частью) исходной строки. Допустим, что мы отбросили один символ исходной строки, образовав подстроку, имеющую длину *n*-1. Чтобы рекурсивное решение было правильным, обратная запись последовательно уменьшающихся строк в сочетании с некоей вспомогательной операцией должна вести к решению исходной задачи. Сравним этот подход с рекурсивным вычислением факториала: вычисление факториала числа *n*-1 в сочетании с умножением на число *n* позволяло найти факториал числа *n*.

Нужно решить, какой символ следует отбросить и что считать вспомогательной операцией. Сначала рассмотрим второй вопрос. Поскольку задача сводится к записи символов, в качестве вспомогательной можно рассматривать операцию записи одного символа. При выборе отбрасываемых символов есть несколько вариантов.

Например,

Отбросить последний символ

или

Отбросить первый символ

Рассмотрим первую из этих альтернатив, отбрасывая первый символ. Эта ситуация проиллюстрирована на рис. 2.6.

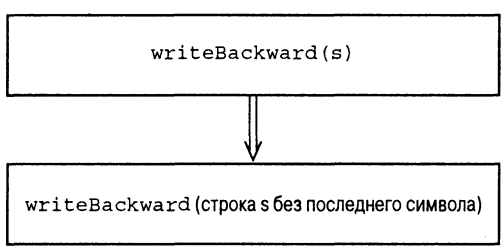


Рис. 2.6. Рекурсивное решение задачи об обратной записи строки

Для того чтобы решение было правильным, последний символ строки должен оказаться первым. Следовательно, последний символ нужно записать перед остальной частью строки, записанной в обратном порядке.

```
writeBackward(in s:string)

   if (строка пуста)
      Ничего не делаем - - это базовая задача
   else
   {
      Записываем последний символ строки s
      writeBackward(строка s без последнего символа)
   }
```

Функция writeBackward записывает строку в обратном порядке

Это концептуальное решение задачи. Чтобы воплотить его на языке C++, нужно решить несколько вопросов, связанных с реализацией. Допустим, функция будет получать два аргумента: строку *s*, которую следует записать в обратном порядке, и целое число *size*, задающее длину этой строки. Для простоты будем предполагать, что строка начинается с позиции 0 и заканчивается позицией *size-1*. Это означает, что все символы, включая пробелы, находящиеся в данном диапазоне индексов, являются частью строки *s*. Функция *writeBackward* на языке C++ принимает следующий вид.

```
void writeBackward(string s, int size)
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Записывает строку символов в обратном порядке.
// Предусловие: строка s содержит size символов, size >= 0.
// Постусловие: строка s записана в обратном порядке,
// оставаясь неизменной.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   if (size > 0)
   {
      // Записываем последний символ
      cout << s.substr(size-1, 1);

      // Записать оставшуюся часть строки в обратном порядке
      writeBackwards(s, size-1); // Точка A
   } // Конец оператора if

   // Вариант s == 0 является базисом - не делаем ничего
} // Конец функции writeBackward
```

Обратите внимание, что рекурсивные вызовы функции *writeBackward* используют последовательно уменьшающиеся значения переменной *size*. Это гарантирует, что при каждом вызове последний символ строки будет отброшен, и базовая задача будет достигнута.

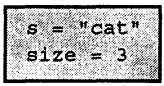
Выполнение функции *writeBackward* можно отследить с помощью метода блок-схем. Как и для функции *fact*, каждый блок содержит локальное окружение рекурсивного вызова — в данном случае входные аргументы *s* и *size*. Процесс трассировки немного отличается от трассировки функции *fact*, проиллюстрированной на рис. 2.5, поскольку функция *writeBackward* ничего не возвращает и, следовательно, не использует оператор *return*. Рис. 2.7 иллюстрирует трассировку вызова функции *writeBackward* для строки "cat".

Функция writeBackward ничего не возвращает

Теперь рассмотрим несколько иной подход к решению поставленной задачи. Напомним, что у нас был выбор: отбрасывать первый или последний символ строки. Выше описано решение, основанное на отбрасывании последнего символа. Интересно проследить за решением задачи, когда отбрасывается первый символ.

Отбросить первый символ

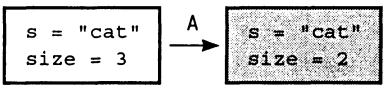
Выполнен первоначальный вызов, начинается выполнение функции:



Выходная строка: `t`

Достигнута точка A `(writeBackward(s, size-1))` выполняется реурсивный вызов.

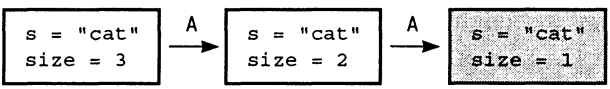
Начинается новое выполнение функции:



Выходная строка: `ta`

Достигнута точка A, выполняется рекурсивный вызов.

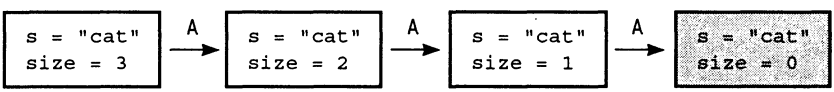
Начинается новое выполнение функции:



Выходная строка: `tac`

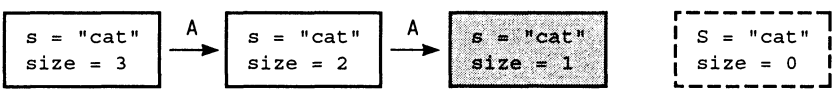
Достигнута точка A, выполняется рекурсивный вызов.

Начинается новое выполнение функции:



Это — базовая задача, поэтому выполнение функции завершается.

Управление возвращается в вызывающий модуль, который продолжает работу:



Это выполнение функции завершено. Управление возвращается в вызывающий модуль, который продолжает работу:



Это выполнение функции завершено. Управление возвращается в вызывающий модуль, который продолжает работу:



Это выполнение функции завершено. Управление возвращается в оператору, следующему за первоначальным вызовом.

Рис. 2.7. Трассировка вызова writeBackward("cat", 3)

Для начала рассмотрим простую модификацию предыдущего псевдокода, заменив слово "последний" словом "первый". Таким образом, функция должна записывать первый, а не последний символ, а затем рекурсивно записывать оставшуюся часть строки в обратном порядке.

```
writeBackward1(in s:string)

   if (строка пуста)
      Ничего не делаем - - это базовая задача
   else
   {
      Записываем первый символ строки s
      writeBackward(строка s без первого символа)
   }
```

Приводит ли это решение к правильному ответу? Немного подумав, легко понять, что эта функция записывает строку в прямом порядке слева направо, а вовсе не в обратном. Кроме того, в псевдокоде выполняются следующие шаги.

```
Записать первый символ строки s
Записать остальную часть строки s
```

Эти шаги просто записывают строку *s*, как обычно. Имя функции *writeBackward* вводит нас в заблуждение — чудес не бывает!

Для записи строки в обратном порядке нужно выполнить следующие рекурсивные операции.

```
Записать строку s без первого символа в обратном порядке
Записать первый символ строки s
```

Иными словами, первый символ сроки *s* нужно записывать только *после* того, как остальная часть строки будет записана в обратном порядке. Таким образом, правильное решение выглядит так.

```
writeBackward2(in s:string)

   if (строка пуста)
      Ничего не делаем - - это базовая задача
   else
   {
      writeBackward2(строка s без первого символа)
      Записываем первый символ строки s
   }
```

Перевод функции *writeBackward2* на язык C++ выполняется аналогично функции *writeBackward*. Это упражнение читатели могут выполнить самостоятельно.

Поучительно проследить, как выполняются псевдокоды функций *writeBackward* и *writeBackward2*. Во-первых, добавим в каждую функцию операторы вывода, позволяющие осуществить трассировку.

Операторы cout позволяют проверить логику рекурсивной функции

```
writeBackward(in s:string)

   cout << "Вход в функцию
writeBackward со строкой: " << s << endl;
   if (строка пуста)
      Ничего не делаем - - это базовая задача
   else
   {
      cout << Запись последнего символа строки:" << s << endl;
```

```
        Записываем последний символ строки s
        writeBackward(строка s без последнего символа) // Точка A
    }
    cout << "Выход из функции writeBackward со строкой: " << s <<
endl;

writeBackward2(in s:string)

cout << "Вход в функцию writeBackward2 со строкой: " << s << endl;

    if (строка пуста)
        Ничего не делаем - - это базовая задача
    else
    {
        writeBackward2(строка s без первого символа)
        cout << Запись первого символа строки:" << s << endl;
        Записываем первый символ строки s
    }
    cout << "Выход из функции writeBackward со строкой: " << s <<
endl;
```

На рис. 2.8 и 2.9 показана информация, которую выводят псевдокоды функций *writeBackward* и *writeBackward2* для строки "cat".

Разница между этими двумя функциями должна быть очевидной. Рекурсивные вызовы, выполняемые этими функциями, генерируют разные последовательности значений аргумента *s*. Несмотря на этот факт, обе функции правильно записывают строку в обратном порядке. Разница между ними компенсируется разным выбором записываемых символов и разными моментами рекурсивного вызова. В терминах блок-схем, изображенных на рис. 2.8 и 2.9, можно сказать, что функция *writeBackward* записывает символы непосредственно перед генерацией следующего блока (перед следующим рекурсивным вызовом), в то время как функция *writeBackward2* записывает символы сразу после вычеркивания блока (сразу после возвращения из рекурсивного вызова). Если собрать эти изменения вместе, можно прийти к выводу, что обе функции придерживаются разных стратегий решения одной и той же задачи.

Эти примеры демонстрируют важность, которую имеет метод блок-схем в сочетании с промежуточными операторами вывода для отладки рекурсивных функций. Операторы *cout* в начале, в середине и в конце функции выводят на печать значение аргумента *s*. При отладке рекурсивных функций нужно контролировать и значения всех локальных переменных, и точки рекурсивных вызовов, как показано в следующем примере.

Временные операторы вывода позволяют отлаживать рекурсивные функции

```
abc(...)

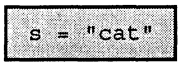
    cout << "Вызов функции abc из точки A.\n";
    abc(...) // Точка A

    cout << "Вызов функции abc из точки B.\n";
    abc(...) // Точка B
```

Операторы *cout* в окончательном варианте функции оставаться не должны.

После отладки временные операторы вывода следует удалить из рекурсивной функции

Выполнен первоначальный вызов, начинается выполнение функции:



Выходной поток:

```
Вход в функцию writeBackward со строкой: cat
Запись последнего символа строки: cat
t
```

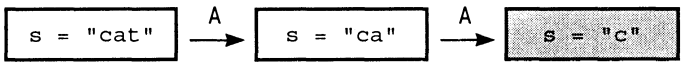
Достигнута точка A, выполняется рекурсивный вызов. Начинается новое выполнение функции:



Выходной поток:

```
Вход в функцию writeBackward со строкой: cat
Запись последнего символа строки: cat
t
Вход в функцию writeBackward со строкой: ca
Запись последнего символа строки: ca
a
```

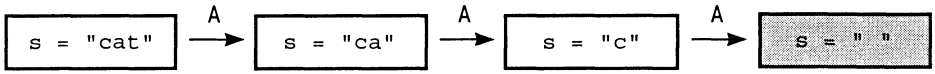
Достигнута точка A, выполняется рекурсивный вызов. Начинается новое выполнение функции:



Выходной поток:

```
Вход в функцию writeBackward со строкой: cat
Запись последнего символа строки: cat
t
Вход в функцию writeBackward со строкой: ca
Запись последнего символа строки: ca
a
Вход в функцию writeBackward со строкой: c
Запись последнего символа строки: c
c
```

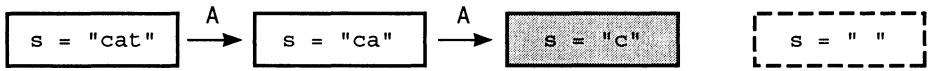
Достигнута точка A, выполняется рекурсивный вызов. Начинается новое выполнение функции:



Это выполнение функции завершено. Управление возвращается в вызывающий модуль.

Выходной поток:

```
Вход в функцию writeBackward со строкой: cat
Запись последнего символа строки: cat
t
Вход в функцию writeBackward со строкой: ca
Запись последнего символа строки: ca
a
Вход в функцию writeBackward со строкой: c
Запись последнего символа строки: c
c
Вход в функцию writeBackward со строкой:
Выход из функции writeBackward со строкой:
```



Это выполнение функции завершено. Управление возвращается в вызывающий модуль.

Выходной поток:

```
Вход в функцию writeBackward со строкой: cat
Запись последнего символа строки: cat
t
Вход в функцию writeBackward со строкой: ca
Запись последнего символа строки: ca
a
Вход в функцию writeBackward со строкой: c
Запись последнего символа строки: c
c
Вход в функцию writeBackward со строкой:
Выход из функции writeBackward со строкой:
Выход из функции writeBackward со строкой: c
```



Это выполнение функции завершено. Управление возвращается в вызывающий модуль.

Выходной поток:

```
Вход в функцию writeBackward со строкой: cat
Запись последнего символа строки: cat
t
Вход в функцию writeBackward со строкой: ca
Запись последнего символа строки: ca
a
Вход в функцию writeBackward со строкой: c
Запись последнего символа строки: c
c
Вход в функцию writeBackward со строкой:
Выход из функции writeBackward со строкой:
Выход из функции writeBackward со строкой: c
Выход из функции writeBackward со строкой: ca
```



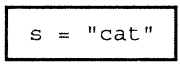
Это выполнение функции завершено. Управление возвращается в вызывающий модуль.

Выходной поток:

```
Вход в функцию writeBackward со строкой: cat
Запись последнего символа строки: cat
t
Вход в функцию writeBackward со строкой: ca
Запись последнего символа строки: ca
a
Вход в функцию writeBackward со строкой: c
Запись последнего символа строки: c
c
Вход в функцию writeBackward со строкой:
Выход из функции writeBackward со строкой:
Выход из функции writeBackward со строкой: c
Выход из функции writeBackward со строкой: ca
Выход из функции writeBackward со строкой: cat
```

Рис. 2.8. Трассировка вызова writeBackward("cat", 3) в псевдокоде

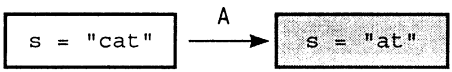
Выполнен первоначальный вызов, начинается выполнение функции:



Выходной поток:

```
Вход в функцию writeBackward2 со строкой: cat
```

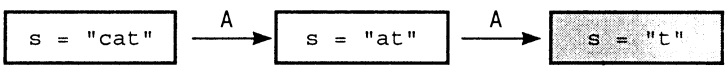
Достигнута точка A, выполняется рекурсивный вызов. Начинается новое выполнение функции:



Выходной поток:

```
Вход в функцию writeBackward2 со строкой: cat
Вход в функцию writeBackward2 со строкой: at
```

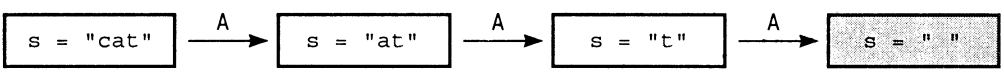
Достигнута точка A, выполняется рекурсивный вызов. Начинается новое выполнение функции:



Выходной поток:

```
Вход в функцию writeBackward2 со строкой: cat
Вход в функцию writeBackward2 со строкой: at
Вход в функцию writeBackward2 со строкой: t
```

Достигнута точка A, выполняется рекурсивный вызов. Начинается новое выполнение функции:



Это выполнение функции завершено. Управление возвращается в вызывающий модуль.

Выходной поток:

```
Вход в функцию writeBackward2 со строкой: cat
Вход в функцию writeBackward2 со строкой: at
Вход в функцию writeBackward2 со строкой: t
Вход в функцию writeBackward2 со строкой:
Выход из функции writeBackward2 со строкой:
Запись первого символа строки: t
t
Выход из функции writeBackward2 со строкой: t
```



Это выполнение функции завершено. Управление возвращается в вызывающий модуль.

Выходной поток:

```
Вход в функцию writeBackward2 со строкой: cat
Вход в функцию writeBackward2 со строкой: at
Вход в функцию writeBackward2 со строкой: t
Вход в функцию writeBackward2 со строкой:
Выход из функции writeBackward2 со строкой:
Запись первого символа строки: t
t
Выход из функции writeBackward2 со строкой: t
Запись первого символа строки: at
a
Выход из функции writeBackward2 со строкой: at
```



Это выполнение функции завершено. Управление возвращается в вызывающий модуль.

Выходной поток:

```
Вход в функцию writeBackward2 со строкой: cat
Вход в функцию writeBackward2 со строкой: at
Вход в функцию writeBackward2 со строкой: t
Вход в функцию writeBackward2 со строкой:
Выход из функции writeBackward2 со строкой:
Запись первого символа строки: t
t
Выход из функции writeBackward2 со строкой: t
Запись первого символа строки: at
a
Выход из функции writeBackward2 со строкой: at
Запись первого символа строки: cat
c
Выход из функции writeBackward2 со строкой: cat
```

Рис. 2.9. Трассировка вызова writeBackward2("cat", 3) в псевдокоде

Перечислимые предметы

Для решения следующих трех задач необходимо подсчитывать события или комбинации событий и предметов. Эти задачи представляют собой яркие примеры, имеющие несколько базисов. Кроме того, они приводят к потрясающе неэффективным рекурсивным решениям. Однако пусть вас это не смущает. Наша цель — освоить рекурсию, разобрав эти примеры. Вскоре мы увидим и полезные, и эффективные рекурсивные решения.

Размножающиеся кролики (последовательность Фибоначчи)

Кролики — очень плодовитые животные. Если бы они никогда не умирали, их популяция быстро вышла бы из-под контроля. Сделаем следующие предположения, относящиеся к случайно выбранным кроликам.

- Кролики бессмертны.
- Кролик достигает половой зрелости через два месяца после своего рождения, т.е. к началу третьего месяца жизни.
- Кролики всегда рождаются парами "мальчик-девочка". В начале каждого месяца каждая половозрелая пара дает жизнь только одной паре.

Допустим, что вначале у нас есть только одна пара кроликов. Сколько пар кроликов у нас будет через шесть месяцев, считая тех кроликов, которые родятся в начале шестого месяца? Поскольку 6 — число относительно небольшое, решение получается довольно легко.

Месяц 1	Одна пара, исходные кролики.
Месяц 2	По-прежнему одна пара остается, поскольку кролики не достигли половой зрелости.
Месяц 3	Две пары. Исходная пара достигла половой зрелости и породила следующую пару.
Месяц 4	Три пары. Исходная пара снова родила двойню, однако пара, родившаяся в начале третьего месяца, еще не достигла половой зрелости.
Месяц 5	Пять пар. Все кролики, жившие в третьем месяце (две пары), достигли половой зрелости. Добавив их отпрысков к парам, родившимся в четвертом месяце, получаем пять пар.
Месяц 6	Восемь пар. Три новорожденные пары от кроликов, родившихся в четвертом месяце, плюс пять пар, живших в пятом месяце.

Теперь можно построить рекурсивное решение задачи, вычислив значение функции *rabbit*(*n*), равное количеству кроликов, живущих в *n* месяце. Для этого нужно найти правило, позволяющее вычислить значение функции *rabbit*(*n*-1). Учтем, что значение *rabbit*(*n*) равно сумме количества пар, живших до *n*-го месяца, плюс количество пар, родившихся в начале *n*-го месяца. В начале *n*-го месяца существует *rabbit*(*n*-1) пар кроликов. Не все из них достигли половой зрелости. Размножаться могут только те из них, кто жил в (*n*-2)-м месяце. Это означает, что количество пар, родившихся в начале *n*-го месяца, равно *rabbit*(*n*-2).

Таким образом, получается следующая рекуррентная формула.

Количество пар в n-м месяце

```
rabbit(n) = rabbit(n-1) + rabbit(n-2)
```

Это отношении проиллюстрировано на рис. 2.10.

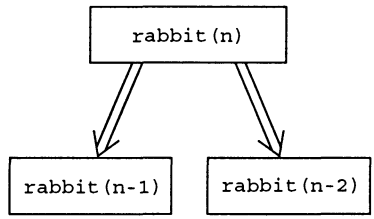


Рис. 2.10. Рекурсивное решение задачи о кроликах

Это рекуррентное решение порождает новые вопросы. В некоторых случаях исходная задача сводится к решению нескольких идентичных задач меньшего размера. Этот факт не создает дополнительных трудностей, однако теперь следует быть внимательным, выбирая базовую задачу. Возникает соблазн просто назвать вычисление значения *rabbit*(1) базисом, поскольку эта величина равна 1. А что можно сказать о величине *rabbit*(2)? Применяя рекурсивное определение, можно получить следующее соотношение.

```
rabbit(2) = rabbit(1) + rabbit(0)
```

Таким образом, рекурсивное решение нужно уточнить, задав количество пар, живущих в начале 0-го месяца, однако это значение мы не определяли.

При решении двух задач меньшего объема необходимо иметь два базиса

Можно определить значение *rabbit*(0), задав его равным нулю, однако такой подход выглядит несколько искусственно. Более естественно рассматривать значение *rabbit*(2) как особый случай и задать его равным 1. Таким образом, рекурсивное решение будет иметь два базиса — *rabbit*(2) и *rabbit*(1). Рекурсивное определение принимает следующий вид.

$$rabbit(n) = \begin{cases} 1, \text{если } n \text{ равно } 1 \text{ или } 2, \\ rabbit(n-1) + rabbit(n-2), \text{если } n > 2. \end{cases}$$

Кстати, последовательность чисел *rabbit*(1), *rabbit*(2), *rabbit*(3) и т.д. называется **последовательностью Фибоначчи** (Fibonacci sequence), в честь известного итальянского математика, впервые решившего эту задачу.

Функция rabbit вычисляет последовательность Фибоначчи, но является неэффективной

Пользуясь определением функции *rabbit*(*n*), приведенным выше, легко создать ее реализацию на языке C++.

```
int rabbit(int n)
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вычисляет члены последовательности Фибоначчи.
// Предусловие: аргумент n является целым и положительным числом.
// Постусловие: возвращает n-й член последовательности Фибоначчи.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   if (n <= 2)
       return 1;

   else // n > 2, поэтому n-1 > 0 и n-2 > 0
       return rabbit(n-1) + rabbit(n-2);
} // Конец функции rabbit
```

Можно ли использовать эту функцию на практике? На рис. 2.11 показаны рекурсивные вызовы, порожденные вызовом `rabbit`(7). Представьте себе количество рекурсивных вызовов, порожденных вызовом `rabbit`(10). Функция `rabbit`, мягко говоря, неэффективна. Таким образом, она непригодна для больших значений n. Более подробно эта проблема будет обсуждаться в конце главы, когда мы освоим приемы, позволяющие генерировать более эффективные решения для аналогичных рекурсивных соотношений.

Организация парада

Представьте себе, что нас попросили организовать парад в честь Дня Независимости, состоящий из музыкальных оркестров и платформ, выстроенных в линейку. В прошлый раз соседние оркестры заглушали друг друга, поэтому спонсоры попросили нас не располагать их в непосредственной близости. Сколько вариантов у нас есть, если парад может состоять лишь из n оркестров и платформ вместе взятых?

Допустим, у нас есть n оркестров и n платформ, из которых можно выбирать. Подсчитывая количество возникающих вариантов, будем предполагать, что парады типа *оркестр-платформа* и *платформа-оркестр* различаются между собой.

Парад может закрываться либо платформой, либо оркестром. Количество вариантов организации парада просто равно сумме парадов каждого типа. Введем следующие обозначения.

- $P(n)$ — Количество вариантов организации парада длины n
- $F(n)$ — Количество вариантов организации парада длины n, завершающегося платформой
- $B(n)$ — Количество вариантов организации парада длины n, завершающегося оркестром

Тогда общее количество вариантов выражается формулой:

$P(n) = F(n) + B(n)$.

Количество допустимых парадов длины n, заканчивающихся платформой

Сначала рассмотрим величину $F(n)$. Парад длины n, завершающийся платформой, получается просто, если платформу разместить в конце *любого* подходящего парада длины n-1. Следовательно, количество допустимых парадов длины n, заканчивающихся платформой, равно общему количеству допустимых парадов длины n-1.

$F(n) = P(n-1)$.

Далее рассмотрим величину $B(n)$. Если парад заканчивается оркестром, значит, перед ним расположена платформа (иначе два оркестра оказались бы соседями). Следовательно, единственный способ организовать парад длины n, завершающийся оркестром, — сначала организовать парад длины n-1, закрывающийся платформой. Итак, количество допустимых парадов длины n, завершающихся оркестром, точно равно количеству допустимых парадов длины n-1, закрывающихся платформой. Это приводит нас к формуле

$B(n) = F(n-1)$.

Используя ранее установленный факт, что $F(n) = P(n-1)$, получаем формулу

Количество допустимых парадов длины n, завершающихся оркестром

$B(n) = P(n-2)$.

Итак, мы выразили величины $F(n)$ и $B(n)$ через величины $P(n-1)$ и $P(n-2)$, соответственно, сведя исходную задачу к идентичным задачам меньшей размерности. Воспользовавшись формулой

Количество допустимых парадов длины n

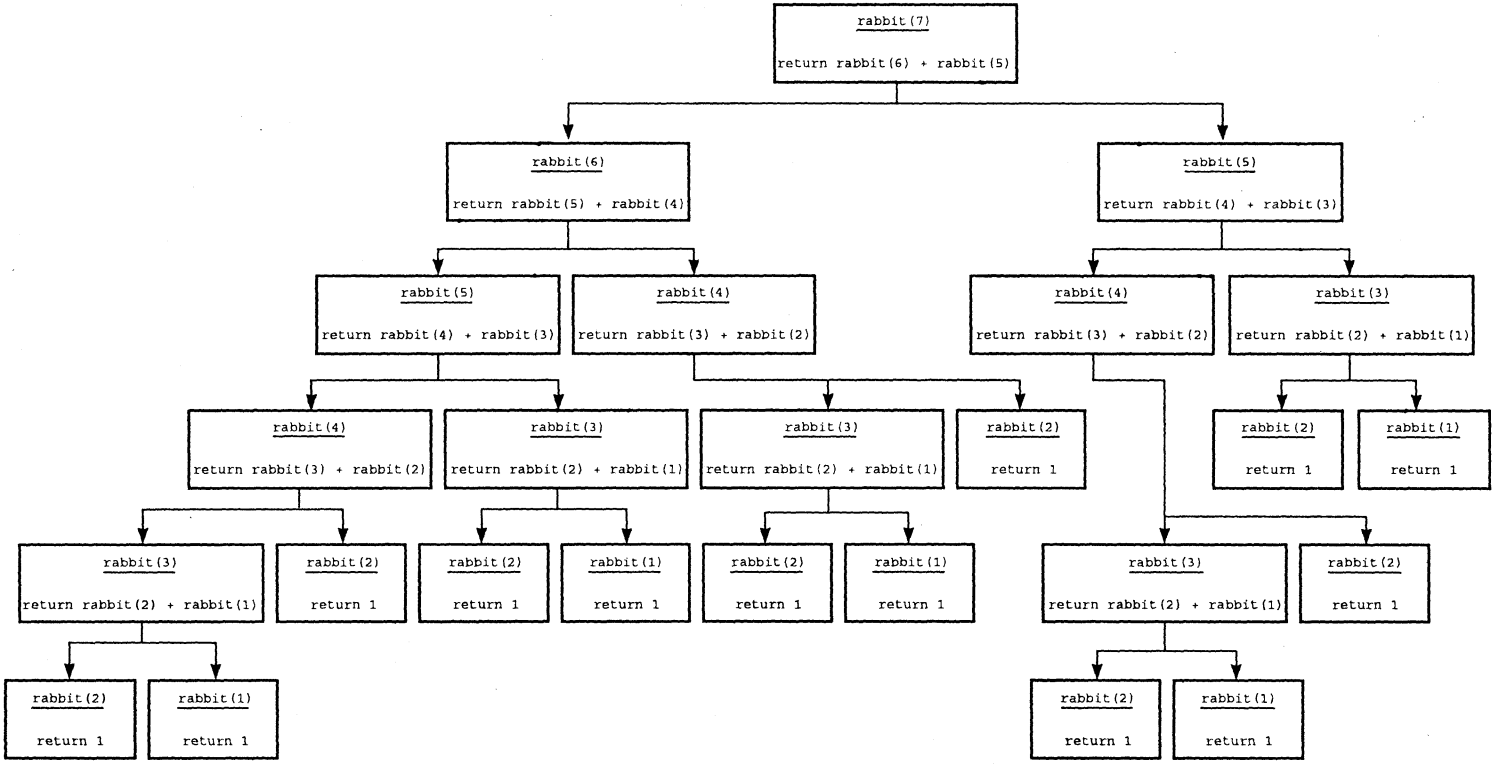


Рис. 2.11. Рекурсивные вызовы, порожденные вызовом rabbit(7)

$P(n) = F(n) + B(n)$,

получим соотношение

$P(n) = P(n-1) + P(n-2)$.

Этот вид рекурсивного соотношения абсолютно идентичен решению задачи Фибоначчи.

Как и прежде, задача имеет два базиса, поскольку рекурсивное решение исходной задачи сводится к решению двух задач меньшего размера. Как и в задаче Фибоначчи, в качестве базиса можно выбрать варианты n=1 и n=2. Хотя, на первый взгляд, базисные задачи в обоих случаях совпадают, не следует думать, что в них используются одни и те же значения. Следовательно, нет причин полагать, что величина *rabbit*(1) равна значению $P(1)$, а величина *rabbit*(2) — значению $P(2)$.

Задача о параде имеет два базиса, поскольку сводится к решению двух идентичных задач меньшего размера

Немного подумав, легко обнаружить, что для задачи о параде следует принять следующие исходные значения.

$P(1) = 2$ Парад длины 1 состоит либо из платформы, либо из оркестра

$P(2) = 3$ Парад длины 2 состоит либо из двух платформ, либо из двух оркестров, либо из платформы и оркестра

Итак, решение задачи имеет следующий вид.

Рекурсивное решение

$P(1) = 2$,

$P(2) = 3$,

$P(n) = P(n-1) + P(n-2)$ для всех n>2.

Этот пример демонстрирует следующие особенности рекурсии.

- Иногда задача сводится к решению нескольких идентичных задач меньшего размера. Например, задача о параде разбивается на задачу о параде, заканчивающемся платформой, и задачу о параде, завершающемся оркестром.
- Значения, используемые в базовой задаче, чрезвычайно важны. Несмотря на то что рекуррентные зависимости для величин P и *rabbit* одинаковы, разные значения, используемые в их базовых задачах (когда n=1 или 2), приводят к разным результатам. Например, *rabbit*(20)=6765, а $P(20)$=17711. Чем больше значение величины n, тем больше результаты отличаются друг от друга.

Дилемма мистера Спока (выбор k из n предметов)

Пятилетний полет космического корабля *U.S.S. Enterprise* должен увенчаться открытием новых миров. Пять лет почти истекли, когда корабль приблизился к неизвестной солнечной системе, состоящей из n планет. Командир корабля, мистер Спок, стал размышлять, сколько разных способов можно применить для исследования k планет, если солнечная система состоит из n планет. Поскольку времени у него было мало, он решил пренебречь порядком посещения планет.

Мистера Спока особенно интересовала планета X. Он стал думать, как выбрать k из n планет. "Есть две возможности: либо мы посещаем планету X, либо нет. Если мы посещаем планету X, другие $k-1$ планет можно выбрать из оставшихся $n-1$ планет. С другой стороны, если мы игнорируем планету X, из остальных $n-1$ планет можно выбрать k планет".

Таким образом, мистер Спок изобрел рекурсивный способ подсчета, сколько групп, состоящих из k планет, можно выбрать из солнечной системы, в которую входит n планет. Отталкиваясь от планеты X, мистер Спок вывел следующую формулу.

$c(n, k)$ = (количество групп, состоящих из k планет, включающих планету X)
+
(количество групп, состоящих из k планет, не включающих планету X).

Однако мистер Спок уже знает, что количество групп, включающих планету X, равно $c(n\text{-}1, k\text{-}1)$, а количество групп, не включающих планету X, равно $c(n\text{-}1, k)$. В таком случае общее количество вариантов посещения планет выражается формулой

Количество способов выбрать k из n предметов равно сумме количества способов выбрать k–1 из n–1 предметов и количества способов выбрать k из n–1 предметов

$$c(n, k) = c(n\text{-}1, k\text{-}1) + c(n\text{-}1, k).$$

Теперь следует подумать о базовых задачах. Для этого нужно показать, что каждая из двух задач меньшего размера в конце концов сводится к базовой. Во-первых, для какой задачи выбора ответ очевиден. Если бы космический корабль мог посетить все планеты (т.е. k=n), делать выбор не пришлось бы — есть только одна группа, состоящая из всех планет.

Итак, первый базис таков:

Базовая задача: выбирается только одна группа, состоящая из всех планет

$$c(k, k) = 1.$$

Если $k < n$, легко увидеть, что второй член рекурсивного определения величины $c(n\text{-}1, k)$ "ближе" к базовой задаче для вычисления $c(k, k)$, чем значение $c(n, k)$. Однако первый член $c(n\text{-}1, k\text{-}1)$ не ближе к величине $c(k, k)$, чем значение $c(n, k)$ — они находятся на "одинаковом" расстоянии. *При решении задачи путем сведения ее к двум (или более) задачам меньшего размера каждая из вспомогательных задач должна быть ближе к базовой, чем исходная.*

Первый член приводит к другой задаче простого выбора. Эта задача является дополнением первой базовой задачи, связанной с вычислением величины $c(k,k)$. В первом случае существовала лишь одна группа, состоявшая из всех планет (k=n), а во втором — есть только одна группа, не содержащая ни одной планеты (k=0). Если у космического корабля совсем нет времени для посещения хотя бы одной планеты, он должен немедленно разворачиваться и следовать домой.

Итак, второй базис таков:

Базовая задача: есть только одна группа, не содержащая ничего

$$c(n, 0) = 1.$$

Добавим завершающую часть решения:

$$c(n, k) = 0, \text{ если } k > n.$$

Хотя в контексте рассматриваемой задачи число k не может превышать число n, эта формула позволяет обобщить рекурсивное решение.

Подводя итоги, получаем следующее рекурсивное решение задачи о выборе k из n предметов:

$$c(n,k) = \begin{cases} 1, если\, k = 0, \\ 1, если\, k = n, \\ 0, если\, k > n, \\ c(n-1,k-1) + c(n-1,k), если\, 0 < k < n. \end{cases}$$

Основываясь на этом определении, легко получить рекурсивную функцию на языке C++.

```
int c(int n, int k)
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вычисляет количество групп, состоящих из k элементов,
// выбранных из n предметов.
// Предусловие: аргументы n и k являются неотрицательными
// целыми числами.
// Постусловие: возвращает значение c(n, k).
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   if ( (k == 0) || (k == n) )
      return 1;
   else if (k > n)
      return 0;
  else
      return c(n-1, k-1) + c(n-1, k);
} // Конец функции c
```

Как и функция *rabbit*, эта функция неэффективна и непрактична. На рис. 2.12 показано количество рекурсивных вызовов, порожденных вызовом *c*(4, 2).

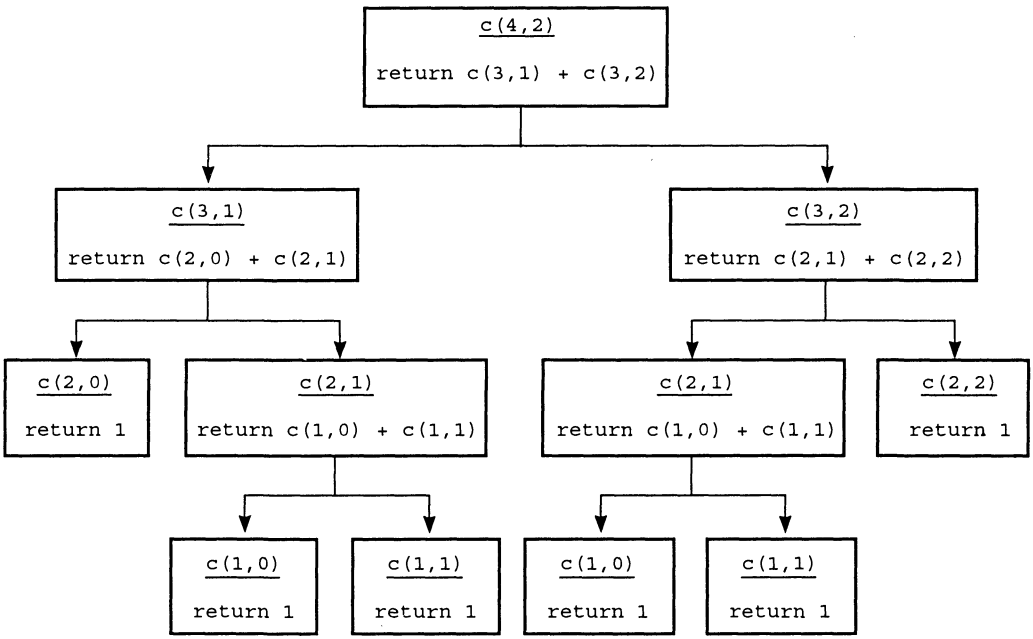


Рис. 2.12. Рекурсивные вызовы, порожденные вызовом c(4, 2)

Поиск элемента в массиве

Поиск — это важная и часто встречающаяся задача. В этой главе мы уже рассмотрели интуитивный подход к алгоритму бинарного поиска. В этом разделе мы подробнее опишем этот алгоритм, а также изучим другие задачи поиска, имеющие рекурсивное решение. Наша цель — углубить свои знания о рекурсии.

Поиск наибольшего элемента в массиве

Допустим, у нас есть массив *anArray*, состоящий из целых чисел, и нам нужно найти наибольшее среди них. Итеративное решение можно создать без каких-либо затруднений. Однако мы рассмотрим его рекурсивную формулировку.

```
if (массив anArray состоит лишь из одного элемента)
   maxArray(anArray) — это элемент из массива anArray
else if (массив anArray состоит из нескольких элементов)
   maxArray(anArray) — это максимальное из двух значений
        maxArray(левая часть массива anArray) и
        maxArray(правая часть массива anArray)
```

Функция maxArray разбивает обе задачи на каждом шаге

Обратите внимание, что эта стратегия основана на принципе "разделяй и властвуй", который использовался в алгоритме бинарного поиска, описанного в начале главы. Это значит, что исходная задача разбивается на подзадачи, которые решаются независимо друг от друга (рис. 2.13). Однако этот алгоритм отличается от бинарного поиска. В алгоритме бинарного поиска пополам делилась только одна из двух подзадач, возникающих на каждом шаге, а функция *maxArray* разбивает обе задачи. Кроме того, после решения каждой из подзадач функция *maxArray* находит максимум среди двух полученных результатов. Рис. 2.14 иллюстрирует вычисления, которые выполняются при поиске максимального элемента в массиве, состоящем из чисел 1, 6, 8 и 3.

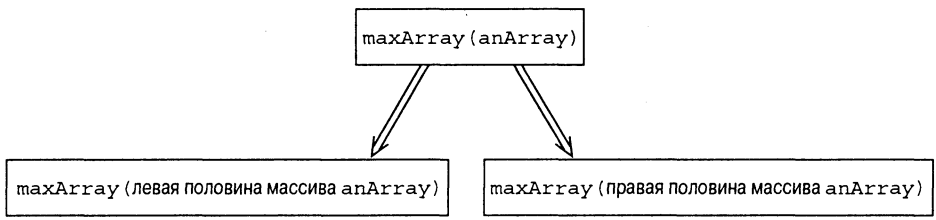


Рис. 2.13. Рекурсивное решение задачи о поиске максимального элемента

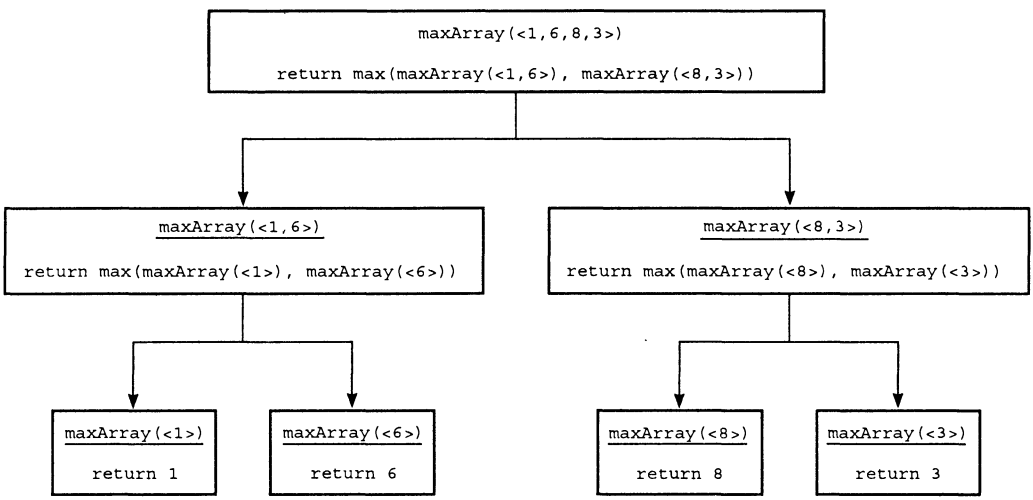


Рис. 2.14. Рекурсивные вызовы, порожденные вызовом maxArray(<1,6,8,3>)

Попробуем найти рекурсивное решение, основанное на этой стратегии. На этом пути мы можем столкнуться с некоторыми проблемами, связанными с программированием. Все эти проблемы возникают уже при реализации алгоритма бинарного поиска.

Бинарный поиск

В начале этой главы в самых общих чертах был описан рекурсивный алгоритм бинарного поиска слова в словаре. Теперь мы уточним этот алгоритм и проиллюстрируем некоторые важные вопросы, связанные с его программированием.

Напомним решение задачи о поиске слова в словаре, полученное нами ранее.

На каждом шаге бинарный поиск разбивает одну из подзадач

```
search(in aDictionary:Dictionary, in word:string)

   if (словарь aDictionary состоит из одной страницы)
      Ищем слово на этой странице
   else
   {
      Открываем словарь aDictionary посередине
      Определяем, какая половина словаря содержит искомое слово
      if (слово word содержится в первой половине aDictionary)
         search(первая половина словаря aDictionary, word)
      else
         search(вторая половина словаря aDictionary, word)
    }
```

Теперь мы немного изменим постановку задачи и будем искать заданное значение в массиве *anArray*, состоящем из целых чисел. Как и словарь, этот массив должен быть упорядоченным, иначе алгоритм бинарного поиска применять нельзя.

Итак, будем предполагать, что

Перед применением бинарного поиска массив нужно упорядочить

$$anArray[0] \le anArray[1] \le anArray[2] \le \ldots \le anArray[size-1],$$

где переменная *size* задает размер массива. В самых общих чертах, алгоритм бинарного поиска заданного элемента в массиве описывается следующим образом.

```
binarySearch(in anArray:ArrayType, in value:ItemType)

   if (размер массива anArray равен 1)
      Присваиваем переменной value значение,
      содержащееся в массиве
   else
   {
      Находим середину массива
      Определяем, какая половина массива содержит искомое число
      if (число value содержится в первой половине массива)
         binarySearch(первая половина массива anArray, value)
      else
         binarySearch(вторая половина массива anArray, value)
    }
```

Несмотря на то что это решение в принципе верно, перед реализацией этого алгоритма нужно рассмотреть несколько важных вопросов.

1. **Как передать "половину массива *anArray*" рекурсивному вызову функции *binarySearch*?** Каждому вызову можно передавать весь массив, при этом функция *binarySearch* должна просматривать только отрезок массива *anArray[first, last]*[1], т.е. часть массива, начинающуюся элементом *anArray[first]* и заканчивающуюся элементом *anArray[last]*. Таким образом, функции *binarySerach* нужно передавать еще два целых аргумента, *first* и *last*:

   binarySearch(anArray, first, last, value).

   Придерживаясь этого соглашения, новую середину массива можно вычислять по формуле

   mid = (first + last) / 2.

   Тогда выражение *binarySearch (первая половина массива anArray, value)* примет вид:

   binarySearch(anArray, first, mid-1, value).

   Половинами массива являются отрезки anArray[first..mid-1] и anArray[mid+1..last], причем ни один из них не содержит элемент anArray[mid]

2. **Как определить, какая из половин массива содержит число *value*?** Одна из возможных реализации выражения

   if (число value содержится в первой половине массива)

   имеет вид:

   if (value < anArray[mid])

   Однако равенство чисел *value* и *anArray[mid]* здесь не проверяется. Это может привести к неправильному выполнению алгоритма. После разделения массива на две части элемент *anArray[mid]* не принадлежит ни одной из половин. (Объединение этих половин не образует целый массив!) Следовательно, нужно проверить, не является ли *теперь* элемент *anArray[mid]* искомым числом, поскольку позднее он будет исключен из рассмотрения. Связь между правилом деления массива и правилом окончания вычислений (базисом) слаба и часто приводит к ошибкам. Нужно пересмотреть базовую задачу.

   Проверьте, не является ли элемент anArray[mid] искомым числом

3. **Что следует считать базовой задачей (базовыми задачами)?** В описании функции *binarySearch* указано, что ее выполнение прекращается, когда размер массива *anArray* становится равным 1. Если изменить процесс разбиения массива так, чтобы элемент *anArray[mid] принадлежал одной из половин*, алгоритм бинарного поиска можно было бы реализовать корректно, поскольку он имел бы лишь один базис. Однако лучше допустить существование двух базисов.

   3.1. *first > last*. Этот базис достигается, когда значения *value* в исходном массиве нет.

   3.2. *value==anArray[mid]*. Этот базис достигается, когда значение *value* в исходном массиве есть.

   Эти базовые варианты немного отличаются от рассмотренных ранее. По существу ответ вычисляется в результате решения базовой задачи. Это свойство присуще многим задачам поиска.

[1] Так в книге обозначается отрезок массива.

4. **Как функция *binarySearch* обозначает результат поиска?** Если функция *binarySearch* успешно обнаружила значение *value* в массиве, она возвращает в качестве ответа его индекс. Поскольку этот индекс никогда не бывает отрицательным, функция *binarySearch* может возвращать отрицательное значение, если значение *value* в массиве не обнаружено.

Ниже приведена функция *binarySearch*, написанная на языке C++ и реализующая изложенные идеи. Два рекурсивных вызова функции *binarySearch* обозначены буквами *X* и *Y*. Эти точки используются при анализе этой функции с помощью блок-схем.

```
int binarySearch(const int anArray[], int first,
                 int last, int value)
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Выполняет поиск значения в массиве, начиная с элемента
// anArray[first] и заканчивая элементом anArray[last].
// Предусловие: 0 <= first, last <= SIZE-1,
// где константа SIZE задает максимальный размер массива,
// причем anArray[first]<= anArray[first+1]<=...<=
// anArray[last].
// Постусловие: если значение аргумента value в массиве есть,
// функция возвращает индекс элемента, равного этому значению;
// в противном случае функция возвращает число -1.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   int index;
   if (first > last)
      index = -1; // Значения аргумента value
                  // в исходном массиве нет

   else
   {
    // Инвариант: если значение аргумента value в массиве есть,
    // то anArray[first] <= value <= anArray[last].
    int mid = (first + last)/2;
    if (value == anArray[mid])
       index = mid; // Значение аргумента value найдено
                    // в элементе anArray[mid]
    else if (value < anArray[mid])
       // Точка X
       index = binarySearch(anArray, first, mid-1, value);

    else
       // Точка Y
       index = binarySearch(anArray, first, mid+1, value);
   } // Конец блока else
   return index;
 } // Конец функции binarySearch
```

Обратите внимание, что функция *binarySearch* имеет следующий инвариант: если значение аргумента *value* в массиве *anArray* есть, то *anArray[first] <= value <= anArray[last]*.

На рис. 2.15 показаны результаты трассировки функции *binarySearch*, когда поиск выполняется в массиве, содержащем числа 1, 5, 9, 12, 15, 21, 29 и 31. Обратите внимание, как метки рекурсивных вызовов *X* и *Y* показаны на диаграмме. В упражнении 13, приведенном в конце главы, предлагается построить другие блок-схемы для трассировки этой функции.

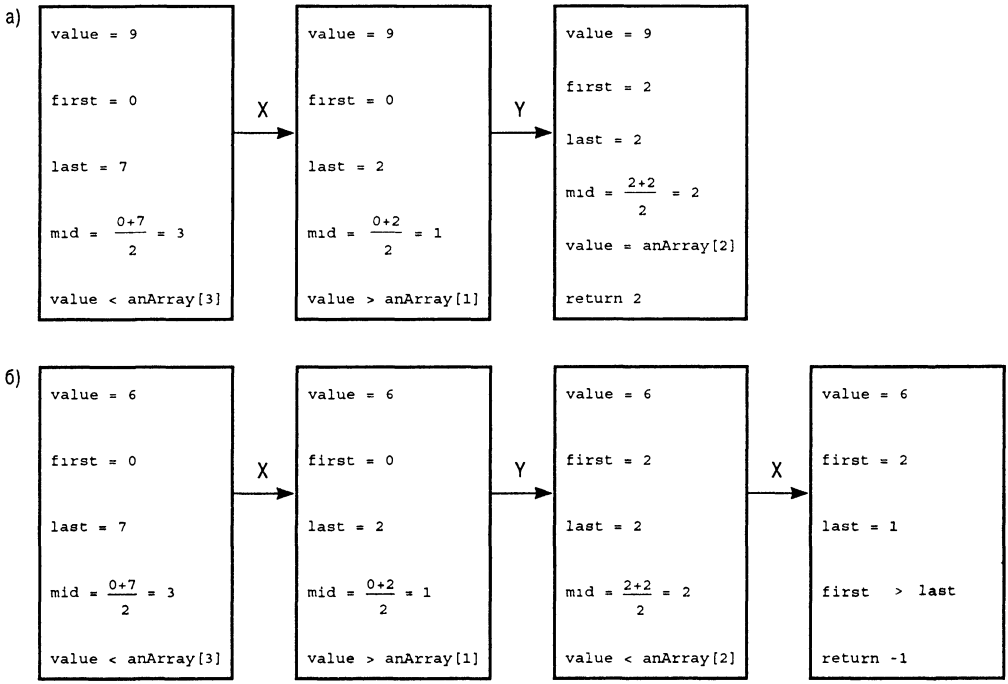


Рис. 2.15. Трассировка функции binarySearch с помощью блок-схем для случая anArray=<1, 5, 9, 12, 15, 21, 29, 31: а) успешно найдено число 9; б) безуспешный поиск числа 6

Поскольку аргумент, являющийся массивом, всегда передается по ссылке, функция может изменять его, если не указать модификатор const

Есть еще один вопрос, связанный с реализацией этой функции на языке C++. Напомним, что массивы никогда не передаются в функцию по значению и, следовательно, никогда не копируются. Эта особенность языка C++ особенно полезна при реализации рекурсивных функций, таких как `binarySearch`. Если массив `anArray` велик, понадобится много рекурсивных вызовов функции `binarySearch`. Если бы при каждом вызове массив `anArray` копировался, было бы потеряно много памяти и времени. Однако поскольку массив `anArray` не копируется, фукнция `binarySearch` может изменять его содержимое, если не использовать при его описании модификатор `const`.

Аргументы, передаваемые по ссылке, на диаграмме трассировки функции изображаются вне блоков

Трассировка рекурсивных функций, имеющих массив в качестве аргумента, приводит к новым проблемам. Поскольку массив `anArray` не передается по значению и не является локальной переменной, он не является частью локального окружения функции. Следовательно, весь массив `anArray` не нужно изображать в каждом блоке. Как показано на рис. 2.16, массив `anArray` изображается вне блоков, и все обращения к нему изображаются одинаково.

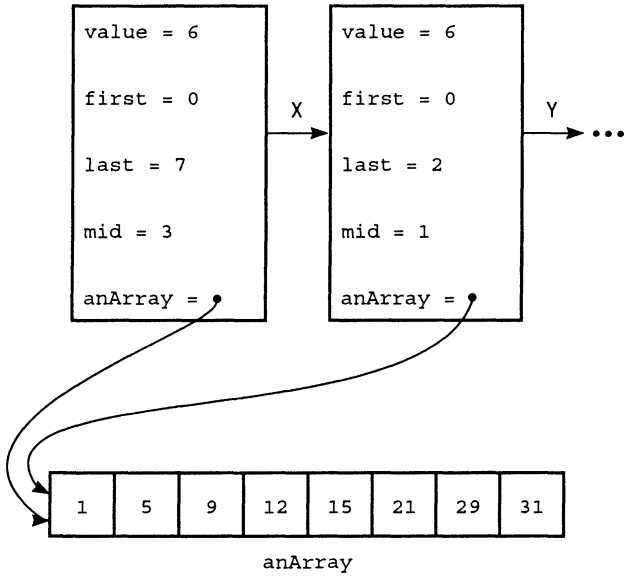


Рис. 2.16. Трассировка функции, аргумент которой передается по ссылке, с помощью блок-схем

Поиск k-го наименьшего элемента массива

Рассмотрим еще более сложную задачу. При первом чтении этот раздел можно пропустить, однако к некоторым проблемам, связанным с алгоритмом сортировки, мы еще вернемся в главе 9.

В предыдущих разделах мы изучили рекурсивные методы поиска наибольшего элемента в произвольном массиве и произвольного элемента в упорядоченном массиве. Рассмотрим теперь задачу поиска k-го наименьшего элемента в произвольном массиве `anArray`. В каких ситуациях возникает эта задача? Статистикам часто нужно вычислить медиану в некотором наборе данных. Медиана в упорядоченном наборе данных находится в середине набора. В неупорядоченном наборе данных количество чисел, не превышающих медиану, и количество чисел, превышающих медиану, равны между собой. Таким образом, если в наборе хранятся 49 чисел, то 25-й наименьший элемент и является медианой набора.

Очевидно, эту задачу можно решить с помощью сортировки массива. Тогда k-м наименьшим элементом будет число `anArray[k-1]`. Несмотря на то что этот подход к решению задачи вполне допустим, он не очень эффективен. Ниже описывается решение, в котором сортировку массива делать не обязательно.

Во всех предыдущих примерах степень уменьшения задачи при каждом рекурсивном вызове была известна заранее

Рекурсивное решение задачи означает ее сведение к идентичным задачам меньшего размера так, чтобы в конце концов рекурсия достигла базиса. Во всех рассмотренных выше задачах уменьшение размера задачи было *предсказуемым* (predictable). Например, функция для вычисления факториала всегда уменьшает размер задачи на 1, а бинарный поиск — вдвое. Кроме того, базовая задача во всех этих примерах, за исключением бинарного поиска, имела заранее известный размер. Таким образом, зная размер исходной задачи, можно определить количество рекурсивных вызовов, которое понадобится, чтобы достигнуть базиса.

Решение задачи поиска k-го наименьшего элемента нарушает общепринятые правила. Несмотря на то что исходная задача сводится к решению идентичной задачи меньшего размера, размер этой вспомогательной задачи зависит от количества элементов, хранящихся в массиве, поэтому его невозможно предсказать. Кроме того, размер базовой задачи также зависит от длины массива, как и при бинарном поиске. (Напомним, что базис при бинарном поиске достигается, когда искомым оказывается средний элемент.)

В задаче поиска k-го наименьшего элемента невозможно заранее предсказать размеры вспомогательной и базовой задач

Эта "непредсказуемость" вытекает из самой природы задачи: отношение между упорядоченными элементами в любой части массива и упорядоченными элементами во всем массиве недостаточно строго, чтобы однозначно определить k-й наименьший элемент. Допустим, что массив `anArray` содержит элементы, показанные на рис. 2.17. Обратите внимание, что число 6, т.е. `anArray[3]`, является третьим наименьшим элементом первой половины массива, а число 8, т.е. `anArray[4]`, — третьим наименьшим элементом второй половины. Можно ли на этом основании сделать вывод о местонахождении третьего наименьшего элемента всего массива `anArray`? Очевидно, нет. Этой информации совершенно недостаточно для решения задачи. Попробуйте убедиться в этом, экспериментируя с другими массивами.

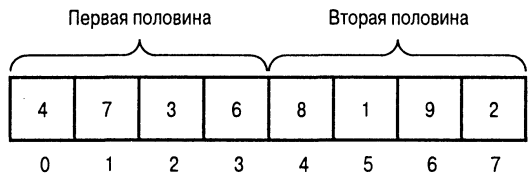


Рис. 2.17. Пример массива

Рекурсивное решение задачи сводится к следующим операциям.

1. Выбор **опорного элемента** (pivot element) в массиве.
2. **Разбиение** (partitioning) по отношению к опорному элементу.
3. Рекурсивное применение этой стратегии к **одной** из частей разбиения.

Допустим, требуется найти k-й наименьший элемент в отрезке массива `anArray[first..last]`. Обозначим через p опорный элемент этого отрезка. (Пока не будем заострять внимание на том, *как именно* выбирается опорный элемент.) Отрезок массива `anArray[first..last]` можно разбить на три части: S_1, состоящую из элементов, меньших опорного; сам элемент p; и S_2, состоящую из элементов, которые больше или равны опорному. Отсюда следует, что все элементы, принадлежащие отрезку S_1, меньше всех элементов, содержащихся в отрезке S_2. Это разбиение массива показано на рис. 2.18.

Разбиение массива anArray на три части: элементы<p, p и элементы>p

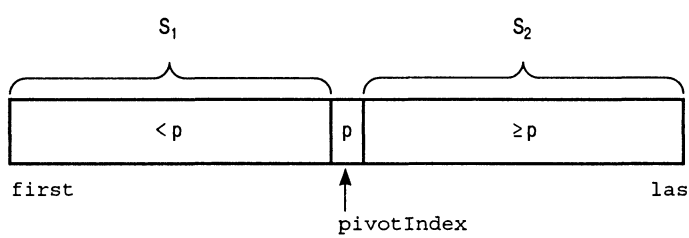


Рис. 2.18. Разбиение массива по отношению к опорному элементу

Все элементы отрезка *anArray[first..pivotIndex-1]* меньше, чем число *p*, а все элементы отрезка *anArray[first..pivotIndex+1]*— больше или равны числу *p*. Обратите внимание, что длины отрезков S_1 и S_2 зависят как от числа *p*, так и от остальных элементов отрезка *anArray[first..last]*.

Это разбиение порождает три задачи меньшего размера, причем решение одной из них приводит к решению исходной задачи.

1. Если отрезок S_1 состоит из *k* и более чисел, то он содержит *k* наименьших элементов отрезка *anArray[first..last]*. В этом случае *k*-й наименьший элемент следует искать в отрезке S_1. Поскольку S_1 — это отрезок массива *anArray[first..pivotIndex-1]*, то эта ситуация возникает, когда *k < pivotIndex-first+1*.
2. Если отрезок S_1 состоит из *k*-1 числа, то *k*-м наименьшим элементом является опорный. Этот вариант является базисным. Он имеет место, когда *k = pivotIndex-first+1*.
3. Если отрезок S_1 содержит меньше, чем *k*–1 элемент, то *k*-й наименьший элемент массива *anArray[first..last]* принадлежит отрезку S_2. Поскольку отрезок S_1 содержит *pivotIndex-first* элементов, *k*-й наименьший элемент отрезка *anArray[first..last]* является *(k-(pivotIndex-first+1)*-м наименьшим элементом отрезка S_2. Эта ситуация возникает, если *k>pivotIndex-first+1*.

Выразим это описание в виде рекурсивного определения. Пусть

kSmall(k, anArray, first, last) = *к*-й наименьший элемент отрезка *anArray[first..last]*

После выбора опорного элемента *p* и разбиения отрезка *anArray[first..last]* на отрезки S_1 и S_2 приходим к следующей формуле.

Формула для определения k-го наименьшего элемента отрезка anArray[first..last]

$$kSmall(k, anArray, first, last) = \begin{cases} kSmall(k, anArray, first, pivotIndex - 1), \\ \qquad \text{если } k < pivotIndex - first + 1, \\ p, \qquad \text{если } k = pivotIndex - first + 1 \\ kSmall(k - (pivotIndex - first + 1), \\ \qquad anArray, pivotIndex + 1, L), \\ \qquad \text{если } k > pivotIndex - first + 1. \end{cases}$$

Поскольку опорный элемент существует всегда, причем он не принадлежит отрезкам S_1 и S_2, длина сегмента, в котором выполняется поиск, на каждом шаге уменьшается по крайней мере на 1. Таким образом, рано или поздно мы достигнем базиса: искомым элементом будет опорный. Ниже приведен псевдокод решения этой задачи.

```
kSmall(in K;indeger, in anArray:ArrayType,
       in first:integer, in last:integer):ItemType
// Возвращает к-й наименьший элемент отрезка anArray[first..last].
   Выбор опорного элемента p в отрезке anArray[first..last]
   Разбиение отрезка anArray[first..last]
   по отношению к опорному элементу p

   if (k < pivotIndex - first + 1)
      return kSmall(k, anArray, first, pivotIndex-1)
```

```
else if (k == pivotIndex - first + 1)
   return p

else
   return kSmall(k-(pivotIndex-first+1), anArray,
                 pivotIndex+1, last)
```

Этот псевдокод очень похож на реальную функцию на языке C++. Осталось только уточнить, как именно выбирается опорный элемент *p* и как разбить массив по отношению к нему. Выбор элемента *p* произволен. Алгоритм будет работать для любого элемента *p*, хотя целенаправленный выбор опорных элементов может ускорить поиск. В главе 9 приведены несколько алгоритмов разбиения массива по отношению к опорному элементу *p*. Там же рассматривается применение функции *kSmall* в алгоритме сортировки.

Организация данных

Иногда возникает необходимость организовать данные по-своему. В этом разделе рассматривается широко известная задача о ханойских башнях. Хотя эта задача не имеет прямого отношения к реальным приложениям, ее решение ярко иллюстрирует применение рекурсии.

Ханойские башни

Много-много лет тому назад в далекой восточной стране — во вьетнамском городе Ханой — умер советник императора. Поскольку император и сам был не глуп, он придумал головоломку и объявил, что решивший ее человек займет место умершего советника.

Эта головоломка состояла из *n* колец (их количество мы уточнять не будем) и трех стержней: *A* (источник), *B* (цель) и *C* (запасной). Кольца имели разные размеры. Их можно было нанизывать на стержни. Из-за большого веса кольца можно было нанизывать только поверх еще большего кольца. В самом начале все кольца находились на стержне *A*, как показано на рис. 2.19, *а*. Задача заключалась в том, чтобы переместить диски один за другим со стержня *A* на стержень *B*. Игрок мог использовать стержень *C* как промежуточное звено, но кольца, как и прежде, должны были нанизываться так, чтобы сверху оказывались маленькие, а внизу — большие.

Поскольку должность советника считалась престижной, соискателей оказалось много. Ученики и крестьяне приносили императору свои решения. Многие решения состояли из тысяч шагов, содержали глубоко вложенные циклы и управляющие структуры. "Я не могу их понять, — кричал император. — Должен существовать простой способ решения этой головоломки."

Такой способ действительно существовал. Великий буддийский монах спустился с гор, чтобы увидеть императора. "Сын мой, — промолвил он, — твоя загадка настолько проста, что ты и сам можешь ее решить." Телохранители хотели вышвырнуть монаха из дворца, однако император остановил их.

"Если у тебя всего одно кольцо (т.е. n=1), перемести его со стержня *A* на стержень *B*. Это понятно и деревенскому дурачку. Если у тебя несколько колец (т.е. n>1), нужно сделать следующее.

1. Забыть на время про нижний диск и решить задачу с n-1 кольцом, считая целью стержень *C*, а запасным — стержень *B* (рис. 2.19, *б*).
2. После этого на стержне C окажется нанизанным n-1 кольцо, а самое большое кольцо останется на стержне *A*. Теперь нужно решить задачу для

n=1 (с этим справится даже ребенок), переместив кольцо со стержня *A* на стержень *B* (рис. 2.19, *в*).

3. Теперь нужно просто переместить все кольца со стержня *C* на стержень *B*, т.е. решить задачу, в которой стержень *C* является источником, стержень *B* — целью, а стержень *A* считается запасным (рис. 2.19, *г*)."

В покоях императора на несколько мгновений воцарилось молчание. Затем император нетерпеливо спросил: "Ну что, ты собираешься изложить нам свое решение или нет?" В ответ монах улыбнулся и исчез.

Очевидно, император не обладал навыками рекурсивного мышления, однако решение монаха было абсолютно правильным. Ключом к решению является разбиение исходной задачи на три идентичные задачи меньшего размера (если размером задачи считать количество колец). Обозначим задачу о перемещении `count` колец со стержня `source` на стержень `destination` с помощью запасного стержня `spare`, как `towers(count, source, destination, spare)`. Обратите внимание, что это определение остается корректным, даже если на стержне `source` нанизано больше, чем `count` колец (в этом случае учитываются лишь верхние `count` колец, а остальные игнорируются). Аналогично, стержни `destination` и `spare` не обязаны быть пустыми. Кольца, нанизанные на них до этого, также игнорируются. Не забывайте однако, что кольца можно нанизывать только поверх больших колец.

Формулировка задачи

Задачу, поставленную императором, можно переформулировать следующим образом. Исходное положение: на стержне *A* нанизано n колец, на стержнях *B* и *C* — ни одного. Требуется решить задачу `towers(n, a, B, C)`.

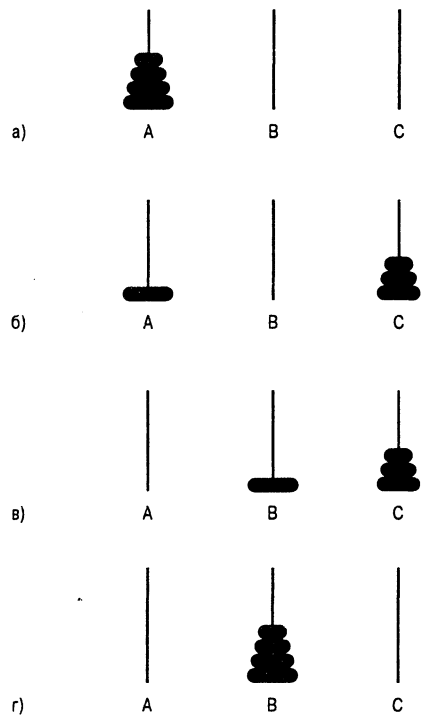


Рис. 2.19. Решение задачи о ханойских башнях: а) начальное состояние; б) перемещаем n–1 кольцо со стержня A на стержень C; в) перемещаем одно кольцо со стержня A на стержень B; г) перемещаем n–1 кольцо со стержня C на стержень B

Решение, предложенное монахом, теперь выглядит так.

Решение

1. Начиная с исходного положения, когда все кольца находятся на стержне *A*, решите задачу

   `towers(n-1, A, C, B).`

   Таким образом, нижнее (самое большоет кольцо) нужно проигнорировать и переместить верхние кольца (n-1 штуку) со стержня *A* на стержень *C*, используя стержень *B* в качестве запасного. После этого самое большое кольцо останется на стержне *A*, а все остальные окажутся на стержне *C*.

2. Теперь, когда самое большое кольцо находится на стержне *A*, а все остальные нанизаны на стержень *C*, решите задачу

   `towers(1, A, B, C).`

   Это значит, что самое большое кольцо нужно переместить со стержня *A* на стержень *B*. Поскольку это кольцо больше всех остальных, уже нанизанных на стержень *C*, запасной стержень использовать нельзя. Однако, к счастью, при решении базовой задачи запасной стержень не нужен. После ее решения самое большое кольцо окажется на стержне *B*, а все остальные кольца останутся на стержне *C*.

3. В заключение, когда самое большое кольцо нанизано на стержень *B*, а все остальные кольца находятся на стержне *C*, решите задачу

   `towers(n-1, C, B, A).`

   Это значит, что n-1 кольцо нужно переместить со стержня *C* на стержень *B*, используя стержень *A* в качестве запасного. Обратите внимание, что на стержне *B* уже нанизано самое большое кольцо, которое мы игнорируем. После этого исходная задача оказывается решенной: все кольца нанизаны на стержень *B*.

Псевдокод решения задачи `towers(count, source, destination, spare)` имеет следующий вид.

```
solveTowers(count, source, destination, spare)

   if (аргумент count равен 1)
      Переместите кольцо непосредственно со стержня source
      на стержень destination
   else
   {
      solveTowers(count-1, source, spare, destination)
      solveTowers(1, source, destination, spare)
      solveTowers(count-1, spare, destination, source)
   } // Конец оператора if
```

Это решение полностью соответствует основным принципам рекурсивного решения, сформулированным ранее.

Решение задачи о ханойских башнях соответствует четырем критериям рекурсивного решения

1. Решение задачи о ханойских башнях сводится к решению идентичных задач.
2. Эти задачи имеют меньший размер: в них требуется переместить меньшее количество колец, причем каждый раз количество колец, подлежащих переносу, уменьшается на 1.

3. Когда остается только одно кольцо — базовая задача, — решение очевидно.
4. Способ, благодаря которому размер задач постоянно уменьшается, гарантирует достижение базиса.

Для того чтобы решить задачу о ханойских башнях, нужно решить несколько идентичных задач меньшего размера. На рис. 2.20 показаны возникающие рекурсивные вызовы, а также их порядок при решении задачи для трех колец.

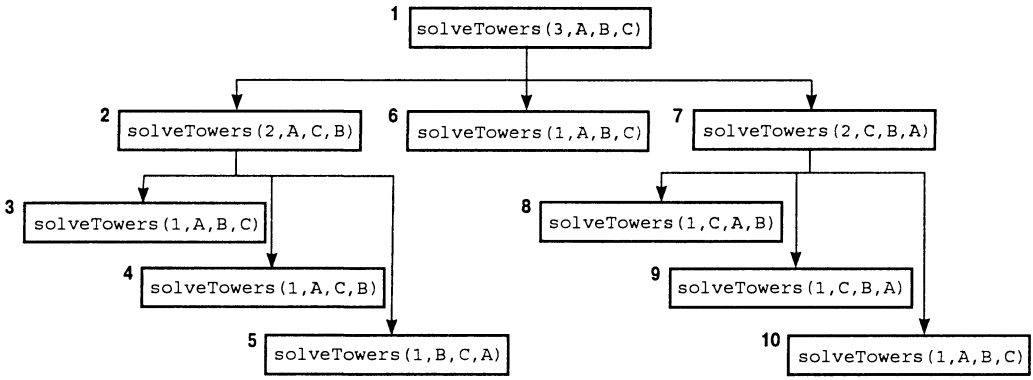


Рис. 2.20. Порядок рекурсивных вызовов, генерируемых вызовом solveTowers(3, A, B, C)

Рассмотрим теперь реализацию этого алгоритма на языке C++. Обратите внимание, что большинство компьютеров пока еще не может перемещать кольца, поэтому функция просто указывает направление перемещения. Таким образом, ее формальные аргументы, представляющие стержни, имеют тип *char*, а соответствующие фактические аргументы могут принимать значения *'A'*, *'B'* и *'C'*.

Вызов *solveTowers(3, 'A', 'B', 'C')* выводит на экран следующие строки.

Решение задачи для трех колец

```
Переместите верхнее кольцо со стержня A на стержень B
Переместите верхнее кольцо со стержня A на стержень C
Переместите верхнее кольцо со стержня B на стержень C
Переместите верхнее кольцо со стержня A на стержень B
Переместите верхнее кольцо со стержня C на стержень A
Переместите верхнее кольцо со стержня C на стержень B
Переместите верхнее кольцо со стержня A на стержень B
```

Соответствующая функция на языке C++ выглядит так.

```
void solveTowers(int count, char source, char destination,char spare)
{
   if (count == 1)
   {
      cout << " Переместите верхнее кольцо со стержня " <<
           source << " на стержень "  << destination << endl;
   }
   else
   {
      solveTowers(count-1, source, spare, destination); // X
      solveTowers(1, source, destination, spare);       // Y
      solveTowers(count-1, spare, destination, source); // Z
   } // Конец оператора if
} // Конец функции solveTowers
```

Три рекурсивных вызова отмечены метками *X*, *Y* и *Z*. Эти метки показаны на диаграмме трассировки вызова `solveTowers(3, 'A', 'B', 'C')` (рис. 2.21). Нумерация рекурсивных вызовов на рис. 2.20 и 2.21 совпадает. (На рис. 2.21 для параметра `destination` используется сокращение `dest`.)

Рекурсия и эффективность

Рекурсия — мощный метод, позволяющий получить простые решения очень сложных задач. Рекурсивные решения легче понять и описать, чем итеративные. Используя рекурсию, можно создавать простые и короткие программы.

Основное предназначение этой главы — дать читателю глубокие знания о рекурсии, позволяющие применить ее для решения своих собственных задач. Рассмотренные примеры, в основном, были простыми. К сожалению, многие рекурсивные решения, описанные в этой главе, были настолько неэффективны, что применять их на практике мы не рекомендуем. Рекурсивные функции `binarySearch` и `solveTowers` — счастливые исключения из этого правила, поскольку они довольно эффективны.[2]

Факторы, обусловливающие неэффективность рекурсии

Неэффективность рекурсии определяется двумя факторами.

- Накладные расходы ресурсов, связанные с вызовами функций
- Изначальная неэффективность некоторых рекурсивных алгоритмов

Первый из этих факторов характерен не только для рекурсивных функций, но и для функций вообще. В большинстве реализаций языка C++ и других высокоуровневых языков программирования вызовы функций сопровождаются накладными расходами, связанными с их регистрацией. Как указывалось ранее, каждый вызов функции порождает активационную запись, представляющую собой аналог блока в блок-схеме. Рекурсивные функции увеличивают эти расходы, поскольку один-единственный вызов такой функции порождает большое количество рекурсивных вызовов. Например, вызов функции *factorial*(*n*) порождает *n* рекурсивных вызовов.

Рекурсия позволяет упрощать сложные решения

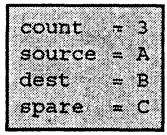
Использование рекурсии, как и модульного подхода в целом, может существенно упростить сложные программы. Такое упрощение часто компенсирует возникающие дополнительные затраты ресурсов. Таким образом, применение рекурсии часто полностью соответствует многомерной точке зрения на эффективность компьютерной программы, описанной в главе 1.

Не применяйте рекурсивное решение, если оно не эффективно, в то время как существует ясное и эффективное итеративное решение

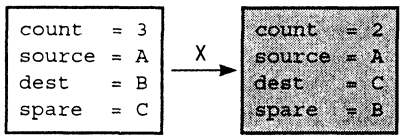
Следует иметь в виду, что применение рекурсии — не самоцель. Например, применять такой алгоритм вычисления факториала на практике не рекомендуется. Для вычисления факториала можно легко написать простую итеративную функцию. Она настолько же проста, как и рекурсивная, но намного эффективнее. Нет причин применять рекурсию, если это ничего не дает. *Рекурсия полезна только тогда, когда задача не имеет простого итеративного решения.*

[2] Другие практические и эффективные приложения рекурсии рассмотрены в главах 5 и 9.

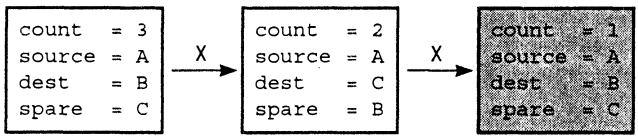
Выполнен первоначальный вызов 1. Начинается выполнение функции `solveTowers`.



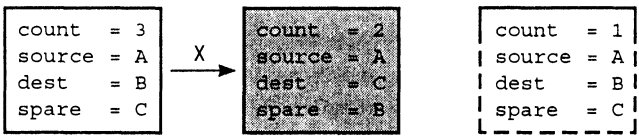
В точке X выполнен рекурсивный вызов 2, начинается новое выполнение функции.



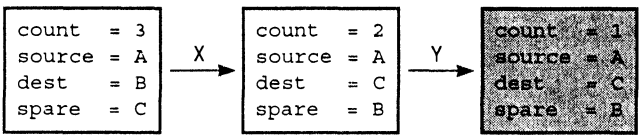
В точке X выполнен рекурсивный вызов 3, начинается новое выполнение функции.



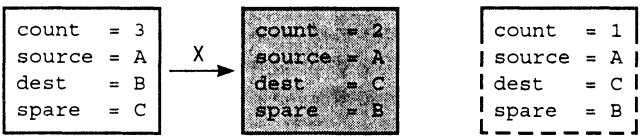
Это — базовая задача, поэтому перемещается кольцо, выполняется возврат управления, и функция продолжает свое выполнение.



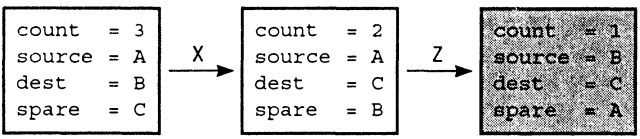
В точке Y выполнен рекурсивный вызов 4, начинается новое выполнение функции.



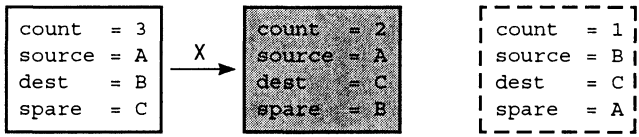
Это — базовая задача, поэтому перемещается кольцо, выполняется возврат управления, и функция продолжает свое выполнение.



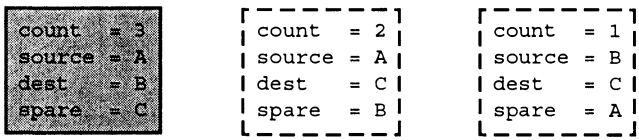
В точке Z выполнен рекурсивный вызов 5, начинается новое выполнение функции.



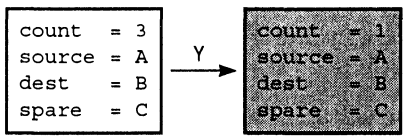
Это — базовая задача, поэтому перемещается кольцо, выполняется возврат управления, и функция продолжает свое выполнение



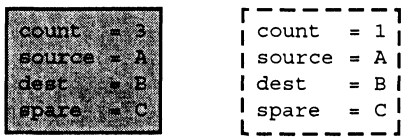
Это выполнение функции завершено. Управление возвращается в вызывающий модуль, и фунция продолжает работатать.



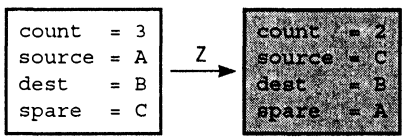
В точке Y выполнен рекурсивный вызов 6, начинается новое выполнение функции.



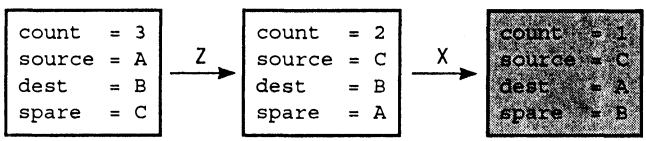
Это — базовая задача, поэтому перемещается кольцо, выполняется возврат управления, и функция продолжает свое выполнение.



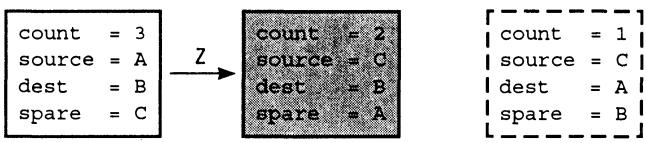
В точке Z выполнен рекурсивный вызов 7, начинается новое выполнение функции.



В точке X выполнен рекурсивный вызов 8, начинается новое выполнение функции.



Это — базовая задача, поэтому перемещается кольцо, выполняется возврат управления, и функция продолжает свое выполнение.



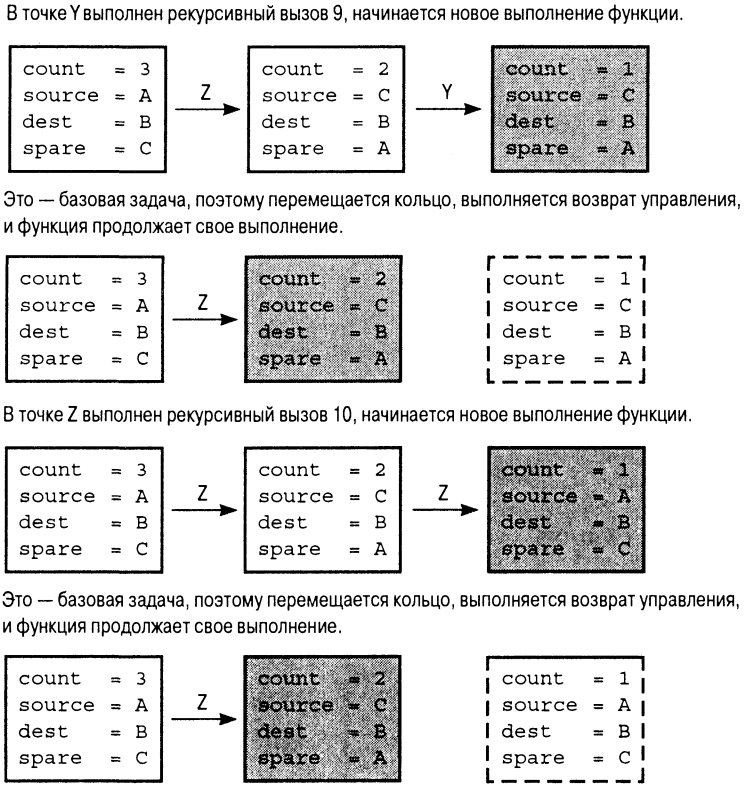


Рис. 2.21. Трассировка вызова solveTowers(3, A, B, C) с помощью блок-схем

Второй фактор, влияющий на эффективность рекурсии, связан с тем, что некоторые рекурсивные алгоритмы изначально неэффективны. Эта неэффективность не имеет ничего общего с накладными расходами, порождаемыми рекурсивными вызовами. Она обусловлена не компьютерной реализацией алгоритма, а самим методом решения задачи.

В качестве примера вернемся к рекурсивному решению задачи о размножающихся кроликах:

Рекурсивная версия функции rabbit изначально неэффективна

$$rabbit(n) = \begin{cases} 1, \textit{если } n \textit{ равно } 1 \textit{ или } 2, \\ rabbit(n-1) + rabbit(n-2), \textit{если } n > 2. \end{cases}$$

Диаграмма, изображенная на рис. 2.11, иллюстрирует вычисление вызова *rabbit*(7). Мы уже предлагали читателям пофантазировать о том, как выглядела бы диаграмма, соответствующая вызову *rabbit*(10). Думаем, вы уже поняли, что эта диаграмма заняла бы большую часть этой главы. А диаграмма, соответствующая вызову *rabbit*(100), вообще заняла бы больше половины Вселенной!

Основная проблема, связанная с функцией *rabbit*, заключается в том, что она вычисляет одно и то же значение снова и снова. Например, на диаграмме, иллюстрирующей вызов *rabbit*(7), видно, что значение `rabbit(3)` вычисляется пять раз. Если число *n* достаточно велико, большинство значений функции повторно

вычислялось бы триллионы раз. Это громадное количество вычислений делает рекурсивное решение совершенно неприемлемым, даже если каждое вычисление само по себе не требует большого объема работы (например, если бы мы могли выполнять миллион таких вычислений в секунду).

Рекуррентные соотношения, применяемые в функции rabbit, можно использовать для создания эффективного итеративного решения

Однако отсюда не следует, что рекурсивное решение абсолютно бесполезно. Используя установленные рекуррентные соотношения, можно решить задачу о размножающихся кроликах итеративным способом. В процессе итеративного решения все вычисления выполняются только по одному разу. Приведенное ниже итеративное решение можно использовать для вычисления величины *rabbit*(*n*) даже для очень большого числа *n*.

```
int iterativeRabbit(int n)
// Итеративное решение задачи о размножающихся кроликах
{
   // Инициализация базисов
   int previous = 1; // Начальное значение rabbit(1)
   int current = 1;  // Начальное значение rabbit(2)
   int next = 1;     // Результат вычислений при n=1 и 2
   // Вычисление следующего значения для n >= 3
   for (int i = 3; i <= n; ++i)
   {
      // Переменная current равна rabbit(i-1),
      // переменная previous равна rabbit(i)
      next = current + previous; // Значение rabbit(i)
      previous = current;        // Подготовка к
      current = next;            // следующей итерации
   } // Конец цикла for
   return next;
} // Конец функции iterativeRabbit
```

Переходите от рекурсивного решения к итеративному, если рекурсивное решение легче, а итеративное - эффективнее

Таким образом, итеративное решение может оказаться эффективнее рекурсивного. Однако в некоторых случаях рекурсивное решение получить легче, чем итеративное. Следовательно, может возникнуть необходимость преобразовать рекурсивное решение в итеративное. Это преобразование становится проще, если рекурсивная функция вызывает саму себя лишь один раз, а не многократно. Будьте внимательны, принимая такое решение. Например, хотя функция *rabbit* вызывает себя дважды, функция *binarySearch* вызывает себя лишь один раз, несмотря на то что в коде на языке C++ указаны два вызова. Эти вызовы находятся в разных ветвях условного оператора *if*, поэтому выполняется лишь один из них.

Конечно-рекурсивные функции

Преобразовать рекурсивное решение в итеративное еще проще, если единственный рекурсивный вызов является *последним* оператором функции. Эта ситуация называется **конечной рекурсией** (tail recursion). Например, функция *writeBackward* демонстрирует пример конечной рекурсии, поскольку ее рекурсивный вызов находится в самом конце. Однако не торопитесь с выводами, посмотрите на функцию *fact*. Несмотря на то что ее рекурсивный вызов также находится в конце определения, последним ее действием является умножение. Следовательно, функция *fact* не является конечно-рекурсивной.

Напомним определение функции *writeBackward*.

```
void writeBackward(string s, int size)
{
   if (size > 0)
   {
      // Записываем последний символ
      cout << s.substr(size-1, 1);
      writeBackwards(s, size-1); // Точка A
   } // Конец оператора if
} // Конец функции writeBackward
```

В большинстве случаев устранить конечную рекурсию легко

Поскольку эта функция является конечно-рекурсивной, ее последний рекурсивный вызов просто повторяет выполнение функции при изменившихся аргументах. Это повторение можно реализовать с помощью итерации, что приводит к более эффективному решению. Например, приведенное ниже определение функции *writeBackward* является итеративным.

```
void writeBackward(string s, int size)
// Итеративная версия.
{
   while(size > 0)
   {
      cout << s.substr(size-1, 1);
      --size;
   } // Конец оператора while
} // Конец функции writeBackward
```

Поскольку конечно-рекурсивные функции часто менее эффективны, чем их итеративные аналоги, а преобразование конечно-рекурсивной функции в эквивалентную итеративную функцию выполняется совершенно механически, некоторые компиляторы автоматически заменяют конечную рекурсию соответствующей итеративной конструкцией. Исключить другие виды рекурсии обычно намного сложнее (детальнее эти вопросы обсуждаются в главе 6).

Некоторые рекурсивные алгоритмы, такие как *rabbit*, изначально неэффективны, в то время как другие алгоритмы, например, бинарного поиска[3], чрезвычайно эффективны. Методы сравнительного анализа эффективности алгоритмов излагаются в рамках курсов по анализу алгоритмов. Некоторые из этих методов кратко рассматриваются в главе 9.

Обсуждение рекурсии будет продолжено в главе 5.

Резюме

1. Рекурсия — это метод решения задач путем сведения их к решению нескольких идентичных задач меньшего размера.
2. При создании рекурсивного решения следует учитывать четыре вопроса.
   2.1. Как свести исходную задачу к идентичным задачам меньшего размера?
   2.2. Как уменьшать размер задачи при каждом рекурсивном вызове?
   2.3. Какая задача является базисной?
   2.4. Можно ли достичь базиса, постоянно уменьшая размер исходной задачи?
3. При создании рекурсивного решения предполагается, что если выполняется постусловие рекурсивного вызова, то должно выполняться и его предусловие.

[3] Алгоритм бинарного поиска также имеет итеративную формулировку.

4. Для трассировки рекурсивной функции можно применить метод блок-схем. Блоки, образующие соответствующую диаграмму, представляют собой активационные записи, которые создаются многими компиляторами для реализации рекурсии. (Вопросы, связанные с реализацией рекурсии, обсуждаются в главе 6.) Несмотря на полезность метода блок-схем, он не может заменить собой интуитивное понимание рекурсии.
5. Рекурсия позволяет решать задачи, — такие как задача о ханойских башнях, — которые трудно решить итеративным путем. Даже очень сложные задачи часто имеют простые рекурсивные решения. Понять, описать и реализовать рекурсивные решения проще, чем итеративные.
6. Некоторые рекурсивные решения гораздо менее эффективны, чем их итеративные аналоги, поскольку алгоритмы, лежащие в их основе, изначально не эффективны, а рекурсивные вызовы приводят к дополнительным затратам ресурсов. Однако рекурсивное решение можно использовать при разработке итеративного.
7. Если существует простое, ясное и эффективное итеративное решение задачи, рекурсию применять не следует.

Предупреждения

1. Рекурсивный алгоритм должен иметь базовую задачу, решение которой очевидно и не требует выполнения рекурсивных вызовов. Если базовой задачи нет, рекурсивная функция порождает бесконечную последовательность вызовов. Если рекурсивная функция содержит несколько рекурсивных вызовов, скорее всего, существует несколько базовых задач.
2. Рекурсивное решение должно сводиться к решению одной или нескольких задач меньшего размера, каждая их которых ближе к базовой, чем исходная. Необходимо убедиться, что базис будет рано или поздно достигнут, в противном случае алгоритм не будет завершен.
3. Применяя рекурсию, необходимо убедиться, что решения задач меньшего размера действительно приводят к решению исходной задачи. Например, функция `binarySearch` работоспособна, поскольку каждый массив меньшей длины упорядочен, а искомая величина лежит между их первыми и последними элементами.
4. Метод блок-схем в сочетании с продуманными операторами промежуточного вывода позволяют успешно отлаживать рекурсивные функции. Такие операторы должны сообщать, в какой точке программы осуществляется рекурсивный вызов, а также какие значения имеют аргументы функции и ее локальные переменные на входе и выходе. Из окончательной версии операторы промежуточного вывода нужно удалить.
5. Рекурсивные решения, которые сводятся к многократным повторным вычислениям одних и тех же величин, могут оказаться совершенно неэффективными. В этих случаях итерация предпочтительнее рекурсии.

Вопросы для самопроверки

1. Приведенная ниже функция вычисляет произведение первых $n \geq 1$ действительных чисел, хранящихся в массиве. Соответствует ли она критериям рекурсивной функции?

```
double product(const double anArray[], int n)
// Предусловие: 1 <= n <= максимальный размер массива anArray
// Постусловие: возвращает произведение первых n элементов
// массива anArray; сам массив anArray остается неизменным.
{
   if (n == 1)
      return anArray[0];
   else
      return anArray[n-1] * product(anArray, n-1);
} // Конец функции product
```

2. Перепишите функцию из вопроса 1 так, чтобы она ничего не возвращала (возвращаемое значение имеет тип *void*).
3. Задано целое число $n > 0$. Напишите рекурсивную функцию *countDown*, выводящую на печать целые числа n, $n-1$, ... , 1. *Подсказка*: какую задачу вы можете решить сами, а какую можно поручить другу?
4. Напишите рекурсивную функцию, вычисляющую произведение элементов массива *anArray[first..last]*.
5. Какие из функций, рассмотренных в данной главе, можно назвать конечно-рекурсивными: *fact*, *writeBackward*, *writeBackward2*, *rabbit*, *c* (в задаче мистера Спока), *P* (в задаче об организации парада), *maxArray*, *binarySearch* и *kSmall*?
6. Вычислите значение *c(4, 2)* в задаче мистера Спока.
7. Осуществите трассировку функции *solveTowers* для решения задачи о ханойских башнях с двумя кольцами.

Упражнения

1. Приведенная ниже рекурсивная функция *getNumberEqual* осуществляет поиск целого числа *desiredValue* в массиве *x*, содержащем *n* целых чисел. Она возвращает количество целых чисел, равных величине *desiredValue*. Например, если массив *x* содержит числа 1, 2, 4, 4, 5, 6, 7, 8, 9 и 12, то *вызов getNumberEqual(x, 10, 4)* вернет значение 2, поскольку число 4 дважды встречается в массиве *x*.

```
int getNumberequal(const int x[], int n, int desiredValue)
{
   int count;
   if (n <= 0)
      return 0;
   else
   {
      if (x[n-1] == desiredValue)
         count = 1;
      else
         count = 0;
      return getNumberequal(x, n-1, desiredvalue) + const;
   } // Конец раздела else
} // Конец функции getNumberEqual
```

Докажите, что эта функция является рекурсивной, проверив критерии рекурсивности.

2. Выполните трассировку следующих вызовов рекурсивных функций, встречавшихся в этой главе. Точно укажите каждый последующий вызов.

   2.1. *rabbit(5)*

   2.2. *countDown(5)* (функция из вопроса 3).

3. Напишите рекурсивную функцию, вычисляющую сумму первых *n* целых чисел, хранящихся в массиве, длина которого больше или равна *n*. *Подсказка*: начните с *n*-го целого числа.

4. Создайте новую версию функции *writeBackward*, рассмотренной в разделе "Рекурсивные функции, не возвращающие никаких значений: обратная запись строки" так, чтобы базис достигался, когда длина строки становится равной 1.

5. Дано целое число $n > 0$. Напишите на языке C++ рекурсивную функцию, выводящую на печать числа 1, 2, ..., *n*.

6. Напишите на языке C++ рекурсивную функцию, выводящую на печать в обратном порядке цифры положительного десятичного целого числа.

7. Выполните следующие задания.

   7.1. Напишите на языке C++ рекурсивную функцию *writeLine*, выводящую на печать *n* одинаковых символов. Например, вызов *writeLine*('*', *5*) выводит на печать строку *****.

   7.2. Напишите рекурсивную функцию *writeBlock*, использующую функцию *writeLine* для вывода на печать *m* строк, состоящих из *n* символов каждая. Например, вызов *writeBlock*('*', *5*, *3*) выводит на печать следующие строки:

```
*****
*****
*****
```

8. Что будет выведено на печать при выполнении следующей программы?

```
#include <iostream.h>
int getValue(int a, int b, int n);

int main()
{
   cout << getValue(1, 7, 7) << endl;
   return 0;
}  // Конец функции main

int getValue(int a, int b, int n)
{
   int returnValue;
   cout << "На входе: a = " << a << " b= " << b << endl;

   int c = (a + b)/2;
   if (c * c <= n)
      returnValue = c;
   else
      returnValue = getValue(a, c-1, n);
   cout << "На выходе: a = " << a << " b = " << b << end;
   return returnValue;
} // Конец функции getValue
```

9. Что будет выведено на печать при выполнении следующей программы?

```
#include <iostream.h>
int search(int first, int last, int n);
int mystery(int n);

int main()
{
   cout << mystery(30) << endl;
   return 0;
} // Конец функции main

int search(int first, int last, int n)
{
   int returnValue;
   cout << "На входе: first = " << first << "last = "
        << last << endl;

   int mid = (first + last)/2;
   if ( (mid * mid <= n) && (n < (mid+1) * (mid+1)) )
      returnValue = mid;
   else if (mid * mid > n)
      returnValue = search(first, mid-1, n);
   else
      returnValue = search(mid+1, last, n);

   cout << "На выходе: first = " << first << " last = "
        << last << endl;
   return returnValue;
} // Конец функции search

int mystery(int n)
{
   return search(1, n, n);
} // Конец функции mystery
```

10. Изучите следующую функцию, преобразующую положительное десятичное число в восьмеричное представление и выводящую его на печать. Опишите, как работает ее алгоритм. Выполните трассировку функции для n=100.

```
void displayOctal(int n)
{
   if (n > 0)
   {
      if (n/8 > 0)
         displayOctal(n/8);
      cout << n % 8;
   } // Конец оператора if
} // Конец функции displayOctal
```

11. Проанализируйте следующую программу.

```
#include <iostream.h>
int f(int n);

int main()
```

```
{
   cout << "Значение f(8) равно " << f(8) << endl;
   return 0;
}  // Конец функции main

int f(int n)
// Предусловие: n >= 0.
{
   cout << "Функция вызвана для значения n = " << n << endl;
   switch(n)
   {
      case 0: case 1: case 2:
         return;
      default:
         return f(n-2) * f(n-4);
   } // Конец оператора switch
} // Конец функции a
```

Что будет выведено на печать в результате выполнения этой программы? Какие значения аргументов можно передать функции *f*, чтобы ее выполнение никогда не завершилось, и можно ли это сделать вообще?

12. Проанализируйте следующую программу.

```
void recurse(int x, int y)
{
   if (y > 0)
   {
      ++x;
      --y;
      cout << x << " " << y << endl;
      recurse(x, y);
      cout << x << " " << y << endl;
   } // Конец оператора if
} // Конец функции recurse
```

Выполните эту функцию при $x = 5$ и $y = 3$. Как изменится вывод, если аргумент *x* будет передаваться по ссылке, а не по значению?

13. Выполните с помощью блок-схем трассировку функции *binarySearch*, описанной в разделе "Бинарный поиск", для массива, состоящего из чисел 1, 5, 9, 12, 15, 21, 29, 31. Искомыми являются следующие значения.

13.1. 5

13.2. 13

13.3. 16

14. Представьте, что у вас есть 101 далматинец. Никакие два далматинца не имеют одинакового количества пятен. Допустим, вы создали массив из 101 целого числа. Первое число представляет собой количество пятен у первого далматинца, второе — у второго и т.д. Ваш друг захотел узнать, есть ли у вас далматинец с 99 пятнами. Таким образом, нужно определить, содержится ли в массиве число 99.

14.1. Если вы собираетесь применить бинарный поиск числа 99, что нужно сделать с массивом прежде всего (и надо ли что-то делать)?

14.2. Какой элемент массива при бинарном поиске проверяется первым?

14.3. Если у всех ваших далматинцев больше, чем 99 пятен, сколько сравнений потребуется, чтобы обнаружить, что числа 99 в массиве нет?

15. В этой задаче рассматриваются несколько способов вычисления функции x^n при некотором $n \geq 0$.

15.1. Напишите итеративную функцию *power1* для вычисления значения xn при некотором $n \geq 0$.

15.2. Напишите рекурсивную функцию *power2* для вычисления значения xn, используя следующее рекурсивное определение:

$x^0 = 1$,

$x^n = x * x^{n-1}$, если $n > 0$.

15.3. Напишите рекурсивную функцию *power3* для вычисления значения x^n, используя следующее рекурсивное определение:

$x^0 = 1$,

$x^n = (x^{n/2})^2$, если $n > 0$ и n — четное число,

$x^n = x * (x^{n/2})^2$, если $n > 0$ и n — нечетное число.

15.4. Сколько умножений выполняется в функциях *power1*, *power2* и *power3* при вычислении значений 3^{32} и 3^{19}?

15.5. Сколько рекурсивных вызовов выполняется в функциях *power2* и *power3* при вычислении значений 3^{32} и 3^{19}?

16. Модифицируйте рекурсивную функцию *rabbit* так, чтобы ее выполнение было легко проследить визуально. Вместо вывода сообщения "На входе:" и "На выходе:" вставьте оператор, выводящий глубину текущего рекурсивного вызова. Например, при вызове *rabbit*(4) на экран будет выведена следующая информация.

```
На входе rabbit:  n = 4
  На входе rabbit:  n = 3
      На входе rabbit:  n = 2
      На выходе rabbit: n = 2  value = 1
      На входе rabbit:  n = 1
      На выходе rabbit: n = 1  value = 1
  На выходе rabbit: n = 3  value = 2
  На входе rabbit:  n = 2
  На выходе rabbit: n = 2  value = 1
На выходе rabbit: n = 4  value = 3
```

Сравните эту информацию с рис. 2.11.

17. Проанализируйте следующее рекурсивное соотношение:

$f(1)=1$; $f(2)=1$; $f(3)=1$; $f(4)=3$; $f(5)=5$;

$f(n)=f(n-1)+3*f(n-5)$ для всех $n > 5$.

17.1. Вычислите функцию $f(n)$ для следующих значений n: 6, 7, 12, 15.

17.2. Если вы проявили осторожность и не стали вычислять величину $f(15)$ с самого начала (как это могла бы сделать рекурсивная функция на языке C++), то могли вычислить последовательно $f(6)$, $f(7)$, $f(8)$, а затем — $f(15)$, выводя на экран вычисленные значения. Это позволяет снизить количество вычислений. (Напомним, что итеративная версия программы *rabbit* обсуждается в конце главы.)

Обратите внимание, что во время вычислений нет необходимости хранить все ранее вычисленные значения — только последние пять. Учитывая этот факт, напишите на языке C++ функцию, вычисляющую значение $f(n)$ для произвольного числа n.

18. Напишите итеративную версию рекурсивных функций `fact`, `writeBackward`, `binarySearch` и `kSmall`.

19. Используя инварианты, докажите, что функция `iterativeRabbit`, описанная в разделе "Рекурсия и эффективность", работает правильно.

20. Проанализируйте задачу вычисления **наибольшего общего делителя** (gcd — greatest common divisor) двух положительных чисел a и b. Описанный ниже алгоритм представляет собой один из вариантов алгоритма Евклида, основанного на следующей теореме.[4]

    Теорема. Если a и b — положительные целые числа, причем $a>b$ и число b не является делителем числа a, то $\gcd(a, b)=\gcd(b, a \bmod b)$.

    Это соотношение между $\gcd(a, b)$ и $\gcd(b, a \bmod b)$ лежит в основе рекурсивного решения. Оно позволяет свести вычисление значения $\gcd(a, b)$ к аналогичной задаче меньшего размера. Кроме того, если число b является делителем числа a, то $b=\gcd(a, b)$, поэтому в качестве базиса можно выбрать соотношение $(a \bmod b) = 0$.

    Эта теорема приводит к следующему рекурсивному определению:

$$gcd(a,b) = \begin{cases} b, если\ (a \bmod b) = 0, \\ gcd(b, a \bmod b), если\ (a \bmod b) \neq 0. \end{cases}$$

    Этот рекурсивный алгоритм реализуется с помощью функции на языке C++.

```
int gcd(int a, int b)
{
   if (a % b == 0) // Базис
      return a;
   else
      return gcd(b, a % b);
} // Конец функции gcd
```

    20.1. Докажите теорему.

    20.2. Что произойдет, если $b>a$?

    20.3. Как уменьшить размер задачи? (Всегда ли можно достичь базиса?) Почему выбранный базис является правильным?

21. Пусть $c(n)$ — количество разных групп целых чисел, которые можно выбрать из чисел от 1 до n-1, так чтобы сумма всех чисел в группе равнялась n (например, 4=1+1+1+1=1+1+2=...=2+2). Напишите рекурсивные определения для вычисления числа $c(n)$ при следующих ограничениях.

    21.1. С учетом перестановок. Например, группы чисел 1, 2, 1 и 1, 1, 2 считаются разными.

    21.2. Без учета перестановок.

22. Проанализируйте следующее рекурсивное определение:

---

[4] Математическая операция modulo (деление по модулю) в книге обозначается как mod. В языке C++ целочисленное деление обозначается символом %.

$$Acker(m,n) = \begin{cases} n+1, если\ m = 0, \\ Acker(m-1,1), если\ n = 0, \\ Acker(m-1, Acker(m, n-1)), если\ m \neq 0\ и\ n \neq 0. \end{cases}$$

Эта функция, называемая **функцией Аккермана** (Ackerman's function), интересна тем, что она быстро растет с увеличением аргументов m и n. Чему равно значение $Acker(1, 2)$? Реализуйте эту функцию на языке C++ и выполните трассировку вызова $Acker(1, 2)$ с помощью блок-схем. (*Внимание*: даже при средних значениях m и n функция Аккермана порождает *много* рекурсивных вызовов.)

Задания по программированию

1. Реализуйте на языке C++ функцию `maxArray`, рассмотренную в разделе "Поиск *k-го* наименьшего элемента массива". Какое еще рекурсивное определение допускает эта функция?
2. Реализуйте на языке C++ функцию `kSmall`, рассмотренную в разделе "Поиск *k-го* наименьшего элемента массива", используя первый элемент массива в качестве опорного.

Абстракция данных: стены

В этой главе ...

Абстрактные типы данных

Спецификации абстрактных типов данных
- Абстрактный список
- Абстрактный упорядоченный список
- Разработка абстрактных типов данных
- Аксиомы

Реализация абстрактных типов данных
- Классы языка C++
- Пространства имен
- Реализация абстрактного списка в виде массива
- Исключительные ситуации в языке C++
- Реализация абстрактного списка с учетом исключительных ситуаций

Резюме

Предупреждения

Вопросы для самопроверки

Упражнения

Задания по программированию

Введение. В этой главе детально изучается абстракция данных, введенная в главе 1, как способ повышения модульности программы. Абстракция данных позволяет возвести "стены" между программой и структурами данных. При решении задач нужно выполнять разные операции над данными, поэтому возникает необходимость определить абстрактные типы данных (АТД). На примере нескольких простых абстрактных типов данных в главе демонстрируются преимущества АТД в целом. Другие важные абстрактные типы данных рассматриваются в части II.

К реализации структур данных можно приступать только после того, как станет ясно, какие операции должны производиться над АТД. В главе рассматриваются вопросы их реализации с помощью классов языка C++.

Абстрактные типы данных

Модульный подход к программированию позволяет сохранить контроль над большими и сложными программами, систематически управляя взаимодействием между их составными частями. Это позволяет сосредоточиться на решении отдельной задачи, не отвлекаясь на другие. Таким образом, модульную программу легче писать, читать и модифицировать. Модульность программы позволяет локализовать ошибки, а также исключить избыточный код.

Модульную программу легче писать, читать и модифицировать

Модульные программы можно создавать, объединяя в одно целое уже готовые компоненты программного обеспечения и вновь написанные функции. При этом следует сосредоточивать внимание на том, *что* именно делает модуль, а не на том, *как* он это делает. Для того чтобы успешно использовать ранее разработанное программное обеспечение, нужно иметь набор четких спецификаций, описывающих детали поведения этих модулей. Чтобы написать новые функции, нужно решить, для чего они предназначены, и определить их взаимодействие с другими частями программы, считая, что эти функции уже существуют и работают. Это позволяет разрабатывать функции в относительной изоляции друг от друга, обращая внимание лишь на то, что они делают, и не вникая в детали их внутреннего устройства. Такой процесс называется **функциональной абстракцией** (functional abstraction).

Перед реализацией каждого модуля записывайте его спецификацию

Формулируя спецификацию модуля, нужно выявить детали, которые можно скрыть от внешнего мира. Принцип **сокрытия информации** (information hiding) подразумевает не только утаивание деталей внутреннего устройства модуля от других частей программы, но и *невозможность доступа* к ним извне. Сокрытие информации ассоциируется со **стенами**, возведенными между разными частями программы. Эти стены предотвращают перепутывание модулей. Стены вокруг модуля *T* скрывают его внутренний мир от "любопытных глаз" других модулей. Таким образом, если модуль *Q* использует модуль *T*, а метод, который реализуется в модуле *T*, в какой-то момент изменился, это никак не повлияет на модуль *Q*. Как показано на рис. 3.1, стены делают модуль *Q* независимым от модуля *T*.

Скрывайте детали внутреннего устройства модуля от других модулей

Однако эта изоляция не может быть абсолютной. Несмотря на то что модуль *Q* не знает, *как* реализован модуль *T*, он должен знать, какую задачу решает модуль *T* и как его вызвать. Допустим, программа должна работать с упорядоченным массивом имен, скажем, искать заданное имя в массиве или выводить на экран имена в алфавитном порядке. Следовательно, программа должна содержать функцию *S*, упорядочивающую массив имен. Несмотря на то что остальным частям программы известно, что функция *S* упорядочивает массив, им абсолютно все равно, как она это делает. Представьте себе, что в каждой стене прорублено крошечное окошко,

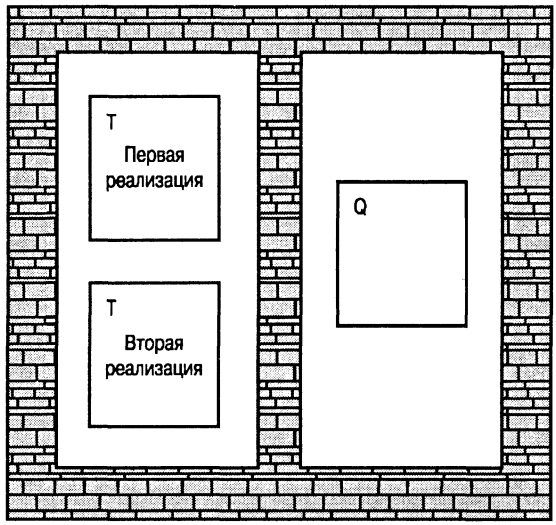


Рис. 3.1. Изолированные модули: реализация модуля T никак не влияет на модуль Q

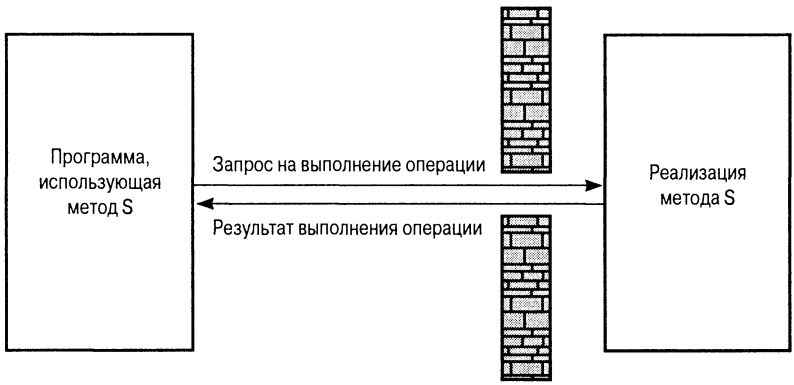


Рис. 3.2. Окошко в стене

как показано на рис. 3.2. Оно невелико и не позволяет увидеть детали внутреннего устройства функции, но достаточно широкое, чтобы обмениваться через него данными. Например, в функцию сортировки через это окошко можно передать массив и получить его обратно уже упорядоченным. То, что функция получает извне, и то, что она возвращает внешнему миру, описывается в терминах ее спецификации, или **контракта** (contract): *в нем указывается, что именно делает функция, и какие условия для этого должны выполняться.*

Типичные операции над данными

При решении задач часто необходимо выполнять операции над данными. В общих чертах, их можно свести к трем разновидностям.

- **Добавление** (add) данных в набор
- **Удаление** (remove) данных из набора
- **Проверка** (ask question) данных, содержащихся в наборе

Разумеется, детали этих операций в разных приложениях варьируются, однако в целом они образуют операции управления данными. Нужно, однако, отдавать себе отчет, что эти операции нужны не для всех задач.

Функциональная абстракция и абстракция данных сводятся к вопросу "что?", а не "как?"

Абстракция данных (data abstraction) описывает, *что* можно сделать с набором данных, игнорируя вопрос "*как* это делается?" Абстракция данных — это способ, позволяющий разрабатывать отдельные структуры данных независимо от остальной части программы. Другие модули программы будут "знать", какие операции можно выполнять над этими данными, но им будет неизвестно, каким образом хранятся данные или как именно выполняются операции. Итак, как и прежде, контракт отвечает на вопрос "что?", а не "как?" Итак, абстракция данных — это естественное расширение функциональной абстракции.

Абстрактный тип данных состоит из данных и набора операций над ними

Набор данных в сочетании с совокупностью операций над ними называется **абстрактным типом данных** (abstract data type), или АТД. Допустим, например, что нам нужно хранить набор имен так, чтобы в нем можно было быстро отыскать заданное имя. Эффективно решить эту проблему позволяет алгоритм бинарного поиска, описанный в главе 2. Таким образом, одно из решений задачи сводится к записи имен в упорядоченный массив и применению алгоритма бинарного поиска заданного имени. *Упорядоченный массив совместно с алгоритмом бинарного поиска* можно рассматривать в качестве абстрактного типа данных, позволяющего решить эту задачу.

В спецификациях указывается, что делают операции АТД, но не описывается их реализация

Описание операций, входящих в АТД, должно быть достаточно строгим, для того чтобы точно указать их воздействие на данные, но в нем не должен указываться способ хранения данных или детали выполнения операций. Например, в описаниях операций не нужно указывать, хранятся данные в смежных ячейках памяти или в разбросанных. Конкретная **структура данных** выбирается только при **реализации** АТД.

Структура данных является частью реализации АТД

Напомним, что структура данных представляет собой конструкцию, определенную в языке программирования для хранения совокупности данных. Например, массивы и структуры, встроенные в язык C++, — это структуры данных. Однако программисты могут создавать свои собственные структуры данных. Допустим, что нам нужна структура данных, в которой одновременно хранятся имена и размер окладов группы сотрудников. Эту структуру можно описать на языке C++ следующим образом.

```
const int MAX_NUMBER = 500;
string names[MAX_NUMBER];
double salaries[MAX_NUMBER];
```

Здесь элемент *names[i]* означает имя сотрудника, а *salaries[i]* — величину его оклада. Два массива *names* и *salaries* образуют структуру данных, однако в языке C++ нет отдельного типа данных, чтобы описать ее.

Тщательно описывайте операции АТД перед их реализацией

Если программа должна выполнять операции, не предусмотренные в языке, сначала нужно разработать абстрактный тип данных, а затем тщательно описать, что именно делают его операции (контракт). Тогда — *и только тогда* — можно приступать к реализации операций над структурой данных. Если операции реализованы правильно, их можно применять и в остальной части программы, т.е. считается, что условия контракта выполнены.

Однако программа не должна зависеть от конкретного способа реализации этих операций.

Абстрактный тип данных — это не синоним структуры данных.

ОСНОВНЫЕ ПОНЯТИЯ

Абстрактные типы данных и структуры данных

- **Абстрактный тип данных — это совокупность данных и операций над ними.**
- **Структура данных — это конструкция, определенная в языке программирования для хранения набора данных.**

Абстрактные типы данных и структуры данных — не одно и то же

Для того чтобы лучше понять разницу между абстрактными типами данных и структурами данных, рассмотрим морозильное устройство, показанное на рис. 3.3. На его вход поступает вода, из которой получается либо охлажденная вода, либо измельченный лед, либо кубики льда, в зависимости от того, на какую кнопку нажать. Кроме того, имеется индикатор, который загорается, если льда внутри нет. Это устройство представляет собой пример абстрактного типа данных. Аналогом данных является вода, а операциями — "*охладить*", "*измельчить*", "*наколоть*" и запрос "*пусто*?" На этом уровне проектирования нас не интересует, как именно морозильное устройство выполняет указанные операции. Если мы хотим измельчить лед, зачем нам вдаваться в технические тонкости устройства холодильника, если он и так работает правильно? Таким образом, описав функции, выполняемые морозильным устройством, операции, использующие измельченный лед, можно применять, не зная, как он получается.

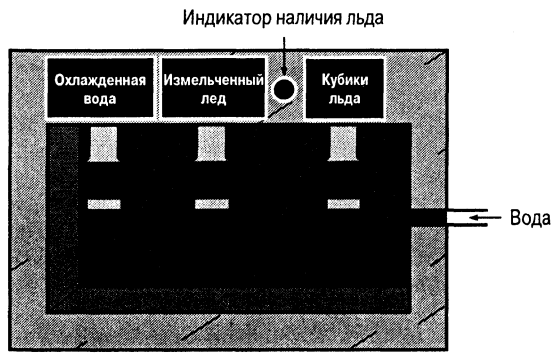


Рис. 3.3. Морозильное устройство для получения охлажденной воды, измельченного льда и кубиков льда

Однако в конце концов кто-то же должен создать морозильное устройство. Как все-таки получается измельченный лед? Сначала нужно сделать кубик льда, а затем измельчить его с помощью двух стальных роликов, или разбить молотком на мелкие кусочки. Для этого можно придумать много способов. Внутренняя структура морозильного устройства соответствует реализации абстрактного типа данных в языке программирования, т.е. структуре данных.

Несмотря на то что владельца холодильника не интересуют технические подробности, он или она хотят, чтобы устройство работало как можно эффективнее. Аналогично, производитель холодильника хотел бы, чтобы его производство бы-

ло простым и как можно более дешевым. Те же самые понятия используются при выборе структуры данных для реализации абстрактного типа данных в языке C++. Даже если вы не реализуете АТД сами, а используете уже готовые компоненты, вы как и человек, купивший холодильник, должны стремиться, по крайней мере, к эффективности.

Обратите внимание на то, что морозильное устройство окружено стальными стенками. Именно они позволяют вводить в машину входные данные (воду) и получать результаты (охлажденную воду, измельченный лед или кубики льда). Таким образом, внутренний механизм устройства не только скрыт от пользователя, но и недоступен. Кроме того, механизм выполнения одной операции недоступен для другой.

Такой модульный подход имеет преимущества. Например, операцию "измельчить" можно усовершенствовать, изменив ее модуль и не вмешиваясь в работу остальных механизмов. Кроме того, можно добавить новую операцию, подключив к машине новый модуль и оставив первые три операции без изменения. Таким образом, в холодильнике применяется и абстракция, и принцип сокрытия информации.

Итак, абстракция данных возводит из операций АТД стены между структурами данных и программой, обращающейся к ним, как показано на рис. 3.4. Если вы находитесь на стороне программы, то видите **интерфейс** (interface), позволяющий взаимодействовать со структурой данных. Через интерфейс пользователь передает запрос на выполнение операций АТД (манипуляции со структурой данных), а в ответ получает их результаты.

> Программа не должна зависеть от деталей реализации абстрактных типов данных

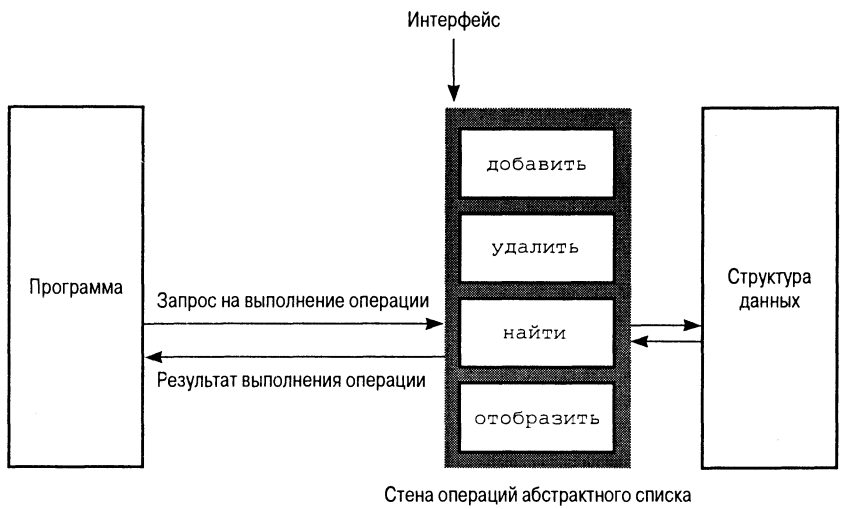


Рис. 3.4. Стены из операций АТД, отделяющие структуры данных от использующей их программы

Этот процесс напоминает использование торговых автоматов. Вы нажимаете кнопки и получаете нечто в ответ. Внешний дизайн автомата подсказывает вам, что нужно делать, так же как спецификации АТД описывают операции и их предназначение. Раз вы пользуетесь подсказками автомата, технические детали его внутреннего устройства становятся для вас безразличными. Аналогично, согласившись на доступ к данным только через операции АТД, можно "забыть" о любых изменениях структур данных, реализующих этот АТД.

Абстрактные типы данных напоминают торговые автоматы

На следующих страницах мы покажем, как использовать абстрактные типы данных для отделения операций над данными от реализации этих операций. Для этого мы рассмотрим несколько примеров АТД.

Спецификации абстрактных типов данных

Для того чтобы уточнить понятие абстрактных типов данных, рассмотрим список, в котором перечисляются неотложные дела, важные даты, адреса или продукты, как показано на рис. 3.5. Куда вы записываете новые пункты, когда заполняете список? Допустим, что все записи расположены одна под другой. Тогда, вероятно, новую запись вы добавляете в конец списка. Кроме того, вы можете вставлять новую запись в начало списка или так, чтобы в нем сохранялся алфавитный порядок. Независимо от этого список представляет собой последовательность записей. У него есть первый и последний элементы. За исключением этих элементов, у всех остальных есть единственный **предшествующий элемент**, или **предшественник** (predessor), и единственный **последующий элемент**, или **преемник** (successor). Первый элемент — **голова** (head), или **начало** (front) списка — не имеет предшественника, а последний элемент — **хвост** (tail), или **конец** (end) списка — не имеет преемника.

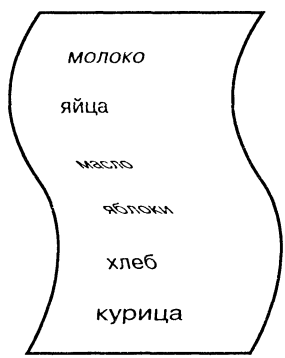


Рис. 3.5. Список продуктов

Списки состоят из однотипных элементов. Список может состоять либо из бакалейных продуктов, либо из номеров телефонов. Что можно делать с элементами списка? Их можно пересчитывать, вычисляя длину списка, добавлять в список, удалять из него, просматривать (**извлекать** (retreave)). Элементы списка вместе с операциями над ними образуют абстрактный тип данных. В этом определении важно, что нас интересует лишь предназначение операций, а не детали их реализации. Это позволяет не уточнять конкретные структуры данных, которые подразумеваются, когда речь идет о списках.

Куда добавить новый элемент и какой элемент вы хотите просмотреть? Разные ответы на эти вопросы приводят в разным видам списков. Можно добавлять, удалять и извлекать элементы только из конца списка, либо только из начала, либо из конца и начала. Читатели могут сами создать спецификации этих списков. Далее мы обсудим самый общий вид списков.

Абстрактный список

Вернемся к списку продуктов, изображенному на рис. 3.5. Описанные выше списки, в которых манипуляции над элементами производились либо в начале, либо в конце, либо и в начале, и в конце, не соответствуют реальному списку продуктов. Нам может понадобиться доступ к любому элементу списка. Это значит, что мы можем просматривать элемент, находящийся на позиции *i*, удалять его или вставлять новый элемент в эту позицию. Эти операции являются частью абстрактного типа данных под названием **список** (list).

ОСНОВНЫЕ ПОНЯТИЯ

Операции над абстрактным списком

1. Создать пустой список.
2. Уничтожить список.
3. Определить, пуст ли список.
4. Определить количество элементов в списке.
5. Вставить элемент в указанную позицию списка.
6. Удалить элемент, находящийся в указанной позиции списка.
7. Просмотреть (извлечь) элемент, находящийся в указанной позиции списка.

Несмотря на то что шесть элементов списка, изображенного на рис. 3.5, перечислены в последовательном порядке, их не обязательно упорядочивать по именам. Список может заполняться по мере покупки продуктов, а чаще всего — как попало. Абстрактный список представляет собой упорядоченный набор элементов, каждый из которых имеет свой номер.

Для обращения к элементу списка используется номер его позиции

В приведенном ниже псевдокоде операции над абстрактным списком описываются более подробно. На рис. 3.6 показана диаграмма абстрактного списка на языке UML.

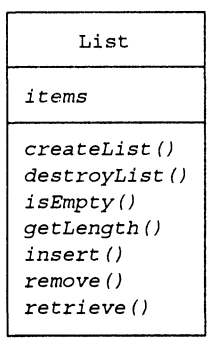


Рис. 3.6. Диаграмма абстрактного списка на языке UML

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операций над абстрактным списком

```
//Элементы списка имеют тип ListItemType.

+createList()
// Создает пустой список.

+destroyList
// Уничтожает список.

+isEmpty():boolean {query}
// Определяет, пуст ли список.

+getLength():integer {query}
// Возвращает количество элементов, содержащихся в списке.

+insert(in index:integer, in newItem:ListItemType,
 out success:boolean)
// Вставляет элемент newItem в список на позицию index,
// если 1<= index <= getLength()+1.
// Если index <= getLength(), элементы перенумеровываются
// в следующем порядке: элемент с позиции index перемещается
// на позицию index+1, элемент с позиции index+1 перемещается
// на позицию index+2 и т.д. Признак success отмечает, успешно ли
// выполнена вставка.

+remove(in index:integer, out success:boolean)
// Удаляет из списка элемент, находящийся на позиции index,
// если 1<= index <= getLength()+1.Если index <= getLength(),
// элементы перенумеровываются в следующем порядке:
// элемент с позиции index перемещается на позицию index+1,
// элемент с позиции index+1 перемещается на позицию index+2
// и т.д. Признак success отмечает, успешно ли выполнено
// удаление.

+retrieve(in index:integer, out dataItem:ListItemType,
 out success:boolean)
// Копирует элемент, находящийся на позиции index, в переменную
// dataItem, если 1<= index <= getLength()+1. Эта операция
// не изменяет список. Признак success отмечает, успешно ли
// выполнено извлечение элемента.
```

Чтобы точнее понять, как работают эти операции, применим их к списку продуктов

молоко, яйца, масло, яблоки, хлеб, курица,

где первым элементом является запись "*молоко*", а последним — запись "*курица*". Для начала попытаемся создать этот список. Сначала можно создать пустой список *aList*, а затем последовательно применить операции вставки, добавляя элементы следующим образом.

```
aList.createList()
aList.insert(1, milk, success)
aList.insert(2, eggs, success)
```

```
aList.insert(3, butter, success)
aList.insert(4, apples, success)
aList.insert(5, bread, success)
aList.insert(6, chicken, success)
```

Запись[1] `aList.O` означает, что операция `O` применяется к списку `aList`.

Обратите внимание, что операция вставки позволяет вставлять новые элементы в любое место списка, а не только в его начало или конец. В соответствии со спецификацией операции `insert`, если новый элемент вставляется на позицию `i`, то индекс каждого элемента, находящегося правее этого места, увеличивается на единицу. Таким образом, например, если к приведенному выше списку применяется операция

```
aList.insert(4, nuts, success),
```

список принимает вид

молоко, яйца, масло, орехи, яблоки, хлеб, курица.

Все элементы, индекс которых перед вставкой был больше или равен четырем, сдвигаются вправо, поскольку их индексы увеличиваются на единицу.

Аналогично, операция удаления означает, что из списка удаляется элемент с индексом `i`, а индекс каждого элемента, находящегося правее, уменьшается на единицу. Таким образом, например, если перед удалением список `aList` имел вид

молоко, яйца, масло, орехи, яблоки, хлеб, курица

и к нему применяется операция

```
aList.remove(5, success),
```

то список станет следующим:

молоко, яйца, масло, орехи, хлеб, курица.

Все элементы, индекс которых перед удалением был больше или равен 5, сдвигаются влево, поскольку их индексы уменьшаются на единицу.

Спецификация абстрактного типа не должна зависеть от его реализации

Эти примеры показывают, что спецификация абстрактного типа данных описывает результаты операций, но не указывает способ хранения данных. Спецификации указанных выше семи операций сформулированы исключительно в терминах контракта абстрактного списка: при выполнении этой операции произойдет то-то и то-то. В спецификациях не упоминается, как именно хранятся данные или каким образом выполняются операции. Они указывают лишь, что можно делать со списком.

Программа должна зависеть только от содержания операции

Принципиально важно, что спецификация абстрактного типа *не затрагивает* вопросов его реализации. Именно это ограничение позволяет воздвигать стены между реализацией АТД и программой, использующей его. (Такая программа называется клиентом (client).) Единственным обстоятельством, влияющим на выполнение программы, является содержание самой операции.

Обратите внимание, что у операций вставки, удаления и извлечения есть аргумент `success`, позволяющий абстрактному типу данных сообщать клиенту о неудачном выполнении операции. Например, при попытке удалить десятый элемент из списка, состоящего из пяти записей, операция `remove` присвоит аргументу `success` значение `false`. Аналогично, операция `insert` присвоит аргументу `success` значение `false`, если список полон или параметр `index` выходит

[1] Эта запись совместима с дальнейшей реализацией абстрактного типа данных на языке C++.

за пределы диапазона допустимых значений. Таким образом, параметр *success* позволяет клиенту обрабатывать ошибочные ситуации независимо от способа реализации абстрактного типа.

Какую информацию о поведении абстрактного типа данных содержит спецификация? Очевидно, что операции над списком распадаются на три категории, уже упоминавшиеся ранее в этой главе.

- Операция *insert* **добавляет** данные в набор.
- Операция *remove* **удаляет** данные из набора.
- Операции *isEmpty*, *getLength* и *retrieve* **выполняют запрос** относительно данных, содержащихся в наборе.

Если поведение абстрактного типа описано правильно, можно приступать к разработке приложения, манипулирующего его данными, пользуясь лишь названиями операций и не вникая в детали их реализации. Допустим, к примеру, что мы хотим вывести на экран элементы списка. Несмотря на то что стены, отделяющие реализацию абстрактного списка от остальной программы, скрывают способ его хранения, можно написать функцию *displayList*, применяя операции, определенные для этого типа. Псевдокод такой функции приведен ниже.[2]

Приложение, не зависящее от реализации абстрактного списка

```
// Выводит на экран элементы списка
// aList.
for (position = 1 до aList.getLength())
{
  aList.retrieve(position, dataItem, success)
    Вывести на экран dataItem
} // Конец цикла for
```

Поскольку абстрактный список реализован правильно, функция *displayList* будет успешно выполнена. В этом случае функция *retrieve* сможет извлечь любой элемент списка, так как значение параметра *position* всегда корректно, следовательно, аргументом *success* можно пренебречь.

Функция *displayList* не зависит от реализации списка. Это означает, что функция будет успешно работать независимо от структуры данных, использованной для хранения списка. Это свойство представляет собой очевидное преимущество абстрактных типов. Кроме того, используя лишь названия операций, можно игнорировать технические детали. На рис. 3.7 изображена стена между функцией *displayList* и реализацией абстрактного списка.

Рассмотрим другое приложение, использующее операции над абстрактным списком. Допустим, нужна функция *replace*, заменяющая элемент, находящийся на позиции *i* новым элементом. Если *i*-й элемент существует, функция *replace* удаляет его и вставляет на эту позицию новый элемент.

```
replace(in aList:List, in i:integer,
        in newItem:ListItemType, out success:boolean)
// Заменяет i-й элемент списка aList элементом newItem.
// Признак success отмечает, успешно ли выполнена замена.

aList.remove(i, success)
if (success)
   aList.insert(i, newItem, success)
```

[2] В этом примере функция *displayList* не является операцией над абстрактным списком, поэтому список передается ей через аргумент *aList*.

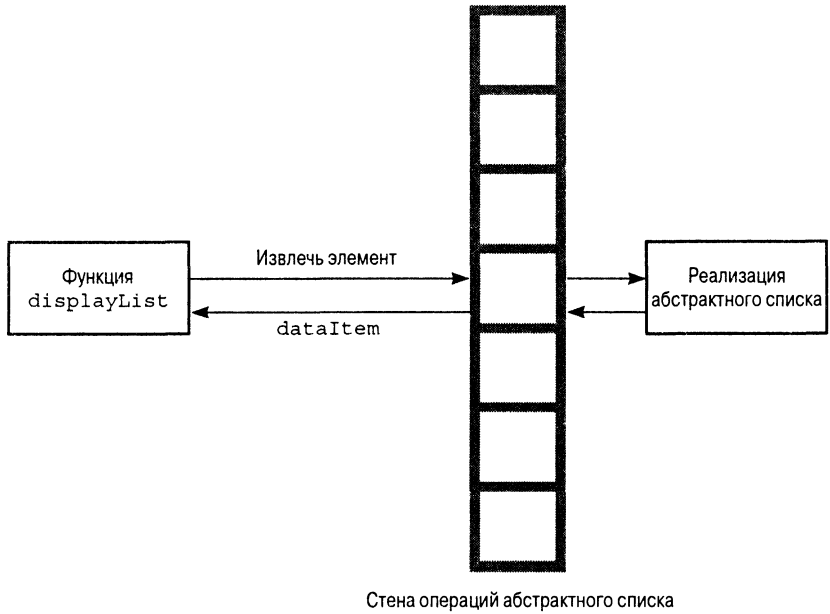


Рис. 3.7. Стена между функцией displayList и реализацией абстрактного списка

Если удаление выполнено успешно, функция `remove` присвоит параметру `success` значение `true`. Проверив значение этого параметра, функция `replace` приступит к вставке, только если удаление действительно произошло. Затем функция `insert` присваивает параметру `success` значение, которое было возвращено функцией `replace`. Если функция `remove` по какой-то причине не смогла выполнить свое задание, например, если значение параметра `i` было задано некорректно, она присвоит параметру `success` значение `false`. В этом случае функция `replace` проигнорирует операцию вставки и вернет значение параметра `success`.

Операции абстрактного типа данных можно применять, не вникая в детали их реализации

В обоих примерах, описанных выше, мы не вникали в детали реализации списка. Нам было все равно, хранятся ли его записи в массиве или в другой структуре данных. Использование операций абстрактного типа данных, таких как `insert` и `replace`, снижает риск появления ошибок, освобождая программиста от необходимости учитывать технические тонкости. Это относится и к созданию программ на языке C++. Кроме того, поскольку операции `insert` и `replace` не зависят от реализации типа, их содержание никак не изменяется. Следовательно, при реализации абстрактного типа данных нет никакой необходимости изменять его спецификацию. Однако, как указывалось в главе 1, разработка программного обеспечения — это не линейный процесс. В процессе реализации абстрактного типа данных его разработчик может прийти к выводу, что спецификацию нужно изменить. Очевидно, любое изменение спецификации какого-либо модуля влечет за собой модификацию всех модулей, использующих его в своей работе.

Итак, функционирование абстрактного типа данных можно определять независимо от его реализации. Имея такую спецификацию и ничего не зная о том, как именно будет реализован АТД, можно приступать к разработке приложения, применяющего операции АТД для доступа к его данным.

Абстрактный упорядоченный список

Одной из наиболее широко распространенных задач является создание и поддержка *упорядоченных* наборов данных. На ум немедленно приходит множество примеров: студенты, рассаженные согласно именам, футболисты, перечисленные по номерам их футболок, корпорации, названные по величине их активов. Это — примеры **упорядоченных** (sorted) множеств. В каждом случае важен принцип, по которому упорядочивается список. Так, список продуктов, приведенный ранее, можно считать упорядоченным, если его записи соответствуют порядку, в котором продукты снимались с полок магазина, однако, если учитывать лишь названия продуктов, то список оказывается неупорядоченным.

Поддержка упорядоченных данных не сводится к их простой сортировке. Часто возникает необходимость вставить новый элемент в уже упорядоченный список. Кроме того, из упорядоченного списка иногда нужно удалить некий элемент. Допустим, например, что в деканате хранятся списки студентов, перечисленных в алфавитном порядке. Регистратор должен вносить в них имена вновь поступивших студентов и удалять оттуда имена выпускников, причем эти операции не должны нарушать алфавитный порядок, установленный в этих списках.

В абстрактном упорядоченном списке элементы перечислены в определенном порядке

Спецификации операций над **абстрактным упорядоченным списком приведены ниже.**

Абстрактный упорядоченный список отличается от обычного абстрактного списка, поскольку вставка и удаление его элементов зависят от их значений, а не индексов. Например, операция `sortedInsert` определяет позицию для вставки элемента `newItem` по его значению. Кроме того, у него есть новая операция, `locatePosition`, определяющая индекс любого элемента по заданному значению. В то же время операции `sortedRetrieve` и `retrieve` совершенно аналогичны: обе они извлекают заданный элемент по его индексу. Операция `sortedRetrieve` позволяет, например, написать другую функцию, извлекающую элемент из упорядоченного списка и выводящую его на экран.

Разработка абстрактных типов данных

Разработка абстрактных типов данных должна естественным образом развиваться на протяжении всего процесса решения задачи. В качестве примера рассмотрим задачу, в которой нужно определить даты всех праздников, которые будут отмечаться в текущем году. Таким образом, мы должны проанализировать каждый день в году и установить, является ли он праздничным. Одно из возможных решений этой задачи описывается следующим псевдокодом.

```
listHolidays(in year:integer)
// Выводит на экран все праздничные даты в заданном году.
date = дата первого дня в заданном году
while (date предшествует первому дню следующего года year+1)
{
   if (date — праздничная дата)
   write (date," — праздничная дата ")
   date = дата следующего дня
}// Конец цикла while
```

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операций над абстрактным упорядоченным списком

```
//Элементы списка имеют тип ListItemType.

+createSortedList()
// Создает пустой упорядоченный список.

+destroySortedList
// Уничтожает упорядоченный список.

+sortedIsEmpty():boolean {query}
// Определяет, пуст ли упорядоченный список.

+sortedGetLength():integer {query}
// Возвращает количество элементов, содержащихся
// в упорядоченном списке.

+sortedInsert(in newItem:ListItemType, out success:boolean)
// Вставляет элемент newItem в соответствующую позицию
// упорядоченного списка. Признак отмечает, успешно ли
// выполнена вставка.

+sortedRemove(in anItem:ListItemType, out success:boolean)
// Удаляет из упорядоченного списка элемент anItem.
// Признак success отмечает, успешно ли выполнено удаление.

+sortedRetrieve(in index:integer, out dataItem:ListItemType,
 out success:boolean) {query}
// Копирует элемент, находящийся на позиции index в переменную
// dataItem, если 1<= index <= sortedGetLength()+1. Эта операция
// не изменяет список. Признак success отмечает,
// успешно ли выполнено извлечение элемента.

+locatePosition(in anItem:ListItemType,
 isPresent:boolean):integer {query}
// Возвращает индекс элемента в упорядоченном списке.
// Признак isPresent отмечает, содержится ли элемент anItem
// в списке. Элемент anItem и список не изменяются.
```

Какие данные рассматриваются в задаче?

В этой задаче рассматриваются даты, состоящие из месяца, дня и года. Какие операции необходимо выполнить, чтобы решить поставленную задачу? Очевидно, нам нужно определить операции над датами так, как это делается, например для целых чисел. Анализируя приведенный выше псевдокод, легко увидеть, что нам потребуются следующие операции.

Какие операции необходимо выполнить, чтобы решить поставленную задачу?

- Определить дату первого дня заданного года.
- Определить, предшествует ли одна дата другой.
- Определить, является ли дата праздничной.
- Определить дату следующего дня.

Итак, для абстрактного типа данных можно определить следующие операции.

```
+firstDay(in year:integer):Date {query}
// Возвращает дату первого дня заданного года.

+isBefore(in date1:Date,
in date2:Date) : boolean {query}
// Возвращает значение true, если дата date1 предшествует
// дате date2, в противном случае возвращает значение false.

+isHoliday(in aDate:Date) : boolean {query}
// Возвращает значение true, если дата является праздничной,
// в противном случае возвращает значение false.

+nextDay(in aDate:Date) : Date {query}
// Возвращает дату следующего дня.
```

Тогда псевдокод функции *listHoliday* примет такой вид.

```
listHolidays(in year:integer)
// Выводит на экран все праздничные даты в заданном году.

date = firstDay(year)
while (isBefore(date, firstDay(year+1)))
{
   if (isHoliday(date))
   write (date," — праздничная дата ")
   date = nextDay(date)
}// Конец цикла while
```

Итак, абстрактные типы данных можно разрабатывать, идентифицируя данные и выбирая для них подходящие операции. Указанные операции используются для решения поставленной задачи независимо от деталей реализации АТД.

Записная книжка. Рассмотрим еще один пример разработки абстрактного типа данных. Представьте себе, что мы создаем компьютерный вариант книги для записи деловых встреч, охватывающий один год. Допустим, что деловую встречу можно назначать только с 8 утра до 5 вечера. Мы хотим, чтобы система хранила краткое описание деловой встречи, а также ее дату и время.

Для того чтобы решить эту проблему, можно определить абстрактный тип данных "Записная книжка". К данным этого АТД относятся дата, время и цель встречи. Какие операции можно выполнять с данными этого типа? Очевидно, нужно предусмотреть следующие две операции.

- Назначить встречу на определенную дату и время, указав ее цель. (Следует быть осторожным, чтобы не назначить встречу на уже занятое время.)
- Отменить встречу, назначенную на определенную дату и время.

В дополнение к этим операциям можно предусмотреть еще несколько операций.

- Запросить, назначена ли встреча на заданное время.
- Определить цель встречи, назначенной на заданное время.

Кроме того, в абстрактных типах данных обычно определяются операции инициализации и уничтожения.

Таким образом, АТД "Записная книжка" может иметь следующие операции.

```
+createAppointmentBook()
// Создает пустую записную книжку.

+isAppointment(in apptDate:Date,
 in apptTime:Time) : boolean {query}
// Возвращает значение true, если на заданные дату и время
// уже назначена другая встреча; в противном случае возвращает
// значение false.

+makeAppointment(in apptDate:Date, in apptTime:Time,
 in purpose:string) : boolean
// Вставляет записи о деловых встречах, указывая дату, время
// и цель в переменных apptDate, apptTime и purpose,
// если на это время не назначена другая встреча.
// Если операция выполнена успешно, возвращает значение true,
// в противном случае возвращает значение false.

+cancelAppointment(in apptDate:Date,
 in apptTime:Time) : boolean
// Удаляет запись о встрече, назначенной на дату и время,
// заданные параметрами apptDate and apptTime.
// Если операция выполнена успешно, возвращает значение true,
// в противном случае возвращает значение false.

+checkAppointment(in apptDate:Date, in apptTime:Time,
 out purpose:string) {query}
// Извлекает запись о цели встречи по заданным значениям
// параметров apptDate/apptTime, если она существует.
// В противном случае аргументу purpose присваивается
// пустая строка.
```

Эти операции можно использовать при разработке других операций над записной книжкой. Например, допустим, что нам понадобилось изменить дату и время конкретной встречи в записной книжке *apptBook*. Приведенный ниже псевдокод иллюстрирует решение этой задачи с помощью описанных ранее операций.

```
// Изменить дату и время встречи
read (oldDate, oldTime, newDate, newTime)

// Установить цель встречи
apptBook.checkAppointment(oldDate, oldTime, oldPurpose)
if (oldPurpose — не пустая строка)
{
   // Проверить, свободны ли дата и время,
   // определенные параметрами date и time
   if (apptBook.isAppointment(newDate, newTime))
   // Новые дата и время, заданные параметрами
   // date и time, уже зарегистрированы
   write ("На дату ", newTime, "и время ", newDate,
          "уже назначена другая встреча")
   else // Дата и время, заданные параметрами date и time,
        // свободны
   {
     apptBook.cancelAppointment(oldDate, oldTime)
     if (apptBook.makeAppointment(newDate, newTime, oldPurpose))
```

```
        write("Встреча назначена на ", newTime, NewDate)
    } // Конец раздела else
} // Конец оператора if
else
    write ("Время ", oldTime, oldDate, "свободно")
```

Абстрактный тип данных можно применять, ничего не зная о деталях его реализации

Еще раз обратите внимание на то, что приложения, использующие операции над абстрактными типами данных, можно разрабатывать, ничего не зная о деталях реализации АТД. Другие примеры решения задач с помощью АТД приведены в упражнениях.

АТД на основе других АТД. В обоих примерах, рассмотренных выше, нужно было работать с датами. В примере, связанном с записной книжкой, к дате добавилось время. В языке C++ есть структура, позволяющая хранить время и дату. Она определена в файле *time.h*. Кроме того, можно разрабатывать собственные абстрактные типы данных, позволяющие представлять эти элементы в объектно-ориентированном виде. На практике часто приходится определять один АТД через другой. В одном из заданий по программированию, приведенном в конце этой главы, читателям предлагается реализовать свой собственный АТД для работы с датой и временем.

АТД можно использовать при реализации другого АТД

В последнем примере описывается АТД, для реализации которого используются другие АТД. Допустим, нам нужно разработать базу данных для хранения рецептов. Эту базу можно считать абстрактным типом данных. Рецепты представляют собой данные, к которым можно применять следующие операции.

```
+insertRecipe(in aRecipe:Recipe, out success:boolean)
// Вносит рецепт в базу данных.

+deleteRecipe(in aRecipe:Recipe, out success:boolean)
// Удаляет рецепт из базы данных.

+retrieveRecipe(in name:string, out aRecipe:Recipe,
 out success:boolean) {query}
// Извлекает указанный рецепт из базы данных.
```

На этом этапе проектирования не указываются никакие подробности, например, где именно операция *insertResipe* размещает рецепт.

Теперь представьте себе, что нам нужно написать функцию, уточняющую рецепт, полученный из базы данных. Например, рецепт может быть рассчитан на n человек, а нам нужно уточнить его для m персон. Допустим, в рецепт входят разные измерения, скажем, $2^1/_2$ стакана, 1 столовая ложка и $^1/_4$ чайной ложки. Как видим, эти величины представляют собой смешанные числа — целые и дроби, — измеренные в стаканах, а также столовых и чайных ложках.

Предполагается, что для АТД "Единица измерения" определены следующие операции.

```
+getMeasure() : Measurement {query}
// Возвращает единицу измерения.

+setMeasure(in m: Measurement)
// Устанавливает единицу измерения.

+scaleMeasure(out newMeasure:Measurement,
              in scaleFactor:float)
```

```
// Умножает единицу измерения на безразмерное дробное число
// scaleFactor и получает новую величину newMeasure.

+convertMeasure(in oldUnits:MeasureUnit,
                out newMeasure:Measurement,
                in newUnits:MeasureUnit) {query}
// Выражает величину newMeasure в новых единицах измерения.
```

Допустим, нам нужен абстрактный тип данных "Единица измерения", чтобы точно вычислять дробные числа. Поскольку мы планируем использовать для этой цели язык C++, в котором нет отдельного типа для дробных чисел, а арифметика с плавающей точкой не точна, нам потребуется другой абстрактный тип данных под названием "Дробь". Его операции должны включать в себя сложение, вычитание, умножение и деление дробей. Например, сложение можно определить следующим образом.

```
+addFractions(in first:Fraction,
              in second:Fraction) : Fraction
// Складывает две дроби и возвращает им сумму,
// сокращенную на младший член.
```

Более того, можно предусмотреть операции для преобразования смешанных чисел в дроби и наоборот, если это возможно.

При реализации АТД "Единица измерения" можно использовать АТД "Дробь".

Аксиомы

Аксиома — это математическое правило

Предыдущие спецификации АТД были сформулированы довольно нечетко. Например, они апеллируют к интуиции, предполагая, будто программисту известно, что означает выражение "элемент находится на $i-1$ позиции" абстрактного списка. Это высказывание достаточно просто и доступно для понимания большинства людей. Однако некоторые абстрактные типы данных гораздо сложнее списков и гораздо труднее поддаются интуитивному пониманию. Для таких АТД следует применять более строгий метод. При описании их операций необходимо задать совокупность математических правил, называемых **аксиомами** (axioms), которые точно определяют смысл каждой операции.

Фактически аксиома — это инвариант (истинное утверждение) операции АТД. Например, известны аксиомы алгебраических операций; в частности, для операции умножения формулируются такие аксиомы.

Аксиомы умножения

$(a \times b) \times c = a \times (b \times c)$
$a \times b = b \times a$
$a \times 1 = a$
$a \times 0 = 0$

Эти правила, или аксиомы, истинны для любых чисел a, b и c и описывают поведение оператора умножения $\times$.

Аксиомы определяют поведение АТД

Совершенно аналогично можно задать совокупность аксиом, полностью описывающих свойства операций над абстрактным списком. Например, высказывание

Вновь созданный список всегда пуст

представляет собой аксиому, поскольку это утверждение истинно для любого вновь созданного списка. В терминах операций над абстрактным списком эту аксиому можно выразить так.

Значение выражения (aList.createList()).isEmpty() равно true

Это означает, что список *aList* пуст.

Высказывание

Если вы вставляете элемент x на i-ю позицию абстрактного списка, то результатом операции извлечения i-го элемента будет элемент x

истинно для любых списков, поэтому его можно считать аксиомой. В терминах операций над абстрактным списком эту аксиому можно выразить следующим образом.[3]

```
(aList.insert(i, x)).retrieve(x) = x
```

Это означает, что операция *retrieve* извлекает из *i*-й позиции списка *aList* элемент *x*, который был вставлен туда операцией *insert*. Для упрощения обозначений аргумент *success* пропускается, а операция *retrieve* выполняется так, как если бы она была функцией, возвращающей какое-то значение.

ОСНОВНЫЕ ПОНЯТИЯ

Аксиомы абстрактного списка

1. `(aList.createList()).getLength() = 0`
2. `(aList.insert(i, x)).getLength() = aList.getLength() + 1`
3. `(aList.remove(i)).getLength() = aList.getLength() - 1`
4. `(aList.createList()).isEmpty() = true`
5. `(aList.insert(i, item)).isEmpty() = false`
6. `(aList.createList()).remove(i) = error`
7. `(aList.insert(i, x)).remove(i) = aList`
8. `(aList.createList()).retrieve(i) = error`
9. `(aList.insert(i, x)).retrieve(i) = x`
10. `aList.retrieve(i) = (aList.insert(i, x)).retrieve(i+1)`
11. `aList.retrieve(i+1) = (aList.remove(i)).retrieve(i)`

Совокупность аксиом не заменяет собой пред- и постусловий операций АТД. Например, приведенные выше аксиомы никак не описывают поведение операции *insert* при попытке вставить элемент в 50-ю позицию списка, состоящего из двух элементов. Для того чтобы справиться с этой ситуацией, необходимо предусмотреть в предусловии операции *insert* ограничение

```
1 <= index <= getLength()+1
```

Еще один способ, который мы применим при реализации абстрактного списка в этой главе, не связан с ограничениями на переменную *index*. Просто, когда значение переменной *index* выходит за пределы допустимого диапазона, параметру *success* присваивается значение *false*. Таким образом, для полного описания операций над абстрактным типом данных необходимы как аксиомы, так и пред- и постусловия.

Аксиомы позволяют вычислять результат выполнения последовательности операций. Например, если *aList* — это список значений, как он изменится после выполнения операций

Используйте аксиомы для вычисления результата выполнения последовательности операций АТД

[3] Символ "=" в этой аксиоме означает алгебраическое равенство.

```
aList.insert(1, b)
aList.insert(1, a)
```

Применив операцию *retrieve*, можно легко убедиться, что символ *a* окажется первым элементом списка, а символ *b* — вторым.

Предыдущую последовательность операций можно записать в виде

```
(aList.insert(1, b)).insert(1, a)
```

либо

```
tempList.insert(1, a),
```

где список *tempList* обозначает результат выполнения операции *aList.insert(1, b)*. Теперь извлечем первый и второй элементы из списка *tempList.insert(1, a)*. Получим, что

```
(tempList.insert(1, a)).retrieve(1) = a по аксиоме 9
```

и

```
(tempList.insert(1, a)).retrieve(2)
= tempList.retrieve(1)                по аксиоме 9
= (aList.insert(1, a)).retrieve(1)    по определению списка tempList
= b                                   по аксиоме 9
```

Таким образом, элемент *a* является первым элементом списка, а символ *b* — вторым.

Реализация абстрактных типов данных

В предыдущих разделах основное внимание уделялось спецификациям абстрактных типов данных. При разработке АТД разработчик концентрируется на том, что делают операции, игнорируя детали их реализации. В результате возникает совокупность точно описанных операций над абстрактным типом данных.

Как реализовать АТД, имея точные спецификации операций? Иными словами, как хранить данные АТД и выполнять его операции? Ранее в этой главе указывалось, что при реализации АТД для представления его данных выбирается соответствующая структура. Таким образом, может возникнуть впечатление, что ответ лежит на поверхности: нужно выбрать подходящую структуру данных, а затем написать функции, обеспечивающие доступ к этим данным с помощью операций над абстрактным типом данных. Несмотря на то что эту точку зрения нельзя назвать неправильной, мы искренне надеемся, что читатели не бросятся сразу создавать код. Прежде чем написать первую строку кода, нужно как следует уточнить АТД, пройдя несколько уровней абстракции. Это значит, что для разработки алгоритма, реализующего каждую из операций АТД, следует применять подход "сверху вниз". Каждое последующее описание абстрактного типа данных становится все более конкретным, уточняя предыдущие, более абстрактные спецификации. Этот процесс останавливается, когда полученную структуру данных можно непосредственно реализовать на языке программирования. Чем более примитивен язык, тем больше уровней реализации придется пройти программисту.

Решение, принимаемое программистом на каждом из этапов реализации, в конечном итоге влияет на эффективность программы. Пока мы будем применять интуитивный подход, однако в главе 9 будут рассмотрены количественные методы, которые можно применять для оценки эффективности принимаемых решений.

Напомним, что программа, в которой используется абстрактный тип данных, должна видеть перед собой только стену, состоящую из операций, которые манипулируют данными. На рис. 3.8 снова изображена упомянутая стена. И структура данных, предназначенная для хранения информации, и реализации операций над абстрактным типом данных скрыты за этой стеной. Преимущества такого подхода совершенно очевидны.

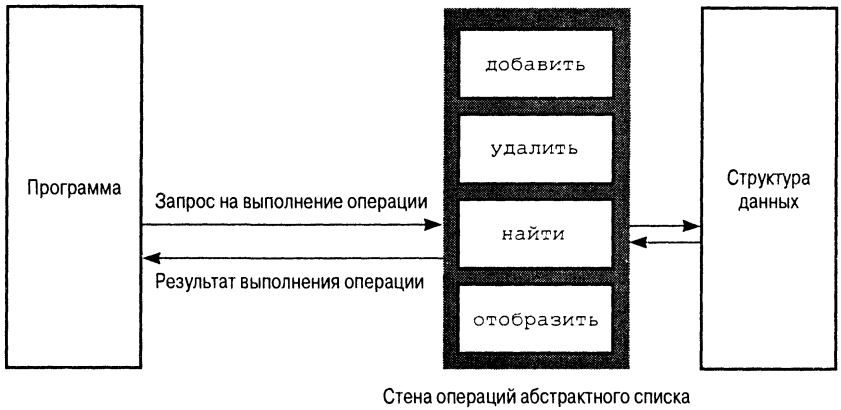


Рис. 3.8. Доступ к структуре данных обеспечивается операциями АТД

В реализации, основанной не на объектно-ориентированном проектировании, структуры данных и операции над ними относятся к разным частям программы. В этом случае клиент кода также может согласиться с существованием стены между ним и данными, используя для доступа к структуре лишь операции АТД. Однако теперь структура данных совершенно не защищена; если пользователь захочет, он может "перелезть" через эту стену! Таким образом, вольно или невольно, клиент может получить непосредственный доступ к структуре данных, как показано на рис. 3.9. Почему эта ситуация крайне нежелательна? Позднее в этой главе для хранения элементов абстрактного списка мы применим массив *items*. В программе, использующей такой список, например, можно случайно проигнорировать операцию *retrieve* и получить доступ к первому элементу списка, написав оператор

```
firstItem = items[0];
```

Если реализация списка изменится, программа станет некорректной. Для того чтобы ее исправить, понадобится найти и изменить все ссылки на элемент *items[0]*, но прежде всего нужно четко уяснить, что такое обращение к первому элементу списка является несомненной ошибкой!

В объектно-ориентированных языках, таких как язык C++, существует способ для создания стены, состоящей из операций АТД и предотвращающей несанкционированный доступ к структуре данных. Настало время изучить эти аспекты языка C++, рассмотрев классы, пространства имен и исключительные ситуации.

Классы языка C++

Напомним, что в рамках объектно-ориентированного подхода (ООП) программа рассматривается не как последовательность операций, а как совокупность компонентов, называемых объектами. Инкапсуляция — один из трех основных

принципов ООП[4] —позволяет возводить крепостные стены из операций АТД. Следовательно, этот принцип важен для реализации абстрактных типов данных, и его необходимо детально изучить.

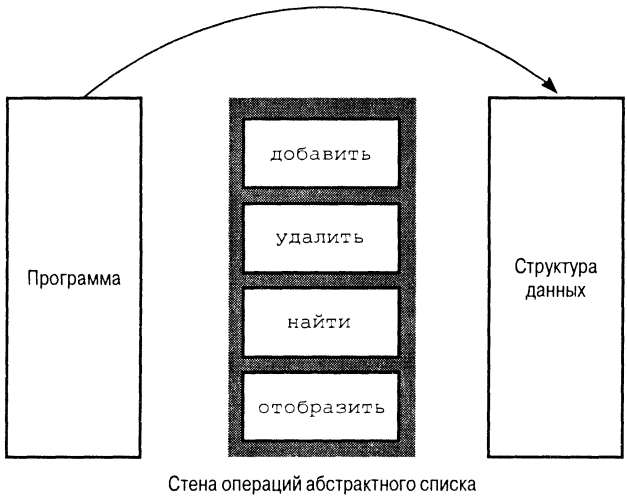


Рис. 3.9. Преодоление стены, состоящей из операций АТД

Инкапсуляция скрывает детали реализации

Инкапсуляция объединяет данные АТД с его операциями, называемыми **методами** (methods), образуя **объекты** (objects). Отвлекитесь от ранее принятой точки зрения на АТД как о наборе многих компонентов (см. рис. 3.8) и попробуйте перейти на более высокий уровень абстракции, рассматривая объект, изображенный на рис. 3.10, как некую отдельную сущность. Этот объект скрывает детали своего внутреннего устройства от пользователя. Таким образом, поведение этого объекта определяется операциями АТД.

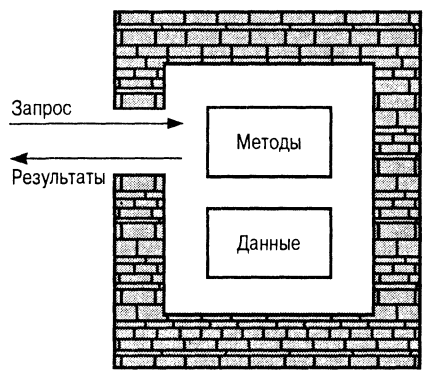


Рис. 3.10. Данные и методы, инкапсулированные в объекте

[4] Остальные принципы, наследование и полиморфизм, будут рассмотрены в главе 8.

В качестве примера такого объекта можно привести мяч. Поскольку баскетбольный, волейбольный, теннисный или футбольный мяч вызывает ненужные ассоциации с играми, а не с объектом как таковым, попробуем абстрагировать это понятие, нарисовав сферу. Сфера заданного радиуса имеет атрибуты, т.е. объем и площадь поверхности. Сфера, как объект, должна иметь возможность сообщать о своем радиусе, объеме, площади поверхности и т.д. Таким образом, объект "сфера" должен иметь методы, вычисляющие и возвращающие эти значения. В этом разделе мы рассмотрим понятие "сфера" с точки зрения объектно-ориентированного программирования. Позднее, в главе 8, мы выведем понятие "мяч" из понятия "сфера".

Класс языка C++ определяет новый тип данных

Как на практике определить объект в языке C++? Напомним, что в главе 1 было введено понятие **класса** как совокупности объектов, обладающих определенными свойствами. В языке C++ классы представляют собой новые типы данных, экземплярами которых являются объекты. Синтаксис класса напоминает синтаксис структуры в языке C++. Как и структура, класс может содержать **данные-члены** (data-members). Обращаться к этим данным можно точно так же, как и к данным-членам структуры. Для этого указывается экземпляр класса и имя его члена.

Объект — это экземпляр класса

Кроме того, класс может содержать **функции-члены** (member functions), которые манипулируют данными-членами класса. Для вызова функции-члена класса нужно указать экземпляр класса и ее имя, как и при обращении к данным-членам.

По умолчанию все члены класса являются **закрытыми** (private) — программа не имеет к ним прямого доступа, пока вы не объявите их **открытыми** (public). Однако реализации функций-членов класса имеют доступ к любым закрытым членам этого класса. В противоположность этому все члены структуры по умолчанию являются открытыми, пока вы не объявите их закрытыми.

Поскольку структуры в языке C++ также могут содержать функции-члены, класс и структура работают одинаково, если в них явно объявлены открытые и закрытые члены. Несмотря на то что ключевые слова `struct` и `class` являются взаимозаменяемыми, делать этого не следует. Структура подходит для хранения данных разных типов, когда не нужно определять новый тип данных. В этой книге структуры используются только для хранения данных и могут иногда содержать одну функцию-член для своей инициализации. Класс нужно применять при реализации АТД для определения нового типа данных. Ключевое слово `class` используется в книге только для определения типов объектов, причем открытые и закрытые разделы класса всегда указываются явно.

Конструктор создает и инициализирует объект

Абстрактные типы данных, рассмотренные нами ранее, содержали операции для своего создания и уничтожения. Классы также имеют такие методы. Они называются **конструкторами** (constructors) и **деструкторами** (destructors). Конструктор создает и инициализирует новый экземпляр класса. Деструктор уничтожает экземпляр класса, время жизни которого истекло. Типичный класс имеет несколько конструкторов, но всегда только один деструктор. Во многих ситуациях деструктор можно не описывать. В этих случаях компилятор сам создаст автоматический деструктор класса (compile-generated destructor). Для классов, рассмотренных в этой главе, автоматического деструктора будет достаточно. Более подробно вопросы разработки деструкторов описаны в главе 4.

В языке C++ имена конструктора и класса должны совпадать. Конструкторы не имеют типа возвращаемого значения — даже типа `void` — и не используют оператор `return`. Более подробно конструкторы будут рассмотрены позднее, после примера определения класса.

Заголовочный файл. Определение каждого класса следует размещать в отдельном **заголовочном файле** (header file), или **файле спецификации** (specification file). По общепринятому соглашению, эти файлы должны иметь расширение *.h*.[5] Приведенный ниже заголовочный файл *Sphere.h* содержит описание объектов сферы.

```
// ************************************************
// Заголовочный файл Sphere.h класса Sphere.
// ************************************************
const double PI = 3.14159;
class Sphere
{
public:
   Sphere();
// Конструктор по умолчанию: создает сферу, инициализируя ее
// радиус значением, заданным по умолчанию.
// Предусловие: нет.
// Постусловие: существует сфера радиуса 1.

Sphere(double initialRadius);
// Конструктор: создает сферу, инициализируя ее
// радиус заданным значением.
// Предусловие: радиус задается аргументом initialRadius.
// Постусловие: существует сфера радиуса initialRadius.

void setRadius(double newRadius);
// Устанавливает (изменяет) радиус существующей сферы.
// Предусловие: радиус задается аргументом newRadius.
// Постусловие: существует сфера радиуса newRadius.

double getRadius() const;
// Вычисляет радиус сферы.
// Предусловие: нет.
// Постусловие: возвращает радиус сферы.

double getDiameter() const;
// Вычисляет диаметр сферы.
// Предусловие: нет.
// Постусловие: возвращает диаметр сферы.

double getCircumference() const;
// Вычисляет длину окружности сферы.
// Предусловие: PI — именованная константа.
// Постусловие: возвращает длину окружности сферы.

double getArea() const;
// Вычисляет площадь поверхности сферы.
// Предусловие: PI — именованная константа.
// Постусловие: возвращает площадь поверхности сферы.

double getVolume() const;
// Вычисляет объем сферы.
// Предусловие: PI — именованная константа.
```

[5] Иногда используются также расширения *.hpp* и *.hxx*.

```
// Постусловие: возвращает объем сферы.

void displayStatistics() const;
// Выводит параметры сферы.
// Предусловие: нет.
// Постусловие: выводит на экран радиус, диаметр, длину
// окружности, площадь поверхности и объем сферы.

private:
   double theRadius; // Радиус сферы
}; // Конец определения класса
// Конец заголовочного файла
```

Данные-члены класса должны быть закрытыми

Данные-члены класса следует всегда помещать в закрытый раздел. Обычно для доступа к данным-членам класса предусматриваются отдельные методы, например *setRadius* и *getRadius*. Это позволяет контролировать доступ к данным-членам из других мест программы, не только облегчая отладку, но и предотвращая появление логических ошибок.

Константные функции не могут изменять данные-члены класса

Некоторые объявления функций содержат ключевое слово *const*.

```
double getRadius() const;
```

Такие функции не могут изменять данные-члены класса. Объявление константной функции *getRadius* повышает надежность программы, поскольку эта функция может лишь возвращать текущее значения радиуса сферы, но не в состоянии изменить его.

Комментарии в заголовочном файле должны описывать функции-члены

Программист, использующий чужой класс в своей программе, обычно видит лишь заголовочный файл. Поэтому документацию, сопровождающую функции-члены, следует помещать в заголовочном файле, а открытые разделы класса должны предшествовать закрытым. (Это просто совет, а не требование языка. — *Прим. ред.*)

Перейдем к реализации класса *Sphere*, начиная с конструкторов.

Конструкторы. Конструктор выделяет память для объекта и может инициализировать его данные конкретными значениями. В классе могут существовать несколько конструкторов, как это показано на примере класса *Sphere*.

Конструктор по умолчанию не имеет аргументов

Первым конструктором класса *Sphere* является **конструктор по умолчанию** (default constructor)

```
Sphere()
```

Конструктор по умолчанию по определению не имеет аргументов. Обычно он инициализирует данные-члены класса конкретными значениями. Например, описанная ниже реализация конструктора по умолчанию присваивает переменной *theRadius* значение 1.0.

```
Sphere::Sphere(): theRadius(1.0)
{
} // Конец конструктора по умолчанию
```

Обратите внимание на квалификатор *Sphere::*, предшествующий имени конструктора. Реализуя любую функцию-член, перед ее именем необходимо указать имя класса, которому она принадлежит, и **оператор разрешения области види-**

мости :: (scope resolution operator), для того чтобы отличить ее от других функций, которые могут иметь такие же имена.

Для присвоения начальных значений данным-членам класса используйте инициализатор

Разумеется, для того чтобы задать начальное значение переменной *theRadius*, можно применить обычный оператор присваивания, однако в таких ситуациях предпочтительнее использовать **инициализатор** (initializer) — в данном случае выражение *theRadious(1.0)*. Каждый инициализатор использует функциональное обозначение, состоящее из имени поля класса, за которым в скобках указывается его начальное значение. Если в конструкторе используются несколько инициализаторов[6], они разделяются запятыми. Перед первым (или единственным) инициализатором ставится двоеточие. Часто реализация конструктора сводится исключительно к инициализаторам, так что его тело остается пустым, как в описанном выше примере. Учтите, что инициализаторы используются только в конструкторах и нигде больше.

Инициализаторы используются только в конструкторах

Конструкторы вызываются неявно, при объявлении экземпляра класса. Так, в следующем примере вызывается конструктор по умолчанию, создающий объект *unitSphere* и устанавливающий его радиус равным 1.0.

```
Sphere unitSphere;
```

Обратите внимание, что после имени объекта скобки не ставятся.

В другом примере создается объект класса *Sphere*, радиус которого равен значению аргумента *initialRadius*.

```
Sphere(double initialRadius)
```

Этот конструктор лишь инициализирует закрытый член *theRadius* значением аргумента *initialRadius*. Эта реализация[7] имеет следующий вид.

```
Sphere::Sphere(double initialRadius):
               theRadius(initialRadius)
{
} // Конец конструктора
```

Этот конструктор неявно вызывается при объявлении

```
Sphere mySphere(5.1);
```

В этом случае объект *mySphere* имеет радиус 5.1.

Если в классе не описан ни один конструктор, компилятор сам сгенерирует конструктор по умолчанию, т.е. конструктор, не имеющий аргументов. Однако полученный при этом **автоматический конструктор** (compiler-generated default constructor) может инициализировать данные-члены неподходящими значениями.

Если в классе определен конструктор, имеющий аргументы, но пропущен конструктор по умолчанию, компилятор не станет генерировать автоматический конструктор. Таким образом, приведенная ниже строка окажется недопустимой.

```
Sphere defaultSphere;
```

---

[6] Если класс имеет несколько данных-членов (полей), конструктор инициализирует их в том порядке, в каком они указаны в определении класса, а не в порядке их перечисления в списке инициализации. Во избежание недоразумений в обоих случаях нужно придерживаться одного и того же порядка, даже если порядок инициализации совершенно не важен.

[7] Эту реализацию мы вскоре усовершенствуем, предусмотрев вариант, когда радиус отрицателен.

Файл реализации. Обычно реализации функций-членов класса размещаются в **файле реализации** (implementation file), имеющем расширение *.cpp*.[8] Файл реализации класса *Sphere* приведен ниже. Обратите внимание, что внутри определения функции-члена можно ссылаться на любые члены класса и вызывать любую другую функцию-член класса, не указывая перед ними имя класса *Sphere* и оператор разрешения области видимости *::*.

Файл реализации содержит определения всех функций-членов класса

```
//
**************************************
******************
// Файл реализации Sphere.cpp.
// ***********************************************************
#include "Sphere.h" // Заголовочный файл
#include <iostream.h>

Sphere::Sphere(): theRadius(1.0)
{
} // Конец конструктора по умолчанию

Sphere::Sphere(double initialRadius)
{
   if (initialRadius > 0)
      theRadius = initialRadius;
   else
      theRadius = 1.0;
} // Конец конструктора по умолчанию

void Sphere::setRadius(double newRadius)
{
   if (newRadius > 0)
      theRadius = newRadius;
   else
      theRadius = 1.0;
} // Конец функции-члена setRadius

double Sphere::getRadius() const
{
   return theRadius;
} // Конец функции-члена getRadius

double Sphere::getDiameter() const
{
   return 2.0 * theRadius;
} // Конец функции-члена getDiameter

double Sphere::getCircumference() const
{
   return PI * getDiameter();
} // Конец функции-члена getCircumference

double Sphere::getArea() const
{
   return 4.0 * PI * theRadius * theRadius;
} // Конец функции-члена getArea
```

[8] Иногда используются также расширения *.c* и *.cxx*.

```
// Локальная переменная, такая как radiusCubed,
// не должна быть членом класса

double Sphere::getVolume() const
{
   double radiusCubed = theRadius * theRadius * theRadius;
   return (4.0 * PI * radiusCubed)/3.0;
} // Конец функции-члена getVolume

// Изнутри функции displayStatistics можно вызывать
// функцию-член getRadius или обращаться к полю theRadius.

void Sphere::displayStatistics() const
{
   cout << "\nРадиус = " << getRadius()
        << "\nДиаметр = " << getDiameter()
        << "\nДлина окружности = " << getCircumference()
        << "\nПлощадь = " << getArea()
        << "\nОбъем = " << getVolume() << endl;
} // Конец функции-члена displayStatistics
// Конец файла реализации.
```

Следует различать данные-члены класса и локальные переменные, необходимые для реализации функции-члена. Эти локальные переменные не следует делать данными-членами класса.

Использование класса Sphere. Применение класса *Sphere* иллюстрируется следующей простой программой.

```
#include <iostream.h>
#include "Sphere.h"
int main()
{
   Sphere unitSphere;        // Радиус равен  1.0
   Sphere mySphere(5.1);     // Радиус равен 5.0

   unitSphere.displayStatistics();
   mySphere.setRadius(4.2); // Устанавливаем радиус, равный 4.2
   cout << mySphere.getDiameter() << endl;

   return 0;
} // Конец функции main
```

Объект, такой как *mySphere*, может по запросу устанавливать новое значение радиуса, вычислять диаметр, площадь поверхности, длину окружности и объем сферы, а также выводить эти параметры на экран. Такие запросы, направляемые объекту, называются **сообщениями** (messages) и представляют собой обычные вызовы функции. Таким образом, для того чтобы вызвать функцию-член объекта, перед ее именем, например, *setRadius*, нужно указать имя объекта, скажем *mySphere*.

Обратите внимание, что в приведенную выше программу включен заголовочный файл *Sphere.h*, но не включен файл реализации *Sphere.cpp*.[9] Файл реализации класса компилируется отдельно от программы, использующей этот класс.

[9] Более подробная информация о заголовочных файлах и файлах реализации содержится в Приложении А в разделе "Библиотеки".

Способ, который используется для связывания программы с этой реализацией, зависит от конкретной операционной системы.

Приведенная выше программа представляет собой пример **клиента** класса (client of a class). Клиентом конкретного класса является программа или модуль, использующая это класс. Термин "пользователь" мы зарезервируем для обозначения человека, применяющего программу.

Наследование. Здесь лишь изложены факты, касающиеся наследования, поскольку этот механизм наиболее часто применяется для создания новых классов в языке C++. Более полно наследование будет описано в главе 8.

Допустим, нам нужно создать класс окрашенных сфер на основе уже существующего класса *Sphere*. Для этого можно написать совершенно новый класс, но поскольку окрашенная сфера все же является в первую очередь собственно сферой и, следовательно, тесно связана с классом *Sphere*, можно повторно использовать реализацию класса *Sphere*, добавив операции окрашивания и новые свойства. Все это позволяет создать механизм наследования классов. Рассмотрим спецификацию класса *ColoredSphere*, использующего наследование.

Класс, производный от класса Sphere

```
#include "Sphere.h"
enum Color {RED, BLUE, GREEN, YELLOW};
class ColoredSphere: public Sphere
{
public:
   ColoredSphere(Color initialColor);
   ColoredSphere(Color initialColor,
                 double initialRadius);
   void setColor(Color newColor);
   Color getColor();
private:
   Color c;
}; //Конец определения класса ColoredSphere
```

Класс *Sphere* называется **базовым классом** (base class), или **суперклассом** (superclass), а класс *ColoredSphere* называется **производным** (derived), или **подклассом** (subclass) класса *Sphere*.

Любой экземпляр производного класса одновременно рассматривается и как экземпляр базового класса и может быть использован в этом качестве. Кроме того, все открытые функции и данные-члены базового класса могут использоваться экземплярами производного класса. Объекты производного класса также могут иметь свои собственные открытые функции и данные-члены, указанные в определении производного класса.

Функции-члены класса *ColoredSphere* реализуются следующим образом.

```
ColoredSphere::ColoredSphere(Color initialColor):Sphere()
{
   c = initialColor;
} // Конец конструктора

ColoredSphere::ColoredSphere(Color initialColor,
                             double initialRadius)
                            :Sphere(initialRadius)
{
   c = initialColor;
} // Конец конструктора
```

```
void ColoredSphere::setColor(Color newColor)
{
   c = newColor;
} // Конец функции-члена setColor

Color ColoredSphere::getColor()
{
   return c;
} // Конец функции-члена getColor
```

Обратите внимание, что в конструкторах класса *ColoredSphere* используются конструкторы *Sphere()* и *Sphere(initialRadius)*. Реализации производных классов часто применяют конструкторы базового класса указанным выше способом, а затем добавляют инициализации членов, характерных только для производного класса.

Рассмотрим функцию, использующую класс *ColoredSphere*.

Экземпляр производного класса может вызывать открытые методы базового класса

```
void useColoredSphere()
{
   ColoredSphere ball(RED);
   ball.setRadius(5.0);
   cout << "Диаметр мяча равен " << ball.getDiameter();
   ball.setColor(BLUE);
   ...
} // Конец функции-члена useColoredSphere
```

Эта функция использует конструктор и метод *setColor* из производного класса *ColoredSphere*. Кроме того, она вызывает методы *setRadius* и *getRadius*, определенные в базовом классе *Sphere*.

Пространства имен

Часто решение проблемы выражается с помощью группы связанных друг с другом классов и других объявлений, например, функций, переменных, типов и констант. В языке C++ предусмотрен механизм, позволяющий осуществлять логическую группировку этих объявлений и определений в общей декларативной области (declarative region), известной под названием **пространство имен** (namespace). Ее объявление выглядит следующим образом.

```
namespace названиеПространстваИмен
{
  // Здесь размещаются объявления
}
```

Содержимое пространства имен доступно любому коду, расположенному как внутри, так и снаружи. Внутри пространства имен код может обращаться к его элементам непосредственно. Однако для обращения к тем же самым элементам извне необходима особая синтаксическая конструкция. Допустим, например, что в программе объявлено пространство имен *smallNamespace*.

```
namespace smallNamespace
{
   int count = 0;
   void abc();
} // Конец пространства smallNamespace
```

Функцию, объявленную внутри этого пространства, можно реализовать как непосредственно внутри него, так и в любом другом месте, применив оператор разрешения области видимости. Рассмотрим, к примеру, реализацию функции *abc*.

```
void smallNamespace::abc()
{
  // Реализация
  . . .
} // Конец функции abc
```

К элементам, находящимся вне пространства имен *smallNamespace*, можно обращаться с помощью оператора разрешения области видимости.

```
smallNamespace::count += 1;
smallNamespace::abc();
```

Если пространство имен содержит много элементов, такой синтаксис становится неудобным. На этот случай в языке C++ предусмотрено **объявление *using*** (using declaration), которое позволяет непосредственно использовать элементы, расположенные за пределами пространства имен, не прибегая к оператору разрешения области видимости. Тогда приведенный выше код можно переписать следующим образом.

```
using namespace smallNamespace;
smallNamespace::count += 1;
abc();
```

Сокращенная форма объявления *using* позволяет сделать код лаконичнее.

```
using smallNamespace::abc;
smallNamespace::count += 1;
abc();
```

Это объявление указывает, что с помощью сокращенной записи можно вызывать только функцию *abc*. Для доступа к переменной *count* по-прежнему нужно применять оператор разрешения области видимости.

Элементы, объявленные в стандартной библиотеке языка C++ (C++ Standard Library), пребывают в пространстве имен *std*. Для того чтобы использовать сокращенную запись при доступе к элементам стандартной библиотеки, нужно включить в программу следующее объявление *using*.

```
using namespace std;
```

Большинство файлов, включаемых в программы на языке C++ с помощью директивы *include*, в последней версии стандарта было обновлено. Обычно в новой версии сохраняются старые имена, а расширение *.h* отбрасывается. Например, чтобы включить в программу функции ввода и вывода данных, в старой версии нужно было включить в текст директиву

```
#include <iostream.h>
```

В новой версии эта директива будет выглядеть так.

```
#include <iostream>
using namespace std;
```

Объявление *using namespace* означает, что в программе можно использовать сокращенную запись.

Если объявление сделано вне какого-либо пространства имен, говорят, что оно принадлежит **глобальному пространству имен** (global namespace). Большинство классов, описанных в книге, для простоты объявлено в глобальном пространстве имен.

Реализация абстрактного списка в виде массива

Перейдем к реализации абстрактного списка в виде класса. Напомним, что в этом типе предусмотрены следующие операции.

```
+createList()
+destroyList()
+isEmpty():boolean
+getLength():integer
+insert(in index:integer, in newItem:ListItemType,
        out success:boolean)
+remove(in index:integer, out success:boolean)
+retrieve(in index:integer,
          out dataItem:ListItemType,
          out success:boolean)
```

Нам нужен способ для представления элементов списка и его длины. На первый взгляд элементы списка удобно хранить в массиве *items*. Можно даже подумать, что список — это синоним массива. Однако это не совсем так. **Реализация списка в виде массива** — вполне естественный выбор, поскольку и массив, и список хранят пронумерованные элементы. Однако абстрактный список предусматривает такие операции, которых нет у массива, например операцию *getLength*. В следующей главе мы увидим другую реализацию абстрактного списка, в которой не используется массив.

В любом случае мы можем хранить *k*-й элемент списка в ячейке *items[k-1]*. Сколько ячеек массива займет список? Может быть, весь массив, а может быть и нет. Иными словами, нужно различать, в каких ячейках массива хранятся элементы списка, а какие —свободны. Максимальная длина массива, т.е. его **физический размер** (physical size), известна и задается константой *MAX_LIST*. Для отслеживания текущего количества элементов списка, записанных в массиве, т.е. его **логического размера** (logical size) , будем использовать переменную *size*. Преимущества этого подхода очевидны — реализация функции *getLength* будет очень простой. Итак, для реализации можно применить следующий код.

```
const int MAX_LIST = 100; // Максимальная длина списка
typedef int ListItemType; // Тип элементов списка
ListItemType items[MAX_LIST]; // Массив элементов списка
int size; // Длина списка
```

На рис. 3.11 показаны данные-члены реализации абстрактного списка целых чисел в виде массива. Для того чтобы вставить новый элемент в заданную позицию массива, нужно сдвинуть все элементы, находящиеся правее, на одну позицию и вставить новый элемент на освободившееся место. Эта операция показана на рис. 3.12.

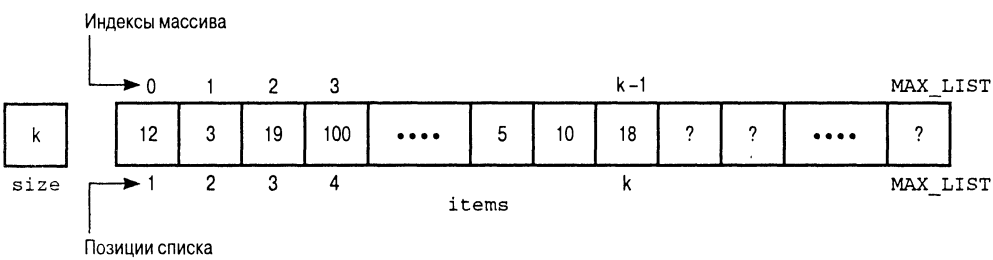


Рис. 3.11. Реализация списка в виде массива

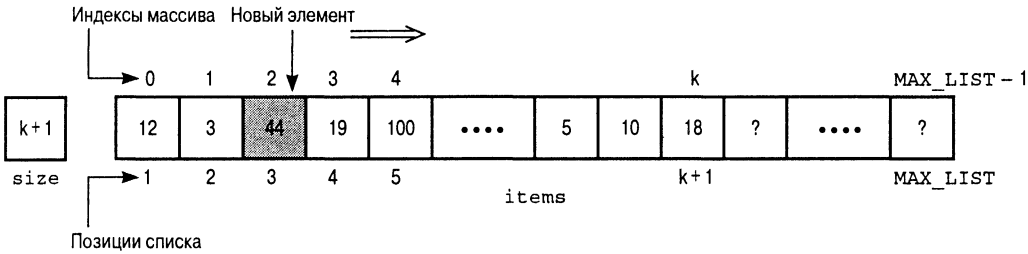


Рис. 3.12. Сдвиг элементов массива для вставки нового элемента списка в третью ячейку

Рассмотрим теперь процедуру удаления элемента из списка. Его можно просто стереть, однако это приведет к образованию пустот в массиве, как показано на рис. 3.13, *а*. Массив, содержащий пустоты, порождает следующие проблемы.

- Значение `size - 1` больше не равно последнему индексу массива. Для его отслеживания нужна новая переменная, `lastPosition`.
- Поскольку элементы разбросаны, функция `retreive` должна проверять каждую ячейку массива, даже если он содержит всего несколько элементов.
- Если ячейка `items[MAX_LIST - 1]` занята, список может показаться полным, даже если количество его элементов намного меньше константы `MAX_LIST`.

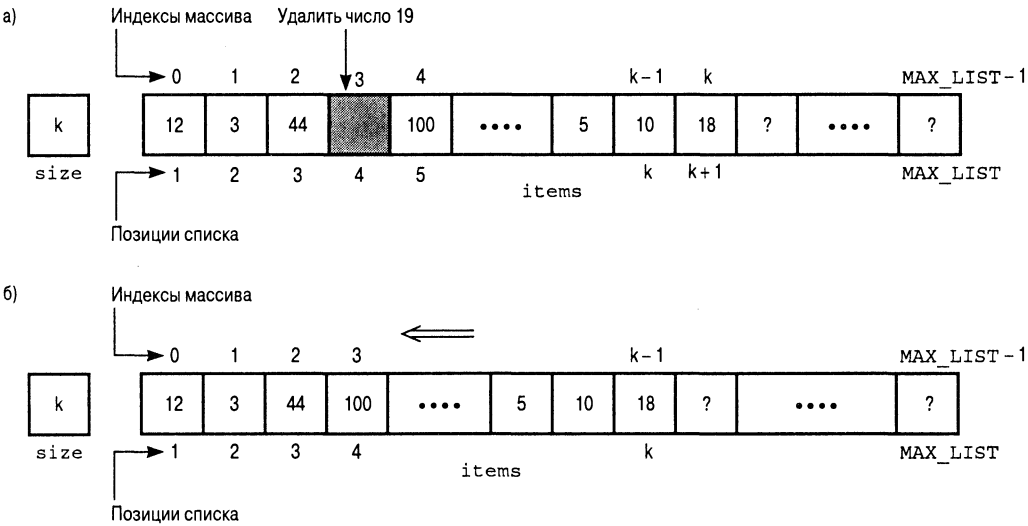


Рис. 3.13. Удаление элемента из списка: а) удаление, порождающее пустоты; б) заполнение пустот путем сдвига элементов

Итак, сдвигать элементы, заполняя образовавшиеся пустоты, как показано на рис. 3.13, *б*, действительно необходимо.

Сдвиг элементов массива при удалении

Каждую операцию АТД следует реализовать в виде функции-члена класса. При этом каждой операции понадобится доступ к массиву `items` и переменной `size`, в которой хранится длина списка, поэтому они должны быть данными-членами класса.

Реализация абстрактного списка в виде класса

Для того чтобы скрыть массив *items* и переменную *size* от клиентов класса, их следует объявить закрытыми. Хотя это пока совсем не очевидно, это окажется полезным при определении функции *translate(position)*, возвращающей индекс ячейки массива, содержащей элемент списка, стоящий на позиции *position*. Иными словами, вызов *translate(position)* должен возвращать величину *position - 1*.

Массив items и переменную size нужно объявить закрытыми

Эта функция не относится к операциям над абстрактным списком и не должна быть доступной клиенту. Следовательно, функцию *hide* нужно скрыть от клиента, определив ее в закрытом разделе класса.

Функция translate является закрытым членом класса

Ниже приводится заголовочный файл *ListA.h* для класса списков. Конструктор этого класса соответствует операции *createList*. Автоматический деструктор, соответствующий операции *destroyList*, целиком удовлетворяет потребности класса, поэтому мы не будем создавать свой собственный конструктор.

```
// ********************************************************
// Заголовочный файл ListA.h для реализации абстрактного
// списка в виде массива
// ********************************************************
const int MAX_LIST = максимальная_длина_списка;
typedef desired-type-of-list-item ListItemType;

class List
{
public:
 List(); // Конструктор по умолчанию
         // Используется автоматический деструктор

// Операции над списком:
bool isEmpty() const;
// Определяет, пуст ли список.
// Предусловие: нет.
// Постусловие: если список пуст, возвращает значение true,
// в противном случае возвращает значение false.

int getLength() const;
// Определяет длину списка.
// Предусловие: нет.
// Постусловие: возвращает текущее количество элементов списка

void insert(int index, ListItemType newItem,
            bool& success);
// Вставляет в список новый элемент на заданную позицию.
// Предусловие: аргумент index задает позицию,
// в которую следует вставить новый элемент списка.
// Постусловие: если вставка прошла успешно, на позиции
// index в списке стоит элемент newItem, а остальные элементы
// соответствующим образом пронумерованы. Переменной success
// присваивается значение true; в противном случае переменной
// success присваивается значение false.
// Замечание: если index < 1 или index > getLength()+1,
// вставка будет безуспешной.
```

```
   void remove(int index, bool& success);
   // Удаляет из списка элемент, стоящий на заданной позиции.
   // Предусловие: аргумент index задает позицию удаляемого элемента.
   // Постусловие: если 1 <= index <= getLength(),
   // элемент, стоявший в списке на позиции index, удален,
   // а остальные элементы соответствующим образом пронумерованы.
   // Переменной success присваивается значение true; в противном
   // случае переменной success присваивается значение false.

   void retrieve(int index, ListItemType& dataItem,
                 bool& success) const;
   // Извлекает из списка элемент, стоящий в заданной позиции.
   // Предусловие: номер извлекаемого элемента задается
   // аргументом index.
   // Постусловие: если 1 <= index <= getLength(),
   // значение указанного элемента хранится в переменной
   // dataItem, а переменной success присваивается значение true;
   // в противном случае переменной success присваивается
   // значение false.

   private:
      ListItemType items[MAX_LIST]; // Массив элементов списка
      int size; // Количество элементов списка
      int translate(int index) const;
      // Преобразует позицию элемента в списке
      // в соответствующий индекс массива.
   }; // Конец определения класса List
   // Конец заголовочного файла.
```

Реализации описанных выше функций хранятся в файле *ListA.cpp*, приведенном ниже.

```
   // ****************************************************************
   // Файл реализации абстрактного класса в виде массива ListA.cpp
   // ****************************************************************
   #include "ListA.h" // Заголовочный файл
   List::List() : size(0)
   {
   } // Конец конструктора по умолчанию.

   bool List::isEmpty() const
   {
      return bool(size == 0);
   } // Конец функции-члена isEmpty

   int List::getLength() const
   {
      return size;
   } // Конец функции-члена getLength

   void List::insert(int index, ListItemType newItem,
                     bool& success)
   {
      success = bool( (index >= 1) &&
      (index <= size+1) &&
      (size < MAX_LIST) );
```

```
    if (success)
    {
      // Освобождаем место для нового элемента путем
      // сдвига всех элементов списка, начиная с позиции
      // positions >= index (если index == size+1
      // сдвиг не выполняется.
      for (int pos = size; pos >= index; --pos)
         items[translate(pos+1)] = items[translate(pos)];

      // Вставляем новый элемент
         items[translate(index)] = newItem;
         ++size; // Увеличивает текущую длину списка на 1
    } // Конец оператора if
} // Конец функции-члена insert

void List::remove(int index, bool& success)
{
   success = bool( (index >= 1) && (index <= size) );
   if (success)
   {
      // Удаляем элемент, сдвигая к началу списка все элементы,
      // стоящие правее index (если index == size сдвиг не
      // выполняется).

      for (int fromPosition = index+1;
          fromPosition <= size; ++fromPosition)
          items[translate(fromPosition-1)] =
          items[translate(fromPosition)];
          --size; // Увеличивает текущую длину списка на 1
   } // end if
} // Конец функции-члена remove

void List::retrieve(int index, ListItemType& dataItem,
                    bool& success) const
{
   success = bool( (index >= 1) &&
                   (index <= size) );
   if (success)
      dataItem = items[translate(index)];
} // Конец функции-члена retrieve

int List::translate(int index) const
{
   return index-1;
} // Конец функции-члена translate
// Конец файла реализации.
```

Клиенты класса не имеют непосредственного доступа к закрытым членам

Обратите внимание на то, что ссылки *aList.size*, *aList.items[4]* и *aList.translate[6]* в программе могут быть некорректными, поскольку переменные *size*, *items* и функция *translate* находятся в закрытой части класса.

Итак, для реализации абстрактного списка на основе указанных операций сначала необходимо выбрать соответствующую структуру данных. Далее нужно определить и реализовать класс в заголовочном файле. Операции над абстрактным типом определяются как открытые функции-члены класса, а данные АТД

обычно объявляются в закрытом разделе класса. Затем в файле реализации необходимо реализовать все функции-члены класса. Программа, использующая такой класс, сможет получить доступ к данным только с помощью операций над абстрактным типом.

Исключительные ситуации в языке C++

Исключительная ситуация — это механизм для обработки ошибок во время выполнения программы

Многие языки программирования, в том числе язык C++, предусматривают **исключительные ситуации** (exceptions), представляющие собой механизм для обработки ошибок. Если в ходе выполнения программы возникла ошибка, можно **сгенерировать** (throw) исключительную ситуацию. Говорят, что код, предназначенный для работы с исключительной ситуацией, **перехватывает** (catch), или **обрабатывает** (handle) ее.

Перехват исключительной ситуации. Для перехвата исключительной ситуации в языке C++ предусмотрены **блоки try-catch** (try-catch bloks). Оператор, который может породить исключительную ситуацию, следует поместить в блок *try*. За этим блоком должны следовать один или несколько блоков *catch*. В каждом блоке *catch* должен быть указан тип исключительной ситуации, для перехвата которой он предназначен. С блоком *try* могут быть связаны несколько блоков *catch*, даже если отдельный оператор может порождать исключительные ситуации нескольких типов. Кроме того, блок *try* может содержать много операторов, каждый из которых может генерировать исключительную ситуацию. Общая синтаксическая конструкция блока *try* приведена ниже.

Для операторов, которые могут породить исключительную ситуацию, следует применять блок try

```
try
{
   оператор(ы);
}
```

Синтаксис блока *catch* выглядит так.

Для каждого типа исключительной ситуации нужно применять отдельный блок catch, предназначенный для ее обработки

```
catch (КлассИсключительнойСитуации
идентификатор)
{
   оператор(ы);
}
```

Когда операторы, помещенные в блок *try*, порождают исключительную ситуацию, оставшаяся часть блока *try* игнорируется, а управление передается операторам, размещенным в блоке *catch*, соответствующем типу возникшей исключительной ситуации. Затем выполняются операторы блока *catch*. Выполнение программы возобновляется, начиная с точки, следующей за последним блоком *catch*. Если для порожденной исключительной ситуации не подходит ни один блок *catch*, программа завершается аварийно.

Обратите внимание, если исключительная ситуация генерируется в середине блока *try*, вызываются деструкторы всех локальных объектов этого блока. Это гарантирует освобождение всех ресурсов, захваченных блоком, даже если он не будет выполнен до конца.

Генерирование исключительных ситуаций. Когда внутри функции обнаруживается ошибка, исключительную ситуацию можно сгенерировать с помощью оператора следующего вида.

```
throw КлассИсключительнойСитуации(строковыйАргумент);
```

Для генерации исключительных ситуаций используйте оператор throw

Здесь обозначение *КлассИсключительнойСитуации* относится к типу исключительной ситуации, которую необходимо сгенерировать, а запись *строковыйАргумент* означает аргумент конструктора этого класса, описывающий возникающую ошибку. При выполнении оператора *throw* оставшийся код функции не выполняется, а исключительная ситуация передается обратно в точку, из которой была вызвана функция. Более детальное описание этого механизма приведено в Приложении А.

Программист может определять свой собственный класс исключительных ситуаций

В стандартной библиотеке C++ можно найти класс исключительной ситуации, удовлетворяющий потребностям программы. Однако программист может определять свой собственный класс исключительных ситуаций. При этом в качестве базового обычно используется класс исключительных ситуаций *exception*, или один из производных от него классов. Это обеспечивает возможности стандартизированной работы с исключительными ситуациями. В частности, все исключительные ситуации, предусмотренные в стандартной библиотеке языка C++, содержат функцию-член *what*, возвращающую сообщение, описывающее возникшую исключительную ситуацию. Если при создании своего собственного класса исключительных ситуаций в качестве базового применяется класс *exception*, нужно использовать пространство имен *std*.

Чтобы указать, какая исключительная ситуация будет генерироваться функцией, включите раздел *throw* в заголовок функции, как показано ниже.

Функция, которая может сгенерировать исключительную ситуацию

```
void myMethod(int x)
throw(BadArgException, MyException)
{
   if (x == MAX)
   throw BadArgException("BadArgException: причина");
   // Какой-то код
   ...
   throw MyException("MyException: причина");
} // Конец функции myMethod
```

Включение раздела *throw* в спецификацию функции гарантирует, что данная функция сможет генерировать только указанные исключительные ситуации. Попытка возбудить любую другую исключительную ситуацию приведет к аварийному завершению работы программы.

Реализация абстрактного списка с учетом исключительных ситуаций

Перейдем теперь к реализации абстрактного списка с учетом исключительных ситуаций. В исходной реализации признак *success* использовался в качестве индикатора успешного выполнения операции. В новой реализации при неудачном выполнении операции будут генерироваться исключительные ситуации.

Класс *List* предусматривает два типа ошибок, сопровождающихся возникновением исключительных ситуаций: выход индекса списка за пределы допустимого диапазона значений и попытка вставить элемент в полный список. Попытка удалить или извлечь элемент из пустого списка будет обрабатываться как ошибка, связанная с выходом индекса за пределы допустимого диапазона.

Рассмотрим определение класса *listIndexOutOfRangeException*, который будет использоваться при обработке ошибок первого типа. Он основан на более универсальном классе *out_of_range* из стандартной библиотеки языка C++.

```
#include <stdexcept>
#include <string>
using namespace std;
class ListIndexOutOfRangeException: public out_of_range
{
public:
   ListIndexOutOfRangeException(const string & message = "")
                        : out_of_range(message.c_str())
{ }
}; // Конец определения класса end ListIndexOutOfRangeException
```

Ниже приводится определение исключительной ситуации *ListException*, используемой для обработки переполненного списка.

```
#include <exception>
#include <string>
using namespace std;
class ListException: public exception
{
public:
   ListException(const string & message = "")
        : exception(message.c_str())
{ }
}; // Конец определения класса ListException
```

Теперь мы можем определить класс *List*, представленный ранее, с учетом исключительных ситуаций.

```
// ************************************************************
// Заголовочный файл ListA.h для реализации абстрактного
// списка в виде массива с учетом исключительных ситуаций
// ************************************************************
#include "ListException.h"
#include "ListIndexOutOfRangeException.h"
const int MAX_LIST = максимальная_длина_списка;
typedef desired-type-of-list-item ListItemType;

class List
{
public:
 List(); // Конструктор по умолчанию
         // Используется автоматический деструктор

// Операции над списком:
bool isEmpty() const;
// Исключительная ситуация: нет

int getLength() const;
// Исключительная ситуация: нет

void insert(int index, ListItemType newItem)
            throw(ListIndexOutOfRangeException, ListException);
// Исключительная ситуация: генерирует исключительную ситуацию
```

```
// класса ListIndexOutOfRangeException, если index < 1 или
// index > getLength()+1.
// Исключительная ситуация: если элемент newItem нельзя
// вставить в список, поскольку массив переполнен, генерируется
// исключительная ситуация типа ListException.

void remove(int index)
   throw(ListIndexOutOfRangeException);
// Исключительная ситуация: генерирует исключительную ситуацию
// класса ListIndexOutOfRangeException, если index < 1 или
// index > getLength()+1.

void retrieve(int index, ListItemType& dataItem,
              bool& success) const;
// Исключительная ситуация: генерирует исключительную ситуацию
// класса ListIndexOutOfRangeException, если index < 1 или
// index > getLength()+1.

private:
   ListItemType items[MAX_LIST]; // Массив элементов списка
   int size; // Количество элементов списка
   int translate(int index) const;
}; // Конец определения класса List
// Конец заголовочного файла.
```

Реализация функции *insert* приведена ниже. Реализация функций *remove* и *retrieve* (вместе с исключительными ситуациями) предлагается читателям в качестве упражнения.

```
void List::insert(int index, ListItemType newItem)
{
   if (size >= MAX_LIST)
   throw ListException("ListException: список переполнен");

   if (index >= 1 && index <= size+1)
   {
      for (int pos = size; pos >= index; --pos)
      items[translate(pos+1)] = items[translate(pos)];
      // insert new item
      items[translate(index)] = newItem;
      ++size; // Увеличивает текущую длину списка на 1
   }
   else // Индекс вышел за пределы допустимого диапазона
      throw ListIndexOutOfRangeException(
      "ListIndexOutOfRangeException: неверный индекс вставки");
   // Конец оператора if
} // Конец функции-члена insert
```

Резюме

1. Абстракция данных — это способ управления взаимодействием между программой и ее структурами данных. Абстракция данных позволяет возвести стену вокруг структур данных, так же как модульный подход возводит стены вокруг алгоритмов, реализованных в программе. Такие стены облегчают отладку, реализацию и модификацию программы и делают ее более читабельной.

2. Спецификации совокупности операций вместе с данными, которыми они манипулируют, образуют абстрактный тип данных (АТД).
3. Формальное математическое изучение абстрактных типов данных используется для определения операций системы аксиом.
4. К реализации абстрактного типа данных можно приступать только после его полного определения. Выбор структуры данных зависит как от деталей выполнения операций, так и от контекста, в котором они применяются.
5. Даже после выбора структуры данных необходимо стремиться к тому, чтобы остальная часть программы не зависела от принятого решения. Иными словами, доступ к структуре данных должен осуществляться только через операции АТД. Таким образом, реализация скрывается за стенами операций над абстрактным типом. Для воплощения этой концепции в языке C++ используются классы.
6. Объекты инкапсулируют данные и операции над ними. В языке C++ объекты представляют собой экземпляры классов, т.е. типов, определенных пользователем.
7. Класс в языке C++ обязан содержать хотя бы один конструктор, т.е. метод его инициализации, и деструктор, т.е. метод очистки памяти, уничтожающий объект по истечении времени его жизни.
8. Если в классе не определен ни один конструктор, компилятор генерирует автоматический конструктор, не имеющий аргументов. Если деструктор также не определен, компилятор генерирует автоматический деструктор. Для классов, описанных в этой главе, автоматического деструктора было вполне достаточно. В главе 4 эта тема рассматривается более подробно.
9. Члены класса по умолчанию считаются закрытыми, если они не объявлены как открытые явным образом. Клиент класса, т.е. программа, использующая данный класс, не может использовать в своей работе закрытые члены. Однако закрытые члены доступны реализациям функций-членов класса. Обычно данные-члены следует объявлять закрытыми, предусматривая для доступа к ним открытые функции.
10. Поскольку некоторые классы широко применяются во многих программах, их применение должно быть максимально удобным. Классы можно определять и реализовывать в заголовочных файлах и файлах реализации, которые можно включать в программу про мере необходимости.
11. Пространства имен представляют собой механизм для логической группировки связанных между собой классов, функций, переменных, типов и констант.
12. Если во время выполнения программы обнаружится ошибка, можно возбудить исключительную ситуацию. Говорят, что код, работающий в исключительной ситуации, перехватывает, или обрабатывает ее.

Предупреждения

1. Спроектировав класс, попытайтесь написать код, использующий его, перед тем как окончательно завершить разработку. Вы увидите не только, правильно ли работает ваша программа, но и проверите корректность всего вашего проекта и правильность сопровождающей его документации.
2. При реализации класса могут возникнуть проблемы, связанные как с проектированием, так и со спецификациями. В этом случае нужно изменить

проект и спецификации, проверить работу класса еще раз и продолжить реализацию. Эти указания полностью соответствуют этапам жизненного цикла программного обеспечения, которые обсуждались в главе 1.

3. Программа не должна зависеть от конкретной реализации абстрактного типа данных. Используя класс для реализации АТД, инкапсулируйте в объектах данные и операции. Таким образом, можно скрыть от программы детали реализации класса. В частности, объявление функций-членов закрытыми позволяет изменять реализацию класса, не прибегая к изменениям в клиентском коде.
4. Закрывая данные-члены класса, вы облегчаете процесс локализации ошибок в программе. Ответственность за работу с данными лежит на абстрактном типе, т.е на классе. Если возникает ошибка, ее источник следует искать в классе. Если бы клиентский код мог непосредственно манипулировать данными (как это происходит, когда они объявляются открытыми), ошибку пришлось бы искать по всей программе.
5. Если функция-член не предназначена для изменения данных-членов класса, объявляйте ее константной. Это предотвратит возможные ошибки при ее реализации.
6. Локальные переменные функций не должны быть членами класса.
7. Если в классе определен собственный конструктор, но пропущен конструктор по умолчанию, компилятор сам сгенерирует автоматический конструктор. В этом случае операторы типа `List myList;` использоваться не должны.
8. Реализация абстрактного списка в виде массива ограничивает количество его элементов. Таким образом, перед вставкой нового элемента реализация должна постоянно проверять, достаточно ли места в структуре данных, а клиент должен предусмотреть вариант, когда операция вставки окажется невозможной.
9. Исключительная ситуация, которая не перехватывается в блоке `try-catch`, может привести к аварийному завершению работы программы.

Вопросы для самопроверки

2. Что вы понимаете под словами "стена" и "контракт"? Как эти понятия помогают решить задачу?
3. Напишите псевдокод функции `swap(aList, i, j)`, которая меняет местами элементы списка под номерами `i` и `j`. Выразите эту функцию в терминах операций над абстрактным списком, так чтобы она не зависела от конкретной реализации списка. Допустим, что список действительно содержит на позициях `i` и `j` какие-то элементы. Как это влияет на решение задачи? (См. упражнение 2.)
4. Как изменится список бакалейных продуктов после применения к нему следующих операций?

   ```
   aList.createList()
   aList.insert(1, butter, success)
   aList.insert(1, eggs, success)
   aList.insert(1, milk, success)
   ```

5. Напишите спецификации списка, в котором операции вставки, удаления и извлечения элементов производятся только в хвосте.

6. Напишите пред- и постусловия для каждой операции над упорядоченным абстрактным списком.
7. Напишите псевдокод функции, создающей упорядоченный список *sortedList* из списка *aList*, используя операции над абстрактным и упорядоченным списками.
8. В спецификациях абстрактного и упорядоченного списков не упоминается случай, когда два или более элементов имеют одинаковые значения. Распространяются ли указанные спецификации на этот случай или их необходимо пересмотреть?

Упражнения

1. Рассмотрите абстрактный список, состоящий из целых чисел. Напишите функцию, вычисляющую сумму элементов, хранящихся в списке *aList*. Определение этой функции не должно зависеть от реализации списка.
2. Реализуйте функцию *swap*, описанную выше, игнорируя предположение, что i-й и j-й элементы списка существуют. Добавьте аргумент *success*, играющий роль индикатора успешного выполнения функции *swap*.
3. Используйте функцию *swap*, реализованную в упражнении 2, и напишите функцию, изменяющую порядок следования элементов списка *aList* на противоположный.
4. В разделе "Абстрактный список" описаны функции *displayList* и *replace*. Как следует из этой главы, эти операции не относятся к абстрактному списку, т.е. не входят в список его операций. Вместо этого, их реализация выражается в терминах операций над абстрактным списком.

   4.1. В чем преимущества и недостатки такой реализации функции *displayList* и *replace*?

   4.2. В чем преимущества и недостатки включения функций *displayList* и *replace* в список операций над абстрактным списком?
5. С математической точки зрения, **множество** (set) — это совокупность разных элементов. Создайте спецификации операций проверки равенства, образования подмножества, объединения и пересечения частей абстрактного множества.
6. Создайте спецификации операций над абстрактными строками символов. Включите в них типичные операции, например, вычисление длины строки и конкатенацию (склеивание двух строк).
7. Напишите псевдокод функции в терминах АТД "Записная книжка", описанный в главе, для каждой из следующих задач.

   7.1. Изменить цель встречи, назначенной на заданную дату и время.

   7.2. Отобразить на экране все встречи, назначенные на указанную дату.

   Нужна ли для выполнения этих задач операция "добавить"?
8. Рассмотрите АТД "Полином" — от одной переменной x, — включающий в себя следующие операции.

```
degree()
// Возвращает степень полинома.
coefficient(power)
// Возвращает коэффициент при члене x^power.
```

```
changeCoefficient(newCoefficient, power)
// Заменяет коэффициент при члене x^power на аргумент
// newCoefficient.
```

Рассмотрите только полиномы с неотрицательными степенями, например $p = 4x^5 + 7x^3 - x^2 + 9$.

Следующие примеры демонстрируют результаты применения абстрактных операций к этому полиному:

`p.degree()` равно 5 (наивысшая степень члена с ненулевым коэффициентом);

`p.coefficient(3)` равно 7 (коэффициент при члене x^3);

`p.coefficient(4)` равно 0 (коэффициент при отсутствующем члене равен 0);

`p.changecoefficient(-3, 7)` порождает полином $p = -3x^7 + 4x^5 + 7x^3 - x^2 + 9$.

Используя указанные абстрактные операции, напишите операторы, выполняющие следующие задания.

8.1. Вывести на экран коэффициент при члене, имеющем наивысшую степень.

8.2. Увеличить коэффициент при члене $x3$ на 8.

8.3. Вычислить сумму двух полиномов.

9. Напишите псевдокод реализаций операций над абстрактными полиномами, определенными в упражнении 8, в терминах операций над абстрактными списками.

10. Представьте себе неизвестную реализацию абстрактного упорядоченного списка, состоящего из целых чисел. Элементы списка упорядочены в порядке возрастания. Допустим, что мы считали n целых чисел в одномерный массив с именем `data`. Напишите на C++ фрагмент программы, использующей операции над абстрактным упорядоченным списком для сортировки массива в порядке возрастания.

11. Используя аксиомы абстрактного списка, приведенные в разделе "Аксиомы", докажите, что последовательность операций

```
вставить элемент A в позицию 2,
вставить элемент B в позицию 2,
вставить элемент C в позицию 2,
```

примененная к непустому списку, эквивалентна следующей последовательности операций:

```
вставить элемент C в позицию 2,
вставить элемент B в позицию 2,
вставить элемент A в позицию 2.
```

12. Определите совокупность аксиом для абстрактного упорядоченного списка и примените их для доказательства того факта, что упорядоченный список символов, определенный последовательностью операций

```
создать пустой упорядоченный список,
вставить элемент S,
вставить элемент T,
вставить элемент R,
вставить элемент T
```

абсолютно эквивалентен упорядоченному списку, полученному в результате выполнения последовательности операций

```
создать пустой упорядоченный список,
вставить элемент T,
вставить элемент R,
вставить элемент T,
вставить элемент S.
```

13. Повторите упражнение 17 из главы 2, используя вариант абстрактного списка для реализации функции $f(n)$.
14. Напишите псевдокод для функции слияния двух списков в третий упорядоченный список, используя лишь операции над абстрактным упорядоченным списком.
15. Реализуйте функции `retrieve` и `remove` из класса `List`, учитывая возможные исключительные ситуации.

Задания по программированию

1. Разработайте и реализуйте абстрактный тип данных для представления треугольника. Данные этого типа должны включать в себя стороны треугольника, а также величины его углов. Эти данные должны быть описаны в закрытом разделе класса, реализующего этот абстрактный тип.

   Предусмотрите по крайней мере две операции инициализации: одну — для задания значений по умолчанию, а другую — по усмотрению клиента. Эти операции нужно реализовать в виде конструкторов класса.

   Абстрактный тип должен предусматривать также операции просмотра и изменения данных, вычисления площади треугольника, а также определения, является ли треугольник прямоугольным, равносторонним или равнобедренным.
2. Разработайте и реализуйте абстрактный тип данных, представляющий время дня. Будем предполагать, что время задается в часах и минутах на основе 24-часового циферблата. Часы и минуты задаются закрытыми членами класса, реализующего этот АТД.

   Предусмотрите по крайней мере две операции инициализации: одну — для задания времени по умолчанию, а другую — по усмотрению клиента. Эти операции нужно реализовать в виде конструкторов класса.

   Реализуйте операции установки времени, увеличения текущего времени на заданное количество минут и отображения времени с помощью 12- и 24-часового циферблата.
3. Разработайте и реализуйте абстрактный тип данных для представления календаря. Он должен представлять день, месяц и год в виде целых чисел (например, 1/4/2002.) Предусмотрите операции увеличения даты на 1 и отображения даты, задавая месяц с помощью цифр или слов.
4. Разработайте и реализуйте абстрактный тип данных для представления стоимости вещи, выраженной в долларах и центах. После завершения реализации этого АТД напишите клиентскую функцию, вычисляющую сдачу, если за предмет стоимостью y долларов заплачено x долларов.
5. Определите класс реализации абстрактного упорядоченного списка в виде массива. Рассмотрите рекурсивную реализацию функции `locatePosition`.

Должны ли функции *sortedInsert* и *sortedRemove* вызывать функцию *locatePosition*?

6. Напишите рекурсивную реализацию функции вставки, удаления и извлечения элемента из абстрактного списка и абстрактного упорядоченного списка.
7. Реализуйте АТД "Множество", описанный в упражнении 5, используя лишь массивы и простые переменные.
8. Реализуйте АТД "Строка символов", описанный в упражнении 6.
9. Реализуйте АТД "Полином", описанный в упражнении 8.
10. Реализуйте АТД "Записная книжка", описанный в разделе "Разработка абстрактных типов данных". При необходимости дополните список его операций. Например, нужно добавить операции чтения и записи данных о встречах.
11. Выполните следующие задания.

    11.1. Опишите и реализуйте АТД "Рациональное число". Предусмотрите операции чтения, записи, сложения, вычитания, умножения и деления рациональных чисел. Результат всех этих операций должен быть приведен к младшему члену с помощью закрытой функции *reduceToLowestTerms*. Разобраться в деталях этой функции позволит упражнение 20 из главы 2. (Должны ли операции чтения и записи использовать функцию *reduceToLowestTerms*?) Для простоты можете считать, что знаменатель дроби положителен.

    11.2. Опишите и реализуйте АТД "Смешанное число", которое состоит из целой и дробной части, приведенной к младшему члену. Будем предполагать, что АТД "Дробь" уже существует. Предусмотрите операции чтения, записи, сложения, вычитания, умножения и деления смешанных чисел. Дробные части результатов всех арифметических операций должны быть приведены к младшему члену. Предусмотрите также операцию для преобразования дроби в смешанное число.

    11.3. Реализуйте АТД "Книга рецептов", описанный в разделе "Разработка абстрактных типов данных", наряду с АТД "Единица измерения". При необходимости дополните список его операций. Например, нужно добавить операции чтения, записи и масштабирования рецептов.

12. Выполните заново задание 2 из главы 1, используя знания об АТД и классы.
13. Выполните заново задание 3 из главы 1, используя знания об АТД и классы.

Связанные списки

В этой главе ...

Предварительные замечания
- Указатели
- Динамические массивы
- Связанные списки, основанные на указателях

Работа со связанными списками
- Вывод на экран содержания связанного списка
- Удаление указанного узла из связанного списка
- Вставка узла в указанную позицию связанного списка
- Реализация абстрактного списка, основанная на указателях
- Реализации списка в виде массива и на основе указателей
- Запись связанных списков в файл и считывание их
- Передача связанного списка в качестве аргумента функции
- Рекурсивная обработка связанных списков
- Объекты, хранящиеся в узлах списка

Разновидности связанных списков
- Кольцевые связанные списки
- Фиктивные головные узлы
- Дважды связанные списки

Приложение: инвентарная ведомость

Стандартная библиотека шаблонов языка C++
- Контейнеры
- Итераторы
- Шаблонный класс list из библиотеки STL

Резюме

Предупреждения

Вопросы для самопроверки

Упражнения

Задания по программированию

Введение. В этой главе рассматриваются указатели и связанные списки. В ней описываются алгоритмы выполнения основных операций над связанными списками, таких как вставка и удаление элементов. Кроме того, в главе приводятся несколько вариантов связанного списка и показывается, как их можно применять для реализации многих абстрактных типов данных, рассмотренных в книге. Материал, изложенный в этой главе, очень важен для дальнейшего изложения.

Предварительные замечания

Абстрактный список, описанный в предыдущей главе, предусматривал операции вставки, удаления и извлечения элементов по их позициям. Детальное изучение реализации абстрактного списка в виде массива показывает, что массив не всегда подходит для хранения набора данных. Массив имеет **фиксированный размер** (по крайней мере в большинстве языков программирования), в то время как длина абстрактного списка не ограничена. Таким образом, строго говоря, массив нельзя использовать для реализации списка, поскольку потенциальное количество элементов списка может превзойти фиксированный размер массива. Эта проблема часто возникает при реализации абстрактных типов данных. Во многих случаях следует предпочесть реализацию с переменным размером.

Разработчики интуитивно стремятся хранить данные в последовательно расположенных ячейках, хотя такой подход имеет ряд недостатков. В этом случае элемент x и его преемник располагаются в смежных ячейках. Как мы уже видели, это приводит к тому, что при выполнении операций вставки и удаления приходится сдвигать элементы массива, затрачивая дополнительное время. Посмотрим, каковы альтернативные решения этой проблемы.

Чтобы понять принципы реализации списка, не использующей сдвига элементов, посмотрите на рис. 4.1. Этот рисунок должен помочь вам избавиться от предубеждения, что единственным способом хранения данных является их запись в смежных ячейках. Каждый элемент списка, изображенного на этой диаграмме, фактически *указывает* на следующий элемент. Таким образом, у каждого элемента известен его преемник, где бы он ни находился. Это позволяет не только вставлять и удалять элементы, не прибегая к сдвигу данных, но и легко изменять длину списка. Для того чтобы вставить новый элемент, достаточно найти его место в списке и задать два **указателя** (pointers). Аналогично, чтобы удалить элемент из списка, нужно изменить значение указателя его предшественника.

Элемент связанного списка указывает на своего преемника

Поскольку элементы в этой структуре данных связаны друг с другом, она называется **связанным списком** (linked list). *Связанный список может неограниченно возрастать*, в то время как массив способен хранить лишь фиксированное количество элементов. Во многих приложениях эта гибкость связанных списков обеспечивает им большое преимущество над остальными структурами данных.

Прежде чем перейти к изучению связанных списков и способов их применения для реализации АТД, нужно побольше узнать об указателях. Как и во многих языках программирования, в языке C++ предусмотрены указатели, которые можно использовать для создания связанных списков.

Указатели

Если в программе на языке C++ объявлена обычная переменная, имеющая тип `int`, компилятор выделяет для ее хранения отдельную ячейку памяти. Для обращения к этой ячейке используется идентификатор x. Чтобы поместить в нее число 5, можно написать оператор

```
x = 5;
```

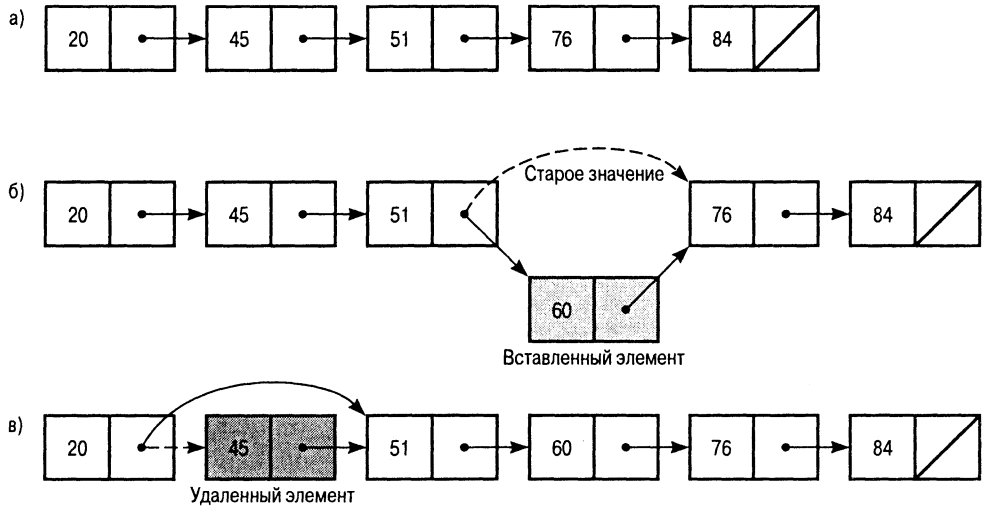


Рис. 4.1. Принципы реализации связанных списков: а) связанный список целых чисел; б) вставка; в) удаление

Чтобы вывести на экран значение, хранящееся в этой ячейке, можно воспользоваться оператором

```
cout << «Значение переменной x равно « << x << endl;
```

Переменная-указатель (pointer variable), или просто **указатель** (pointer), хранит информацию о местоположении, или **адрес** (address) ячейки памяти. Используя указатель на конкретную ячейку памяти, можно определять местонахождение ячейки и, например, просматривать ее содержимое.

На рис. 4.2 приведен указатель *p*, ссылающийся на ячейку памяти, содержащую целое число.

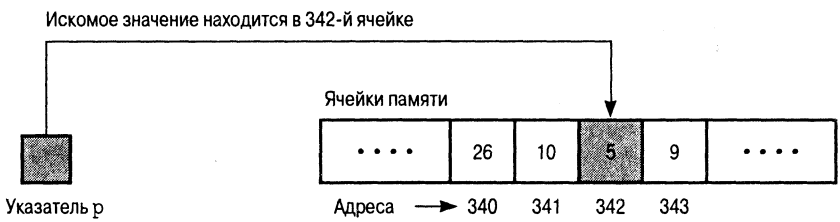


Рис. 4.2. Указатель на целое число

Понятие о ячейке памяти, ссылающейся на другую ячейку памяти, довольно хитроумно. Следует иметь в виду, что содержимое указателя *p*, изображенного на рис. 4.2, — не обычное число. Это значение представляет собой информацию о местонахождении в памяти целого числа 5. Таким образом, доступ к числу 5 можно получить *косвенным* путем, используя адрес, содержащийся в указателе *p*.

С указателями связаны два важных вопроса.

- Как сделать так, чтобы указатель *p* ссылался на заданную ячейку памяти?
- Как с помощью указателя *p* получить доступ к содержимому ячейки памяти, на которую он ссылается?

Прежде всего нужно объявить переменную *p* как указатель. Например, приведенное ниже объявление означает, что переменная *p* является указателем целого типа, т.е. указатель *p* может ссылаться лишь на ячейки памяти, которые содержат целые числа. Указатели могут ссылаться на данные любых типов, кроме файлов.

```
int *p;
```

Переменная p — это указатель

Объявляя несколько указателей, следует быть внимательным. Например, в приведенном ниже объявлении переменная *p* является указателем на целое число, а переменная *q* — это обычная целочисленная переменная.

```
int *p, q;
```

Переменная q — это не указатель

Иными словами, это объявление эквивалентно следующему.

```
int *p;
int q;
```

Чтобы правильно объявить указателями обе переменные, нужно написать

```
int *p;
int *q;
```

или[1]

```
int *p, *q;
```

Память для указателей *p* и *q*, а также для целочисленной переменной *x*, заданной с помощью объявления

```
int x;
```

выделяется во время компиляции, т.е. до начала выполнения программы. Такой механизм распределения памяти называется **статическим** (static allocation), а переменные, соответственно, **статическими** (statically allocated). Выполнение программы не влияет на размеры памяти, выделенной для статических переменных.

В исходном положении, как показано на рис. 4.3, *а*, содержимое переменных *p*, *q* и *x* остается неопределенным. Однако переменной *p* можно присвоить адрес переменной *x*, и тогда указатель *p* станет ссылаться на переменную *x*. Для этого нужно применить **оператор взятия адреса** & (address-of operator).

```
p = &x;
```

На рис. 4.3, *б* показан результат этого присваивания. Обратите внимание, что использовать оператор

```
p = x; // ЭТО ОШИБКА
```

ни в коем случае нельзя, поскольку он порождает конфликт типов: переменная *x* является целочисленной, а переменная *p* — указателем, в котором хранится адрес ячейки памяти, выделенной для целочисленной переменной.

Теперь указатель *p* ссылается на некую ячейку памяти. Выражение **p* означает: *"Ячейка памяти, на которую ссылается указатель p"*. Чтобы записать некое значение в ячейку памяти, на которую ссылается указатель *p*, можно применить оператор присваивания

Значение *p - это адрес ячейки, на которую ссылается указатель p

```
*p = 6;
```

[1] В отношении указателей оператор * является унарным (как, например, оператор !) и правоассоциативным. В объявлениях *int *p* или *int* p* оператор * применяется к переменной *p*, а не к данным, имеющим тип *int*.

как показано на рис. 4.3, *в*. (Разумеется, то же самое можно сделать и с помощью оператора *x = 6*.) После этого присваивания выражение **p* имеет значение 6, поскольку именно это число теперь записано в ячейку памяти, на которую ссылается указатель *p*. Таким образом, например, с помощью оператора *cout << *p;* можно вывести на экран число 6.

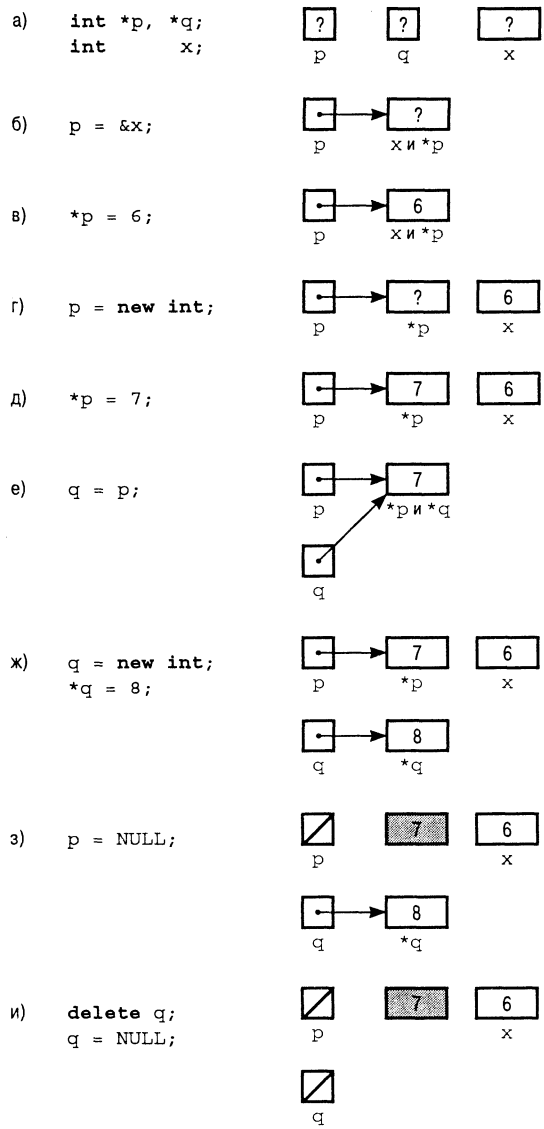


Рис. 4.3. Изменение указателей в ходе выполнения программы: а) объявление указателей; б) указатель на статически выделенную память; в) присваивание значения; г) динамическое выделение памяти; д) присваивание значения; е) копирование указателя; ж) динамическое выделение памяти и присваивание значения; з) присваивание указателю константы NULL; и) освобождение памяти

Память для переменных можно выделять и во время выполнения программы. Такой механизм называется **динамическим распределением памяти** (dynamic allocation). Переменные, память для которых выделяется в ходе выполнения программы, называются **динамическими** (dynamically allocated). Динамическое выделение памяти происходит с помощью оператора *new*, действующего на тип данных.

Оператор new выделяет память в ходе выполнения программы

```
p = new int;
```

Выражение *new int* выделяет новую ячейку памяти, в которой будет храниться целочисленная переменная, и возвращает указатель на нее, как показано на рис. 4.3, *г*. В исходном положении содержимое новой ячейки остается неопределенным. Обратите внимание, что выражение *new char* может выделять ячейку памяти для хранения данных типа *char* и т.д.

Учтите, что вновь созданные ячейки памяти не имеют имен, заданных программистом. Единственный способ доступа к их содержимому обеспечивается косвенным образом через указатель, создаваемый оператором *new*, т.е. с помощью выражения **p*. Как показано на рис. 4.3, *д*, оператор **p = 7* помещает число 7 во вновь созданную ячейку памяти.

Допустим, что указателю *q* присваивается значение указателя *p*.

Копирование указателя

```
q = p;
```

Теперь указатель *q* ссылается на ту же ячейку памяти, что и указатель *p*, как показано на рис. 4.3, *е*. Вместо этого можно позволить указателю *q* ссылаться на новую ячейку памяти, как показано на рис. 4.3, *ж*, и записать в эту ячейку новое значение. Эти шаги проиллюстрированы на рис. 4.3, *д* и 4.3, *е*.

Допустим теперь, что значение указателя нам больше не нужно. Иными словами, нам больше не нужен указатель, ссылающийся на конкретную ячейку памяти. Для этой ситуации в языке C++ предусмотрена константа *NULL*[2], которую можно присваивать указателям любого типа. По умолчанию указатель, имеющий значений *NULL*, ни на что не ссылается. Не путайте указатель, имеющий значение *NULL*, и указатель, значение которого не определено! Пока вы явно не присвоите конкретное значение вновь объявленному указателю, его содержимое остается неопределенным, как это происходит с любой другой переменной. Не следует предполагать, что это неопределенное значение равно константе *NULL*. На рис. 4.3, *а* показаны примеры указателей *p* и *q*, значения которых не определены.

Допустим теперь, что нам больше не нужна ячейка динамической памяти. Если изменить значения указателей, ссылающихся на эту ячейку, это приведет в бессмысленному расходу памяти, поскольку сама ячейка будет существовать, даже будучи недоступной. Например, на рис. 4.3, *з* показан результат присвоения указателю *p* константы *NULL*. (На всех рисунках диагональная линия обозначает константу *NULL*.) Ячейка, на которую до сих пор ссылался указатель *p* (в ней по-прежнему записано число 7), теперь оказалась потерянной навсегда. Чтобы избежать этой ситуации, называемой **утечкой памяти** (memory leak), в языке C++ предусмотрен оператор *delete*, являющийся дополнением оператора *new*.

По определению выражение

Оператор delete освобождает память

```
delete q;
```

[2] Значение этой константы определяется в нескольких заголовочных файлах, например, *cstdlib* и (довольно часто) *cstddef*. Ее значение равно 0. Многие программисты, работающие с языком C++, предпочитают использовать число 0 вместо константы *NULL*. Однако для большей ясности в книге используется константа *NULL*.

освобождает ячейку памяти, на которую ссылался указатель *q*. Таким образом, эта ячейка становится доступной для повторного использования в дальнейшем. Поскольку оператор *delete* не удаляет сам указатель *q* и оставляет его содержимое неопределенным, ссылка на значение этого указателя **q* становится небезопасной и может иметь разрушительные последствия. Таким образом, после применения оператора *delete* для предотвращения ссылки на удаленную ячейку указателю *q* необходимо присвоить константу *NULL*. Результаты этих действий показаны на рис. 4.3, *и*.

Указатель на удаленную ячейку памяти может представлять опасность

Рассмотрим теперь следующую ситуацию, проиллюстрированную на рис. 4.4.

```
p = new int;
q = p;
delete p;
p = NULL;
```

Несмотря на то что указатель *p* имеет значение *NULL*, указатель *q* продолжает ссылаться на удаленный узел. Позднее система может выделить эту же ячейку памяти вновь — с помощью оператора *new* — и окажется, что указатель *q* по-прежнему ссылается на нее. Работа программы в этом случае станет непредсказуемой, поскольку указатель *q* больше не имеет никакого отношения к нашей структуре данных! Во избежание подобных последствий, нужно чтобы оператор *delete* обнулял все ссылки на удаляемую ячейку памяти. К сожалению, этого очень трудно добиться. Поскольку оператор *delete* не способен определять, какие еще переменные, кроме указателя *p*, ссылаются на удаляемый узел, ответственность за правильное решение этой задачи возлагается на программиста.

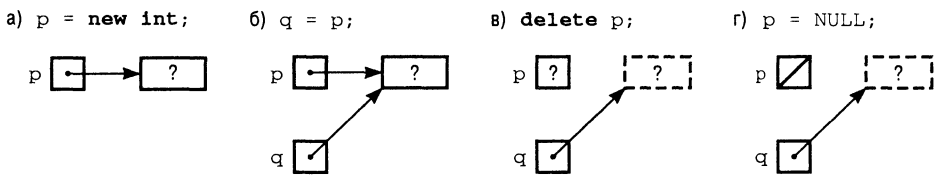


Рис. 4.4. Некорректный указатель на удаленный узел

Работа с указателями проиллюстрирована на рис. 4.5. Реализации АТД и структуры данных, использующие указатели, называются **основанными на указателях** (pointer-based).

ОСНОВНЫЕ ПОНЯТИЯ

Указатели

1. Объявление

```
int *p;
```

выделяет статическую память для указателя *p*, значение которого остается неопределенным и не равно константе *NULL*. В данном примере указатель *p* может ссылаться на ячейки памяти, содержащие целочисленные переменные.

2. Оператор

```
p = new int;
```

статически выделяет новую ячейку памяти, которая может хранить целое число. Указатель *p* ссылается на эту ячейку. (Однако учтите п. 5 этого списка.)

3. Выражение *p обозначает ячейку памяти, на которую ссылается указатель *p*.
4. Если указатель *p* содержит константу *NULL*, он ни на что не ссылается.
5. Если по некоторым причинам оператор *new* не может выделить память, он возвращает указатель *NULL*. Таким образом, после выполнения оператора

```
p = new int;
```

можно сравнить значение указателя *p* с константой *NULL*, чтобы удостовериться, успешно ли была выделена память.

6. Оператор

```
delete p;
```

освобождает ячейку памяти, на которую ссылается указатель *p*. Сам указатель *p* при этом не уничтожается. Помните, что указатель *p* — это переменная, и оператор *delete* не может повлиять на продолжительность ее жизни.

Динамические массивы

Рассмотрим следующий фрагмент программы.

В языке C++ память для обычных массивов выделяется статически

```
const int MAX_SIZE = 50;
double anArray[MAX_SIZE];
```

Обнаружив такое объявление, компилятор зарезервирует для массива указанное число ячеек — в данном случае *MAX_SIZE*. Это произойдет до начала выполнения программы, поэтому значение константы *MAX_SIZE* нужно знать заранее. Проблемы, связанные со структурами данных, имеющими фиксированный размер, мы уже обсуждали выше.

Для динамического выделения памяти во время выполнения программы можно применять оператор *new*. В предыдущем разделе выделялась память для отдельной переменной. Теперь покажем, как это можно сделать для многих переменных одновременно.

Предположим, что для размещения массива мы написали следующий фрагмент программы.

Для динамического размещения массивов используется оператор new

```
int arraySize = 50;
double *anArray = new dou-
ble[arraySize];
```

Указатель *anArray* указывает на первый элемент массива, состоящего из 50 элементов. В отличие от константы *MAX_SIZE*, переменная *arraySize* может изменяться в ходе выполнения программы. Ей можно присвоить конкретное значение, определив размер массива динамически. Хорошо, а как работать с таким массивом?

Независимо от способа размещения массива в памяти — статического, как в первом примере, или динамического, как во втором — для доступа к его элементам можно использовать индексы. Например, выражения *anArray[0]* и *anArray[1]* обозначают первый и второй элементы массива *anArray*.

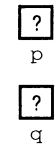
Кроме того, для доступа к любому элементу массива можно использовать указатели. В языке C++ имя массива интерпретируется как имя указателя на его первый элемент. Например,

Имя массива является именем указателя на его первый элемент

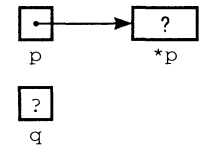
```
выражение *anArray эквивалентно выражению anArray[0], а
выражение *(anArray+1) эквивалентно выражению anArray[1] и т.д.
```

Эта система обозначений называется **арифметикой указателей** (pointer arithmetic). Однако мы не рекомендуем ее использовать.

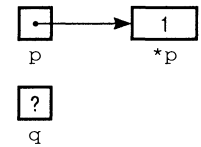
```
int *p, *q;
```



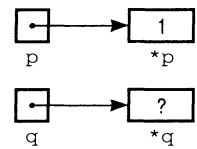

```
p = new int;          // Выделяется ячейка
                      // для целочисленной переменной.
```



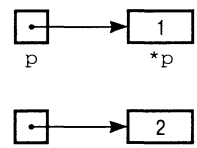

```
*p = 1;               // В новую ячейку записывается
                      // некое значение.
```



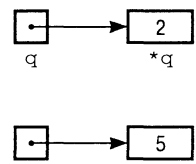

```
q = new int;          // Выделяется ячейка
                      // для целочисленной переменной.
```



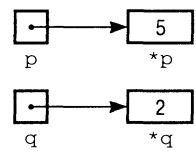

```
*q = 2;               // В новую ячейку записывается
                      // некое значение.
```




```
cout << *p << " "     // Строка вывода: 1 2
     << *q << endl;   // Эти значения находятся
                      // в ячейках, на которые ссылаются
                      // указатели p и q.
```

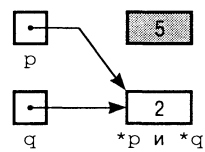



```
*p = *q + 3;          // Значение, записанное в ячейке,
                      // на которую ссылается указатель q,
                      // равно 2. К ней добавляется число 3.
                      // Результат заносится в ячейку,
                      // на которую ссылается указатель p.
```




```
cout << *p << " "     // Строка вывода: 5 2
     << *q << endl;
```

```
p = q;                // Теперь указатели p и q ссылаются
                      // на одну и ту же ячейку. Ячейка,
                      // на которую прежде ссылался
                      // указатель p, потеряна навсегда.
                      // Ее уже нельзя будет использовать
                      // снова.
```



```
cout << *p << " "    // Строка вывода: 2 2
     << *q << endl;

*p = 7;              // Ячейка, на которую ссылается
                     // указатель p (и указатель q)
                     // теперь содержит число 7.

cout << *p << " "    // Строка вывода: 7 7
     << *q << endl;

p = new int;         // Этот оператор касается только
                     // ячейки, на которую ссылается
                     // указатель p, но не q.

delete p ;           // Освобождает ячейку, на которую
                     // ссылался указатель p.

p = NULL;            // Присваивает указателю p константу
                     // NULL.
                     // Это следует делать после каждого
                     // выполнения оператора delete.

q = NULL;            // Ячейка, на которую прежде ссылался
                     // указатель q, теперь потеряна
                     // навсегда. Теперь к ней невозможно
                     // обратиться.
```

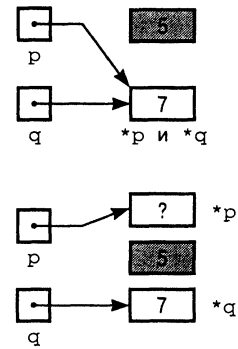


Рис. 4.5. Работа с указателями и динамически выделяемой памятью

Оператор delete удаляет динамический массив из памяти

Если для массива была выделена динамическая память, ее нужно освободить, когда массив станет не нужен. Как и в предыдущем разделе, для выполнения этой работы используется оператор *delete*. Чтобы удалить массив *anArray*, нужно выполнить следующий оператор.

```
delete [] anArray;
```

Для удаления массива после ключевого слова *delete* нужно поставить квадратные скобки.

Размер динамического массива можно увеличивать

Допустим, что в ходе выполнения программы массив *anArray* оказался полностью заполненным. В таком случае можно выделить память для нового, более крупного массива, скопировать в него элементы старого массива, а затем удалить старый массив из памяти. Например, в приведенном ниже фрагменте программы размер массива *anArray* удваивается.

```
double* oldArray = anArray; // Копируем указатель на массив
anArray = new double[2*arraySize]; // Удваиваем размер массива
for (int index = 0; index < arraySize); ++index)
   anArray[inex] = oldArray[index[; // Копируем старый массив
delete [] oldArray;          // Удаляем старый массив
```

В дальнейшем по ходу книги мы будем работать как со статическими, так и с динамическими массивами. Реализация абстрактного списка, описанная в предыдущей главе, для простоты использовала статические массивы. Реализация такого списка на основе динамических массивов предлагается читателям в качестве самостоятельного упражнения.

Связанные списки, основанные на указателях

Связанный список, изображенный на рис. 4.1, *а*, содержит связанные друг с другом компоненты. Каждый компонент, обычно называемый **узлом** (node), содержит данные, например, целое число, и "указатель" на следующий элемент. Обычно этими "указателями" являются переменные-указатели, предусмотренные в языке C++. Несмотря на то что мы уже рассмотрели основные приемы работы с указателями, некоторые вопросы, связанные с их применением для создания связанных списков, могут показаться довольно сложными. Рассмотрим их подробнее.

Поскольку каждый узел связанного списка должен состоять из двух частей — данных и указателя на следующий элемент, — естественно представить его в виде структуры, особой конструкции, предусмотренной в языке C++. Один член структуры представляет собой данные, а второй — указатель на следующий узел списка, как показано на рис. 4.6.

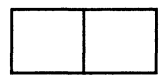


Рис. 4.6. Узел

Допустим, что в каждом узле хранится целое число. Какой тип должен иметь указатель, записанный в этом узле, и на что он должен ссылаться? Возможно, вы подумали, что указатель должен ссылаться на целое число. Вы ошиблись! Указатель должен ссылаться на структуру, поскольку узлами связанного списка являются структуры, а не целые числа. Поскольку указатели могут ссылаться на любой тип данных, кроме файлов, они могут ссылаться и на структуры. Таким образом, структура типа *Node*, например, будет состоять из целого числа и указателя на следующую структуру типа *Node*.

Описание структуры типа *Node* выглядит следующим образом.

Узел связанного списка обычно является структурой

```
struct Node
{
   int item;
   Node *next;
}; // Конец описания структуры
```

Теперь можно определить указатель *p* на структуру типа *Node*.

Определение указателя на узел

```
Node *p;
```

Узлы связанного списка должны размещаться в динамической памяти. Например, чтобы разместить в памяти узел типа *Node*, нужно выполнить оператор

Динамическое размещение узлов

```
p = new Node;
```

Теперь нам нужно получить доступ к членам этой структуры. Для этого нам нужна новая

Ссылка на член узла

синтаксическая конструкция, поскольку узлы не имеют имен. Например, для ссылки на член *item*, принадлежащий узлу, на который ссылается указатель *p*, нужно выполнить следующий оператор[3]

```
p->item
```

Завершая полное описание связанного списка, мы должны рассмотреть еще два вопроса. Во-первых, чему равно значение члена *next* в последнем узле списка? Установив этот член равным константе *NULL*, можно легко распознать конец списка.

Указатель на голову ссылается на первый элемент связанного списка

Во-вторых, как распознать начало списка? Если мы не получим доступ к началу списка, мы не сможем обнаружить второй узел. Не найдя второго узла, мы никогда не попадем в третий узел и т.д. Для того чтобы решить эту проблему, нужно предусмотреть отдельный указатель, единственным предназначением которого является хранение адреса первого элемента связанного списка. Такой узел называется **головой списка** (head), а соответствующий указатель — **указателем на голову** связанного списка (head pointer).

На рис. 4.7 показан связанный список с указателем на его голову. Напомним, что каждый указатель в связанном списке ссылается на структуру, иными словами, на целый узел.

Указатель на голову списка, изображенный на рис. 4.7, назван именем *head*. Он отличается от остальных указателей на узлы списка и не принадлежит ни одной из структур. Это обычный указатель, внешний по отношению к списку, в то время как члены *next* являются внутренними указателями и принадлежат узлам. Переменная *head* просто предоставляет доступ к началу списка. Кроме того, обратите внимание, что указатель *head* существует всегда, даже когда в списке нет ни одного узла.

Оператор

```
Node *head;
```

Если указатель head равен константе NULL, связанный список пуст

создает переменную *head*, значение которой, подобно другим неинициализированным переменным, никак не определено. Каким значением следует инициализировать указатель *head* и какое значение ему нужно присвоить, когда список станет пустым? В обоих случаях указателю *head* следует присвоить константу *NULL*, отметив тот факт, что этот указатель ни на что не ссылается.

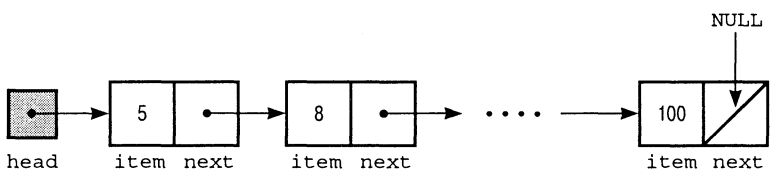


Рис. 4.7. Указатель на голову списка

Распространенная ошибка

Широко распространено заблуждение, что перед присваиванием указателю *head* конкретного значения нужно выполнить оператор *head = new Node*. Эта ошибка возникает из-за убеждения, что переменная *head* не существует, пока не вызван опе-

[3] Можно воспользоваться и альтернативным обозначением *(*p).item*, однако мы его применять не будем. Несмотря на то что это выражение приводит к тому же самому результату, оператор -> более нагляден.

ратор *new*. Это не совсем так. Указателю *head* нужно просто присвоить конкретное значение. Таким образом, ему можно присвоить константу *NULL*, не выполняя перед этим оператор *new*. Фактически последовательность операторов

```
head = new Node; // Неправильное использование оператора new
head = NULL;
```

разрушает содержимое только одного указателя — *head* — на вновь созданный узел, как показано на рис. 4.8. Нет никакой необходимости создавать новый узел, а затем делать его недоступным. Помните, что каждый раз, вызывая оператор *new* для выделения памяти, вам следует в конце работы освобождать ее с помощью оператора *delete*.

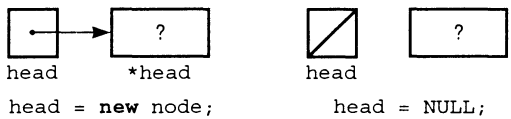


Рис. 4.8. Потерянная ячейка

Как указывалось выше, для реализации связанного списка не обязательно использовать исключительно указатели. В девятом задании по программированию, приведенном в конце главы, для представления связанного списка используется массив. Несмотря на то что иногда такая реализация оказывается полезной, она все же редко применяется.

Работа со связанными списками

В предыдущем разделе было описано применение указателей при реализации связанных списков. Теперь мы приступаем к разработке алгоритмов, предназначенных для вывода на экран, вставки и удаления элементов, хранящихся в связанном списке. Эти операции образуют основу многих структур данных, которые будут рассмотрены в остальных главах. Таким образом, материал, изложенный в этом разделе, весьма важен.

Вывод на экран содержания связанного списка

Допустим, что мы работаем со связанным списком, изображенным на рис. 4.7, и хотим вывести на экран хранящиеся в нем данные. Псевдокод решения этой задачи выглядит следующим образом.

```
Установить текущий указатель на первый узел связанного списка
while (текущий указатель не равен константе NULL)
{
  Вывести на экран данные, хранящиеся на первом узле
  Установить текущий указатель на следующий узел
} // Конец цикла while
```

Для решения этой задачи нужно отслеживать положение текущего указателя в связанном списке. Таким образом, нам понадобится переменная-указатель *cur*, ссылающаяся на текущий узел. В исходном положении указатель *cur* должен быть установлен на первый узел. Поскольку на первый узел списка ссылается указатель *head*, его можно просто скопировать в указатель *cur* с помощью оператора

```
Node *cur = head;
```

Для вывода на экран данных из текущего узла можно использовать оператор

```
cur << cur->item << endl;
```

Чтобы перейти на следующий узел, нужно выполнить оператор

```
cur = cur->next;
```

Это действие проиллюстрировано на рис. 4.9. Если операторы, приведенные выше, оказались не вполне понятными, рассмотрите следующий фрагмент программы.

```
temp = cur->next;
cur = temp;
```

Проанализировав эти операторы, вы поймете, что временная переменная *temp* не нужна.

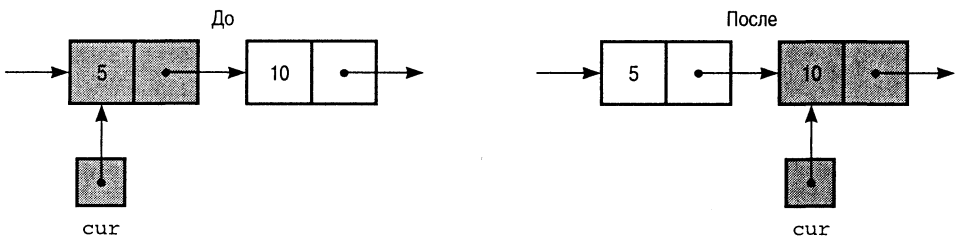


Рис. 4.9. Эффект присваивания cur->cur->next

Итак, изложенные выше соображения приводят нас к следующему коду на языке C++.

```
// Вывести данные, хранящиеся в связанном списке,
// на который ссылается указатель head.
// Инвариант цикла: указатель cur ссылается на следующий узел,
// подлежащий выводу на экран
for (Node *cur = head; cur != NULL; cur = cur-> next;
   cout << cur->item << endl;
```

В ходе выполнения цикла *for* указатель *cur* пробегает каждый узел непустого связанного списка, при этом на экран выводятся данные, хранящиеся в этих узлах. После того как на экран будут выведены данные из последнего узла, указатель *cur* станет равным константе *NULL*, и выполнение цикла *for* будет завершено. Если список пуст, т.е. указатель *head* равен константе *NULL*, цикл *for* не выполняется.

Работая с циклом *for*, неопытные программисты часто делают характерную ошибку: они сравнивают с константой *NULL* не текущий указатель *cur*, а указатель на следующий узел *cur->next*. Когда указатель *cur* устанавливается на последний узел непустого связанного списка, указатель *cur->next* оказывается равным константе *NULL*, поэтому цикл *for* завершается раньше, чем нужно: на предпоследнем узле. Кроме того, если список пуст, т.е. указатели *head* и *cur* равны константе *NULL*, указатель *cur->next* оказывается неопределенным. Такие ссылки являются неправильными, их следует избегать.

При обходе списка просматривается каждый узел связанного списка

Вывод на экран содержимого связанного списка представляет собой пример распространенной операции обхода списка (list traversal). При обходе списка просматривается каждый его узел, пока не будет обнаружен последний. В приведенном выше примере в ходе просмотра каждого узла на эк-

ран выводились данные, хранящиеся в нем. Позднее мы увидим, что при посещении узла с ним можно производить и другие манипуляции.

Вывод данных на экран не изменяет содержимого связанного списка. Чтобы модифицировать список, нужно выполнить операции вставки и удаления элементов, причем для этого список должен быть заранее создан. Позднее мы увидим, как можно создать новый список, используя операцию вставки элементов в пустой список.

Удаление указанного узла из связанного списка

Для того чтобы сосредоточить внимание на операции удаления конкретного узла, будем считать, что связанный список, приведенный на рис. 4.10, уже существует. Обратите внимание, что в дополнение к указателю *head* на диаграмме показаны два новых внешних указателя *cur* и *prev*, имеющие тип **Node*. Задача состоит в удалении узла, на который ссылается указатель *cur*. Как мы вскоре увидим, для выполнения операции удаления указатель *prev* необходим. Пока об этих указателях можно не беспокоиться.

Как показано на рис. 4.10, узел *N*, на который ссылается указатель *cur*, можно удалить, изменив значение указателя *next* в предыдущем узле: достаточно присвоить этому указателю адрес узла, следующего за узлом *N*. (Старый указатель изображен пунктирной линией.) Обратите внимание, что изменение этого указателя не влияет непосредственно на узел *N*. Узел *N* по-прежнему существует, и данные, и указатель в нем остаются неизменными. Однако этот узел оказывается вне связанного списка. Например, при выполнении цикла *for* из предыдущего примера данные, хранящиеся в узле *N*, выводиться на экран не будут.

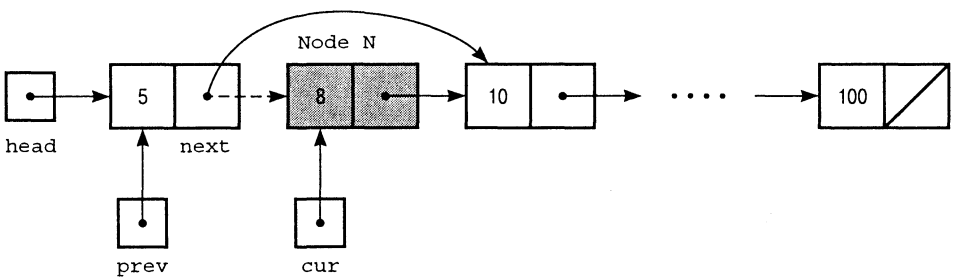


Рис. 4.10. Удаление узла из связанного списка

Чтобы изменить указатель предыдущего узла, сначала нужно учесть, что одного указателя на узел *N* для доступа к предыдущему списку недостаточно. Кроме того, мы вообще не имеем возможности продвигаться по списку в обратном направлении. Однако следует обратить внимание на указатель *prev*, приведенный на рис. 4.10. Он ссылается на узел, предшествующий узлу *N*, и позволяет изменить его указатель *next*, что в конечном итоге приводит к удалению узла *N* из связанного списка.

Для удаления узла, на который ссылается указатель *cur*, достаточно выполнить оператор

Удаление внутреннего узла

```
prev->nex = cur->next;
```

В связи с этим возникает вопрос: применим ли описанный выше метод к любому узлу *N*, независимо от его места в списке?

Нет, этот способ нельзя применять для удаления *первого* элемента списка, поскольку для него теряет смысл указатель *prev*, ссылающийся на предыдущий

узел! Итак, удаление первого узла связанного списка представляет собой отдельную задачу, как показано на рис. 4.11. В этом случае указатель *cur* ссылается на первый узел, а указатель *prev* равен константе *NULL*.

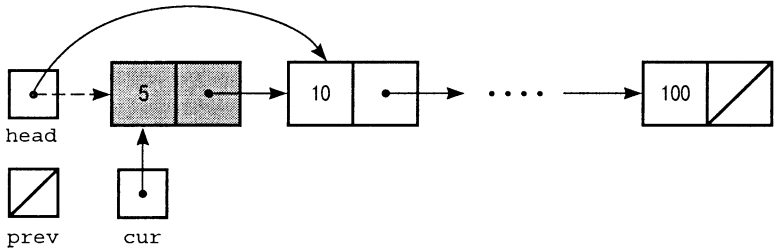


Рис. 4.11. Удаление первого узла

Удаляя первый элемент списка, необходимо изменить значение указателя *head*, отразив тот факт, что после удаления первым узлом списка становится другой элемент. Иными словами, узел, ранее бывший вторым, теперь становится первым.

Удаление первого узла представляет собой отдельную задачу

Изменить значение указателя *head* можно с помощью оператора присваивания

```
head = head->next;
```

Как и в случае удаления внутреннего узла, указатели минуют старый первый узел, хотя физически он продолжает существовать. Обратите также внимание, что если удаленный элемент был *единственным* элементом списка — первым и последним, — оператор, приведенный выше, присвоит указателю *head* константу *NULL*. Это будет означать, что список пуст, поэтому удаление единственного узла из списка выполняется правильно.

На самом деле существование узла *N* после его удаления из связанного списка приводит к неэффективному использованию памяти. Независимо от положения узла *N* в списке, указатель *cur* продолжает на него ссылаться. Если изменить значение указателя *cur* так, чтобы он перестал ссылаться на узел *N*, этот узел будет потерян навсегда — он по-прежнему будет занимать определенную часть памяти, даже если программа больше к нему не обращается. (Это может привести к утечке памяти.) Если таких узлов окажется достаточно много, объем занимаемой ими памяти может стать неприемлемым.

Освобождение памяти, занимаемой удаленными узлами, с помощью оператора delete

Следовательно, прежде чем изменить значение указателя *cur*, нужно выполнить следующие операторы.

```
cur->next = NULL;
delete cur;
cur = NULL;
```

Это позволит освободить память, занимаемую узлом *N*. Теперь система может вновь использовать освобожденную ячейку памяти и даже разместить в ней новую переменную с помощью оператора *new*. Допустим, это действительно произошло, и область памяти, которую ранее занимал узел *N*, теперь выделена для другого узла. Оператор *cur->next = NULL*, выполненный перед освобождением этой ячейки, гарантирует, что вновь созданный узел никак не будет связан со списком. Присвоение константы *NULL* указателям *cur->next* и *cur* представляет собой пример защиты, предотвращающей появление незаметных, на первый взгляд, ошибок, имеющих разрушительные последствия.

До сих пор мы удаляли узел *N*, на который ссылался указатель *cur*, когда указатель *prev* ссылался на предыдущий узел. Однако остался без ответа следующий вопрос: как установить указатели *cur* и *prev* на нужные узлы?

Чтобы ответить на него, рассмотрим контекст, в котором удаляется узел. Во-первых, узел можно удалять, задавая его позицию в списке. Именно так удаляются узлы из абстрактного списка. Во-вторых, можно удалять узел, содержащий определенные данные. Именно так удаляются узлы из абстрактного упорядоченного списка. В обоих случаях указатели *cur* и *prev* не передаются функции, удаляющей узел. Вместо этого функция сама вычисляет эти указатели, выполняя поиск узла *N* в связанном списке либо по его позиции, либо по содержанию, а затем удаляя его. Вычисление указателей *cur* и *prev* выполняется так же, как и при вставке, поэтому мы опишем его в следующем разделе.

Подведем итоги. Удаление узла из связанного списка распадается на три этапа.

Три этапа удаления узла из связанного списка

1. Найти узел, подлежащий удалению.
2. Отсоединить его от связанного списка, изменив значения указателей.
3. Освободить память, занятую удаленным узлом.

Позднее мы включим операцию удаления в реализацию абстрактного списка.

Вставка узла в указанную позицию связанного списка

На рис. 4.12 показана технология вставки нового узла в указанную позицию связанного списка. Новый узел, на который ссылается указатель *newPtr*, вставляется между двумя узлами, на которые ссылаются указатели *prev* и *cur*.

Как показано на диаграмме, вставку можно выполнить с помощью пары операторов присваивания

Вставка нового узла между двумя узлами

```
newPtr->next = cur;
prev->next = newPtr;
```

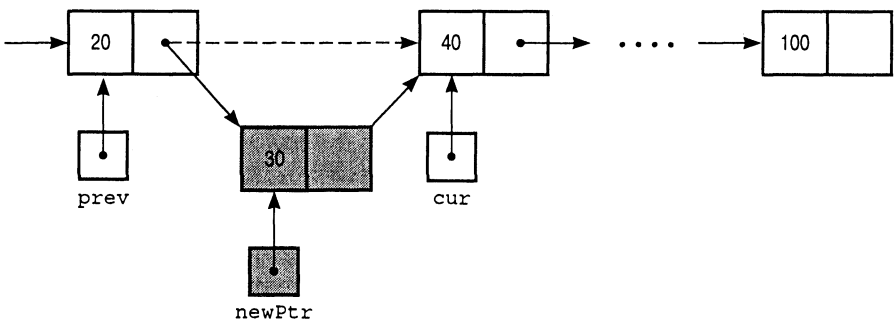


Рис. 4.12. Вставка нового узла в связанный список

В связи с этим возникают два вопроса, которые нами уже рассматривались при удалении узлов.

- Как установить указатели *newPtr*, *cur* и *prev* на нужные узлы?
- Можно ли применять этот метод для вставки элемента в произвольную позицию связанного списка?

Для ответа на первый вопрос, как и в случае удаления узлов, необходимо рассмотреть контекст, в котором используется операция вставки. Для вычисления значений указателей *cur* и *prev* выполняется обход списка. Когда обнаруживается искомый узел, этим указателям присваиваются соответствующие адреса. Затем создается новый узел, на который ссылается указатель *newPtr*. Для этого нужно выполнить оператор

```
newPtr = new Node;
```

После инициализации данных, хранящихся в этом узле, он вставляется в список, как описано выше.

Ответ на второй вопрос сводится к следующему: *операция вставки, как и удаления, должна учитывать особые ситуации*. Во-первых, рассмотрим вставку узла в начало связанного списка, как показано на рис. 4.13. На этот узел нужно установить указатель *head*, записав в него указатель на узел, который до этого был первым.

Вставка нового узла в начало связанного списка

Для этого нужно выполнить следующие операторы.

```
newPtr->next = head;
head = newPtr;
```

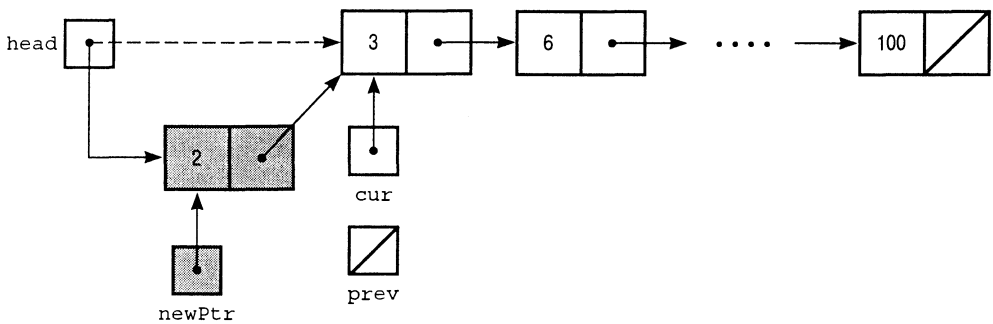


Рис. 4.13. Вставка нового узла в начало связанного списка

Если до вставки список был пустым, то указатель *head* был равен константе *NULL*, следовательно, указатель *next*, принадлежащий вставленному узлу, также будет равен константе *NULL*. Этот этап выполняется совершенно правильно, поскольку новый элемент одновременно является и первым, и последним.

На рис. 4.14 показана операция вставки нового узла в конец связанного списка. Лучше этот случай рассматривать отдельно, поскольку операторы

Если указатель cur равен константе NULL, вставка нового элемента в конец списка не является отдельной задачей

```
newPtr->next = cur;
prev->next = newPtr;
```

предназначены для вставки нового узла между двумя существующими, на которые ссылаются указатели *cur* и *prev*. На что должен ссылаться указатель *cur*, если вставка производится в конец списка? В этой ситуации имеет смысл присвоить указателю *cur* константу *NULL*, поскольку когда при обходе списка текущий указатель *cur* смещается правее последнего узла, его значение становится равным *NULL*. Если указатель *cur* равен константе *NULL*, указатель *prev* ссылается на последний узел списка, и приведенная выше пара операторов действительно вставляет новый узел в конец списка. Таким образом, вставка нового элемента в конец связанного списка не является отдельной задачей.

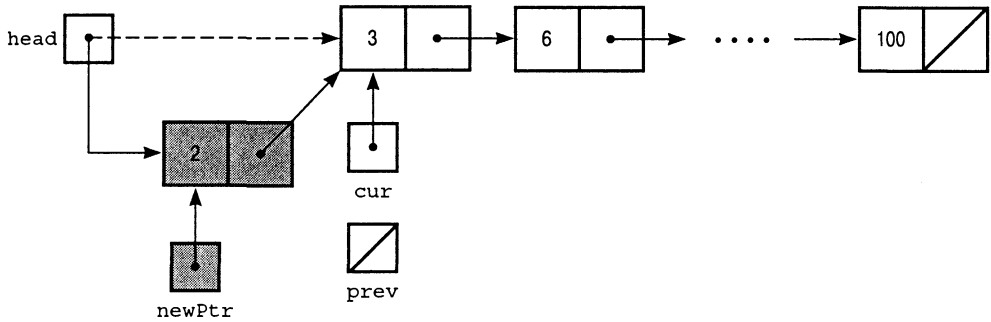


Рис. 4.14. Вставка нового узла в конец связанного списка

Подведем итоги. Вставка нового узла в связанный список распадается на три этапа.

Три этапа вставки нового узла в связанный список

1. Найти позицию, в которую нужно вставить узел.
2. Создать новый узел и записать в него данные.
3. Соединить новый узел со связанным списком, изменив значения соответствующих указателей.

Вычисление указателей cur и prev. Рассмотрим подробнее, как вычисляются указатели *cur* и *prev* при выполнении операции вставки нового узла. Как уже указывалось, эти вычисления зависят от контекста, в котором происходит вставка. Например, рассмотрим связанный список, содержащий целые числа, упорядоченные по возрастанию. Для простоты будем предполагать, что все числа разные, т.е. список не содержит дубликатов.

Чтобы определить позицию, куда следует вставить значение *newValue*, нужно выполнить обход списка с самого начала, пока не будет найдено подходящее место. Оно находится непосредственно перед узлом, содержащим первое число, превышающее значение *newValue*. Указатель *cur* нужно установить на узел, следующий за новым элементом. Таким образом, указатель *cur* должен ссылаться на узел, содержащий первое число, превышающее значение *newValue*. Кроме того, указатель *prev* нужно установить на узел, предшествующий новому элементу списка. Иными словами, указатель *prev* должен ссылаться на последнее число, не превышающее значение *newValue*. Таким образом, при обходе связанного списка нужно хранить текущий указатель *cur* и *добавочный* (trailing) указатель *prev*. При обнаружении узла, содержащего первое число, превышающее значения *newValue*, указатель *prev* будет ссылаться на предыдущий узел. Теперь новый узел можно вставлять между двумя узлами, на которые ссылаются указатели *prev* и *cur*, как описано выше.

В первом приближении псевдокод выглядит следующим образом.

Первый вариант псевдокода

```
// Определить место в связанном списке, куда нужно вставить
// новый элемент.

// Инициализировать указатели prev и cur перед началом обхода
// списка с головы

prev = NULL
cur = head
```

```
// Переместить указатели prev и cur вперед, поскольку
// значение newValue больше значения узла, на который
// ссылается текущий указатель
// Инвариант цикла: значение newValue больше чисел,
// хранящихся в любых узлах, предшествующих элементу,
// на который ссылается указатель prev

while (newValue > cur->item) // Порождает проблемы!
{
   prev = cur
   cur = cur->next
} // Конец цикла while
```

К сожалению, если новое значение больше любого значения, хранящегося в списке, т.е. если вставка осуществляется в конец связанного списка (или когда список пуст), цикл *while* порождает проблемы. В конечном счете цикл *while* сравнивает значение *newValue* со значением, хранящимся в последнем узле, при этом указателю *cur* будет присвоена константа *NULL*. После этого значение *newValue* будет снова сравниваться с указателем *cur->item*, которое становится некорректным, если указатель *cur* равен *NULL*.

Чтобы решить эту проблему, условие окончания цикла *while* нужно проверять иначе, так чтобы цикл завершался, когда указатель *cur* становится равным константе *NULL*. Таким образом, нужно заменить оператор *while* следующим оператором.

```
while (cur ≠ NULL и newValue > cur->item)
```

Новый вариант псевдокода выглядит так.

Правильное решение

```
// Определить место в связанном списке,
// куда нужно вставить новый элемент.

// Инициализировать указатели prev и cur перед началом обхода
// списка с головы

prev = NULL
cur = head

// Переместить указатели prev и cur вперед, поскольку
// значение newValue больше значения узла, на который
// ссылается текущий указатель
// Инвариант цикла: значение newValue больше чисел,
// хранящихся в любых узлах, предшествующих элементу,
// на который ссылается указатель prev

while (cur ≠ NULL и newValue > cur->item)
{
   prev = cur
   cur = cur->next
} // Конец цикла while
```

Обратите внимание на то, как оператор *while* применяется для вставки узла в конец списка. Если значение *newValue* больше всех чисел, хранящихся в списке, указатель *prev* ссылается на последний узел, а указатель *cur* становится равным константе *NULL* (рис. 4.15). Следовательно, новый элемент можно вставить в конец списка, используя стандартную пару операторов

Вставка в конец связанного списка не является отдельной задачей

```
newPtr->next = cur;
prev->next = newPtr;
```

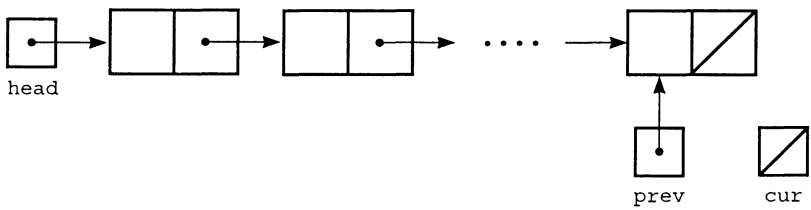


Рис. 4.15. Когда указатель prev ссылается на последний узел, а указатель cur равен константе NULL, вставка нового узла производится в конец связанного списка

Когда значение указателя prev равно константе NULL, вставка производится в начало списка

Рассмотрим теперь вариант, когда узел вставляется в начало связанного списка. Эта ситуация возникает, когда вставляемое значение меньше всех чисел, хранящихся в списке. В этом случае цикл `while` из предыдущего псевдокода никогда не будет выполнен, поскольку указатели `prev` и `cur` не изменят своих первоначальных значений (рис. 4.16). В частности, указатель `prev` останется равным константе *NULL*. Это единственная ситуация, в которой значение указателя `prev` равно константе *NULL* после выполнения цикла `while`. Таким образом, вставка нового элемента должна производиться в начало связанного списка.

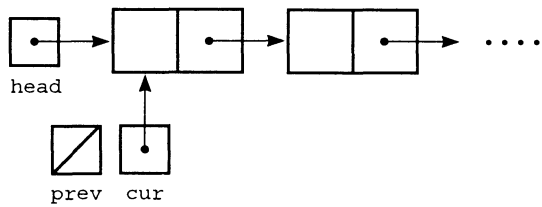


Рис. 4.16. Когда указатель prev равен константе NULL, а указатель cur ссылается на первый узел, вставка и удаление узлов производится в начале связанного списка

Вставка в пустой связанный список представляет собой вставку в начало списка

Это решение распространяется и на случай пустого списка. Вставка в пустой связанный список представляет собой вставку в начало списка. Когда список пуст, оператор `cur = head` сначала присваивает указателю `cur` константу *NULL*, и поэтому цикл `while` никогда не выполняется. Следовательно, указатель `prev` сохранит свое первоначальное значение, равное константе *NULL*. Это значит, что вставка нового узла будет производиться в начало списка.

Немного поразмыслив, легко понять, что приведенные выше рассуждения касаются и операции удаления. Если нужно удалить заданный узел из упорядоченного списка, нужно обойти список и найти узел, хранящий искомое значение. Именно это делает приведенный выше псевдокод: указатель `cur` будет ссылаться на искомый узел, а указатель `prev` — на предшествующий ему узел. Если удаляемым узлом окажется первый элемент списка, указатель `prev` будет равным константе *NULL*, как показано на рис. 4.16.

Определение места вставки или удаления на языке C++

```
// Определение места вставки или
// удаления узла
// упорядоченного связанного списка
// Инвариант цикла: значение newValue
// больше чисел, хранящихся в любых
// узлах, предшествующих элементу,
// на который ссылается указатель prev

for (prev = NULL, cur = head;
     (cur != NULL) && (newValue > cur->item);
     prev = cur, cur = cur->next);
```

Напомним, что оператор && (логическое "и") в языке C++ не вычисляет свой второй операнд, если первый операнд оказался равен 0 (т.е. ложным). Таким образом, указатель *cur* становится равным константе *NULL*, и цикл завершается, не сделав попытки вычислить указатель *cur->item*. Следовательно, очень важно, чтобы проверка условия *cur != NULL* выполнялась первой.

Вычисление указателей *cur* и *prev* упрощается, если элемент вставляется или удаляется по своей позиции в списке, а не по значению.

Реализация абстрактного списка, основанная на указателях

В этом разделе мы рассмотрим, как, применяя указатели, можно реализовать связанный список, не прибегая к помощи массивов. В отличие от реализации списка в виде массива, в реализации списка, основанной на указателях, не требуется сдвигать элементы при вставке или удалении узлов. Кроме того, эта реализация не навязывает фиксированный размер структуры данных, за исключением, конечно, физических ограничений, накладываемых операционной системой.

Как и в других частях книги, абстрактные типы данных реализуются в виде классов. Реализация списка в виде массива содержала объявления открытых функций-членов, соответствующих операциям над абстрактным списком. Эти объявления сохраняются неизменными и в реализации, основанной на указателях.

Нам нужно средство для представления элементов абстрактного списка и его длины. На рис. 4.17 изображен один из возможных вариантов представления этих данных с помощью указателей. Здесь указатель *head* ссылается на связанный список, в котором первый узел хранит первый элемент абстрактного списка и т.д. Целочисленная переменная *size* хранит текущее количество элементов списка. Обе переменные *head* и *size* являются закрытыми членами класса.

Указатели cur и prev не должны быть членами класса

Для манипуляции со связанными списками, как и ранее, используются указатели *cur* и *prev*. Эти указатели должны быть локальными переменными внутри соответствующих функций-членов. Они не должны быть членами класса.

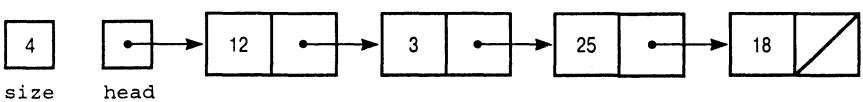


Рис. 4.17. Реализация абстрактного списка, основанная на указателях

Напомним, что при выполнении операций вставки, удаления и извлечения элемента абстрактного списка требуется задать его позицию *I*. Для того чтобы на основании числа *I* вычислить указатели *cur* и *prev*, определим функцию *find(I)*,

возвращающую указатель на *I*-й узел связанного списка. Если функция *find(I)* возвращает указатель *cur* на *I*-й узел, чему равен указатель *prev* на предыдущий, т.е. (*I-1*)-й узел? Можно ли получить указатель *prev*, выполнив вызов *find(I-1)*? Выполнять двойной вызов функции *fund* необязательно. Достаточно заметить, что указатель *cur* равен указателю *prev->next*, так что для его вычисления достаточно знать указатель *prev*. Единственное исключение составляет первый узел, но по значению *I* эту ситуацию легко распознать. Указатель на первый элемент вычислять не нужно, ведь он задается переменной *head*.

Функция find является закрытым членом класса

Функция *find* не является операцией над абстрактным списком. Поскольку она возвращает указатель, ни один клиент не должеч иметь к ней доступ. Клиенты должны использовать абстрактный список, ничего не зная о деталях его реализации, которая в свою очередь зависит от указателей. Лучше всего все переменные и функции, входящие в реализацию, сделать закрытыми членами класса. Следовательно, функцию *find* следует также сделать закрытым членом класса, доступ к которой могут иметь только реализации операций над абстрактным классом.

Ниже приведен заголовочный файл реализации абстрактного списка, основанной на указателях. Он вобрал в себя все особенности, которые мы обсудили выше. Обратите внимание, что в заголовочном файле узел связанного списка *объявляется*, но не *определяется*. Его определение скрыто в файле реализации. Кроме того, следует отдельно предусмотреть деструктор и **конструктор копирования** (copy constructor). Пред- и постусловия операций над абстрактным списком остаются прежними (см. главу 3). Здесь они пропущены для экономии места. Кроме того, классы исключительных ситуаций, определенных для списков в главе 3, предполагаются неизменными.

```
// ********************************************************
// Заголовочный файл ListP.h для реализации
// абстрактного списка, основанной на указателях
// ********************************************************
#include «ListException.h»
#include «ListIndexOutOfRangeException.h»
typedef тип_элемента_списка ListItemType;

class List
{
public:
// Конструкторы и деструкторы:
   List(); // Конструктор по умолчанию
   List(const List& aList); // Конструктор копирования
   ~List(); // Деструктор

// Операции над списком:
   bool isEmpty() const;
   int getLength() const;
   void insert(int index, ListItemType newItem)
        throw(ListIndexOutOfRangeException, ListException);
   void remove(int index)
        throw(ListIndexOutOfRangeException);
   void retrieve(int index, ListItemType& dataItem) const
        throw(ListIndexOutOfRangeException);
  // Конструктор копирования и деструктор необходимы
  // для реализации абстрактного списка на основе указателей
```

```
private:
   struct ListNode     // Узел списка
   {
     ListItemType item; // Данные, хранящиеся в узле
     ListNode *next;  // Указатель на следующий узел
   }; // Конец структуры

   int size;        // Количество элементов списка
   ListNode *head;    // Указатель на связанный список
   ListNode *find(int index) const;
   // Возвращает указатель на узел с номером index
}; // Конец описания класса
// Конец заголовочного файла
```

Файл реализации начинается так.

```
// ********************************************************
// Файл реализации ListP.cpp для абстрактного списка.
// Реализация на основе указателей.
// ********************************************************
#include «ListP.h» // Заголовочный файл
#include <cstddef> // Определение константы NULL
#include <cassert> // Определение макроса assert()
// Определения функций-членов:
...
```

Определения функций-членов класса содержатся в файле реализации. Проверим каждую из них.

Конструктор по умолчанию. Он просто инициализирует члены класса *size* и *head*.

```
List::List(): size(0), head(NULL)
{
} // Конец конструктора по умолчанию
```

Поскольку автоматический конструктор не инициализирует переменные *size* и *head* заданными значениями, следует предусмотреть свой собственный конструктор по умолчанию.

Конструктор копирования. Второй конструктор класса List является конструктором копирования.

```
List(const List& aList);
```

Когда применяется конструктор копирования

Конструктор копирования создает копию объекта. Он вызывается неявно, когда объект передается функции по значению, когда функция возвращает объект в качестве результата, а также при определении и инициализации вновь создаваемого объекта, как показано ниже.

```
List yourList = myList;
```

Здесь предполагается, что объект *myList* уже существует.

Автоматический конструктор создает поверхностную копию объекта

Копирование объекта называется **поверхностным** (shallow copying), если оно сводится лишь к дублированию его данных-членов. Если этого достаточно, конструктор копирования можно не предусматривать. В этом случае компилятор сгенерирует автоматический конструктор, который выполнит поверхностное копирование. Именно этот механизм копирования применялся в

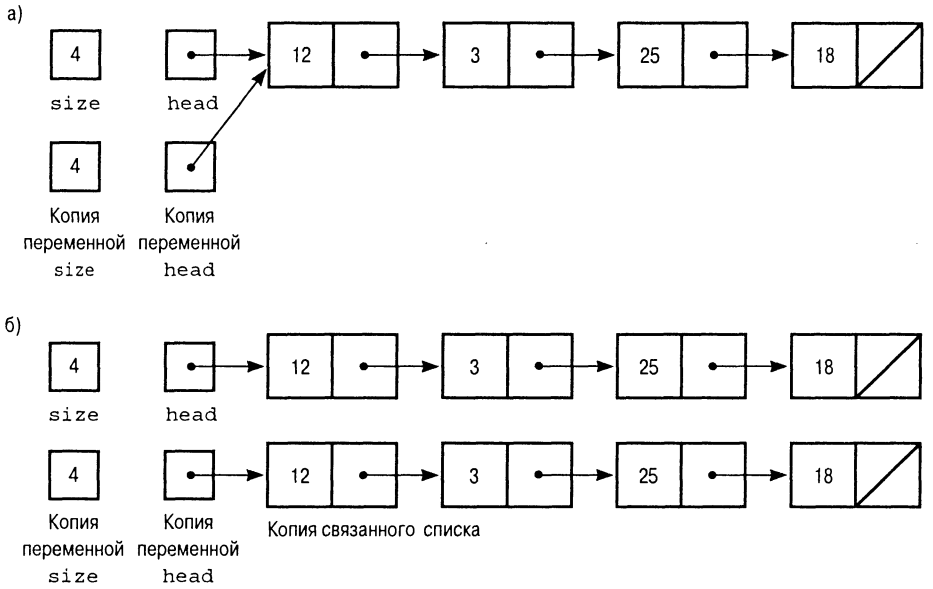


Рис. 4.18. Копии связанного списка, изображенного на рис. 4.17: а) поверхностная копия; б) глубокая копия

главе 3. Например, реализация абстрактного списка в виде массива использовала автоматический конструктор для копирования массива, содержащего элементы списка, и переменной, хранящей его размер.

В новой реализации списка, основанной на указателях, автоматический конструктор скопировал бы лишь переменные *size* и *head*. На рис. 4.18, *а* показан результат поверхностного копирования связанного списка, изображенного на рис. 4.17. Исходный указатель *head* и его копия ссылаются на один и тот же связанный список. Иными словами, сами узлы связанного списка не копируются. Если нужно создать копию всего списка, напишите свой собственный конструктор. В этом случае будет выполнено **глубокое копирование** (deep copying), как показано на рис. 4.18, *б*. Таким образом, конструктор копирования принимает следующий вид.

```
List::List(const List& aList): size(aList.size)
{
   if (aList.head == NULL)
      head = NULL; // Исходный список пуст
   else
   {
      // Копирование первого узла
      head = new ListNode;
      assert(head != NULL); // Проверка оператора new
      head->item = aList.head->item;

      // Копирование остальной части списка
      ListNode *newPtr = head; // new list pointer
      // Указатель newPtr ссылается на последний узел нового списка
      for (ListNode *origPtr = aList.head->next;
          origPtr != NULL;
          origPtr = origPtr->next)
      {
```

```
            newPtr->next = new ListNode;
            assert(newPtr->next != NULL);
            newPtr = newPtr->next;
            newPtr->item = origPtr->item;
        } // Конец цикла for

        newPtr->next = NULL;
    } // Конец цикла if
} // Конец конструктора копирования
```

Легко увидеть, что конструктор копирования выполняет довольно затратную операцию. Он выполняет обход списка и создает дубликат каждого элемента, помещая его в новый список.

Кроме того, конструктор копирования использует макрос *assert*, проверяя правильно ли выделена память для нового узла. В качестве альтернативы можно генерировать исключительную ситуацию, однако это требует особой осторожности. Нужно быть абсолютно уверенным, что вся память, выделенная конструктором, правильно освобождается до генерации исключительной ситуации.

Деструктор. Каждый класс имеет только один деструктор. Единственное его предназначение — уничтожение экземпляра класса, т.е. объекта, по истечении времени его жизни. Обычно деструктор вызывается неявно, когда происходит выход из блока или функции, в которой был создан (определен) объект.

Деструкторы необходимы, если классы выделяют динамическую память

Классы, использующие только статическую память, могут использовать автоматический деструктор (compiler-generated destructor), как показано в главе 3. Однако если класс выделяет динамическую память, как это было в реализации списка, основанной на указателях, нужно разработать собственный деструктор, освобождающий эту память.

Имя деструктора образуется путем приписывания тильды (~) к имени класса. Деструктор не может иметь аргументов и не имеет возвращаемого значения, даже типа *void*, поэтому в нем нельзя применять оператор *return*.

Деструктор, используемый в реализации класса *List*, должен освобождать память, занятую связанным списком. Эту операцию можно выполнить, просто применив операцию *remove* для удаления каждого элемента списка.

```
List::~List()
{
   while (!isEmpty())
   remove(1);
} // Конец деструктора
```

Учтите, что даже после удаления первого элемента все остальные элементы списка перенумеровываются. Таким образом, чтобы удалить все элементы, можно постоянно удалять первый элемент.

Деструкторы объектов, занимающих динамическую память, для ее освобождения должны использовать оператор *delete*. Как мы вскоре убедимся, функция *remove* использует оператор *delete*, освобождая таким образом память при вызове деструктора.

Операции над списком. Операции *isEmpty* и *getLength* имеют очевидные реализации.

```
bool List::isEmpty() const
{
   return bool(size == 0);
} // Конец функции isEmpty
```

```
int List::getLength() const
{
   return size;
} // Конец функции getLength
```

Поскольку в связанном списке не предусмотрен прямой доступ к элементу по заданной позиции, для выполнения операций извлечения, вставки и удаления необходимо осуществлять обход всего списка от начала до искомого элемента. Для этого предназначена функция *find*. Ее реализация имеет следующий вид.

```
List::ListNode *List::find(int index) const
// ------------------------------------------------
// Обнаруживает указанный узел в связанном списке.
// Предусловие: переменная index задает номер искомого узла.
// Постусловие: возвращает указатель на искомый узел.
// Если index < 1 или index больше количества узлов списка,
// возвращается константа NULL.
// ------------------------------------------------
{
  if ( (index < 1) || (index > getLength()) )
     return NULL;
  else // Отсчет от начала списка
  {
     ListNode *cur = head;
     for (int skip = 1; skip < index; ++skip)
     cur = cur->next;
     return cur;
  } // Конец оператора if
} // Конец функции find
```

Функция *find* возвращает константу *NULL* в качестве признака неверного значения переменной *index*.

Операция извлечения элемента вызывает функцию *find* для обнаружения искомого узла.

```
void List::retrieve(int index,
          ListItemType& dataItem) const
{
  if ((index < 1) || (index > getLength()))
     throw ListIndexOutOfRangeException(
  else
  {
     // Вычислить указатель на узел, а затем
     // извлечь из него данные
     ListNode *cur = find(index);
     dataItem = cur->item;
  } // Конец оператора if
} // Конец функции retrieve
```

Операции вставки и удаления элементов связанного списка используют уже описанную ранее технологию. Для вставки элемента в середину списка сначала нужно получить указатель на предыдущий элемент. Вставка элемента на первую позицию представляет собой отдельную задачу.

```
void List::insert(int index, ListItemType newItem)
{
  int newLength = getLength() + 1;
```

```
    if ((index < 1) || (index > newLength))
       throw ListIndexOutOfRangeException(
       "ListOutOfRangeException: неверный индекс");
    else
    {
       // Создаем новый узел и помещаем в него объект newItem
       ListNode *newPtr = new ListNode;
       if (newPtr == NULL)
          throw ListException(
          "ListException: выделить память невозможно");
       else
       {
          size = newLength;
          newPtr->item = newItem;

          // Присоединяем к списку новый элемент
          if (index == 1)
          {
             // Вставляем новый узел в начало списка
             newPtr->next = head;
             head = newPtr;
          }
          else
          {
             ListNode *prev = find(index-1);
             // Вставляем новый узел после узла,
             // на который ссылается указатель prev
             newPtr->next = prev->next;
             prev->next = newPtr;
          } // Конец оператора if
       } // Конец оператора if
    } // Конец оператора if
} // Конец функции insert
```

Операция удаления аналогична операции вставки. Для того чтобы удалить элемент из середины списка, сначала нужно получить указатель на его предшественника. Удаление первого элемента представляет собой отдельную задачу.

```
void List::remove(int index)
{
  ListNode *cur;
  if ((index < 1) || (index > getLength()))
     throw ListIndexOutOfRangeException(
     "ListOutOfRangeException: неверный индекс");
  else
    {
       --size;
       if (index == 1)
       {
           // Удаляем первый элемент списка
          cur = head; // Сохраняем указатель на узел
          head = head->next;
       }
       else
       {
          ListNode *prev = find(index-1);
          // Удаляем узел, находящийся после узла,
```

```
            // на который ссылается указатель prev
            cur = prev->next; // Сохраняем указатель на узел
            prev->next = cur->next;
        } // Конец оператора if

        // Освобождаем память, занятую узлом
        cur->next = NULL;
        delete cur;
        cur = NULL;
    } // Конец оператора if
} // Конец функции remove
```

Реализации списка в виде массива и на основе указателей

Обычно каждая реализация, рассматриваемая программистом, имеет свои преимущества и недостатки. При выборе подходящей реализации нужно тщательно взвесить все плюсы и минусы. Как мы убедимся в дальнейшем, абстрактный список допускает множество разнообразных реализаций. В этом разделе сравним две реализации абстрактного списка.

Массив легко использовать, но он имеет фиксированный размер

Реализация списка в виде массива, описанная в главе 3, — вполне разумный выбор. Массив похож на список, кроме того, его легко использовать. Однако, как уже указывалось, массив имеет фиксированный размер. Вполне вероятна ситуация, когда количество элементов списка превысит длину массива. На практике, выбирая подходящую реализацию абстрактного типа данных, необходимо ответить на вопрос, создает ли фиксированный размер массива особые проблемы для конкретного приложения?

Можете ли вы предсказать максимальное количество элементов списка?

Ответ на этот вопрос зависит от двух факторов. Во-первых, совершенно очевидно, он зависит от того, можете ли вы предсказать максимальное количество элементов списка. Если нет, вполне возможно, что отдельная операция и, следовательно, вся программа в целом будут работать неверно, если для хранения элементов списка понадобится больше ячеек памяти, чем предусмотрено в массиве.

Эффективно ли массив использует память?

Иногда в конкретном приложении можно предсказать максимально возможное количество элементов абстрактного списка. Тогда нужно ответить на второй вопрос: эффективно ли используется память, если массив достаточно велик и способен хранить все элементы списка? Представьте себе, что максимально возможное количество элементов велико, но ситуация, когда это происходит, возникает редко. Допустим, что список может состоять из 10000 элементов, но на практике его длина редко превышает 50. Если при компиляции зарезервировать массив, состоящий из 10000 ячеек, то по крайней мере 9950 ячеек большую часть времени будут пустовать. В обеих ситуациях, описанных выше, реализация списка в виде массива нежелательна.

А если использовать динамический массив? Поскольку память для него выделяется динамически, с помощью оператора *new* можно зарезервировать ровно столько ячеек, сколько нужно для хранения элементов списка (естественно, не выходя за ограничения, налагаемые конкретным компьютером). Итак, знать заранее максимально возможное количество элементов списка не нужно.

Увеличение размера динамического массива приводит к неэффективному использованию памяти и затратам времени

Однако если каждый раз при заполнении массива удваивать его размер (кстати, вполне разумный подход), может остаться много незаполненных ячеек. В приведенном выше примере нам необходимо было хранить 50 элементов

списка. Если на самом деле длина списка достигает 10000, при дублировании массива его размер в конце концов достигнет 12800, т.е. на 2800 ячеек больше, чем нужно. Кроме того, при копировании элементов массива и удалении их из памяти нужны вспомогательные переменные.

Реализация списка в виде массива — хороший выбор для небольших списков

Допустим теперь, что список никогда не будет содержать больше, чем 25 элементов. Если реализовать его в виде массива, перерасход памяти будет пренебрежимо мал. В этой ситуации реализация списка в виде массива предпочтительнее всех остальных.

Связанные списки не имеют фиксированного размера

Реализация списка с помощью указателей позволяет преодолеть ограничения, связанные с фиксированным размером массива. Элементы списка в этой реализации создаются динамически, с помощью оператора *new*, причем заранее их количество задавать не требуется. Поскольку в каждый момент времени создается один узел, список будет иметь точно такую длину, какая нужна. Таким образом, дополнительные затраты памяти не возникают.

Преемник каждого элемента массива подразумевается; в связанном списке преемник элемента задается явно

Между реализациями списка в виде массива и на основе указателей есть и другие различия. Они касаются как временных ограничений, так и затрат памяти. Каждый раз, когда в массив или связанный список записывается набор данных, элементы упорядочиваются. Иными словами, среди них есть первый, второй и т.д. В этом случае каждый элемент имеет предшественника и преемника. В массиве `anArray` местоположение преемника элемента `anArray[i]` подразумевается *неявно* — `anArray[i+1]`. Однако в связанном списке позиция следующего элемента всегда задается *явно*.

Реализация списка в виде массива экономит память

Различие между явным и неявным заданием преемника в связанном списке и массиве является одним из наиболее важных факторов. Преимущество реализации списка в виде массива состоит в том, что не нужно явно хранить информацию о следующем элементе, следовательно, памяти затрачивается меньше, чем при реализации списка на основе указателей.

Доступ к элементам массива осуществляется непосредственно и за постоянное время

Другим важным преимуществом реализации списка в виде массива является тот факт, что в ней осуществляется **прямой доступ** (direct access) к указанному элементу. Например, если для реализации абстрактного списка используется массив `items`, то заранее известно, что элемент, находящийся в *i*-й позиции списка, находится в ячейке `items[i-1]`. Доступ к элементам `item[0]` или `item[49]` занимает одинаковое время. Это означает, что **время доступа** (access time) к элементам массива является постоянным.

Для доступа к i-му элементу нужно обойти весь список

Однако если абстрактный список реализуется в виде связанного списка, у нас нет прямого доступа к узлу, содержащему *i*-й элемент. Чтобы найти нужный элемент, нужно обойти весь список с самого начала. Другими словами, нужно найти первый узел, извлечь из него указатель на второй узел, найти второй узел, извлечь из него указатель на третий узел и т.д., пока не обнаружится *i*-й узел.

Время доступа к i-му узлу зависит от числа i

Очевидно, что время, необходимое для доступа к первому элементу, намного меньше времени, затрачиваемого на доступ к 50-му элементу. Таким образом, время доступа к *i*-му узлу зависит от числа *i*.

От выбранной реализации зависит эффективность операций над абстрактным списком. Операция *retrieve* в реализации списка в виде массива выполняется практически мгновенно, независимо от узла списка. Однако в связанном списке для доступа к *i*-му узлу операция *retrieve* должна выполнить *i* шагов.

Выше уже говорилось, что при вставке элемента в массив или удалении из него необходимо сдвигать остальные элементы. Например, при удалении первого элемента из списка, состоящего из 20 элементов, придется сдвинуть 19 элементов. В общем, при удалении *i*-го элемента из массива, длина которого равна *size*, нужно выполнить *size-i* сдвигов. Таким образом, при удалении первого элемента потребуется выполнить *size-1* сдвиг, а при удалении последнего элемента сдвиг выполнять не нужно. Функция *insert* работает аналогично.

Для вставки и удаления элементов связанного списка сдвигать данные не нужно

В противоположность этому, при вставке и удалении элементов связанного списка сдвигать данные не нужно. Объем работы, выполняемой функциями *insert* и *remove*, не зависит от длины связанного списка.

Для вставки и удаления элементов нужно выполнить обход связанного списка

Однако при вставке или удалении элемента придется выполнить обход связанного списка. Время, затрачиваемое на эту операцию, зависит от номера элемента. Напомним, что обход списка выполняется закрытой функцией-членом *find*. Проанализировав определение функции *find*, легко обнаружить, что для вычисления значения *find(i)* необходимо выполнить *i* операций присваивания. Таким образом, количество операций, необходимых для вычисления значения *find(i)*, растет с увеличением числа *i*.

Сравнение разных решений между собой встретится нам еще не раз. В главе 3 будет описан более формальный способ оценки эффективности алгоритмов, а пока наши обсуждения будут носить неформальный характер.

Запись связанных списков в файл и считывание их из файла

Связанные списки можно записывать во внешний файл и считывать их оттуда, сохраняя данные между разными сеансами работы программы. Для демонстрации этих операций снова рассмотрим связанный список, состоящий из целых чисел. Алгоритм считывания связанного списка показывает также, как его можно создать с нуля.

Напомним определение структуры.

```
struct Node
{
   int item;
   Node *next;
}; // Конец структуры

Node *head;
```

Не записывайте указатели в файл

Если в файл нужно записать связанный список целых чисел, а в дальнейшем извлечь его оттуда, возникает вопрос: что именно следует сохранять в файле? На первый взгляд, нужно записать в файл все узлы целиком, т.е. числа и указатели, хранящиеся в каждом узле. Однако записывать указатели в файл бесполезно, поскольку по завершении программы они теряют смысл. Эти указатели хранят адреса ячеек памяти, в которых были записаны узлы списка перед тем, как они были выгружены в файл, и после завершения работы программы эти адреса не нужны. Когда ваша программа будет запущена вновь, записанные в файле адре-

са могут оказаться занятыми совершенно другой структурой данных, и даже совсем другой программой. Таким образом, записывать в файл весь узел целиком — не слишком удачная идея.

> В файл следует записывать только данные, хранящиеся в узлах

Иногда требуется создать новый список, содержащий те же данные, что и прежний. При этом совершенно неважно, будет ли этот список занимать те же ячейки памяти, что и его предшественник. Таким образом, в файл следует записывать только данные, хранящиеся в узлах.

Сохранение связанного списка в файле проиллюстрировано на рис. 4.19. (Для экономии места на рисунке не показаны символы конца файла.) Запись данных (в данном случае целых чисел) в текстовый файл выполняется следующими операторами языка C++.

```
// Записываем данные, хранящиеся в связанном списке,
// в текстовый файл.

// Имя создаваемого файла задается строкой fileName
ofstream outFile(fileName);

// Обходим список от начала до конца,
// записывая каждый узел в файл
for (Node *cur = head; cur != NULL; cur = cur->next)
    outFile << cur->item << endl;

outFile.close();
// Предположение: текстовый файл содержит данные, хранящиеся
// в связанном списке и перечисленные в исходном порядке.
```

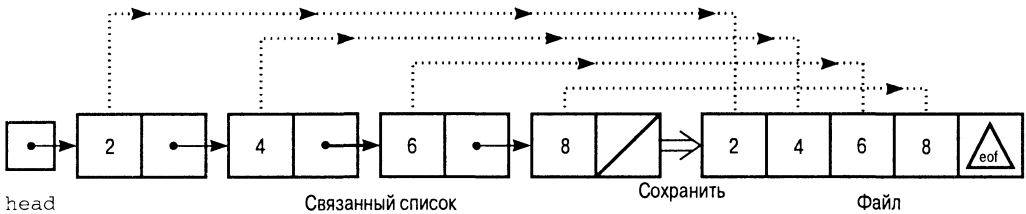


Рис. 4.19. Сохранение связанного списка в файле

После записи в файл данные хранятся в нем и после завершения работы программы, создавшей исходный список. Таким образом, в любое время можно вновь создать связанный список, содержащий те же самые данные. Обратите внимание, что в приведенном выше фрагменте программы целые числа записывались в файл в том порядке, в котором они встречаются в списке. Таким образом, для воссоздания связанного списка нужно вновь прочитать содержимое файла и поместить прочитанное целое число в конец списка, как показано на рис. 4.20.

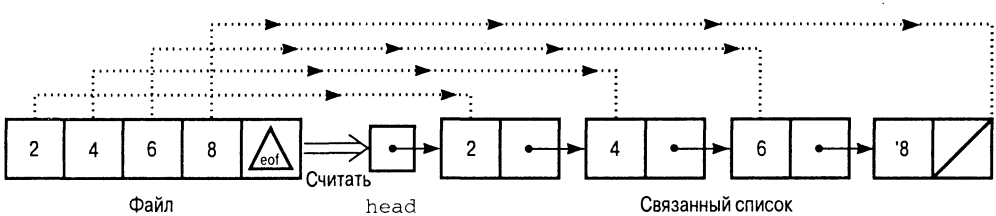


Рис. 4.20. Восстановление связанного списка, хранящегося в файле

Псевдокод полученного решения приведен ниже.

```
while (не конец файла)
{
   Считать следующее целое число
   Добавить это число в конец связанного списка
} // Конец оператора while
```

1. Выделить память для нового узла.
2. Установить указатель в последнем узле на новый узел.
3. Записать целое число в новый узел.
4. Установить указатель в новом узле равным константе *NULL*.

Четыре шага, которые нужно выполнить, чтобы добавить новый элемент в конец списка

Каждый раз, когда считывается очередное целое число, нужно найти последний элемент связанного списка. Это можно сделать путем обхода всего списка с самого начала, однако намного эффективнее использовать **указатель на хвост списка** (tail pointer) *tail*, в котором должен храниться адрес его последнего элемента. Как и указатель на голову списка, указатель *tail* не принадлежит самому списку. Связанный список с указателями на голову и хвост показан на рис. 4.21.

Для добавления узлов в конец списка используйте указатель на его хвост

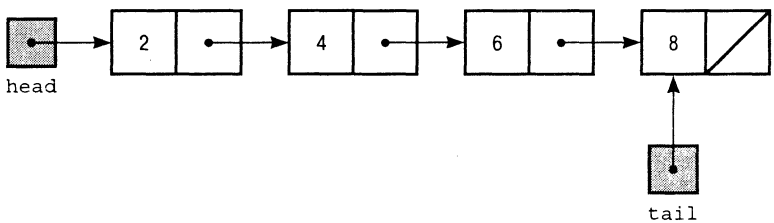


Рис. 4.21. Связанный список с указателями на голову и хвост

Используя указатель *tail*, шаги 1 и 2 можно выполнить с помощью одного оператора.

```
tail->next = new Node;
```

Этот оператор устанавливает указатель *next* в последнем узле списка на новый узел. Таким образом, существует простой способ вставки нового элемента в связанный список.

Однако вначале, когда в пустой список вставляется новый элемент, указатель *tail*, как и указатель *head*, равен константе *NULL*. Если рассматривать вставку нового элемента как отдельную задачу, мы получим решение, приведенное ниже. Здесь для простоты предполагается, что оператор *new* работает безошибочно, а уточнить детали читатели могут в качестве самостоятельного упражнения.

Вставка первого элемента представляет собой отдельную задачу

```
// Программа создает связанный список
// на основе данных, записанных в файл.
// Указатели head и tail вначале равны
// константе NULL.
// Имя внешнего текстового файла задается строкой fileName
```

Создание связанного списка на основе данных, записанных в файл

```
ifstream inFile(fileName);
int nextItem;

if (inFile >> nextItem) // Пуст ли файл?
{
   // Файл не пуст:
   head = new Node;
   // Добавляем в список первый элемент
   head->item = nextItem;
   head->next = NULL;
   tail = head;

   // Добавляем в список остальные элементы
   while (inFile >> nextItem)
   {
      tail->next = new Node;
      tail = tail->next;
      tail->item = nextItem;
      tail->next = NULL;
    } // Конец оператора while
} // конец оператора if

inFile.close();

// Диагностическое утверждение: указатель head указывает на
// первый элемент созданного связанного списка, а указатель
// tail - на его хвост. Если файл пуст, указатели head and tail
// равны константе NULL (список пуст).
```

Обратите внимание, что указатели *head* и *tail* должны быть локальными переменными и существовать или уничтожаться одновременно.

Допустим, что в файле *inFile* хранятся целые числа, записанные в неправильном порядке. Например, нам нужен возрастающий порядок, а они записаны как попало. Решить эту проблему помогает функция *linkedListInsert*, псевдокод которой приведен ниже.

Создание упорядоченного списка из произвольно записанного набора данных

```
head = NULL
while (inFile >> nextItem)
linkedListInsert(head, nextItem)
```

Этот алгоритм известен под названием **сортировка вставками** (insertion sort). Вместе с другими алгоритмами сортировки мы рассмотрим его в главе 9.

Операции *save* и *restore*, рассмотренные выше, можно включить в список операций над абстрактным списком.

Передача связанного списка в качестве аргумента функции

Функции, имеющие доступ к указателю на связанный список, имеют доступ ко всему списку

Каким образом функция может получить доступ к связанному списку? Для этого достаточно передать ей указатель на голову списка. Имея это указатель, функция может получить доступ к любому элементу списка. В реализации абстрактного списка на основе указателей, рассмотренной выше, указатель *head* был закрытым членом класса *List*. Функции-члены этого класса могут непосредственно использовать указатель *head* для манипуляций со списком.

В каких случаях указатель *head* может стать аргументом функции? Разумеется, функции-члены класса *Link* в этом не нуждаются, а вот для внешних

функций список недоступен. Однако, хотя, на первый взгляд, указатель *head* никогда не передается функциям-членам класса в качестве аргумента, это не совсем так. Например, рекурсивные функции должны получать указатель на голову списка как аргумент. Примеры таких функций мы рассмотрим в следующем разделе. Эти функции не могут быть открытыми членами класса. Если бы это было так, клиенты класса могли бы иметь прямой доступ к списку, разрушая защитные стены, возведенные вокруг абстрактного типа данных.

Как именно следует передавать указатель на голову списка: по значению или по ссылке? Если список должен быть изменен, указатель нужно передавать по ссылке, однако причины этого требования пока не совсем ясны. На первый взгляд, указатель *headPtr* нужно передавать по ссылке, потому что функция, получившая его, будет изменять содержимое узлов списка, на который ссылается указатель *headPtr*. Вывод верен, а предпосылка — нет.

> Указатель на голову списка нужно передавать по ссылке

Посмотрим, что случится, если указатель передать по значению. На рис. 4.22 показано, что хотя функция и копирует фактический аргумент *head* в формальный аргумент *headPtr*, сам список *не* копируется. Следовательно, если функция изменит содержимое какого-либо узла, это будет касаться только исходного узла, а не его локальной копии. Иными словами, *любые изменения, произведенные функцией со списком, не будут локализованы в ней самой.*

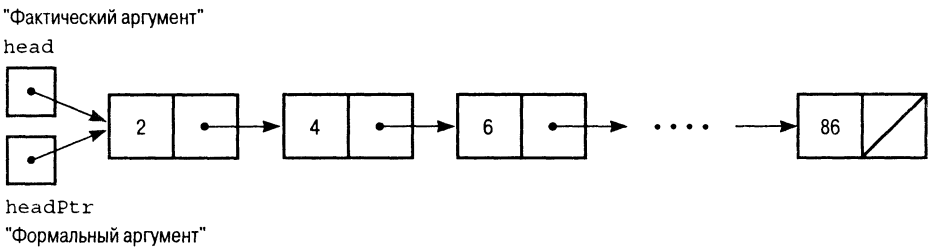


Рис. 4.22. Указатель на голову списка, передаваемый по значению

Следовательно, передача указателя на голову списка по значению позволяет функции модифицировать список, изменяя, вставляя и удаляя его узлы. Итак, на первый взгляд, указатель *head* нужно передавать по значению. Это было бы верно, если бы функция не могла изменить сам указатель *head*. Например, если функция вставляет в список первый элемент, она изменяет указатель *headPtr*, и это изменение должно отразиться на значении указателя *head*, который связан с указателем *headPtr*. Это единственная причина, по которой указатель *headPtr* нужно передавать по ссылке!

> Связанный список, передаваемый функции как аргумент, не копируется, даже если указатель head передается по значению

Рекурсивная обработка связанных списков

Иногда возможно, и даже желательно, обрабатывать связанный список рекурсивно. В этом разделе мы рассмотрим рекурсивный обход списка, а также рекурсивные операции вставки узлов. Рекурсивные функции, рассмотренные в этом разделе, должны быть закрытыми членами класса, поскольку им в качестве аргумента нужно передавать указатель на голову списка.

Обход. Для начала рассмотрим связанный список символов, образующих строку.

Допустим, в нашей программе определена следующая структура.

Связанный список символов, образующих строку

```
struct Node
{
   char item;
   Node *next;
}; // Конец структуры
Node *stringPtr;
```

Указатель *stringPtr* ссылается на голову связанного списка, содержащего строку.

Допустим, нам нужно вывести эту строку на экран. Иными словами, нам нужно вывести на экран символы, образующие эту строку, в том же самом порядке, в каком они встречаются в списке. Рекурсивная стратегия проста.

```
Вывести первый символ строки
Вывести строку без первого символа
```

Эту стратегию реализует следующая функция на языке C++.

```
void writeString(Node *stringPtr)
// ----------------------------------------------------------
// Выводит строку на экран.
// Предусловие: строка представлена в виде связанного списка,
// на который ссылается указатель stringPtr.
// Постусловие: строка выведена на экран. Связанный список и
// указатель stringPtr не изменились.
// ----------------------------------------------------------
{
  if (stringPtr != NULL)
  {
     // Выводим первый символ
     cout << stringPtr->item;
     // Выводим строку без первого символа
     writeString(stringPtr->next);
  } // Конец оператора if
} // Конец функции writeString
```

Сравните рекурсивную функцию writeString с ее рекурсивным аналогом

Эта функция довольно проста. Ей лишь нужен прямой доступ к первому символу строки. Связанный список предоставляет ей такой доступ, поскольку ей передается указатель *stringPtr*, ссылающийся именно на этот узел. Более того, этой функции легко передать оставшуюся часть строки, без первого символа. Поскольку указатель *stringPtr* ссылается на начало строки, то указатель *stringPtr->next* ссылается на остальную часть строки без первого символа. Сравните рекурсивную функцию *writeString* с ее рекурсивным аналогом, рассмотренным ранее.

Предположим теперь, что строку нужно вывести в обратном порядке. В главе 2 нами уже были разработаны две рекурсивные стратегии решения этой задачи. Напомним, что стратегия функции *writeBackward* заключалась в следующем.

```
Вывести последний элемент строки
Вывести строку без последнего элемента
```

Стратегия функции *writeBackward2* была иной.

```
Вывести в обратном порядке строку без первого элемента
Вывести первый элемент строки
```

Мы уже доказали, что обе эти стратегии одинаково хорошо работают, когда строка хранится в массиве. Однако, когда строка представляет собой связанный список, первую стратегию чрезвычайно трудно реализовать. Как получить доступ к последнему элементу, если указатель *stringPtr* ссылается на узел, содержащий первый символ строки? Даже если существует способ, позволяющий быстро найти последний узел, — например, с помощью указателя на хвост списка, — очень сложно каждый раз проходить список с начала при каждом рекурсивном вызове. Иными словами, при генерации рекурсивных вызовов очень трудно дойти до концов постоянно сокращающихся строк. (Позднее мы увидим, как эту проблему можно решить с помощью дважды связанных списков.)

Итак, основной недостаток связанных списков заключается в том, что они, в отличие от массивов, не предоставляют прямого доступа к своим элементам. К счастью, функция *writeBackward2* реализована так, что ей нужен прямой доступ лишь к первому символу строки. Это относится и к функции *writeString*. Обеим этим функциям нужно передать в качестве параметра указатель *stringPtr*, ссылающийся на первый символ строки, тогда указатель *stringPtr->next* будет ссылаться на строку без первого символа.

> Если строка хранится в связанном списке, функцию writeBackward2 намного легче реализовать рекурсивно, чем функцию writeBackward

Описанная выше стратегия реализуется в виде следующего кода.

```
void writeBackward2(Node *stringPtr)
// ------------------------------------------------------
// Записывает строку в обратном порядке.
// Предусловие: строка представлена в виде связанного списка,
// на который ссылается указатель stringPtr.
// Постусловие: на экране выводится строка, записанная
// в обратном порядке. Связанный список и указатель stringPtr
// не изменяются.
// ------------------------------------------------------
{
  if (stringPtr != NULL)
  {
     // Записываем в обратном порядке строку
     // без первого символа
     writeBackward2(stringPtr->next);
     // Записываем первый символ
     cout << stringPtr->item;
  } // Конец оператора if
} // Конец функции writeBackward2
```

В упражнении 8, приведенном среди вопросов для самопроверки, читателям предлагается выполнить трассировку этой функции. Эта трассировка похожа на трассировку с помощью блок-схем, изображенную на рис. 2.9. В упражнении 5 требуется написать альтернативную версию этой функции. Какой из этих вариантов более эффективен?

Вставка. Рассмотрим теперь операцию вставки узла в упорядоченный список с новой, рекурсивной, точки зрения. Позднее нам понадобится рекурсивный алгоритм для вставки нового элемента в список. Интересно, что рекурсивная вставка не требует хранить указатель на текущий узел и выполнять вставку первого элемента отдельно.

С рекурсивной точки зрения связанный список считается упорядоченным, если его первый элемент меньше второго, а список, начинающийся со второго элемента, упорядочен. Можно дать более формальное определение.

Связанный список, на который ссылается указатель *head*, является упорядоченным, если указатель *head* равен константе *NULL* (пустой связанный список является упорядоченным) или указатель *head->next* равен константе *NULL* (связанный список, состоящий из одного элемента, является упорядоченным), или *head->item < head->next->item*, причем указатель *head->next* ссылается на упорядоченный связанный список.

Рекурсивное определение упорядоченного связанного списка

На основе этого определения можно разработать рекурсивный алгоритм вставки. Функция, приведенная ниже, вставляет в список новый узел либо когда список пуст, либо когда значение, содержащееся в новом узле, меньше всех остальных значений, хранящихся в списке. В обоих случаях новый элемент нужно вставлять в начало списка.

```
void linkedListInsert(Node *& headPtr,
           ItemType newItem)
{
  if ((headPtr == NULL) || (newItem < headPtr->item))
  {
     // Базовый вариант: вставка элемента newItem
     // в начало связанного списка, на который ссылается
     // указатель headPtr
     Node *newPtr = new Node;
     if (newPtr != NULL)
        throw ListException(
        "ListException: невозможно выделить память");
     else
     {
        newPtr->item = newItem;
        newPtr->next = headPtr;
        headPtr = newPtr;
     } // Конец оператора if
  }
  else
    linkedListInsert(headPtr->next, newItem);
}  // Конец функции linkedListInsert
```

Несмотря на то что функция *linkedListInsert* не предусматривает указатель на текущий узел списка, вставка нового элемента осуществляется очень легко, если выполняются условия базовой задачи. Принципиальную трудность создает лишь оператор

Вставка представляет собой базовую задачу

```
headPtr = newPtr;
```

Этого оператора присваивания вполне достаточно, чтобы установить указатель *next* соответствующего элемента списка на новый узел. Обратите внимание, что указатель *headPtr* всегда ссылается на начало упорядоченного связанного списка. Для того чтобы вставить в начало этого списка новый узел, на который ссылается указатель *newPtr*, его адрес нужно присвоить указателю *headPtr*. Изменится ли при этом соответствующий указатель, являющийся фактическим аргументом функции? Да, если указатель *headPtr передается по ссылке*.

Указатель headPtr нужно передавать по ссылке

Чтобы понять это утверждение, сначала рассмотрим ситуацию, когда новый элемент вставляется в начало исходного списка, на который ссылается внешний указатель *head*. В этом случае рекурсивный вызов не выполняется. Следовательно, когда будут выполнены условия базовой задачи (*newItem == headPtr->item*),

фактический аргумент, соответствующий указателю *headPtr*, станет равным указателю *head*, как показано на рис. 4.23, *а*. Следовательно, если указатель *headPtr* передается по ссылке, оператор присваивания *headPtr = newPtr* устанавливает указатель *head* на новый узел (рис. 4.23, *б*).

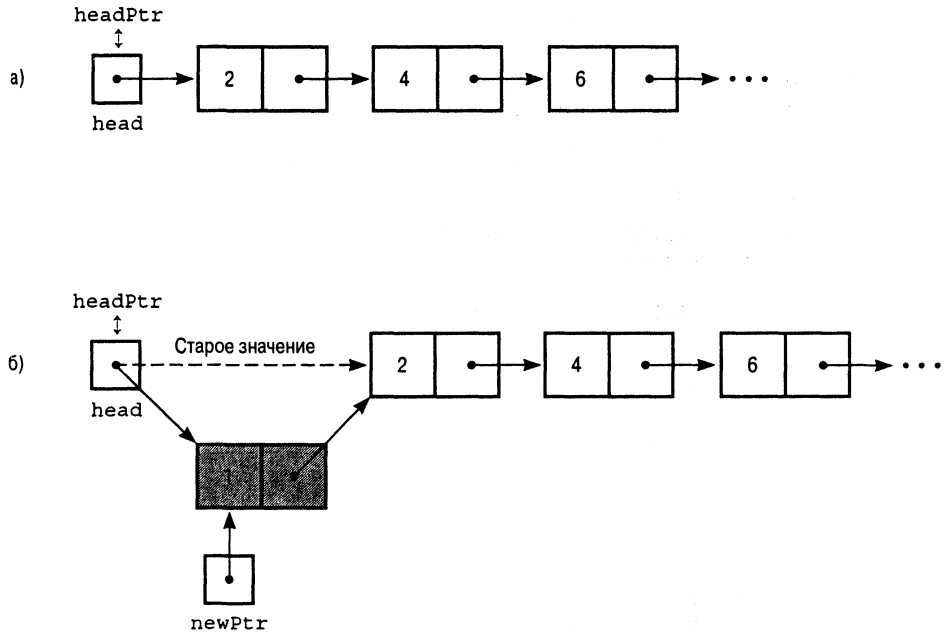


Рис. 4.23. Вставка нового элемента в начало упорядоченного связанного списка: а) упорядоченный связанный список; б) для вставки узла в начало списка выполняется оператор присваивания

Общий случай, когда новый узел вставляется в середину списка, на который ссылается указатель *head*, очень похож на описанный выше. Чему равен фактический аргумент, соответствующий формальному аргументу *headPtr* при выполнении условий базовой задачи? Он равен указателю *next* на узел, предшествующий новому узлу. Иными словами, указатель *next* ссылается на последний узел, значение которого меньше чем значение *newItem*. Следовательно, поскольку указатель *headPtr* передается по ссылке, оператор присваивания *headPtr = newPtr* устанавливает указатель *next* соответствующего узла на новый узел. На рис. 4.24 показаны результаты трассировки рекурсивных вызовов при вставке узла в середину списка.

В заключение рассмотрим контекст функции *linkedListInsert*. Напомним, что в главе 3 был описан упорядоченный абстрактный список. Для этого списка предусмотрена операция *sortedInsert(newItem)*, предназначенная для вставки элемента *newItem* в соответствующее место упорядоченного списка. В реализации упорядоченного списка на основе указателей функция *sortedInsert* должна быть открытым членом класса. Эта функция может вызывать функцию *linkedListInsert* для выполнения рекурсивной вставки. Однако функции *linkedListInsert* в качестве аргумента необходимо передавать указатель на голову списка. Поскольку этот указатель является закрытым членом и скрыт от клиента, функцию *linkedListInsert* не следует включать в перечень операций

над упорядоченным абстрактным списком. Она должны быть закрытым членом класса. Детали этого сценария читатель может уточнить самостоятельно. (См. задание по программированию 2.)

Хотя рекурсивные операции над упорядоченным списком заслуживают отдельного рассмотрения (помимо всего прочего, в них не требуется хранить указатель на текущий узел и выполнять вставку первого узла отдельно), мы описали их с педагогической целью, стремясь подготовить читателей к восприятию алгоритма бинарного поиска узла дерева, описанного в главе 10.

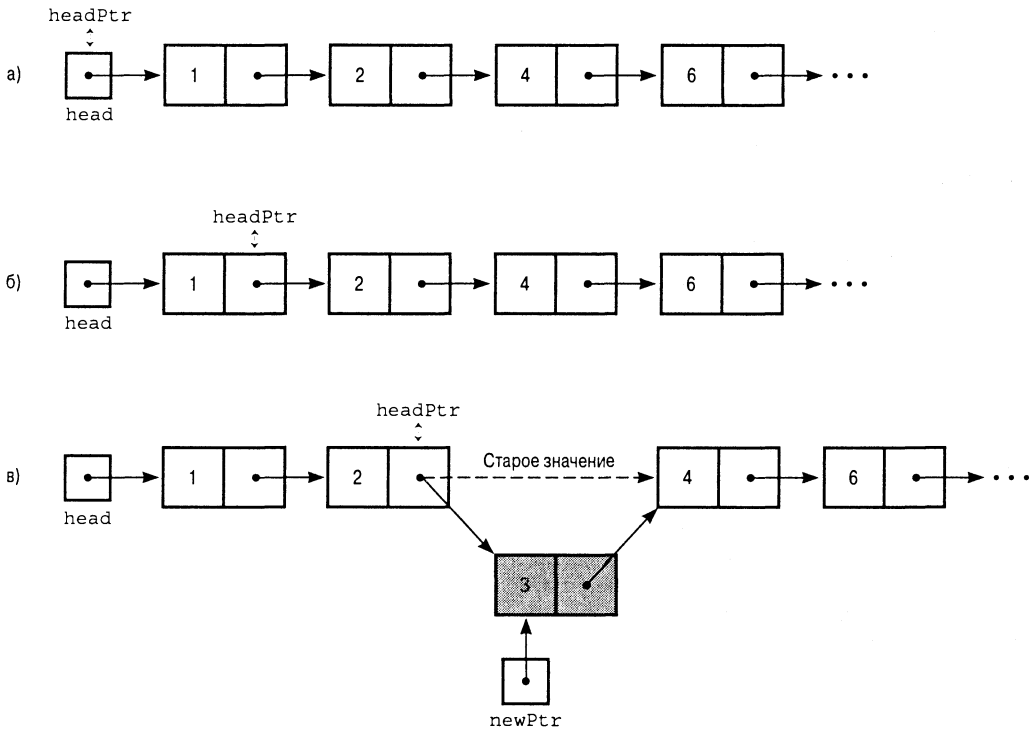


Рис. 4.24. Результаты трассировки рекурсивных вызовов при вставке узла в середину списка: а) первоначальный вызов linkedListInsert(head, 3); б) первый рекурсивный вызов; в) второй рекурсивный вызов, когда в начало списка, на который ссылается указатель headPtr, вставляется новый элемент

Объекты, хранящиеся в узлах списка

В предыдущем разделе мы рассмотрели связанный список, в котором хранились целые числа. Однако это было сделано лишь для простоты. Нет никаких оснований ограничиваться лишь простыми типами данных. На самом деле преимущества связанных списков проявляются намного ярче, когда в их узлах хранятся экземпляры некоторого класса. Допустим, что в связанном списке хранится информация о людях. В этом случае можно определить класс `Person`, содержащий данные, например, имена и адреса, а также операции над ними. Этот связанный список описывается следующими операторами.

```
typedef Person ItemType;
struct Node
{
   ItemType item;
   Node *next;
}; // Конец структуры

Node *head;
```

Данные, хранящиеся в узле связанного списка, могут представлять собой экземпляр класса

Оператор *typedef* позволяет легко изменить тип данных, хранящихся в связанном списке.

Разновидности связанных списков

В этом разделе кратко описываются различные варианты связанных списков. Они часто оказываются очень полезными, и мы будем их применять в дальнейшем. Многие детали реализации этих списков предлагаются читателям в качестве самостоятельных упражнений. Заметим, что кроме списков, рассмотренных в этой главе, существуют много других структур данных, например, массивы указателей или связанные списки, узлами которых являются другие связанные списки. Эти структуры данных также рассматриваются в качестве упражнений.

Кольцевые связанные списки

Работая в компьютерной сети, вы, наравне с другими клиентами, используете ресурсы главного компьютера, называемого **сервером** (server). Аналогичная ситуация возникает, когда вы работаете с главным компьютером, находясь за удаленным терминалом. Система должна организовать работу пользователей так, чтобы в каждый момент времени с главным компьютером работал только один клиент. Упорядочивая пользователей, система выстраивает их в очередь. Поскольку пользователи постоянно входят в систему и выходят из нее, связанные списки их имен позволяют системе поддерживать порядок, не прибегая к их сдвигу при вставке и удалении узлов. Таким образом, система может просматривать связанный список с самого начала и давать каждому пользователю, указанному в списке, возможность использовать ресурсы главного компьютера. Что делать, если система достигла конца списка? Естественно, она должна повторить все с начала. Однако последний узел связанного списка ни на что не ссылается. Это создает определенные неудобства.

Если с последнего узла списка нужно переместиться на первый, нам нужен указатель на голову. Допустим, что вместо константы *NULL* в последний узел записан указатель на голову списка. В результате мы получим **кольцевой связанный список** (circular linked list), показанный на рис. 4.25. В отличие от него, связанный список, описанный ранее, называется **линейным** (linear).

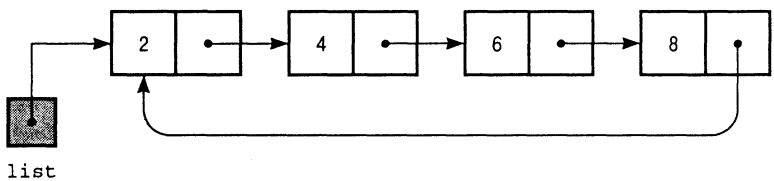


Рис. 4.25. Кольцевой связанный список

Каждый узел в кольцевом связанном списке ссылается на своего преемника, поэтому весь список можно обойти, начиная с любого узла. Может показаться, что кольцевой связанный список не имеет ни головы, ни хвоста. Однако нам по-прежнему будет нужен внешний указатель на один из его узлов. Таким образом, естественнее по-прежнему считать, что в кольцевом связанном списке есть голова и хвост. Если внешний указатель ссылается на "первый" узел, то для того, чтобы достичь последнего узла, нужно, как и раньше, пройти по всему списку. Однако, если внешний указатель — назовем его *list* — ссылается на "последний" узел, как показано на рис. 4.26, чтобы достичь первого и последнего узлов, обход списка выполнять не нужно, поскольку указатель *list->next* ссылается на первый узел.

В кольцевом связанном списке каждый узел имеет преемника

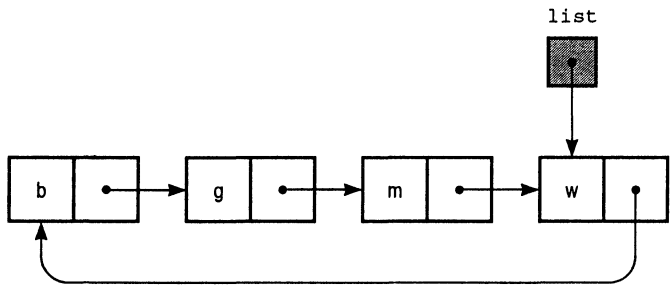


Рис. 4.26. Кольцевой связанный список с внешним указателем на последний узел

Если внешний указатель равен константе *NULL*, значит, список пуст. Однако в кольцевом списке, в отличие от линейного, ни один узел не содержит константы *NULL*. Таким образом, алгоритм полного обхода списка необходимо изменить. Распознать последний узел можно, просто сравнив текущий указатель с внешним указателем *list*. Например, фрагмент кода, приведенный ниже, выводит на экран данные, хранящиеся в каждом узле кольцевого списка, в предположении, что указатель *list* ссылается на последний узел, а функция, предназначенная для вывода данных, использует соответствующий формат.

Вывод на экран данных, содержащихся в кольцевом списке

```
// Выводим на экран данные,
// содержащиеся в кольцевом списке;
// указатель list ссылается на его
// последний узел
if (list != NULL)
{
   // Список не пуст
   Node *first = list->next; // Ссылка на первый узел
   Node *cur = first; // Начало обхода с первого узла
   // Инвариант цикла: указатель cur ссылается на следующий
   // узел, подлежащий выводу на экран
   do
   {
      display(cur->item); // Записываем данные
      cur = cur->next; // Ссылаемся на следующий узел
   } while (cur != first); // Обход завершен?
} // Конец оператора if
```

Операции вставки и удаления элементов кольцевого списка остаются в качестве самостоятельного упражнения.

Фиктивные головные узлы

В алгоритмах вставки и удаления элементов линейного списка первый узел всегда обрабатывался отдельно. Однако многие предпочитают алгоритмы, в которых не возникают особые ситуации. Для этого нужно предусмотреть **фиктивный головной узел** (dummy head node), показанный на рис. 4.27. Он всегда существует, даже если список пуст. В этом случае первый элемент списка на самом деле является вторым. Кроме того, в алгоритмах вставки и удаления элементов указатель *prev* нужно инициализировать адресом фиктивного головного элемента, а не константой *NULL*. Таким образом, в алгоритме удаления элемента, например, оператор

```
prev->next = cut->next;
```

удаляет из списка узел, на который ссылается указатель *cur*, независимо от того, является ли этот узел первым или нет.

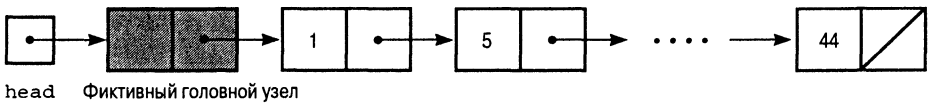


Рис. 4.27. Фиктивный головной узел

И все же несмотря на то что введение фиктивного головного узла позволяет вставлять и удалять все узлы списка аналогичным образом, изменение структуры данных — слишком дорогая цена за это удовольствие. Однако в дважды связанных списках фиктивный головной узел оказывается полезным.

Дважды связанные списки

Каждый узел дважды связанного списка хранит указатели на своих предшественника и преемника

Допустим, нужно удалить конкретный элемент из связанного списка. Если его положение можно вычислить явно, не выполняя обхода списка, ссылка на предшествующий узел не понадобится. Однако для того, чтобы удалить элемент, необходимо знать указатель на его предшественника. Разрешить это противоречие позволяют **дважды связанные списки** (doubly linked lists), поскольку их узлы содержат указатели как на предыдущий, так и на следующий узлы.

Рассмотрим упорядоченный связанный список поставщиков, каждый узел которого содержит, кроме анкетных данных, два указателя, *precede* и *next*. Как обычно, указатель *next* узла *N* ссылается на узел, следующий за узлом *N* в списке. Указатель *precede* ссылается на узел, предшествующий узлу *N* в списке. Вид этого связанного списка поставщиков изображен на рис. 4.28.

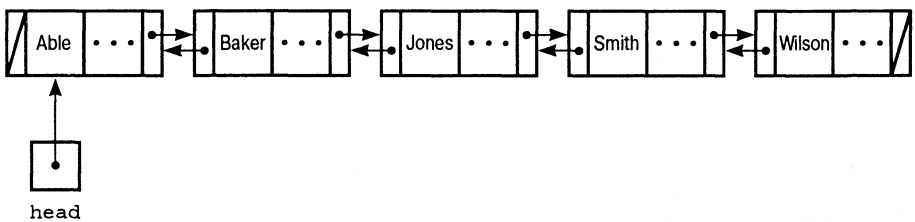


Рис. 4.28. Дважды связанный список

Если указатель *cur* ссылается на узел *N*, указатель на его предшественника можно получить с помощью оператора

```
prev = cur->precede;
```

Таким образом, дважды связанный список позволяет удалить узел, не выполняя обхода всего списка.

Фиктивный головной узел в дважды связанных списках оказывается полезным

Поскольку при вставке и удалении элементов дважды связанного списка задействуется больше указателей, чем в односвязных списках, механизм этих операций немного усложняется. Кроме того, особые ситуации, связанные с первым и последним элементами списка, также становятся сложнее. Ранее мы заметили, что фиктивный головной узел не стоит включать в односвязные списки, однако в случае дважды связанных списков он оказывается намного полезнее.

Как показано на рис. 4.29, *а*, внешний указатель `listHead` всегда ссылается на фиктивный головной узел. Этот узел ничем не отличается от других узлов: он также содержит указатели *precede* и *next*. Список можно замкнуть, образовав **кольцевой дважды связанный список** (circular doubly linked list) . Указатель *next*, хранящийся в фиктивном головном узле, ссылается на "истинно первый" узел списка, а указатель *precede*, записанный в "истинно первом" узле, ссылается обратно, на фиктивный головной узел. Аналогично, указатель *precede*, хранящийся в фиктивном головном узле, ссылается на последний узел списка, а указатель *next* последнего узла — на фиктивный головной узел. Фиктивный головной узел существует, даже если список пуст. В этом случае оба его указателя ссылаются на него, как показано на рис. 4.29, *б*.

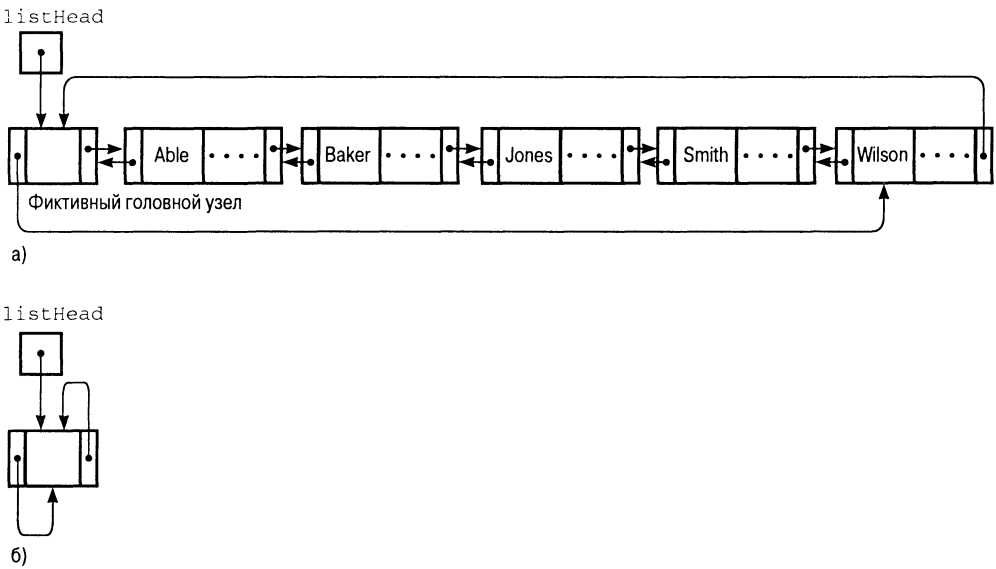


Рис. 4.29. Применение фиктивного головного узла: а) кольцевой дважды связанный список с фиктивным головным узлом; б) пустой список с фиктивным головным узлом

Используя дважды связанные списки, вставку и удаление элементов можно выполнять единообразно: первая и последняя позиции ничем не отличаются от остальных. Рассмотрим, например, процедуру удаления узла *N*, на который ссылается указатель *cur*. Как показано на рис. 4.30, для этого нужно выполнить следующие действия.

В дважды связанном списке вставка и удаление всех элементов выполняются одинаково

1. Изменить указатель *next* в предшественнике узла *N* так, чтобы он ссылался на преемника узла *N*.
2. Изменить указатель *precede* в преемнике узла *N* так, чтобы он ссылался на предшественника узла *N*.

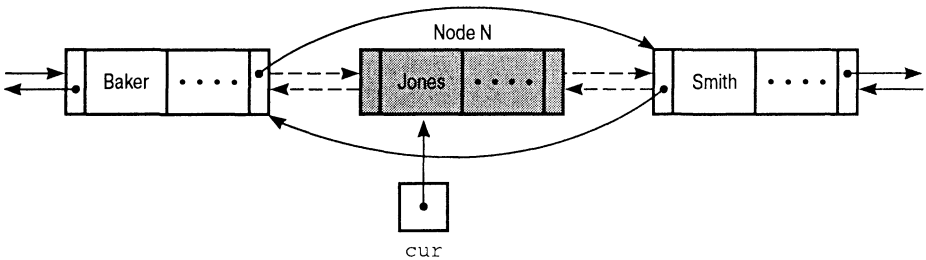


Рис. 4.30. Изменения указателей при удалении элемента

Эти два шага реализуются следующими операторами на языке C++.

Удаление узла

```
// Удаление узла, на который ссылается указатель cur
(cut->precede)->next = cur->next;
(cur->next)->precede = cur->precede
```

Убедитесь сами, что этот код работает одинаково правильно, независимо от того, какой узел удаляется: первый, последний или внутренний. Кроме того, обратите внимание, что скобки, приведенные в этом фрагменте, излишни, поскольку оператор -> является левоассоциативным.

Рассмотрим теперь процедуру вставки узла в кольцевой дважды связанный список. В принципе из факта, что список является дважды связанным, не следует, что достичь любого его элемента можно без обхода. Например, если понадобится вставить в упорядоченный связанный список фамилию нового поставщика, для него сначала нужно найти подходящее место. В псевдокоде, приведенном ниже, указатель *cur* ссылается на узел, содержащий первое имя, которое больше, чем имя *newName* (отношение "больше" авторы понимают в лексикографическом смысле. — *Прим. ред.*). Следовательно, указатель *cur* будет ссылаться на преемника нового узла.

Обход списка для определения места вставки

```
// Поиск места вставки
// cur = listHead->next
// Ссылка на первый узел, если он  есть
while (cur ≠ listHead и newName > cur->item)
   cur = cur->next
```

Если новый узел нужно вставить в конец списка или в пустой список, цикл установит указатель *cur* на фиктивный головной узел.

Как показано на рис. 4.31, если указатель *cur* установлен на преемник нового узла, нужно выполнить следующие действия.

1. Установить указатель *next* нового узла на его преемника.
2. Установить указатель *precede* нового узла на его предшественника.
3. Установить указатель *precede* преемника нового узла на сам новый узел.
4. Установить указатель *next* предшественника нового узла на сам новый узел.

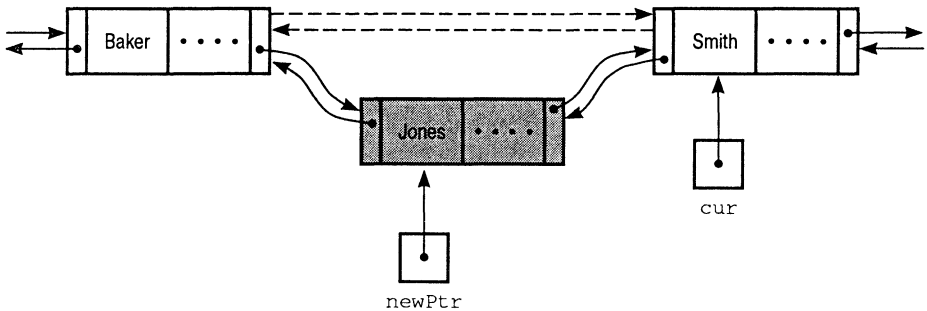


Рис. 4.31. Изменение указателей при вставке нового элемента

Эти четыре шага реализуются следующим фрагментом кода на C++ (предполагается, что указатель *newPtr* ссылается на новый узел).

Вставка нового узла

```
// Вставка нового узла, на который
// ссылается указатель newPtr,
// перед узлом, на который ссылается указатель cur
newPtr->next = cur;
newPtr->precede = cur->precede;
cur->precede = newPtr;
newPtr->precede->next = newPtr;
```

Убедитесь сами, что этот код работает одинаково правильно, независимо от того, какой узел вставляется: первый или последний (в этом случае указатель *cur* ссылается на головной узел), а также пуст ли список (в этом случае указатель *cur* также ссылается на головной узел).

Приложение: инвентарная ведомость

Представьте себе, что вы получили временную работу в местном магазине видеокассет. Узнав, что вы разбираетесь в компьютерах, хозяин магазина предложил вам написать интерактивную программу для работы с инвентарной ведомостью, в которой указаны видеокассеты, предназначенные для продажи. Инвентарная ведомость представляет собой список названий фильмов и сопроводительных комментариев.

- **Имеющиеся кассеты:** количество кассет, имеющихся в магазине.
- **Ожидаемые поступления:** количество кассет, заказанных магазином. (Когда количество имеющихся кассет становится меньше количества ожидаемых, заказываются дополнительные кассеты.)
- **Список очередности:** список людей, заказавших кассеты.

Поскольку хозяин магазина обычно выключает компьютер перед уходом, ваша интерактивная программа не будет работать круглосуточно. Следовательно, данные должны храниться в файле и загружаться по мере надобности.

Входная информация

- Файл, содержащий инвентарную ведомость.
- Файл, содержащий информацию о новых поступлениях. (См. команду D.)
- Однобуквенные команды — с аргументами, если это необходимо, — интерактивно выполняющие запросы или модификацию инвентарной ведомости по требованию пользователя.

Выходная информация

- Файл, содержащий обновленную инвентарную ведомость. (Из списка удаляются записи, в полях которых записаны одни нули: их нет в магазине, они не заказаны магазином и в очереди за ними никто не стоит.)
- Вывод по команде.

Программа должна выполнять следующие команды.

Команды, выполняемые программой

H	Помощь	Выводит на экран краткое описание команд
I<*title*>	Запрос	Выводит на экран комментарии, сопровождающие конкретное название
L	Список	Список всех названий (в алфавитном порядке)
A<*title*>	Добавить	Добавить в список новое название, предложив пользователю ввести количество первоначально заказанных кассет
M<*title*>	Модифицировать	Изменить количество заказанных кассет, имеющих заданное название
D	Доставка	Оформить купленные магазином кассеты при условии, что клерк ввел всю информацию (название и количество) в файл. Считать файл, зарезервировать кассеты для людей, ожидающих очереди, и обновить количество кассет, имеющихся в магазине. Если доставленной кассеты в ведомости еще не было, программа должна добавлять в инвентарную ведомость новую запись
O	Заказ	Сделать заказ на дополнительное количество кассет, сравнив количество имеющихся и ожидаемых кассет в инвентарной ведомости, так чтобы количество заказанных кассет не превышало количество имеющихся
R	Снятие заказа	Снять заказ, сравнив количество имеющихся и ожидаемых кассет в инвентарной ведомости и уменьшив количество заказанных кассет. Цель — сократить количество имеющихся кассет до уровня заказанных
S<*title*>	Продажа	Уменьшить количество кассет с данным названием <*title*> на 1. Если все кассеты с таким названием распроданы, записать имя покупателя в список очередности
Q	Выход	Сохранить инвентарную запись в файле и завершить работу программы.

Решение задачи, от постановки до программы, распадается на три этапа.

1. Разработка решения.
2. Реализация решения.
3. Уточнение программы.

Однако каждый из этих этапов невозможно выполнить по отдельности, независимо от других. На каждом из этих этапов программист должен принимать решения и делать выбор. Хотя дальнейшие рассуждения создают впечатление полной ясности, так бывает не всегда. На самом деле в процессе решения задачи приходится перепробовать много разных вариантов, и наилучший выбор не всегда очевиден.

Поставленная перед нами задача сводится к управлению данными и требует выполнения определенных команд. Эти команды предполагают выполнение следующих операций.

Операции над инвентарной ведомостью

- Перечислить инвентарные записи по алфавиту (команда L).
- Найти инвентарную запись по названию (команды I, M, D, O и S).
- Заменить инвентарную запись, соответствующую заданному названию (команды M, D, R и S).
- Вставить новую инвентарную запись (команды A и D).
- Добавить нового заказчика в конец списка очередности, в котором перечислены люди, ожидающие новых поступлений уже распроданных кассет (команда S).
- Удалить запись о людях, указанных в начале списка очередности, при поступлении заказанных кассет (команда D).
- Вывести имена из списка очередности, в котором перечислены люди, ожидающие конкретную кассету (команды I и L).
- Сохранить текущую инвентарную запись и связанные с ней списки очередности по завершению работы программы (команда Q).
- Загрузить текущую инвентарную ведомость и связанные с ней списки очередности при повторном выполнении программы.

Эти операции можно рассматривать как часть абстрактного типа данных под названием “Инвентарная ведомость”. На следующем шаге каждую операцию нужно полностью описать. Поскольку эта глава посвящена связанным спискам и вопросам их реализации, читатели могут самостоятельно создать полные спецификации этих операций. А мы вернемся к структуре данных, которая воплощает собой инвентарную ведомость.

Каждый элемент инвентарной ведомости представляет собой некий фильм и содержит его название, количество кассет, закупленных магазином, и список очередности. Как реализовать список очередности? Если представить его в виде массива, то придется не только зафиксировать его длину, но и мириться с тем обстоятельством, что каждая инвентарная запись станет довольно большой. Если реализовать его в виде связанного списка, то инвентарная запись будет просто содержать указатель на его голову. Поскольку в конец списка очередности нужно вставлять новые элементы, следует предусмотреть указатель на его хвост, позволяющий более эффективно выполнять операцию вставки. Таким образом, каждый элемент инвентарной ведомости будет также содержать указатель на последнее имя, записанное в списке очередности, как показано на рис. 4.32, *а*.

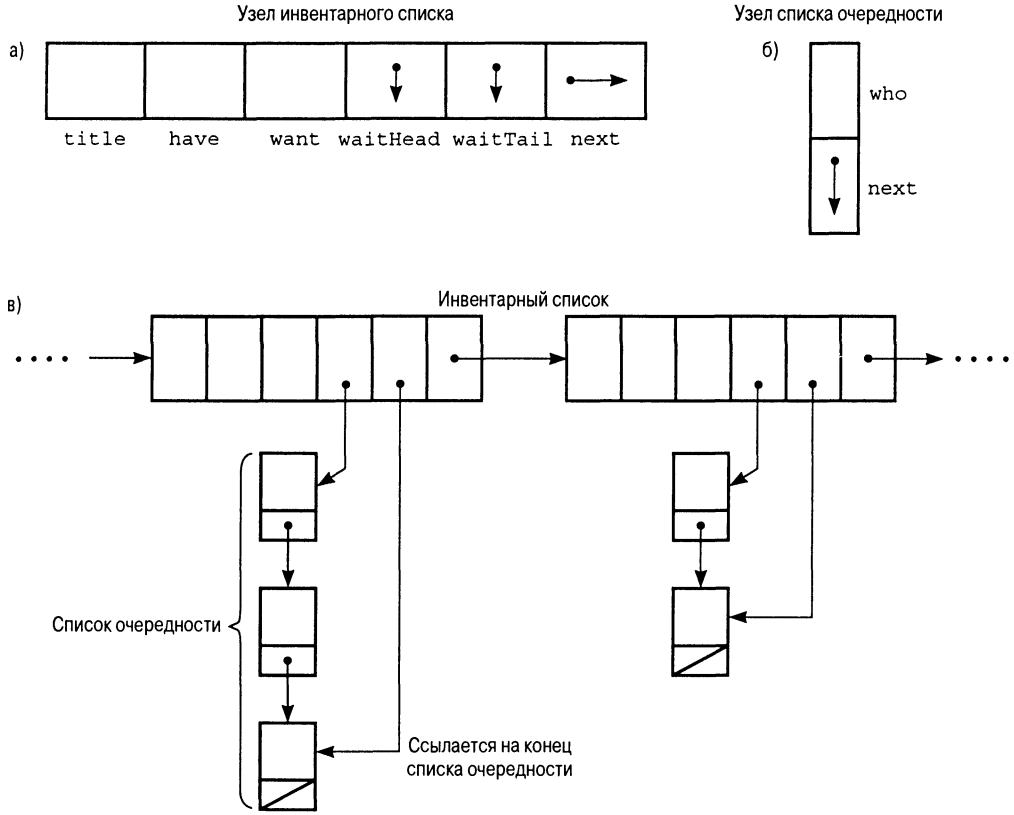


Рис. 4.32. Свойства инвентарной ведомости: а) узел инвентарного списка; б) узел списка очередности; в) ортогональная структура инвентарной ведомости

Поддержка инвентарной ведомости в алфавитном порядке облегчает поиск конкретного названия. Если бы эта ведомость хранилась в массиве, можно было бы применить бинарный поиск. Однако вставка и удаление элементов вынуждают сдвигать элементы массива, что приводит к потере эффективности, если его размер достаточно велик. Применение связанного списка позволяет избежать сдвига данных, однако делает бинарный поиск практически бесполезным. (Как быстро найти середину связанного списка?) Взвешивая все "за" и "против", для реализации инвентарной ведомости мы выбрали связанный список.

Подведем итоги.

Инвентарная ведомость представляется в виде связанного списка

- Инвентарная ведомость представляет собой связанный список элементов, упорядоченных по названиям фильмов.

- Каждая инвентарная запись содержит название, количество имеющихся экземпляров, количество заказанных экземпляров, указатель на начало связанного списка имен людей, ожидающих данной кассеты (список очередности), и указатель на последнее имя в списке очередности.

Сделанный выбор иллюстрируется рис. 4.32 и следующим фрагментом программы на языке C++.

```
// Список очередности — люди, ожидающие определенную кассету
struct WaitNode
{
   string who;
   WaitNode *next;
}; // Конец структуры struct

// Инвентарный список — список кассет, имеющихся в магазине
struct StockNode
{
   string title;
   int have, want;
   WaitNode *waitHead, *waitTail;
   StockNode *next;
}; // Конец структуры
```

Прежде всего следует рассмотреть вопрос, как инвентарная ведомость хранится в файле. Напомним, что в файле должны храниться только данные, содержащиеся в узлах связанного списка. Однако в нашей задаче есть небольшая сложность: каждая инвентарная запись содержит указатели и соответствующий подсписок — список очередности. Итак, можно предпринять следующие действия.

- Можно изменить структуру инвентарной ведомости, как показано на рис. 4.33.
- Можно использовать вспомогательный файл, состоящий из имен, указанных в списке очередности.
- Для того чтобы восстановить списки очередности, нужно знать их длину. В частности, если все имена лиц, перечисленных в списке очередности, хранятся вместе в одном файле, можно определить, где заканчивается один список и начинается следующий.

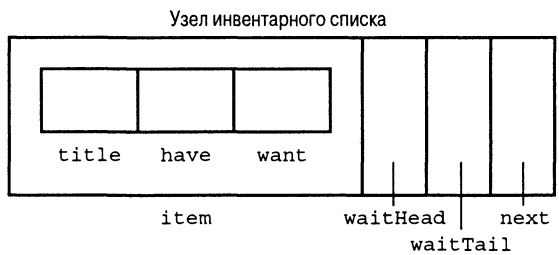


Рис. 4.33. Модифицированная структура узла

Вспомогательный файл не создает никаких проблем, однако нам придется изменить первоначальную постановку задачи. Это изменение вполне возможно.

Прежде чем перейти к третьему утверждению, напомним, что количество кассет, имеющихся в магазине, не может быть отрицательным. Следовательно, имеет смысл принять следующее соглашение: отрицательное количество кассет, имеющихся в магазине, означает количество недостающих кассет — а именно это число и определяет длину списка очередности!

Уточним предыдущие определения с учетом сделанных замечаний.

Уточненная структура данных

```
// Список очередности — люди, ожидающие
//определенную кассету
struct WaitNode
{
   string who;
   WaitNode *next;
}; // Конец структуры

// Инвентарная запись
class StockItem
{
public:
. . .
private:
   string title;
   int have, want;
}; // Конец определения класса
// Инвентарный список — список инвентарных записей
struct StockNode
{
   StockItem item;
   WaitNode *waitHead, *waitTail;
   StockNode *next;
}; // Конец структуры
```

В этом фрагменте нет операции удаления инвентарной записи. Из инвентарной ведомости удаляются лишь записи, которые содержат в своих полях нули и чей список очередности пуст. Они просто не сохраняются в файле при выходе из программы по команде Q. Поскольку это происходит при выходе из программы — и вся память при этом освобождается автоматически, — нет смысла предусматривать явное удаление конкретных узлов.

Полное решение этой задачи читатели могут закончить самостоятельно.

Стандартная библиотека шаблонов языка C++

Во многих современных языках программирования, таких как язык C++, предусмотрены стандартные классы, которые используются наиболее широко. В языке C++ большинство таких классов определено в **стандартной библиотеке шаблонов** (Standard Template Library — STL). Эта библиотека содержит огромное количество шаблонных классов, которые можно применять для реализации любых типов данных.

Стандартная библиотека шаблонов содержит шаблонные классы для некоторых широко распространенных абстрактных типов данных

Многие из абстрактных типов данных (АТД), рассмотренных в книге, уже реализованы в виде шаблонного класса из библиотеки STL. Например, в библиотеке STL содержится класс *list*, совершенно аналогичный классу *List*, рассмотренному нами выше. Возникает вопрос: "Зачем мы потратили столько времени на разработку собственного абстрактного типа данных, если он уже реализован в виде шаблона?" Для этого есть несколько причин.

Причины, по которым имеет смысл разрабатывать свои собственные реализации абстрактных типов данных

- Разработка простых абстрактных типов данных — первая ступенька к более сложным проектам.
- Нельзя слепо полагаться на шаблоны. Может оказаться, что для конкретной задачи понадобится разработать

оригинальный абстрактный тип данных, поэтому нужно хорошо знать, как это делается.

- Если шаблонный класс не полностью соответствует вашим требованиям, нужно разрабатывать свой собственный АТД.

Контейнер — это объект, в котором хранятся другие объекты

Элементы стандартной библиотеки шаблонов разделяются на три вида: контейнеры, алгоритмы и итераторы. **Контейнеры** (containers) — это объекты, внутри которых содержатся другие объекты. Например, список — типичный контейнер. К контейнерам применяются **алгоритмы**, например, алгоритм сортировки списка. **Итераторы** (iterators) осуществляют просмотр содержимого контейнеров. В данный момент нас интересуют контейнеры и итераторы.

Контейнеры

В основе контейнеров лежит понятие **шаблонного класса** (class template), принятое в языке C++. Шаблоны позволяют разрабатывать класс, не уточняя типы используемых в нем данных до момента его применения. Например, наш класс *list* был разработан совершенно независимо от типа элементов, которые в нем содержатся. В его реализации был использован оператор *typedef*, который определял истинный тип *ListItemType*. В шаблонах этот тип задается в качестве **шаблонного параметра** (data-type parameter). Перед определением класса указываются ключевые слова *template <class T>*. Здесь параметр *T* задает тип данных, указанных клиентом. Рассмотрим пример простого шаблонного класса.

```
template <class T> class MyClass
{
public:
   MyClass();
   MyClass(T initialData);

   void setData(T newData);
   T getData();
private:
   T theData;
};
```

Создание собственных шаблонных классов описано в главе 8.

Объявляя экземпляр этого класса, необходимо указывать фактический тип данных, представленных параметром *T*. Простая программа, использующая этот шаблонный класс, может начинаться следующим образом.

```
int main()
{
   MyClass<int> a;
   MyClass<double> b(5.4);

   a.setData(5);
   cout << b.getdata() << endl;
}
```

Обратите внимание, как в объявлениях объектов *a* и *b* задан тип переменной *theData*, являющейся членом класса *MyClass*.

Фактические шаблоны из библиотеки STL используют два шаблонных параметра. Первый из них представляет собой обычный параметр, задающий тип данных, содержащихся в контейнере. Второй параметр называется **распредели-**

телем (allocator). В большинстве случаев он используется по умолчанию и является объектом класса *allocator*. По этой причине в определениях шаблонов, приведенных в книге, мы будем игнорировать этот параметр.

Итераторы

Итераторы — это обобщенные указатели. Они дают возможность перемещаться по объектам, содержащимся в контейнере, совершенно аналогично тому, как обычный указатель позволяет перемещаться по элементам списка. Предположим, что итератор называется *curr*. В этом случае получить доступ к объекту, на который ссылается итератор *curr*, можно с помощью выражения **curr*.

Итераторы, предусмотренные в библиотеке STL, разделяются на пять категорий, в зависимости от операций, которые можно выполнить над итератором. Шаблонные контейнерные классы, представленные в книге, являются **двунаправленными итераторами** (bidirectional iterator). Такие итераторы позволяют перемещаться по контейнеру в обоих направлениях. Для перемещения на следующий элемент контейнера используется оператор инкрементации

```
++curr;
```

Для перемещения на предыдущий элемент контейнера используется оператор декрементации

```
--curr;
```

В контейнерных классах обычно содержатся по крайней мере две функции. Первая из них инициализирует итератор ссылкой на первый элемент контейнера.

```
iterator begin();
```

Вторая функция возвращает значение, которое сообщает, достигнут ли конец контейнера.

```
iterator end();
```

Это же значение возвращается функцией *begin*, если список пуст (оно не равно константе *NULL*).

```
list<int> myList;
list<int>::iterator curr;

// В данный момент список пуст;
//итератор ссылается на начало списка myList
curr = myList.begin();
// Пуст ли список?
if (curr == myList.end())
   cout << "Список пуст" << endl;

// Вставить в список myList пять элементов
for (int j = 0; j < 5; j++)
// Поместить элемент j в начало списка
curr = myList.insert(curr, j);

// Вывести все элементы списка, начиная с первого,
// перемещая итератор на следующий элемент, пока не будет
// достигнут конец списка
cout << *curr << " ";
cout << endl;
```

Если итератор предварительно не проинициализировать, его поведение может оказаться непредсказуемым.

В главе 8 показано, как создать свой собственный итератор.

Шаблонный класс list из библиотеки STL

Рассмотрим листинги нескольких методов из класса *list*, принадлежащего библиотеке STL.

```
template <class T> class std::list
{
public:
   list();
   // Конструктор по умолчанию; создает пустой список.
   // Предусловие: нет.
   // Постусловие: создан пустой список.

   list(size_type num, const T& val = T());
   // Конструктор. Создает список, состоящий из num элементов,
   // имеющих значение val.
   // Предусловие: нет.
   // Постусловие: создан список, состоящий из num элементов.

   list(const list<T> & anotherList);
   // Конструктор. Создает список, состоящий из тех же
   // элементов,что и список anotherList.
   // Предусловие: нет.
   // Постусловие: создан список, состоящий из тех же
   // элементов,что и список anotherList.

   bool empty() const;
   // Распознает, пуст ли список.
   // Precondition: None.
   for (curr = myList.begin(); curr != myList.end(); curr++)
   // Постусловие: если список пуст, возвращает значение true.
   // В противном случае возвращает значение false.

   size_type size() const;
   // Вычисляет длину списка.
   // Тип size_type является целочисленным.
   // Предусловие: нет.
   // Постусловие: возвращает количество элементов,
   // хранящихся в списке в данный момент.

   size_type max_size();
   // Определяет максимально возможное количество элементов,
   // которые могут храниться в списке.
   // Предусловие: нет.
   // Постусловие: возвращает максимально возможное
   // количество элементов

   iterator insert(iterator i, const T& val = T());
   // Вставляет элемент val в список непосредственно
   // перед элементом, на который установлен итератор.
   // Предусловие: итератор должен быть проинициализирован,
   // даже если список пуст.
```

```
    // Постусловие: элемент val вставлен в список, возвращен
    // итератор, установленный на вновь вставленный список.

    void remove(const T& val);
    // Удалить из списка все элементы, имеющие значение val.
    // Предусловие: нет.
    // Постусловие: в списке нет элементов, имеющих значение val.

    iterator erase(iterator i);
    // Удаляет из списка элемент, на который установлен
    // итератор i.
    // Предусловие: итератор должен быть проинициализирован
    // каким-либо элементом списка.
    // Постусловие: возвращает итератор, установленный
    // на элемент, следующий за удаленным. Если удаленный
    // элемент был последним, значение итератора должно быть
    // таким же, как возвращаемое значение функции end().

    iterator begin();
    // Возвращает итератор, установленный
    // на первый элемент списка.
    // Предусловие: нет.
    // Постусловие: если список пуст, значение итератора
    // должно быть таким же, как возвращаемое значение
    // функции end().

    iterator end();
    // Возвращает значение итератора, которое можно использовать
    // для проверки, достигнут ли конец списка.
    // Предусловие: нет.
    // Постусловие: возвращается итератор, установленный
    // на конец списка

} // Конец шаблонного класса
```

Посмотрим, как используется этот класс для работы со списком бакалейных товаров.

```
#include <list>
#include <iostream>
#include <string>
using namespace std;

int main()
{
   list<string> groceryList; // Создаем пустой список
   list<string>::iterator i = groceryList.begin();
   i = groceryList.insert(i, "яблоки");
   i = groceryList.insert(i, "хлеб");
   i = groceryList.insert(i, "сок");
   cout << "Количество элементов списка: "
   << groceryList.size() << endl;

   cout << "Элементы списка:" << endl;
   i = groceryList.begin();
   while (i != groceryList.end())
   {
      cout << *i << endl;
```

```
      i++;
    } // Конец оператора while
} // Конец функции main
```

Вывод программы имеет следующий вид.

```
Количество элементов списка: 3
Элементы списка:
яблоки
хлеб
сок
```

Резюме

1. Оператор *new* позволяет выделять динамическую память для массива или связанного списка. Оператор *delete* освобождает память, выделенную оператором *new*.
2. Указатели — чрезвычайно полезный тип данных. Их можно применять для реализации связанного списка, используя следующие операторы.

   ```
   struct Node
   {
     тип_данных item;
     Node *next;
   } // Конец структуры
   ```

   Каждый указатель в связанном списке ссылается на структуру, т.е. на узел списка. Например, если указатель *p* имеет тип *Node** и ссылается на узел связанного списка, то

   значением выражения **p* является узел;

   значением выражения *p->item* является поле узла, в котором записана информация;

   значением выражения *p->next* является указатель на следующий узел.
3. Алгоритмы вставки данных в связанный список и удаления их оттуда включают в себя следующие шаги: обход списка от начала до искомой позиции; изменение указателей, приводящее к модификации структуры данных. Кроме того, во время выполнения операции вставки производится выделение динамической памяти для нового узла с помощью оператора *new*, а при удалении элемента из списка — освобождение занимаемой им памяти с помощью оператора *delete*.
4. Вставка и удаление первого элемента списка представляют собой отдельные задачи. Они отличаются от вставки и удаления других узлов.
5. Реализация абстрактного списка в виде массива подразумевает неявное упорядочение. Например, за элементом *anArray[i]* в массиве обязательно следует элемент *anArray[i+1]*. Реализация списка с помощью указателей подразумевает явное упорядочение: для того чтобы найти узел *N*, нужно вычислить указатель на этот узел. Следовательно, в реализации списка с помощью указателей нужно выделить дополнительную память для их хранения.
6. К любому элементу массива можно обратиться непосредственно по его индексу. В связанном списке для доступа к узлу необходимо выполнить обход списка. Следовательно, время доступа к элементам массива постоянно, а к элементам списка — зависит от их местоположения.

7. При вставке и удалении элементов связанного списка не нужно выполнять сдвиг данных. Это свойство является очень важным преимуществом связанных списков над массивами.
8. Класс, в котором выделяется динамическая память, должен содержать явный конструктор копирования, предназначенный для копирования экземпляров этого класса. Этот конструктор вызывается неявно, если объект передается функции по значению, возвращается функцией в виде результата, а также когда при определении объекта выполняется его инициализация.
9. Класс, в котором выделяется динамическая память, должен содержать явный деструктор. Этот деструктор должен использовать оператор `delete`, освобождая память, занятую объектом. Если такой деструктор не задать явно, то компилятор сам сгенерирует его. Автоматический деструктор подходит только для классов, в которых выделяется исключительно статическая память.
10. Несмотря на то что с помощью оператора `new` можно выделять динамическую память и для массива, и для списка, операция увеличения размера на один элемент для связанных списков выполняется эффективнее, чем для массивов.
11. Бинарный поиск в связанном списке совершенно неэффективен, поскольку в таком списке невозможно быстро вычислить средний элемент.
12. Связанный список можно хранить в файле, записывая туда данные, содержащиеся в узлах. В результате связанный список можно легко восстановить, прочитав его данные из файла.
13. Для выполнения операций над связанным списком можно применять рекурсию. Это позволяет единообразно выполнять все операции и не хранить указатель на текущий узел.
14. Рекурсивный алгоритм вставки нового элемента в упорядоченный связанный список основан на предположении, что связанный список меньшего размера также упорядочен. Этот алгоритм выполняет вставку в начало одного из таких списков, причем вставляемый элемент оказывается на нужном месте исходного списка. Завершение этого алгоритма гарантируется условием, что список меньшего размера содержит меньше элементов, чем его предшественник, а вставка элемента в пустой список является базовой задачей.
15. В кольцевом связанном списке последний узел ссылается на первый, так что у каждого узла есть предшественник. Если внешний указатель на список ссылается на последний узел, а не на первый, для доступа к первому и последнему элементу не обязательно выполнять обход списка.
16. Фиктивный головной узел позволяет избежать особых случаев, когда новый элемент вставляется в начало списка или удаляется оттуда. Его применение является делом вкуса, однако в дважды связанных списках он, несомненно, полезен.
17. Дважды связанный список можно обходить в любом направлении. Каждый его узел ссылается как на своего предшественника, так и на преемника. Поскольку вставка и удаление узлов дважды связанного списка немного сложнее, чем аналогичные операции над односвязным списком, имеет смысл применять фиктивный головной узел и кольцевую структуру, что позволяет исключить особые случаи, связанные с первым и последним элементами.
18. Шаблонный класс позволяет отложить уточнение типа данных до момента его применения.

19. Стандартная библиотека шаблонов содержит шаблонные классы для наиболее распространенных абстрактных типов данных.
20. Контейнер — это объект, в котором содержатся другие объекты. Итератор позволяет перемещаться по объектам внутри контейнера.

Предупреждения

1. Поскольку значением указателя является адрес, он редко выводится на экран.
2. Хотя несколько переменных, имеющих одинаковый тип, можно объявлять одновременно, разделяя их имена запятой, к указателям это не относится. Например, в строке

   ```
   int a, b, c;
   ```

   объявлены целочисленные переменные `a`, `b` и `c`, однако объявление

   ```
   char *s, t;
   ```

   означает

   ```
   char *s; char t;
   ```

   а вовсе не

   ```
   char *s; char* t;
   ```

   Таких конструкций следует избегать.
3. Неинициализированный указатель имеет неопределенное значение. Он не равен константе *NULL*.
4. Если указатель *p* не инициализирован, ссылка **p* может оказаться непредсказуемой и даже привести к катастрофическим результатам.
5. Не следует ссылаться на указатель, равный константе *NULL*, хотя это не запрещено. Например, если указатель *p* равен константе *NULL*, нужно избегать выражений типа *p->item* и *p->next*, иначе программа будет работать неправильно.
6. Операторы

   ```
   p = new int;
   p = NULL;
   ```

   сначала выделяют одну ячейку памяти, а затем разрушают единственное средство для доступа к ней. Не используйте оператор *new*, если вы хотите просто присвоить указателю какое-то значение.
7. Помните, что оператор *delete p* уничтожает узел, на который ссылался указатель *p*. Сам указатель *p* не удаляется. Он продолжает существовать, имея неопределенное значение. После выполнения оператора *delete p* нельзя использовать ни сам указатель *p*, ни любой другой указатель, ссылающийся на ту же область памяти. Чтобы избежать таких ошибок, после выполнения оператора *delete p* указателю *p* нужно присвоить константу *NULL*. Однако если в программе существуют другие указатели, ссылающиеся на тот же узел, опасность ошибки остается.
8. Если память была выделена с помощью оператора *new*, ее нужно освободить с помощью оператора *delete*. Если память была выделена с помощью оператора *new []*, она освобождается с помощью оператора *delete []*.

9. Вставка и удаление первого элемента связанного списка представляет собой отдельную задачу, если не использовать фиктивный головной узел. В противном случае возможна ссылка на указатель, равный константе *NULL*.

10. При обходе связанного списка с помощью указателя *cur* нужно быть осторожным и не ссылаться на него, когда он "заходит" за последний элемент списка и становится равным константе *NULL*. Например, цикл

```
while (value > cur->item)
   cur = cur->next;
```

неверен, если значение *value* больше, чем все значения, хранящиеся в связанном списке, поскольку указатель *cur* становится равным константе *NULL*. Вместо этого нужно написать

```
while ((cur != NUL) && (value > cur->item))
   cur = cur->next;
```

Поскольку в языке C++ принято сокращенное вычисление логических выражений, то когда указатель *cur* станет равным константе *NULL*, выражение *cur->item* вычисляться не будет.

11. Программисты слишком часто применяют дважды связанные списки. Однако они вполне приемлемы, если существует прямой доступ к элементам списка. В этих случаях не обязательно выполнять обход списка с самого начала. Если список односвязный, в нем нет указателей на предыдущий узел. Поскольку в дважды связанных списках и предшественник, и преемник одинаково легкодоступны, удаление элемента выполняется очень просто.

12. Если массив передается функции в качестве аргумента, то фактически передается лишь указатель на его первый элемент, а доступ к остальным элементам массива осуществляется с помощью этого указателя. Следовательно, любое присваивание значения какому-либо элементу, приводит к изменению исходного массива. По этой причине массив невозможно передать по значению.

Вопросы для самопроверки

1. Выполните вручную трассировку следующей программы.

```
#include <iostream>
using namespace std;

int main()
{
   int *p = new int;
   int *q = new int;
   cout << p << " " << q << endl;
   *p = 7;
   *q = 11;
   int *r = q;
   cout << *p << " " << *q << " " << *r << endl;

   *r = *q + *p;
   cout << *p << " " << *q << " " << *r << endl;

   q = new int;
```

```
    *q = 4;
    cout << *p << " " << *q << " " << *r << endl;

    delete r;
    r = NULL;
    return 0;
} // Конец функции main
```

2. Рассмотрите алгоритм удаления узла из связанного списка, описанный в этой главе.

   2.1. Является ли удаление последнего узла отдельной задачей? Обоснуйте свой ответ.

   2.2. Является ли удаление единственного узла отдельной задачей? Обоснуйте свой ответ.

   2.3. Какой из узлов сложнее удалить: первый или последний? Обоснуйте свой ответ.

3. Выполните следующие задания.

   3.1. Напишите фрагмент программы на C+, который создавал бы связанный список, изображенный на рис. 4.34, следующим образом: сначала создается пустой список, затем создается и присоединяется узел, содержащий символ J, затем создается и присоединяется узел, содержащий символ E, и в заключение создается и присоединяется узел, содержащий букву B.

   3.2. Повторите задание 3.1, создавая узлы в следующем порядке: B, E, J.

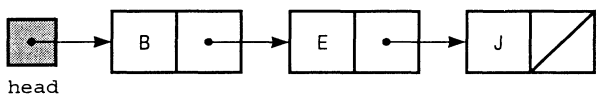


Рис. 4.34. Связанный список для упражнений 3, 4 и 8

4. Проанализируйте упорядоченный связанный список, состоящий из отдельных символов, изображенный на рис. 4.34. Предположим, что указатель *prev* ссылается на первый узел этого списка, а указатель *cur* — на второй.

   4.1. Напишите фрагмент программы на языке C++, который удалял бы второй элемент и освобождал занятую им память. (*Подсказка*: сначала измените рис. 4.34.)

   4.2. Допустим теперь, что указатель *cur* ссылается на первый из оставшихся двух узлов исходного списка. Напишите фрагмент программы на языке C++, который удалял бы первый из них, освобождая область памяти, которую тот занимал.

   4.3. Пусть теперь указатель *head* ссылается на единственный элемент, оставшийся в списке. Напишите фрагмент программы на языке C++, который вставлял бы новый элемент, содержащий букву A, оставляя список упорядоченным.

   4.4. Измените рис. 4.34 так, чтобы новая диаграмма отображала результаты предыдущих вставок и удалений.

5. Напишите фрагмент программы на языке C++, который выводил бы на экран только i-е целое число из связанного списка. Предполагается, что $i \geq 1$, а связанный список состоит по крайней мере из i узлов.

6. Сколько операторов присваивания выполняет функция, разработанная вами в рамках предыдущего задания?

7. Напишите на языке C++ рекурсивную функцию, извлекающую i-е целое число из связанного списка. Предполагается, что $i \geq 1$, а связанный список состоит по крайней мере из i узлов. (*Подсказка*: если $i = 1$, возвращается первое целое число; в противном случае из оставшейся части списка извлекается (i-1)-е целое число.)

8. Выполните трассировку функции *writeBackward2(head)*, где указатель *head* ссылается на связанный список символов, изображенных на рис. 4.34. (Функция *writeBackward2* упоминается в этой главе на стр. 204.)

9. Реализация деструктора в классе *List* содержит следующий цикл.

```
while (!isEmpty())
   remove()
```

   9.1. Зачем нужен этот цикл?

   9.2. Можно ли заменить этот цикл следующим фрагментом кода?

```
for (int position = getLength(); position >= 1)
   --position)
remove(1);
```

   9.3. Изменится ли ваш ответ на вопрос 9.2, если оператор *remove(1)* заменить оператором *remove(position)*?

   9.4. Изменятся ли ваши ответы на вопросы 9.2 и 9.3, если оператор *for* заменить оператором

```
for (int position = 1; position <= getLength(); ++position)
```

10. Измените деструктор класса *List* так, чтобы он непосредственно удалял каждый узел связанного списка, не вызывая функцию *remove*.

Упражнения

1. Предположим, что вы уже выполнили задание 4 из предыдущего раздела и знаете финальное состояние связанного списка. Выполняя каждое из следующих заданий, напишите на языке C++ функции, выполняющие соответствующие операции над связанным списком. Изобразите на рисунке состояние списка после выполнения каждой из этих операций. Удаляя узел из списка, освобождайте занимаемую им память. Все вставки в список должны сохранять его упорядоченным. Не используйте функции, описанные в главе.

   1.1. Допустим, что указатель *prev* ссылается на первый узел списка, а указатель *cur* — на второй. Вставьте в список букву F.

   1.2. Допустим, что указатель *prev* ссылается на второй узел, а указатель *cur* — на третий узел списка, полученного в результате выполнения предыдущего задания. Удалите последний узел списка.

   1.3. Допустим, что указатель *prev* ссылается на последний узел списка, полученного в результате выполнения предыдущего задания. Вставьте в список букву G.

2. Рассмотрим связанный список, элементы которого записаны в произвольном порядке.

2.1. Напишите функцию, вставляющую узел в начало связанного списка и функцию, удаляющую этот узел.

2.2. Повторите предыдущее задание, на этот раз вставляя и удаляя не первый, а последний элемент списка.

2.3. Повторите предыдущее задание, считая что существуют указатели на голову и хвост списка.

3. Напишите функцию для подсчета количества узлов в связанном списке.

3.1. Итеративно.

3.2. Рекурсивно.

4. Напишите функцию для удаления узла, содержащего наибольшее целое число, из связанного списка целых чисел. Можно ли решить эту задачу, выполнив лишь один обход списка?

5. В разделе "Рекурсивная обработка связанных списков" был рассмотрен связанный список символов, образующих строку.

5.1. Напишите итеративную функцию, выводящую на экран эту строку. Сравните эффективность этой функции и функции *writeString*.

5.2. Напишите итеративную функцию, выводящую на экран строку, записанную в обратном порядке. Сравните эффективность этой функции и функции *writeString2*.

6. Напишите функцию для слияния двух связанных списков, упорядоченных в порядке возрастания. В результате должен возникнуть третий список, представляющий собой упорядоченное объединение двух исходных списков. Исходные списки разрушаться не должны.

7. В разделе "Запись связанных списков в файл и считывание их из файла" во фрагменте программы, выполняющем считывание списка из файла, требовалось, чтобы указатель *head* сначала был равен константе *NULL*. Что случится, если это условие будет нарушено? Иными словами, что произойдет, если указатель *head* уже ссылается на голову какого-то непустого связанного списка и в это время выполняется считывание прежнего списка из файла?

8. Сравните количество операций, необходимых для вывода на экран каждого узла связанного списка, состоящего из целых чисел, и количество операций, выполняемых при выводе на экран элементов массива. Цикл, выводящий на экран каждый элемент связанного списка, описан в разделе "Вывод на экран содержания связанного списка".

9. Сравните реализации абстрактной операции *remove(index)* для массива и связанного списка. Опишите ее работу в каждом из этих вариантов при разных значениях индекса. Как изменится эффективность этой операции, если стоимость сдвига элементов массива намного превышает стоимость отслеживания указателя? Когда возникают такие операции? Оцените эффективность операции, если указанные выше стоимости приблизительно равны.

10. Допустим, что указатель *list* ссылается на последний узел кольцевого связанного списка, изображенного на рис. 4.26. Напишите цикл, выводящий на экран данные, содержащиеся в каждом узле этого списка.

11. Предположим, задан класс, в котором кольцевой связанный список является его единственным членом. Напишите его функции-члены.

11.1. Деструктор.

11.2. Конструктор копирования.

12. Напишите функцию, удаляющую i-й узел кольцевого связанного списка.
13. Представьте себе кольцевой связанный список целых чисел, упорядоченных в возрастающем порядке, как показано на рис. 4.35, *а*. Внешний указатель `list` ссылается на последний узел, содержащий наибольшее целое число. Напишите функцию для сортировки этого списка в убывающем порядке, как показано на рис. 4.35. Выделять или освобождать память не нужно.

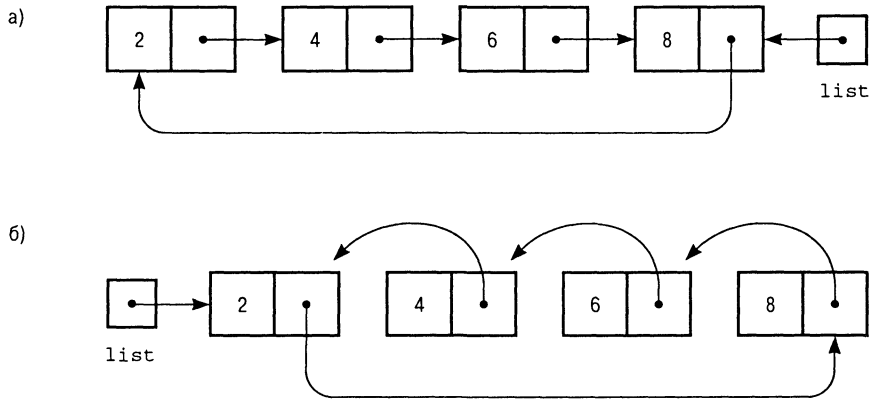


Рис. 4.35. Два кольцевых связанных списка

14. Измените реализацию операций `insert` и `remove` над абстрактным списком, при условии что связанный список содержит фиктивный головной узел.
15. Добавьте в реализацию абстрактного списка на основе указателей операции `save` и `restore`. Эти операции записывают список в файл и считывают его оттуда.
16. Проанализируйте дважды связанный список, изображенный на рис. 4.29, *а*. Этот список является кольцевым и содержит фиктивный головной узел. Допустим, что узел `newName` содержит имя, которое нужно добавить в этот список. Напишите на языке C++ код, вставляющий в упорядоченный список новый узел, содержащий имя `newName`.
17. Проанализируйте дважды связанный список, изображенный на рис. 4.29, *а*. Этот список является кольцевым и содержит фиктивный головной узел. Допустим, что узел `newName` содержит имя, которое нужно удалить из списка. Напишите на языке C++ код, удаляющий из списка узел, содержащий имя `newName`.
18. Повторите упражнения 16 и 17 для упорядоченного дважды связанного списка, показанного на рис. 4.28. Этот список не является кольцевым и не содержит фиктивный головной узел. Рассмотрите отдельные задачи, связанные с началом и концом этого списка.
19. Допустим, что информация о некоторых людях — имя, номер их карточки социального страхования, адрес и т.п. — представлена в виде набора структур. Предположим также, что для хранения этих структур используется абстрактный список. Какие изменения нужно внести в спецификацию и реализацию списка и нужно ли это делать? Что произойдет, если вместо структур в списке используются классы?
20. Можно создать связанный список, состоящий из связанных списков, как показано на рис. 4.32. Будем придерживаться определений классов, приве-

денных на стр. 218. Допустим, что указатель *cur* ссылается на искомую инвентарную запись (узел). Напишите фрагмент программы на языке C++, который добавлял бы имя в конец списка очередности, связанного с узлом, на который ссылается указатель *cur*.

Задания по программированию

1. В главе 3 был рассмотрен абстрактный упорядоченный список. Например, список имен, перечисленных в алфавитном порядке, и список целых чисел, записанный в порядке возрастания, являются упорядоченными. Операции над абстрактным упорядоченным списком были перечислены в разделе "Абстрактный упорядоченный список" в главе 3.

   Некоторые операции — например, *sortedIsEmpty*, *sortedLength* и *sortedRetrieve*, — полностью совпадают с аналогичными операциями над обычным списком. Однако операции вставки и удаления выполняются в зависимости от значения элемента, а не от его позиции в списке. Например, при вставке нового элемента в упорядоченный список его позиция не указывается. Вместо этого искомая позиция вставляемого элемента определяется путем сравнения его значения со значениями элементов списка. Для этого предусмотрена новая операция, *locatePosition*, позволяющая по значению элемента определить его положение в списке.

   В спецификациях, приведенных в главе 3, ничего не сказано о дубликатах, которые могут содержаться в списке. В зависимости от конкретного приложения дубликаты могут допускаться или запрещаться. Например, в упорядоченном списке номеров карточек социального страхования дубликаты должны быть запрещены. Следовательно, попытка вставить номер, совпадающий с уже существующим, окажется безуспешной.

   Пользуясь указателями, напишите нерекурсивную реализацию абстрактного упорядоченного списка, состоящего из целых чисел. Эта реализация должна представлять собой класс на языке C++, удовлетворяющий следующим условиям.

   1.1. Дубликаты допускаются.

   1.2. Дубликаты запрещаются.

2. Выполните задание 1, пользуясь рекурсией. Напомним, что рекурсивные функции должны быть закрытыми членами класса.

3. Напишите реализацию абстрактного списка, используя динамический массив.

4. Пользуясь указателями, напишите реализацию списка, открытого с двух сторон, в котором вставка и удаление происходят с обоих концов.

   4.1. Не пользуйтесь указателем на хвост списка.

   4.2. Воспользуйтесь указателем на хвост списка.

5. Реализуйте абстрактный список символов, образующих строку, описанный в упражнении 6 главы 3. Представьте его в виде связанного списка. Как сделать так, чтобы длину строки можно было вычислить, не выполняя обхода списка и не подсчитывая его элементы?

6. Усовершенствуйте абстрактный список символов (задание 5), предусмотрев более сложные операции над строками. Например, найдите индекс самого левого вхождения заданного символа в строку; определите, является ли одна строка подстрокой другой строки.

7. Проанализируйте разреженную реализацию абстрактного полинома, в которой хранятся лишь коэффициенты при ненулевых элементах. Например, представьте полином *p* из упражнения 8 главы 3 в виде связанного списка, изображенного на рис. 4.36.

   7.1. Осуществите разреженную реализацию.

   7.2. Определите операцию обхода разреженного полинома, позволяющую складывать два разреженных полинома, игнорируя члены с нулевыми коэффициентами.

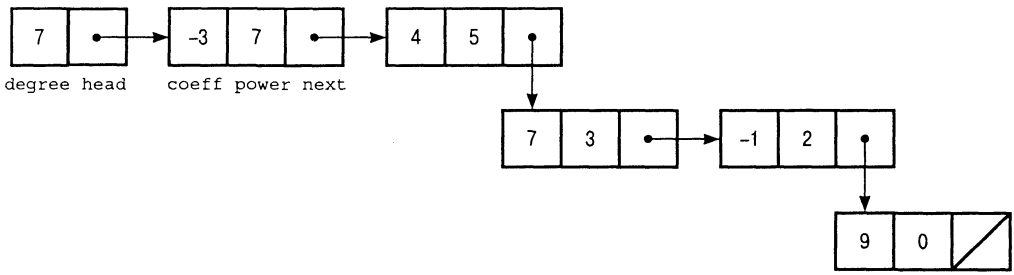


Рис. 4.36. Разреженный полином

8. Играя в настольные игры или пользуясь совместными компьютерными ресурсами, вы становитесь в очередь и ждете, когда наступит ваш черед. Количество игроков, принимающих участие в игре, практически постоянно, в то же время количество пользователей компьютерных ресурсов часто изменяется. Будем предполагать, что эти изменения носят постоянный характер.

   Разработайте абстрактный тип данных, предназначенный для отслеживания очередей. Предусмотрите операции вставки и удаления элементов очереди, а также способ определения, чья очередь наступила в данный момент.

   Допустим задана определенная группа людей. Присвойте этим людям первоначальные номера (в произвольном или заданном порядке). Новый человек, присоединившийся к группе, должен занять место в конце очереди, позади всех. Каждый следующий вновь прибывший человек должен занять место после всех предыдущих и т.д.

   Разработайте абстрактный тип данных для представления информации об этих людях. Очередь должна состоять из экземпляров этого типа данных.

   Реализуйте указанные выше абстрактные типы данных в виде классов на языке C++. В качестве основной структуры данных используйте кольцевой связанный список. Воспользуйтесь указателями и не применяйте никаких других классов.

   Напишите программу для полного тестирования разработанного абстрактного типа данных. Программа должна выполнить несколько операций вставки и удаления, гарантируя правильную очередность.

9. Иногда бывает полезным реализовать связанные структуры, не пользуясь указателями. Одна из таких структур использует массив, элементы которого "связаны" с помощью индексов. На рис. 4.37, *a* показан массив узлов связанного списка, изображенного на рис. 4.34. Каждый узел состоит из двух членов, `item` и `next`. Член `next` представляет собой целочисленный индекс элемента массива, содержащего следующий узел связанного списка. Член `next` последнего узла равен `-1`. Целочисленная переменная `head` содержит индекс первого узла списка.

Элементы массива, которые в данный момент не являются узлами связанного списка, образуют **свободный список** (free list) доступных узлов. Эти узлы образуют другой связанный список. Индекс первого свободного узла содержится в целочисленной переменной *free*. Чтобы вставить элемент в исходный связанный список, нужно извлечь узел из начала свободного списка и вставить его в данный связанный список (рис. 4.37, *б*). Чтобы удалить элемент из связанного списка, нужно вставить его в начало свободного списка (рис. 4.37, *в*). Это позволяет не сдвигать элементы массива.

Реализуйте абстрактный список в виде массива, содержащего узлы связанного списка.

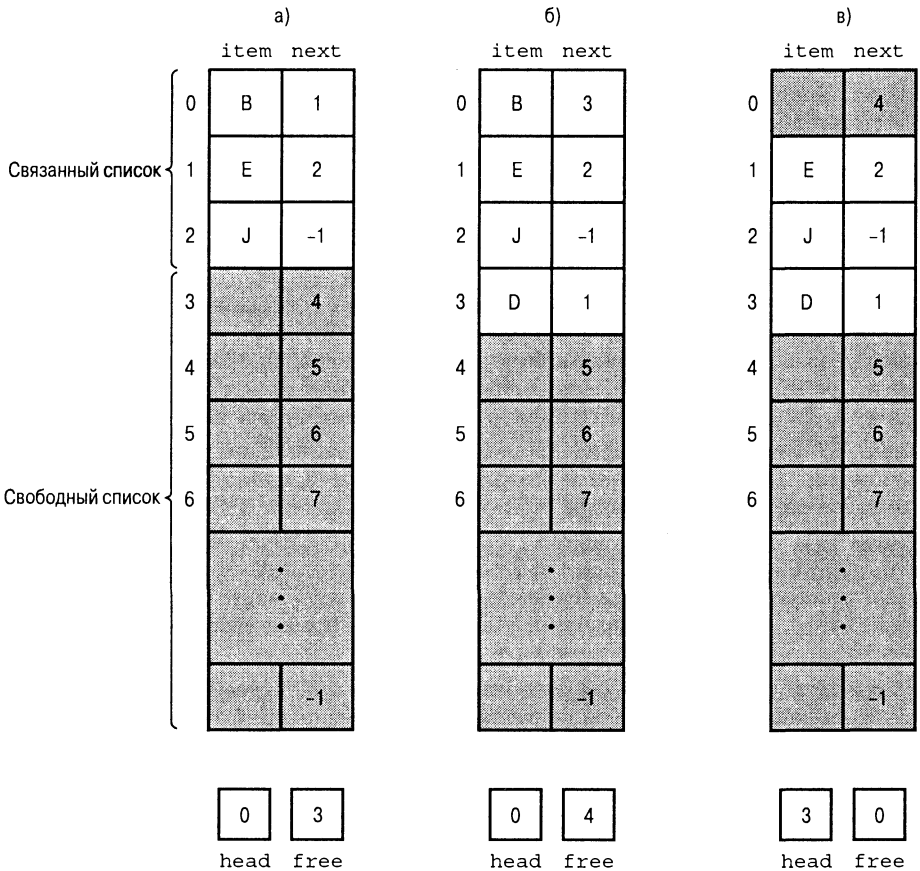


Рис. 4.37. Массив узлов связанного списка: а) реализация связанного списка, изображенного на рис. 4.34, в виде массива; б) вид упорядоченного списка после вставки буквы D; в) вид списка после удаления буквы B

10. Напишите программу, реализующую инвентарную ведомость, описанную в главе.
11. Усовершенствуйте программу для инвентаризации видеокассет, придерживаясь следующих соглашений.

11.1. Предусмотрите возможность работы с несколькими инвентарными ведомостями.

11.2. Предусмотрите возможность сохранения статистических данных о каждой кассете (например, среднее количество продаж за неделю или 10 недель).

11.3. Предусмотрите возможность изменения количества заказанных магазином кассет (например, если кассета повреждена или возвращена поставщику). Проанализируйте связь между количеством заказанных кассет и размером соответствующего списка очередности.

11.4. Усложните структуру списка очередности. Например, запишите туда имена и адреса заказчиков. Предусмотрите возможность уведомления заказчиков при поступлении нужной кассеты.

11.5. Усложните процедуру заказа. Например, не допускайте повторного заказа кассеты, которая еще не была доставлена.

Рекурсивный метод решения задач

В этой главе ...

Поиск с возвратом
- Задача о восьми ферзях

Определение языков
- Основы грамматики
- Два простых языка
- Алгебраические выражения

Связь между рекурсией и математической индукцией
- Правильность рекурсивной функции для вычисления факториала
- Количество ходов при решении задачи о ханойских башнях

Резюме
Предупреждения
Вопросы для самопроверки
Упражнения
Задания по программированию

Введение. Основные понятия, связанные с рекурсией, были рассмотрены в главе 1. Теперь мы переходим к описанию ее чрезвычайно полезных и сложных приложений в компьютерных науках. Рекурсивные решения задач намного элегантнее и лаконичнее своих итеративных аналогов.

В этой главе рассмотрены две концепции: поиск с возвратом и формальные грамматики. Поиск с возвратом — это метод решения задач, который использует последовательный перебор возможных вариантов. Формальные грамматики позволяют определять, например, синтаксически правильные алгебраические выражения. В заключение рассматривается тесная связь между рекурсией и математической индукцией. Будет показано, как использовать математическую индукцию для анализа свойств алгоритмов.

Многочисленные приложения рекурсии будут рассмотрены в дальнейших главах.

Поиск с возвратом

Поиск с возвратом — это стратегия последовательного перебора вариантов решения, предусматривающая возврат назад, если алгоритм зашел в тупик

В этом разделе рассматриваются методы систематического перебора вариантов решения задачи. Если какой-либо из вариантов заводит алгоритм в тупик, выполняется возврат назад и замена гипотезы на другую. Эта стратегия отката с последующим выполнением новой последовательности шагов называется *поиском с возвратом* (backtracking). Рассмотрим комбинацию поиска с возвратом и рекурсии при решении следующей задачи.

Задача о восьми ферзях

Разместите восемь ферзей на шахматной доске так, чтобы ни один из них не был атакован другим

Шахматная доска разбита на 64 клетки, образующие восемь горизонталей и восемь вертикалей. Наиболее сильной шахматной фигурой является ферзь, поскольку он может атаковать любую клетку горизонтали, вертикали и диагоналей, на пересечении которых он стоит. В задаче о восьми ферзях предлагается разместить их на шахматной доске так, чтобы ни один из них не был атакован другим.

Одна из стратегий решения этой задачи сводится к перебору вариантов решения. Однако существует 4426165368 способов разместить восемь ферзей на шахматной доске. Перебрать все эти варианты практически невозможно. Несмотря на это с помощью простого наблюдения можно заранее исключить из рассмотрения большое количество неприемлемых ситуаций: ни один ферзь не может находиться на горизонтали или вертикали, на которой уже расположен другой ферзь. Иначе говоря, на каждой горизонтали и вертикали может стоять только один ферзь. Таким образом, остается 8!=40320 вариантов, которые подлежат проверке. Задача становится намного проще.

Разместите ферзей по одному на каждой вертикали

Допустим, что, следуя стратегии перебора вариантов, мы разместили по одному ферзю на каждой вертикали, начиная с первой клетки первой вертикали. Рассматривая вторую вертикаль, мы обязаны исключить из нее первую клетку, поскольку на первой горизонтали уже стоит другой ферзь, а также вторую клетку, поскольку это ферзь атакует диагональ. В итоге второго ферзя следует поместить в третью клетку второй вертикали. На рис. 5.1, *а* показано расположение пяти ферзей при выполнении этой процедуры. Точки на рисунке обозначают клетки, атакованные другими ферзями.

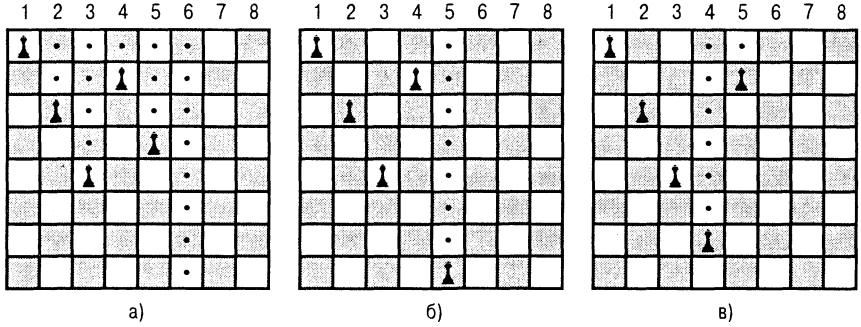


Рис. 5.1. Задача о пяти ферзях: а) пять ферзей, которые не могут атаковать друг друга, но при этом атакуют все клетки шестой вертикали; б) откат на пятую вертикаль, чтобы найти другую клетку для ферзя; в) откат на четвертую вертикаль, чтобы найти другую клетку для ферзя, а затем снова перейти к пятой вертикали

Обратите внимание, что пять ферзей, изображенных на рис. 5.1, *а*, атакуют все клетки шестой вертикали. Следовательно, на эту вертикаль невозможно поставить нового ферзя. Это вынуждает нас вернуться к пятой вертикали и переставить ферзя на другую клетку. Как показано на рис. 5.1, *б*, следующая возможная клетка на пятой вертикали расположена на последней горизонтали. Вновь проанализировав шестую вертикаль, мы убеждаемся, что поставить на нее ферзя по-прежнему невозможно. Исчерпав все возможные варианты расположения ферзя на пятой вертикали, мы должны вернуться к четвертой вертикали. Следующая свободная клетка на четвертой вертикали расположена на седьмой горизонтали, как показано на рис. 5.1, *в*. Затем нужно снова рассмотреть пятую вертикаль и поставить ферзя на вторую горизонталь.

Как применить рекурсию для решения этой задачи? Рассмотрим алгоритм, который размещает ферзя на вертикали, при условии, что на предыдущих вертикалях остальные ферзи расставлены правильно. Во-первых, если текущая вертикаль оказалась последней, задача решена. Это — базовая задача. В противном случае после размещения ферзя на текущей вертикали нужно перейти к следующей. Иными словами, нужно решить ту же самую задачу с меньшим количеством вертикалей. Это — шаг рекурсии. Таким образом, процедура решения задачи начинается с восьми вертикалей, а затем на каждом шаге рекурсии количество вертикалей уменьшается на единицу. Базовая задача возникает, когда все вертикали оказываются использованными.

На первый взгляд, это решение удовлетворяет критериям применения рекурсии. Однако мы до сих пор не знаем, как правильно поставить ферзя на текущую вертикаль. Если ферзь поставлен правильно, можно переходить к следующей вертикали. Если поставить ферзя на текущую вертикаль невозможно, нужно выполнить откат, как показано выше. Приведенный ниже псевдокод описывает алгоритм решения задачи о восьми ферзях, при условии что ферзи, расположенные на предыдущих вертикалях, не атакуют друг друга.

Решение задачи достигается с помощью комбинации поиска с возвратом и рекурсии

```
placeQueens(in currColumn:integer)
// Размещает ферзей на вертикалях,
// номера которых изменяются
// в диапазоне от currColumn до 8.
if(currColumn > 8)
   Задача решена
else
```

```
{
  while(на вертикали currColumn остались не рассмотренные
        клетки, и проблема остается нерешенной)
  {
      Определить следующую клетку на вертикали currColumn,
      не атакованную ферзями, расположенными на предыдущих
      вертикалях
      if (такая клетка существует)
      {
         Поставить ферзя на эту клетку
         placeQueens(currColumn+1) // Перейти на следующую
                                   // вертикаль
        if(на вертикали currColumn+1
           невозможно поставить ферзя)
           Снять ферзя с вертикали currColumn и рассмотреть
           следующую клетку на этой вертикали
      } // Конец оператора if
   } // Конец оператора while
} // Конец оператора if
```

Функция *placeQueens* используется в следующем контексте.

Применение функции placeQueens

```
Снять ферзей со всех клеток
placeQueens(1) // Начать с первой вертикали
if (решение найдено)
    Вывести решение на экран
else
    Вывести на экран сообщение // Решение не найдено
```

Этот контекст предполагает существование следующего класса. Для имитации шахматной доски в этом классе используется двумерный массив. Перечисление отражает состояние клетки: либо на ней стоит ферзь, либо она пуста.

```
const int BOARD_SIZE = 8; // Количество клеток на вертикали

class Queens
{
public:
   Queens(); // Создает пустую доску.
void clearBoard(); // Очищает все клетки.
void displayBoard(); // Выводит доску на экран.

bool placeQueens(int currColumn);
// ----------------------------------------------------
// Размещает ферзей на вертикалях, начиная с указанной.
// Предусловие: на предыдущих вертикалях ферзи
// расставлены правильно.
// Постусловие: если решение найдено, то на каждой вертикали
// стоит по ферзю, и функция возвращает значение true.
// В противном случае функция возвращает значение false
// (на вертикаль currColumn ферзя поставить невозможно).
// ----------------------------------------------------

private:
   enum Square {QUEEN, EMPTY}; // Состояния клетки
   Square board[BOARD_SIZE][BOARD_SIZE]; // Шахматная доска
```

```
    void setQueen(int row, int column);
  // Ставит ферзя на заданную клетку.

  void removeQueen(int row, int column);
  // Снимает ферзя с заданной клетки.

  bool isUnderAttack(int row, int column);
  // Определяет, атакована ли данная клетка ферзями,
  // расположенными на вертикалях от 1 до column-1.
  // Предусловие: на каждой из вертикалей от 1 до column-1
  // в клетке на горизонтали row стоит ферзь. Эти ферзи не
  // атакуют друг друга.
  // Постусловие: если клетка атакована, функция возвращает
  // значение true; в противном случае она возвращает
  // значение false.

  int index(int number);
  // Возвращает индекс массива, соответствующего номеру
  // горизонтали или вертикали.
  // Предусловие: 1 <= number <= BOARD_SIZE.
  // Постусловие: возвращает установленный индекс.
}; // end class
```

Реализация функции *placeQueen* имеет следующий вид.

```
bool Queens::placeQueens(int currColumn)
// Calls: isUnderAttack, setQueen, removeQueen.
{
  if  currColumn > BOARD_SIZE)
      return true; // Базовая задача
  else
  {
      bool queenPlaced = false;
      int row = 1; // Номер клетки на вертикали

      while ( !queenPlaced && (row <= BOARD_SIZE) )
      {
         // Если клетка атакована
         if (isUnderAttack(row, currColumn))
            ++row; // Рассмотреть другую клетку
                   // на вертикали currColumn,
         else      // в противном случае поставить на нее
                   // ферзя и перейти к следующей вертикали
         {
            setQueen(row, currColumn);
            queenPlaced = placeQueens(currColumn+1);
            // Если на следующую вертикаль
            // ферзя поставить невозможно
            if (!queenPlaced),
            {
               // откат: удалить ранее поставленного ферзя
               // и проверить следующую клетку на вертикали
               removeQueen(row, currColumn);
               ++row;
            } // Конец оператора if
         } // Конец оператора if
```

```
        } // Конец оператора while
        return queenPlaced;
    } // Конец оператора if
} // Конец функции placeQueens
```

Решение, обнаруженное с помощью описанного выше алгоритма, показано на рис. 5.2.

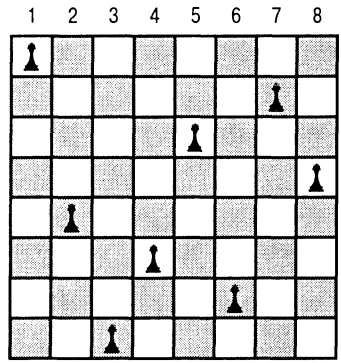


Рис. 5.2. Решение задачи о восьми ферзях

Модифицируя функцию `placeQueens`, можно получить другие решения задачи о восьми ферзях. Кроме того, описанный алгоритм можно усовершенствовать. Для простоты шахматная доска имитировалась двумерным массивом размером 8 на 8. Однако такая реализация слишком неэкономно использует память. Помимо всего прочего, из 64 клеток реально используются лишь 8. В одном из заданий по программированию, приведенных в конце главы, читателям предлагается улучшить алгоритм решения этой задачи.

Определение языков

Язык — это совокупность строк, состоящих из символов

Если вы читаете эту книгу, то знаете как минимум два языка: русский и C++. **Язык** — это не более чем совокупность строк, состоящих из символов. Например, если представить программу на языке C++ в виде длинной строки символов, можно определить множество всех синтаксически правильных программ на языке C++. Это множество и является языком C++.

ПрограммыC++ = {строки w,
где w — синтаксически правильная программа на C++}.

Хотя программы — это строки, не все строки являются программами. Компилятор языка C++ — это программа, которая, помимо всего прочего, определяет, является ли данная строка элементом языка *ПрограммыC++*. Иными словами, компилятор устанавливает, является ли заданная строка синтаксически правильной программой языка C++. Разумеется, такого описания языка *ПрограммыC++* совершенно недостаточно, чтобы создать компилятор. Это определение дает характеристику строк во множестве *ПрограммыC++*, а именно: эти строки являются синтаксически правильными программами на языке C++. Однако в этом определении нет правил, позволяющих указать, какие строки считаются синтаксически правильными программами на языке C++.

Слово "язык" не обязательно относится лишь к языкам программирования или языкам человеческого общения. Например, множество алгебраических выражений тоже образует язык.

АлгебраическиеВыражения = {строки w, где w — алгебраическое выражение}.

Язык *АлгебраическоеВыражение* представляет собой множество строк, соответствующих определенным синтаксическим правилам. Однако в определении эти правила не формулируются.

Правила образования строк в языке задаются грамматикой

В обоих рассмотренных нами примерах правила формирования строк не указывались. Эти правила задаются **грамматикой** языка. Грамматика, изучаемая в этой главе, является рекурсивной по своей природе. Одно из ее неоспоримых преимуществ заключается в том, что, основываясь на такой грамматике, можно создать рекурсивный алгоритм, позволяющий определить, принадлежит ли указанная строка данному языку. Такой алгоритм называется **алгоритмом распознавания** языка (recognition algorithm).

Поскольку грамматика языка *ПрограммыC++* весьма сложна, мы рассмотрим грамматики более простых языков, включая алгебраические выражения.

Основы грамматики

Символы, которые используются грамматикой

Грамматика использует несколько специальных символов:

- выражение $x \mid y$ означает "x или y";
- $x\ y$ означает "символ y следует за символом x"; если контекст требует дополнительных разъяснений, используется обозначение $x \bullet y$, символ $\bullet$ означает конкатенацию, или склеивание строк;
- выражение < *word* > означает любой экземпляр слова *word*, используемый в определении.

Начнем с простой грамматики языка

C++Ids = { строки w, где w — допустимый идентификатор языка C++}.

Как известно, идентификаторы в языке C++ начинаются с буквы, за которой может следовать нуль, другие буквы или цифры. В этом контексте символ подчеркивания (_) считается буквой. Приведенное выше определение можно изобразить в виде синтаксической диаграммы, показанной на рис. 5.3.

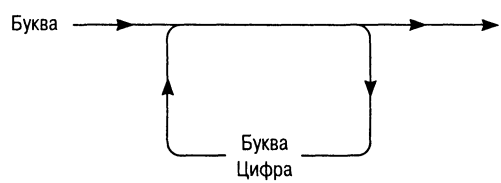


Рис. 5.3. Синтаксическая диаграмма идентификаторов в языке C++

Синтаксическая диаграмма очень наглядна, однако создание функции, распознающей идентификатор, лучше начать с грамматики. Грамматика языка *C++Ids* имеет следующий вид.

Грамматика языка C++Ids

```
< идентификатор > = < буква >|
      < идентификатор >< буква >|
      < идентификатор >< цифра >
< буква > = a |b|...|z|A|B|...|Z|_
< цифра > = 0|1|...|9
```

Это определение читается следующим образом.

Идентификатор — это буква, или идентификатор, за которым следует буква, или идентификатор, за которым следует цифра.

Многие грамматики являются рекурсивными

Обратите особое внимание, понятие *идентификатор* используется в своем собственном определении. Таким образом, эта грамматика является рекурсивной, как и многие другие.

Используя эту грамматику, можно определить, принадлежит ли строка *w* языку *C++Ids*. Применим для этого следующий алгоритм распознавания: если длина строки *w* равна 1 и состоит из буквы, она принадлежит данному языку. (Это — базис рекурсивного алгоритма.) Если длина строки *w* больше единицы, она принадлежит языку, если последним символом строки *w* является буква или цифра, а строка *w* без последнего символа является идентификатором.

Псевдокод рекурсивной функции, устанавливающей, принадлежит ли заданная строка языку *C++Ids*, приведен ниже.

Алгоритм распознавания идентификаторов языка C++

```
isId(in w:string):boolean
// Возвращает значение true, если строка w является
// идентификатором языка C++; в противном случае
// возвращает значение false.

if (длина строки w равна 1) // Базовая задача
   if (строка w является буквой)
       return true
   else
       return false
else if (последний символ строки w является буквой или цифрой)
   return isId(строка w без последнего символа) // Точка X
else
   return false
```

Результаты трассировки этой функции для строки *A2B* приведены на рис. 5.4.

Два простых языка

Рассмотрим теперь два более простых языка, их грамматики и алгоритмы распознавания.

Палиндромы. Палиндром — это строка, которая слева направо и справа налево читается одинаково. Например, строка "радар" является палиндромом. Язык палиндромов можно определить следующим образом.

Палиндромы = {строки *w*,
которые слева направо и справа налево читаются одинаково }.

Как определить язык *Палиндромы* с помощью грамматики? Нужно сформулировать правило, которое позволит определить, является ли заданная строка *w* палиндромом. В духе рекурсивных определений следует сначала выяснить, будет

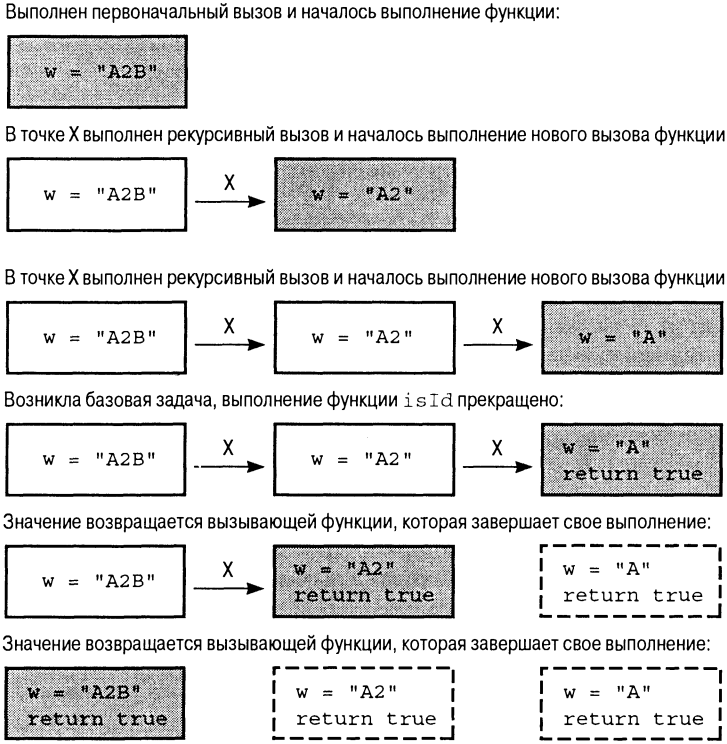


Рис. 5.4. Трассировка вызова isId("A2B")

ли палиндромом *строка меньшего размера*. На первый взгляд, в качестве меньшей строки можно выбрать строку w без последнего (или первого символа). Однако этот выбор неверен, поскольку между утверждением

строка w является палиндромом

и утверждением

строка w без последнего символа является палиндромом

нет никакой причинно-следственной связи. Иными словами, строка w может оказаться палиндромом, а строка w без последнего символа — нет, как это и происходит со словом "радар". Аналогично, строка w без последнего символа может быть палиндромом, хотя вся строка w палиндромом не является, например "радары".

Немного поразмыслив, легко понять, что символы нужно рассматривать парами, поскольку утверждения

строка w является палиндромом

и

строка w без первого и последнего символа является палиндромом

эквивалентны.

Итак, строка w является палиндромом тогда и только тогда, когда выполняются два условия:

Рекурсивное определение палиндрома

- первый и последний символы строки w совпадают;
- строка w без первого и последнего символа является палиндромом.

Отбрасывая символы с обоих концов строки, мы должны достичь базиса рекурсии. Если строка *w* состоит из четного количества символов, в итоге останется только два символа. Строка, имеющая длину 0, называется **пустой** и считается палиндромом. Если строка *w* состоит из нечетного количества символов, то в результате отбрасывания символов останется только одна буква. Следовательно, задача имеет второй базис: строка, состоящая из одной буквы, является палиндромом.

Приведенные выше рассуждения приводят нас к следующей грамматике языка палиндромов.

Грамматика языка палиндромов

```
<палиндром>=пустая строка|<символ>|а<палиндром>а| б<палиндром>б|
            ... Я<палиндром>Я
<символ>=а|б|...|я|А|Б|...|Я
```

Основываясь на этой грамматике, можно создать рекурсивную функцию для распознавания палиндромов.

Рекурсивный алгоритм распознавания палиндромов

```
isPal(in w:string):boolean
// Возвращает значение true,
// если строка w является палиндромом;
// в противном случае возвращает значение false.

   if (строка w является пустой или имеет длину, равную 1)
      return true

   else if (первый и последний символы строки w одинаковы)
      return isPal(строка w без первого и последнего символов)

   else
      return false
```

Строки вида A^nB^n. Символ A^nB^n является стандартным обозначением строки, состоящей из n символов A, за которыми следуют n символов B. Рассмотрим язык, состоящий из таких строк:

L = {строки w, имеющие вид A^nB^n, где $n \geq 0$}.

Грамматика этого языка практически совпадает с грамматикой языка палиндромов. Нужно отбрасывать символы с обоих концов строки и проверять, является ли первый символ буквой A, а последний — буквой B.

Итак, грамматика принимает следующий вид.

Грамматика языка, состоящего из строк вида A^nB^n

```
<допустимая-строка> = пустая строка|
                      A<допустимая-строка>B
```

Псевдокод функции распознавания этого языка приведен ниже.

Алгоритм распознавания языка, состоящего из строк вида A^nB^n

```
isAnBn(in w:string):boolean
// Возвращает значение true, если строка w имеет вид AnBn;
// в противном случае возвращает значение false.

   if (длина строки w равна нулю)
      return true

   else if (строка w начинается с буквы A и заканчивается буквой B)
      return isAnBn(строка w без первой и последней букв)

   else
      return false
```

Алгебраические выражения

Компилятор должен распознавать и вычислять алгебраические выражения. Рассмотрим следующее выражение на языке C++.

```
y = x + z * (w/k + z * (7 * 6));
```

Компилятор языка C++ должен определить, является ли правая часть этого оператора присваивания синтаксически правильным алгебраическим выражением. Если да, компилятор должен указать способ его вычисления.

Понятие синтаксически правильного алгебраического выражения имеет несколько определений. Некоторые из них требуют, чтобы выражение было заключено в скобки (fully parenthesized expression), т.е. чтобы каждая пара операндов была заключена в скобки вместе со своим оператором. Итак, в этом случае нужно вместо выражения $a*b*c$ писать $((a*b)*c)$. В принципе, чем строже определение, тем легче распознать синтаксически правильное выражение. Однако навязывание таких жестких ограничений создает неудобства в процессе программирования.

В этом разделе рассмотрены три разных языка алгебраических выражений. Эти выражения легко распознавать и вычислять, но неудобно использовать. Однако эти языки демонстрируют очень поучительные и нетривиальные приложения грамматик. В дальнейшем мы изучим другие языки алгебраических выражений, которые сложно распознавать и вычислять, но удобно применять в программах. Для того чтобы избежать ненужных осложнений, будем предполагать, что в языке предусмотрены лишь четыре бинарных оператора +, -, * и /, а унарных операторов и возведения в степень в нем нет. Кроме того, будем считать, что все операнды в выражениях являются идентификаторами, состоящими из одной буквы.

Инфиксные, префиксные и постфиксные выражения. Алгебраические выражения, которые изучаются в школе, являются инфиксными. Термин "инфиксный" означает что каждый бинарный оператор находится *между* своими операндами, например, выражение

$a+ b$

является инфиксным, поскольку оператор + находится между своими операндами a и b. Это соглашение вынуждает устанавливать правила ассоциативности, приоритеты, а также использовать скобки во избежание неоднозначности. Например, выражение

$a+ b*c$

неоднозначно. Что является вторым операндом оператора +? Переменная b или выражение $(b*c)$? Аналогично, первым операндом оператора * может быть переменная b либо выражение $(a+b)$. Правило, устанавливающее, что оператор * имеет более высокий приоритет, чем оператор +, устраняет эту неоднозначность. При этом переменная b является первым операндом оператора *, а выражение $(b*c)$ — вторым операндом оператора +. Если выражение $a+ b*c$ нужно интерпретировать иначе, используйте скобки.

$(a+ b)*c$

Даже правила, устанавливающие приоритеты операторов, не устраняют неоднозначности, которую создает выражение.

$a/b*c$

Обычно операторы / и * имеют одинаковый приоритет, поэтому выражение с одинаковым успехом можно интерпретировать как $(a/b)*c$ или как $a/(b*c)$.

Обычно выражение вычисляется *слева направо*, поэтому правильной считается первая интерпретация.

У традиционной инфиксной записи существуют две альтернативы: **префиксная** (prefix) и **постфиксная** (postfix) формы. В этом случае оператор указывается либо перед своими операндами (префиксная запись), либо после них (постфиксная запись). Таким образом, инфиксная запись

a+b

в префиксной форме выглядит как

+ab

а в постфиксной — как

ab+

В качестве дальнейшей иллюстрации этих соглашений рассмотрим две интерпретации инфиксного выражения *a+b*c*, приведенного выше. В префиксной форме выражение

В префиксной форме записи операнд предшествует своим операндам, а в постфиксной - наоборот

*a+(b*c)*

принимает вид

*+a *bc*

Оператор + предшествует своим операндам *a* и (**bc*), причем оператор * также предшествует своим операндам *b* и *c*. Это же выражение в постфиксной форме записи выглядит так.

abc+*

Оператор * стоит после своих операндов *b* и *c*, а оператор + — после своих операндов *a* и (*bc**).

Аналогично, выражение

*(a+b)*c*

в префиксной форме имеет вид

**+abc*

Здесь оператор * стоит перед своими операндами (*+ab*) и *c*, а оператор + — перед операндами *a* и *b*. В постфиксной форме это выражение выглядит следующим образом.

*ab+c**

Здесь оператор + стоит перед своими операндами *a* и *b*, а оператор * — перед операндами (*ab+*) и *c*.

Если инфиксная запись полностью заключена в скобки, ее легко преобразовать в префиксную или постфиксную форму. Поскольку каждый оператор с соответствующими операторами заключен внутри пары скобок, можно просто переместить оператор на позицию открывающей скобки (чтобы получить префиксную форму записи) или на позицию закрывающей скобки (чтобы получить постфиксную форму). Открывающая и закрывающая скобки находятся, соответственно, перед и после операндов. После этого все скобки из выражения следует убрать.

Рассмотрим инфиксное выражение, полностью заключенное в скобки.

*((a+b)*c)*

Чтобы преобразовать его в префиксную форму, сначала переместим каждый оператор на позицию соответствующей открывающей скобки.

Преобразование в префиксную форму

((*ab*)*c*)

*+

Теперь удалим скобки из полученного выражения.

*+*abc*

Преобразование в постфиксную форму

Аналогичным образом можно преобразовать инфиксное выражение в постфиксную форму, переместив каждый оператор на позицию соответствующей закрывающей скобки.

(*ab*)*c*)

*+

Теперь удалим скобки из полученного выражения.

*ab+c**

Если инфиксное выражение не полностью заключено в скобки, преобразования становятся более сложными. Общая схема преобразования инфиксного выражения в постфиксную форму обсуждается в главе 6.

Для префиксных и постфиксных выражений не нужны приоритеты, правила ассоциативности и скобки

Основным преимуществом префиксной и постфиксной форм записи алгебраических выражений является то, что им не нужны приоритеты, правила ассоциативности и скобки. Следовательно, грамматика префиксных и постфиксных выражений, а также алгоритмы их распознавания и вычисления более просты.

Префиксные выражения. Грамматика, определяющая язык префиксных выражений, имеет следующий вид.

< *префикс* > = < *идентификатор* > | < *оператор* >< *префикс* >< *префикс* >
< *оператор* > = + | - | * | /
< *идентификатор* > = A | B | . . . | Z

На основе этой грамматики можно создать алгоритм распознавания префиксных выражений. Если длина строки равна 1, она является префиксным выражением тогда и только тогда, когда строка содержит единственную прописную букву. Если длина строки больше единицы, то для того чтобы быть префиксным выражением, она должна иметь следующий вид.

< *оператор* >< *префикс* >< *префикс* >

Таким образом, алгоритм должен проверять следующие условия.

- Первый символ строки является оператором.
- Остальная часть строки состоит из последовательных префиксных выражений.

Первая задача тривиальна, а вот вторая довольно сложна. Как распознать последовательные префиксные выражения? Ключом к ответу является следующий факт: если к конце префиксного выражения приписать *любую* непустую строку, не состоящую из пробелов, оно перестанет быть префиксным. Иными словами, если E — префиксное выражение, а Y — произвольная непустая строка, не состоящая из пробелов, то строка EY не будет префиксным выражением. Это очень тонкий момент. Доказать этот факт предлагается в упражнении 14.

Если E — префиксное выражение, то EY не может им быть

Отталкиваясь от этого факта, можно построить алгоритм распознавания последовательных префиксных выражений. Сначала поищем первое префиксное выражение. Если оно найдено, то его конец обнаруживается автоматически.

Если конец префиксного выражения обнаружен в точке *end1*, то искать второе префиксное выражение нужно, начиная с точки *end1+1*. Если второе выражение найдено, следует выполнить проверку, не находимся ли мы в конце исходной строки.

Используя эту идею, можно показать, что выражение *+/ab-cd* является префиксным. Для этого выражение должно иметь вид $+E_1E_2$, где E_1 и E_2 — постфиксные выражения. Теперь можно записать следующие выражения:

$E_1 = /E_3E_4$, где
$E_3 = a$
$E_4 = b$

Поскольку выражения E_3 и E_4 являются префиксными, выражение E_1 является префиксным. Аналогично можно записать

$E_2 = -E_5E_6$, где
$E_5 = c$
$E_6 = d$

Это доказывает, что выражение E_2 также является префиксным.

Если в классе для префиксных выражений предусмотреть закрытый член *strExp*, представляющий собой строку, можно написать функцию, определяющую, является ли заданное выражение префиксным. Сначала создадим рекурсивную функцию *endPre(first)*, возвращающую индекс конца префиксного выражения, которое начинается с позиции *first* в строке *strExp*. Если такого префиксного выражения нет, функция *endPre* возвращает число –1. Ниже приведен псевдокод этой функции.

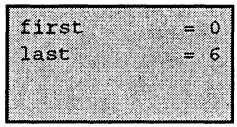
```
+endPre(in first: integer)
// Находит конец префиксного выражения, если оно существует.
// Предусловие: подстрока строки strExp, начинающаяся с индекса
// first и заканчивающаяся в конце строки, не содержит пробелов.
// Постусловие: возвращает индекс последнего символа
// префиксного выражения, которое начинается с индекса first
// строки strExp. Если такого префиксного выражения не
// существует, функция endPre должна возвращать число -1.
   last = strExp.length - 1
   if (first < 0 или first > last)
      return -1
   ch = символ, стоящий в позиции first строки strExp
   if (ch является идентификатором)
      // Индекс последнего символа в простом
      // префиксном выражении
   return first
   else if (ch является оператором)
   {
      // Найти конец первого префиксного выражения
      firstEnd = endPre(first+1) // Точка X

      // Если конец первого префиксного выражения найден,
      // найти конец второго префиксного выражения
      if (firstEnd > 0)
          return endPre(firstEnd+1) // Точка Y
      else
          return -1
   } // Конец оператора if
else
   return -1
```

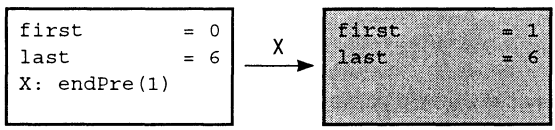
На рис. 5.5 показаны результаты трассировки функции *endPre* на примере выражения *+/ab-cd*.

Функцию *endPre* можно применить для определения, является ли строка *strExp* префиксным выражением.

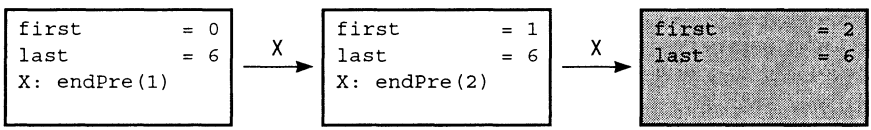
Выполнен первоначальный вызов и начинается выполнение функции `endPre`:



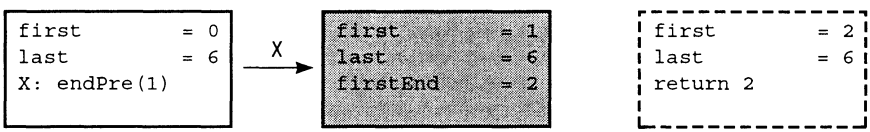
Первым символом строки является оператор +, поэтому в точке X выполнен рекурсивный вызов и начинается новое выполнение функции `endPre`:



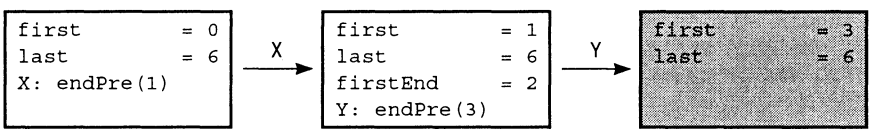
Следующим символом строки является оператор /, поэтому в точке X выполнен рекурсивный вызов и начинается новое выполнение функции `endPre`:



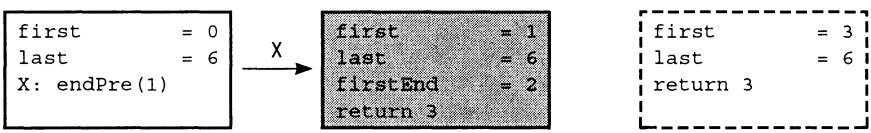
Следующим символом строки является буква `a`. Это — базовая задача. Текущее выполнение функции `endPre` прекращается и возвращается его результат:



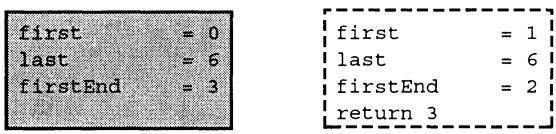
Поскольку `firstEnd>-1`, выполняется рекурсивный вызов из точки Y и начинается новое выполнение функции `endPre`:



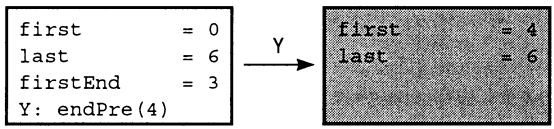
Следующим символом строки является буква `b`. Это — базовая задача. Текущее выполнение функции `endPre` прекращается и возвращается его результат:



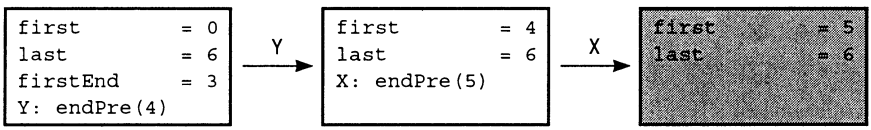
Текущее выполнение функции `endPre` прекращается и возвращается его результат:



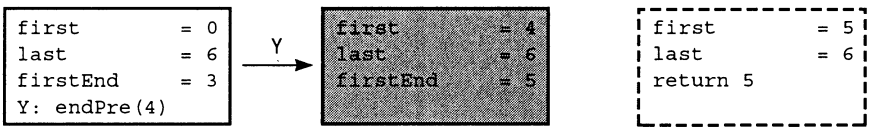
Поскольку `firstEnd>-1`, выполняется рекурсивный вызов из точки Y и начинается новое выполнение функции `endPre`:



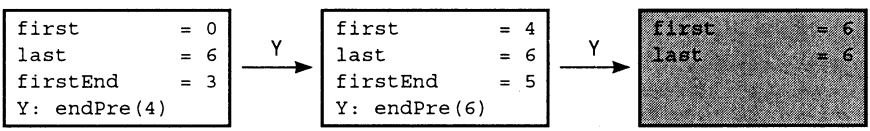
Следующим символом строки является оператор -, поэтому в точке X выполнен рекурсивный вызов и начинается новое выполнение функции `endPre`:



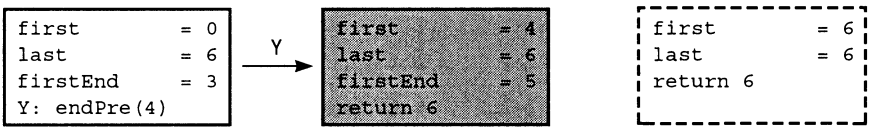
Следующим символом строки является буква `c`. Это — базовая задача. Текущее выполнение функции `endPre` прекращается и возвращается его результат:



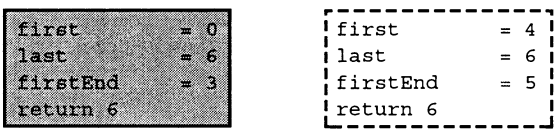
Поскольку `firstEnd>-1`, выполняется рекурсивный вызов из точки Y и начинается новое выполнение функции `endPre`:



Следующим символом строки является буква `d`. Это — базовая задача. Текущее выполнение функции `endPre` прекращается и возвращается его результат:



Текущее выполнение функции `endPre` прекращается и возвращается его результат:



Текущее выполнение функции `endPre` прекращается, а его результат возвращается исходному вызову функции `endPre`:

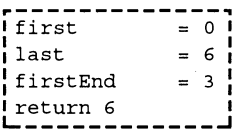


Рис. 5.5. Трассировка вызова endPre(0), где строка strExp представляет собой выражение +/ab-cd

Алгоритм распознавания префиксных выражений

```
// Определяет, является ли выражение
// префиксным.
// Предусловие: в классе есть член
// strExp, содержащий строку без пробелов.
// Постусловие: возвращает значение true, если выражение
// является префиксным; в противном случае возвращает
// значение false.

lastChar = endPre(0)

return (lastChar >= 0 and lastChar == strExp.length()-1)
```

Итак, мы можем распознать префиксное выражение. А как его вычислить? Поскольку каждый операнд сопровождается двумя операндами, их значения нужно знать заранее. Однако эти операнды сами могут оказаться префиксными выражениями, которые нужно вычислить. Эти префиксные выражения являются подвыражениями исходного и, следовательно, имеют меньший размер. Следовательно, естественно применить рекурсию.

Ниже приведен псевдокод функции, вычисляющей префиксное выражение.

```
evaluatePrefix(in strExp:string):float
// Вычисляет префиксное выражение strExp.
// Предусловие: аргумент strExp — это строка, содержащая
// правильное префиксное выражение без пробелов.
// Постусловие: возвращает значение префиксного выражения.

ch = первый символ строки strExp
Удалить первый символ строки strExp
if (ch является идентификатором)
   // Базовая задача — отдельный идентификатор
   return значение идентификатора
else if (ch является оператором op)
{
   operand1 = evaluatePrefix(strExp)
   operand2 = evaluatePrefix(strExp)
   return operand1 op operand2
} // Конец оператора if
```

Обратите внимание, что рекурсивный вызов функции *evaluatePrefix* удаляет из строки *strExp* префиксное выражение, вычисленное позже всех. Для реализации этой функции на языке C++ нужно, чтобы выражение *strExp* передавалось по значению. Тогда изменения, происходящие с копией этой строки внутри функции, не отразятся на оригинале.

Постфиксные выражения. Грамматика, определяющая язык постфиксных выражений имеет следующий вид.

< *постфикс* > = < *идентификатор* > | < *оператор* >< *постфикс* >< *постфикс* >
< *оператор* > = + | - | * | /
< *идентификатор* > = A | B | . . . | Z

В некоторых калькуляторах для вычисления операции сначала нужно ввести операнды. Очевидно, что в таких калькуляторах используются постфиксные выражения.

В этом разделе мы разработаем алгоритм преобразования префиксного выражения в постфиксное. Алгоритм итеративного вычисления постфиксных выражений будет рассмотрен в главе 6. Оба этих алгоритма дают возможность вы-

числять префиксные выражения. Для простоты будем считать, что мы работаем с правильными префиксными выражениями.

Преобразование префиксного выражения в постфиксное достаточно очевидно. Если префиксное выражение *exp* состоит лишь из одной буквы, то

postfix(*exp*) = *exp*

В противном случае выражение *exp* должно иметь следующий вид.

<*оператор*><*prefix1*>< *prefix2*>

Соответствующее постфиксное выражение выглядит следующим образом:

<*postfix1*>< *postfix2*><*оператор*>

где выражение <*prefix1*> преобразуется в выражение <*postfix1*>, а < *prefix2*> — в <*postfix2*> . Следовательно,

postfix(*exp*) = *postfix*(*prefix1*)+ *postfix*(*prefix2*)+<*оператор*>

В общем виде алгоритм преобразования можно записать так.

Алгоритм преобразования префиксного выражения в постфиксное

```
if (exp является отдельной буквой)
   return exp
else
   return postfix(prefix1)+ postfix(prefix2)+<operator>
```

Этот алгоритм уточнен в псевдокоде функции *convert*. Строка *pre* содержит префиксное выражение, а строка *post* — постфиксное. При каждом рекурсивном вызове функции *convert* длина строки *pre* уменьшается на единицу, поскольку из строки удаляется первый символ, а строка, полученная в результате этой операции, передается очередному рекурсивному вызову функции *convert*. Как и в функции *evaluatePrefix*, строка *pre* должна передаваться по значению. В исходном положении строка *post* должна быть пустой.

Рекурсивный алгоритм преобразования префиксного выражения в постфиксное

```
convert(in pre:string, out post:string)
// Предусловие: строка pre является
// правильным префиксным выражением.
// Постусловие: строка post является
// эквивалентным постфиксным выражением.

   // Проверка первого символа заданного выражения
   ch = первый символ строки pre
   Удалить первый символ строки pre

   if (ch является строчной буквой) // Проверка символа
   // Базовая задача — отдельный идентификатор
   post = post+ch // приписать к строке post

   else // ch является оператором
   {

       // Выполнить рекурсивное преобразование
       convert(pre, post) // Первое префиксное выражение
       convert(pre, post) // Второе префиксное выражение
       post = post+ch // Оператор конкатенации
   } // Конец оператора if
```

Выражения, полностью заключенные в скобки. Большинство программистов стремятся избежать префиксных и постфиксных алгебраических выражений, поэтому в большинстве языков программирования принята инфиксная форма записи. Однако для устранения неоднозначности инфиксная запись вынуждает вводить приоритеты операторов, правила ассоциативности и скобки.

Грамматика языка алгебраических выражений, полностью заключенных в скобки

Приоритеты и правила ассоциативности становятся ненужными, если каждый оператор вместе со своими операндами заключается в скобки. Грамматика, определяющая язык алгебраических выражений, полностью заключенных в скобки, имеет следующий вид.

< *инфикс* > = < *идентификатор* > | < *инфикс* >< *оператор* >< *инфикс* >
< *оператор* > = + | - | * | /
< *идентификатор* > = A | B | . . . | Z

Несмотря на простоту этой грамматики, на таком языке сложно программировать. По этой причине в большинстве языков программирования для алгебраических выражений вводятся приоритеты операторов и правила ассоциативности, так что заключать выражения в скобки не обязательно. Это приводит к усложнению их грамматик, а также алгоритмов распознавания и вычисления выражений. В задании 7, приведенном в конце главы, читателям предлагается описать грамматику, не содержащую правил ассоциативности слева направо, и написать соответствующий алгоритм распознавания. Итеративные алгоритмы вычисления алгебраических выражений, использующие приоритеты операторов и правила ассоциативности слева направо, описаны в главе 6.

Связь между рекурсией и математической индукцией

Рекурсия и математическая индукция тесно связаны между собой. Рекурсия позволяет решать задачи, постепенно понижая их размерность и сводя их к базовым задачам, которые решаются очевидным образом. Аналогично, математическая индукция позволяет доказывать свойства натуральных чисел, исходя из базового варианта, обычно относящегося к числам 0 или 1. Предполагается, что доказываемое свойство выполняется для произвольного натурального числа n, если оно справедливо для всех чисел, которые меньше n.

Для доказательства, что рекурсивный алгоритм правилен или выполняет определенный объем работы, можно использовать математическую индукцию

Исходя их этого, становится понятным, почему индукция часто применяется для доказательства свойств рекурсивных алгоритмов. Каких именно свойств? Например, можно доказать, что алгоритм действительно решает задачу, для которой он предназначен. В качестве иллюстрации мы докажем, что рекурсивный алгоритм вычисления факториала, описанный в главе 2, действительно вычисляет факториал своего аргумента. Кроме того, с помощью математической индукции можно доказать, что алгоритм выполняет определенный объем работы. Например, мы докажем, что решение задачи о ханойских башнях — также из главы 2 — достигается ровно за 2^N-1 ходов, где N — количество колец.

Правильность рекурсивной функции для вычисления факториала

Псевдокод, приведенный ниже, описывает функцию, вычисляющую факториал неотрицательного числа *n*.

```
fact(in n:integer):integer
   if (n is 0)
      return 1
   else
      return n *fact(n - 1)
```

Докажем, что функция `fact` возвращает следующие значения.

```
fact(0) = 0! = 1

fact(n) = n! = n * (n - 1) * (n - 2) * · · · * 1, если n > 0
```

Доказательство проводится по индукции.

Базис. *Докажем, что это свойство выполняется для n = 0.* Иными словами, необходимо показать, что вызов `fact(0)` возвращает число 1. Однако этот результат просто является базисом функции: вызов `fact(0)` возвращает число 1 по определению.

Теперь нам нужно доказать следующее утверждение.

> *Если свойство выполняется для произвольного k,*
> *то оно выполняется и для k+1.*

Индуктивное предположение. *Допустим, что свойство выполняется для n=k.* Иными словами, предположим, что

```
fact(k) = k! = k * (k - 1) * (k - 2) * · · · * 1
```

Индуктивное заключение. *Покажем, что свойство выполняется для n=k+1.* Иначе говоря, нужно показать, что

```
fact(k+1) = (k+1)! = (k + 1) * k * (k - 1) * (k - 2)· · · * 1
```

По определению функции `fact`, вызов `fact(k+1)` возвращает значение

```
(k+1)*fact(k)
```

По индуктивному предположению, вызов `fact(k)` возвращает значение

```
k * (k - 1) * (k - 2) * · · · * 1
```

Таким образом, вызов `fact(k+1)` возвращает значение

```
(k + 1) * k * (k - 1) * (k - 2)· · · * 1,
```

что и требовалось доказать. Итак, справедливо следующее выражение.

> *Если свойство выполняется для произвольного k,*
> *то оно выполняется и для k+1.*

Индуктивное доказательство закончено.

Количество ходов при решении задачи о ханойских башнях

В главе 2 было получено следующее решение задачи о ханойских башнях.

```
solveTowers(count, source, destination, spare)

   if (значение count равно 1)
      Переместить диск со стержня source
      на стержень destination
   else
   {
      solveTowers(count-1, source, spare, destination)
      solveTowers(1, source, destination, spare)
      solveTowers(count-1, spare, destination, source)
   } // Конец оператора if
```

Поставим следующий вопрос: если в начальный момент времени на стержне находилось N колец, сколько ходов понадобится, чтобы решить задачу?

Пусть величина *moves(N)* равна количеству ходов, необходимых для решения задач с N кольцами. Если $N = 1$, решение очевидно.

```
moves(1)=1
```

Если $N > 1$, то значение *moves(N)* уже не так легко вычислить. Однако проверка алгоритма *solveTowers* показывает, что в нем содержится три рекурсивных вызова. Следовательно, если бы мы знали, сколько ходов занимает решение задачи с *N-1* кольцами, то нашли бы ответ на исходный вопрос. Количество ходов выражается следующей формулой.

```
moves(N) = moves(N - 1) + moves(1) + moves(N - 1)
```

Таким образом, мы получаем рекуррентное соотношение между количеством ходов, необходимых для решения задачи с N кольцами.

Рекуррентное соотношение для количества ходов, необходимых для решения задачи *solveTowers* с N кольцами

```
moves(1) = 1
moves(N) = 2 * moves(N - 1) + 1, если  N > 1
```

Например,

```
moves(3) = 2 * moves(2) + 1
= 2 * (2 * moves(1) + 1) + 1
= 2 * (2 * 1 + 1) + 1
= 7
```

Хотя это рекуррентное соотношение позволяет вычислять значение *moves(n)*, **замкнутая формула** (closed-form formula), — например, алгебраическое выражение — была бы более предпочтительной, поскольку в нее можно подставить произвольное число N и получить ответ. Однако рекуррентное соотношение оказывается полезным, поскольку на его основе можно получить замкнутую формулу. Поскольку этот прием не относится к рассматриваемой нами теме, мы просто приводим эту формулу и доказательство ее корректности.

Решение предыдущего рекуррентного соотношения имеет следующий вид.

Замкнутая формула для вычисления количества шагов при решении задачи solveTowers для N колец

```
moves(N) = 2^N - 1 для всех N ≥ 1
```

Обратите внимание, что 2^3-1 равно 7, как и величина *moves*(3).

Доказательство, что $moves(N)=2^N-1$ выполняется по индукции.

Базис. *Докажем, что свойство выполняется для* $N=1$. Поскольку $2^1-1=1$ и $moves(1)=1$, свойство выполняется.

Теперь требуется доказать, что

> *если свойство выполняется для произвольного* k,
> *то оно выполняется и для* $k+1$.

Индуктивное предположение. *Допустим, что свойство выполняется для* $N=k$. Иными словами, предположим, что

$$moves(k)=2^k-1.$$

Индуктивное заключение. *Покажем, что свойство выполняется для* $N=k+1$. Иными словами, нужно показать, что $moves(k+1)=2^{k+1}-1$.

```
moves(k + 1) = 2 * moves(k) + 1  по рекуррентному соотношению
= 2 * (2^k - 1) + 1 по индуктивному предположению
= 2^(k+1) - 1,
```

что и требовалось доказать. Итак:

> *если свойство выполняется для произвольного* k,
> *то оно выполняется и для* $k+1$.

Индуктивное доказательство закончено.

Может возникнуть ложное впечатление, что доказательство свойств — сравнительно легкое дело. Приведенные выше доказательства являются наиболее простыми среди всех возможных доказательств. Однако хорошо организованная программа намного легче поддается исследованию, чем запутанная и непонятная.

Дополнительные сведения о математической индукции приводятся в Приложении Г.

Резюме

1. Поиск с возвратом — это стратегия решения задач, основанная на рекурсии и последовательном переборе вариантов. Если конкретный вариант ведет в тупик, выполняется возврат назад, замена варианта, и все повторяется снова.
2. Грамматика — это средство, позволяющее описать язык, представляющий собой совокупность строк, состоящих из символов. Используя грамматику, можно создать алгоритм распознавания выражений. Грамматики часто бывают рекурсивными, что приводит к лаконичным описаниям языков.
3. Для иллюстрации применения грамматик мы рассмотрели несколько разных языков алгебраических выражений. Каждый из них имеет свои преимущества и недостатки. Префиксные и постфиксные выражения трудны для восприятия, однако имеют простую грамматику и устраняют неоднозначности. Однако инфиксные выражения легче использовать, но для них нужны скобки, приоритеты операторов и правила ассоциативности, позволяющие избежать неоднозначности. Следовательно, грамматики инфиксных выражений более запутанны.
4. Математическая индукция и рекурсия тесно связаны между собой. Индукцию можно применять для доказательства свойств рекурсивных алгоритмов. Например, можно доказать правильность рекурсивного алгоритма и вычислить объем работы, необходимый для его выполнения.

Предупреждения

1. Подзадачи, порождаемые во время рекурсивного решения, должны в конце концов сводиться к базовой задаче. В противном случае алгоритм может оказаться бесконечным. В решениях, использующих поиск с возвратом, такие ошибки возникают особенно часто.
2. Грамматики, как и рекурсивные алгоритмы, должны иметь тщательно подобранный базис. Нужно гарантировать, что при достаточно долгом разложении строки она примет вид одного из базисов грамматики.
3. Тонкости некоторых алгоритмов, рассмотренных в этой главе, демонстрируют, что для доказательства их корректности необходимо применять математические методы доказательства. Применение этих методов при разработке компонентов решения задачи позволяет исключить логические ошибки в программе. Одним из таких методов является математическая индукция. Другой способ использует инварианты циклов, рассмотренные в главе 1 и в дальнейших главах.

Вопросы для самопроверки

1. Рассмотрите задачу о четырех ферзях. Она формулируется точно так же, как и задача о восьми ферзях, но использует доску размером 4x4. Найдите все решения этой задачи, применяя поиск с возвратом.
2. Напишите инфиксное выражение $(a/b)*c-(d+e)*f$ в префиксном виде.
3. Напишите инфиксное выражение $(a*b-c)/d+(e-f)$ в постфиксном виде.
4. Напишите префиксное выражение $-a/b+c*def$ в инфиксном виде.
5. Является ли строка $+-/abc*+def*gh$ префиксным выражением?
6. Рассмотрите язык, содержащий следующие строки: `$`, `cc$d`, `cccc$dd`, `cccccc$ddd` и т.д. Напишите рекурсивную грамматику этого языка.

Упражнения

1. Выполните трассировку следующих рекурсивных функций.

   1.1. Функция *isPal* применяется к строке *abcdeba*.

   1.2. Функция *isAnBn* применяется к строке *AABB*.

   1.3. Функция *endPre* применяется к строке *-*/abcd*.
2. Проанализируйте язык, определяемый следующей грамматикой.

   `<S>=$|<W>|$<S>`

   `<W>=abb|a<W>bb`

   Напишите все строки, существующие в этом языке, количество символов в которых не превышает семи.
3. Напишите рекурсивную грамматику для языка, состоящего из строк. Строки могут состоять из одного или нескольких символов. Первая буква каждой строки должна быть прописной, а остальные — строчными.
4. Проанализируйте язык, состоящий из строк, которые содержат только точки и тире. Все строки этого языка содержат не менее четырех символов и начинаются либо двумя точками, либо двумя тире. Если первыми двумя

символами являются точки, то последним символом должно быть тире. Если первыми двумя символами являются тире, то последним символом должна быть точка. Напишите рекурсивную грамматику этого языка.

5. Рассмотрите язык слов, каждое из которых представляет строку, состоящую из точек и тире. Этот язык описывается следующей грамматикой:

   <*слово*> = <*точка*> | <*тире*><*слово*> | <*слово*><*точка*>

   <*точка*>=•

   <*тире*>=-

   5.1. Напишите все строки этого языка, состоящие из трех символов.

   5.2. Принадлежит ли этому языку строка ••••? Обоснуйте свой ответ.

   5.3. Напишите строку этого языка, состоящую из семи символов, в которой больше тире, чем точек. Докажите, что ваш ответ правилен.

   5.4. Напишите псевдокод рекурсивной функции `isIn(str)` для распознавания выражений. Эта функция должна возвращать значение `true`, если строка `str` принадлежит данному языку, и значение `false` — в противном случае.

6. Пусть L — язык, определение которого дано ниже.

   L={строки S, имеющие вид A^nB^{2n}, где $n>0$}

   Таким образом, строка принадлежит языку L, только если она начинается с последовательности символов A, за которой идет последовательность символов B удвоенной длины. Например, строка AABBBB принадлежит языку L, а строки ABBB, ABBABB и пустая строка — нет.

   6.1. Опишите грамматику языка L.

   6.2. Напишите рекурсивную функцию, определяющую принадлежность строки `strExp` языку L.

7. Является ли выражение $+*a-b/c++de-fg$ префиксным? Поясните свой ответ, пользуясь терминами грамматики префиксных выражений.

8. Является ли выражение $ab/c*efg*h/+d$ постфиксным? Поясните свой ответ, пользуясь терминами грамматики постфиксных выражений.

9. Проанализируйте язык, имеющий следующую грамматику.

   <*S*>=<*L*> | <*D*><*S*><*S*>

   <*L*>=A | B

   <*D*>=1 | 2

   9.1. Напишите строки этого языка, состоящие из трех символов.

   9.2. Напишите строку этого языка, состоящую из более чем трех символов.

10. Рассмотрите язык, состоящий из следующих символьных строк: буква A; буква B; буква C; буква C, за которой следует строка, принадлежащая данному языку; буква D, за которой следует строка, принадлежащая данному языку. Например, этому языку принадлежат такие строки: A, CA, CCA, DCA, B, CB, CCB, DB и DCCB.

    10.1. Напишите грамматику этого языка.

    10.2. Принадлежит ли этому языку строка CAB? Объясните свой ответ.

    10.3. Напишите рекурсивную функцию для распознавания строк, принадлежащих данному языку.

11. Проанализируйте язык, имеющий следующую грамматику.

<*слово*> = $|a<*слово*>a|b<*слово*>b|...|y<*слово*>y|z<*слово*>z

Иными словами,

L={строки $w$$reverse(w), где w — строка символов}

Обратите внимание, что этот язык очень похож на язык палиндромов. Единственное отличие заключается в том, что строка содержит внутри специальный символ.

Для этого языка можно легко адаптировать функцию распознавания палиндромов, описанную выше. Рекурсивный алгоритм, обрабатывающий строку str с обоих концов, основан на следующих фактах.

- Пустая строка не принадлежит языку.
- Строка, состоящая только из одного символа, принадлежит языку, только если этим символом является знак $.
- Более длинные строки принадлежат языку, если на их концах стоят одинаковые буквы, а внутренняя подстрока (начиная со второго символа и заканчивая предпоследним символом строки str) принадлежит языку.

Опишите рекурсивный алгоритм распознавания, прочитывающий строку слева направо посимвольно и не сохраняющий строку для будущей обработки. Напишите функцию на языке C++, реализующую этот алгоритм.

12. Проанализируйте следующую рекурсивную функцию.

```
int p(int x)
{
   if (x < 3)
      return x;
   else
      return p(x-1) * p(x-3);
} // Конец функции p
```

Обозначим через $m(x)$ количество операций умножения, выполняемых функцией $p(x)$.

12.1. Напишите рекурсивное определение величины m(x).

12.2. Докажите правильность вашего ответа методом математической индукции.

13. Проанализируйте палиндромы, состоящие только из строчных букв, например, “радар” или “топот”, но не “РадаР”, “АДА” или “101”. Обозначим через $c(n)$ количество палиндромов длины n.

13.1. Напишите рекурсивное определение величины c(n).

13.2. Докажите правильность вашего ответа методом математической индукции.

14. Докажите следующие факты для операций над отдельными буквами: если E — префиксное выражение, а Y — непустая строка, то EY не может быть префиксным выражением. (*Подсказка*: используйте индукцию по длине строки E.)

15. В главе 2 дано следующее определение величины $c(n,k)$, где n и k — неотрицательные целые числа.

$$c(n,k)=\begin{cases}1, \text{если } k=0,\\ 1, \text{если } k=n,\\ 0, \text{если } k>n,\\ c(n-1,k-1)+c(n-1,k), \text{если } 0<k<n.\end{cases}$$

Пользуясь индукцией по n, докажите, что

$$c(n,k)=\frac{n!}{(n-k)!k!}.$$

Задания по программированию

1. Напишите программу, решающую задачу о восьми ферзях.
2. Усовершенствуйте программу, решающую задачу о восьми ферзях так, чтобы она давала ответы на следующие вопросы.
   - 3.1. Сколько откатов было выполнено в ходе решения задачи? Иными словами, сколько раз программа снимала ферзей с доски?
   - 3.2. Сколько раз выполнялся вызов функции `isUnderAttack`?
   - 3.3. Сколько раз выполнялся рекурсивный вызов функции `placeQueens`?
   - 3.4. Можно ли улучшить функцию `isUnderAttack`? Например, если обнаружено, что ферзь на заданной клетке подвергается атаке, нужно ли продолжать анализ остальных ферзей?
3. Решение задачи о восьми ферзях можно начать не с первой клетки первой вертикали, а со второй. Затем можно вызвать функцию `placeQueens`, начиная со второй вертикали. Это изменение приводит к новому решению. Напишите программу, позволяющую найти все возможные решения задачи о восьми ферзях.
4. Вместо двумерного массива размером 8x8 для имитации шахматной доски можно использовать одномерный массив, которые представляет лишь клетки, на которых стоят ферзи. Пусть `col` — это массив, содержащий восемь целых чисел, так что

   `col[k]`=номер горизонтали, на которой на (k+1)-й вертикали стоит ферзь.

   Например, если `col[2]` равно 4, то ферзь стоит на пересечении четвертой горизонтали и третьей вертикали, т.е. на клетке `board[3][2]`. Таким образом, позицию ферзя можно задать числом `col[k]`, а не `board[col[k]-1][k]`.

   В этой схеме необходимо также хранить информацию, подвергается ли ферзь атаке. Поскольку на каждой вертикали может стоять только один ферзь, вертикали проверять не нужно. Для проверки горизонталей необходимо предусмотреть массив `rowAttack`, в котором элемент `rowAttack[k]` не равен нулю, если ферзь, стоящий на (k+1)-й вертикали, может быть атакован ферзем, стоящим на этой горизонтали.

   Для проверки атак по диагонали следует учесть, что диагонали могут иметь положительный или отрицательный наклон. Диагонали, имеющие положительный наклон, параллельны диагоналям, идущим из нижнего левого угла доски в правый верхний. Диагонали с отрицательным наклоном параллельны диагоналям, идущим из левого верхнего угла в правый нижний. Убедитесь, что если элемент `board[i][j]` представляет некую клетку, то величи-

на *i+j* является константой для клеток диагоналей, имеющих положительный наклон, а величина *i-j* постоянна для клеток диагоналей с отрицательным наклоном. Оказывается, что величина *i+j* изменяется от 0 до 14, а число *i-j* варьируется от −7 до +7. Таким образом, можно определить массивы *posDiagonal* и *negDiagonal*, так что

- элемент *posDiagonal[k]* равен значению *true* тогда и только тогда, когда ферзь, стоящий на (*k*+1)-й вертикали, может быть атакован ферзем, стоящим на диагонали с положительным наклоном;
- элемент *negDiagonal[k]* равен значению *true* тогда и только тогда, когда ферзь, стоящий на (*k*+1)-й вертикали, может быть атакован ферзем, стоящим на диагонали с отрицательным наклоном.

Используйте эту идею для создания программы, решающей задачу о восьми ферзях.

5. Знаете ли вы, как найти выход из лабиринта? Написав такую программу, вы больше никогда не заблудитесь!

   Допустим, что лабиринт представляет собой прямоугольный массив ячеек, некоторые из них заблокированы, имитируя стены. Лабиринт имеет один вход и один выход. Например, если символ x обозначает стену, то лабиринт можно изобразить следующим образом.

```
xxxxxxxxxxxxxxxxxx x
x x xxxx x
x xxxxx xxxxx xx x
x xxxxx xxxxxxx xx x
x x xx xx x
x xxxxxxxxxx xx x
xxxxxxxxxxxxoxxxxxxx
```

   Человек, обозначаемый на предыдущей диаграмме буквой o, сидит прямо перед входом в лабиринт. Допустим, что он может передвигаться только в четырех направлениях: на север, юг, восток и запад. На диаграмме север находится вверху, юг — внизу, восток — справа, запад — слева. Задача заключается в том, чтобы пройти весь лабиринт от входа до выхода, если это возможно. Перемещаясь по лабиринту, человек должен отмечать свой путь. На выходе из лабиринта нужно сравнить правильный путь и безуспешные попытки.

   Ячейки лабиринта имеют несколько состояний: СВОБОДНА (ячейка пуста), СТЕНА (ячейка заблокирована и представляет собой часть стены), ПУТЬ (ячейка лежит на пути к выходу) и ПРОЙДЕНА (человек уже был в этой ячейке, но зашел в тупик).

   При решении этой задачи нужно предусмотреть два абстрактных типа данных, взаимодействующих между собой. АТД "Человек" представляет координаты человека в лабиринте и содержит операции, выполняющие его перемещение по ячейкам. Человек должен перемещаться на север, юг, восток и запад на одну ячейку за один ход, иметь возможность опрашивать их состояние и помечать.

   АТД "Лабиринт" имитирует лабиринт, который представляется двумерным массивом ячеек. Горизонтали лабиринта можно пронумеровать сверху вниз, начиная с нуля, а вертикали — слева направо, также начиная с нуля. Номера горизонталей и вертикалей можно использовать для однозначного вычисления координат человека в лабиринте. Для имитации лабиринта необходима соответствующая структура данных. Нужно также задать длину и ширину лабиринта, измеренные количеством ячеек, длину стороны ячейки, а также координаты входа и выхода.

АТД "Лабиринт" должен также содержать операции для создания конкретного лабиринта, информация о котором хранится в текстовом файле; для вывода лабиринта на экран; для определения, является ли заданная ячейка частью стены и т.д.

Алгоритм поиска и сопутствующие функции не должны включаться в абстрактные типы данных. Таким образом, лабиринт и человек будут имитироваться аргументами, которые передаются этим функциям.

Текстовый файл, описывающий лабиринт, довольно прост. В качестве примера рассмотрим следующее описание.

```
20 7 ← длина и ширина лабиринта
0 18 ← координаты выхода из лабиринта
6 12 ← координаты входа в лабиринт
xxxxxxxxxxxxxxxxxx x
x x xxxx x
x xxxxx xxxxx xx x
x xxxxx xxxxxxx xx x
x x xx xx x
x xxxxxxxxxx xx x
xxxxxxxxxxxx xxxxxxx
```

Каждая строка файла соответствует горизонтали лабиринта; каждый символ в строке соответствует вертикали лабиринта. Буква x обозначает заблокированные ячейки (стены), а пробелы — свободные ячейки.

Если вы находитесь перед входом в лабиринт, то можете систематически перебирать варианты обхода лабиринта с помощью следующего алгоритма поиска. Он использует стратегию поиска с возвратом, т.е. возврат из тупика в исходную точку обхода.

1. Во-первых, проверьте, не стоите ли вы перед выходом из лабиринта. Если да, задача решена (очень простой лабиринт), если нет, переходите к выполнению п. 2.
2. Перемещайтесь по лабиринту прямо на север, вызывая функцию *goNorth* (описанную ниже).
3. Если выполнение функции *goNorth* завершено успешно, задача решена. Если нет, идите прямо на запад, вызывая функцию *goWest* (описанную ниже).
4. Если выполнение функции *goWest* завершено успешно, задача решена. Если нет, идите прямо на юг, вызывая функцию *goSouth* (описанную ниже).
5. Если выполнение функции *goSouth* завершено успешно, задача решена. Если нет, идите прямо на восток, вызывая функцию *goEast* (описанную ниже).
6. Если выполнение функции *goEast* завершено успешно, задача решена. Если нет, алгоритм все равно завершается, поскольку из лабиринта нет выхода.

Функция *goNorth* проверяет все варианты путей, начинающихся с ячейки, расположенной к северу от текущей. Если эта ячейка свободна, находится внутри лабиринта и еще не помечена, переходите в нее и сделайте отметку. (Обратите внимание, что вы движетесь с юга на север.) Проверьте, найден ли выход. Если да, задача решена. В противном случае попробуйте найти выход, двигаясь в любую сторону, кроме юга (вы оттуда пришли). Для этого вызовите функцию *goNorth*. Если она не привела вас к выходу, вызывайте функцию *goWest*. Если она тоже не помогла, вызывайте функцию

goEast. Если и это не привело вас к решению задачи, пометьте ячейку, вернитесь назад в исходное положение и начинайте снова.

Функция *goNorth* описывается следующим псевдокодом.

```
goNorth(maze, creature, success)

    if (ячейка, расположенная севернее, свободна,
        находится внутри лабиринта и не помечена)
    {
        Идти на север
        Пометить пройденную ячейку
        if (найден выход)
            success = true
        else
        {
            goNorth(maze, creature, success)
            if (!success)
            {
               goWest(maze, creature, success)
               if (!success)
               {
                  goEast(maze, creature, success)
                  if (!success)
                  {
                     Пометьте пройденную ячейку
                     Вернитесь на юг
                  } // Конец оператора if
               }  // end if
            } // Конец оператора if
        } // end if
    }
else
success = false
```

Функция *goWest* проверяет все варианты путей, начинающихся с ячейки, расположенной к западу от текущей. Если эта ячейка свободна, находится внутри лабиринта и еще не помечена, переходите в нее и сделайте отметку. (Обратите внимание, что вы движетесь с запада на восток.) Проверьте, найден ли выход. Если да, задача решена. В противном случае попробуйте найти выход, двигаясь в любую сторону, кроме юга (вы оттуда пришли). Для этого вызовите функцию *goNorth*. Если она не привела вас к выходу, вызывайте функцию *goSouth*. Если и это не привело вас к решению задачи, пометьте ячейку, вернитесь назад в исходное положение и начинайте снова.

Функции *goEast* и *goSouth* описываются аналогично.

6. Напишите программу, распознающую и вычисляющую префиксные выражения. Сначала разработайте и реализуйте класс префиксных выражений. Этот класс должен содержать методы распознавания и вычисления префиксных выражений. Алгоритмы, необходимые для реализации этих методов, обсуждались в выше.

7. Ниже приведена грамматика, позволяющая игнорировать скобки в инфиксных алгебраических выражениях, если приоритеты операторов устраняют неоднозначности. Например, выражение $a+b*c$ эквивалентно выражению $a+(b*c)$. Однако грамматика требует использования скобок, если неодно-

значность все равно возникает. Иными словами, грамматика не позволяет выполнять операторы слева направо, если они имеют одинаковый приоритет. Например, выражение *a/b*c* этой грамматикой не допускаются.

<выражение>=*<терм>*|*<терм>*+*<терм>*|*<терм>*-*<терм>*

<терм>=*<фактор>*|*<фактор>***<фактор>*|*<фактор>*/*<фактор>*

<буква>=A|B|...|Z

Алгоритм распознавания основан на рекурсивной цепочке подзадач: *найти выражение* → *найти терм* → *найти фактор*. Рекурсивный характер этой цепочки выражается в том, что задача *найти выражение* использует задачу *найти терм*, которая в свою очередь сводится к задаче *найти фактор*. Задача *найти фактор* либо оказывается базовой, либо сводится к задаче *найти выражение*, образуя рекурсивную цепочку.

Ниже приведен псевдокод алгоритма распознавания.

```
НАЙТИ ВЫРАЖЕНИЕ
// Выражением считается либо отдельный терм, либо терм,
// за которым следуют символы + или -,
// а за ними — второй терм.

   Найти терм
   if (следующим символом является + или -)
      Найти терм

НАЙТИ ТЕРМ
// Термом считается либо отдельный фактор, либо фактор,
// за которым следуют символы * или /,
// а за ними — второй фактор.

   Найти фактор
   if (следующим символом является * или /)
       Найти фактор

НАЙТИ ФАКТОР
// Фактором считается либо отдельная буква (базовая задача),
// либо выражение, заключенное в скобки.

   if (первым символом является буква)
       Задача решена
   else if (первым символом является '(')
   {
       Найти выражение, начинающееся символом,
       следующим за '('
       Найти символ ')'
   }
   else
       Фактора не существует
```

Разработайте и реализуйте класс инфиксных выражений, подчиняющихся данной грамматике. Включите в него метод распознавания допустимых инфиксных выражений.

II

Решение задач с помощью абстрактных типов данных

В части I были рассмотрены аспекты решения задач, тесно связанные с вопросами программирования. В ней был описан процесс абстракции данных, позволяющий решать задачи с помощью объектно-ориентированного подхода; введены классы языка C++, скрывающие детали реализации и повышающие модульность программ; рассмотрены связанные списки, широко используемые во всех главах книги, а также разработан рекурсивный подход к построению алгоритмов. Вторая часть книги посвящена вопросам *управления данными*, т.е. идентификации и реализации наиболее распространенных операций.

В части II мы покажем, как организовать данные, используя их позицию (как в абстрактном списке) или значение (как в абстрактном упорядоченном списке). Принципы организации данных, изложенные в этой части, применяются в приложениях самой разной природы. Например, если приложение должно определить имя человек, записанное в первой строке, данные нужно организовывать по позиции. С другой стороны, если нужно извлечь информацию о конкретном сотруднике, данные следует организовывать по значению. Во второй части мы опишем другие абстрактные типы, использующие эти два способа организации данных.

Изучение вопросов управления данными предследует три цели. Во-первых, выделить разумное множество полезных операций, т.е. идентифицировать абстрактный тип данных. Во-вторых, проверить приложения, в которых используются абстрактные типы. Третья цель — создать реализации абстрактных типов данных, т.е. разработать структуры данных и классы. Выбор конкретной реализации сильно зависит от природы операций над абстрактными типами данных и приложения, в котором она используется.

Стеки

В этой главе ...

Абстрактный стек
- Разработка абстрактных типов данных в процессе решения задачи

Простые примеры использования абстрактного стека
- Проверка баланса фигурных скобок
- Распознавание строк языка

Реализации абстрактного стека
- Реализация абстрактного стека в виде массива
- Реализация абстрактного стека в виде связанного списка
- Реализация стека в виде абстрактного списка
- Сравнение реализаций
- Класс stack из стандартной библиотеки шаблонов

Приложение: алгебраические выражения
- Вычисление постфиксных выражений
- Преобразование инфиксного выражения в постфиксное

Приложение: поиск
- Итеративное решение с помощью стеков
- Рекурсивное решение

Взаимосвязь между стеками и рекурсией

Резюме

Предупреждения

Вопросы для самопроверки

Упражнения

Задания по программированию

Введение. В главе рассматривается хорошо известный абстрактный тип данных под названием *стек*, а также его приложения и реализации. Описываются операции над стеком, основанные на принципе "последним вошел — первым вышел". Показано применение стеков для вычисления алгебраических операций и поиска пути между двумя точками. В заключение раскрывается тесная связь, существующая между стеками и рекурсией.

Абстрактный стек

Спецификация абстрактного типа данных, предназначенного для решения конкретной задачи, может возникать в самом процессе проектирования. Одним из наиболее важных абстрактных типов данных является стек.

Разработка абстрактных типов данных в процессе решения задачи

Набирая строку текста на клавиатуре, вы наверняка делаете опечатки. Если для их исправления вы используете клавишу <Backspace>, то каждое ее нажатие стирает предыдущий символ, введенный вами ранее. Последовательное нажатие этой клавиши позволяет стереть всю цепочку символов. Например, если вы набрали строку

```
abcc←ddde←←←ef←fg
```

где символ ← обозначает нажатие клавиши <Backspace>, то на экране появится строка

```
abcdef
```

Как запрограммировать ввод исходной строки и ее последующие исправления? Проектируя решение этой задачи, вы в конце концов будете вынуждены выбрать конкретный способ хранения строки. В соответствии с принципом абстракции данных, окончательное решение следует отложить до того момента, когда станет ясно, какие операции будут выполняться со строкой.

В качестве первого приближения можно принять следующий псевдокод.

Первое приближение

```
// Считывание строки с последующими
// исправлениями ошибок
while(не конец строки)
{
   Считать символ ch
   if (символ ch не '←')
      Добавить символ ch в абстрактный тип данных
   else
      Удалить из абстрактного типа данных элемент,
      добавленный последним
} // Конец оператора while
```

Две операции над абстрактным типом данных

Это решение акцентирует внимание на двух операциях над абстрактным типом данных.

- Добавление нового элемента в АТД.
- Удаление из АТД элемента, введенного последним.

В этом алгоритме скрыта потенциальная опасность, возникающая, если нажать клавишу <Backspace>, когда в абстрактном типе данных нет символов. В

этом случае у разработчиков есть две возможности: 1) прекратить выполнение программы и выдать сообщение об ошибке; 2) проигнорировать символ ← и продолжить выполнение программы. Каждая из этих возможностей вполне разумна. Попробуем разобраться, что произойдет, если мы примем решение игнорировать символ ← и продолжим выполнение программы. В этом случае алгоритм примет следующий вид.

Алгоритм "считывания и исправлений"

```
// Считывание строки с последующими
// исправлениями ошибок
while(не конец строки)
{
   Считать символ ch
   if (символ ch не ←')
      Добавить символ ch в абстрактный тип данных
   else if (АТД не пуст)
      Удалить из абстрактного типа данных элемент,
      добавленный последним
   else
      Игнорировать символ '←'
} // Конец оператора while
```

Итак, анализ этого псевдокода показывает, что для абстрактного типа данных нужно предусмотреть третью операцию.

Новая операция над АТД

- Определить, пуст ли АТД.

Это позволяет поместить в АТД правильно введенную строку. Допустим теперь, что нам нужно вывести строку на экран. На первый взгляд, для этого достаточно уже рассмотренных операций.

Ошибочный псевдокод

```
// Вывод строки на экран
while (АТД не пуст){
   Удалить из абстрактного типа данных элемент,
      добавленный последним
   Вывести ... Увы и ах!
```

Этот псевдокод неверен по двум причинам.

Причины, по которым этот псевдокод неверен

1. Когда элемент удаляется из АТД, он становится недоступным, и его невозможно вывести на экран. Его нужно не удалять, а *извлекать* из АТД. Операция извлечения, описанная в главе 3, означает "*просмотреть, но не изменять*". Удалять элемент можно только тогда, когда он будет извлечен из АТД и выведен на экран.
2. Элемент, введенный последним, представляет собой последний символ строки ввода. Очевидно, что его не следует выводить первым. Решение это проблемы читатели могут найти самостоятельно.

Принимая во внимание лишь первую проблему из указанных выше, можно получить псевдокод, выводящий на экран строку, записанную в обратном порядке.

Алгоритм вывода строки, записанной в обратном порядке

```
// Вывод строки, записанной в обратном
// порядке
while (АТД не пуст)
{
   Извлечь из абстрактного типа данных элемент,
      добавленный последним, и поместить его в переменную ch
```

```
    Вывести переменную ch на экран
    Удалить из абстрактного типа данных элемент,
       добавленный последним
} // Конец оператора while
```

Таким образом, в абстрактном типе данных нужно предусмотреть четвертую операцию.

Еще одна операция над АТД

- Извлечь из абстрактного типа данных элемент, добавленный последним.

Итак, размышляя о реализации АТД, нужно иметь в виду именно эти четыре операции[1]. Они определяют абстрактный тип данных, известный под названием **стек** (stack). Как мы видели в главе 3, программист может сам решать, включать ли в абстрактный тип данных операции его инициализации и уничтожения. Итак, перечислим операции над абстрактным стеком.

ОСНОВНЫЕ ПОНЯТИЯ

Операции над абстрактным стеком

1. Создание пустого стека.
2. Уничтожение стека.
3. Определение, пуст ли стек.
4. Добавление в стек нового элемента.
5. Удаление из стека элемента, добавленного последним.
6. Извлечение из стека элемента, добавленного последним.

Стеки довольно часто встречаются в обыденной жизни. В качестве примера можно привести стек тарелок в студенческой столовой, стек книг на полке или стек заданий, ждущих своего выполнения. Возникает иллюзия, что стек — это синоним слов "стопка", "кипа", "штабель". Однако, с научной точки зрения, это совершенно неверно. Стек обладает одной особенностью — элемент, помещенный в стек последним, удаляется из него первым. Это свойство часто называют принципом "**последним вошел — первым вышел**", или просто **LIFO** ("last in, first out").

Стопка тарелок в студенческой столовой является прекрасным примером стека. Он изображен на рис. 6.1. Как только в стек добавляется новая тарелка, тарелки, находившиеся в нем до этого, проталкиваются вниз. В каждый момент времени на поверхности стека находится лишь одна тарелка — та, которая была помещена в него последней. Эта тарелка называется **вершиной** стека (top) . Именно эта тарелка будет удалена из него первой. Итак, тарелки удаляются из стека в обратном порядке.

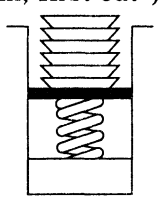


Рис. 6.1. Стек тарелок в столовой

Принцип LIFO кажется в корне несправедливым. Представьте себе беднягу, который получил последнюю тарелку из стека. Возможно, последний раз ее мыли шесть лет назад. А что вы скажете, если вам предложат стать первым в стеке ожидания — в противоположность обычной очереди. Если речь идет о кинотеатре, вы попадете туда последним! По этим причинам стеки не очень распространены в обыденной жизни. Более справедливым является принцип "первым вошел — первым вышел", или **FIFO** ("first in, first out") . Этот принцип лежит в основе **очереди** (queue) — абстрактного типа данных, который будет рассмотрен в следующей главе. Большинство людей предпочитают

[1] Выполнив упражнение 5, приведенное в конце главы, вы сможете убедиться, что для вывода строки в прямом порядке новых операций не потребуется.

стоять в *очереди*, а не в *стеке*. Несмотря на то что стеки для обычной жизни не очень подходят, в большинстве задач, возникающих в области компьютерных наук, они незаменимы.

Обратите внимание, насколько удачной оказалась аналогия между абстрактным стеком и стопкой тарелок. Операции над абстрактным стеком полностью совпадают с операциями над тарелками в стопке. Можно узнать, пуста ли стопка, однако невозможно подсчитать, сколько тарелок в ней осталось. В стопке доступна лишь верхняя тарелка, остальные скрыты от постороннего взгляда. Тарелку можно заменять только на вершине стопки, остальные тарелки недоступны. Удалять тарелку также можно лишь сверху. Если выполнить эти операции по каким-либо причинам невозможно, или существуют какие-то дополнительные операции, то соответствующий абстрактный тип данных окажется чем угодно, только не стеком.

Несмотря на то что аналогия со стопкой тарелок оказалась весьма наглядной, не стоит полностью переносить ее на абстрактный стек. Когда вы снимаете тарелку с вершины стопки или ставите туда новую, остальные тарелки перемещаются. В абстрактном стеке все операции сосредоточены исключительно на вершине и требуют лишь, чтобы остальные элементы в стеке были расположены последовательно. Реализации операций над абстрактным стеком могут перемещать элементы стека, а могут этого и не делать. Например, в реализациях стека, рассмотренных в этой главе, его элементы остаются на месте.

Уточнение определения стека. Прежде чем перейти к детальному анализу операций над стеком, рассмотрим повнимательнее операции удаления и извлечения элементов стека. Определение стека, данное выше, позволяет удалять вершину стека без предварительной проверки и проверять вершину без ее удаления. Обе эти операции вполне разумны и часто встречаются на практике. Однако если мы захотим проверить *и* удалить вершину стека — вполне обычная задача, — нам придется выполнять ряд операций.

- Извлечь из стека элемент, вставленный в него последним.
- Удалить из стека элемент, вставленный в него последним.

Эти операции можно объединить в одну.

Приведенный ниже псевдокод более детально описывает операции над абстрактным стеком, включая объединенную операцию извлечения и удаления. Имена операций добавления и удаления вершины стека являются общепринятыми. На рис. 6.2 показана UML-диаграмма класса `Stack`.

В главе 1 подчеркивалось, что спецификация модуля должна предшествовать его реализации. Записав спецификацию АТД в виде псевдокода, следует использовать ее для проверки своего проекта. Такая проверка выявляет все недостатки,

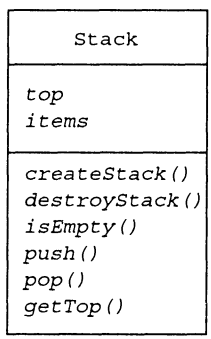


Рис. 6.2. UML-диаграмма класса Stack

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операций над абстрактным стеком

```
+createStack()
// Создает пустой стек.

+destroyStack()
// Уничтожает стек.

+isEmpty():boolean {query}
// Определяет, пуст ли стек.

+push(in newItem:StackItemType) throw StackException
// Добавляет элемент newItem на вершину стека.
// Если вставку выполнить невозможно,
// генерирует исключительную ситуацию StackException,

+pop() throw StackException
// Удаляет вершину стека; иными словами, удаляет элемент,
// вставленный последним. Если удаление выполнить невозможно,
// генерирует исключительную ситуацию StackException.

+pop(out stackTop:StackItemType) throw StackException
// Извлекает вершину стека в переменную stackTop, а затем
// удаляет ее. Таким образом, извлекается и удаляется элемент,
// вставленный в стек последним. Если удаление оказалось
// невозможным, генерируется исключительная ситуация StackException.
```

присущие либо спецификации, либо самому проекту. Например, спецификации операций над стеком можно применить для уточнения алгоритмов, разработанных ранее в начале главы.

Применение абстрактного стека. Спецификации операций над абстрактным стеком позволяют уточнить разработанные ранее алгоритмы.

Уточненные алгоритмы

```
+readAndCorrect(out aStack:Stack)
// Считывает входную строку.
// Проверяет, входит ли считанный символ в стек aStack.
// Если символ равен '←', изменяет содержимое стека.

   aStack.createStack[2]
   Считать символ newChar
   while (символ newChar не является признаком конца строки)
   {
      if (символ newChar не равен '←')
         aStack.push(newChar)
      else if (!aStack.isEmpty())
         aStack.pop()
```

[2] При реализации функции *aStack.createStack()* на языке C++ стек *aStack* следует объявить экземпляром класса *Stack*, поскольку функция *createStack* реализована в виде конструктора класса.

```
      Считать символ newChar
  } // Конец оператора while

+displayBackward(in aStack:Stack)
// Выводит на экран строку, записанную в обратном порядке,
// извлекая содержимое стека aStack.

   while (!aStack.isEmpty())
   {
    aStack.pop(newChar)
    Вывести символ newChar
   } // Конец оператора while

   Ввести новую строку
```

Мы использовали операции над стеком, ничего не зная об их реализациях и даже не представляя, как выглядит сам стек. Поскольку абстрагирование возводит защитные стены вокруг реализации стека, программа может использовать стек независимо от его реализации. Если программа правильно использует операции над абстрактным типом данных, т.е. выполняет контракт, она будет корректно работать независимо от способа реализации АТД.

Следовательно, этот контракт нужно точно сформулировать. Иными словами, перед реализацией любой операции над абстрактным типом необходимо указать ее пред- и постусловия. Однако при разработке программы первые варианты спецификаций часто носят неформальный характер, и лишь позднее их можно воплотить в виде пред- и постусловий. Например, спецификации абстрактного стека, приведенные выше, оставили без ответа несколько вопросов.

Вопросы, оставленные без ответа неформальными спецификациями

- Как действуют операции *pop* и *getTop* на пустой стек?
- Какое значение присвоят операции *pop* и *getTop* переменной *stackTop*, если стек пуст?

Ответы на эти вопросы приводятся в комментариях, содержащихся в программе, реализующей стек.

Аксиомы. Как указывалось в главе 3, интуитивных спецификаций, аналогичных приведенным выше, недостаточно для формального определения абстрактного типа данных. Например, чтобы формально описать интуитивное представление о том, что последний элемент, вставленный в стек *aStack*, является первым элементом, подлежащим удалению, можно использовать следующую аксиому.

```
(aStack.push(newItem)).pop() = aStack
```

Пример аксиомы

Иными словами, если новый элемент заталкивается в стек, а затем из него выталкивается, стек *aStack* не изменяется. Дальнейшее обсуждение аксиом стека содержится в упражнении 12.

Простые примеры использования абстрактного стека

В этом разделе рассматриваются два относительно простых примера, в которых используется принцип LIFO. Мы по-прежнему будем использовать операции над абстрактным стеком, несмотря на то, что нам пока ничего не известно об их реализациях.

Проверка баланса фигурных скобок

В языке C++ используются фигурные скобки { и }, позволяющие разделять группы операторов. Например, фигурные скобки окаймляют тело функции. Если рассматривать программы на языке C++ как строки, для проверки баланса фигурных скобок можно применить стек. Например, в строке

```
abc{defg{ijk}{l{mn}}op}qr
```

фигурные скобки сбалансированы, а в строке

```
abc{def}}{ghi{jk}m
```

баланса нет. Просмотрев строку слева направо, можно определить, содержит ли она фигурные скобки. Перемещаясь по строке, можно сопоставлять каждую встречающуюся фигурную скобку "}" с последней открывающей фигурной скобкой "{". Иными словами, слева от закрывающей фигурной скобки "}" должна находиться открывающая скобка "{". Баланс соблюден, если выполняются два условия.

Требования, необходимые для соблюдения баланса фигурных скобок

1. Символ "}" всегда встречается после символа "{".
2. По достижении конца строки каждая открывающая скобка должна быть "закрыта".

Чтобы решить задачу, нужно отслеживать каждую незакрытую скобку "{", удаляя ее при обнаружении соответствующей закрывающей скобки "}". Для этого достаточно затолкнуть все открывающие скобки в стек и выталкивать их оттуда по одной, каждый раз когда встречается закрывающая скобка. В первом приближении псевдокод этой функции выглядит следующим образом.

Первое приближение

```
while (не конец строки)
{
   if (следующим символом является '{')
       aStack.push('{')
   else if (символ равен '}')
       aStack.pop()
} // Конец оператора while
```

Хотя это решение правильно отслеживает фигурные скобки, их баланс не проверяется. Чтобы убедиться в выполнении условия 1 при обнаружении символа "}", нужно проверить, пуст ли стек перед выталкиванием из него элемента. Если он пуст, цикл завершается, выводя сообщение, что строка не сбалансирована. Чтобы убедиться в выполнении условия 2, нужно проверить, пуст ли стек по достижении конца строки.

Таким образом, псевдокод функции должен быть изменен.

Псевдокод решения, включающий проверку баланса скобок

```
aStack.createStack()
balancedSoFar = true
i = 0

while (balancedSoFar и i меньше длины строки aString)
{
   ch = первый символ строки aString
   ++i

   // Заталкиваем открывающую фигурную скобку
   if (символ ch равен '{')
       aStack.push('{')
```

```
        // Закрывающая фигурная скобка
    else if (символ ch равен '}')
        if (!aStack.isEmpty())
        aStack.pop() // Выталкиваем соответствующую
                     // открывающую скобку
    else // Соответствия нет
        balancedSoFar = false
    // Все символы, кроме фигурных скобок, игнорируются
}   // Конец оператора while

if (balancedSoFar и aStack.isEmpty())
    Строка aString сбалансирована
else
    Строка aString не сбалансирована
```

На рис. 6.3 показаны стеки, возникающие при выполнении алгоритма для разных строк.

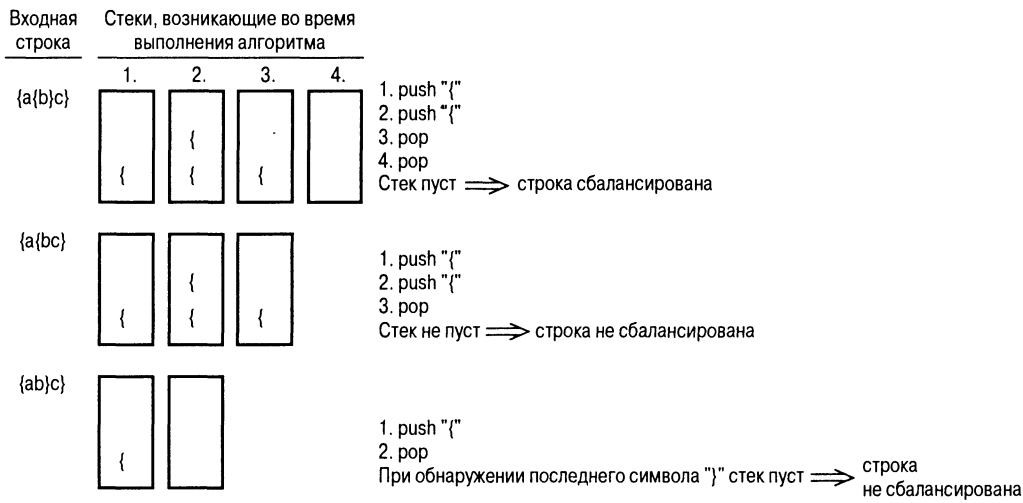


Рис. 6.3. Трассировка алгоритма проверки баланса фигурных скобок

Обратите внимание, что операция *push* может работать неправильно. Причины этого явления зависят от конкретной реализации. Например, когда стек реализуется в виде массива, функция *push* генерирует исключительную ситуацию, если массив полон. Когда стек реализуется в виде связанного списка, исключительная ситуация возникает, если оператор *new* не может выделить динамическую память. Принципы надежного программирования вынуждают нас применять блок *try-catch* перед генерированием исключительной ситуации *StackException*. В качестве альтернативы можно вызвать другую функцию, обрабатывающую данную исключительную ситуацию.

Существуют более простые решения этой задачи. Например, можно учитывать лишь текущее количество незакрытых открывающих скобок.[3] При этом хранить открывающие скобки в стеке не обязательно. Однако решение этой за-

[3] Каждый раз, когда встречается открывающая скобка, счетчик увеличивается на единицу; каждый раз, когда встречается закрывающая скобка, счетчик уменьшается на единицу. Если по достижении конца строки счетчик меньше или больше нуля, скобки не сбалансированы.

дачи на основе стека весьма поучительно и подготавливает нас к более сложным приложениям. Например, в упражнении 6, приведенном в конце главы, читателям предлагается распространить алгоритм проверки баланса на обычные круглые и квадратные скобки.

Распознавание строк языка

Рассмотрим задачу распознавания строки в языке

> L={w\$$w$', где строка w может быть пустой или содержать символы, отличные от \$, а строка w'=reverse(w)}.

Например, строки A\$A, ABC\$CBA и \$ принадлежат языку L, а строки AB\$AB и ABC\$CB — нет. (Аналогичный язык рассматривается в упражнении 11 главы 5.) Этот язык похож на язык палиндромов, рассмотренный в главе 5, однако внутри строк этого языка содержится специальный символ.

Для распознавания строк языка L можно применить стеки. Проходя первую половину строки, будем заталкивать каждый встреченный символ в стек. Достигнув символа \$, выполним откат: встречая очередной символ во второй половине строки будем выталкивать из стека один символ. При этом следует сравнивать символ, выталкиваемый из стека, с соответствующим символом во второй половине строки, чтобы проверить, что вторая часть строки представляет собой зеркальное отражение первой. Стек должен быть пуст тогда и только тогда, когда мы достигнем конца строки. В противном случае одна "половина" строки окажется длиннее другой, и, следовательно, строка не будет принадлежать языку L.

Эта стратегия реализована в алгоритме, описанном ниже. Чтобы не усложнять изложение, мы предположили, что строка *aString* содержит только один символ \$.

```
aStack.createStack()
// Затолкнуть в стек символы строки w, предшествующие
// символу $

i = 0
ch = символ, находящийся в i-й позиции строки aString
while (символ ch не равен '$')
{
   aStack.push(ch)
   ++i
   ch = символ, находящийся в i-й позиции строки aString
} // Конец оператора while

// Пропустить символ $
++i

// Сравнить с зеркальным отражением строки w
inLanguage = true // Строка принадлежит языку
while (inLanguage и i меньше длины строки aString)
  try
  {
     aStack.pop(stackTop)
     ch = символ, находящийся в i-й позиции строки aString
     if (элемент stackTop равен ch)
        ++i // Символы совпадают
     else
```

```
        // На вершине стека нет символа ch
        // (символы не совпадают)
        inLanguage = false // Строка не принадлежит языку
    } // Конец блока try
    catch StackException
    {
      // Выталкивание невозможно — стек пуст
      // (первая часть строки короче второй)
      inLanguage = false
    } // Конец блока catch

if (inLanguage и aStack.isEmpty())
   Строка aString принадлежит языку
else
   Строка aString не принадлежит языку
```

Алгоритмы, описанные в этом разделе, зависят только от спецификаций операций над стеком, но не от их реализаций.

Реализации абстрактного стека

В этом разделе разработаны три реализации абстрактного стека. Первая реализация использует для представления стека массив, вторая — связанный список, а третья — абстрактный список. Все три реализации показаны на рис. 6.4.

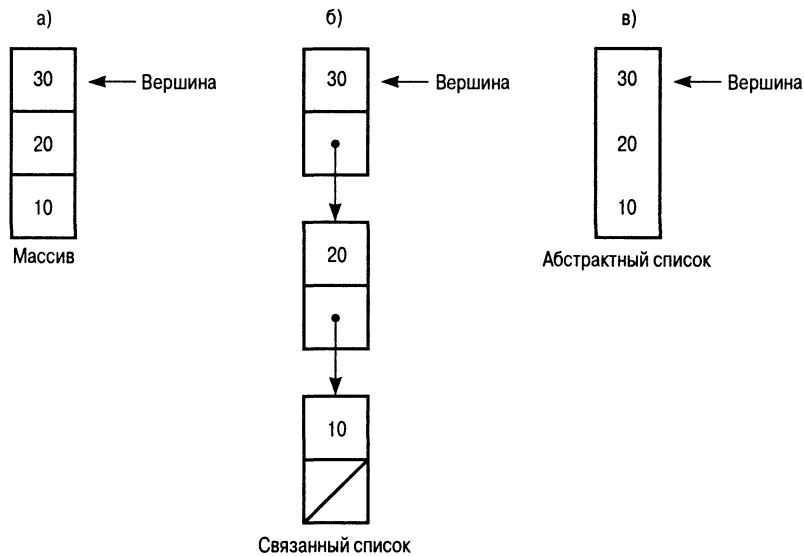


Рис. 6.4. Реализации абстрактного стека; а) в виде массива; б) в виде связанного списка; в) в виде абстрактного списка

Класс `StackException`, использованный в реализации стека, имеет следующий вид.

```
#include <exception>
#include <string>
using namespace std;
```

```
class StackException: public exception
{
public:
   StackException(const string & message="")
                  : exception(message.c_str())
   {}
}; // Конец класса StackException
```

Реализация абстрактного стека в виде массива

На рис. 6.5 показан массив *items*, в котором хранятся элементы стека, а индекс *top* соответствует вершине стека. Определим класс, экземпляры которого представляют собой стеки, а закрытыми членами являются функции *items* и *top*.

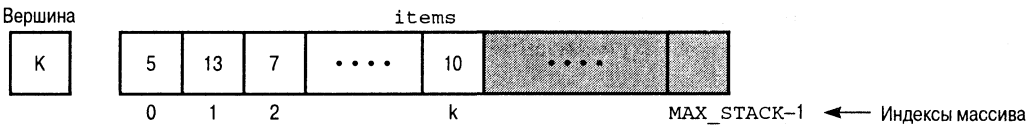


Рис. 6.5. Реализация абстрактного стека в виде массива

В этом классе конструктор по умолчанию соответствует операции *createStack*, а деструктор — операции *destroyStack*. Поскольку данные хранятся в статическом массиве, достаточно использовать автоматический деструктор и конструктор копирования.

```
// ***********************************************************
// Заголовочный файл StackA.h абстрактного стека.
// Реализация в виде массива.
// ***********************************************************
#include "StackException.h"
const int MAX_STACK = максимальный-размер-стека;
typedef тип-элементов-стека StackItemType;
class Stack
{
public:
// Конструкторы и деструктор:
   Stack(); // Конструктор по умолчанию.
   // Автоматический конструктор копирования и деструктор
   // генерируются компилятором.

// Операции над стеком:
   bool isEmpty() const;
   // Проверяет, пуст ли стек.
   // Предусловие: нет.
   // Постусловие: если стек пуст, возвращает значение true,
   // в противном случае возвращает значение false.

   void push(StackItemType newItem) throw(StackException);
   // Добавляет элемент на вершину стека.
   // Предусловие: добавляемый элемент задается аргументом newItem
   // Постусловие: если вставка выполнена успешно,
   // переменная newItem находится на вершине стека.
   // Исключительная ситуация: если элемент невозможно
```

```
    // затолкнуть в стек, генерируется исключительная
    // ситуация StackException.

    void pop() throw(StackException);
    // Удаляет вершину стека.
    // Предусловие: нет.
    // Постусловие: если стек не пуст, удаляется элемент,
    // который был вставлен последним. Если стек пуст,
    // удаление невозможно.
    // Исключительная ситуация: если стек пуст,
    // генерируется исключительная ситуация StackException

    void pop(StackItemType& stackTop) throw(StackException);
    // Извлекает и удаляет вершину стека.
    // Предусловие: нет.
    // Постусловие: если стек не пуст, переменная stackTop
    // содержит элемент, который был вставлен последним,
    // затем он удаляется. Если стек пуст, удаление невозможно,
    // и переменная stackTop остается без изменений.
    // Исключительная ситуация: если стек пуст,
    // генерируется исключительная ситуация StackException.

    void getTop(StackItemType& stackTop) const
         throw(StackException);
    // Извлекает и удаляет вершину стека.
    // Предусловие: нет.
    // Постусловие: если стек не пуст, переменная stackTop
    // содержит элемент, который был вставлен последним.
    // Если стек пуст, удаление невозможно, и переменная
    // stackTop остается без изменений. Стек не изменяется.
    // Исключительная ситуация: если стек пуст,
    // генерируется исключительная ситуация StackException.

private:
    StackItemType items[MAX_STACK]; // массив элементов стека
    int top; // Индекс вершины стека
}; // Конец класса
// Конец заголовочного файла
```

Реализации функций, описанных в предыдущем заголовочном файле, содержатся в файле реализации *StackA.cpp*.

```
// ********************************************************
// Файл реализации StackA.cpp абстрактного стека.
// Реализация в виде массива.
// ********************************************************
#include "StackA.h" // Файл спецификаций класса

StackStack::Stack(): top(-1)
{
} // Конец конструктора по умолчанию

bool Stack::isEmpty() const
{
    return top < 0;
} // Конец функции isEmpty
```

```
void Stack::push(StackItemType newItem)
{
// Если в стеке больше нет места
   if (top >= MAX_STACK-1)
      throw StackException(
      "StackException: push — стек переполнен");
   else
   {
      ++top;
      items[top] = newItem;
   } // Конец оператора if
} // Конец функции push

void Stack::pop()
{
   if (isEmpty())
      throw StackException(
            "StackException: pop — стек пуст");
   else
      --top; // Стек не пуст. Вытолкнуть вершину
} // Конец функции pop

void Stack::pop(StackItemType& stackTop)
{
   if (isEmpty())
       throw StackException(
       "StackException: pop — стек пуст");
   else
   {
       // Стек не пуст; извлечь вершину
       stackTop = items[top];
       --top; // Вытолкнуть вершину
   } // Конец оператора if
} // Конец функции pop

void Stack::getTop(StackItemType& stackTop) const
{
   if (isEmpty())
       throw StackException(
       "StackException: getTop — стек пуст");
   else
      // Стек не пуст. Извлечь вершину
      stackTop = items[top];
} // Конец функции getTop

// Конец файла реализации
```

Программа, использующая класс *Stack*, выглядит следующим образом.

```
#include <iostream>
#include "StackA.h"
using namespace std;

int main()
{
   StackItemType anItem;
   Stack aStack;
```

```
    cin >> anItem; // Считать элемент
    aStack.push(anItem); // Затолкнуть элемент в стек
. . .
```

Закрытые данные-члены скрыты от клиентов

Реализуя стек в виде класса и объявляя массив *item* и переменную *top* его закрытыми членами, вы гарантируете, что клиент не сможет разрушить стены, защищающие абстрактный тип данных. Если реализацию не скрыть внутри класса или сделать массив *items* открытым членом класса, клиент получит прямой доступ ко всем элементам стека, а не только к его вершине. Это может показаться привлекательным решением, однако оно нарушает спецификации абстрактного стека. Если вам действительно нужен доступ ко всем элементам массива, не используйте стек!

Исключительная ситуация StackException предотвращает неправильную работу со стеком

Исключительная ситуация *StackException* предупреждает о возникновении отклонений в работе стека, например, о попытке вставить элемент в полный стек или удалить элемент из пустого стека.

В заключение отметим, что функция *push* получает элемент *newItem* по значению. Следовательно, она копирует значение переменной *newItem* и лишь потом использует его. Если элементами стека являются обычные целые числа или символы, такие копии не требуют дополнительных затрат. Однако если элементами стека являются объекты другого класса, копирование может оказаться неэффективным. Этого следует избегать, передавая аргумент *newItem* по ссылке.

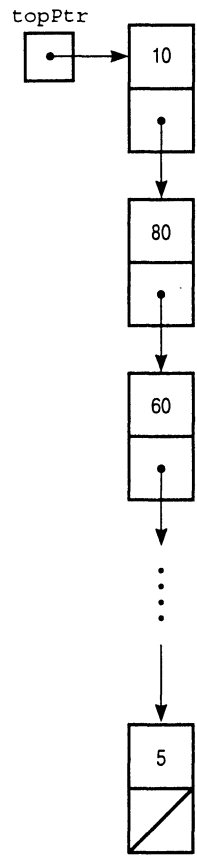


Рис. 6.6. Реализация абстрактного стека в виде связанного списка

Реализация абстрактного стека в виде связанного списка

Во многих приложениях размеры стека могут динамически увеличиваться и уменьшаться. На рис. 6.6 показана реализация абстрактного стека в виде связанного списка, где указатель *topPtr* ссылается на первый элемент. Поскольку для связанных списков выделяется динамическая память, класс стека должен иметь конструктор копирования и деструктор.

Как всегда, программа состоит из заголовочного файла и файла реализации. Обратите внимание, что пред- и постусловия функций-членов класса полностью совпадают с их аналогами при реализации стека в виде массива, поэтому они не указываются.

```
//************************************
// Заголовочный файл StackP.h
// абстрактного стека.
// Реализация в виде связанного
// списка.
```

```
// ********************************************************
#include "StackException.h"
typedef тип-элементов-стека StackItemType;
class Stack
{
public:
// Конструкторы и деструктор:
   Stack(); // Конструктор по умолчанию
   Stack(const Stack& aStack); // Конструктор копирования
   ~Stack(); // Деструктор

// Операции над стеком:
   bool isEmpty() const;
   void push(StackItemType newItem) throw(StackException);
   void pop() throw(StackException);
   void pop(StackItemType& stackTop) throw(StackException);
   void getTop(StackItemType& stackTop) const
        throw(StackException);
private:
   struct StackNode // Узел стека
   {
      StackItemType item; // Данные, содержащиеся в узле стека
      StackNode *next; // Указатель на следующий узел
   }; // end struct

   StackNode *topPtr; // Указатель на первый элемент стека
}; // Конец класса Stack
// Конец заголовочного файла.

// ********************************************************
// Файл реализации StackP.cpp абстрактного стека.
// Реализация в виде связанного списка.
// ********************************************************
#include "StackP.h" // Заголовочный файл
#include <cstddef> // Определение константы NULL
#include <cassert> // Определение макроса assert

Stack::Stack() : topPtr(NULL)
{
} // Конец конструктора по умолчанию

Stack::Stack(const Stack& aStack)
{
   if (aStack.topPtr == NULL)
      topPtr = NULL; // Исходный список пуст
   else
   {
      // Копируем первый узел
      topPtr = new StackNode;
      assert(topPtr != NULL);
      topPtr->item = aStack.topPtr->item;

      // Копируем остальную часть списка
     StackNode *newPtr = topPtr; // Новый указатель на список
     for (StackNode *origPtr = aStack.topPtr->next;
```

```
                    origPtr != NULL;
                    origPtr = origPtr->next)
      {
         newPtr->next = new StackNode;
         assert(newPtr->next != NULL);
         newPtr = newPtr->next;
         newPtr->item = origPtr->item;
      } // Конец цикла for

      newPtr->next = NULL;
   } // Конец оператора if
} // Конец конструктора копирования

Stack::~Stack()
{
   // Выталкивать элементы, пока стек не станет пустым
   while (!isEmpty())
   pop();
   // Диагностическое утверждение: topPtr == NULL
}  // Конец деструктора

bool Stack::isEmpty() const
{
   return topPtr == NULL;
} // Конец функции isEmpty

void Stack::push(StackItemType newItem)
{
   // Создаем новый элемент
   StackNode *newPtr = new StackNode;
   if (newPtr == NULL) // Проверка выделения памяти
      throw StackException(
      "StackException: push — выделить память невозможно");
   else
   {
      // Выделение памяти прошло успешно.
      // Записать данные в новый узел
      newPtr->item = newItem;
      // Вставить новый узел
      newPtr->next = topPtr;
      topPtr = newPtr;
   } // Конец оператора if
} // Конец функции push

void Stack::pop()
{
   if (isEmpty())
       throw StackException(
       "StackException: pop — стек пуст");
   else
   {
       // Стек не пуст. Удалить вершину
       StackNode *temp = topPtr;
       topPtr = topPtr->next;
```

```
        // Освободить память, занятую удаленным элементом
        temp->next = NULL; // Защита
        delete temp;
    } // Конец оператора if
} // Конец функции pop

void Stack::pop(StackItemType& stackTop)
{
    if (isEmpty())
        throw StackException(
        "StackException: pop — стек пуст");
    else
    {
        // Стек не пуст. Извлечь и удалить вершину
       stackTop = topPtr->item;
       StackNode *temp = topPtr;
       topPtr = topPtr->next;

       // Освободить память, занятую удаленным элементом
       temp->next = NULL; // Защита
       delete temp;
    } // Конец оператора if
} // end pop

void Stack::getTop(StackItemType& stackTop) const
{
    if (isEmpty())
       throw StackException(
       "StackException: getTop — стек пуст");
    else
       // Стек не пуст. Извлечь вершину
       stackTop = topPtr->item;
} // Конец функции getTop
// Конец файла реализации.
```

Необходимо предусмотреть явный конструктор копирования

Конструктор копирования, предусмотренный в этом классе, выполняет глубокое копирование стека. Если в классе не предусмотрен такой конструктор, то автоматический конструктор копирования, сгенерированный компилятором, скопирует лишь указатель на вершину стека. В этом случае исходный указатель и его копия будут ссылаться на один и тот же связанный список. Сам стек скопирован не будет. По этой причине необходимо явно предусматривать свой собственный конструктор копирования.

Реализация стека в виде абстрактного списка

Для представления элементов стека можно применять абстрактный список, показанный на рис. 6.7. Если считать, что элемент, находящийся на первой позиции списка, является вершиной стека, то операцию над стеком *push(newItem)* можно реализовать с помощью операции над абстрактным списком *insert(1, newItem)*. Аналогично, операцию *pop()* можно реализовать с помощью операции *remove(1)*, а операцию *getTop(stackTop)* — с помощью операции *retrieve(1, stackTop)*.

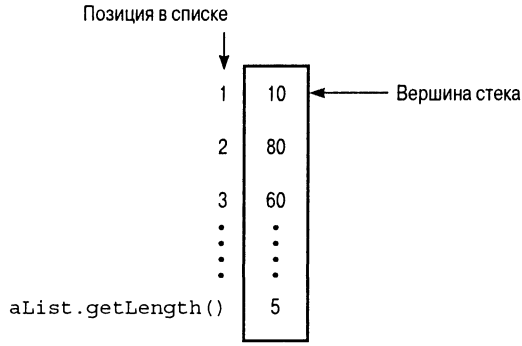


Рис. 6.7. Реализация абстрактного стека в виде абстрактного списка

Напомним, что в главах 3 и 4 абстрактный список был представлен классом *List*. Класс, приведенный ниже, использует класс *List* для реализации стека. Пост- и предусловия функций-членов остаются без изменения.

```
// ********************************************************
// Заголовочный файл StackL.h абстрактного стека.
// Реализация в виде абстрактного списка.
// ********************************************************
#include "StackException.h"
#include "ListP.h" // Операции над списком

typedef ListItemType StackItemType;
class Stack
{
public:
// Конструкторы и деструктор:
   Stack(); // Конструктор по умолчанию
   Stack(const Stack& aStack); // Конструктор копирования
   ~Stack(); // Деструктор

// Операции над стеком:
   bool isEmpty() const;
   void push(StackItemType newItem) throw(StackException);
   void pop() throw(StackException);
   void pop(StackItemType& stackTop) throw(StackException);

private:
   List aList; // Список элементов стека
}; // Конец класса
// Конец заголовочного файла.

// ********************************************************
// Файл реализации StackL.cpp абстрактного стека.
// Реализация в виде абстрактного списка.
// ********************************************************
#include "StackL.h" // Заголовочный файл
Stack::Stack()
{
} // Конец конструктора по умолчанию
```

```
Stack::Stack(const Stack& aStack): aList(aStack.aList)
{
} // Конец конструктора копирования

Stack::~Stack()
{
} // Конец деструктора

void getTop(StackItemType& stackTop) const
     throw(StackException);

bool Stack::isEmpty() const
{
   return aList.isEmpty();
} // Конец функции isEmpty

void Stack::push(StackItemType newItem)
{
   try
   {
       aList.insert(1, newItem);
   } // Конец блока try

   catch (ListException e)
   {
      throw StackException(
            "StackException: невозможно затолкнуть элемент");
   } // Конец блока catch
} // Конец функции push

void Stack::pop()
{
   try
   {
      aList.remove(1);
   } // Конец блока try

   catch (ListIndexOutOfRangeException e)
   {
      throw StackException(
            "StackException: pop — стек пуст");
   } // Конец блока catch
} // Конец функции pop

void Stack::pop(StackItemType& stackTop)
{
   try
   {
      aList.retrieve(1, stackTop);
      aList.remove(1);
   } // Конец блока try

   catch (ListIndexOutOfRangeException e)
   {
      throw StackException(
            "StackException: pop — стек пуст");
```

```
    } // Конец блока catch
} // Конец функции pop

void Stack::getTop(StackItemType& stackTop) const
{
    try
    {
        aList.retrieve(1, stackTop);
    } // Конец блока try

    catch (ListIndexOutOfRangeException e)
    {
        throw StackException(
              "StackException: getTop — стек пуст");
    } // Конец блока catch
} // Конец функции getTop
// Конец файла реализации.
```

Член *aList* является экземпляром другого класса. В принципе члены класса могут быть экземплярами любого другого класса, отличающегося от определяемого (в нашем примере определяемым является класс *Stack*). Кроме того, конструктор класса *List* вызывается перед конструктором класса *Stack*. Конструктор копирования класса *Stack* копирует объект *aList*, вызывая конструктор копирования класса *List*.

Тело деструктора пусто. Если в определении класса объявляется явный деструктор, его необходимо реализовать, даже если он не выполняет никаких действий. Однако в данном случае деструктор можно было бы пропустить, положившись на компилятор. Поскольку в классе *List* предусмотрен правильный деструктор, деструктор класса *Stack* также будет работать корректно.

Сравнение реализаций

Мы рассмотрели реализации стека в виде массива, связанного списка и абстрактного списка. Массивы и связанные списки являются структурами данных, а список — это абстрактный тип данных, который сам должен быть реализован с помощью массива или связанного списка. Таким образом, в конечном итоге реализация стека будет использовать либо массив, либо связанный список.

Фиксированный и переменный размеры

Преимущества и недостатки реализаций стека в виде массива или связанного списка совершенно аналогичны рассмотренным нами ранее. Реализация стека использует статический массив. Это предотвращает выполнение операции *push* над заполненным стеком, поскольку при этом полностью исчерпывается память, выделенная для массива. Если это ограничение неприемлемо, нужно использовать либо динамический массив, либо связанный список. Для задачи чтения и исправления строк, например, фиксированный размер не создает проблем. Поскольку на экране строка не может превышать 80 символов, имеет смысл применить статический массив.

Использование готовых классов экономит время

Допустим, для реализации стека мы решили использовать указатели. Какую реализацию выбрать: связанный или абстрактный список? Поскольку элементы абстрактного списка фактически представляются в виде связанного списка, такая опосредованная реализация стека может оказаться неэффективной. Возможно, это и так, однако абстрактный список намного нагляднее. Если у нас уже есть реализация абстрактного списка в виде класса, зачем

вновь изобретать велосипед? При реализации новых абстрактных типов следует по возможности применять уже готовые классы. Вопросы повторного использования готовых классов рассматриваются в главе 8.

Класс stack из стандартной библиотеки шаблонов

Класс stack из библиотеки STL является шаблонным

В главе 4 мы уже встречались со стандартной библиотекой шаблонов и контейнерным типом *list*. Библиотека STL содержит также контейнер *stack*, очень похожий на класс *Stack*, рассмотренный нами выше. Однако класс *stack* из библиотеки STL является шаблонным. Кроме того, в классе *stack* предусмотрена дополнительная функция *size*, определяющая количество элементов стека. Функции-члены *isEmpty* и *getTop* в шаблонном классе имеют другие имена. Рассмотрим немного упрощенную спецификацию шаблонного класса *stack*.

```
template <class T, class Container = deque <T> >
class stack
{
public:
   explicit stack(const Container& cnt = Container());
   // Конструктор по умолчанию; инициализирует пустой стек.
   // Предусловие: нет.
   // Постусловие: создан пустой стек.

   bool empty() const;
   // Определяет, пуст ли стек.
   // Предусловие: нет.
   // Постусловие: если стек пуст, возвращает значение true;
   // в противном случае возвращает значение false.

   size_type size() const;
   // Определяет размер стека. Тип возвращаемого значения
   // size_type является целочисленным.
   // Предусловие: нет.
   // Постусловие: возвращает количество элементов в стеке.

   T &top();
   // Возвращает ссылку на вершину стека.
   // Предусловие: нет.
   // Постусловие: элемент остается в стеке.

   void pop();
   // Удаляет вершину стека.
   // Предусловие: нет.
   // Постусловие: из стека удален элемент,
   // добавленный последним.

   void push(const T& x);
   // Добавляет элемент на вершину стека.
   // Предусловие: нет.
   // Постусловие: элемент x расположен на вершине стека.
}  // Конец класса stack из библиотеки STL
```

Спецификация шаблона имеет два параметра. Первый параметр, *T*, задает тип данных, хранящихся в элементах стека. Этот параметр аналогичен параметру класса *list* из библиотеки STL, задающему тип данных, хранящихся в элементах списка. Второй параметр, *Container*, представляет собой контейнерный класс, который используется стандартной библиотекой шаблонов для реализации класса *stack*, точно так же, как мы использовали класс *List* для реализации нашего класса *Stack*. Тип контейнера передается конструктору класса *stack* в качестве формального параметра. Этот момент для нас пока не важен, поскольку мы будем использовать шаблонные параметры и аргументы конструктора, задаваемые по умолчанию. Эти идеи более подробно изложены в главе 7 при описании конструктора по умолчанию в шаблонном классе *deque*.

В шаблонном классе stack указывается контейнерный класс

Обратите внимание на ключевое слово *explicit* в объявлении конструктора. Оно ограничивает возможности конструктора при создании новых экземпляров. В частности, для вызова конструктора нельзя использовать оператор присваивания.

Ключевое слово explicit в объявлении конструктора не позволяет использовать вызов конструктора в операторе присваивания

Рассмотрим пример использования шаблонного класса *stack*.

```
#include <iostream>
#include <stack>
using namespace std;

int main()
{
   stack<int> aStack;
   int item;

   // В данный момент стек пуст
   if (aStack.empty())
      cout << "Стек пуст" << endl;

   for (int j = 0; j < 5; j++)
      aStack.push(j); // Помещает элемент на вершину стека

   while (!aStack.empty())
   {
      cout << aStack.top() << " ";
      aStack.pop();
   } // Конец оператора while
   return 0;
} // Конец функции main
```

Приложение: алгебраические выражения

В этом разделе рассматриваются две задачи, которые остроумно решаются с помощью абстрактного стека. Обратите внимание, что для их решения применяется именно абстрактный стек, поэтому при выполнении операции мы не будем делать предположений о конкретной реализации стека. Выбор конкретной реализации следует делать в последний момент.

Использование операций над абстрактным типом данных не должно зависеть от его реализации

В главе 5 была описана рекурсивная грамматика языка алгебраических выражений. Напомним, что для устранения неоднозначности, присущей инфиксным выражениям, используются префиксная и постфиксная формы записи. Посмотрим, как можно вычислить значение инфиксного и постфиксного выражении с помощью стека. Чтобы не отвлекаться на мелочи, будем предполагать, что допускаются только бинарные операторы *, /,+ и -, а унарные операторы и оператор возведения в степень запрещены.

Чтобы вычислить инфиксное выражение, его сначала нужно преобразовать в постфиксную форму, а затем вычислить полученное постфиксное выражение

Стратегия решения задачи заключается в следующем. Сначала мы разработаем алгоритм вычисления постфиксных выражений, а затем — алгоритм преобразования инфиксных выражений в постфиксные. Объединение этих алгоритмов позволит вычислить значение инфиксного выражения. Таким образом, мы избежим непосредственного вычисления инфиксного выражения, которое представляет собой довольно сложную проблему, рассмотренную в задании 6.

Вычисление постфиксных выражений

Как указывалось в главе 5, некоторые калькуляторы используют постфиксную форму записи. Например, чтобы вычислить значение выражения

```
2 * (3 + 4)
```

с помощью постфиксного калькулятора, нужно ввести последовательность символов 2, 3, 4, +, *, соответствующую постфиксному выражению

```
2 3 4 + *
```

Напомним, что оператор в постфиксном выражении применяется к двум операндам, стоящим непосредственно перед ним. Таким образом, калькулятор должен иметь возможность распознавать операнды, введенные последними. Эту возможность предоставляет абстрактный стек. Фактически каждый раз, когда в калькулятор вводится новый операнд, он заталкивается в стек. Когда вводится новый оператор, из стека выталкиваются два операнда, а обратно заталкивается результат. Эта последовательность действий проиллюстрирована на рис. 6.8. Результат, число 14, находится на вершине стека.

Нажатые клавиши	Действия калькулятора		После выполнения операции: Стек (от дна к вершине)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

Рис. 6.8. Действия постфиксного калькулятора при вычислении выражения 2(3+4)*

Упрощения

Формализуя действия калькулятора, можно разработать алгоритм вычисления постфиксного выражения, заданного строкой символов. Для простоты сделаем следующие предположения.

- Строка представляет собой синтаксически правильное постфиксное выражение.
- Унарные операторы не допускаются.
- Оператор возведения в степень не допускается.
- Операторы задаются строчными буквами.

Псевдокод алгоритма, вычисляющего постфиксное выражение

- Псевдокод алгоритма имеет следующий вид.

```
for (каждый символ ch в строке)
{
   if (символ ch является операндом)
      Затолкнуть в стек значение, заданное операндом ch
   else // Символ ch является операндом op
   {
      // Вычислить результат и затолкнуть его в стек
      operand2 = вершина стека
      Вытолкнуть элемент из стека
      operand1 = вершина стека
      Вытолкнуть элемент из стека
      result = operand1 op operand2
      Затолкнуть результат в стек
   } // Конец оператора if
} // Конец оператора for
```

По завершении этого алгоритма значение выражения будет расположено на вершине стека. Реализация этого алгоритма предлагается читателям в качестве задания 3, приведенного в конце главы.

Преобразование инфиксного выражения в постфиксное

Теперь мы умеем вычислять постфиксные выражения. Следовательно, мы сможем вычислить инфиксное выражение, если преобразуем его в постфиксную форму. Рассмотрим уже знакомое нам инфиксное выражение *(a+b)*c/d-e*. Здесь используются скобки, а операторы выполняются слева направо.

Нужно ли вообще вычислять инфиксные выражения? Разумеется, ведь они чаще всего встречаются в программах. Компилятор, транслирующий программу, должен генерировать машинные команды для вычисления заданного выражения. Для этого компилятор сначала преобразует инфиксные выражения в постфиксные. Таким образом, преобразование инфиксной формы записи в постфиксную не только позволяет вычислять инфиксные выражения, но и проливает свет на некоторые детали компиляции.

Некоторые сведения о преобразовании инфиксной формы записи в постфиксную

Выполняя преобразования вручную, можно обнаружить следующее.

- Порядок следования операндов никогда не изменяется.
- Оператор всегда находится справа от операндов. Иными словами, если в инфиксном выражении операнд *x* предшествовал оператору *op*, этот порядок сохранится и в постфиксной форме записи.
- Все скобки удаляются.

В результате приходим к выводу, что основной задачей алгоритма является распознавание операторов и определение их позиции в строке.

Ниже приведено первое приближение алгоритма преобразования инфиксного выражения в префиксную форму *postfixExp*.

Первое приближение алгоритма преобразования инфиксного выражения в префиксную форму

```
Инициализировать строку postfixExp
пустой строкой
for (каждый символ ch в инфиксном выражении)
{
   switch (ch)
   {
      case символ ch является операндом:
        Добавить символ ch в конец выражения postfixExp
        break
      case символ ch является оператором:
        Записывать символы ch, пока есть место
      break
      case символ ch равен '(' или ')':
        Игнорировать символ ch
      break
   } // Конец оператора switch
} // Конец оператора for
```

Возможно, вы уже догадались, что просто игнорировать скобки нельзя, поскольку они влияют на порядок выполнения операторов. В любом инфиксном выражении все символы, заключенные в скобки, представляют собой изолированное подвыражение, состоящее из оператора и двух операндов. Следовательно, подвыражение необходимо вычислять независимо от остальной части выражения. Что бы ни представляла собой остальная часть выражения, оператор, указанный в подвыражении, применяется лишь к своим операндам. Скобки можно интерпретировать следующим образом.

После вычисления подвыражения остается его результат; все, что находится внутри скобок, можно игнорировать.

Скобки, приоритеты операторов и правила ассоциативности позволяют определить, куда поместить оператор в постфиксном выражении

Таким образом, скобки, а также приоритеты операторов и правила ассоциативности, позволяют определить место оператора в постфиксном выражении.

В главе 5 был рассмотрен простой способ преобразования инфиксного выражения, полностью заключенного в скобки, в постфиксную форму. Поскольку каждый оператор находился внутри пары скобок, мы просто перемещали каждый оператор на место соответствующей закрывающей скобки, а затем удаляли все скобки из полученного выражения.

Реальная задача намного сложнее, поскольку инфиксное выражение не всегда полностью заключается в скобки. На практике задача усложняется приоритетами операторов и правилами ассоциативности. Ниже перечислены инструкции, которым необходимо следовать при считывании символов инфиксной строки слева направо.

Пять этапов преобразования инфиксного выражения в префиксное

1. Обнаружив операнд, добавьте его в строку *postfixExp*. *Обоснование*: порядок следования операторов в постфиксном и инфиксном выражениях одинаков, а операнды всегда находятся слева от операторов.

2. Затолкнуть символ "(" в стек.
3. Если в момент обнаружения оператора стек оказался пустым, затолкните оператор в стек. Если стек не пуст, вытолкните оттуда операторы более высокого или одинакового приоритета, добавив их в строку *postfixExp*. Остановитесь, если обнаружен символ "(" или оператор более низкого приоритета, либо если стек опустел. Затем затолкните новый оператор в стек. Таким образом соблюдается приоритет операторов и правила ассоциативности. Обратите внимание, что выталкивание элементов из стека продолжается до тех пор, пока не обнаружится оператор, имеющий более низкий приоритет, чем текущий. Если операторы имеют одинаковый приоритет, выталкивание элементов продолжается, поскольку это соответствует установленному порядку выполнения операторов. Первым выполняется самый левый оператор — этот оператор уже находится в стеке.
4. Обнаружив символ ")", вытолкните операторы из стека и добавьте их в конец строки *postfixExp*, пока не обнаружится соответствующая открывающая скобка "(". *Обоснование*: внутри скобок приоритеты и ассоциативность операторов определяется порядком их следования. На этапе 3 операторы уже упорядочены в соответствии с этими правилами.
5. Достигнув конца строки, добавьте оставшееся содержимое стека в конец строки *postfixExp*.

На рис. 6.9 показаны результаты трассировки этого алгоритма на примере выражения *a-(b+c*d)/e*. Предполагается, что в исходном положении стек *aStack* и строка *posfixExp* пусты. В результате выполнения алгоритма строка *postfixExp* содержит постфиксное выражение *abcd*+e/-*.

ch	Стек (от дна к вершине)	postfixExp	
a		a	
-	-	a	
(	-(	a	
b	-(	ab	
+	-(+	ab	
c	-(+	abc	
*	-(+*	abc	
d	-(+*	abcd	
)	-(+	abcd*	
	-(	abcd*+	Переносить операторы из стека в строку postfixExp, пока не обнаружится символ ")"
	-	abcd*+	
/	-/	abcd*+	
e	-/	abcd*+e	
		abcd*+e/-	Копировать операторы из стека в строку postfixExp

*Рис. 6.9. Трассировка алгоритма, преобразующего инфиксное выражение a-(b+c*d)/e в постфиксное*

Описанные выше этапы решения задачи позволяют сформулировать совершенно точный псевдокод. Символ +, использованный в псевдокоде, означает конкатенацию (склеивание) строк, так что выражение *postfixExp+x* приписывает к строке *postfixExp* символ *x*.

Псевдокод алгоритма, преобразующего инфиксное выражение в постфиксное

```
for (каждый символ ch в инфиксном
выражении)
{
    switch (ch)
```

```
    {
        case operand: // Добавить операнд в конец
                      // постфиксного выражения
             postfixExp = postfixExp + ch
             break
        case '(': // записать символ '(' в стек
             aStack.push(ch)
             break
        case ')': // выталкивать вершину стека,
                  // пока не обнаружится символ '('
             while (на вершине стека нет символа '(')
             {
               postfixExp = postfixExp + (вершина стека aStack)
               aStack.pop()
             } // Конец оператора while

             aStack.pop() // remove the open parenthesis
             break
        case operator: // Выполнить операторы с
                       // более высоким приоритетом
             while (!aStack.isEmpty()
                    и на вершине стека нет символа '('
                    и приоритет ch не превосходит
                    приоритет вершины стека)
             {
               postfixExp = postfixExp + (вершина стека aStack)
               aStack.pop()
             } // Конец оператора while

            aStack.push(ch) // Записать новый оператор
            break
    } // Конец оператора switch
} // Конец оператора for

// Добавить к строке postfixExp операторы, оставшиеся в стеке
while (!aStack.isEmpty())
{
   postfixExp = postfixExp + (вершина стека aStack)
   aStack.pop()
} // Конец оператора while
```

Поскольку в этом алгоритме предполагается, что инфиксное выражение является синтаксически правильным, исключительную ситуацию *StackException* можно игнорировать. В задании 5 читателю предлагается снять это предположение. Для этого вам придется использовать блоки *try-catch* при выполнении каждой операции над стеком.

Приложение: поиск

Рассмотрим применение стеков при решении абстрактной **задачи поиска** (search problem). Конкретным примером задачи поиска является, например, вычисление пути из одной точки в другую. Эту задачу можно решить с помощью стеков и рекурсии. Рекурсивное решение этой задачи позволяет выявить тесную связь между стеками и рекурсией.

Существует ли маршрут из одного города в другой

Допустим, авиакомпания *High Planes Airline (HPA)* разрабатывает программу для обработки запросов на полеты из одного города в другой. Поскольку нас интересует лишь один аспект этой проблемы — использование стеков, — упростим задачу: достаточно ответить, существует ли последовательность рейсов, соединяющая пункт отправления и пункт назначения. В реальной задаче нужно также вычислять реальный рейс (см. задание 11).

Будем предполагать, что вся информация о полетах содержится в трех текстовых файлах.

- Названия городов, в которые летают самолеты авиакомпании HPA.
- Пары названий городов, соединенных рейсами компании HPA (пункт отправления — пункт назначения).
- Пары названий городов, для которых заказан полет (пункт отправления — пункт назначения).

Программа должна выводить на экран следующее сообщение.

```
Заказан полет из Провиденса в Сан-Франциско.
Компания HPA осуществляет полеты из Провиденса в Сан-Франциско.

Заказан полет из Филадельфии в Альбукерк.
Компания HPA не осуществляет полеты из Филадельфии в Альбукерк.

Заказан полет из Солт-Лейк-Сити в Париж.
Извините. Самолеты компании HPA в Париж не летают.
```

Представление данных. На карте, изображенной на рис. 6.10, показаны авиарейсы компании HPA. Стрелка из города C_1 в город C_2 означает полет из точки C_1 в точку C_2. В этом случае говорят, что точка C_2 является **соседней** (adjacent) по отношению к точке C_1, а путь из точки C_1 в точку C_2 называется **направленным** (directed). Из того, что точка C_2 является соседней по отношению к точке C_1 не следует, что точка C_1 является соседней по отношению к точке C_2. Например, на рис. 6.10 существует рейс из города R в город X, но нет рейса из города X в город R. Как мы увидим в главе 13, карта, изображенная на рис. 6.10, называется **графом** (graph).

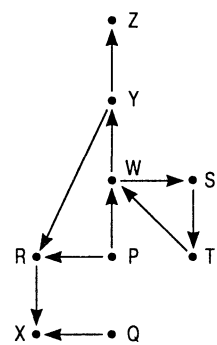


Рис. 6.10. Карта авиарейсов компании HPA

Итеративное решение с помощью стеков

Получив запрос на полет из одного города в другой, нужно определить, существует ли подходящий рейс. Например, анализ карты, изображенной на рис. 6.10, показывает, что заказчик может перелететь из города *P* в город *Z*, вылетев сначала в город *W*, затем — в город *Y*, а уже оттуда — в город *Z*. Иными словами, существует направленный путь из точки *P* в точку *Z*: *P*→*W*, *W*→*Y*, *Y*→*Z*. Следовательно, нужно разработать алгоритм поиска направленного пути из пункта отправления в пункт назначения. Такой рейс может быть прямым, либо представлять собой последовательность рейсов. Описанный ниже метод решения называется **полным перебором** (exhaustive search). Он заключается в том, что алгоритм перебирает все возможные последовательности рейсов, начинающиеся с точки отправления, пока не обнаружится искомая последовательность, либо не станет ясно, что задача не имеет решения. Для организации такого перебора хорошо подходит абстрактный стек.

Сначала попробуем решить задачу вручную. Начнем с точки C_0 и выберем произвольный рейс, начинающийся в этом городе. Этот рейс приведет нас в новый город, C_1. Если город C_1 является пунктом назначения, задача решена; в противном случае нужно продолжить поиск рейса, начиная с города C_1. Это приведет нас в город C_2. Если город C_2 является пунктом назначения, задача решена; в противном случае нужно продолжить поиск рейса, начиная с города C_2, и т.д.

Возможные исходы стратегии полного перебора

Рассмотрим возможные исходы стратегии полного перебора.

1. Пункт назначения достигнут, следовательно, искомый маршрут существует.
2. Рейс заканчивается в городе *C*, из которого авиакомпания вылеты не совершает.
3. Произошло зацикливание. Например, из C_1 в C_2, из C_2 в C_3 и из C_3 обратно в C_1. Эти перелеты могут быть бесконечными, иными словами, алгоритм содержит бесконечный цикл.

Если бы первый исход был единственным, все было бы прекрасно. Однако поскольку авиакомпания HPA не совершает полеты из каждого города, у нас нет основания ожидать, что алгоритм всегда будет приводить к положительному решению. Например, если пунктом отправления на рис. 6.10 является город *P*, а пунктом назначения — город *Q*, то алгоритм не найдет искомого решения.

Даже если последовательность перелетов из одного города в другой существует, стратегия, описанная выше, не всегда приводит к правильному результату — на каждом шаге алгоритм должен выбирать подходящий рейс. Например, хотя путь из города *P* в город *Z* на рис. 6.10 существует, алгоритм может его не обнаружить и вместо этого сообщить, что решения нет, или зациклиться. Иными словами, допустим, что из города *P* алгоритм решил выбрать рейс в город *R*. Из города *R* можно отправиться в город *X*, а оттуда самолеты не летают (исход 2). Однако алгоритм может выбрать рейс из города *P* в город *W*. Из города *W* можно вылететь в город *S*. Оттуда можно добраться до города *T* и обратно в город *W*. Из города *W* алгоритм может снова выбрать рейс в город *T* (исход 3).

Алгоритм нужно немного усложнить, чтобы он находил правильный рейс, если тот существует, а в противном случае сообщал, что решения нет. Допустим, что описанная выше стратегия привела нас в город *C*, откуда самолеты не летают. Естественно, отсюда вовсе не следует, что рейс из точки отправления в пункт назначения не существует. Можно лишь утверждать, что из города *C* невозможно попасть в пункт назначения. Иными словами, лететь в город *C* не сле-

довало. Обнаружив ошибку, алгоритм может вернуться в предыдущий город *C'*, выполнив *откат* (backtracking). Вернувшись в город *C'*, алгоритм может выбрать другой пункт назначения, отличный от *C*. Не исключено, что из города *C'* просто нет других рейсов, кроме рейса в город *C*. Значит, перелет в город *C'* тоже является ошибочным. Это вынуждает нас снова выполнить откат, на этот раз в город, из которого мы прилетели в город *C'*.

Вернемся к рис. 6.10. Пытаясь достичь города *Z* из города *P*, алгоритм может выбрать сначала рейс из города *P* в город *R*, а оттуда — в город *X*. Поскольку из города *X* нет ни одного рейса, алгоритм должен вернуться в город *R*, откуда мы прибыли в город *X*. Вернувшись в город *R*, алгоритм может попытаться выбрать какой-нибудь другой город, кроме города *X*. Оказывается, это невозможно. Тогда алгоритм должен выполнить еще один откат, на этот раз в город *P*, из которого мы прибыли в город *R*. Из города *P* мы можем отправиться в город *W*, сделав первый шаг в правильном направлении!

Итак, в ходе выполнения алгоритма, основанного на последовательном переборе вариантов, нужно хранить информацию о посещенных городах. Заметим сначала, что при откате в город *C* алгоритм должен вернуться в город, откуда он прибыл в город *C* в первый раз. Это наводит нас на мысль, что названия посещенных городов следует хранить в стеке. Каждый раз, когда мы решаем посетить какой-нибудь город, мы будем заталкивать его имя в стек, как показано на рис. 6.11, *а–в*. Следующий кандидат на посещение выбирается среди городов, соседних с городом, имя которого находится на вершине стека. Если нужно выполнить откат к городу *C*, находящемуся на вершине стека (например, если мы зашли в тупик), достаточно просто вытолкнуть название города из стека, как показано на рис. 6.11, *г*. После этого на вершине стека окажется название города, из которого мы прибыли в город *C* в первый раз. На рис. 6.11, *д, е* показан откат в город *P* и последующий перелет в город *W*.

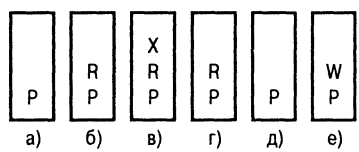


Рис. 6.11. Стек названий городов при путешествии: а) из города P; б) в город R; в) в город X; г) назад в город R; д) назад в город P; е) в город W

Псевдокод разработанного алгоритма имеет следующий вид.

```
aStack.createStack()

aStack.push(originCity) // затолкать название точки
                        // отправления в стек aStack
while (последовательность перелетов из точки
       отправления в пункт назначения)
{
   if (необходимо выполнить откат из города,
      название которого находится на вершине стека)
      aStack.pop()
   else
   {
      Выбрать пункт назначения C для перелета из города,
      название которого находится на вершине стека
```

```
            aStack.push(C)
        } // Конец оператора if
    } // Конец оператора while
```

Содержимое стека соответствует последовательности перелетов, составляющих изучаемый рейс. Название города, находящееся на вершине стека, относится к городу, в котором мы находимся, ниже расположено название города, откуда мы прибыли, и т.д. вплоть до дна стека, на котором находится точка отправления. Иными словами, инвариант цикла `while` имеет следующий вид.

Инвариант

Стек содержит направленный путь из пункта отправления, имя которого находится на дне стека, в пункт назначения, название которого расположено на вершине стека.

Следовательно, всегда существует возможность повторно выполнить все шаги алгоритма, возвращаясь назад по отмеченному рейсу.

Теперь рассмотрим следующую тему.

Откат из города, название которого расположено на вершине стека

Во-первых, откат из города, название которого расположено на вершине стека, необходим, если из этого города не вылетают самолеты. Во-вторых, откат необходим при возникновении зацикливания.

Откат, если на карте не осталось городов, в которых мы не были

Стратегия отката сводится к следующему: *назад нужно возвращаться лишь тогда, когда нет возможности двигаться вперед.* Следовательно, откат нужно выполнять только тогда, когда на карте не осталось городов, где мы еще не побывали. В качестве обоснования этого принципа рассмотрим две ситуации.

Две причины, по которым не следует возвращаться в прежние места

- Если вы уже побывали в городе C и его название еще хранится в стеке — т.е. принадлежит последовательности городов, через которые проходит изучаемый маршрут, — не следует посещать его вновь. Если маршрут начинается в точке C, проходит через точки C_1, C_2, ..., C_k, затем вновь через точку C, а затем — через точку C', его можно заменить прямым рейсом из точки C в точку C', минуя все промежуточные точки.

  Допустим, что алгоритм начал путешествие из города P, показанного на рис. 6.10, и, пытаясь найти путь в город Y, проходит через города W, S и T. Нет никаких причин искать перелет из города T в город W, поскольку название города W уже записано в стек. Если бы можно было перелететь из города W в город S, из города S — в город T, затем — обратно в город W, а оттуда — в город Y, то перелеты в города S и T были бы излишни. Поскольку мы накладываем запрет на вторичное посещение города W, алгоритм выполняет откат из городов S и T в город W, а затем следует прямо в город Y. На рис. 6.12 показано состояние стека в двух ситуациях: когда повторное посещение городов разрешено, и при откате, когда повторное посещение городов запрещено.

- Если мы должны перелететь в город C, названия которого в стеке больше нет, — поскольку мы выполнили откат из него и вытолкнули его название из стека, — посещать город C не следует. Это очень непростая ситуация.

  Посмотрим, когда выполняется откат из города. Если откат выполнен, поскольку из города C не выполняются полеты, не имеет смысла посещать

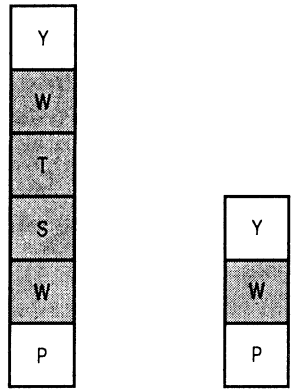


Рис. 6.12. Стек названий городов: а) если повторное посещение городов разрешено; б) при откате, если повторное посещение городов запрещено

этот город вновь. Например, если рейс начинается в городе *P*, показанном на рис. 6.10, и алгоритм выбирает перелет в город *R*, а затем — в город *X*, то из города *X* будет выполнен откат. В этот момент, хотя названия города *X* в стеке больше нет, нет смысла посещать его вновь, поскольку уже известно, что из него никуда улететь невозможно.

Предположим теперь, что из города *C* выполнен откат, поскольку во всех соседних городах мы уже побывали. Это означает, что все возможные перелеты из города *C* уже были рассмотрены и отброшены. Следовательно, нет причины возвращаться в город *C* снова. Допустим, что пунктом отправления алгоритма является город *P*, изображенный на рис. 6.10. Рассмотрим следующую последовательность действий: посещаем город *R*, затем — город *X*, выполняем откат в город *R* (поскольку из города *X* самолеты не вылетают), выполняем откат в город *P* (поскольку во всех соседних городах мы уже побывали), прилетаем в город *W*, перелетаем в город *Y*. В этот момент стек содержит названия городов *P-W-Y*, причем название города *Y* находится на вершине стека, как показано на рис. 6.12, *б*. Теперь нужно выбрать перелет из города *Y*. Город *R* не подходит, поскольку мы уже побывали и в нем, и в соседних городах.

В обоих случаях повторное посещение города ничего не дает и может привести к зацикливанию.

Правило "*никогда не возвращайся в прежние места*" довольно просто реализовать — для этого достаточно помечать города, в которых мы уже побывали. Затем, выбирая кандидата на перелет, следует ограничиться лишь непомеченными городами, соседними с городом, находящимся на вершине стека. Псевдокод алгоритма принимает следующий вид.

Пометка посещенных городов

Второй вариант псевдокода

```
Пометить пункт отправления как
посещенный город
while (последовательность перелетов из пункта отправления
      в пункт назначения не найдена)
{
```

```
    // Инвариант цикла: стек содержит направленный путь
    // из пункта отправления, находящегося на дне стека,
    // в пункт назначения, находящийся на вершине стека

    if (из города, название которого находится на вершине стека,
        вылеты в новые города не совершаются)
        aStack.pop() // Откат
    else
    {
        Выбрать полет из города, название которого находится
        на вершине стека, в город C, который еще не посещался
        aStack.push(C)
        Пометить город C как посещенный
    } // Конец оператора if
} // Конец оператора while
```

Осталось лишь уточнить условие в операторе *while*, т.е. разъяснить, что означает выражение "последовательность перелетов из пункта отправления в пункт назначения не найдена". Инвариант цикла, утверждающий, что стек содержит направленный путь из пункта отправления, находящегося на дне, в пункт назначения, находящийся на вершине, означает, что алгоритм может завершиться успешно, если на вершине стека окажется пункт назначения. Однако алгоритм может выдать отрицательный ответ, если все варианты исчерпаны, т.е. алгоритм выполнил откат к пункту отправления и на карте не осталось городов, в которых мы не побывали. В этот момент название пункта отправления выталкивается из стека, и он становится пустым.

Окончательный вариант алгоритма поиска

```
+searchS(in originCity:City,
      in destinationCity:City):boolean
// Выполняет поиск последовательности
// перелетов из пункта отправления в пункт назначения

    aStack.createStack()
    Снять метки со всех городов

    aStack.push(originCity) // Затолкнуть название пункта
                            // отправления в стек
    Пометить пункт отправления как посещенный город

    while (!aStack.isEmpty() и название пункта назначения
           не находится на вершине стека)
    {
      // Инвариант цикла: стек содержит направленный путь
      // из пункта отправления, находящегося на дне стека,
      // в пункт назначения, находящийся на вершине стека

      if (из города, название которого находится на вершине
          стека, вылеты в новые города не совершаются)
       aStack.pop() // Откат
       else
       {
          Выбрать полет из города, название которого находится
          на вершине стека, в город C, который еще не посещался
          aStack.push(C)
          Пометить город C как посещенный
      }   // Конец оператора if
} // Конец оператора while
```

```
if (aStack.isEmpty())
   return false // Рейса не существует
else
   return true  // Рейс существует
```

В этом алгоритме не указан определенный порядок выбора городов, в которых мы еще не побывали. На самом деле критерий этого выбора не имеет значения, поскольку он не влияет на окончательный результат: последовательность перелетов либо существует, либо нет. Однако этот выбор влияет на конкретные перелеты, которые рассматриваются в ходе алгоритма. Допустим, что алгоритм всегда упорядочивает по алфавиту названия городов, которые еще не посещались. Трассировка, соответствующая такому выбору, показана на рис. 6.13. Здесь пунктом отправления считается город *P*, а пунктом назначения — город *Z*. Алгоритм завершается успешно.

Действие	Причина	Содержимое стека (от дна к началу)
Затолкнуть P	Инициализировать	P
Затолкнуть R	Следующий соседний город, в котором мы еще не были	PR
Затолкнуть X	Следующий соседний город, в котором мы еще не были	PRX
Pop X	Соседних городов, в которых мы еще не были, не существует	PR
Pop R	Соседних городов, в которых мы еще не были, не существует	P
Затолкнуть W	Следующий соседний город, в котором мы еще не были	PW
Затолкнуть S	Следующий соседний город, в котором мы еще не были	PWS
Затолкнуть T	Следующий соседний город, в котором мы еще не были	PWST
Pop T	Соседних городов, в которых мы еще не были, не существует	PWS
Pop S	Соседних городов, в которых мы еще не были, не существует	PW
Затолкнуть Y	Следующий соседний город, в котором мы еще не были	PWY
Затолкнуть Z	Следующий соседний город, в котором мы еще не были	PWYZ

Рис. 6.13. Трассировка алгоритма поиска рейса по карте, изображенной на рис. 6.10

Рассмотрим теперь операции, которые выполняются при поиске рейса. Алгоритм помечает посещенные города, определяет, были ли мы уже в данном городе, и находит соседние города. Карту рейсов можно представить в виде абстрактного типа данных, для которого предусмотрены три указанные выше операции, а также сама операция поиска. Кроме того, в список операций желательно включить запись данных, вставку соседнего города, вывод карты на экран и вывод списка всех соседних городов.

Операции над картой рейсов

```
+createFlightMap()
// Создает пустую карту рейсов

+destroyFlightmap()
// Уничтожает карту рейсов

+readFlightMap(in cityFileName:string,
               in flightFileName:string)
// Считывает информацию о рейсах из карты

+displayFlightMap()
// Выводит на экран информацию о рейсах

+displayAllCities()
// Выводит на экран названия всех городов,
// в которые выполняются авиарейсы компании HPA

+displayAdjacentCities(in aCity:City)
```

```
// Выводит на экран все города,
// соседние по отношению к данному городу

+markVisited(in aCity:City)
// Помечает город как посещенный

+unvisitAll()
// Снимает метки со всех городов

+isVisited(in aCity:City):boolean
// Определяет, посещался ли город ранее

+insertAdjacent(in aCity:City, in adjCity:City)
// Вставляет в карту рейсов следующий соседний город

+getNextCity(in fromCity:City, out nextCity:City):boolean
// Определяет следующий город, который ранее не посещался,
// если он существует, т.е. соседний по отношению к данному
// городу. Возвращает значение true, если найден соседний
// город, который еще не посещался, в противном случае
// возвращает значение false

+isPath(in originCity:City, in destinationCity:City):boolean
// Определяет, существует ли последовательность
// рейсов из одного города в другой
```

Ниже приведена функция, реализующая операцию *isPath* с помощью алгоритма *searchS*. Предполагается, что класс *Stack* реализует операции над стеком, а класс *Map* — операции над картой полетов. Для повышения эффективности города изображаются целыми числами.

Реализация функции searchS на языке C++

```
bool Map::isPath(int originCity, int
destinationCity)
// ----------------------------------
// Определяет, существует ли последовательность рейсов между
// двумя городами. Используется итеративная версия стека.
// Предусловие: аргументы originCity и destinationCity
// представляют собой пункт оправления и пункт назначения.
// Постусловие: если последовательность рейсов из города
// originCity в город destinationCity существует, возвращает
// значение true; в противном случае возвращает значение false.
// Города, посещенные во время поиска, помечаются.
// Примечания: используется стек целых чисел, которые
// обозначают города. Вызываются функции unvisitAll,
// markVisited и getNextCity.
// ----------------------------------------------------
{
   Stack aStack;
   int topCity, nextCity;
   bool success;
   unvisitAll(); // Снять пометки со всех городов

   // Затолкнуть пункт отправления в стек и пометить его как
   // посещенный город.
   aStack.push(originCity);
   markVisited(originCity);

   aStack.getTop(topCity);
```

```
    while (!aStack.isEmpty() && (topCity != destinationCity))
    {
      // Инвариант цикла: стек содержит направленный путь
      // из пункта отправления, находящегося на дне стека,
      // в пункт назначения, находящийся на вершине стека

      // Найти город, который является соседним по отношению
      // к вершине стека и еще не посещался
      success = getNextCity(topCity, nextCity);
      if (!success)
          aStack.pop(); // Город не найден! Откат
      else // Посетить город
      {
          aStack.push(nextCity);
          markVisited(nextCity);
      } // Конец оператора if

      aStack.getTop(topCity);
    } // Конец оператора while

    if (aStack.isEmpty())
        return false; // Пути не существует
    else
        return true;  // Путь существует
} // Конец функции isPath
```

Полное решение этой задачи читатели могут завершить самостоятельно, выполнив задание 9, приведенное в разделе "Задания по программированию".

Рекурсивное решение

Вспомним, как мы пытались вручную решить задачу о перелетах из одного города в другой. Начиная с пункта отправления C_0 мы выбирали произвольный рейс, начинающийся в этом городе. Этот рейс приводил нас в новый город, C_1. Если город C_1 являлся пунктом назначения, задача была решена; в противном случае нужно было продолжить поиск подходящего рейса, начиная с города C_1. Это приводило нас в город C_2. Если город C_2 являлся пунктом назначения, задача была решена; в противном случае нужно было продолжить поиск очередного рейса, начиная с города C_2, и т.д. Эта задача имеет отчетливо выраженный рекурсивный характер.

Рекурсивную стратегию поиска можно сформулировать так.

Рекурсивная стратегия поиска

```
Чтобы перелететь из пункта отправления в пункт назначения,

   выберите город C, соседний с пунктом отправления
   if (город C — пункт назначения)
      Задача решена
   else
      Найдите путь из города C в пункт назначения
```

Такое описание задачи наглядно демонстрирует ее рекурсивный характер. На первом шаге выполняется перелет из пункта отправления в город C. Прибыв в город C, мы сталкиваемся с той же проблемой — нужно перелететь отсюда в пункт назначения.

Рекурсивное описание представляет собой нечто большее, чем просто иную формулировку задачи. Эта стратегия может иметь три исхода.

Возможные исходы рекурсивной стратегии

1. Пункт назначения достигнут, следовательно, искомый маршрут существует.
2. Рейс заканчивается в городе *C*, из которого авиакомпания вылеты не совершает.
3. Произошло зацикливание.

Первый из перечисленных исходов соответствует базису рекурсии. Если мы достигли пункта назначения, задача решена, и выполнение алгоритма прекращается. Однако, как указывалось выше, алгоритм может оказаться безрезультатным, т.е. базовая задача никогда не возникнет. Алгоритм может привести нас в город *C*, из которого не вылетает ни один самолет нашей авиакомпании. (Обратите внимание, что в этом случае алгоритм не определяет никаких действий, т.е. в этом смысле он неполон.) Кроме того, в ходе выполнения алгоритма может произойти зацикливание, и он никогда не завершится.

Решить эти проблемы можно, используя принцип зеркального отражения, который мы применяли в предыдущем случае. Рассмотрим следующее уточнение алгоритма, в котором все посещенные города отмечались на карте и никогда не посещались дважды.

Уточнение алгоритма рекурсивного поиска

```
+searchR(in originCity:City,
         in destinationCity:City):boolean
// Поиск последовательности перелетов
// из города originCity в город destinationCity.

   Пометить пункт отправления как посещенный город

   if ( город originCity является пунктом назначения)
      Прекратить выполнение алгоритма — задача решена
   else
      for (каждый соседний город еще не посещавшийся C)
          searchR(C, destinationCity)
```

Посмотрим, что произойдет, когда мы окажемся в городе, у которого все соседние города нами уже посещались. Рассмотрим фрагмент карты полетов, изображенной на рис. 6.14. Когда алгоритм *searchR* приводит нас в город *X*, т.е. аргументом функции *originCity* является значение *X*, цикл *for* игнорируется, так как у города *X* нет ни одного соседа, где мы уже не побывали бы. Следовательно, функция *searchR* возвращает управление в вызывающий модуль. Это действие аналогично откату в город *W*, из которого мы прибыли в город *X*. Пользуясь терминами предыдущего псевдокода, можно сказать, что управление передается в точку, из которой был сделан вызов *searchR(X, destinationCity)*. Эта точка находится внутри цикла *for*, который перебирает все еще не посещенные города, соседние с городом *W*, т.е. аргументом функции *originCity* является значение *W*.

После отката из города *X* в город *W* цикл *for* выполняется снова. На этот раз выбирается город *Y*, являющийся результатом рекурсивного вызова *searchR(V, destinationCity)*. В этот момент алгоритм либо достигает пункта назначения и прекращает работу, либо снова выполняет откат в город *W*. При откате в город *W* выполнение цикла *for* будет прекращено, поскольку у города *W* больше не осталось соседей, где мы еще не побывали. Выполняется возврат управления из функции *searchR*. В результате будет выполнен откат в город,

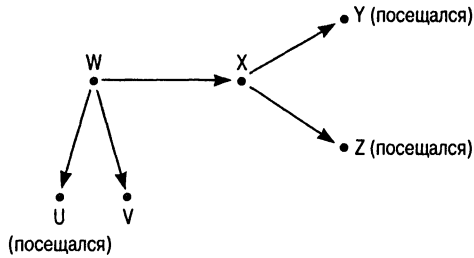


Рис. 6.14. Фрагмент карты полетов

из которого мы прибыли в город *W*. Если алгоритм выполнит откат в исходный пункт отправления, у которого не окажется ни одного соседа, где мы еще не были, то выполнение алгоритма будет прекращено, поскольку в таком случае маршрута из пункта отправления в пункт назначения не существует. Обратите внимание, что описанный выше алгоритм рано или поздно завершается, так как он либо достигает пункта назначения, либо прекращает перебор.

Рассмотрим функцию на языке C++, которая реализует алгоритм *searchR*.

Реализация алгоритма searchR на языке C++

```
bool Map::isPath(int originCity,
                 int destinationCity)
{
   int nextCity;
   bool success, done;

   // Отметить текущий город как посещенный
   markVisited(originCity);

   // Базис: достигнут пункт назначения
   if (originCity == destinationCity)
      return true;

   else // Проверить перелеты во все не посещенные города
   {
      done = false;
      success = getNextCity(originCity, nextCity);

      while (success && !done)
      {
          done = isPath(nextCity, destinationCity);
          if (!done)
             success = getNextCity(originCity, nextCity);
      } // Конец оператора while
      return done;
   } // Конец оператора if
} // Конец функции isPath
```

Возможно, вы уже заметили большую схожесть между алгоритмами *searchR* и *searchS*. Фактически эти два алгоритма просто используют разные способы реализации одной и той же стратегии поиска. В следующем разделе мы подробнее изучим взаимосвязь между этими алгоритмами.

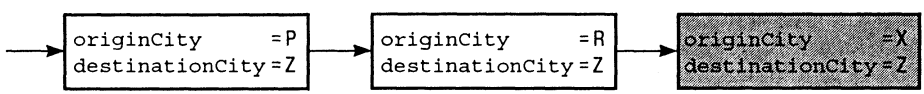
Взаимосвязь между стеками и рекурсией

В предыдущем разделе мы решили задачу о перелетах, используя абстрактный стек и рекурсию. В этом разделе мы покажем, как связаны между собой способ организации стека и реализация поиска в рекурсивном алгоритме. Мы покажем, что концепция стека неявно использует рекурсию и что стеки играют важную роль в компьютерной реализации рекурсии.

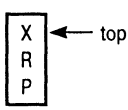
Рассмотрим, как два алгоритма поиска, описанных выше, реализуют три ключевых аспекта их общей стратегии.

Сравнение ключевых аспектов двух алгоритмов поиска

- **Посещение нового города.** Рекурсивный алгоритм *searchR* посещает новый город *C*, вызывая функцию *searchR(C, destinationCity)*. Алгоритм *searchS* посещает город *C*, заталкивая его название в стек. Если для трассировки вызова *searchR(C, destinationCity)* применить метод блок-схем, то город *C* окажется связан с формальным аргументом *originCity* функции *searchR*.
- Например, на рис. 6.15 показаны результаты трассировки и состояние стека при выполнении функции *searchS* в соответствующей точке маршрута из города *P* в город *Z* (см. рис. 6.10).
- **Откат.** Оба алгоритма поиска перебирают соседние города, которые еще не посещались. Обратите внимание, что текущий город задается формальным аргументом *originCity* в наиболее глубоком (самом правом) блоке схемы трассировки функции *searchR*. Аналогично, текущий город расположен на вершине стека функции *searchS*. Если у текущего города нет соседей, которые еще не посещались, алгоритм должен выполнить откат в предыдущий город. В алгоритме *searchR* откат выполняется с помощью возврата управления из рекурсивной функции. На блок-схеме это действие изображается путем перечеркивания наиболее глубоко вложенного блока. Алгоритм *searchS* выполняет откат, явно выталкивая название города из стека. Например, состояние, изображенное на рис. 6.15, соответствует откату алгоритмов в город *R*, а затем — в город *P* (рис. 6.16).
- **Прекращение работы алгоритма.** Алгоритм поиска прекращает свою работу, если достигнут пункт назначения или исчерпаны варианты перебора. Вторая ситуация возникает, если после отката в пункт отправления не осталось ни одного соседнего города, где мы еще не были. При этом на блок-схеме функции *searchR* все блоки окажутся перечеркнутыми, а управление будет возвращено в точку первоначального вызова функции. Для функции *searchS* отсутствие еще не посещенных соседних городов приводит к полному опустошению стека.



а) Трассировка



б) Стек

Рис. 6.15. Посещаем город P, затем R, затем X: а) результаты трассировки; б) стек

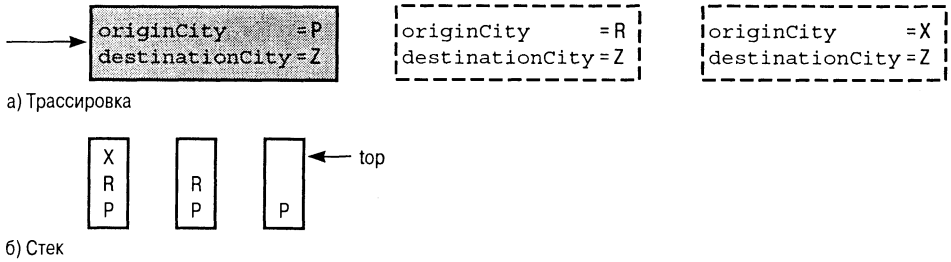


Рис. 6.16. Откат из города X в город R, а затем — в город P: а) трассировка; б) стек

Итак, два алгоритма поиска, описанные выше, действительно выполняют одинаковые действия. Если они применяют одни и те же правила выбора не посещенных городов, то они проходят через одни и те же города в одном и том же порядке. Это совпадение отнюдь не случайно. На самом деле действия, выполняемые рекурсивной функцией, всегда можно зафиксировать в стеке.

Обычно для реализации рекурсивных функций используется стек

Тесная связь между стеками и рекурсией наиболее ярко проявляется при компьютерной реализации рекурсивных функций. Обычно для реализации рекурсивных функций компиляторы используют стек, который очень напоминает блок-схему. При вызове рекурсивной функции в соответствующем блоке нужно запоминать определенную информацию о локальном окружении — значения аргументов и локальных переменных, — а также ссылку на точку, из которой был сделан рекурсивный вызов.

Каждый рекурсивный вызов генерирует новую запись активации, которая заталкивается в стек

В ходе выполнения программа должна управлять этими блоками информации, которые называются **записями активации** (activation records), или **активационными записями**. Для манипуляций с этими записями в стеке необходимо предусмотреть отдельные операции. При каждом рекурсивном вызове создается активационная запись, которая заталкивается в стек. Это действие соответствует созданию нового блока в наиболее глубоко вложенной точке последовательности вызовов. Затем выполняется возврат управления из рекурсивной функции, и активационная запись, содержащая соответствующие переменные окружения, перемещается на вершину стека. Это действие соответствует перечеркиванию наиболее глубоко вложенного блока и переходу на предыдущий блок. На стеках активационных записей основаны многие компьютерные реализации рекурсии.

Стеки можно применять и в итеративных реализациях рекурсивных алгоритмов

Аналогичную стратегию можно применять и в итеративных реализациях рекурсивных алгоритмов. Нужно лишь преобразовать рекурсивный алгоритм в итеративный. Способы таких преобразований изложены в предыдущих главах.

Резюме

1. Операции над абстрактным стеком основаны на принципе "последним вошел — первым вышел" (LIFO).
2. Одним из наиболее важных приложений стеков являются алгоритмы вычисления алгебраических выражений. Принцип LIFO лучше всего соответствует природе алгоритма вычисления постфиксных выражений. Кроме того, стеки применяются для преобразования инфиксных выражений в пост-

фиксные, что позволяет отменить приоритеты операторов, правила ассоциативности и не использовать скобки.

3. Стек можно применять для решения задачи о поиске пути из одного города в другой. В стеке можно хранить последовательность посещенных городов. Кроме того, стеки позволяют легко выполнять откат. Однако вывести на экран маршрут из одного города в другой нелегко, поскольку название пункта отправления оказывается на дне стека, а пункт назначения — на вершине.
4. Между рекурсией и стеками существует тесная связь. Большинство реализаций рекурсии основано на стеках активационных записей, которые очень похожи на блок-схемы.

Предупреждения

1. Если стек пуст, операции *getTop* и *pop* должны выполнять разумные действия. Например, они могут генерировать исключительную ситуацию *StackException*.
2. Алгоритмы вычисления инфиксных выражений или преобразования их в постфиксные должны распознавать операнды. Для этого достаточно учесть приоритеты операторов и правила их ассоциативности, проигнорировав скобки.
3. При поиске последовательности перелетов из одного города в другой следует учесть возможность ошибочного выбора. Например, алгоритм должен выполнять откат из тупика и предотвращать зацикливание.

Вопросы для самопроверки

1. Допустим, вы затолкнули в стек буквы A, B, C и D, а затем выталкиваете их оттуда. В каком порядке они будут удаляться из стека?
2. Опишите состояние стеков *stack1* и *stack2* после выполнения следующей последовательности операций.

```
stack1.push(1)
stack1.push(2)
stack2.push(3)
stack2.push(4)
stack1.pop()
stack2.getTop(stackTop)
stack1.push(stackTop)
stack1.push(5)
stack2.pop(stackTop)
stack2.push(6)
```

3. Алгоритмы, описанные в разделе "Простые применения абстрактного стека" работают со строками. При каких условиях стек, использованный в этих алгоритмах, можно реализовать в виде массива? При каких условиях следует предпочесть связанный список?
4. Перечислите изменения, которые необходимо произвести для преобразования программы, использующей реализацию стека в виде массива, в программу, работающую со стеком в виде связанного списка.

5. Для каждой из указанных строк выполните трассировку алгоритма проверки баланса фигурных скобок и покажите состояние стека на каждом шаге.

   5.1. `x{{yz}}`

   5.2. `{x{y{{z}}}}`

   5.3. `{{{x}}}`

6. Примените алгоритмы, описанные в этой главе, для вычисления постфиксного выражения *ab-c+*. Допустим, что идентификаторы имеют следующие значения: *a*=7, *b*=3, *c*=−2. Покажите состояние стека после каждого шага.

7. Примените алгоритмы, описанные в этой главе, для преобразования инфиксного выражения *a/b*c* в постфиксную форму. Учтите правила ассоциативности операторов. Покажите состояние стека после каждого шага.

8. Обоснуйте важность соблюдения приоритетов при преобразовании инфиксных выражений в постфиксные. Почему используется проверка условия "больше или равно", а не просто "больше"?

9. Выполните алгоритм поиска маршрута перелетов для карты, показанной на рис. 6.17, для каждого из указанных вариантов. Покажите состояние стека после каждого шага.

   9.1. Перелет из пункта A в пункт B.

   9.2. Перелет из пункта A в пункт D.

   9.3. Перелет из пункта C в пункт G.

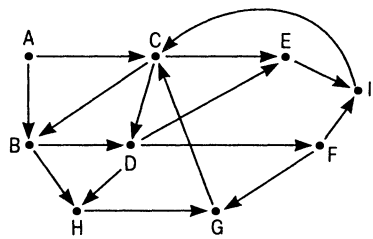


Рис. 6.17. Карта полетов для вопроса 9 и упражнения 11

Упражнения

1. Допустим, что у нас есть стек `aStack` и пустой вспомогательный стек `auxStack`. Как выполнить каждое из перечисленных ниже заданий, пользуясь лишь операциями над абстрактным стеком?

   1.1. Вывести на экран содержимое стека `aStack` в обратном порядке, т.е. вершина выводится последней.

   1.2. Определить количество элементов в стеке `aStack`, оставив его неизменным.

   1.3. Удалить все вхождения заданного элемента в стеке `aStack`, оставив порядок следования его элементов неизменным.

2. Диаграмма железнодорожной стрелки, изображенная на рис. 6.18, часто используется для иллюстрации концепции стека. Укажите три стека, изображенных на рисунке, и раскройте взаимосвязи между ними. Как применить эту стрелку для выполнения всевозможных переключений?

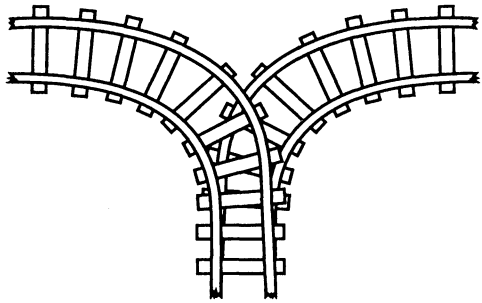


Рис. 6.18. Железнодорожная стрелка из упражнения 2

3. Операция вывода на экран содержимого стека может оказаться полезной при отладке программы. Добавьте метод *display* в список операций над абстрактным стеком, учитывая следующие варианты условий

   3.1. Метод должен использовать только операции над абстрактным стеком, иными словами, не зависеть от реализации стека.

   3.2. Метод должен использовать реализацию стека в виде связанного списка.

4. Оцените эффективность реализации абстрактного стека в виде абстрактного списка при условии, что абстрактный список реализуется в виде массива.

5. В разделе "Разработка абстрактных типов данных в процессе решения задачи" описан алгоритм *readAndCorrect*, предназначенный для считывания строки символов и последующего исправления опечаток.

   5.1. Выполните трассировку алгоритма *readAndCorrect* на примере строки `abc←de←←fg←h` и покажите состояние стека на каждом шаге.

   5.2. Содержимое стека легко выводить на экран в обратном порядке, но сложно — в прямом. Напишите псевдокод алгоритма, выводящего на экран строку, записанную в прямом порядке.

   5.3. Реализуйте алгоритм *readAndCorrect* в виде функции на языке C++. Стек должен быть локальным и не передаваться в качестве аргумента. Функция должна создавать строку, содержащую правильно введенные символы в прямом порядке, и возвращать ее в качестве результата.

6. Примените решение задачи о балансе фигурных скобок для выражений, содержащих три типа разделителей — `()`, `[]` и `{}`. Таким образом, выражение `{ab(c[d])e}` является правильным, а выражение `{ab((c)` — нет.

7. Для каждой из перечисленных ниже строк выполните трассировку алгоритма распознавания выражений, описанного в разделе "Распознавание строк языка". Покажите состояние стека на каждом шаге.

   7.1. `xx$xy`

   7.2. `xy$x`

   7.3. `y$yx`

   7.4. `xx$xx`

   7.5. `xy$y`

8. Напишите псевдокод функции, использующей стек, для определения принадлежности строки языку *L*.

8.1. $L=\{w$, где строка w содержит одинаковое количество букв A и B$\}$

8.2. $L=\{w$, где строка w имеет вид $A^nB^n\}$

9. Вычислите указанные постфиксные выражения, используя алгоритм, описанный в этой главе. Покажите состояние стека на каждом шаге. Предполагается, что идентификаторы имеют следующие значения: *a=7, b=3, c=12, d=-5, e=1*.

9.1. *abc+-*

9.2. *abc-d*+*

9.3. *ab+c-de*+*

10. Преобразуйте указанные инфиксные выражения в постфиксные. Покажите состояние стека на каждом шаге.

10.1. *a-b+c*

10.2. *a/(b*c)*

10.3. *(a+b)*c*

10.4. *a-(b+c)*

10.5. *a-(b/c*d)*

10.6. *a/b/c-(d+e)*f*

10.7. *a* (b/c/d) +e*

10.8. *a- (b+c*d) /e*

11. Выполните алгоритм определения маршрута по карте, изображенной на рис. 6.17 (см. вопрос 9), при указанных условиях. Покажите состояние стека на каждом шаге.

11.1. Перелет из пункта A в пункт F.

11.2. Перелет из пункта D в пункт A.

11.3. Перелет из пункта A в пункт G.

11.4. Перелет из пункта I в пункт G.

11.5. Перелет из пункта F в пункт H.

12. Как указывалось в главе 3, операции над абстрактным списком можно выразить с помощью аксиом. Например, указанные ниже аксиомы позволяют формально определить абстрактный стек. Здесь *aStack* — произвольный стек, а *item* — произвольный элемент стека. Для простоты предполагается, что функция *getTop* в качестве результата возвращает вершину стека.

```
(aStack.createStack()).isEmpty() = true
(aStack.push(item)).isEmpty() = false
(aStack.createStack()).pop() = error
(aStack.push(item)).pop() = aStack
(aStack.createStack()).getTop() = error
(aStack.push(item)).getTop() = item
```

С помощью этих аксиом можно доказать, что стек, определенный последовательностью операций

```
Создать пустой стек
Push 5
```

```
Push 7
Push 3
Pop (3)
Push 9
Push 4
Pop (4)
```

которую можно переписать в более сжатом виде

```
(((((((aStack.createStack()).push(5)).push(7)).push(3)).
pop()).push(9)).push(4)).pop()
```

эквивалентен стеку, определенному последовательностью операций

```
Создать пустой стек
Push 5
Push 7
Push 9
```

которую в свою очередь можно кратко записать в виде

```
(((aStack.createStack()).push(5)).push(7)).push(9)
```

Аналогично, аксиомы позволяют доказать, что выражение

```
(((((((aStack.createStack()).push(1)).push(2)).pop()).
push(3)).pop()).pop()).isEmpty()
```

является истинным.

12.1. Приведенное ниже представление стека с помощью последовательности, состоящей из операций *push* и не содержащей ни одной операции *pop*, называется **каноническим** (canonical).

```
(···(aStack.createStack()).push()).push()) ··· ).push()
```

Докажите, что любой стек можно представить в каноническом виде.

12.2. Докажите, что каноническое представление стека является единственным. Иными словами, для любого стека существует только один эквивалентный стек, записанный в каноническом виде.

12.3. Используя систему аксиом, докажите, что значение выражения

```
(((((((((aStack.createStack()).push(6)).push(9)).
pop()).pop()).push(2)).pop()).push(3)).push(1)).
pop()).stackTop()
```

равно 3.

13. Деструктор, определенный в реализации абстрактного стека в виде связанного списка, вызывает функцию *pop*. Этот деструктор не эффективен, поскольку вызывает функцию *pop* несколько раз. Напишите другую реализацию деструктора, которая удаляет связанный список, не вызывая функции *pop*.

Задания по программированию

1. Напишите реализацию абстрактного стека, использующую динамические массивы. При заполнении стека увеличьте размер массива вдвое.

2. В разделе "Распознавание строк языка" описан алгоритм распознавания выражений языка

   $L=\{w\$w'$, где строка w может быть пустой или содержать символы, отличные от $, а строка w'=reverse(w)$\}$.

   Реализуйте этот алгоритм.

3. Разработайте и реализуйте класс постфиксных калькуляторов. Используйте алгоритм вычисления постфиксных выражений, описанный в этой главе. Допускаются лишь операторы +, -, * и /. Предполагается, что постфиксные выражения являются корректными.

4. Проанализируйте простое инфиксное выражение, состоящее из операндов, представляющих собой одну цифру, операторов +, -, * и /, а также скобок. Унарные операторы не допускаются. Выражение не должно содержать пробелы.

   Разработайте и реализуйте класс инфиксных калькуляторов. Используйте алгоритм вычисления инфиксных выражений, описанный в этой главе. Перед вычислением инфиксное выражение следует преобразовать в постфиксную форму, а затем вычислить полученное постфиксное выражение.

5. В алгоритме преобразования инфиксных выражений в постфиксные, описанном в этой главе, предполагалось, что задаваемые инфиксные выражения являются синтаксически правильными. Устраните это ограничение и выполните задание 4.

6. Повторите задание 4, используя следующий алгоритм вычисления инфиксного выражения *infixExp*. Алгоритм использует два стека: стек *opStack* содержит операторы, а стек *valStack* — операнды и промежуточные результаты. Обратите внимание, что этот алгоритм рассматривает скобки как операторы, имеющие самый низкий приоритет.

```
for (каждый символ ch в строке infixExp)
{
   switch (ch)
   {
     case символ ch является операндом, т.е. цифрой
          valStack.push(ch)
          break
     case символ ch равен '('
          opStack.push(ch)
          break
     case символ ch является оператором
        if (opStack.isEmpty())
            opStack.push(ch)
        else if (precedence(ch) > precedence(вершина pStack))
            opStack.push(ch)
        else
        {
            while (!opStack.isEmpty() и
                precedence(ch) <= precedence(вершина opStack))
            Выполнить
            opStack.push(ch)
        } // Конец оператора if
        break
     case символ ch равен ')'
        while (вершина стека opStack не равна '(')
           Execute
          opStack.pop()
```

```
            break
        } // Конец оператора switch
    } // Конец оператора for
    while (!opStack.isEmpty())
        Execute
    valStack.getTop(result)
```

Операция *Execute* означает:

```
valStack.pop(operand2)
valStack.pop(operand1)
opStack.Pop(op)
result = operand1 op operand2
valStack.push(result)
```

Выберите для реализации этого алгоритма один из двух подходов.

- Стек операторов *opStack* содержит символы, а стек операндов *valStack* — целые числа.
- Стек операторов *opStack* содержит целые числа, символизирующие собой операторы, так что оба стека содержат целые числа.

(В задании 12 из главы 8 предлагается решить эту задачу, пользуясь шаблонными классами.)

7. В алгоритме вычисления инфиксных выражений, описанном в задании 6, предполагалось, что задаваемые инфиксные выражения являются синтаксически правильными. Устраните это ограничение и выполните задание 6 снова.
8. Используя стеки, напишите итеративную версию функции *solveTowers*, определенную в главе 2.
9. Завершите решение задачи о нахождении авиарейсов. Входную информацию разместите в трех текстовых файлах.

CityFile	В каждой строке записано название города, в который летают самолеты авиакомпании НРА
FlightFile	В каждой строке записана пара городов, представляющих собой пункты отправления и пункты назначения авиарейсов компании НРА
RequestFile	В каждой строке записана пара городов, представляющих собой запрос на полет из указанного пункта отправления в заданный пункт назначения

Можно сделать следующие предположения.

- Название каждого города не превышает 15 символов. Названия городов, указанных в паре, разделяются скобками.
- Авиакомпания НРА обслуживает не больше 20 городов.
- Входные данные являются корректными.

Например, входные файлы могут содержать следующие записи.

```
cityFile:       Альбукерк
                Чикаго
                Сан-Диего
flightFile:     Чикаго,     Сан-Диего
                Чикаго,     Альбукерк
                Альбукерк,  Чикаго
```

```
requestFile:  Альбукерк, Сан-Диего
              Альбукерк, Париж
              Сан-Диего, Чикаго
```

Получив эту информацию, программа должна выдать следующий результат.

```
Заказан полет из Альбукерка в Сан-Диего.
Компания HPA осуществляет полеты из Альбукерка в Сан-Диего.

Заказан полет из Альбукерка в Париж.
Извините. Самолеты компании HPA в Париж не летают.

Заказан полет из Сан-Диего в Чикаго.
Извините. Самолеты компании HPA не летают из Сан-Диего в  Чикаго.
```

Начните с реализации абстрактного типа данных "Карта полетов" в виде класса *Map* на языке C++. Используйте итеративную версию функции *isPath*. Поскольку основной операцией алгоритма является функция *getNextCity*, необходимо создать ее эффективную реализацию. Если города пронумерованы целыми числами *1*, *2*, . . . , *N*, для представления карты можно использовать *N* связанных списков. Узел, представляющий город *j*, вставляется в список города *i* тогда и только тогда, когда из города *i* в город *j* есть направленный путь. Такая структура данных называется **списком смежности** (adjacency list). На рис. 6.19 показан список смежности для карты полетов, изображенной на рис. 6.10. В главе 13 списки смежности обсуждаются как один из способов представления графов.

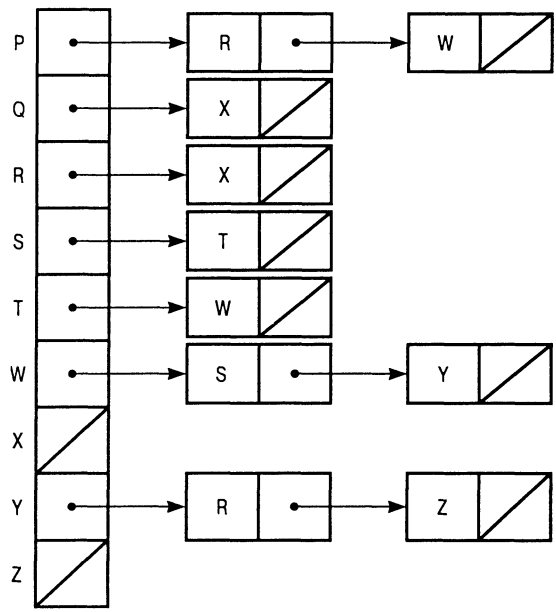


Рис. 6.19. Список смежности для карты полетов, изображенной на рис. 6.10.

Хотя списки смежности можно создавать "с нуля" , используйте *N* экземпляров класса *List*.

Для повышения эффективности название каждого города можно закодировать целым числом. Это позволит считывать названия городов в массив *namesOfCites*. Затем на город *i* можно будет ссылаться через элемент *namesOfCites[i]*. Эта схема позволяет хранить в стеке целые числа, а не строки. Таким образом, основными структурами данных, используемыми при решении задачи о перелетах, является массив *namesOfCities* и АТД "Карта полетов".

Поскольку для идентификации городов используются целые числа, необходимо предусмотреть способ преобразования кода города в его название. Для этого в АТД "Карта полетов" нужно предусмотреть следующие операции.

```
+cityName(in number:integer):string
// Определяет название города по его номеру

+cityNumber(in name:string):integer
// Определяет номер города по его названию
```

Поскольку названия в файле *cityFile* должны быть упорядочены по алфавиту, массив *nameofCities* должен быть упорядочен по возрастанию. Это позволяет применять бинарный поиск заданного названия в массиве, а затем определять его номер.

Для упрощения процедуры считывания текстовых файлов определите класс, содержащий следующие функции-члены.

```
+getName(out name:string)
// Считывает название из следующей строки текстового файла

+getNamePair(out name1:string, out name2:string)
// Считывает два названия из следующей строки текстового файла
```

10. При решении задачи о перелетах (задание 9) поиск следующего не посещенного города, соседнего по отношению к городу *i*, всегда начинается с начала *i*-го связанного списка. Это не очень эффективно, поскольку один и тот же город дважды не посещается. Модифицируйте программу, так чтобы поиск следующего города начинался с предыдущего. Для этого придется предусмотреть массив указателей *tryNext* на списки смежности.

11. Решите расширенный вариант задачи о перелетах. Кроме пункта отправления и пункта назначения входная информация должна содержать номер рейса и стоимость билета (целые числа). Модифицируйте программу так, чтобы она выдавала на экран весь маршрут, соответствующий запросу, включая номера каждого рейса, стоимость каждого билета и общую стоимость путешествия.

Например, входные файлы могут содержать следующие записи.

```
cityFile:       Альбукерк
                Чикаго
                Сан-Диего
flightFile:     Чикаго,     Сан-Диего    703    325
                Чикаго,     Альбукерк    111    250
                Альбукерк,  Чикаго       178    250
requestFile:    Альбукерк,  Сан-Диего
                Альбукерк,  Париж
                Сан-Диего,  Чикаго
```

Получив эту информацию, программа должна выдать следующий результат.

```
Заказан полет из Альбукерка в Сан-Диего.
Рейс #178 из Альбукерка в Сан-Диего. Стоимость: $250
Рейс #703 из Чикаго в Сан-Диего.     Стоимость: $325
Общая стоимость: ................. $575

Заказан полет из Альбукерка в Париж.
Извините. Самолеты компании HPA в Париж не летают.

Заказан полет из Сан-Диего в Чикаго.
Извините. Самолеты компании HPA не летают из Сан-Диего в  Чикаго.
```

Когда итеративная функция *isPath* находит последовательность авиарейсов из пункта отправления в пункт назначения, в стеке содержится весь маршрут. Камнем преткновения для вывода этого пути на экран является обратный порядок записи городов в стеке. Пункт назначения находится на вершине стека, а пункт отправления — на дне. Например, если программа используется для нахождения пути из города *P* в город *Z* (см. рис. 6.10), то в стеке будет записана последовательность символов *P-W-Y-Z*, причем символ *Z* окажется на вершине. Естественно потребовать, чтобы символ *P* был выведен на экран первым, однако он находится на дне стека. Если ограничиться только операциями над стеком, придется использовать вспомогательный стек, вытолкнув в него содержимое основного стека, а затем вытолкнуть все элементы из вспомогательного стека. Эту процедуру для каждого города придется выполнять дважды.

Очевидно, что стек — не самое удобное средство для записи маршрута в правильном порядке, для этого больше подходит **прослеживаемый стек** (traversable stack). Кроме стандартных операций над стеком, *isEmpty*, *push*, *pop* и *getTop*, прослеживаемый стек предусматривает операцию *traverse*. Эта операция начинается на одном из концов стека и просматривает каждый элемент стека, пока не достигнет противоположного конца. В рамках данного проекта следует предусмотреть операцию *traverse*, начинающуюся со дна стека и выполняющую перемещение к его вершине.

12. Какие изменения нужно внести в решение задачи 11, чтобы находить наиболее дешевый маршрут для каждого запроса? Как учесть время полета?

Очереди

В этой главе ...

Абстрактная очередь

Некоторые применения абстрактной очереди
- Считывание строки символов
- Распознавание палиндромов

Реализация абстрактной очереди
- Реализация очереди в виде связанного списка
- Реализация очереди в виде массива
- Реализация очереди с помощью абстрактного списка
- Шаблонный класс queue из библиотеки STL
- Сравнение реализаций

Абстрактные типы данных, основанные на позиционном принципе

Приложение: моделирование

Резюме

Предупреждения

Вопросы для самопроверки

Упражнения

Задания по программированию

Введение. В отличие от стека, организованного по принципу "последним вошел — первым вышел", очередь функционирует по принципу "первым вошел — первым вышел". В этой главе вводятся определения операций над очередью и рассматриваются стратегии их реализации. Как мы убедимся, очереди часто встречаются в повседневной жизни. Принцип, лежащий в их основе, полностью соответствует ситуациям, при которых возникает ожидание. Очереди играют важную роль в моделировании, позволяя анализировать поведение сложных систем.

Абстрактная очередь

Принцип FIFO: элемент, первым поставленный в очередь, покидает ее раньше всех.

Очередь (queue) напоминает людей, выстроенных в ряд. Человек, стоящий первым, обслуживается раньше других, а затем покидает очередь. Новые люди могут становиться только в **конец** (back) , или **хвост** (rear) очереди. Операции над очередью выполняются только с концов. Это поведение полностью соответствует принципу "первым вошел — первым вышел". В противоположность этому можно считать, что стек имеет только один конец, поскольку все операции над ним выполняются на его вершине. Такое функционирование стека описывается правилом "последним вошел — первым вышел".

Над абстрактной очередью выполняются следующие операции.

ОСНОВНЫЕ ПОНЯТИЯ

Операции над абстрактной очередью

1. Создать пустую очередь.
2. Уничтожить очередь.
3. Определить, пуста ли очередь.
4. Добавить в очередь новый элемент.
5. Удалить из очереди элемент, поставленный туда раньше всех.
6. Извлечь из очереди элемент, поставленный туда раньше всех.

Очереди возникают в повседневной жизни

Очереди возникают во многих реальных ситуациях. Люди стоят в очереди очень часто, например, покупая билет в кинотеатр, оплачивая купленные книги в кассе или пользуясь банкоматами. Первыми обслуживаются клиенты, стоящие в начале очереди, а вновь прибывающие люди становятся в ее конец. Даже резервируя авиабилет, вы становитесь в очередь к кассиру.

Очереди и компьютерные науки

Очереди широко применяются в компьютерных науках. Когда вы печатаете текст, компьютер пересылает его строки на принтер. Скорость этого процесса намного выше, чем скорость работы принтера. Таким образом, строки текста выстраиваются в очередь на принтер, который извлекает их оттуда по принципу FIFO. Если принтер одновременно используется несколькими компьютерами, они также образуют очередь.

Поскольку все эти операции связаны с ожиданием, основным предметом их изучения стали способы сокращения времени ожидания в очереди. Раздел компьютерных наук, который занимается этими вопросами, называется **моделированием** (simulations). Позднее мы рассмотрим моделирование очереди клиентов в банке.

Приведенный ниже псевдокод более подробно описывает операции над абстрактной очередью. UML-диаграмма класса `Queue` показана на рис. 7.1. Как и для стека, здесь предусмотрены операции извлечения и дальнейшего удаления элемента из головы очереди.

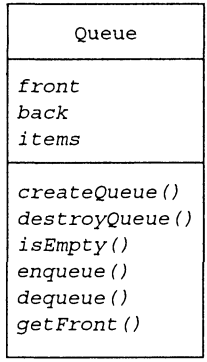


Рис. 7.1. UML-диаграмма класса Queue

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операции над абстрактной очередью

```
// QueueItemType — тип элементов, поставленных в очередь

+createQueue()
// Создает пустую очередь

+destroyQueue()
// Уничтожает очередь

+isEmpty():boolean {query}
// Определяет, пуста ли очередь

+enqueue(in newItem:QueueItemType) throw QueueException
// Вставляет элемент newItem в конец очереди. Если вставка
// невозможна, генерирует исключительную ситуацию QueueException

+dequeue() throw QueueException
// Удаляет голову очереди, т.е. элемент, поставленный в нее
// раньше всех. Если удаление невозможно, генерирует
// исключительную ситуацию QueueException

+dequeue(out queueFront:QueueItemType) throw QueueException
// Извлекает первый элемент в переменную queueFront, а затем
// удаляет его из очереди. Иными словами, извлекает и удаляет
// голову очереди. Если удаление невозможно, генерирует
// исключительную ситуацию QueueException

+getFront(out queueFront:QueueItemType) throw QueueException
// Извлекает первый элемент в переменную queueFront. Иными
// словами, извлекает элемент, поставленный в очередь раньше
// всех. Если удаление невозможно, генерирует исключительную
// ситуацию QueueException
```

На рис. 7.2 показано, как эти операции применяются к очереди, состоящей из целых чисел.

Операция	Очередь после операции
	front
aQueue.createQueue()	
aQueue.enqueue(5)	5
aQueue.enqueue(2)	5 2
aQueue.enqueue(7)	5 2 7
aQueue.getFront(queueFront)	5 2 7 queueFront = 5
aQueue.dequeue(queueFront)	5 2 7 queueFront = 5
aQueue.dequeue(queueFront)	2 7 queueFront = 2

Рис. 7.2. Операции над очередью

Некоторые применения абстрактной очереди

В этом разделе описаны два простых применения абстрактной очереди. Эти программы используют операции над абстрактной очередью совершенно независимо от ее реализации.

Считывание строки символов

При вводе текста с клавиатуры система должна соблюдать их очередность. Для этой цели хорошо подходит абстрактная очередь. Рассмотрим ее псевдокод.

Очередь, содержащая символы в порядке их ввода

```
// Считать и поставить в очередь
// символы вводимой строки
aQueue.createQueue()
while(не конец строки)
{
   Считать новый символ ch
   aQueue.enqueue(ch)
} // Конец оператора while
```

Поскольку символы находятся в очереди, система может обрабатывать их по мере необходимости. Например, если нужно ввести целое число — без ошибок, но с пробелами, — очередь будет состоять из цифр и, возможно, пробелов. Если цифрами являются 2, 4 и 7, система может создать из них число 247, вычислив значение выражения

```
10 * (10*2 + 4) + 7
```

Псевдокод этого преобразования имеет следующий вид.

```
// Преобразовать цифры из очереди aQueue в десятичное целое
// число n

// Считать первую цифру и удалить незначащие пробелы
do
{
   aQueue.dequeue(ch)
} while(символ ch является пробелом)

// Диагностическое утверждение: символ ch содержит
// первую цифру

// Вычислить число n по цифрам, стоящим в очереди
n = 0
```

```
done = false
do
{
   n = 10 * n + целое число, представленное символом ch
   if (!aQueue.isEmpty())
     aQueue.dequeue(ch)
   else
     done = true
} while (!done и символ ch является цифрой)
// Диагностическое утверждение: число n — результат
```

Распознавание палиндромов

Очередь в сочетании со стеком позволяет распознавать палиндромы

Напомним, что палиндромом является строка символов, которые одинаково читаются слева направо и справа налево. В предыдущем разделе мы показали, как можно применить стек для изменения порядка следования символов на противоположный. Обратите внимание, что очередь, наоборот, позволяет сохранить исходный порядок. Итак, очередь в сочетании со стеком позволяет распознавать палиндромы.

Обходя строку слева направо, мы можем вставлять каждый прочитанный символ и в стек, и в очередь одновременно. Результат этих действий на примере строки *abcbd* показан на рис. 7.3. Эта строка не является палиндромом. Наблюдателю доступен первый символ строки, записанный в начале очереди, и последний символ строки, находящийся на вершине стека. Итак, символы, удаляемые из очереди, будут появляться в порядке их следования в строке, а символы, удаляемые из стека, — в обратном порядке.

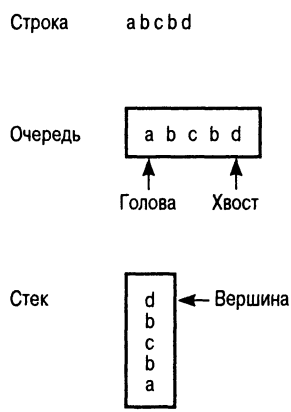


Рис. 7.3. Результат вставки символов строки в очередь и стек

Будем сравнивать символы, стоящие в начале очереди и на вершине стека. Если они совпадают, их можно удалить. Повторение этой процедуры приведет либо к полному опустошению очереди (значит, исходная строка — палиндром), либо к появлению двух разных символов (следовательно, строка палиндромом не является).

Псевдокод итеративного алгоритма распознавания палиндромов имеет следующий вид.

```
isPal(in str:string):boolean
// Определяет, является ли строка str палиндромом.

   // Создать пустую очередь и пустой стек

   aQueue.createQueue()
   aStack.createStack()

   // Вставить каждый символ строки в очередь и стек.
   length = длина строки
   for (i = 1 through length)
   {
      nextChar = i-й символ строки str
      aQueue.enqueue(nextChar)
      aStack.push(nextChar)
   } // Конец оператора for

   // Сравнить символы, стоящие в очереди,
   // с символами из стека
   charactersAreEqual = true
   while (очередь aQueue не пуста и charactersAreEqual)
   {
      aQueue.getFront(queueFront)
      aStack.getTop(stackTop)

      if (символ queueFront совпадает с символом stackTop)
      {
         aQueue.dequeue()
         aStack.pop()
      else
         charactersAreEqual = false
      } // Конец оператора while
return charactersAreEqual
```

Реализация абстрактной очереди

Подобно стекам, очереди можно реализовывать в виде массивов или связанных списков. Во всех этих реализациях можно использовать следующее определение исключительной ситуации *QueueException*.

```
#include <exception>
#include <string>
using namespace std;

class QueueException: public exception
{
public:
   QueueException(const string & message="")
                  : exception(message.c_str())
   {}
}; //  Конец класса QueueException
```

Для очереди реализация в виде связанного списка является более естественной, поэтому начнем с нее.

Реализация очереди в виде связанного списка

Очередь можно представить в виде линейного или кольцевого связанного списка

Для реализации очереди можно применить линейный связанный список с двумя внешними указателями на начало и конец, соответственно, как показано на рис. 7.4, *а*. На рис. 7.4, *б* показано, что на самом деле можно обойтись одним внешним указателем, если сделать связанный список кольцевым. В задании 1, приведенном в конце главы, читателям предлагается самим рассмотреть детали реализации очереди в виде кольцевого связанного списка.

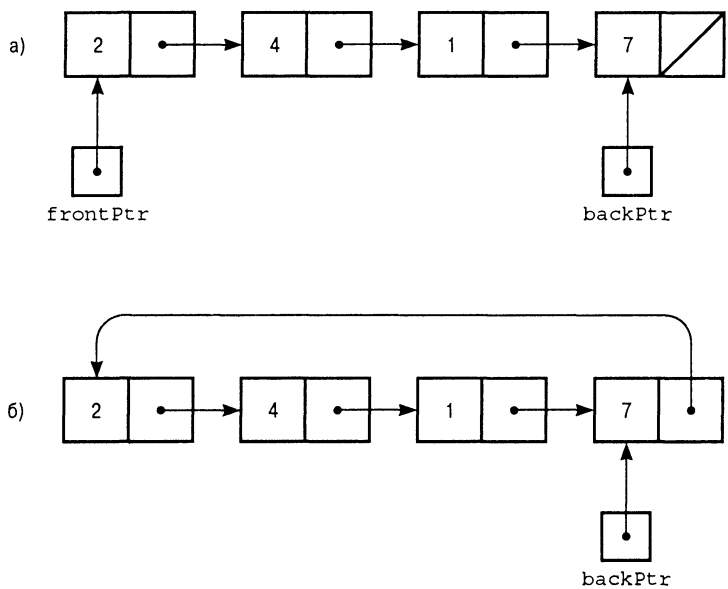


Рис. 7.4. Реализация очереди: а) в виде линейного связанного списка с двумя внешними указателями; б) в виде кольцевого связанного списка с одним внешним указателем

Операции вставки последнего элемента и удаления первого вполне очевидны. На рис. 7.5 показана операция вставки элемента в непустую очередь. Для вставки в конец очереди нового узла, на который ссылается указатель `newPtr`, необходимо изменить значения трех указателей: указателя на следующий узел в новом узле, указателя на следующий узел в текущем узле и внешнего указателя `backPtr`. Эти изменения и порядок их выполнения показаны на рис. 7.5. (Пунктирные линии указывают на значения этих указателей до их изменения.) Вставка нового узла в пустой список представляет собой отдельную задачу, проиллюстрированную на рис. 7.6.

Удаление из начала очереди представляет собой более простую задачу, чем вставка нового элемента в конец очереди. На рис. 7.7 показано удаление из очереди нескольких элементов. Обратите внимание, что при этом изменяется только указатель `frontPtr`. Удаление из очереди единственного элемента рассматривается отдельно, поскольку при этом указатели `backPtr` и `frontPtr` обнуляются.

Приведенный ниже файл содержит реализацию абстрактной очереди в виде связанного списка. Поскольку данные хранятся в динамической памяти, необходимо предусмотреть свой собственный конструктор копирования и деструктор.

```
1. newPtr->next = NULL;
2. backPtr->next = newPtr;
3. backPtr = newPtr;
```

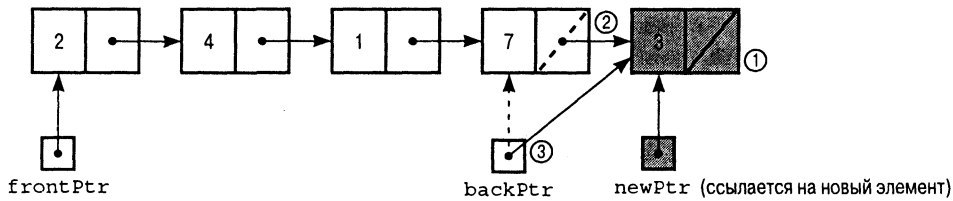


Рис. 7.5. Вставка элемента в непустую очередь

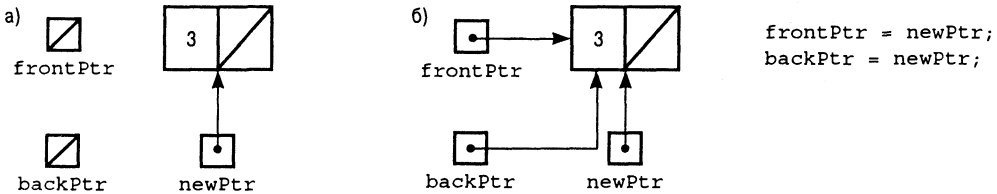


Рис. 7.6. Вставка элемента в пустую очередь: а) до вставки; б) после вставки

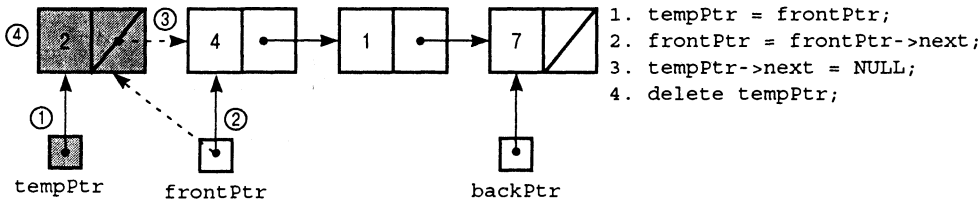


Рис. 7.7. Удаление из очереди нескольких элементов

```
// ********************************************************
// Заголовочный файл QueueP.h абстрактной очереди.
// Реализация в виде связанного списка.
// ********************************************************
#include "QueueException.h"
typedef тип-элемента-очереди QueueItemType;
class Queue
{
public:
// Конструкторы и деструктор:
   Queue(); // Конструктор по умолчанию
   Queue(const Queue& Q); // Конструктор копирования
   ~Queue(); // Деструктор

// Операции класса Queue:
   bool isEmpty() const;
   // Определяет, пуста ли очередь.
   // Предусловие: нет.
   // Постусловие: если очередь пуста, возвращает значение
   // true, в противном случае возвращает значение false.
```

```
    void enqueue(QueueItemType newItem) throw(QueueException);[1]
    // Вставляет элемент в конец очереди
    // Предусловие: вставляемый элемент задается
    // аргументом newItem.
    // Постусловие: если вставка прошла успешно,
    // аргумент newItem стоит в конце очереди.
    // Исключительная ситуация: если элемент newItem в очередь
    // поставить невозможно, генерируется исключительная
    // ситуация QueueException.

    void dequeue() throw(QueueException);
    // Удаляет голову очереди
    // Precondition: None.
    // Предусловие: нет.
    // Постусловие: если очередь не пуста, из нее удаляется
    // элемент, добавленный раньше всех
    // Исключительная ситуация: если очередь пуста,
    // генерируется исключительная ситуация QueueException.

    void dequeue(QueueItemType& queueFront)
                 throw(QueueException);
    // Извлекает и удаляет голову списка.
    // Предусловие: нет.
    // Постусловие: если очередь не пуста, аргумент queueFront
    // содержит элемент, добавленный в нее раньше всех.
    // Затем этот элемент удаляется.
    // Исключительная ситуация: если очередь пуста,
    // генерируется исключительная ситуация QueueException.

    void getFront(QueueItemType& queueFront) const
                  throw(QueueException);
    // Извлекает голову списка.
    // Предусловие: нет.
    // Постусловие: если очередь не пуста, аргумент queueFront
    // содержит элемент, добавленный в нее раньше всех.
    // Исключительная ситуация: если очередь пуста,
    // генерируется исключительная ситуация QueueException.

private:
    // Очередь реализуется в виде связанного списка с двумя
    // внешними указателями — на голову и конец очереди

    struct QueueNode
    {
       QueueItemType item;
       QueueNode *next;
    }; // Конец структуры
    QueueNode *frontPtr;
    QueueNode *backPtr;
}; // Конец класса
// Конец заголовочного файла.
```

[1] Если аргумент *newItem* является экземпляром некоего класса, его нужно передавать не по значению, а по ссылке, как константный параметр. Это позволит избежать дополнительных затрат на копирование объекта.

```
// ********************************************************
// Файл реализации QueueP.cpp абстрактной очереди.
// Реализация в виде связанного списка.
// ********************************************************
#include "QueueP.h" // Заголовочный файл
#include <cstddef>
#include <cassert>

Queue::Queue() : backPtr(NULL), frontPtr(NULL)
{
} // Конструктор по умолчанию

Queue::Queue(const Queue& Q)
{
  // Реализуйте эту функции самостоятельно (упражнение 4).
} // Конец конструктора копирования

Queue::~Queue()
{
   while (!isEmpty())
   dequeue();
   // Диагностическое утверждение: указатели frontPtr и backPtr
   // равны константе NULL
} // Конец деструктора

bool Queue::isEmpty() const
{
   return bool(backPtr == NULL);
} // Конец функции isEmpty

void Queue::enqueue(QueueItemType newItem)
{
   // Создать новый элемент
   QueueNode *newPtr = new QueueNode;
   if (newPtr == NULL) // Проверка выделения памяти
      throw QueueException(
      "QueueException: enqueue — недостаточно памяти");
   else
   {
      // Выделение памяти прошло успешно;
      // записать данные в новый узел
      newPtr->item = newItem;

      // insert the new node
      if (isEmpty())
         // Вставка элемента в пустую очередь
         frontPtr = newPtr;
      else
         // Вставка элемента в непустую очередь
         newPtr->next = backPtr;
         backPtr = newPtr; // Новый элемент
                           // стоит в конце очереди
      } // Конец оператора if
} // Конец функции enqueue
```

```
void Queue::dequeue()
{
   if (isEmpty())
       throw QueueException(
       "QueueException: dequeue — очередь пуста");
   else
   {
       // Очередь не пуста. Удалить первый элемент.
       QueueNode *tempPtr = frontPtr;
       if (frontPtr == backPtr) // Отдельная задача?
       {
          // Да, в очереди только один узел
          frontPtr = NULL;
          backPtr = NULL;
       }
       else
          frontPtr = frontPtr->next;

       tempPtr->next = NULL; // Меры предосторожности
       delete tempPtr;
   } // Конец оператора if
} // Конец функции dequeue

void Queue::dequeue(QueueItemType& queueFront)
{
   if (isEmpty())
       throw QueueException(
       "QueueException: dequeue — очередь пуста");
   else
   {
       // Очередь не пуста. Удалить первый элемент
       queueFront = frontPtr->item;
       dequeue(); // Удалить первый элемент
   } // Конец оператора if
} // Конец функции dequeue

void Queue::getFront(QueueItemType& queueFront) const
{
   if (isEmpty())
       throw QueueException(
       "QueueException: getFront — очередь пуста");
   else
       // Очередь не пуста. Извлечь первый элемент.
       queueFront = frontPtr->item;
} // Конец функции getFront
// Конец файла реализации.
```

Программа, использующая эту реализацию очереди, может начинаться так.

```
#include "QueueP.h"
int main()
{
   Queue aQueue;
   aQueue.enqueue(15);
   . . .
```

Реализация очереди в виде массива

Если фиксированный размер очереди не создает особых проблем, для ее представления можно использовать массив. Как показано на рис. 7.8, *а*, наивная реализация очереди в виде массива может содержать следующее определение.

Наивная реализация очереди в виде массива

```
const int MAX_QUEUE = максимальный-
размер-очереди;
typedef тип-элемента-очереди
QueueItemType;

QueueItemType items[MAX_QUEUE];
int front;
int back;
```

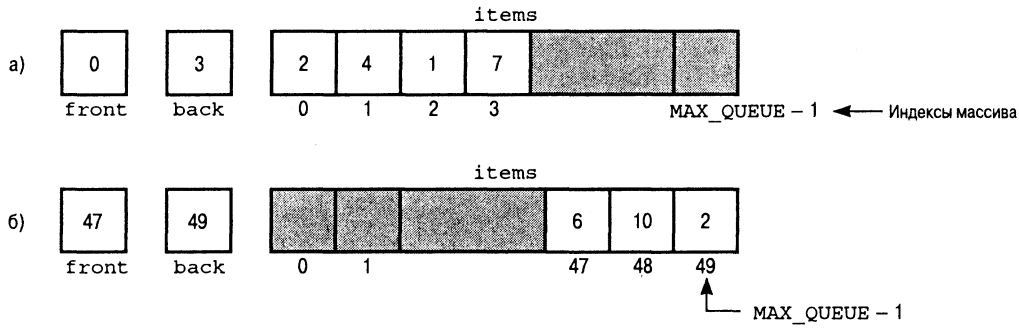


Рис. 7.8. Реализация очереди в виде массива: а) наивная реализация очереди в виде массива; б) дрейф вправо может привести к переполнению очереди

Здесь переменные *front* и *back* являются индексами первого и последнего элементов очереди соответственно. В исходном положении переменная *front* равна 0, а переменная *back* равна -1. Для того чтобы вставить в очередь новый элемент, нужно увеличить значение переменной *back* на единицу и записать новый элемент в ячейку *items[back]*. Для того чтобы удалить элемент, нужно просто увеличить на единицу значение переменной *front*. Очередь опустошается, когда индекс *back* становится меньше индекса *front*. Очередь переполняется, если значение переменной *back* становится равным *MAX_QUEUE*-1.

Дрейф вправо может привести к переполнению очереди, даже если в очереди есть место

С этой стратегией связана проблема **дрейфа вправо** (rightward drift). Она состоит в следующем. После выполнения последовательности вставок и удалений элементы очереди смещаются в направлении конца массива, и переменная *back* может стать равной значению *MAX_QUEUE*-1, даже если очередь состоит из небольшого количества элементов. Эта ситуация показана на рис. 7.8, *б*.

Смещение элементов, компенсирующее дрейф вправо, неэффективно

Для решения этой проблемы после каждого удаления можно смещать все элементы массива влево. То же самое можно сделать, если переменная *back* стала равной значению *MAX_QUEUE*-1. Это гарантирует, что очередь всегда будет состоять не больше, чем из *MAX_QUEUE* элементов. Однако это решение не удовлетворительно, поскольку сдвиг элементов массива представляет собой слишком затратную операцию.

Более элегантное решение достигается, если применить кольцевой массив, как показано на рис. 7.9. При этом индексы *front* (при удалении элемента) и *back* (при вставке элемента) перемещаются вдоль массива вперед по часовой стрелке. На рис. 7.10 показан результат трех последовательных операций над переменными *front*, *back* и массивом. Когда переменные *front* или *back* становятся равными значению *MAX_QUEUE*-1, они обнуляются. Это позволяет избежать дрейфа вправо, поскольку кольцевой массив не имеет конца.

В этой схеме есть только один недостаток, связанный с определением условий опустошения и переполнения очереди. Условие опустошения очереди можно было бы сформулировать следующим образом.

Ячейка с индексом front непосредственно предшествует ячейке с индексом back.

Иными словами, когда очередь становится пустой, индекс *front* догоняет индекс *back*. Эта ситуация изображена на рис. 7.11, *а*. Однако эта же ситуация возникает, когда очередь заполняется полностью: поскольку массив является кольцевым, при заполнении очереди индекс *back* может догнать индекс *front*. Эта ситуация изображена на рис. 7.11, *б*.

Подсчитывая количество элементов, можно отличить полную очередь от пустой

Очевидно, нам необходимо различать две эти ситуации. Для этого можно подсчитывать количество элементов, стоящих в очереди. Перед вставкой в очередь следует проверить, не равен ли счетчик элементов значению *MAX_QUEUE*. Если да, очередь полностью заполнена. Перед удалением элемента из очереди следует проверить, не равен ли счетчик нулю. Если да, очередь пуста.

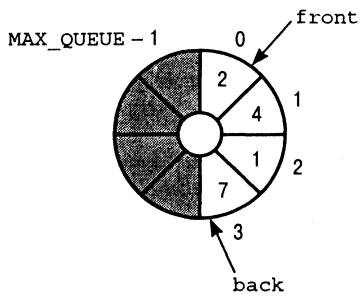


Рис. 7.9. Кольцевая реализация очереди

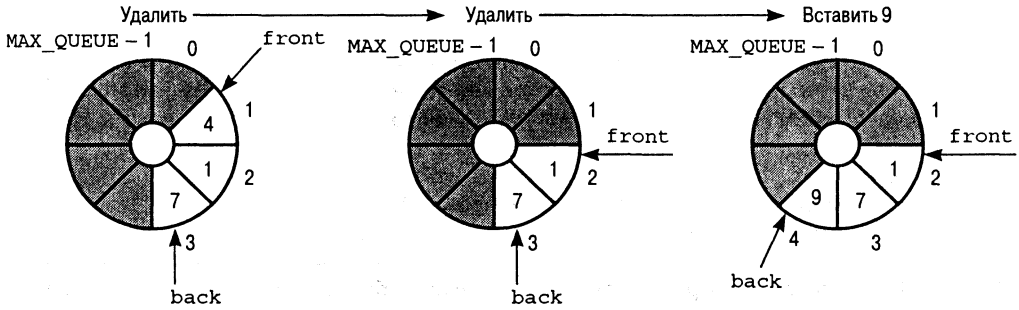


Рис. 7.10. Результат выполнения операций над очередью, показанной на рис. 7.9

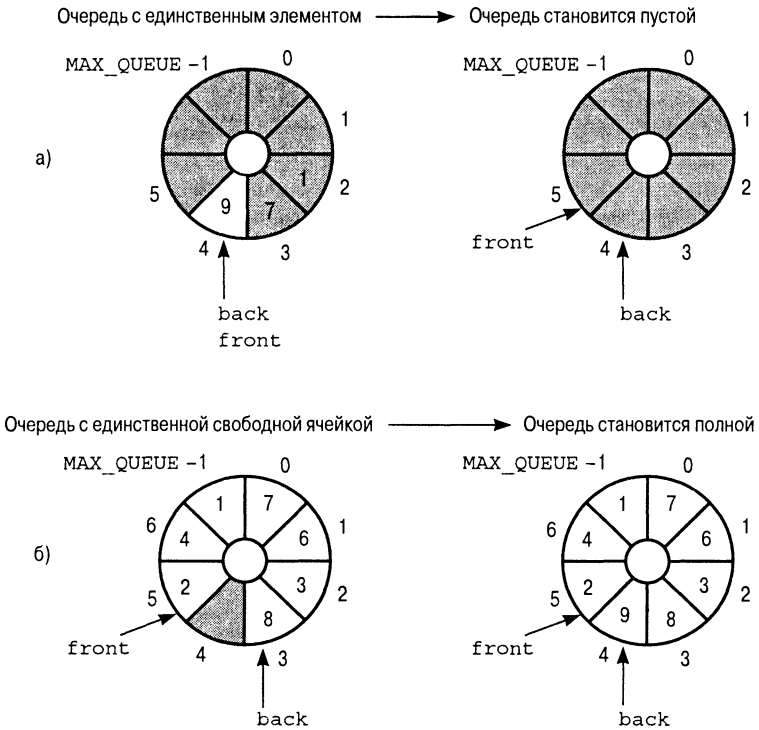


Рис. 7.11. Пустая и полностью заполненная очереди: а) когда очередь становится пустой, индекс front догоняет индекс back; б) при заполнении очереди индекс back догоняет индекс front

Для инициализации очереди следует положить переменную *front* равной 0, а переменную *back* — равной величине *MAX_QUEUE*-1. Эффект циклического возврата кольцевой очереди при увеличении переменных *front* и *back* достигается с помощью модульной арифметики (т.е. оператора целочисленного деления %, предусмотренного в языке C++). Например, для вставки в очередь нового элемента *newItem* можно выполнить следующий фрагмент программы.

Вставка элемента в очередь

```
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;
```

Если перед вставкой элемента *newItem* переменная *back* равна *MAX_QUEUE*-1, выполнение первого оператора, back = (back+1) % MAX_QUEUE, приведет к циклическому возврату переменной *back* вокруг нуля.

Удаление элемента из очереди

```
front = (front+1) % MAX_QUEUE;
--count;
```

Ниже приводится реализация абстрактной очереди в виде кольцевого массива на языке C+. Поскольку данные хранятся в статическом массиве, достаточно применить автоматический конструктор копирования и деструктор.[2] Для эконо-

[2] Если бы использовался динамический массив, нужно было бы создавать свой собственный конструктор копирования и деструктор.

мии места пред- и постусловия функций не приводятся. Они полностью совпадают с их аналогами в реализации очереди на основе связанного списка.

```
// ********************************************************
// Заголовочный файл QueueA.h абстрактной очереди.
// Реализация в виде массива.
// ********************************************************
#include "QueueException.h"
const int MAX_QUEUE = максимальный-размер-очереди;
typedef тип-элемента-очереди QueueItemType;

class Queue
{
public:
// Конструкторы и деструктор:
   Queue(); // Конструктор по умолчанию
   // Конструктор копирования и деструктор
   // генерируются компилятором

// Операции класса Queue:
bool isEmpty() const;

void enqueue(QueueItemType newItem)3
       throw(QueueException);
void dequeue() throw(QueueException);
void dequeue(QueueItemType& queueFront)
       throw(QueueException);
void getFront(QueueItemType& queueFront) const
       throw(QueueException);

private:
   QueueItemType items[MAX_QUEUE];
   int front;
   int back;
   int count;
}; // Конец класса Queue
// Конец заголовочного файла.

// ********************************************************
// Файл реализации QueueA.cpp абстрактной очереди.
// Реализация в виде кольцевого массива.
// Массив хранит индексы начала и конца очереди.
// Счетчик отслеживает текущее количество элементов очереди.
// ********************************************************

#include "QueueA.h" // header file

Queue::Queue():front(0), back(MAX_QUEUE-1), count(0)
{
} // Конец конструктора по умолчанию

bool Queue::isEmpty() const
{
   return bool(count == 0);
} // Конец функции isEmpty
```

```
void Queue::enqueue(QueueItemType newItem)
{
   if (count == MAX_QUEUE)
       throw QueueException(
       "QueueException: enqueue — очередь переполнена");
   else
   {
       // Очередь не полностью заполнена.
       // Вставить новый элемент
       back = (back+1) % MAX_QUEUE;
       items[back] = newItem;
       ++count;
   } // Конец оператора if
} // Конец функции enqueue

void Queue::dequeue()
{
   if (isEmpty())
      throw QueueException(
      "QueueException: dequeue — очередь пуста");
   else
   {
      // Очередь не пуста. Удалить первый элемент.
      front = (front+1) % MAX_QUEUE;
      --count;
   } // Конец оператора  if
} // Конец функции dequeue

void Queue::dequeue(QueueItemType& queueFront)
{
   if (isEmpty())
      throw QueueException(
      "QueueException: dequeue — очередь пуста");
   else
   {
      // Очередь не пуста. Извлечь первый элемент
      queueFront = items[front];
      front = (front+1) % MAX_QUEUE;
      --count;
   } // Конец оператора if
} // Конец функции dequeue

void Queue::getFront(QueueItemType& queueFront) const
{
   if (isEmpty())
      throw QueueException(
      "QueueException: getFront — очередь пуста");
   else
      // Очередь не пуста. Извлечь первый элемент
      queueFront = items[front];
} // Конец функции getFront
// Конец файла реализации.
```

В некоторых достаточно широко распространенных случаях подсчитывать количество элементов, стоящих в очереди, не обязательно.

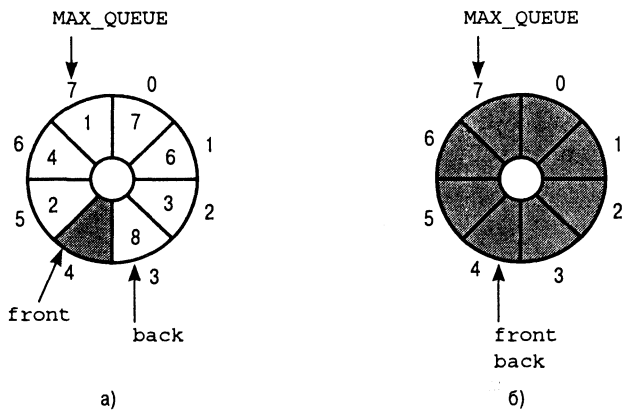
Вместо счетчика можно использовать признак isFull

Например, можно различать пустую и полную очереди с помощью признака *isFull*. Однако затраты на поддержку признака *isFull* практически такие же, как и при использовании счетчика. Более эффективная реализация получается, если в массиве *items* зарезервировать *MAX_QUEUE*+1 ячеек, используя при этом лишь *MAX_QUEUE* элементов очереди. Пожертвовав одной ячейкой, можно добиться, чтобы индекс *front* всегда ссылался на ячейку, предшествующую началу очереди. Как показано на рис. 7.12, очередь оказывается полной, если

переменная front равна (back-1) % (MAX_QUEUE+1)

и пустой, если

переменная front равна back



а) б)

Рис. 7.12. Более эффективная кольцевая реализация: а) полная очередь; б) пустая очередь

Использование дополнительной ячейки массива экономит время

В этой реализации не используется счетчик или признак *isFull*, и при этом она работает быстрее. Для стандартных типов данных эта реализация потребует столько же места, как и реализации, использующие счетчики или признак *isFull* (почему?). Однако если в реализации используются более сложные данные, потери памяти могут стать значительными. Еще две реализации очереди рассмотрены в заданиях 3 и 4.

Реализация очереди с помощью абстрактного списка

Для представления элементов, стоящих в очереди, можно применять абстрактный список, как показано на рис. 7.13. Если элемент списка, стоящий на первой позиции, представляет собой начало очереди, функцию *dequeue()* можно реализовать с помощью операции *remove(1)*, а функцию *getFront(queuefront)* — с помощью операции *retrieve(1, queuefront)*. Аналогично, если последний элемент списка представляет собой конец очереди, функцию *enqueue(newItem)* можно реализовать с помощью операции *insert(getLength()+1, newItem)*.

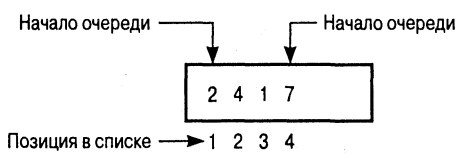


Рис. 7.13. Реализация очереди с помощью абстрактного списка

В главах 3 и 4 абстрактный список был представлен в виде класса *List*. Приведенный ниже класс, реализующий абстрактную очередь, использует экземпляр класса *List*. Предусловия и постусловия функций остаются без изменений и поэтому не повторяются.

```
// ********************************************************
// Заголовочный файл QueueL.h абстрактной очереди.
// Реализация в виде абстрактного списка.
// ********************************************************
#include "ListP.h" // Операции над абстрактным списком #include
"QueueException.h"
typedef ListItemType QueueItemType;

class Queue
{
public:
// Конструкторы и деструктор:
   Queue(); // Конструктор по умолчанию
   Queue(const Queue& Q); // Конструктор копирования
   ~Queue(); // Деструктор

// Операции класса Queue:
   bool isEmpty() const;
   void enqueue(QueueItemType newItem)
      throw(QueueException);
   void dequeue() throw(QueueException);
   void dequeue(QueueItemType& queueFront)
      throw(QueueException);
   void getFront(QueueItemType& queueFront) const
      throw(QueueException);

private:
   List aList; // Список элементов очереди
}; // Конец класса queue
// Конец заголовочного файла.

// ********************************************************
// Файл реализации QueueL.cpp абстрактной очереди.
// Реализация в виде абстрактного списка.
// ********************************************************
#include "QueueL.h" // Заголовочный файл

Queue::Queue()
{
} // Конец конструктора по умолчанию

Queue::Queue(const Queue& Q): L(Q.L)
{
} // Конец конструктора копирования

Queue::~Queue()
{
} // Конец деструктора

bool Queue::isEmpty() const
{
   return (aList.getLength() == 0);
} // Конец функции isEmpty
```

```
void Queue::enqueue(QueueItemType newItem)
{
   try
   {
      aList.insert(aList.getLength()+1, newItem);
   } // Конец блока try

   catch (ListException e)
   {
      throw QueueException(
      "QueueException: невозможно вставить элемент");
   } // Конец блока catch
} // Конец функции enqueue

void Queue::dequeue()
{
   if (aList.isEmpty())
      throw QueueException(
      "QueueException: dequeue — очередь пуста");
   else
      aList.remove(1);
} // Конец функции dequeue

void Queue::dequeue(QueueItemType& queueFront)
{
   if (aList.isEmpty())
      throw QueueException(
      "QueueException: dequeue — очередь пуста");
   else
   {
      aList.retrieve(1, queueFront);
      aList.remove(1);
   } // Конец оператора if
} // Конец функции dequeue

void Queue::getFront(QueueItemType& queueFront) const
{
   if (!aList.isEmpty())
      throw QueueException(
      "QueueException: getFront — очередь пуста");
   else
      aList.retrieve(1, queueFront);
} // Конец функции getFront
// Конец файла реализации.
```

Как использование стека, так и использование абстрактного списка намного упрощает реализацию очереди. В упражнении 6, приведенном в конце главы, читателям предлагается оценить эффективность этой реализации.

Шаблонный класс queue из библиотеки STL

В библиотеке STL предусмотрен контейнерный класс *queue*, аналогичный классу *Queue*, разработанному в этой главе. В классе *queue* также предусмотрены операции вставки элементов в конец очереди и удаления их из начала очереди, од-

нако они называются *push* и *pop*, как их аналоги в классе *stack*. Это выбор названий следует признать совершенно неудачным, поскольку смысл этих функций не совпадает.

В шаблонном классе *queue* предусмотрена функция для извлечения первого элемента очереди. В библиотеке STL эта функция называется *front*, а в нашем классе — *getFront*. В шаблонном классе *queue* есть две функции, которых не было в нашем классе *Queue*. Это функция *back*, возвращающая ссылку на последний элемент очереди, и функция *size*, вычисляющая количество элементов, стоящих в очереди. Несколько упрощенная спецификация шаблонного класса *queue* приводится ниже.

```
template <class T, class Container = deque <T> >
class queue
{
public:
   explicit queue(const Container& cnt = Container());
   // Конструктор по умолчанию, инициализирующий пустую строку.
   // Предусловие: нет.
   // Постусловие: создана пустая очередь.

   bool empty() const;
   // Определяет, пуста ли очередь.
   // Предусловие: нет.
   // Постусловие: если очередь пуста, возвращается значение
   // true, в противном случае возвращается значение false.

   size_type size() const;
   // Определяет размер очереди. Переменная size_type имеет
   // целочисленный тип.
   // Предусловие: нет.
   // Постусловие: возвращает количество элементов очереди.

   T &front();
   // Возвращает ссылку на первый элемент очереди.
   // Предусловие: нет.
   // Постусловие: элемент из списка не удаляется.

   T &back();
   // Возвращает ссылку на последний элемент очереди.
   // Предусловие: нет.
   // Постусловие: элемент из списка не удаляется.

   void pop();
   // Удаляет первый элемент очереди.
   // Предусловие: нет.
   // Постусловие: удален первый элемент очереди.

   void push(const T& x);
   // Вставляет новый элемент в конец очереди.
   // Предусловие: нет.
   // Постусловие: элемент x поставлен в конец очереди
} // Конец шаблонного класса queue из библиотеки STL
```

Шаблонный класс *queue*, как и класс *stack*, реализован с помощью более общих контейнерных типов. Классы, которые для своей реализации используют другие классы, называются **адаптерными контейнерами** (adapter container). Такой контейнер предоставляет ограниченный доступ к другому контейнеру, использованному в его реализации.

Адаптерный контейнер использует основные контейнерные классы

В библиотеке STL предусмотрены три основных контейнерных типа, используемых адаптерными контейнерами: *vector*, *list* и *deque*. С классом *list* мы уже встречались в главе 4. Класс *vector* основан на динамическом массиве. В классе *deque* применяется **двусторонняя очередь** (double-ended queue), представляющая собой слегка измененную абстрактную очередь, в которой элементы можно вставлять и удалять с обоих концов. Эти три основных контейнерных класса обычно реализуются непосредственно, поскольку выражать их один через другой нецелесообразно.

Однако классы *stack* и *queue* можно просто и эффективно реализовать с помощью основных контейнерных классов. Например, реализация шаблонного класса *queue* может использовать шаблонный класс *list* из библиотеки STL. Обратите внимание, что класс *vector* для реализации очереди применять нельзя, поскольку в нем не предусмотрены некоторые необходимые операции, в то время как для реализации класса *stack* можно использовать любой из основных контейнерных классов.

Если в объявлении экземпляра шаблонного класса *stack* или *queue* не указан контейнерный класс, по умолчанию используется класс *deque*. Например, объявление

По умолчанию для реализации класса queue используется контейнерный класс deque

```
queue<int> myQueue;
```

создает пустую очередь *myQueue*, реализуя ее с помощью класса *deque*. Кроме того, в основу очереди можно положить класс *list*, объявив ее следующим образом.

```
queue<int, list<int> > myQueue;
```

Это объявление создает пустую очередь *myQueue*, реализованную с помощью класса *list*.

Используя в конструкторе существующий контейнер, можно задать начальное состояние очереди или стека. Например, приведенная ниже программа создает очередь и стек на основе уже существующего списка.

Используя конструкторы, можно задавать начальные значения

```
#include <iostream>
#include <list>
#include <queue>
#include <stack>
using namespace std;

int main()
{
   list<int> myList; // Создает пустой список
   list<int>::iterator i = myList.begin();

   for (int j = 1; j < 5; j++)
   {
      i = myList.insert(i, j);
      i++;
   } // Конец оператора for
```

```
    cout << "myList: ";
    i = myList.begin();
    while (i != myList.end())
    {
       cout << *i << " ";
       i++;
    } // Конец оператора while

    cout << endl;

    // Предполагается, что начало списка
    // является началом очереди
    queue<int, list<int> > myQueue(myList);

    // Предполагается, что конец списка является вершиной стека
    stack<int, list<int> > myStack(myList);

    cout << "myQueue: ";
    while (!myQueue.empty())
    {
      cout << myQueue.front() << " ";
      myQueue.pop();
    } // Конец оператора while

    cout << endl;
    cout << "myStack: ";
    while (!myStack.empty())
    {
       cout << myStack.top() << " ";
       myStack.pop();
    } // Конец оператора while

    cout << endl;
    return 0;
} // Конец функции main
```

Эта программа выводит на экран следующий результат.

```
myList:  1 2 3 4
myQueue: 1 2 3 4
myStack: 4 3 2 1
```

Как видим, здесь предполагается, что начало списка является началом очереди, а конец очереди является вершиной стека.

Сравнение реализаций

Рассмотренные выше реализации абстрактной очереди использовали линейный связанный список, кольцевой связанный список, массив, кольцевой массив, а также абстрактный список. Мы подробно рассмотрели три из перечисленных реализаций. Все изученные нами реализации абстрактной очереди так или иначе использовали массив или связанный список.

Критерии выбора между массивом и связанным списком уже обсуждались нами в разделе "Сравнение реализаций" главы 6. Повторим основные моменты, учитывая специфику очереди.

Фиксированный и переменный размеры

Реализация, основанная на статическом массиве, предотвращает вставку нового элемента в полностью заполненный массив. Если это ограничение является слишком жестким, следует применить динамический массив или связанный список. Допустим, мы решили использовать связанный список. Какой вид связанного списка предпочесть? А может применить абстрактный список? Хотя абстрактный список намного облегчает программирование, использовать его не следует, поскольку он реализуется через связанный. Намного эффективнее применить связанный список непосредственно.

Какой вид связанного списка следует выбрать для реализации очереди? Ответ на это вопрос вы найдете, выполнив задание 1, приведенное в конце главы.

Абстрактные типы данных, основанные на позиционном принципе

Операции над абстрактным списком, стеком и очередью зависят от позиции их элементов

Мы рассмотрели три абстрактных типа данных — список, стек и очередь. Они объединены общей темой: все их операции зависят от позиций элементов. Стеки и очереди сильно ограничивают выбор позиций, к которым можно применять операции — в них доступны только крайние элементы. Список не имеет таких ограничений.

Сравнение операций над стеком и очередью

Стеки очень похожи на очереди. Это сходство еще больше усиливается, если сравнить их операции попарно.

- **Операции createStack и createQueue.** Эти операции создают пустой абстрактный стек и пустую абстрактную очередь.
- **Операции isEmpty для стека и очереди.** Эти операции позволяют определить, пуста ли соответствующая структура данных.
- **Операции push и enqueue.** Эти операции вставляют новый элемент в один из концов структуры данных (вершину стека и конец очереди соответственно).
- **Операции push и enqueue.** Операция pop удаляет элемент, добавленный последним, из вершины стека, а операция *dequeue* удаляет первый элемент, находящийся в начале очереди.
- **Операции getTop и getFront.** Операция стека *getTop* извлекает элемент из его вершины, а операция очереди *getFront* извлекает первый элемент из очереди.

Абстрактный список позволяет вставлять, удалять и проверять элементы независимо от их позиции. Таким образом, эти операции являются наиболее гибкими среди всех **позиционно-ориентированных** (position-oriented) абстрактных типов данных. Операции над абстрактным списком можно рассматривать как обобщенные варианты операций над стеками и очередями.

Операции над абстрактным списком обобщают операции над стеками и очередями

- **Операция getLength.** Если распространить операцию *isEmpty* на несколько элементов, то можно определить, сколько элементов содержится в списке.
- **Операция insert.** Если снять ограничения, наложенные операциями *push* и *enqueue*, то элементы можно будет вставлять в любую позицию. Именно это делает операция вставки нового элемента в список.

- **Операция remove.** Если снять ограничения, наложенные операциями *pop* и *deque*, получим операцию удаления элемента из любой позиции списка.
- **Операция retrieve.** Если снять ограничения, наложенные операциями *getTop* и *getFront*, то можно будет извлекать элемент из любой позиции списка.

Поскольку операции над каждым их трех абстрактных типов данных, указанных выше, зависят от позиции элементов, доступ к этим элементам должен быть прост и эффективен. Например, реализация стека должна обеспечивать быстрый доступ к вершине, а реализация очереди — к первому и последнему элементу.

Приложение: моделирование

Моделирование имитирует поведение системы

Моделирование — основная область применения компьютеров — это способ имитации поведения природных и технических систем. Обычно целью моделирования является определение статистических параметров, характеризующих производительность реальной системы или предварительная оценка эффективности гипотетической системы. В этом разделе мы изучим простой пример, иллюстрирующий важность моделирования.

Рассмотрим следующую задачу. Мисс Симпсон, президент банка города Спрингфилда, узнала, что клиенты жалуются на большие очереди. Опасаясь, что клиенты переведут свои счета в другой банк, она стала подумывать, не нанять ли второго кассира.

Прежде чем сделать это, мисс Симпсон решила оценить среднее время ожидания в очереди к единственному кассиру ее банка. Как это сделать? Конечно, она могла бы встать в вестибюле с секундомером и простоять там целый день, но эта перспектива ее не прельщала. Вместо этого она решила применить метод, позволяющий предсказать, насколько ускорится обслуживание клиентов, если банк наймет дополнительных кассиров. Разумеется, о том, чтобы нанять кассира временно и оценить эффективность работы банка в течение испытательного срока, не может быть и речи.

Мисс Симпсон решила получить интересующую информацию с помощью компьютера, смоделировав работу ее банка. Моделируя такую систему, как банк, нужно сначала построить его математическую модель, которая учитывает всю необходимую информацию. Например, сколько кассиров работает в банке? Как часто приходят клиенты? Если модель адекватно описывает реальную систему, то моделирование может точно оценить ее эффективность. Например, моделирование может предсказать среднее время ожидания в очереди на обслуживание. С помощью моделирования можно также оценить эффективность предлагаемых изменений. Например, можно сказать, стоит ли нанимать новых кассиров. Существенное сокращение времени ожидания в очередях может оправдать понесенные затраты, связанные с наймом дополнительных кассиров.

Моделируемый период времени

Основным понятием моделирования является моделируемый период времени. Представьте себе секундомер, измеряющий время, прошедшее с начала моделирования. Допустим, например, что в модели банка предусмотрен только один кассир. В момент 0, когда начинается очередной банковский день, моделируемая система может находиться в исходном состоянии (клиенты еще не пришли). Затем секундомер начинает отсчитывать минуты, и происходят определенные события. Через 12 минут прибывает первый клиент. Поскольку очереди еще нет, он идет непосредственно в кассу и начинает выполнять свою банковскую операцию. Че-

рез 20 минут с момента открытия банка прибывает второй клиен
первый клиент еще не закончил свою операцию, второй посетите
встать в очередь. На 38-й минуте первый клиент заканчивает свои
нается обслуживание второго посетителя. Ситуации, соответствую
тырем моментам времени, изображены на рис. 7.14.

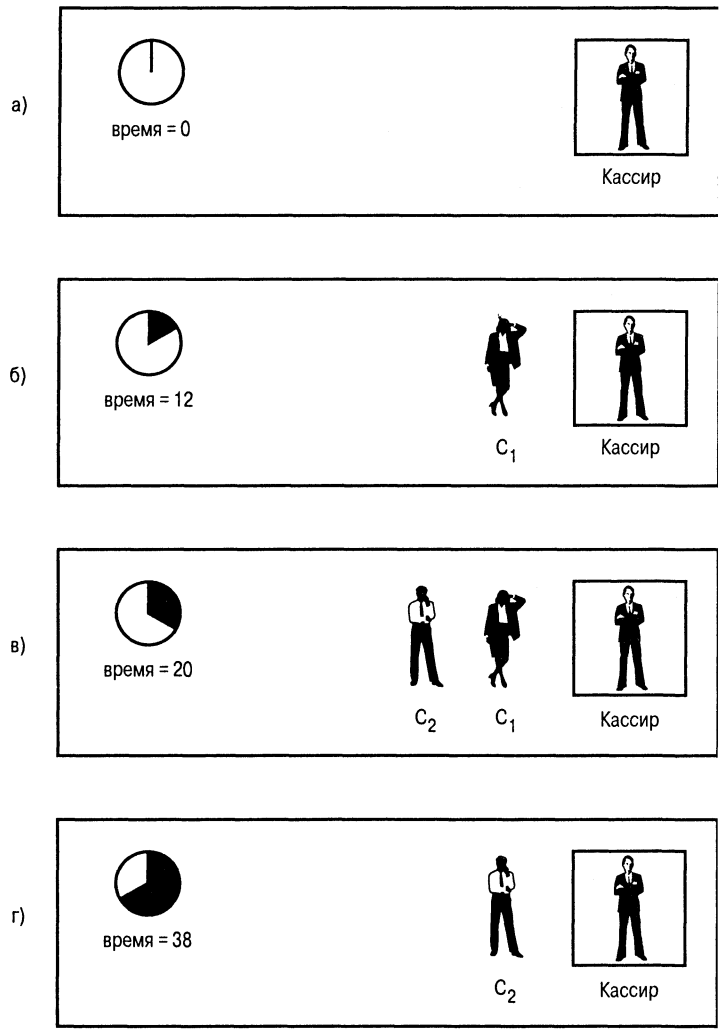


Рис. 7.14. Очередь в банке в разные моменты времени: а) в начале; б) через 12 мин.; в) через 20 мин. г) через 38 мин.

Чтобы собрать необходимую информацию, нужно выполнить мс
системы для заданного периода времени. В ходе моделирования не
бирать статистические показатели, например, среднее время ожидан
на обслуживание. Обратите внимание, что в небольшом примере, из
на рис. 7.14, первый клиент ожидал начала операции 0 минут, а
минут, поэтому среднее время ожидания равно 9 минутам.

В предыдущем описании мы упустили один момент — как распознать наступление определенных событий? Например, почему мы утверждаем, что первый посетитель пришел через 12 минут после открытия банка, а второй — через 20? Исследуя реальные системы, подобные банку, математики изучают модели событий, например, прибытие посетителей, с помощью методов теории вероятностей. Статистическая информация становится неотъемлемой частью математической модели системы и используется для генерации событий, отражающих реальность. Эти события используются в процессе моделирования и поэтому такое моделирование называется **событийным** (event-driven simulation). Обратите внимание, что целью моделирования является не предсказание конкретного события, а долговременная оценка средних параметров поведения системы.

Хотя способы генерации событий, отражающих реальность, весьма интересны и важны, этот вопрос может увести нас далеко в математические дебри. Поэтому будем просто предполагать, что мы умеем генерировать такие события. В частности, будем считать, что время прибытия каждого посетителя (событие) и длительность его обслуживания записаны в текстовом файле.

Например, данные

Пример событий и длительности обслуживания

```
20    5
22    4
23    2
30    3
```

означают, что первый посетитель прибыл через 20 минут после начала моделирования и что его операции заняли 5 минут; второй клиент пришел через 22 минуты после начала моделирования и оформлял сделку в течение 4 минут и т.д. Будем считать, что входной файл упорядочен в порядке возрастания времени прибытия.

Учтите, что файл не содержит времени **отбытия** клиента. В нем не указывается, когда посетитель закончит свои дела и уйдет из банка. Этот момент должен определяться в ходе моделирования. Используя время прибытия и длительность обслуживания, легко определить, когда клиент уйдет из банка. Для этого сначала потренируемся на простом примере.

Результаты моделирования

Время	*Событие*
20	Посетитель № 1 прибыл в банк и начал оформление сделки
22	Посетитель № 2 прибыл в банк и встал в очередь
23	Посетитель № 3 прибыл в банк и встал в очередь
25	Посетитель № 1 покинул банк; началось обслуживание посетителя № 2
29	Посетитель № 2 покинул банк; началось обслуживание посетителя № 3
30	Посетитель № 4 прибыл в банк и встал в очередь
31	Посетитель № 3 покинул банк; началось обслуживание посетителя № 4
34	Посетитель № 4 покинул банк

Время ожидания посетителя — это интервал между его прибытием в банк и началом обслуживания. Нас интересует среднее время ожидания в очереди.

Итак, моделирование связано с двумя типами событий.

- **Прибытие.** Эти события означают прибытие в банк нового посетителя. Время прибытия задается во входном файле. Иными словами, такие события являются **внешними** (external events). Когда посетитель приходит в банк, происходит одно из двух. Если кассир в этот момент свободен, об-

служивание клиента начинается немедленно. Если кассир занят, посетитель становится в очередь и ожидает, пока тот освободится.

- **Отбытие.** Эти события означают уход посетителя из банка после выполнения своих операций. Момент отбытия из банка определяется в ходе моделирования. Иными словами, это — **внутренние события** (internal events). Когда посетитель заканчивает операции, он уходит из банка, и начинается обслуживание следующего клиента.

Основная задача алгоритма, предназначенного для моделирования, — определить, какое событие произошло и отреагировать на него соответствующим образом. В общих чертах, этот алгоритм выглядит так.

Первый вариант алгоритма моделирования

```
// Выполнить инициализацию
currentTime = 0
Инициализировать пустую очередь

while (currentTime не превосходит времени последнего события)
{
   if (в момент currentTime прибывает новый клиент)
   Обработка прибытия

   if (в момент currentTime из банка уходит очередной клиент)
   Обработка отбытия

// Если прибытие и отбытие происходят в одно и то же время,
// прибытие обрабатывается первым

   ++currentTime
} // Конец оператора while
```

Временное моделирование имитирует течение времени

Следует ли увеличивать переменную *currentTime* на единицу? Можно прибегнуть к **временному моделированию** (time-driven simulation), в котором время прибытия и отбытия является случайным, и сравнивать его с переменной *currentTime*. В этом случае для имитации течения времени нужно увеличивать переменную *currentTime* на единицу.

Событийное моделирование учитывает лишь моменты наступления определенных событий, в данном случае прибытия и отбытия клиентов

Однако напомним, что мы используем событийное моделирование, поэтому моменты прибытия клиентов и длительность их обслуживания задаются во входном файле. Поскольку нас интересуют только эти моменты времени и в интервале между ними ничего не происходит, можно просто присваивать переменной *currentTime* момент наступления следующего события.

Уточненный алгоритм моделирования

Итак, уточненный вариант псевдокода принимает следующий вид.

```
// Инициализировать пустую очередь
while (продолжается обработка события)
{
   currentTime = момент следующего события
   if (прибыл новый клиент)
      Обработка прибытия
   else
      Обработка отбытия
```

```
// Если прибытие и отбытие происходят в одно и то же время,
// прибытие обрабатывается первым
} // Конец оператора while
```

Для того чтобы выполнить оператор

```
currentTime = момент следующего события
```

Список событий содержит все будущие события

необходимо определить момент следующего прибытия или отбытия. Для этого нужно иметь **список событий** (event list). Моменты наступления событий, указанных в этом списке, упорядочены в порядке возрастания, поэтому время следующего события, подлежащего обработке, всегда находится в начале списка. Алгоритм просто извлекает его из начала списка, присваивает счетчику времени момент наступления следующего события и обрабатывает текущее событие. Трудность, однако, заключается в самой обработке списка событий.

Список событий содержит по крайней мере одно прибытие и одно отбытие

Поскольку каждое прибытие порождает только одно отбытие, возникает впечатление, что нужно считать весь входной файл и создать список всех моментов прибытия и отбытия, упорядоченный по времени. Ответив на вопрос 5, приведенный в конце главы, читатели легко убедятся, что такой подход весьма не практичен. Вместо этого список событий можно обработать так, чтобы он содержал события только одного вида.

Напомним, что моменты прибытия клиентов задаются во входном файле и упорядочены по времени. О прибытии клиента не следует беспокоиться, пока не будут обработаны все предыдущие прибытия, указанные в файле. В списке событий нужно просто хранить время наиболее раннего необработанного прибытия. Когда обработка добирается до этого события — т.е. когда прибывает очередной посетитель, — указанное событие в списке событий заменяется следующим событием, записанным во входном файле.

Аналогично, в списке событий следует хранить только время следующего ожидаемого отбытия. Вот только как определить это время? Обратите внимание, что следующее отбытие всегда относится к посетителю, который в данный момент обслуживается кассиром. Как только начинается его обслуживание, его время отбытия совершенно точно вычисляется по следующей формуле.

Время следующего отбытия = момент начала операции + длительность обслуживания.

Напомним, что длительность обслуживания задается во входном файле. Итак, как только началось обслуживание клиента, нужно поместить в список событий время его отбытия. Типичный пример списка событий показан на рис. 7.15.

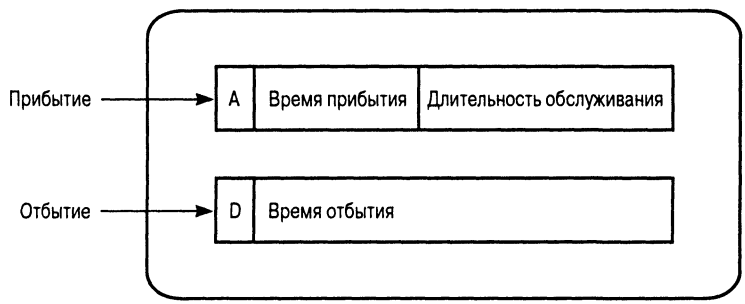


Рис. 7.15. Типичный пример списка событий

Теперь рассмотрим обработку наступившего события. Для этого необходимо выполнить следующие действия.

Две задачи, которые необходимо выполнить при обработке каждого события

- **Обновить очередь:** добавить или удалить посетителей.
- **Обновить список событий:** добавить или удалить событие.

По прибытии клиенты должны становиться в очередь. Текущий клиент, стоящий в начале очереди, в данный момент обслуживается, и именно этого посетителя мы удалим из системы следующим. Для этого естественно использовать абстрактную очередь. Единственная информация, которую нужно хранить в очереди, — время прибытия и длительность обслуживания.

Абстрактная очередь имитирует реальную очередь посетителей банка

Список событий упорядочен по времени, поэтому он не является очередью. Вскоре мы рассмотрим этот вопрос подробнее.

Список событий не является очередью

Итак, алгоритм обработки прибытия имеет следующий вид.

Алгоритм обработки прибытия

```
ДЛЯ ОБРАБОТКИ ПРИБЫТИЯ

    // Обновить список событий
    Удалить время прибытия клиента С из списка событий

    if (новый клиент С начинает операции немедленно)
    Вставить время отбытия клиента С в список событий
    (время отбытия = текущее время + длительность обслуживания)

    if (входной файл не исчерпан)
    Считать время нового прибытия и
        добавить его в список событий
    (время прибытия = время, указанное в файле)
```

Поскольку клиенты обслуживаются по очереди, новый посетитель всегда должен становиться в очередь, даже если она пуста. Затем из списка событий удаляется время прибытия нового клиента. Если новый посетитель обслуживается немедленно, время его отбытия вставляется в список событий. В заключение из списка событий считывается время нового прибытия, которое может происходить как до, так и после отбытия.

Новый посетитель всегда становится в очередь и обслуживается, когда окажется в ее начале

Алгоритм обработки отбытия

```
ДЛЯ ОБРАБОТКИ ОТБЫТИЯ

    // Обновить очередь
    Удалить клиента из начала очереди

    if (очередь не пуста)
    Начать обслуживание клиента, стоящего в начале очереди

    // Обновить список событий
    Удалить время отбытия из списка событий

    if (очередь не пуста)
    Вставить в список событий время отбытия
       обслуживаемого клиента
    (время отбытия = текущее время + длительность обслуживания)
```

После обработки отбытия не следует считывать время следующего прибытия из файла. Если конец файла еще не достигнут, список событий будет содержать время прибытия, предшествующее любому из моментов прибытия, указанных во входном файле.

Более глубокое изучение списка событий позволит лучше понять работу алгоритма. Список событий не имеет заранее определенной формы. Однако в данном моделировании возможны четыре конфигурации.

Четыре конфигурации списка событий в данном моделировании

- В исходном положении, когда время первого прибытия уже прочитано из входного файла, но еще не обработано, список событий содержит время прибытия *A*.
- Список событий: *A* (исходное состояние).
- При моделировании очереди в банке список событий обычно содержит время только двух событий: прибытия *A* и отбытия *D*. В зависимости от их очередности список может принимать две формы.
- Список событий: *D A* (общий случай — следующим событием является отбытие).
- Список событий: *A D* (общий случай — следующим событием является прибытие).
- Если первым является отбытие и после него очередь становится пустой, то новое отбытие не заменяет только что обработанного. В этом случае список событий принимает следующий вид.
- Список событий: *A* (после отбытия очередь становится пустой).
- Обратите внимание на то, что этот образец списка событий совпадает с исходным состоянием.
- Если первым является прибытие и после его обработки достигается конец входного файла, то список событий содержит только отбытие.
- Список событий: *D* (входная информация исчерпана).

Все остальные ситуации сводятся к одному из указанных состояний списка событий.

Время новых событий вставляется либо в начало списка, либо в его конец. Это зависит от времени наступления нового события и события, находящегося в списке. Допустим, что список событий содержит лишь время прибытия *A*, в то время как другой посетитель находится в начале очереди и начинает выполнение банковских операций. Для этого клиента нужно сгенерировать его отбытие *D*. Если он уходит из банка раньше, чем приходит новый клиент (прибытие *A*), то время отбытия *D* нужно поставить в список перед временем прибытия *A*. Однако если он уходит позже указанного срока, то время отбытия *D* нужно поставить в список после времени прибытия *A*. Если время этих событий совпадает, нужно установить их приоритет. В нашем решении мы произвольно ставим отбытие после прибытия.

Теперь можно объединить и уточнить фрагменты решения задачи и создать алгоритм, моделирующий банковскую очередь с помощью абстрактной очереди.

Окончательный псевдокод событийного моделирования

```
+simulate()
// Выполнить моделирование

   Создать пустую очередь bankQueue,
      имитирующую очередь в банке
```

```
   Создать пустой список событий eventList

   Имитировать первое прибытие,
      считав его время из входного файла
   Поместить время прибытия в список eventList

   while (список eventList не пуст)
   {
      newEvent = первое событие в списке eventList
      if (событие newEvent является прибытием)
          processArrival(newEvent, arrivalFile,
                         eventList, bankQueue)
      else
          processDeparture(newEvent, eventList, bankQueue)
   }  // Конец оператора while

+processArrival(in arrivalEvent:Event,
                in arrivalFile:File,
                inout anEventList:EventList,
                inout bankQueue:Queue)
// Обработка прибытия

   atFront = bankQueue.isEmpty() // Текущее состояние очереди

   // Обновить очередь bankQueue, вставив в нее клиента,
   // (событие arrivalEvent)
   bankQueue.enqueue(arrivalEvent)

   // Обновить список событий
   Удалить событие arrivalEvent из списка anEventList

   if (atFront)
   {
      // Очередь пуста, поэтому новый клиент становится первым
      // и обслуживается немедленно
      Вставить в список anEventList время отбытия
         нового клиента
      currentTime = currentTime + длительность обслуживания
   } // Конец оператора if

   if (конец входного файла не достигнут)
   {
     Прочитать из файла arrivalFile время следующего прибытия
     Добавить время события в список anEventList --
        оно указано во входном файле
   } // Конец оператора if

+processDeparture(in departureEvent:Event,
                  inout anEventList:EventList,
                  inout bankQueue:Queue)
// Обработка отбытия

   // Обновить очередь, удалив из нее первого клиента
   bankQueue.dequeue()
```

```
    // Обновить список событий
    Удалить время события departureEvent из списка anEventList
    if (!bankQueue.isEmpty())
    {
        // Начинается обслуживание первого клиента
        Вставить в список anEventList время отбытия
           первого клиента
        currentTime = currentTime + длительность обслуживания
} // Конец оператора if
```

На рис. 7.16 показаны результаты частичной трассировки этого алгоритма для данных, приведенных на стр. 27. Отвечая на вопрос 6, сформулированный в конце этой главы, читатели должны завершить эту трассировку полностью.

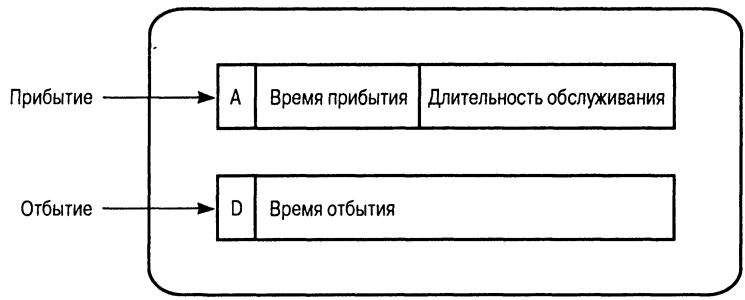


Рис. 7.16. Частичная трассировка алгоритма моделирования банковской очереди для данных, указанных на стр. 27

Фактически список событий сам является абстрактным типом данных. Анализируя псевдокод, приведенный выше, легко увидеть, что этот АТД должен предусматривать по крайней мере следующие операции.

Операции над абстрактным списком событий

```
+createEventList()
// Создать пустой список событий

+destroyEventList()
// Уничтожить список событий

+isEmpty():boolean {query}
// Определить, пуст ли список событий

+insert(in anEvent:Event)
// Вставить время события anEvent в список событий,
// так чтобы события оказались упорядоченными по времени
// Если прибытие и отбытие происходят одновременно,
// то прибытие должно предшествовать отбытию.

+delete()
// Удалить первый элемент списка событий.

+retrieve(out anEvent:Event)
// Поставить событие anEvent первым в списке событий.
```

В задании 6, указанном в конце главы, читателям предлагается полностью завершить описанное моделирование.

Резюме

1. Операции над абстрактной очередью выполняются в соответствии с правилом "первым вошел — первым вышел" (FIFO).
2. Для вставки и удаления элементов очереди необходим быстрый доступ к ее первому и последнему элементам. Следовательно, реализация очереди должна использовать кольцевой связанный список или линейный связанный список, имеющий два внешних указателя на его голову и хвост.
3. Реализация очереди в виде массива приводит к дрейфу элементов вправо. Это явление может привести к фиктивному переполнению очереди, хотя на самом деле она может не быть полностью заполненной. Для компенсации правого дрейфа можно сдвигать элементы массива. Наиболее эффективным решением этой проблемы является кольцевой массив.
4. Если очередь реализована в виде кольцевого массива, нужно отличать полностью заполненную и пустую очередь. Для этого предусматриваются счетчик элементов, признак `isFull` или дополнительная пустая ячейка.
5. При моделировании реальных систем очереди используются довольно часто. Событийное моделирование, описанное в этой главе, использовало абстрактную очередь для имитации реальной очереди посетителей банка.
6. Основным понятием моделирования является временной интервал моделирования. В рамках временного моделирования счетчик времени постоянно увеличивается на одну единицу измерения (секунду, минуту и т.п.), а при событийном моделировании время изменяется скачкообразно, т.е. от момента наступления одного события — до момента наступления следующего. Для реализации событийного моделирования необходимо предусматривать список, состоящий из моментов времени еще не наступивших событий. Этот список событий должен быть упорядочен по времени, так чтобы следующее событие всегда находилось в начале списка.

Предупреждения

1. При использовании линейного связанного списка, имеющего только указатель на голову очереди, операция вставки оказывается неэффективной, поскольку при вставке нового элемента приходится обходить весь связанный список от начала до конца. По мере удлинения очереди время обхода — и, следовательно, время выполнения вставки — возрастает.
2. Управление списком событий при моделировании реальных систем обычно представляет собой намного более сложную задачу, чем показано в рассмотренном примере. Например, если в банке работают несколько кассиров, структура списка событий намного усложняется.

Вопросы для самопроверки

1. Допустим, очередь содержит буквы `A`, `B`, `C` и `D` в алфавитном порядке. В каком порядке они будут удаляться оттуда?
2. Предположим, что в исходном положении очереди `queue1` и `queue2` были пусты. Опишите их состояние после выполнения следующих операций.

```
queue1.enqueue(1)
queue1.enqueue(2)
```

```
queue2.enqueue(3)
queue2.enqueue(4)
queue1.dequeue()
queue2.getFront(queueFront)
queue1.enqueue(queueFront)
queue1.enqueue(5)
queue2.dequeue(queueFront)
queue2.enqueue(6)
```

Сравните полученные результаты с ответом на вопрос 2 из главы 6.

3. Выполните трассировку алгоритма распознавания палиндромов, описанного в разделе "Простые применения абстрактной очереди", для каждой из указанных ниже строк.

   3.1. `abcda`

   3.2. `radar`

4. Определите наиболее подходящую структуру данных для каждой из указанных ниже ситуаций: 1) очередь; 2) стек; 3) список; 4) не подходит ни одна.

   4.1. Покупатели в гастрономе, получившие номерки.

   4.2. Алфавитный список имен.

   4.3. Неупорядоченный набор целых чисел.

   4.4. Блоки на блок-схеме рекурсивной функции.

   4.5. Список товаров, заполненный по мере их покупки.

   4.6. Записи на ленте кассового аппарата.

   4.7. Текстовый процессор, позволяющий исправлять опечатки с помощью клавиши <Backspace>.

   4.8. Программа, использующая откат.

   4.9. Список идей, записанных в хронологическом порядке.

   4.10. Самолеты, кружащие над аэропортом в ожидании разрешения на посадку.

   4.11. Люди, ожидающие очереди на покупку авиабилета.

   4.12. Работодатель, увольняющий сотрудника, нанятого последним.

5. Почему при моделировании банковской очереди нецелесообразно считывать весь входной файл, создавать список всех моментов прибытия и отбытия клиентов и только потом начинать моделирование?

6. Выполните полную трассировку алгоритма, моделирующего банковскую очередь (см. рис. 7.16), используя данные, указанных на стр. 27. Опишите состояние очереди и списка событий на каждом шаге алгоритма.

Упражнения

1. Проанализируйте алгоритм распознавания палиндромов, описанный в разделе "Простые применения абстрактной очереди". Нужно ли просматривать всю очередь и стек? Иными словами, можно ли сократить количество выполняемых циклов?

2. Рассмотрите язык

   $L=\{w\$w'$, где строка w может быть пустой или содержать символы, отличные от \$, а строка w'=reverse(w)},

описанный в главе 6 Напишите алгоритм распознавания выражений этого языка, используя очередь и стек. Вставляйте каждый символ строки w в очередь, а строки w' — в стек. Предполагается, что каждая входная строка содержит только один символ $.

3. Измените алгоритм преобразования инфиксных выражений в постфиксные, описанный в главе 6 так, чтобы для представления постфиксных выражений применялась очередь.

4. Напишите конструктор копирования для связанного списка, представляющего абстрактную очередь. *Подсказка*: проанализируйте конструктор копирования стека, описанный в главе 6.

5. Деструктор связанного списка, представляющего абстрактную очередь, постоянно вызывает функцию *dequeue*. Такой деструктор легко написать, но он не эффективен. Создайте другой деструктор, удаляющий связанный список, не вызывая функцию *dequeue*.

6. Проанализируйте реализацию очереди, использующей абстрактный список. Оцените эффективность операций вставки и удаления ее элементов, если абстрактный список реализован в одном из двух вариантов.

   6.1. В виде массива.

   6.2. В виде связанного списка.

7. Операция вывода на экран содержимого очереди может оказаться полезной для отладки программы. Включите операцию *display* в список операций над абстрактной очередью, так чтобы выполнялись следующие условия.

   7.1. Используются только операции над абстрактной очередью.

   7.2. Абстрактная очередь представлена в виде связанного списка.

8. Рассмотрим вариант абстрактной очереди, в которой элементы добавляются и удаляются с обоих концов. Обычно такая очередь называется **двусторонней** (doubly-ended queue), или **очередью с двусторонним доступом** (deque). Используйте эту очередь для ввода текста и последующего исправления опечаток с помощью клавиши <Backspace> (глава 6.). Каждое нажатие клавиши <Backspace> стирает символ, введенный последним. Напишите псевдокод, в котором введенная строка символов выводится на экран в прямом порядке.

9. Выполните вручную трассировку алгоритма моделирования банковской очереди для указанных ниже данных. Каждая строка данных содержит время прибытия и длительность обслуживания клиента. Покажите состояние очереди и списка событий на каждом шаге алгоритма.

```
 5   9
 7   5
14   5
30   5
32   5
    34   5
```

   Обратите внимание, что через 14 минут после начала моделирования прибытие и отбытие клиентов происходят одновременно.

10. Может ли список событий в этом алгоритме быть очередью, а также абстрактным или упорядоченным списком?

11. Проанализируйте решение задачи об авиаперелетах из одного города в другой (глава 6.) В алгоритме *searchS* стек можно заменить очередью. Иными словами, каждый вызов функции *push* можно заменить вызовом функции *enqueue*, каждый вызов функции *pop* — вызовом функции *dequeue*, а каждый вызов функции *getTop* — вызовом функции *getFront*. Выполните трассировку полученного алгоритма, при условии, что из пункта *P* нужно попасть в пункт *Z*. Карта полетов показана на рис. 6.10.

12. Как указывалось в главе 3, для формального определения операций над абстрактным типом данных можно использовать систему аксиом. Рассмотрите аксиомы для абстрактной очереди, приведенные ниже. Здесь *aQueue* означает произвольную очередь, а *item* — произвольный элемент очереди. Для простоты будем считать, что функция *getTreat* возвращает элемент, стоящий в начале очереди.

```
(aQueue.createQueue()).isEmpty() = true
(aQueue.enqueue(item)).isEmpty() = false
(aQueue.createQueue()).dequeue() = aQueue.createQueue()
(или ошибка)
((aQueue.createQueue()).enqueue(item)).dequeue()
   = aQueue.createQueue()
aQueue.isEmpty() = false ⇒
   (aQueue.enqueue(item)).dequeue() =
(aQueue.dequeue()).enqueue(item)
(aQueue.createQueue()).getFront() = error
aQueue.isEmpty() = false ⇒
   (aQueue.enqueue(item)).getFront() = aQueue.getFront()
```

    12.1. Обратите внимание на рекурсивный характер функции *getFront*. Что является ее базисом? В чем заключается шаг рекурсии? Зачем выполняется проверка *isEmpty*? Почему операция *getFront* для очереди имеет рекурсивный характер, а операция *getTop* для стека — нет.

    12.2. Представление стека в виде последовательности операций *push*, не содержащей ни одной операции *pop*, называется каноническим. (См. упражнение 12 из главы 6.) Будет ли каноническим представление абстрактной очереди, использующее только последовательность операций *enqueue*? Иными словами, существует ли для каждой очереди эквивалентная ей очередь, записанная только с помощью последовательности операции *enqueue*? Обоснуйте свой ответ.

Задания по программированию

1. Напишите реализацию очереди в виде кольцевого связанного списка. Вам понадобится один указатель на хвост очереди. Сравните полученную реализацию с линейным связанным списком, в котором используется два внешних указателя. Какая реализация проще? Какая реализация понятнее? Какая реализация более эффективна?
2. Напишите реализацию очереди в виде динамического кольцевого массива.
3. Проанализируйте реализацию очереди, приведенную в тексте. Вместо подсчета ее элементов примените признак *isFull*, позволяющий отличить полностью заполненную очередь от пустой.

4. В главе описана еще одна реализация очереди, в которой не предусматриваются никакие специальные члены, типа счетчика элементов *count* или признака *isFull* (см. задание 3). Для распознавания пустых и полностью заполненных очередей в массиве *items* резервируется одна пустая ячейка, так что он состоит из *MAX_QUEUE*+1 ячеек, из которых на самом деле используются только *MAX_QUEUE* ячеек. Благодаря этому индекс *front* никогда не превышает индекса *back*. Очередь оказывается полной, когда индекс *front* становится равным величине (*back*+1)%(*MAX_QUEUE*+1), и пустой, когда индексы *front* и *back* равны между собой.

   4.1. Почему такая реализация для очереди, состоящей из элементов стандартного типа, занимает столько же памяти, сколько и реализации, предусматривающие счетчик элементов и признак *isFull*?

   4.2. Реализуйте этот способ, используя массив.

5. В упражнении 8 упоминалась двусторонняя очередь, или очередь с двусторонним доступом. В такой очереди элементы вставляются и удаляются с обоих концов. Реализуйте двустороннюю очередь с помощью связанного списка и массива.

6. Создайте программу для событийного моделирования очереди посетителей банка. Очередь моментов прибытия посетителей можно отождествить с очередью самих клиентов. Сохраните моменты прибытия и отбытия посетителей в связанном списке событий, упорядоченном по времени.

   Задайте во входном текстовом файле моменты прибытия клиентов и длительности их обслуживания, записав их попарно. Моменты прибытия упорядочиваются по возрастанию.

   Программа должна подсчитывать посетителей и определять общее время ожидания в очереди. Это позволит вычислить среднее время ожидания посетителей.

   Выведите на экран результаты трассировки алгоритма и полученные статистические данные (общее количество посетителей и среднее время ожидания в очереди). Например, в левой колонке таблицы, приведенной ниже, записаны входные данные, а в правой — результаты моделирования.

Входные данные	Результаты
1 5	Начало моделирования
2 5	Обработка прибытия в момент: 1
4 5	Обработка прибытия в момент: 2
20 5	Обработка прибытия в момент: 4
22 5	Обработка отбытия в момент: 6
24 5	Обработка отбытия в момент: 11
26 5	Обработка отбытия в момент: 16
28 5	Обработка прибытия в момент: 20
30 5	Обработка прибытия в момент: 22
88 3	Обработка прибытия в момент: 24
	Обработка отбытия в момент: 25
	Обработка прибытия в момент: 26
	Обработка прибытия в момент: 28
	Обработка прибытия в момент: 30

```
Обработка отбытия в момент:  30
Обработка отбытия в момент:  35
Обработка отбытия в момент:  40
Обработка отбытия в момент:  45
Обработка отбытия в момент:  50
Обработка прибытия в момент:  88
Обработка отбытия в момент:  91
Конец моделирования

Результаты:
   Общее количество посетителей: 10
   Среднее время ожидания в очереди: 5.6
```

7. Усовершенствуйте программу для событийного моделирования из задания 6 при следующих предположениях.

   7.1. Добавьте операцию вывода на экран списка событий и воспользуйтесь ею для проверки результатов трассировки, выполненной в упражнении 9.

   7.2. Добавьте несколько новых статистических показателей. Например, вычислите максимальное время ожидания, а также среднюю и максимальную длину очереди.

   7.3. Выполните моделирование банковской очереди, если в банке работают три кассира.

   - Предусмотрите три очереди, по одной к каждому кассиру.
   - Укажите правило выбора очереди вновь прибывшим клиентом (например, посетитель всегда выбирает самую короткую очередь).
   - Предусмотрите три разных вида отбытия для клиентов из разных очередей.
   - Сформулируйте правила обработки одновременных событий.
   - Выполните моделирование банковской очереди, пользуясь разными алгоритмами и на разных входных данных. Сравните полученные результаты.

   7.4. Допустим, что вместо трех отдельных очередей (по одной к каждому кассиру) в банке существует одна очередь сразу к трем кассирам. Посетитель, стоящий первым, ожидает, когда освободится один из кассиров. Выполните задание 7.3 при новых условиях. Выполните моделирование банковской очереди, пользуясь разными алгоритмами и на разных входных данных. Сравните полученные результаты (среднее и максимальное время ожидания). Какой вариант эффективнее: с одной или тремя очередями?

8. Клиенты компании Motor Vehicle Department (MVD) испытывают затруднения. Они хотели бы ускорить процедуру покупки автомобиля. В настоящее время в компании принят следующий порядок работы с клиентами.

   - Посетитель, вошедший в помещение, должен расписаться в журнале.
   - Записавшись на прием, он становится либо в очередь на оформление регистрации, либо в очередь на получение лицензии.

- Выполнив необходимые формальности, посетитель становится в очередь к кассиру.
- Дождавшись очереди к кассиру, посетитель узнает, что его чек должен быть заверен. Для этого он должен обратиться к сотруднику, заверяющему чеки, а затем вновь отстоять очередь в кассу.

Оцените описанную ситуацию с помощью методов событийного моделирования.

Каждая строка входного файла должна содержать следующую информацию.

- Код операции (L — получение лицензии, R — регистрация).
- Способ оплаты ($ — наличными, C — чеком).
- Время прибытия (целое число).
- Имя посетителя

Напишите спецификации каждого события (когда, кто, что и т.д.).

Выведите на экран результаты моделирования.

- Общее количество оформленных лицензий и среднее время ожидания (от момента прибытия до момента оплаты).
- Общее количество регистраций и среднее время ожидания (от момента прибытия до момента оплаты).

Предусмотрите в своей программе следующие детали.

- Определите следующие события: прибытие, запись, оформление лицензии, регистрация и сражение с кассиром (оплата или направление к сотруднику, заверяющему чеки).
- В случае одновременного наступления разных событий порядок их обработки определяется записями в списке событий (т.е. прибытия имеют наивысший приоритет).
- Допустим, что операции занимают следующие интервалы времени.

Запись	10 с
Оформление лицензии	90 с
Регистрация автомобиля	60 с
Оплата	30 с
Направление на проверку чека	10 с

- Как это ни странно, будем считать, что клиенты, ожидающие оформления лицензии, вызываются в алфавитном порядке. Однако эти клиенты не покидают очередь, если их обслуживание уже началось.
- Предположим, для простоты, что чеки подтверждаются мгновенно. Следовательно, если клиент стоит в начале очереди с незаверенным чеком, он автоматически перемещается в конец очереди, но уже держа в руках заверенный чек.

Особенности языка C++

В этой главе ...

Еще раз о наследовании
- Открытое, закрытое и защищенное наследование
- Отношения "является", "содержит" и "подобен"

Виртуальные функции и позднее связывание
- Абстрактные базовые классы

Дружественные функции и классы

Новая реализация абстрактного и упорядоченного списка
- Реализации абстрактного упорядоченного списка на основе абстрактного списка

Шаблонные классы

Перегруженные операторы

Итераторы
- Реализация абстрактного списка с помощью итераторов

Резюме

Предупреждения

Вопросы для самопроверки

Упражнения

Задания по программированию

Введение. В языке C++ абстракция данных осуществляется с помощью классов, инкапсулирующих абстрактные типы данных и операции над ними. Однако объектно-ориентированный подход выходит далеко за рамки простой инкапсуляции. Применяя наследование и полиморфизм, в языке C++ можно на основе существующих классов создавать новые. В главе описываются способы создания компонентов программного обеспечения, пригодных к повторному использованию. На эту тему можно распространяться очень много, поэтому данную главу следует считать лишь введением.

Еще раз о наследовании

Размышляя о наследовании, вы, наверное, представляете себе завещание, по которому ваш богатый дядюшка оставил вам миллион долларов. Не надейтесь! В объектно-ориентированном мире все совершенно иначе. **Наследование** (inheritance) — это способность класса приобретать свойства ранее определенного класса. Эти свойства напоминают генетические особенности, полученные вами от родителей: некоторые из них остаются неизменными, некоторые похожи, но немного отличаются от предков, а некоторые — совершенно новые.

Класс может наследовать особенности функционирования и структуру другого класса

Фактически наследование — это отношение между классами. Один класс может наследовать поведение и структуру другого класса. На рис. 8.1 показаны отношения между разнообразными часами. Например, к электронным часам относятся часы, вмонтированные в панель вашего автомобиля, часы на вывеске банка и часы, встроенные в микроволновую печь. Все электронные часы имеют одинаковую структуру и выполняют одинаковые операции.

```
Установить время
Изменить показатели
Показать время
```

Электронный будильник — это разновидность электронных часов

Электронный будильник — это разновидность электронных часов, которые, помимо всего прочего, умеют выполнять новые операции.

```
Установить будильник
Включить будильник
Включить звонок
Выключить звонок
```

Иными словами, электронный будильник имеет структуру обычных электронных часов, но кроме этого умеет кое-что еще.

Группу электронных часов и группу электронных будильников можно представить в виде классов. Класс электронных будильников является **производным** (derived), или **подклассом** (subclass) класса электронных часов. Класс электронных часов является **базовым** (base), или **суперклассом** (superclass) по отношению к электронным будильникам.

Производный класс наследует члены своего базового класса

В языке C++ производный класс **наследует** (inherits) все члены своего базового класса, за исключением конструкторов и деструктора. Иными словами, производный класс содержит в себе все данные-члены и функции-члены базового класса, добавляя к ним новые члены, определенные в нем самом. Кроме того, производный класс может изменять любую наследуемую функцию-член. Например, как следует из рис. 8.1, часы с кукушкой являются наслед-

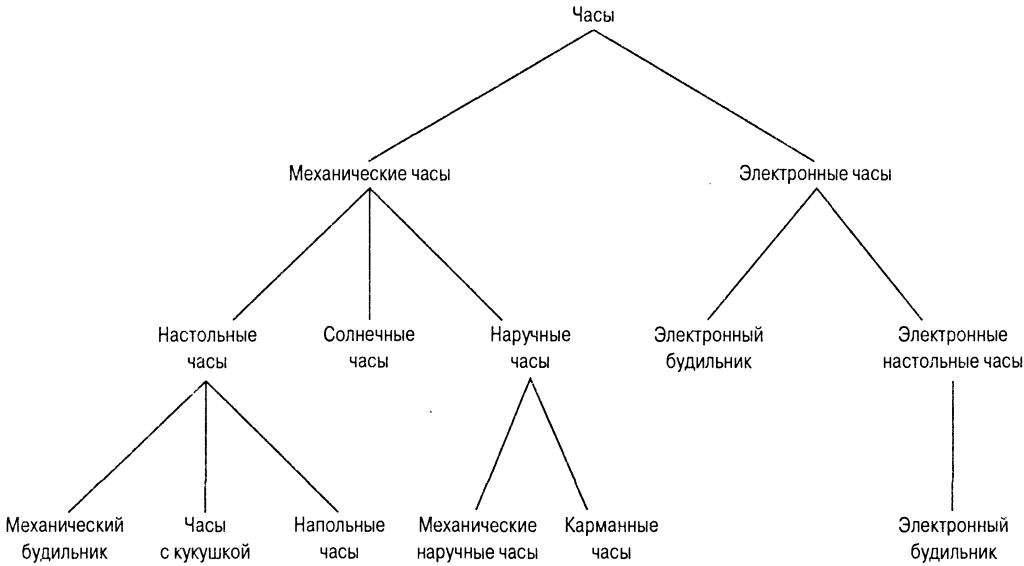


Рис. 8.1. Наследование: отношения между часами

ником механических часов. Они наследуют структуру и поведение механических часов, но изменяют способ регистрации времени, предусматривая кукушку.

Иногда производный класс имеет несколько базовых классов. Например, как показано на рис. 8.2, класс радиочасов можно вывести из класса электронных часов и класса радиоприемников. Эти отношения называются **множественным наследованием** (multiple inheritance). В дальнейшем этот вид наследования не рассматривается.

Наследование позволяет использовать существующие классы

Наследование позволяет **повторно использовать** (reuse) компоненты программного обеспечения при определении нового класса. Например, структуру и реализацию механических часов можно использовать при создании часов с кукушкой. Рассмотрим более простой пример, демонстрирующий детали такого повторного использования, и механизм наследования, предусмотренный в языке C++.

Наследование облегчает добавление новых свойств к существующему объекту

В главе 3 мы рассматривали в качестве объектов шары, позволяющие имитировать волейбольный и футбольный мячи. Разрабатывая класс шаров, `Ball`, каждый шар можно считать сферой, имеющей какое-то имя. В этом определении очень важно, что класс `Sphere` — класс сфер — уже существует. Таким образом, если класс `Sphere` рассматривать в качестве базового, то класс `Ball` можно реализовать, не описывая заново свойства сферы. Напомним определение класса `Sphere` из главы 3.

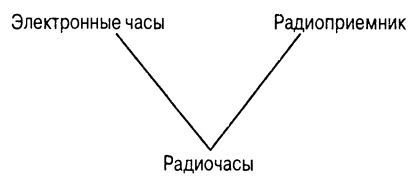


Рис. 8.2. Множественное наследование

```
class Sphere
{
public:
   // Конструкторы:
   Sphere();
   Sphere(double initialRadius);
   // Конструктор копирования и деструктор
   // генерируются компилятором

   // Операции класса Sphere:
   void setRadius(double newRadius);
   double getRadius() const;
   double getDiameter() const;
   double getCircumference() const;
   double getArea() const;
   double getVolume() const;
   void displayStatistics() const;

private:
   double theRadius; // радиус сферы
}; // Конец класса
```

Производный класс может содержать новые члены, кроме наследуемых

Производный класс *Ball* может наследовать все члены класса *Sphere* — за исключением конструкторов и деструктора, — а также вносить некоторые изменения. Он может содержать имя шара и функции-члены, позволяющие получать доступ к этому имени, а также задавать или изменять имя и радиус шара. Кроме того, производный класс может изменить функцию *displayStatistics*, заставляя ее выводить на экран имя шара, помимо параметров сферы.

Производный класс может переопределять унаследованную функцию-член базового класса

В производный класс можно добавлять сколько угодно новых членов. Хотя доступа к закрытым данным-членам базового класса у производного класса нет, остальные функции-члены можно **переопределить** (redefine). Говорят, что функция-член производного класса переопределяет функцию-член базового класса, если обе эти функции имеют одинаковые параметры и одинаковый тип возвращаемого значения. Невозможно переопределить функцию, изменив лишь тип возвращаемого значения. Отношения между классами *Sphere* и *Ball* показаны на рис. 8.3.

Класс *Ball* можно определить следующим образом.

```
class Ball: public Sphere
{
public:
   // Конструкторы:
   Ball();

   Ball(double initialRadius, const string initialName);
   // Создает объект класса Ball, имеющий радиус initialRadius
   // и имя initialName

   // Конструктор копирования и деструктор
   // генерируются компилятором

   // Дополнительные или измененные операции:
   void getName(string currentName) const;
```

```
    // Определяет имя объекта класса Ball.

    void setName(const string newName);
    // Задает (изменяет) имя существующего объекта класса Ball

    void resetBall(double newRadius, const string newName);
    // Задает (изменяет) радиус и имя существующего объекта
    // класса Ball, используя аргументы newRadius и newName,
    // соответственно

    void displayStatistics() const;
    // Выводит параметры объекта класса Ball

private:
    string theName; // Имя объекта класса Ball
}; // end class
```

Двоеточие и ключевое слово *public*, поставленные после слов *class Ball*, указывают, что класс *Sphere* является базовым по отношению к классу *Ball* или, что то же самое, класс *Ball* является производным от класса *Sphere*.

> Экземпляр производного класса обладает всеми свойствами базового класса

Каждый экземпляр класса *Base* имеет два члена — унаследованную переменную *theRadius* и новую переменную *theName*. Поскольку экземпляр производного класса может вызывать любую открытую функцию-член базового класса, экземпляр класса *Ball* содержит все методы, определенные в классе *Sphere*: новые конструкторы, новый деструктор, генерируемый компилятором, новые функции-члены *getName*, *setName* и *resetName*, а также переопределенный метод *displayStatistics*. Несмотря на то что экземпляр производного класса содержит все копии унаследованных данных-членов, код наследуемых функций-членов не копируется.

> Производный класс наследует закрытые члены базового класса, но не имеет к ним прямого доступа

Производный класс не имеет прямого доступа к закрытым членам базового класса, несмотря на то что они наследуются. Наследование не открывает доступ к закрытым членам. Это вполне естественно: ведь вы можете унаследовать запертый склеп, не имея ключей к нему. В нашем примере переменная *theRadius*, являющаяся закрытым членом класса *Sphere*, доступна исключительно в определении класса *Sphere*, но не в

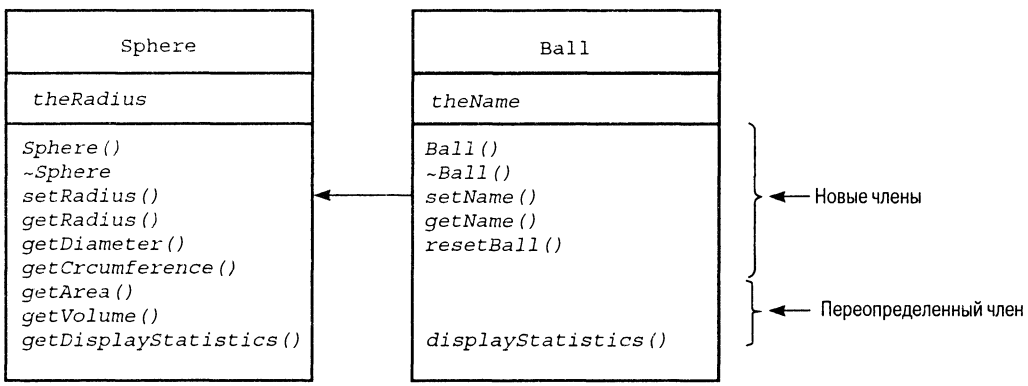


Рис. 8.3. Производный класс Ball наследует члены базового класса Sphere, а также переопределяет и добавляет новые функции-члены

классе *Ball*. Однако внутри определения класса *Ball* можно вызывать открытые функции-члены класса *Sphere*, например, функции *setRadius* и *getRadius*, получая значение переменной *theRadius* косвенным путем.

Дополнительные функции-члены класса *Ball* реализуются следующим образом.

```
Ball::Ball() : Sphere()
{
   setName("");
} // Конец конструктора по умолчанию

Ball::Ball(double initialRadius,
           const string initialName)
           : Sphere(initialRadius)
{
   setName(initialName);
} // Конец конструктора

void Ball::getName(string currentName) const
{
   currentName = theName;
} // Конец функции getName

void Ball::setName(const string newName)
{
   theName = newName;
} // Конец функции setName

void Ball::resetBall(double newRadius,
                     const string newName)
{
   setRadius(newRadius);
   setName(newName);
} // Конец функции resetBall

void Ball::displayStatistics() const
{
   cout << "Параметры объекта " << theName << ":";
   Sphere::displayStatistics();
} // Конец функции displayStatistics
```

Функции-члены производного класса могут вызывать открытые функции-члены базового класса

Конструкторы класса *Ball* вызывают соответствующие конструкторы класса *Sphere*, используя синтаксис инициализатора. В реализации класса *Ball* можно вызывать функции-члены, наследуемые от класса *Sphere*. Например, новая функция-член *resetBall* вызывает наследуемую функцию *setRadius*. Кроме того, функция *displayStatistics* из класса *Ball* вызывает унаследованную версию функции *displayStatistics*. Этот факт отмечается синтаксической конструкцией *Sphere::displayStatistics*. Такое обозначение позволяет различать два варианта одной и то же функции, принадлежащей разным классам. Итак, доступ к члену базового класса, даже если он был переопределен, можно получить с помощью оператора разрешения области видимости *::*.

Клиенты производного класса могут вызывать открытые функции-члены базового класса

Клиенты производного класса могут вызывать открытые функции-члены базового класса. Рассмотрим вызов:

```
Ball myBall(5.0, "Волейбол");
```

Функция *myBall.getDiameter()* возвращает диаметр объекта *myBall*, равный 10.0 (удвоенный радиус объекта *myBall*), используя функцию-член *getDiameter*, унаследованную классом *Ball* от класса *Sphere*. Если новая функция имеет точно такое же имя, что и ее предок, — например, функция *displayStatistics*, — экземпляры нового класса будут вызывать новую функцию, а экземпляры базового класса — старую функцию. Следовательно, если объект *mySphere* является экземпляром класса *Sphere*, вызов *mySphere.displayStatistics()* относится к функции *displayStatistics* из класса *Sphere*, а вызов *myBall.displayStatistics()* — к функции *displayStatistics* из класса *Ball*. Эта ситуация проиллюстрирована на рис. 8.4.

Поскольку решение, какой из вариантов метода вызвать, принимается компилятором, а не в ходе выполнения программы, такой процесс называется **ранним** (early), или **статическим** (static) связыванием.

Порядок выполнения конструкторов и деструкторов

Конструктор производного класса выполняется после конструктора базового класса. Например, конструктор класса *Ball* выполняется после конструктора класса *Sphere*. Деструктор производного класса выполняется перед деструктором базового класса. Например, деструктор класса *Ball* выполняется перед деструктором класса *Sphere*. Это относится ко всем конструкторам и деструкторам, в том числе и автоматическим.

Открытые, закрытые и защищенные разделы класса. Кроме открытых и закрытых разделов, класс имеет **защищенный** (protected) раздел. Члены, помещенные в защищенный раздел, скрыты от клиентов класса, но доступны его наследникам. Иными словами, производный класс может ссылаться на закрытые члены своего базового класса, а клиенты базового или производного класса — нет.

Например, класс *Sphere* имеет закрытый член *theRadius*, к которому класс *Ball* не имеет прямого доступа. Если вместо этого переменную *theRadius* объявить защищенной, то класс *Ball* получит к ней прямой доступ. Однако клиенты классов *Ball* и *Sphere* не могут непосредственно ссылаться на переменную *theRadius*. Доступ к открытым, закрытым и защищенным членам класса проиллюстрирован на рис. 8.5.

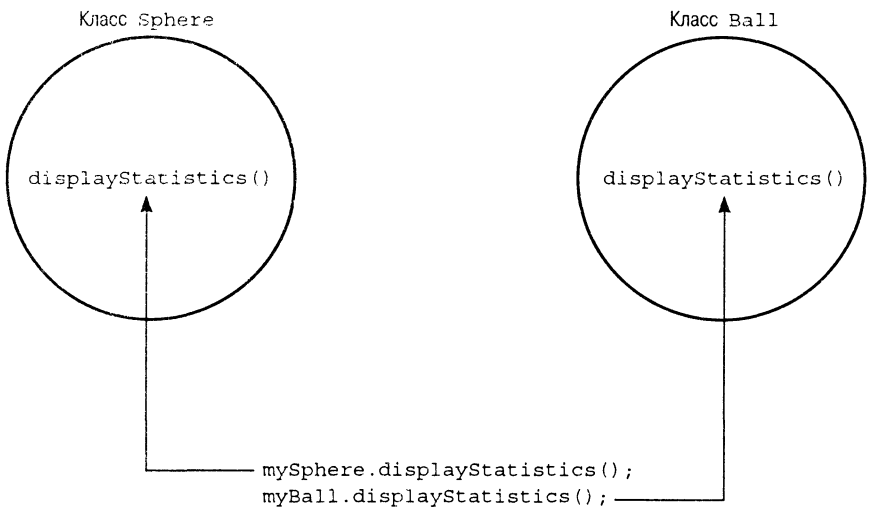


Рис. 8.4. Раннее, или статическое связывание: компилятор определяет, какой вариант метода вызвать

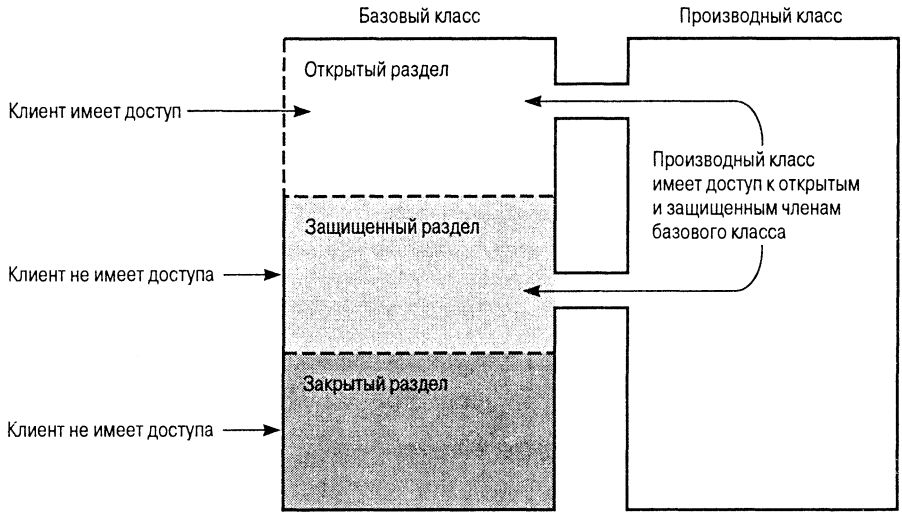


Рис. 8.5. Доступ к открытым, закрытым и защищенным разделам класса из клиента и производного класса

Как правило, данные-члены класса должны быть закрытыми

Правила хорошего стиля требуют закрывать все данные-члены класса, обеспечивая при необходимости косвенный доступ к ним с помощью открытых или защищенных функций-членов. В то время как открытые члены класса доступны всем, защищенные члены доступны лишь собственным членам класса (и **друзьям**[1] (friends)), а также функциям-членам (и друзьям) производного класса.

ОСНОВНЫЕ ПОНЯТИЯ

Категории членов класса

1. Открытые члены доступны всем.
2. Закрытые члены доступны только функциям-членам класса (и его друзьям).
3. Защищенные члены доступны только функциям-членам базового и производных классов (и их друзьям).

Открытое, закрытое и защищенное наследование

Существует три вида наследования, позволяющих управлять доступом к наследуемым членам в иерархии производных классов. Независимо от этого производный класс имеет доступ ко всем открытым и защищенным членам базового класса, но не к его закрытым членам. Начнем с определения производного класса.

```
class DerivedClass : kind BaseClass
```

Здесь параметр *kind* может заменяться ключевым словом `public`, `private` или `protected`. Наследование, которое рассматривалось до сих пор, всегда было открытым. Перечислим все возможные виды наследования.

[1] *Друзьями* класса называются функции, не являющиеся его членами, но имеющие доступ к его закрытым и защищенным членам. Эта тема обсуждается ниже.

ОСНОВНЫЕ ПОНЯТИЯ

Виды наследования

1. **Открытое наследование.** Открытые и защищенные члены базового класса остаются, соответственно, открытыми и защищенными членами производного класса.
2. **Защищенное наследование.** Открытые и защищенные члены базового класса становятся защищенными членами производного класса.
3. **Закрытое наследование.** Открытые и защищенные члены базового класса становятся закрытыми членами производного класса.

   Во всех случаях закрытые члены базового класса остаются закрытыми и недоступными из производных классов.

Среди всех перечисленных видов наследования открытое наследование является наиболее важным. Именно оно будет применяться во всех последующих главах. Открытое наследование применяется для расширения определения класса. Закрытое наследование необходимо при определении одного класса в терминах другого. Защищенное наследование используется редко, поэтому мы не будем его рассматривать.

Отношения "является", "содержит" и "подобен"

Наследование устанавливает между классами отношения "предок-наследник". Между классами могут существовать и другие отношения. Когда программист создает новый класс, используя уже существующий, важно идентифицировать их взаимоотношения. Это позволит правильно выбрать вид наследования, которое наилучшим образом отражает отношения между классами. Существует три основных вида отношения: "является" ("is-a"), "содержит" ("has-a") и "подобен" ("as-a").

Открытое наследование определяет отношение "является"

Отношение "является". Ранее в этой главе мы использовали открытое наследование, чтобы создать класс *Ball* на основе класса *Sphere*. Открытое наследование следует применять лишь тогда, когда между базовым и производным классами существует отношения "является". В нашем примере шар "является" сферой, как показано на рис. 8.6. Иными словами, все, что является истинным в отношении объектов класса *Sphere*, справедливо и для объектов класса *Ball*. Это свойство называется **совместимостью типов объектов** (object type compatibility). В принципе производный класс совместим по типу со всеми своими предками. Таким образом, объект производного класса можно применять вместо объекта базового класса, но не наоборот.

В частности, в качестве формального аргумента функции можно использовать экземпляр производного, а не базового класса. Допустим, что в программе используются классы *Sphere* и *Ball*. Рассмотрим функцию, не являющуюся членом какого-либо класса.

```
void displayDiameter(Sphere thing)
{
   cout << "Диаметр равен " << thing.getDiameter() << ".\n";
} // Конец функции displayDiameter
```

Определим объекты *mySphere* и *myBall* следующим образом.

```
Sphere mySphere(2.0);
Ball myBall(5.0, "Волейбол");
```

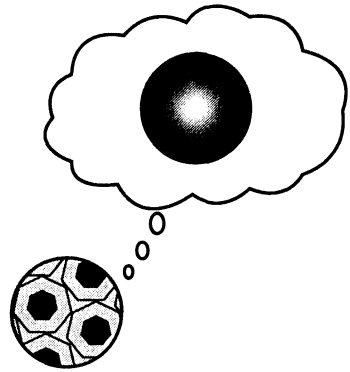


Рис. 8.6. Шар "является" сферой

В этом случае допускаются такие вызовы функции.

```
displayDiameter(mySphere); // Диаметр объекта mySphere
displayDiameter(myBall);   // Диаметр объекта myBall
```

Поскольку шар является сферой, его можно использовать вместо нее

Первый вызов ничем не примечателен, поскольку типы фактического параметра *mySphere* и формального аргумента *thing* совпадают. Второй вызов более интересен: тип фактического параметра *myBall* является производным от типа формального аргумента *thing*. Поскольку шар является сферой, он ведет себя как сфера. Иными словами, объект *myBall* способен имитировать сферу, и его можно применять вместо сферы. Обратите внимание, что совместимость типов относится к аргументам, передаваемым как по значению, так и по ссылке.

Если между классами нет отношения "является", открытое наследование применять не следует

Отношение "содержит". Шариковая авторучка в качестве пишущего узла содержит шарик, как показано на рис. 8.7. Если бы мы захотели использовать класс *Ball* для определения класса *Pen*, то не должны были бы применять открытое наследование, поскольку шариковая ручка шаром не является. Наследование вообще не применяется для отношения "содержит". Вместо этого нужно определить в классе *Pen* переменную-член *point*, имеющую тип *Ball*.

```
class pen
{
   ...
private:
   Ball point;
};
```

Напомним, что переменная-член может быть объектом любого класса, кроме определяемого. В данном случае она не может являться экземпляром класса *Pen*. Таким образом, экземпляр класса *Pen* "содержит" внутри экземпляр класса *Ball*. Отношение "содержит" ("has-a") иногда называют отношением **включения** (containment).

Отношение "содержит", или включение означает, что некий класс содержит в качестве члена объект другого класса

Конструкторы объекта, являющегося членом класса, выполняются до конструкторов данного класса. Например, конструктор класса *Ball* выполняется до конструктора класса *Pen*. Деструктор объекта, являющегося членом класса, выполняется после деструктора класса. Например, деструктор класса *Ball* выполняется после деструктора класса *Pen*.

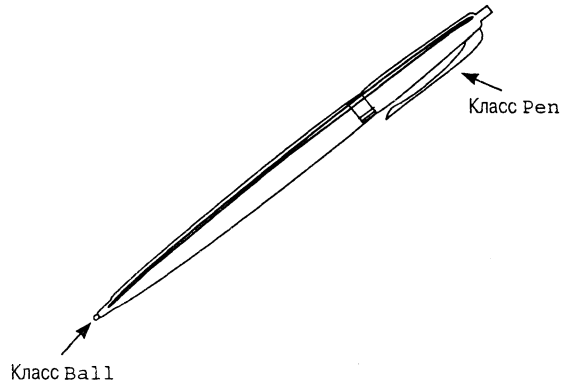


Рис. 8.7. Шариковая авторучка "содержит" шар

Мы уже сталкивались с отношениями включения между классами. Например, в главе 6 показана реализация класса *Stack*, использующая абстрактный список в качестве своих элементов. В главе 7 рассматривалась аналогичная реализация абстрактной очереди. Класс *Stack* содержит закрытый член *aList*, имеющий тип *List*. Это означает, что экземпляр класса *Stack* содержит внутри экземпляр класса *List* в качестве своего элемента.

Отношение "подобен". В заключение рассмотрим последний вид отношений между классами. В частности, вернемся к отношениям между классами *Stack* и *List*. Используя закрытое наследование, абстрактный стек можно реализовать *как* список. Начнем определение класса следующим образом.

```
class Stack: private List
```

Открытые и защищенные члены класса *List* станут закрытыми членами класса *Stack*. Таким образом, в реализации класса *Stack* элементами стека можно манипулировать с помощью методов класса *List*. Однако ни наследники, ни клиенты класса *Stack* не имеют доступа к членам класса *List*. Следовательно, список, лежащий в основе стека, скрыт от его клиентов.

Ни отношение "содержит", ни отношение "подобен" не допускают применения открытого наследования. Если классу нужен доступ к защищенным членам другого класса, либо возникла необходимость переопределить функции данного класса, используется отношение "подобен", т.е. закрытое наследование. В других случаях более предпочтительным и простым является отношение включения. Позднее в этой главе мы рассмотрим реализацию абстрактного упорядоченного списка с помощью указанных трех отношений.

Виртуальные функции и позднее связывание

Как показано выше, если объект *mySphere* является экземпляром класса *Sphere*, а объект *myBall* — экземпляром класса *Ball*, вызов *mySphere.displayStatistics()* относится к функции *displayStatistics* из класса *Sphere*, а вызов *myBall.displayStatistics()* относится к функции *displayStatistics* из класса *Ball* (см. рис. 8.4). Правильный вариант функции *displayStatistics* выбирается компилятором. Этот выбор, представляющий собой пример раннего связывания, невозможно изменить в ходе выполнения программы.

Раннее связывание может породить проблемы. Рассмотрим в качестве примера объявление

```
Sphere *spherePtr = &mySphere;
```

Здесь указатель *spherePtr* ссылается на объект *mySphere*. Как и следовало ожидать, вызов

```
spherePtr->displayStatistics();
```

относится к функции *displayStatistics* из класса *Sphere*. К сожалению, если указатель *spherePtr* ссылается на объект *myBall*, например,

```
spherePtr = &myBall;
```

то вызов

```
spherePtr->displayStatistics();
```

продолжает относиться к функции *displayStatistics* из класса *Sphere*, а не из класса *Ball*. В этом случае компилятор связал вызов метода с типом указателя *spherePtr*, а не с типом объекта, на который он ссылается.

Простое переопределение функции *displayStatistics* в классе *Ball* ничего не дает. Нужно как-то объяснить компилятору, что классы, производные от класса *Sphere*, могут модифицировать функцию *displayStatistics*. Это достигается с помощью механизма **виртуальных функций** (virtual functions). Для того чтобы сделать функцию-член виртуальной, нужно просто указать ключевое слово *virtual* перед ее объявлением в определении базового класса. Например, описание класса *Sphere* изменится следующим образом.

```
class Sphere
{
public:
   // Все как раньше, за исключением функции displayStatistics
   . . .
   virtual void displayStatistics() const;
   . . .
}; // Конец класса
```

Реализация функции *displayStatistics* остается без изменений.

Теперь, если указатель *spherePtr* ссылается на объект *myBall*, оператор

```
spherePtr->displayStatistics();
```

вызывает функцию *displayStatistics* из класса *Ball*. Таким образом, правильная версия функции-члена выбирается в ходе выполнения программы, а не на этапе компиляции, и ее выбор зависит от типа объекта, на который ссылается указатель *spherePtr*. Эта ситуация называется **поздним** (late), или **динамическим** (dynamic), связыванием, а вызываемая функция — **полиморфной** (polymorphic). Полиморфизм (буквально "много форм") позволяет выбирать вызываемую функцию в ходе выполнения программы. Таким образом, результат конкретной операции зависит от объектов, к которым применяется эта операция. В таких случаях говорят, что версия функции *displayStatistics* из класса *Ball* **замещает** (override) версию этой функции из класса *Sphere*.

Позднее связывание означает, что правильная версия функции-члена выбирается в ходе выполнения программы

Виртуальная функция-член производного класса может замещать виртуальную функцию-член базового класса, если они имеют одинаковые объявления. Замещение функций похоже на их переопределение. Однако замещаться могут только виртуальные функции.

Виртуальную функцию можно замещать

Если функция-член базового класса является виртуальной, она считается виртуальной в любом производном классе

Если нам нужно, чтобы в классе *Ball* замещалась функция *displayStatistics*, его можно определить точно так же, как и раньше. Иными словами, в производном классе можно пропускать ключевое слово *virtual*. Любая функция-член виртуального класса, имеющая одинаковое объявление с виртуальной функцией базового класса, — например, функция *displayStatistics*, — неявно считается виртуальной. И все же ключевое слово *virtual* желательно указывать явно. Это больше соответствует хорошему стилю программирования.

Рассмотрим более тонкие примеры раннего и позднего связывания. Допустим, нам нужно, чтобы поведение метода *getArea* из класса *Ball* отличалось от метода *getArea* из класса *Sphere*. Для этого достаточно заместить функцию *getArea* в классе *Ball* так, чтобы она, например, вычисляла площадь поперечного сечения шара. Итак, добавим функцию *getArea* в качестве открытого члена класса *Ball*.

```
double getArea() const;
```

Реализация функции *getArea* такова.

```
double Ball::getArea() const
{
   double r = getRadius();
   return PI * r * r; // PI — глобальная константа
} // Конец функции getArea
```

Поскольку функция-член *getArea* из класса *Sphere* замещается, ее следует сделать виртуальной. Однако посмотрим сначала, что случится, если этого не сделать. Допустим, класс *Sphere* определен следующим образом.

```
class Sphere
{
public:
   . . . // Все, как раньше

   double getArea() const;
   virtual void displayStatistics() const;
   . . .
}; // Конец класса

class Ball: public Sphere
{
public:
   . . . // Все, как раньше,
         // но функция displayStatistics пропущена,
         // а функция getArea модифицирована:
   double getArea() const; // Площадь поперечного сечения
   . . .
}; // Конец класса
```

Допустим, что объект *mySphere* представляет собой экземпляр класса *Sphere*, а объект *myBall* — экземпляр класса *Ball*.

На первый взгляд, все благополучно: функция *mySphere.getArea()* возвращает площадь поверхности сферы, а функция *myBall.getArea()*— площадь поперечного сечения шара, как и ожидалось. Проблема возникает, если в классе *Ball* пропустить функцию *displayStatistics*. Такой пропуск вполне объясним, поскольку класс *Ball* наследует и вызывает функцию-член *displayStatistics* из класса *Sphere*. Напомним, что ее определение выглядит следующим образом.

```
void Sphere::displayStatistics() const
{
   cout << "\nРадиус = " << getRadius()
   << "\nДиаметр = " << getDiameter()
   << "\nДлина окружности = " << getCircumference()
   << "\nПлощадь = " << getArea()
   << "\nОбъем = " << getVolume() << endl;
} // Конец функции displayStatistics
```

Эта функция вызывает функцию *getArea*, однако оператор *myBall.displayStatistics()* выводит на экран площадь поверхности шара *myBall*, а не площадь его поперечного сечения. Иными словами, вызывается функция *getArea* из класса *Sphere*, как показано на рис. 8.8. Поскольку функция *getArea* не виртуальна, компилируя функцию *Sphere::displayStatistics*, компилятор выбирает ее версию из класса *Sphere*, а не из класса *Ball*.

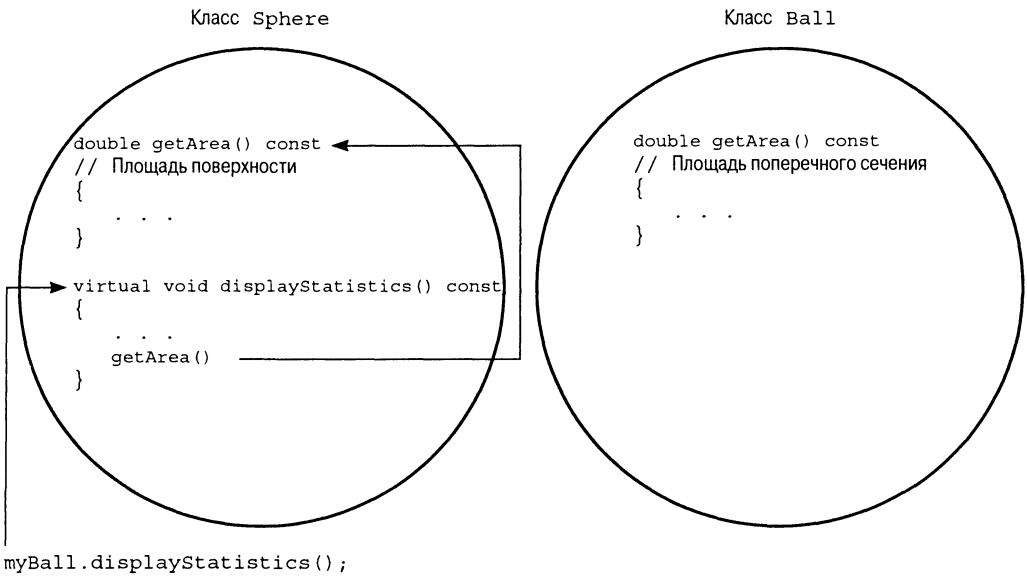


Рис. 8.8. Функция getArea не виртуальна: оператор myBall.displayStatistics( ) вызывает функцию Sphere::getArea( )

Раннее связывание снова порождает проблемы. Базовый класс *Sphere* должен объявлять функцию *getArea* виртуальной, чтобы класс *Ball* мог правильно заместить ее. (Функция *displayStatictics* при этом не обязана быть виртуальной, поскольку в данной ситуации это не влияет на результат.) Итак, объявления классов *Sphere* и *Ball* следует изменить.

```
class Sphere
{
public:
   . . .
   virtual double getArea() const; // Площадь поверхности
   virtual void displayStatistics() const;
   . . .
}; // Конец класса
```

```
class Ball: public Sphere
{
public:
   . . .
   // Ключевое слово указывать желательно, но не обязательно.
   // Функция displayStatistics пропущена
   . . .
}; // Конец класса
```

Теперь, когда экземпляр класса *Sphere* вызовет функцию *displayStatistics*, она, в свою очередь, вызовет функцию *Sphere::getArea()* (рис. 8.9, *а*), а когда функцию *displayStatistics* вызовет экземпляр класса *Sphere*, она вызовет функцию *Ball::getArea()* (рис. 8.9, *б*). Таким образом, назначение функции *displayStatistics* зависит от типа вызывающего объекта.

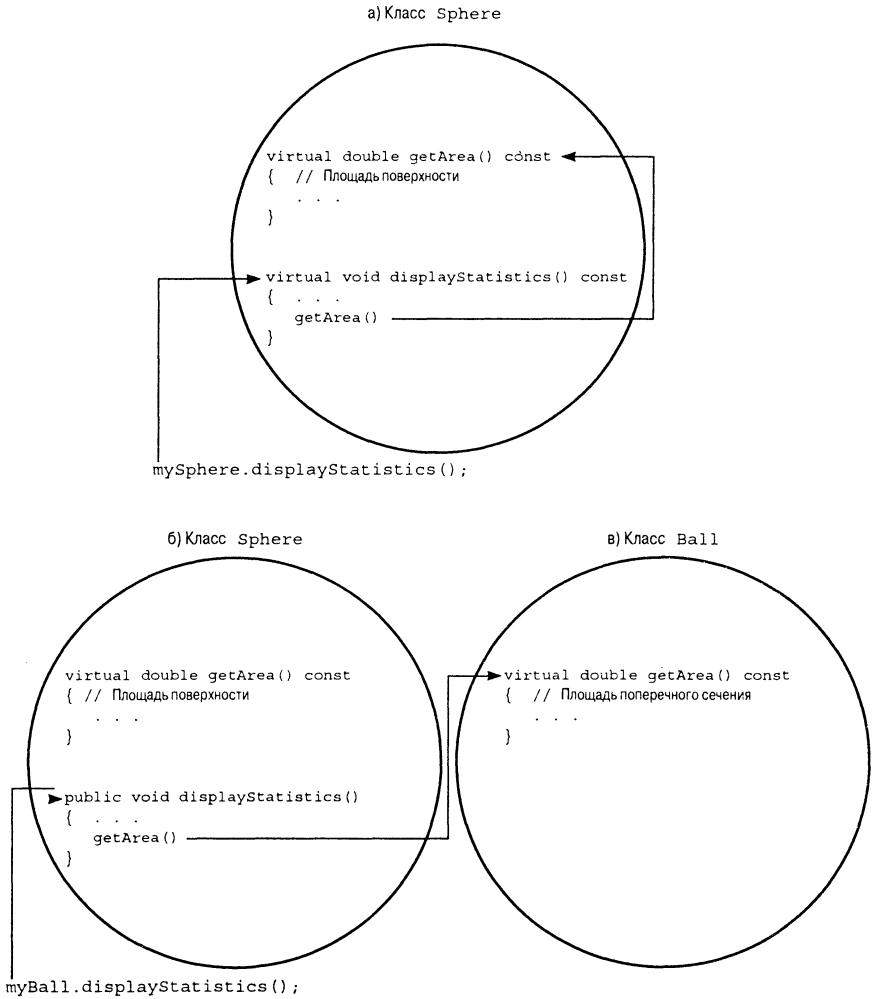


Рис. 8.9. Функция getArea виртуальна: а) оператор myBall.displayStatistics( ) вызывает функцию Sphere::getArea( ); б) оператор myBall.displayStatistics( ) вызывает функцию Ball::getArea( )

Использование виртуальных функций существенно влияет на будущее применение класса. Представьте себе, что мы скомпилировали класс `Sphere` и его реализацию, а потом написали производный класс `Ball`. Если при создании класса `Ball` подразумевался свободный доступ к членам класса `Sphere`, то функцию `getArea` можно было бы заместить, поскольку в классе `Sphere` она является виртуальной. В результате поведение функции `displayStatistics` по отношению к экземплярам класса `Ball` изменится, даже если класс `Sphere` уже скомпилирован. Таким образом, классы, определяющие виртуальные функции, являются **расширяемыми** (extensible): в них можно добавлять новые функциональные возможности независимо от доступа к исходному коду базовых классов.

Если класс содержит виртуальные функции, он является расширяемым

Как правило, классы должны использовать виртуальные функции. Однако, если вы не собираетесь замещать в производных классах конкретную функцию-член, ее не следует объявлять виртуальной. Например, функция `Sphere::getRadius` всегда возвращает значение переменной `theRadius`, поэтому ее не нужно замещать в производном классе.

Каждый класс, определяющий виртуальную функцию, содержит **таблицу виртуальных методов** (virtual method table — VMT), остающуюся невидимой для программиста. Для каждой виртуальной функции, объявленной в классе, таблица виртуальных методов содержит указатель на фактические инструкции, реализующие этот метод. Этот указатель задается при вызове конструктора. Иными словами, конструктор, создающий объект, устанавливает в таблице виртуальных методов указатели на версии виртуальных функций, соответствующих данному объекту. Таким образом, таблица виртуальных методов — это механизм реализации позднего связывания.

Подытожим сказанное.

ОСНОВНЫЕ ПОНЯТИЯ

Виртуальные функции

1. Виртуальной называется функция, замещаемая в производном классе.
2. Виртуальные функции, предусмотренные в классе, должны иметь реализацию. (К чисто виртуальным функциям, описанным в следующем разделе, это требование не относится.)
3. В производном классе не обязательно замещать существующую реализацию наследуемой виртуальной функции.
4. Как правило, функции-члены класса должны быть виртуальными. Если функции не будут замещаться в производных классах, их не обязательно объявлять виртуальными.
5. Конструкторы не могут быть виртуальными.
6. Деструкторы могут и должны быть виртуальными. Виртуальные деструкторы гарантируют, что будущие потомки объекта могут корректно освободить занимаемую ими память.
7. Тип значения, возвращаемого виртуальной функцией, не замещается.

Абстрактные базовые классы

Допустим, у нас есть видеомагнитофон и обычный магнитофон. Они обладают разными свойствами. Оба устройства работают с магнитными лентами, заправленными в кассеты. Эти кассеты можно вставлять, извлекать, воспроизводить, записывать, прокручивать вперед и назад, а также останавливать. Некоторые из этих операций совершенно одинаковы в обоих устройствах, а некоторые, в частности функции воспроизведения и записи, — незначительно отличаются друг от

друга. Таким образом, можно сказать, что видеомагнитофон — это обычный магнитофон, который может воспроизводить и записывать видеоизображения.

Если бы мы рассматривали классы ACR (audio-cassette recorder) и VCR (video-cassette recorder), то класс VCR был бы производным от класса ACR. Экземпляр класса VCR содержит все функции класса ACR, замещая функции воспроизведения и записи, как показано на рис. 8.10. В классе VCR предусмотрены дополнительные операции, например, установка времени записи.

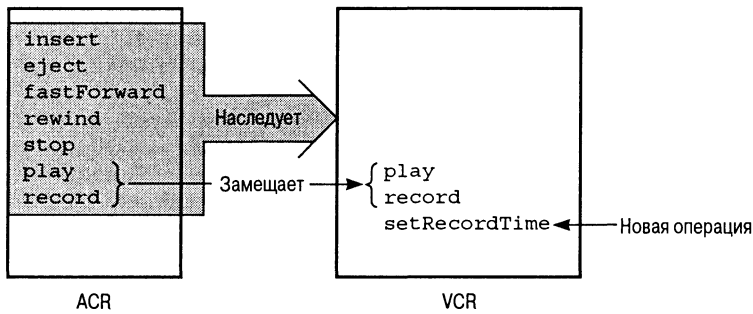


Рис. 8.10. Класс VCR является производным от класса ACR

Несмотря на то что операции *play* и *record* класса ACR отличаются от их аналогов в классе VCR, обе эти операции прокручивают ленту вперед на небольшой скорости. У классов SACR и VCR есть еще несколько общих операций. Это позволяет определить новый класс, пользуясь определениями классов ACR и VCR, — класс устройств для перемотки ленты.

Операции перемотки ленты

```
+insert()
// Вставить кассету в устройство

+eject()
// Извлечь кассету из устройства

+fastForward()
// Прокрутить кассету вперед на большой скорости

+rewind()
// Прокрутить кассету назад  на большой скорости

+goForward()
// Прокрутить кассету вперед на маленькой скорости

+stop()
// Остановить прокрутку
```

За исключением операции *goForward*, все остальные операции присущи также классам ACR и VCR. Несмотря на то что операции *goForward* в классах ACR МСК нет, их операции *play* и *record* могут ее использовать. Итак, можно констатировать следующие факты.

- Класс ACR — это устройство для перемотки ленты, позволяющее воспроизводить и записывать звуки.
- Класс VCR — это устройство для перемотки ленты, позволяющее воспроизводить и записывать видеоизображения.

В терминах классов это означает, что класс устройств для перемотки лент является базовым для класса ASR, который, в свою очередь, является базовым для класса VCR, как показано на рис. 8.11.

Нужны ли нам экземпляры класса устройств для перемотки ленты? Очевидно, нет. Хотя большинство его операций весьма полезны (помимо всего прочего, если пропустить операцию *goForward*, устройство позволяет прокручивать ленту в обратном направлении), операция *goForward* просто перематывает ленту, не выполняя ни записи, ни воспроизведения. Следовательно, класс устройств для перемотки ленты не имеет операций и является основой для создания других классов.

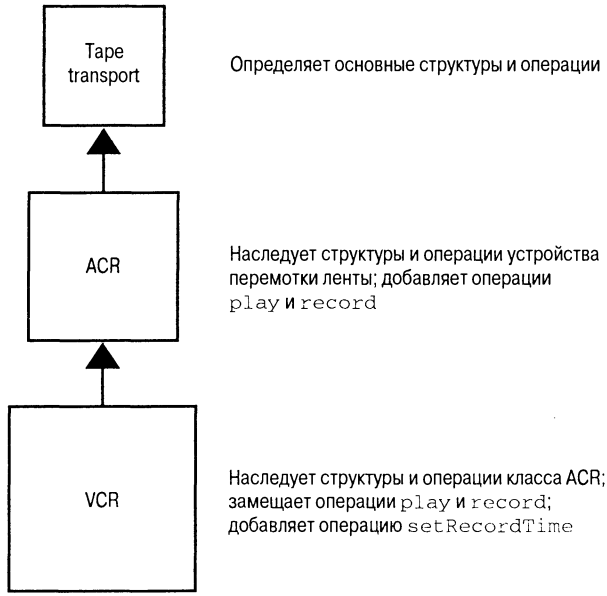


Рис. 8.11. Класс ACR имеет абстрактный базовый класс (устройство для перемотки ленты) и производный класс (VCR)

Если класс не имеет экземпляров, его функции-члены не реализуются. Такие функции, однако, должны быть виртуальными, чтобы в производных классах их можно было заместить. Виртуальная функция, не имеющая тела, называется **чисто виртуальной функцией** (pure virtual function) и объявляется в определении класса следующим образом.

```
virtual prototype = 0;
```

Чисто виртуальная функция не имеет тела

Реализация чисто виртуальной функции поручается производному классу.

Класс, содержащий хотя бы одну чисто виртуальную функцию, называется **абстрактным базовым классом** (abstract base class). Такой класс не может иметь экземпляров и предназначен только для создания других классов. Таким образом, класс устройств для перемотки ленты является абстрактным базовым классом. Любой производный класс, в котором не реализованы все *чисто* виртуальные функции, также является абстрактным базовым классом и не может иметь экземпляров.

Класс, содержащий хотя бы одну чисто виртуальную функцию, называется абстрактным базовым классом

Другой пример. Класс *Sphere* и *Ball* описывают множество точек в трехмерном пространстве, равноудаленных от начала координат. Класс *EquidistantShape*, определенный ниже, объявляет операции, позволяющие задавать и определять расстояние точки от начала координат. Этот класс можно считать абстрактным базовым классом по отношению к классу *Sphere*.

Абстрактный базовый класс по отношению к классу Sphere

```
class EquidistantShape
// Абстрактный базовый класс
{
public:
   virtual void setRadius(double newRadius) = 0;
   virtual double getRadius() const = 0;
   virtual void displayStatistics() const = 0;
}; // Конец класса
```

В этом классе объявлены чисто виртуальные функции *setRadius*, *getRadius* и *displayStatistics*.

Определим класс *Sphere*, используя *EquidistantShape* в качестве базового класса.

```
class Sphere: public EquidistantShape
{
public:
   Sphere();
   Sphere(double initialRadius);
   Sphere(const Sphere& aSphere);
   virtual ~Sphere();

   virtual void setRadius(double newRadius);
   virtual double getRadius() const;
   virtual double getDiameter() const;
   virtual double getCircumference() const;
   virtual double getArea() const;
   virtual double getVolume() const;
   virtual void displayStatistics() const;

private:
   double theRadius;
}; // Конец класса
```

Включая в абстрактный базовый класс чисто виртуальные функции *setRadius*, *getRadius* и *displayStatistics*, мы поручаем их реализацию производному классу, в данном случае — классу *Sphere*. Теперь класс *Ball* может быть наследником класса *Sphere*, как и раньше.

Переменную *theRadius* можно объявить закрытым членом класса *EquidistantShape*, а не класса *Sphere*. Тогда производный класс не сможет обращаться к этой переменной по имени. Таким образом, класс *EquidistantShape* должен содержать методы *setRadius* и *getRadius*, реализации которых по умолчанию наследуются производным классом. Иными словами, методы *setRadius* и *getRadius* не могут быть чисто виртуальными. Более того, они не должны быть и просто виртуальными, поскольку производный класс не может их замещать. Итак, определение класса *EquidistantShape* необходимо изменить.

```
class EquidistantShape // Абстрактный
базовый класс
{
public:
   void setRadius(double newRadius);
   double getRadius() const;
   virtual void displayStatistics() const = 0;

private:
   double theRadius;
}; // Конец класса
```

Другой абстрактный базовый класс для класса Sphere; функции-члены setRadius и getRadius не являются чисто виртуальными, поэтому их необходимо реализовать

В качестве альтернативы, переменную *theRadius* можно определить как защищенный член класса *EquidistantShape*, дав производному классу возможность как прямого доступа к ней, так и косвенного — через функции *setRadius* и *getRadius*. И все же, как правило, данные-члены должны быть закрытыми.

Все объекты должны иметь виртуальные деструкторы

В абстрактном базовом классе можно предусмотреть конструктор. Поскольку конструкторы не бывают виртуальными, они не могут быть и чисто виртуальными. В то же время деструктор абстрактного базового класса не должен быть чисто виртуальным, поскольку он вызывается производными классами и, следовательно, должен иметь реализацию. Как правило, рекомендуется использовать виртуальные деструкторы.

Подытожим сказанное об абстрактных базовых классах.

ОСНОВНЫЕ ПОНЯТИЯ

Абстрактный базовый класс

1. По определению, абстрактный базовый класс — это класс, содержащий хотя бы одну чисто виртуальную функцию.
2. Абстрактный базовый класс используется только в качестве основы для производных классов и поэтому содержит минимум интерфейса.
3. Абстрактный базовый класс не имеет экземпляров.
4. Как правило, все реализации функций-членов абстрактного класса следует пропускать, за исключением деструктора и функций, обеспечивающих доступ к закрытым данным-членам. Другими словами, виртуальные функции в абстрактном базовом классе должны быть чисто виртуальными.
5. Все виртуальные функции-члены абстрактного базового класса, не являющиеся чисто виртуальными, должны быть реализованы так, чтобы производные классы могли наследовать их по умолчанию.

Дружественные функции и классы

Функции и классы могут быть друзьями класса

В языке C++ существует еще один механизм доступа к закрытым и защищенным членам класса. Этот доступ обеспечивается **дружественными функциями и классами** (friends). Если функция, не являющаяся членом класса, объявляется его другом, она получает доступ ко все закрытым и защищенным членам этого класса. Например, дружественными часто объявляются функции ввода и вывода.

```
class Sphere
{
public:
   // Конструкторы и операции остаются без изменений
   . . .
   friend void readSphereData(Sphere& s);
   friend void writeSphereData(Sphere& s);

private:
   double theRadius; // радиус сферы
}; // Конец класса
```

Функции *readSphereData* и *writeSphereData* не являются членами класса *Sphere*. Они определяются вне класса *Sphere*. Например, если данные, содержащиеся в объекте класса *Sphere*, можно считать с помощью стандартных средств ввода и вывести в стандартный поток вывода, эти функции принимают следующий вид.

Функции, дружественные по отношению к классу Sphere

```
void readSphereData(Sphere& s)
{
   cout << "Введите радиус сферы: ";
   cin >> s.theRadius;
} // Конец функции readSphereData

void writeSphereData(Sphere& s)
{
   cout << "Радиус сферы равен " << s.theRadius << endl;
} // Конец функции writeSphereData
```

Дружественные функции не являются членами класса Sphere

Обратите внимание, что эти функции имеют доступ к закрытому члену *theRadius* из класса *Sphere*. На первый взгляд, это нарушает принцип сокрытия информации, но ситуация остается под контролем. Класс *Sphere* должен явно предоставить доступ к своим закрытым членам, объявив эти функции дружественными.

Функции *readSphereData* и *writeSphereData* могут быть не друзьями, а членами класса *Sphere*. Однако ввод и вывод данных с помощью функций, определенных вне класса, позволяет достичь большей гибкости. Зная контекст, в котором используется класс, можно точно настраивать функции ввода и вывода. Кроме того, при изменении способа ввода и вывода не нужно компилировать класс повторно.

Объявляя некий класс другом класса *C*, мы открываем всем его функциям-членам доступ к закрытым и защищенным членам класса *C*. Это особенно полезно, когда один класс использует реализацию другого. Например, реализация абстрактного типа данных в виде связанного списка предполагает, что в каждом узле содержится закрытый раздел. Узел можно реализовать в виде отдельного класса, объявив класс, реализующий абстрактный тип данных, его другом. В таком случае можно дать следующее определение узла абстрактного списка.

Дружественный класс — это класс, являющийся другом другого класса

```
#include <cstddef>
typedef тип-элемента-списка ListItemType;

class ListNode // Узел списка
{
```

```
private:
   ListNode(): next(NULL) {};
   ListNode(const ListItemType& nodeItem, ListNode *ptr)
            :item(nodeItem), next(ptr) {};
   ListItemType item; // Данные, содержащиеся в списке
   ListNode *next; // Указатель на следующий узел

   // Дружественный класс - имеет доступ к закрытым разделам
   friend class List;
}; // Конец класса
```

Класс List является другом класса ListNode

Класс *List* имеет привилегии доступа к членам *item* и *next*, которые принадлежат узлам, как если бы класс *ListNode* был простой структурой. Теперь конструктор класса *List* может инициализировать узлы списка (его можно предусмотреть и в самой структуре).

Друзья базового класса не являются друзьями производного класса

Друзья базового класса не имеют доступа к закрытым и защищенным членам, добавляемым производным классом. Доступ открыт только к закрытым и защищенным членам базового класса. В качестве примера рассмотрим следующие определения классов *Sphere* и *Ball*.

```
class Sphere
{
public:
// Конструкторы и операции остаются без изменения
   . . .
   friend void readSphereData(Sphere& s);
   friend void writeSphereData(Sphere& s);

private:
   double theRadius; // Радиус сферы
}; // Конец класса

class Ball: public Sphere
{
public:
   Ball();
   Ball(double initialRadius, const string initialName);
   void getName(string currentName) const;
   void setName(const string newName);
   void resetBall(double newRadius, const string newName);
   void displayStatistics() const;

private:
   string theName; // Имя шара
}; // Конец класса
```

Поскольку дружественные функции *readSphereData* и *writeSphereData* имеют формальный аргумент типа *Sphere*, им можно передавать объекты класса *Ball*. Однако эти функции получат доступ только к члену *theRadius*, но не к члену *theName*. Подытожим сказанное.

ОСНОВНЫЕ ПОНЯТИЯ

Друзья

1. Дружественные функции имеют доступ к закрытым и защищенным разделам класса.
2. Дружественные функции не являются членами класса.
3. Когда некий класс объявляется другом класса C, все его функции-члены получают доступ к закрытым и защищенным разделам класса C.
4. Дружба не наследуется. Закрытые и производные члены, объявленные в производном классе, не доступны друзьям базового класса.

Новая реализация абстрактного и упорядоченного списка

Абстрактный список и абстрактный упорядоченный список были рассмотрены нами в главе 3. У этих списков есть общие свойства и операции. Например, каждый из них имеет длину и операцию, позволяющую проверять, пуст ли список. Эти общие свойства и операции можно вынести в абстрактный базовый класс и впоследствии создавать на его основе любые списки.

Абстрактный базовый класс для списков

```
class BasicADT // Абстрактный базовый
класс
{
public:
   virtual ~BasicADT(); // Деструктор
   virtual bool isEmpty() const = 0;
   virtual int getLength() const = 0;
}; // Конец класса
```

Поскольку деструктор не является чисто виртуальной функцией, его необходимо реализовать. Остальные функции-члены являются виртуальными, поэтому они должны быть реализованы в каком-нибудь из производных классов. Иными словами, поместив функции-члены *isEmpty* и *getLength* в описание абстрактного базового класса, разработчик предусматривает их дальнейшую реализацию во всех производных классах.

Обратите внимание, что класс *BasicADT* можно было бы использовать в качестве основы для списка, стека и очереди. Все эти абстрактные типы данных предусматривают функцию *isEmpty*. Кроме того, в них легко реализовать и функцию *getLength*. Использование базового класса *BasicADT* гарантирует, что все производные классы имеют, по крайней мере, функции-члены *isEmpty* и *getLength*.

Приведенный ниже класс, реализующий абстрактный список, открыто наследует свойства базового класса. Предполагается, что элементы списка имеют тип *ListItemType*. Пред- и постусловия методов класса сформулированы в главе 3.

```
// ********************************************************
// Заголовочный файл List.h абстрактного списка.
// Используются классы ListNode и BasicADT.
// ********************************************************
#include "ListNode.h" // Описание класса ListItemType
#include "ListException.h"
#include "ListIndexOutOfRangeException.h"
#include "BasicADT.h"
```

```
class List : public BasicADT
{
public:
// Конструкторы и деструктор:
   List();
   List(const List& aList);
   virtual ~List();

// Операции над списком:
   virtual bool isEmpty() const;
   virtual int getLength() const;
   virtual void insert(int index, ListItemType newItem)
   throw(ListIndexOutOfRangeException, ListException);
   virtual void remove(int index)
         throw(ListIndexOutOfRangeException);
   virtual void retrieve(int index,
                         ListItemType& dataItem) const
         throw(ListIndexOutOfRangeException);

protected:
   void setSize(int newSize); // Устанавливает размер
   ListNode *getHead() const; // Возвращает указатель на голову
   void setHead(ListNode *newHead); // Задает указатель
                                    // на голову

   // Следующие две функции возвращают элемент списка
   // и указатель на следующий узел связанного списка
   ListItemType getNodeItem(ListNode *ptr) const;
   ListNode *getNextNode(ListNode *ptr) const;

private:
   int size; // Количество элементов в списке
   ListNode *head; // Указатель на связанный список

   ListNode *find(int index) const;
}; // Конец класса
```

Необходимо реализовать хотя бы функции *isEmpty* и *getLength*, которые класс *List* открыто наследует от класса *BasicADT*, поскольку в абстрактном базовом классе эти методы являются чисто виртуальными. В противном случае класс *List* сам станет абстрактным базовым классом.

Защищенные функции-члены открывают производному классу доступ к закрытым данным-членам

Поскольку в классе *List* данные-члены *size* и *head* являются закрытыми, доступ к ним имеют лишь члены этого класса. Определяя защищенные функции *setSize*, *getHead* и *setHead*, мы предоставляем наследникам класса *List* косвенный доступ к этим данным-членам. (Заметьте, что защищенная функция-член не обязана возвращать значение переменной *size*, поскольку для этого предназначен открытый метод *getLength*.) Доступ к узлам связанного списка обеспечивается защищенными функциями-членами *getNodeItem* и *getNextNode*. Эти функции используются в следующем разделе.

Функция *find* является закрытой, поэтому производный класс не может использовать ее. Однако, объявив эту функцию защищенной, можно сделать ее доступной.

Реализация абстрактного списка в виде связанного рассмотрена в главе 4.

Реализации абстрактного упорядоченного списка на основе абстрактного списка

Допустим, что нам необходимо определить и реализовать абстрактный упорядоченный список, для которого предусмотрены следующие операции.

Операции над абстрактным упорядоченным списком

```
+createSortedList()
+destroySortedList()
+sortedIsEmpty():boolean {query}
+sortedLength():integer {query}
+sortedInsert(in newItem:ListItemType)
              throw ListException
+sortedRemove(in anItem:ListItemType)
              throw ListException
+sortedRetrieve(in index:integer,
                out dataItem:ListItemType) {query}
                throw ListIndexOutOfRangeException
+locatePosition(in anItem:ListItemType,
out isPresent:boolean):integer {query}
```

Разумеется, для реализации упорядоченного списка можно применить связанный список или массив, однако это вынудит нас практически полностью скопировать функции-члены класса *List*.

Класс List можно повторно использовать для создания класса SortedList

К счастью, этого повторения можно избежать, применив для реализации класса *SortedList* определенный ранее класс *List*. Между этими классами существуют три типа отношений: "является", "содержит" и "подобен". Однако, как правило, все три типа отношений одновременно не применяются. В конкретных ситуациях наилучшим оказывается лишь один из них, причем отношение подобия часто оказывается наименее приемлемым. Несмотря на это мы продемонстрируем все три варианта.

Упорядоченный список *является* списком. В главе 3 показано, что абстрактный список — это просто перечень пронумерованных элементов. Можно ли создать упорядоченный список, установив определенный порядок среди элементов абстрактного списка? Большинство операций абстрактного списка и абстрактного упорядоченного списка одинаково, хотя и называются по-разному. Однако операции вставки и удаления элементов отличаются. Кроме того, для абстрактного упорядоченного списка предусмотрена дополнительная операция *locatePosition*.

Для того чтобы вставить в упорядоченный список новый элемент, сначала нужно определить подходящую позицию, используя операцию *locatePosition*, а затем применить операцию вставки из класса *List*. Операция удаления элемента из упорядоченного списка выполняется аналогично.

Отношение "является" предполагает открытое наследование

Итак, упорядоченный список *является* списком, поэтому можно применить открытое наследование. Иными словами, класс *SortedList* может быть производным от класса *List*, наследуя его члены, добавляя новую функцию-член *locatePosition* и модифицируя операции вставки и удаления элементов.

```
#include "List.h"
class SortedList: public List
{
public:
// Конструкторы и деструктор:
```

```
    SortedList();
    SortedList(const SortedList& sList);
    virtual ~SortedList();

// Новые операции:
    virtual void sortedInsert(ListItemType newItem)
          throw(ListException);
    virtual void sortedRemove(ListItemType anItem)
          throw(ListException);
    virtual int locatePosition(ListItemType anItem,
                               bool& isPresent);
}; // Конец класса SortedList
```

Реализации функций-членов класса *SortedList* имеют следующий вид.

```
SortedList::SortedList()
{
} // Конец конструктора по умолчанию

SortedList::SortedList(const SortedList& sList):
                       List(sList)
{
} // Конец конструктора копирования

SortedList::~SortedList()
{
} // Конец деструктора

void SortedList::sortedInsert(ListItemType newItem)
{
   bool found;
   int newPosition = locatePosition(newItem, found);
   insert(newPosition, newItem);
} // Конец функции sortedInsert

void SortedList::sortedRemove(ListItemType anItem)
{
   bool found;
   int position = locatePosition(anItem, found);
   if (found) // Элемент найден
       remove(position);
   else
       throw ListException(
       "ListException: удаляемый элемент не найден");
} // Конец функции sortedRemove

int SortedList::locatePosition(ListItemType anItem,
                               bool& isPresent)
{
   ListNode *trav = getHead();
   int position = 1;
   while ((trav != NULL) && (getNodeItem(trav) < anItem))
   {
      trav = getNextNode(trav);
      position++;
   } // Конец оператора while
   if ((trav != NULL) && (anItem == getNodeItem(trav)))
```

```
      isPresent = true;
   else
      isPresent = false;
   return position;
} // Конец функции locatePosition
```

Для доступа к структуре данных, лежащей в основе списка, следует применять защищенные функции-члены класса List

Обратите внимание на защищенные функции-члены *getHead*, *getNodeItem* и *getNextNode*.

В классе *SortedList* есть функции, унаследованные им от класса *List*, — к ним относятся функции *isEmpty*, *getLength* и *retrieve*, — а также новые функции *sortedInsert* и *sortedRemove*. Такие имена функций сбивают с толку. Исправить ситуацию можно двумя способами.

1. В класс *SortedList* можно добавить функцию-член *sortedGetLength*.

```
int SortedList::sortedGetLength()
{
    return getLength();
}   // Конец функции sortedGetLength
```

2. Этот подход позволяет перечислить операции над упорядоченным списком, причем вставку элемента можно производить как по значению (с помощью функции *sortedInsert*), так и по позиции (с помощью унаследованной функции *insert*). В зависимости от конкретного приложения эта гибкость иногда оказывается полезной, а иногда — вредной.

Этот подход позволяет перечислить операции над упорядоченным списком

3. Операции вставки и удаления элементов упорядоченного списка можно назвать именами *insert* и *remove*. Для этого достаточно предусмотреть новую реализацию для каждой из них.

Лучше всего иметь только одно множество имен

```
void SortedList::insert(ListItemType newItem)
{
   bool found;
   int newPosition = locatePosition(newItem, found);
   List::insert(newPosition, newItem);
} // end insert
```

4. Поскольку операции вставки в классах *List* и *SortedList* названы одинаково, при их вызове необходимо указывать имя класса, например *List::insert*. Это соответствует духу полиморфизма, поскольку если операции вставки и удаления в классе *List* являются виртуальными, они будут замещены. Итак, для упорядоченного списка предусмотрена только одна операция вставки и одна операция удаления элемента. Для того чтобы заместить эти функции в производных классах, их заместители должны иметь те же самые имена.

Упорядоченный список *содержит* список. Если между новым классом и его предшественником нет отношения "является", то открытое наследование становится неприемлемым. Однако существующий класс все же можно использовать для реализации нового класса. В приведенном ниже объявлении класса *SortedList* *содержится* закрытый член, представляющий собой экземпляр класса *List*, содержащий элементы упорядоченного списка.

```
class SortedList
{
public:
// Конструкторы и деструктор:
   SortedList();
   SortedList(const SortedList& sList);
   virtual ~SortedList();

// Операции над упорядоченным списком:
   virtual bool sortedIsEmpty() const;
   virtual int sortedGetLength() const;
   virtual void sortedInsert(ListItemType newItem)
         throw(ListException);
   virtual void sortedRemove(ListItemType anItem)
         throw(ListException);
   virtual void sortedRetrieve(int index,
         ListItemType anItem) const
         throw(ListIndexOutOfRangeException);
   virtual int locatePosition(ListItemType anItem,
         bool& isPresent);

private:
   List aList;
}; // Конец класса
```

Экземпляр класса List может быть реализацией связанного списка

Необходимо реализовать конструктор, конструктор копирования, деструктор и каждый метод данного класса. Например, операция вставки реализуется следующим образом.

```
void SortedList::sortedInsert(ListItemType newItem)
{
   bool found;
   int newPosition = locatePosition(newItem, found);
   aList.insert(newPosition, newItem);
} // Конец функции sortedInsert
```

Член *aList* класса *SortedList* является экземпляром класса *List*. Обозначение *aList.insert* обозначает вызов операции вставки из класса *List*.

Полную реализацию этого класса читатели могут осуществить сами, выполнив задание 3, приведенное в конце главы. В процессе решения этой задачи станет ясно, что для доступа к элементам списка *aList* функция *locatePosition* нуждается в помощи функции *retrieve*. Иными словами, реализация класса *List* скрыта от класса *SortedList*. Обратите внимание, что клиент класса *SortedList* не имеет доступа к объекту *aList* и может выполнять только операции над упорядоченным списком.

Упорядоченный список "подобен" списку. И вновь, если между новым и существующим классами нет отношения "является", открытое наследование применять не следует. Однако, если нам нужно унаследовать члены существующего класса, можно применить закрытое наследование.

```
class SortedList: private List
{
public:
   . . . совпадает с открытым разделом класса,
                  соответствующего отношению "содержит"
}; // Конец класса
```

Открытый раздел этого класса совпадает с открытым разделом класса, соответствующего отношению "*содержит*". Поскольку клиенты класса *SortedList* не имеют доступа в функциям-членам базового класса *List*, необходимо обеспечить полный набор операций над упорядоченным списком. При открытом наследовании делать это было необязательно.

В реализации класса *SortedList* можно использовать открытые и защищенные члены класса *List*. Фактически реализации этих функций-членов совпадают, как и при открытом наследовании.

И в этом примере, и в примере, связанном с отношением "содержит", объект класса *List* скрыт от клиентов класса *SortedList*. Таким образом, для сокрытия реализации упорядоченного списка можно применять либо закрытое наследование, либо отношение включения. Однако следует иметь в виду, что, в отличие от открытого наследования, закрытое не позволяет использовать экземпляр класса *SortedList* вместо экземпляра класса *List*, т.е. классы *SortedList* и *List* не совместимы по типу.

В дальнейшем мы не будем рассматривать закрытое наследование.

Шаблонные классы

До сих пор, рассматривая абстрактные типы данных, мы манипулировали элементами, имеющими конкретный тип. Например, тип элементов списка, реализованного в виде класса *List*, зависел от определения класса *ListItemType*.

```
typedef double ListItemType; // Тип элемента списка
```

Допустим, однако, что в нашем алгоритме нужны два списка: список действительных чисел и список символов. Экземпляр класса *List*, в котором тип *ListItemType* определен как тип *double*, может содержать только действительные числа. Для того чтобы создать экземпляр списка, содержащего символы, придется разрабатывать другой класс.

Тип данных в шаблонном классе задается с помощью параметров

Многочисленных определений классов можно избежать, используя **шаблонный класс** (class template), в котором тип данных задается с помощью параметров. Когда клиент объявляет экземпляр шаблонного класса, он указывает фактический тип данных. Шаблонные классы используются во всех контейнерных классах библиотеки STL.

Рассмотрим пример определения шаблонного класса, в котором класс *T* играет роль **шаблонного параметра** (data-type parameter).

```
template <class T>
class NewClass
{
public:
   NewClass();
   NewClass(T initialData);

   void setData(T newData);
   T getData();

private:
   T theData;
}; // Конец класса
```

Перед определением класса стоит выражение *template <class T>*, в котором параметр *T* задается клиентом. В этом классе содержится одна переменная, две функции и два конструктора.

Программа, использующая данный шаблонный класс, может начинаться так.

Фактический тип данных задается клиентом при объявлении экземпляра шаблонного класса

```
int main()
{
   NewClass<int> first;
   NewClass<double> second(4.8);

   first.setData(5);
   cout << second.getData() << endl;
```

Обратите внимание, что объявления объектов *first* и *second* задают тип, который внутри шаблонного класса представлен параметром *T*.

Рассмотрим реализацию этого класса.

```
template <class T>
NewClass<T>::NewClass()
{
} // Конец конструктора по умолчанию

template <class T>
NewClass<T>::NewClass(T initialData)
                      :theData(initialData)
{
} // Конец конструктора

template <class T>
void NewClass<T>::setData(T newData)
{
   theData = newData;
} // Конец функции setData

template <class T>
T NewClass<T>::getData()
{
   return theData;
} // Конец функции getData
```

Перед определением каждой функции вновь стоит выражение *template <class T>*, а имя типа объекта *NameClass* обязательно сопровождается суффиксом *<T>*.

Выполняя операции над объектами типа *T* внутри шаблонного класса, следует быть осторожным. Допустим, что в класс *NewClass* добавлена новая функция *display*.

```
void display();
```

Ее реализация выглядит так.

```
template <class T>
void NewClass<T>::display()
{
   cout << theData;
} // Конец функции display
```

Предполагается, что в функции *display* для объектов типа *T* определен оператор вывода в поток <<. Для стандартных типов, таких как *int*, *char* и даже *string*, это происходит по умолчанию, но для типов, определенных пользователем, это условие может не выполняться. В следующем разделе мы покажем, как определять операторы для таких объектов. В шаблонном классе необходимо подробно описать все операции над объектами типа *T*.

Реализацию шаблонного класса не следует компилировать отдельно от клиентской программы

Описание шаблонного класса и его реализацию следует хранить в двух разных файлах, как это предусмотрено для обычных классов. Однако их использование немного отличается. Большинство компиляторов должно знать, как клиент использует шаблонный класс перед компиляцией файла, содержащего его реализацию. Иными словами, фактический тип данных, заданный параметром *T*, должен быть известен компилятору заранее. Следовательно, реализацию шаблонного класса не следует компилировать отдельно от клиентской программы, в которой он используется. Вместо этого файл реализации шаблонного класса нужно вставить в конец заголовочного файла, а затем включить заголовочный файл в клиентскую программу.

Ниже показана реализация абстрактного списка в виде связанного списка с помощью шаблонного класса. Отличия между шаблонной версией и вариантами, описанными ранее, выделены серым цветом. В качестве параметра вместо типа *ListItem* используется более короткое обозначение *T*. Чтобы избежать ненужных осложнений, абстрактный базовый класс не применяется.

Шаблонная версия заголовочного файла для узла списка

```
//
************************************
*****************
// Заголовочный файл ListNodeT.h абстрактного списка.
// Реализация в виде связанного списка -- ШАБЛОННАЯ ВЕРСИЯ
// ********************************************************
#include <cstddef>

template <class T> class List;

template <class T>
class ListNode
{
private:
   ListNode(): next(NULL) {};
   ListNode(const T & nodeItem, ListNode *ptr)
            : item(nodeItem), next(ptr) {};
   T item; // Элемент списка
   ListNode *next; // Указатель на следующий узел

   // Дружественный класс - имеет доступ к закрытым разделам
   friend class List <T>;
}; // Конец класса
```

Шаблонная версия заголовочного файла для абстрактного списка

```
//
************************************
*****************
// Заголовочный файл ListT.h абстрактного списка.
// Реализация в виде связанного списка -- ШАБЛОННАЯ ВЕРСИЯ
// ********************************************************
#include "ListNodeT.h"
#include "ListException.h"
```

```
#include "ListIndexOutOfRangeException.h"
template <class T>
class List
{
public:
// Конструкторы и деструктор:
   List();
   List(const List<T> & aList);
   virtual ~List();
   // Список операций над списком:
   virtual bool isEmpty() const;
   virtual int getLength() const;
   virtual void insert(int index, T newItem)
         throw(ListIndexOutOfRangeException, ListException);
   virtual void remove(int index)
         throw(ListIndexOutOfRangeException);
   virtual void retrieve(int index, T & dataItem) const
         throw(ListIndexOutOfRangeException);
protected:
   void setSize(int newSize);
   ListNode<T> *getHead() const;
   void setHead(ListNode<T> *newHead);
   T getNodeItem(ListNode<T> *ptr) const;
   ListNode<T> *getNextNode(ListNode<T> *ptr) const;

private:
   int size;
   ListNode<T> *head;

   ListNode<T> *find(int position) const;
}; // Конец класса
#include "ListT.cpp"
```

Сравнивая шаблонный класс с предыдущими объявлениями класса *List*, можно обнаружить небольшие отличия. Например, перед объявлениями

```
class ListNode
```

и

```
class List
```

в заголовочных файлах обязательно указываются ключевые слова

```
template <class T>
```

При упоминании типов *List* и *ListNode* после них обязательно указывается суффикс *<T>*

Ниже приводятся фрагменты файла реализации, иллюстрирующие отличия между шаблонной реализацией списка и реализацией, описанной в главе 4. Отличия выделены серым цветом.

```
// ********************************************************
// Фрагменты файла реализации ListT.cpp
// ********************************************************
#include <cstddef> // Определение константы NULL
template <class T>
List<T>::List(): size(0), head(NULL)
```

```
{
} // Конец конструктора по умолчанию

template <class T>
void List<T>::insert(int index, T newItem)
{
   int newLength = getLength() + 1;

  if ((index < 1) || (index > newLength))
      throw ListIndexOutOfRangeException(
      "ListOutOfRangeException: insert — неверный индекс");
  else
  {
      // Создать новый узел и поместить в него элемент newItem
      ListNode<T> *newPtr = new ListNode<T>;
      if (newPtr == NULL)
          throw ListException(
         "ListException: insert — невозможно выделить память ");
      else
      {
         size = newLength;
         newPtr->item = newItem;

         // Добавить в список новый узел
         if (index == 1)
         {
            // Добавить новый узел в начало списка
            newPtr->next = head;
            head = newPtr;
         }
         else
         {
            ListNode<T> *prev = find(index-1);
            // Вставить новый узел после узла,
            // на который ссылается указатель prev
            newPtr->next = prev->next;
            prev->next = newPtr;
         } // Конец оператора if
      } // Конец оператора if
   } // Конец оператора if
} // Конец функции insert

template <class T>
ListNode<T> *List<T>::find(int index) const
{
   if ( (index < 1) || (index > getLength()) )
      return NULL;
   else // Считая от начала списка
   {
      ListNode<T> *cur = head;
      for (int skip = 1; skip < index; ++skip)
          cur = cur->next;
      return cur;
   } // Конец оператора if
} // Конец функции find
```

В файле реализации, описанном в главе 4, метод *insert* из класса *List* начинался следующим образом.

```
void List::insert(int index, ListItemType newType);
```

В шаблонном варианте его начало выглядит иначе.

```
template <class T>
void List<T>::insert(int index, T newItem)
```

Перед именем класса *List* указывается выражение *template <class T>* и тип *void*, за ним — суффикс *<T>*, а тип *ListItemType* заменяется параметром *T*.

Программа, использующая шаблонный класс, описанный выше, может начинаться следующим образом.

```
#include "ListT.h"
int main()
{
   List<double> floatList;
   List<char> charList;

   floatList.insert(1, 1.1);
   floatList.insert(2, 2.2);

   charList.insert(1, 'a');
   charList.insert(2, 'b');

   . . .
```

Как и в первом случае, в объявлениях объектов *floatList* и *charList* указывается тип элементов списка.

Преобразование обычных классов в шаблонные не представляет труда

Обычно шаблонные классы производят сильное впечатление, особенно поначалу, однако превращение обычного класса в шаблонный представляет собой рутинную процедуру. При этом следует иметь в виду, что разработка самого класса может оказаться довольно сложной. Хотя в предыдущих примерах использовались простые типы данных, клиент может объявить экземпляры собственных классов. Следовательно, разработчик должен гарантировать, что шаблонный класс будет правильно работать при любых типах данных.

В заключение отметим, что шаблонный класс может иметь несколько параметров, например:

```
template <class T1, class T2>
```

Перечислим основные сведения, касающиеся шаблонных классов.

ОСНОВНЫЕ ПОНЯТИЯ

Шаблонные классы

1. Тип данных, используемых в шаблонном классе, задается его параметрами.
2. Фактические типы данных задаются при объявлении экземпляра шаблонного класса.
3. Все операции над фактическими типами данных в шаблонном классе должны быть подробно описаны.
4. Шаблонные классы могут иметь несколько параметров.

Перегруженные операторы

Стандартные арифметические операторы языка C++ имеют несколько значений. Хотя результаты выражений *2 + 3* и *2.0 + 3.0* кажутся одинаковыми, на самом деле это не так. Поскольку внутреннее представление целых чисел, например, чисел 2 и 3, отличается от представления чисел с плавающей точкой, таких как 2.0 и 3.0, алгоритм сложения двух целых чисел должен отличаться от алгоритма сложения двух чисел с плавающей точкой. В языке C++ можно было бы предусмотреть разные обозначения этих операторов, однако это излишне. Фактическое значение оператора + определяется типами его операндов. Операторы, имеющие несколько значений, называются **перегруженными** (overloaded) и представляют собой простейший вид полиморфизма.

Перегруженный оператор может иметь несколько значений

Типы в языке C++ определяются в виде классов. Клиенты таких классов должны работать с ними как со стандартными типами. В частности, клиент должен применять к ним операторы, предусмотренные в языке C++, и получать осмысленные результаты. Для того чтобы конкретный оператор можно было применить к экземплярам какого-либо класса, необходимо определить его новый смысл, т.е. перегрузить его.

Допустим, что объекты *myList* и *yourList* являются экземплярами класса *List*, и мы написали следующий фрагмент программы.

```
if (myList == yourList)
   cout << "Списки одинаковы.\n");
```

В классе *List* необходимо определить оператор ==, поскольку компилятор не может интерпретировать его, если операнды имеют нестандартный тип. Заметим, что списки *myList* и *yourList* являются эквивалентными, если выполняются следующие условия.

Два списка эквивалентны, если они имеют одинаковую длину и содержат одинаковые элементы

- Списки *myList* и *yourList* имеют одинаковую длину.
- Списки *myList* и *yourList* содержат одинаковые элементы.

Для того чтобы перегрузить оператор, нужно определить **операторную функцию** (operator function), имя которой имеет следующую форму

```
operator символ
```

Здесь *символ* означает перегружаемый оператор. Операторная функция, соответствующая оператору ==, носит имя *operator==* и имеет один аргумент — объект, стоящий в правой части оператора. В определение класса *List* следует добавить объявление[2]

```
virtual bool operator==(const List& rhs) const;
```

Для того чтобы разобраться в этих обозначениях, рассмотрим функцию

```
virtual bool isEqual(const List& rhs) const;
```

Чтобы сравнить между собой объекты *myList* и *yourList*, нужно написать следующий фрагмент программы.

---

[2] Полагая, что функция *operator==* определена для типа *T*, в шаблонный класс следует добавить объявление `virtual bool operator==(const List<T>& rhs) const;`

```
if (myList.isEqual(yourList))
   cout << "Списки эквивалентны.\n";
```

С функцией *operator==* можно работать точно так же, как и с функцией *isEqual*.

```
if (myList.operator==(yourList))
   cout << "Списки эквивалентны.\n";
```

Это стало возможным потому, что *operator==* — это всего лишь имя функции. Однако вместо выражения *myList.operator==(yourList)* можно использовать более естественную форму записи *myList == yourList*.

Рассмотрим реализацию функции operator== для связанных списков.

```
bool List::operator==(const List& rhs) const
{
   bool isEqual;
   if (size != rhs.size)
      isEqual = false; // Списки имеют разную длину

   else if ( (head == NULL) && (rhs.head == NULL) )
      isEqual = true; // Оба списка пусты

   else // Списки имеют одинаковую положительную длину
        // Указатель на голову списка не равен константе NULL
   {
        // Сравниваем элементы
        ListNode *leftPtr = head;
        ListNode *rightPtr = rhs.head;
        for (int count = 1;
            (count <= size) &&
            (leftPtr->item == rightPtr->item);
            ++count)
        {
            leftPtr = leftPtr->next;
            rightPtr = rightPtr->next;
        } // Конец оператора for

        isEqual = count > size;
   } //Конец оператора if
   return isEqual;
} // Конец функции operator==
```

Обратите внимание, что эта функция зависит от реализации оператора == для элементов списка. Если элементами списка являются экземпляры какого-либо класса, то для них, в свою очередь, нужно предусматривать перегрузку оператора ==.

Аналогичным образом можно перегрузить операторы сравнения (<, <=, >, <+). Однако оператор присваивания (=) создает определенные проблемы. Допустим, что объекты *myList* и *yourList* представляют собой два экземпляра класса *List*. Если в список *yourList* записать несколько элементов, а затем написать оператор

```
myList = yourList;
```

то естественно было бы ожидать, что объект *myList* будет точной копией объекта *yourList*. Однако если не перегрузить оператор присваивания, вместо глубокой копии объекта *yourList* мы получим его поверхностную копию (рис. 4.18). Иными словами, будут скопированы только переменные-члены экземпляра *yourList*.

Для статических структур данных этого может оказаться вполне достаточно, однако для динамических структур необходимо выполнять глубокое копирование. Например, если класс *List* реализует связанный список, поверхностная копия будет содержать только переменные *size* (длину списка) и *head* (указатель на первый элемент списка). Сами элементы списка не будут скопированы.

Для перегрузки оператора присваивания в классе *List* можно написать следующее объявление[3].

```
virtual List& operator=(const List& rhs);
```

Аргумент *rhs* представляет собой объект, подлежащий копированию, т.е. объект, стоящий в правой части оператора присваивания. Функция operator== возвращает ссылку на вызывающий объект. Это позволяет выполнять цепочки операторов присваивания, например *myList=yourList=theirList*.

В реализации этой функции есть несколько тонкостей. Допустим, что объекты *myList* и *yourList* содержат несколько элементов. Что произойдет с элементами списка *myList* при выполнении оператора

Оператор присваивания сначала должен удалить объект myList, т.е. объект, стоящий в левой части

```
myList = yourList;
```

Как будто ничего страшного — все элементы списка *yourList* будут скопированы в список *myList*. Однако, если список является связанным, возникают проблемы. Перед копированием элементов связанного списка *yourList* нужно удалить из памяти связанный список *myList*. В противном случае произойдет утечка памяти, т.е. память, занятая объектом *myList*, не будет освобождена и станет недоступной. Следовательно, при выполнении присвоения объекта *yourList* объекту *myList* нужно последовательно выполнить несколько операций.

```
Освободить память, занятую объектом myList
for (каждый элемент в списке yourList)
{
   Выделить память для нового элемента списка myList
   Скопировать данные, содержащиеся в узле списка yourList
}
```

Аналогичные операции мы выполняли при создании деструктора и конструктора копирования класса *List*. Чтобы избежать избыточного кода, класс *List* можно реорганизовать, предусмотрев функцию-член *removeAll*, предназначенную для удаления всех элементов списка, и функцию-член *copyListNodes*, копирующую узлы связанного списка. Эти функции можно использовать не только для оператора присваивания, но и для других целей. Деструктор может вызывать функцию *removeAll*, а конструктор копирования — функцию *copyListNodes*. В то время как функция *copyListNodes* должна быть закрытой или защищенной, функцию *removeAll* следует объявить открытой. Реализация этих двух функций предоставляется читателям в виде задания 1, приведенного в конце главы.

Допустим, что в своей программе мы написали следующий оператор присваивания

Повышайте надежность оператора присваивания, проверяя особые ситуации

```
myList = myList;
```

[3] Полагая, что функция *operator==* определена для типа *T*, в шаблонный класс следует добавить объявление `virtual bool operator==(const List<T>& rhs);`

Попробуем проследить последовательность выполняемых операций. Сначала из памяти удаляется объект *myList*, а затем создается его копия. Однако, удалив объект, мы ничего не сможем скопировать! Эту ситуацию необходимо предусмотреть, чтобы операнды оператора присваивания не совпадали.

В ходе такой проверки необходимо сравнивать адреса списков, а не их элементы. В языке C++ для ссылки на вызывающий объект используется ключевое слово *this*. Итак, для сравнения адресов объектов, стоящих слева и справа от символа =, необходимо написать следующее выражение.

```
if (this != &rhs)
```

Рассмотрим реализацию оператора присваивания для связанных списков.

```
List & List::operator=(const List& rhs)
{
   // Проверка самоприсваивания
   if (this != &rhs)
   {
      removeAll(); // Удалить из памяти левый операнд
      copyListNodes(rhs); // Скопировать узлы списка
      size = rhs.size; // Скопировать размер списка
   } // Конец оператора if
   return *this;
} // Конец оператора operator=
```

Функция возвращает результат разыменования указателя this, т.е. текущий объект.

ОСНОВНЫЕ ПОНЯТИЯ

Принципы перегрузки операторов

1. Перегружать можно любой оператор языка C++, за исключением операторов
   `.  .*  ::  ?:  sizeof`
2. Нельзя определить новый оператор, перегрузив символ, который не являлся оператором языка C++.
3. Изменить приоритеты операторов и количество их операндов невозможно.
4. По крайней мере один операнд перегружаемого оператора должен быть экземпляром какого-либо класса.
5. Как правило, в классе перегружаются операторы присваивания (=), операторы проверки равенства и неравенства (== и !=), а также операторы сравнения (<. <=, >, >=).
6. Количество операндов перегружаемого метода изменить невозможно.

Итераторы

Итератор — это объект, который может перемещаться по коллекции аналогичных объектов, перебирая их один за другим. Иными словами, итератор "пробегает" набор объектов. Примером такой коллекции объектов является абстрактный список, описанный в главах 3 и 4. Значение индекса объекта в списке в сочетании с операцией *retrieve* представляет собой примитивную форму итератора. Индекс указывает искомый элемент списка, а функция *retrieve* извлекает его оттуда. Для доступа к сле-

Итератор — это объект, который может перемещаться по коллекции аналогичных объектов

дующему элементу списка значение индекса увеличивается на единицу, и снова применяется функция *retrieve*.

Как правило, итератор имеет операцию доступа к элементу по ссылке. Обычно такая операция реализуется путем перегрузки оператора разыменования *. Например, если для итератора *i* определить оператор *, то результатом выражения **i* будет элемент, на который ссылается итератор *i*.

Для итераторов предусматриваются также операции перемещения вперед и назад по коллекции объектов. Довольно часто эти операции являются перегрузкой операторов ++ и --. Кроме того, операторы == и != обычно перегружаются для проверки равенства итераторов.

ОСНОВНЫЕ ПОНЯТИЯ

Операции над итераторами

Операция	Описание
*	Возвращает элемент, на который ссылается итератор
++	Перемещает итератор на следующий элемент списка
--	Перемещает итератор на предыдущий элемент списка
==	Сравнивает два итератора на равенство
!=	Сравнивает два итератора на неравенство

Рассмотрим спецификацию класса *ListIterator*, который можно использовать для перемещения по абстрактному списку. Эта версия абстрактного списка будет описана в следующем разделе. Обратите внимание, что в классе *ListIterator* не предумотрен оператор --, следовательно, по списку можно перемещаться только вперед.

```
// ********************************************************
// Заголовочный файл ListIterator.h.
// Используется в версии абстрактного списка с итератором.
// ********************************************************
#include "ListNode.h" // Определение классов ListNode
                      // и ListItemType; класс ListNode
                      // объявляет класс ListIterator
                      // своим другом

class ListIterator
{
public:
   ListIterator(List *container, ListNode *nodePtr);

   const ListItemType & operator*();
   ListIterator operator++(); // Префикс ++

   bool operator==(const ListIterator& rhs) const;
   bool operator!=(const ListIterator& rhs) const;

   friend class List;

private:
   const List *container; // Абстрактный тип,
                          // связанный с итератором
   ListNode *cur; // Текущий узел списка
}; // end class
```

Здесь использована префиксная форма оператора ++. Он перемещает итератор на следующий элемент списка. Операторы == и != сравнивают итераторы на равенство и неравенство. Итератор содержит закрытые переменные, отслеживающие элементы списка, а также указатель на текущую позицию итератора в списке. Поскольку в классе *List* предусмотрены функции, возвращающие итераторы, членам класса *List* необходим доступ к закрытым разделам итератора. Для этого класс *List* объявляется другом класса *ListIterator*. Кроме того, класс *ListIterator* считается другом класса *ListNode*.

> Класс List является другом класса ListIterator, а класс ListIterator — другом класса ListNode

Реализации функций-членов класса *ListIterator* приведены ниже.

```
// ********************************************************
// Файл реализации ListIterator.cpp.
// ********************************************************
#include "ListIterator.h"
ListIterator::ListIterator(const List *aList,
                           ListNode *nodePtr)
                           : container(aList),cur(nodePtr)
{
} //Конец конструктороа

const ListItemType & ListIterator::operator*()
{
   return cur->item;
} // Конец функции operator*

ListIterator ListIterator::operator++()
{
   cur = cur->next;
   return *this;
} // Конец функции operator++

bool ListIterator::operator==(const ListIterator& rhs) const
{
   return ((container==rhs.container) &&
           (cur == rhs.cur));
} // Конец функции operator ==

bool ListIterator::operator!=(const ListIterator& rhs) const
{
   return !(*this == rhs);
} // Конец оператора !=
```

Реализация абстрактного списка с помощью итераторов

Теперь мы можем переписать класс *List*, используя клас *ListIterator*. Там, где раньше использовался индекс элемента, теперь используется значение итератора. Кроме того, многие операции возвращают итераторы в качестве результатов.

```
// ********************************************************
// Заголовочный файл ListI.h абстрактного списка.
// Реализация использует класс ListIterator.
// ********************************************************

#include "ListIterator.h"
```

```
#include "ListException.h"
class List
{
public:
// Конструкторы и деструктор:
   List();
   List(const List& aList);
   ~List();

// Операции над списком:
   bool isEmpty() const;
   int getLength() const;
   ListIterator insert(ListIterator iter,
                       ListItemType newItem)
                       throw(ListException);
   // Вставляет в список элемент после элемента,
   // на который ссылается итератор iter. Возвращается
   // итератор, ссылающийся на вставленный элемент.
   // Предусловие: итератор iter ссылается либо на элемент
   // списка, либо на его конец.
   // Постусловие: если итератор iter равен значению,
   // возвращенному функцией end(), элемент newItem
   // вставляется в конец списка. Возвращает  итератор,
   // установленный на элемент newItem.
   // Исключительные ситуации: если итератор инициализирован
   // неправильно, генерирует исключительную ситуацию
   // ListException.

   void retrieve(ListIterator iter,
                 ListItemType& dataItem) const
                 throw(ListException);
   // Извлекает элемент из списка.
   // Предусловие: итератор iter ссылается на элемент списка.
   // Постусловие: переменная dataItem хранит значение
   // искомого элемента.
   // Исключительные ситуации: если итератор инициализирован
   // неправильно, генерирует исключительную ситуацию
   // ListException.

   ListIterator remove(ListIterator iter) throw(ListException);
   // Удаляет элемент из списка и возвращает итератор,
   // установленный на элемент, следующий за удаленным.
   // Предусловие: итератор iter ссылается на элемент списка.
   // Постусловие: элемент, указанный итератором, удален из
   // списка. Возвращает итератор, ссылающийся на элемент,
   // следующий за удаленным. Если из списка удален последний
   // элемент, возвращает итератор, равный результату функции
   // end().
   // Исключительные ситуации: если итератор инициализирован
   // неправильно, генерирует исключительную ситуацию
   // ListException.

   ListIterator begin() const;
   // Возвращает итератор, ссылающийся на первый
   // элемент списка.
   // Предусловие: нет.
```

```
    // Постусловие: Возвращает итератор, ссылающийся на первый
    // элемент списка. Если список пуст, возвращает итератор,
    // равный результату функции end().

    ListIterator end() const;
    // Возвращает значение итератора, которое можно
    // использовать для проверки, достиг ли итератор конца
    // списка.
    // Предусловие: нет.
    // Постусловие: нет.

private:
    int size; // Количество элементов в списке
    ListNode *head; // Указатель на связанный список

    ListNode *findPrev(ListIterator iter);
    // Находит узел, предшествующий узлу, на который
    // ссылается итератор.
    // Предусловие: список не пуст (head != NULL).
    // Постусловие: возвращает указатель на узел, предшествующий
    // узлу, на который ссылается итератор. Если iter == end(),
    // возвращает указатель на последний узел списка.
}; // Конец класса
```

Функция-член *begin* позволяет клиенту инициализировать итератор, установив его на начало списка. Функция *end* возвращает значение, которое можно использовать для проверки, достиг ли итератор конца списка.

Ниже приведена простая программа, демонстрирующая применение класса *List*, использующего класс *ListIterator*.

```
#include "ListI.h"
#include <iostream>
using namespace std;

int main()
{
    List aList;

    ListIterator i = aList.begin();
    for (int j=1; j<=5; j++)
    {
        i = aList.insert(i, j);
    } // Конец оператора for

    i = aList.begin();
    while (i != aList.end())
    {
        cout << *i << " ";
        ++i;
    } // Конец оператора while
    cout << endl;

    return 0;
} // Конец функции main
```

Рассмотрим реализации функций-членов *begin*, *end* и *insert*.

```
#include "ListI.h"
ListIterator List:: begin() const
{
   ListIterator iter(this, head);
   return iter;
} // Конец функции begin

ListIterator List::end() const
{
   ListIterator iter(this, NULL);
   return iter;
} // Конец функции end

ListIterator List::insert(ListIterator iter,
                          ListItemType newItem)
{
   // Ссылается ли итератор на данный список
   if ((iter.container == this))
   {
      // Создать новый узел и поместить в него элемент newItem
      ListNode *newPtr = new ListNode(newItem, NULL);
      if (newPtr == NULL)
         throw ListException(
         "ListException: insert — невозможно выделить память");
      else
      {
         size++;
         // Добавить в список новый элемент
         if (iter == begin())
         {
            // Вставить новый узел в начало списка
            newPtr->next = head;
            head = newPtr;
         }
         else
         {
            ListNode *prev = findPrev(iter);
            // Вставить новый элемент перед узлом,
            // на который ссылается итератор iter
            newPtr->next = prev->next;
            prev->next = newPtr;
         } // Конец оператора if
      } // Конец оператора if
      return ListIterator(this, newPtr);
   } // Конец оператора if
   else
      throw ListException(
      "ListException: insert — неверное значение итератора");
} // Конец функции insert
```

Обработка исключительных ситуаций и реализация остальных функций-членов предоставляется читателям в качестве задания 14, приведенного в конце главы.

Резюме

1. Классы могут находиться в отношениях предка и потомка. Производный класс, или потомок, наследует все члены базового класса, определенного ранее, однако доступ имеет только к его открытым и защищенным членам. Закрытые члены базового класса доступны только функциям-членам этого класса (и друзьям). Доступ к защищенным членам класса открыт для функций-членов (и друзей) базового и производного классов, но не для клиентов этих классов.
2. При открытом наследовании открытые и защищенные члены базового класса становятся, соответственно, закрытыми и защищенными членами производного класса. Такие производные классы являются совместимыми со своим базовым классом. Это значит, что экземпляр базового класса можно заменять экземпляром производного класса. Между базовым и производным классами существует отношение "является" ("is-a").
3. Функция-член производного класса переопределяет функцию-член базового класса, если их объявления совпадают. Виртуальная функция-член производного класса замещает виртуальную функцию-член базового класса, если их объявления совпадают.
4. Виртуальная функция-член класса — это функция, которую можно замещать в производном классе. Если функция-член является виртуальной, ее можно либо реализовать, либо сделать чисто виртуальной. Чисто виртуальные функции не имеют тел.
5. Производный класс наследует интерфейс каждой функции из базового класса. Производный класс наследует реализации каждой не виртуальной функции из базового класса.
6. Класс, содержащий хотя бы одну чисто виртуальную функцию, называется абстрактным базовым классом. В таком классе определяются только самые важные члены, необходимые для его потомков и, следовательно, он может служить в качестве базового класса для целого семейства классов.
7. Раннее, или статическое связывание описывает ситуацию, в которой компилятор заранее определяет, какую функцию следует вызвать. Позднее, или динамическое связывание описывает ситуацию, в которой вызываемая функция идентифицируется в ходе выполнения программы.
8. Если при вызове метода используется указатель на объект, например, `spherePtr->displayStatistics()`, то при раннем связывании выбор вызываемого метода зависит от типа указателя, а при позднем — от типа объекта.
9. Шаблонные классы позволяют передавать тип данных, используемых внутри класса, в качестве параметра.
10. Операторам, существующим в языке C++, можно придать новый смысл, перегружая их для экземпляров заданного класса. Как правило, в любом классе следует перегружать операторы присваивания, проверки равенства и сравнения.
11. Итераторы предоставляют альтернативный способ перемещения по коллекции объектов.

Предупреждения

1. Если функция-член не является виртуальной, ее необходимо реализовать. В результате каждый производный класс вынужден наследовать все такие функции.
2. Если функция-член класса является виртуальной, но не чисто виртуальной, ее следует реализовать. Производный класс может либо использовать ее, либо замещать своей собственной реализацией. Преимущество чисто виртуальных функций заключается в том, что производный класс не может непреднамеренно унаследовать их реализации, поскольку их не существует.
3. Открытое наследование следует применять только, если между классами существует отношение "является".
4. Отказ от перегрузки операторов присваивания, проверки равенства и сравнения может привести к ошибкам, когда клиент попытается объединить экземпляры класса с операторами, значение которых для них не определено.

Вопросы для самопроверки

В первых трех вопросах рассматриваются классы *Sphere* и *Ball*, описанные в разделе "Еще раз о наследовании".

1. Напишите на языке C++ фрагменты программ, решающие следующие задачи.
   1.1. Объявить экземпляр *mySphere* класса *Sphere*, радиус которого равен 2.
   1.2. Объявить экземпляр *myBall* класса *Ball*, имеющий радиус, равный 6, и имя *Beach Ball*.
   1.3. Выведите на экран радиусы объектов *mySphere* и *myBall*.
2. Определите класс *Planet*, открыто наследующий класс *Ball*, как описано в главе. Новый класс должен иметь закрытую переменную, задающую расстояние от планеты до солнца, и открытую функцию-член, обеспечивающую доступ к этой переменной или изменение ее значения.
3. Ответьте на следующие вопросы.
   3.1. Может ли функция *resetBall*, являющаяся членом класса *Ball*, получить доступ к переменной-члену *theRadius* непосредственно, или для этого она должна вызвать функцию *resetBall* из класса *Sphere*? Обоснуйте свой ответ.
   3.2. Предположим, что переменная *theRadius* является защищенным, а не закрытым членом класса *Sphere*. Изменится ли ответ на предыдущий вопрос?
4. Проанализируйте классы ACR и VCR, описанные в разделе "Абстрактные базовые классы". Какие из членов класса ACR следует сделать виртуальными, предоставив их реализацию классу VCR?
5. Рассмотрите класс *SortedList*, описанный в главе. Допустим, объект *aList* представляет собой список имен, упорядоченных по алфавиту. Следует ли делать его экземпляром класса *SortedList*, или экземпляром одного из классов, производных от класса *SortedList*? Обоснуйте свой ответ.
6. Изучите класс, производный от абстрактного класса *BasicADT*, описанного в главе. Если объект *aList* является экземпляром производного класса, нужно ли в этом классе реализовывать функции *isEmpty* и *getLength*? Обоснуйте свой ответ.

7. Почему закрытые функции-члены не могут быть виртуальными?
8. Напишите фрагмент программы, в котором определяется экземпляр шаблонного класса *NewClass* из раздела "Шаблонные классы", содержащий символы. Присвойте переменной-члену объекта *myClass* символ *'c'*. В заключение напишите фрагмент программы для вывода на экран данных, содержащихся в объекте *myClass*.

Упражнения

1. Напомним классы *Sphere* и *Ball*, описанные в разделе "Еще раз о наследовании".

```
class Sphere
{
public:
   . . .
   double getArea() const; // Площадь поверхности
   void displayStatistics() const;
   . . .
}; // Конец класса

class Ball: public Sphere
{
public:
   . . .
   double getArea() const; // Площадь поперечного сечения
   void displayStatistics() const;
   . . .
}; // Конец класса
```

Допустим, что в реализации каждого варианта функции *displayStatistics* вызывается функция *getArea*.

1.1. Если объект *mySphere* является экземпляром класса *Sphere*, а объект *myBall* — экземпляром класса *Ball*, какая версия функции *getArea* вызывается при выполнении следующих операторов.

```
mySphere.displayStatistics();
myBall.displayStatistics();
```

1.2. Если операторы

```
Sphere *spherePtr;
Ball *ballPtr;
```

определяют указатели *spherePtr* и *ballPtr*, какая версия функции *getArea* будет вызвана при выполнении следующих операторов? Обоснуйте свой ответ.

```
spherePtr->displayStatistics();
spherePtr = &myBall;
spherePtr->displayStatistics();
ballPtr->displayStatistics();
```

2. Определите и реализуйте класс *Pen*, содержащий экземпляр класса *Ball* в качестве одного из своих членов. Опишите несколько членов класса *Pen*, например, переменную *color* и функции-члены *isEmpty* и *write*.

3. В разделе "Абстрактные базовые классы" описана версия абстрактного базового класса *EquidistantShape*, содержащая закрытую переменную-член *theRadius*.

   3.1. Модифицируйте класс *Sphere*, производный от класса *EquidistantShape*. Какие методы необходимо реализовать?

   3.2. Определите класс окружностей, производный от класса *EquidistantShape*.

   3.3. Модифицируйте абстрактный базовый класс *EquidistantShape* так, чтобы переменная *theRadius* стала защищенным членом, а не закрытым. Какие методы можно сделать виртуальными, а какие нужно реализовать?

   3.4. Выполните задания 3.1 и 3.2, имея в виду модификации, осуществленные в задании 3.3.

4. Проанализируйте следующие классы.

```
class Expr
{
public:
   . . .
   int getLength() const;
   virtual void display() const;
   . . .

private:
   char Array[MAX_STRING+1];
}; // Конец класса

class AlgExpr: public Expr
{
public:
   . . .
   bool isExpression() const;
   bool isBlank(int first, int last) const;
}; // Конец класса

class InfixExpr: public AlgExpr
{
public:
   . . .
   bool isExpression() const;
   int valueOf() const;
   void display() const;
   . . .

protected:
   int endFactor(int first, int last) const;
   int endTerm(int first, int last) const;
   int endExpression(int first, int last) const;

private:
```

```
    Stack<int> values;
    Stack<char> operators;
}; // Конец класса
```

Класс *AlgExpr* представляет алгебраические выражения, включая префиксные, постфиксные и инфиксные. Его функция-член *isExpression* просто проверяет, содержит ли выражение допустимые символы, но не анализирует их порядок.

Класс *InfixExpr* представляет инфиксные выражения. Его функция-член *isExpression* вызывает функцию *isBlank*, а функция-член *display* вызывает фу7нкцию *valueOf*.

4.1. Следует ли объявлять функцию-член *isBlank* открытой, защищенной или закрытой? Обоснуйте свой ответ.

4.2. Может ли объект *inExp* вызвать функцию *endExpression*, если объект *inExp* является экземпляром класса *InfixExpr* в функции *main*? Обоснуйте свой ответ.

4.3. Какие изменения нужно внести в рассмотренные выше классы, чтобы при вызове гарантировать выбор правильной версии функции *isExpression*?

5. Рассмотрим классы, описанные в предыдущем вопросе, и функцию *main*, содержащую такие объявления.

```
Expr exp;
AlgExpr aExp;
InfixExpr inExp;
```

5.1. Какие из этих объектов правильно вызывают функцию *getLength*?

5.2. Какие из этих объектов правильно вызывают функцию *isExpression*?

5.3. Какие из этих объектов правильно вызывают функцию *valueOf*?

5.4. Приведите пример совместимости объектов, написав объявление функции и вызвав ее в главном модуле.

6. Проанализируйте абстрактный фронтальный список, допускающий применение операции вставки, удаления и извлечения только первого элемента списка.

6.1. Определите класс *FrontList*, использующий связанный список и являющийся потомком абстрактного базового класса *BasicADT*.

6.2. Определите и реализуйте класс абстрактного стека, являющийся потомком класса *Frontlist*.

7. Определите абстрактный базовый класс *Person*, описывающий обычного человека. Затем определите производный класс *Student*, описывающий типичного студента. В заключение выведите из класса *Student* класс *GradStudent*, описывающий типичного аспиранта.

8. В разделе "Шаблонные классы" описан шаблонный класс *List*. Используя этот шаблон, напишите фрагмент программы, определяющий абстрактный список, состоящий из пяти целых чисел, заданных пользователем.

9. Перегрузите оператор присваивания (=) для реализации классов *Stack* (глава 6) и *Queue* (глава 7) в виде связанного списка. *Подсказка*: проанализируйте конструктор копирования.

Задания по программированию

1. В разделе "Перегруженные операторы" рассмотрена модификация класса *List*, предусматривающая включение открытой функции-члена *removeAll* и закрытой функции-члена *copyListNodes*. Завершите реализацию класса *List* в виде связанного списка, определив эти функции, а также перегруженные операторы проверки равенства (== и !=) и присваивания (=).

2. Определите и реализуйте класс *List* на основе массива, считая его производным от абстрактного базового класса *BasicADT*.

3. Выполните реализацию класса *SortedList*, содержащий экземпляр класса *List* в качестве своего члена.

4. Класс *List*, описанный в данной главе, не содержит метод *position*, возвращающий номер конкретного элемента по заданному значению. Такой метод, например, позволил бы удалять узел, передавая его номер функции *remove*.

   Определите потомка класса *List*, который имел бы функцию-член *position*, а также функции-члены для вставки, удаления и извлечения элементов по их значениям, а не по позициям в списке. Вставка всегда должна выполняться только в начало списка. Хотя элементы этого списка не упорядочены, новый абстрактный тип аналогичен классу *SortedList*, содержащему метод *locatePosition*.

5. Изучите абстрактный кольцевой список, в котором первый элемент следует непосредственно за последним. Например, если кольцевой список состоит из шести элементов, то извлечение или удаление восьмого элемента относится ко второму узлу. Будем считать, что вставка элемента в кольцевой список выполняется как обычно. Определите и реализуйте абстрактный кольцевой список в качестве производного от класса *List*.

6. В задании 11 из главы 6 описан абстрактный список, допускающий обход. В дополнение к стандартным операциям над стеком — *isEmpty*, *push*, *pop* и *getTop* — список, допускающий обход, предусматривает операцию *traverse*. Эта операция начинается со дна стека и выводит на экран каждый его элемент, пока не будет достигнута вершина.

   Определите и реализуйте абстрактный стек, допускающий обход, в качестве производного от класса *Stack*, описанного в главе 6.

7. В упражнении 8 из главы 7 определена двусторонняя очередь, или очередь с двусторонним доступом. Определите и реализуйте абстрактную двустороннюю очередь, считая ее класс производным от класса *Queue*, описанного в главе 7.

8. Завершите реализацию шаблонного класса *List*, описанного в разделе "Шаблонные классы".

9. Определите и реализуйте шаблонный класс абстрактного стека, описанного в главе 6.

   9.1. Используйте массивы.

   9.2. Используйте связанный список.

10. Определите и реализуйте шаблонный класс абстрактной очереди, описанной в главе 7.

    10.1. Используйте массивы.

10.2. Используйте связанный список.

11. Определите и реализуйте шаблонный класс абстрактного упорядоченного списка, производный от шаблонного класса *List*. Упорядоченный список должен предусматривать операции сравнения элементов неизвестного типа.

12. Поскольку алгебраические выражения являются символьными строками, их класс можно вывести из класса строк. Начните решать эту задачу с создания своего собственного класса строк *SimpleString* (задания 5 и 6 из главы 4). Предусмотрите следующие операции.

    - Ввод и вывод строки.
    - Вычисление длины строки.
    - Доступ к *n*-му символу, считая что первый символ имеет номер 1.

    Перегрузите операторы =, ==, !=, <, >, <= и >=.

    В задании 7 из главы 5 описаны грамматика и алгоритм распознавания инфиксных алгебраических выражений. Эта грамматика не допускает ассоциативности слева направо при выполнении операторов, имеющих одинаковый приоритет. Таким образом, выражение *a/b*c* считается недопустимым, а выражения *a/(b*c)* и *(a/b)*c* — правильными.

    В задании 6 из главы 6 описан алгоритм вычисления инфиксного синтаксически правильного выражения с помощью двух стеков.

    Разработайте и реализуйте класс алгебраических выражений, производный от класса *SimpleString*. Предусмотрите операцию *isExpression*, основанную на алгоритме распознавания из главы 5, а также операцию *valueOf*, основанную на алгоритме вычисления выражений из главы 6. Воспользуйтесь шаблонным классом стека, описанным в задании 9.

13. В главе 5 описан класс *Queen*, который используется при решении задачи о восьми ферзях. Шахматная доска имитируется двумерным массивом, являющимся членом класса. В задании 1 из главы 5 предлагалось написать программу решения задачи о восьми ферзях, основываясь на этих идеях. Модифицируйте программу, заменив двумерный массив классом, имитирующим шахматную доску.

14. Завершите версию класса *List*, использующую итераторы, предусмотрев обработку исключительных ситуаций.

Эффективность алгоритмов и сортировка

В этой главе ...

Измерение эффективности алгоритмов
- Быстродействие алгоритмов
- Степень роста временных затрат
- Оценка порядка величины и обозначение О-большое
- Перспективы
- Эффективность алгоритмов поиска

Алгоритмы сортировки и их эффективность
- Сортировка методом пузырька
- Сортировка методом вставок
- Сортировка слиянием
- Быстрая сортировка
- Поразрядная сортировка
- Сравнение алгоритмов сортировки

Резюме
Предупреждения
Вопросы для самопроверки
Упражнения
Задания по программированию

Введение. В этой главе описаны математические методы анализа алгоритмов. Эти методы являются одной из главных тем, изучаемых компьютерными науками. Они позволяют сравнивать эффективность алгоритмов, пользуясь точными оценками. В качестве примеров рассматриваются алгоритмы, описанные в предыдущих главах, в частности алгоритм поиска. Кроме того, в главе рассматривается сортировка данных, начиная с простых вопросов и заканчивая более сложными темами, связанными с рекурсией. Показано, что эффективность алгоритмов сортировки оценивается относительно легко.

Измерение эффективности алгоритмов

Сравнение алгоритмов между собой — основная тема компьютерных наук. Измерение эффективности алгоритмов чрезвычайно важно, поскольку выбор алгоритма сильно влияет на работу приложения. Эффективность алгоритмов, положенных в основу программы, определяет ее успех, будь то текстовый процессор, кассовый аппарат, банкомат, видеоигра или что-нибудь еще.

Допустим, два алгоритма решают одну и ту же задачу, например осуществляют поиск данных. Как их сравнить между собой и решить, какой из них лучше? В главе 1 были указаны факторы, влияющие на стоимость компьютерной программы. Некоторые из этих факторов касались стоимости работы, затраченной на разработку, сопровождение и использование программы. Другие факторы определяли стоимость ее выполнения, т.е. эффективность, выраженную объемом компьютерного времени, необходимого для выполнения программы.

Выбирая алгоритм, оцените его эффективность

До сих пор основное внимание мы уделяли человеческому фактору. В предыдущих главах акцент делался на стиле и читабельности программ. Было показано, что хорошо продуманные алгоритмы позволяют сократить время, необходимое для их программирования, а также облегчают сопровождение и модификацию программ. Основной задачей первой части книги было описание правильных приемов и стиля программирования. Хотя эта тема останется в поле нашего зрения и в дальнейшем, не стоит забывать о другом факторе — эффективности алгоритмов, которая определяет их выбор и способ реализации. Программы, представленные в книге, являются не только образцами хорошего стиля программирования, но и основаны на относительно эффективных алгоритмах.

Сравнение эффективности алгоритмов должно быть сосредоточено на их существенных различиях

Анализ алгоритмов (analysis of algorithms) — это область компьютерных наук, изучающая способы сравнения эффективности разных методов решения задач. Обратите внимание, что в этом определении использован термин *"метод решения задачи"*, а не *"программа"*. Следует подчеркнуть, что анализ алгоритмов, как правило, исследует *существенные* различия эффективности, которые обусловлены собственно методами решения задач, а не остроумными программистскими трюками. Изощренные приемы кодирования, позволяющие снизить стоимость вычислений, чаще всего снижают читабельность программы, тем самым повышая затраты на ее сопровождение и модификацию. Сравнение алгоритмов должно быть сосредоточено на их существенных различиях, поскольку именно их эффективность является основным фактором, определяющим общую стоимость решения. Если два алгоритма выполняются несколько часов, а разница между временем их выполнения составляет несколько секунд, их эффективность одинакова.

При анализе эффективности одинаково важны как время выполнения алгоритма, так и занимаемая им память. Для анализа этих факторов используются аналогичные методы. Поскольку в книге рассматриваются алгоритмы, не требующие значительных объемов памяти, в дальнейшем основное внимание будет уделяться их быстродействию.

Как сравнить быстродействие двух алгоритмов, решающих одну и ту же задачу? Для этого их можно запрограммировать на языке C++ и запустить обе программы. У этого подхода есть три существенных недостатка.

Три недостатка подхода, основанного на сравнении программ

1. **Как запрограммированы алгоритмы?** Допустим, алгоритм A_1 выполняется быстрее, чем алгоритм A_2. Это может быть связано с тем, что программа, реализующая алгоритм A_1, просто лучше написана. Следовательно, сравнивая время выполнения программ, вы на самом деле сравниваете реализации алгоритмов, а не сами алгоритмы. Реализации алгоритмов сравнивать бессмысленно, поскольку они очень сильно зависят от стиля программирования и не позволяют определить, какой из алгоритмов эффективнее.
2. **На каком компьютере должны выполняться программы?** Особенности конкретного компьютера также не позволяют сравнить эффективность алгоритмов. Один компьютер может работать намного быстрее другого, поэтому для выполнения программ необходимо применять один и тот же компьютер. Какой компьютер выбрать? Конкретные операции, составляющие основу алгоритма A_1, на одном из компьютеров могут выполняться быстрее, чем операции алгоритма A_2, а на другом компьютере — наоборот. Сравнение эффективности алгоритмов не должно зависеть от особенностей конкретного компьютера.
3. **Какие данные вводятся в программы?** Возможно, наиболее сложной проблемой является выбор тестовых данных. Всегда существует опасность, что при выборе конкретной тестовой задачи алгоритмы окажутся эффективнее, чем на самом деле. Например, сравнивая между собой последовательный и бинарный поиск элемента в упорядоченном массиве, можно предложить алгоритмам найти наименьший элемент. В этом случае алгоритм последовательного поиска сразу найдет искомый элемент. Следовательно, анализ эффективности не должен зависеть от выбора конкретных данных.

Анализ алгоритма не должен зависеть от конкретных реализаций, компьютеров и данных

Чтобы преодолеть эти трудности, специалисты в области компьютерных наук разработали математические методы анализа алгоритмов, не зависящие от их конкретных реализаций, компьютеров и выбора тестовых данных. Как показано в следующем разделе, эти методы начинаются с подсчета основных операций, выполняемых при решении задачи.

Быстродействие алгоритмов

Основной способ оценки эффективности алгоритма — подсчет его операций

В предыдущих главах мы неформально сравнивали между собой разные решения задач, подсчитывая количество выполняемых операций. Например, в главе 4 при сравнении реализаций абстрактного списка в виде массива и связанного списка оказалось, что функция `retrieve` позволяет непосредственно извлекать из массива n-й элемент списка, поскольку он хранится в ячейке `items[n-1]`. В то же время при извлечении n-го элемента из связанного списка нужно обойти весь список от начала до искомого элемента, на что потребуется n шагов.

Быстродействие алгоритма связано с количеством выполняемых операций, поэтому оценить его эффективность можно путем их простого подсчета. Рассмотрим еще несколько примеров.

Связанный список, допускающий обход. Напомним, что содержимое связанного списка, на который ссылается указатель *head*, можно вывести на экран с помощью следующего фрагмента программы.[1]

```
Node *cur = head;                    ← 1 присваивание
while(cur != NULL)                   ← n+1 сравнений
{
   cout << cur->item << endl;        ← n операций записи
   cur = cur->next;                  ← n присваиваний
} // Конец цикла while
```

Время, которое занимает вывод на экран содержимого связанного списка, прямо пропорционально количеству его узлов

В предположении, что связанный список состоит из n узлов, эти операторы выполняют $n+1$ присваивание, $n+1$ сравнение и n операций записи. Если каждое присваивание, сравнение и операция записи выполняется за a, b и c единиц времени, то на выполнение данного фрагмента программы уйдет $(n+1)*(a+c)+n*w$ единиц времени.[2] Итак, можно интуитивно догадаться, что вывод на экран содержимого 100 узлов связанного списка будет выполняться дольше, чем вывод содержимого 10 узлов.

Ханойские башни. В главе 5 доказано, что решение задачи о ханойских башнях с n кольцами достигается за 2^n-1 шагов. Если каждый шаг выполняется за m единиц времени, то решение будет найдено за $(2^n-1)*m$ единиц времени. Как мы вскоре убедимся, при увеличении количества колец время решения задачи о ханойских башнях резко возрастает.

Вложенные циклы. Рассмотрим алгоритм, содержащий вложенные циклы.

```
for (i = 1 до n)
   for (j = 1 до i)
      for (k = 1 до 5)
            Задача T
```

Если задача T решается за t единиц времени, то на выполнение наиболее глубоко вложенного цикла по переменной k уйдет $5 * t$ единиц времени. Цикл по переменной j затратит $5 * t * i$ единиц времени, а внешний цикл по переменной i будет выполняться за

$$\sum_{i=1}^{n}(5 * t * i) = 5 * t * (1 + 2 + \ldots + n) = 5 * t * n * (n + 1)/2$$

единиц времени.

Степень роста временных затрат

Время выполнения алгоритма выражается функцией, зависящей от размера задачи

Описанные выше примеры демонстрируют, что время выполнения алгоритма выражается функцией, зависящей от размера задачи. Способ измерения размера задачи зависит от конкретного приложения — количества узлов связанного списка, количества колец в задаче о ханойских башнях, размера массива или количества элементов стека. Итак, мы приходим к следующим выводам.

---

[1] На самом деле в главе 4 использовался оператор *for*. Здесь оператор *while* применен для упрощения анализа.

[2] Хотя в алгебре принято пропускать знак умножения, мы приводим его для наглядности.

Для решения задачи, имеющей размер n, алгоритм A затрачивает $n^2/5$ единиц времени.

*Для решения задачи, имеющей размер n, алгоритм B затрачивает 5*n единиц времени.*

Единицы времени, используемые при оценке эффективности этих алгоритмов, должны быть одинаковыми. Например, утверждение может выглядеть так.

Для решения задачи, имеющей размер n, алгоритм A затрачивает $n^2/5$ секунд.

Выше мы уже перечислили трудности, возникающие на этом пути. На каком компьютере алгоритм будет выполнен за $n^2/5$ секунд? Какая реализация этого алгоритма выполняется за $n^2/5$ секунд? При каких данных алгоритм выполняется за $n^2/5$ секунд?

Что конкретно нужно знать о быстродействии алгоритма? Важнее всего знать, насколько быстро возрастает время его выполнения с увеличением размера задачи. **Степень роста временных затрат** (growth rate) выражается следующими высказываниями.

Время выполнения алгоритма A прямо пропорционально n^2.

Время выполнения алгоритма B прямо пропорционально n.

Сравнение эффективности алгоритмов при решении больших задач

По этим утверждениям нельзя определить, сколько именно времени выполняется алгоритм *A* или *B*. Главное, что при решении больших задач алгоритм *B* работает намного быстрее. Иными словами, объем времени, затрачиваемый алгоритмом *B*, выраженный функцией, зависящей от размера задачи, растет медленнее, чем время выполнения алгоритма *A*, поскольку линейная функция растет медленнее квадратичной. Даже если *B* действительно затрачивает 5 * *n* секунд, в то время как алгоритм *A* выполняется за $n^2/5$ секунд, в целом алгоритм *B* выполняется значительно быстрее алгоритма *A*. Эта ситуация проиллюстрирована на рис. 9.1. Таким образом, выражение *Время выполнения алгоритма A прямо пропорционально n^2* точно характеризует эффективность алгоритма и не зависит от конкретных компьютеров и реализаций.

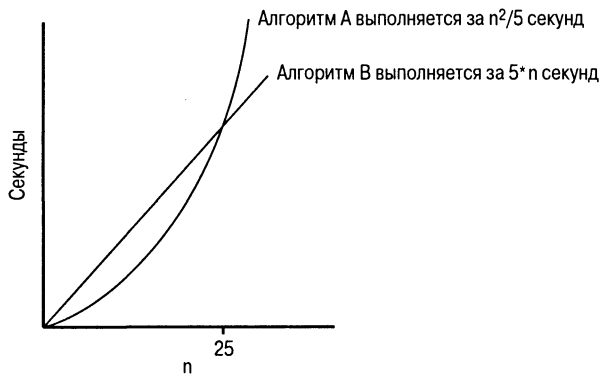


Рис. 9.1. Время выполнения алгоритмов как функция, зависящая от размера задачи n

Оценка порядка величины и обозначение О-большое

Допустим, выполняется следующее утверждение.

Время выполнения алгоритма A прямо пропорционально функции f(n).

В таких случаях говорят, что алгоритм A имеет **порядок f(n)** (order f(n)). Этот факт обозначается как **O(f(n))**. Функция *f(n)* называется **сложностью алгоритма** (growth-rate function). Поскольку в обозначении используется прописная буква O (первая буква слова *order* (порядок)), оно называется **обозначением О-большое** (Big-O notation). Если время решения задачи прямо пропорционально ее размеру n, то сложность задачи равна O(n), т.е. имеет порядок n. Если время решения задачи прямо пропорционально квадрату ее размера, т.е. n^2, то сложность задачи равна O(n^2) и т.д.

ОСНОВНЫЕ ПОНЯТИЯ

Определение порядка алгоритма

Алгоритм A имеет порядок $f(n)$. Этот факт обозначается как O(f(n)), если существуют константы k и n_0 такие, что при решении задачи, имеющей размер $n \geq n_0$, алгоритм A выполняется не более чем за $k * f(n)$ единиц времени.

Условие $n \geq n_0$ формализует интуитивное понятие большой задачи. Как правило, этому определению удовлетворяет большинство значений переменных k и n.

Проиллюстрируем определение несколькими примерами.

- Допустим, что при решении задачи, имеющей размер n, алгоритм выполняется за $n^2-3*n+10$ секунд. Если существуют такие константы k и n_0, что

  $$k * n^2 > n^2 - 3*n + 10 \text{ для всех } n \geq n_0,$$

  то алгоритм имеет порядок n^2. Фактически если константа k равна 3, а число n_0 равно 2, то

  $$3 * n^2 > n^2 - 3*n + 10 \text{ для всех } n \geq 2,$$

  как показано на рис. 9.2. Таким образом, при $n \geq n_0$ для выполнения алгоритма потребуется не более $k * n^2$ единиц времени.

- Ранее мы показали, что для вывода на экран первых n элементов связанного списка потребуется $(n+1)*(a+c)+n*w$ единиц времени. Поскольку неравенство $2*n \geq n+1$ выполняется для всех $n \geq 1$, имеет место неравенство

  $$(2*a+2*c+w)*n \geq (n+1)*(a+c)+n*w \text{ для всех } n \geq 1.$$

  Следовательно, сложность задачи имеет порядок O(n). Здесь константа k равна числу $2*a+2*c+w$, а константа n_0 равна 1.

- Для решения задачи о ханойских башнях потребуется $(2^n-1)*m$ единиц времени. Поскольку для всех $n \geq 1$ выполняется неравенство

  $$m*2^n > (2^n-1)*m,$$

  эта задача имеет сложность O(2^n).

Требование $n \geq n_0$ в определении величины O($f(n)$) означает, что оценка времени будет корректной лишь для достаточно больших задач. Иными словами, если задача имеет относительно небольшие размеры, то оценка времени ее решения будет слишком заниженной. Например, функция log n равна 0, если число n равно 1. Итак, из того, что число k * log 1 равно 0 при любых значениях константы k, следует неправильная оценка времени. Для выполнения любого алгоритма требуется ненулевое

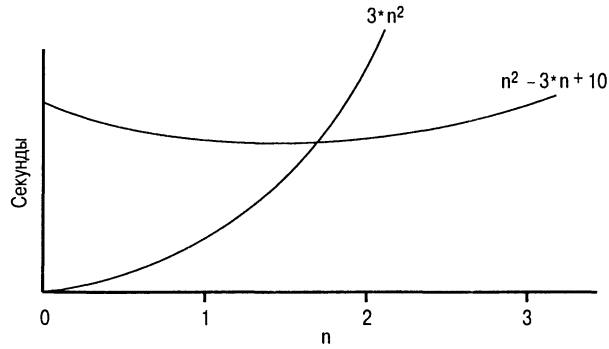


Рис. 9.2. Если $n \geq 2$, то $3 * n^2$ больше, чем $n^2 - 3 * n + 10$

количество единиц времени, даже если размер задачи равен 1. Следовательно, если $f(n) = \log n$, задачу при $n = 1$ следует рассматривать отдельно.

Скорость роста некоторых функций

Чтобы подчеркнуть значение правильной оценки степени роста функции, рассмотрим таблицу и график, представленные на рис. 9.3. В таблице (рис. 9.3, *а*) показаны разные значения аргумента n и приближенные значения некоторых функций, зависящих от n, в порядке увеличения скорости их роста.

$$O(1) < O(\log_2 n) < O(n) < O(n*\log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

По этой таблице можно оценить относительную скорость роста значений различных функций. (На рис. 9.3, *б* показаны графики этих функций.[3])

Интуитивная интерпретация сложности алгоритма

1	Константа 1 означает, что время выполнения алгоритма постоянно и, следовательно, не зависит от размера задачи
$\log_2 n$	Время выполнения **логарифмического алгоритма** (logarithmic algorithm) медленно возрастает с увеличением размера задачи. Если размер задачи возводится в квадрат, ее сложность увеличивается всего в два раза. Позднее мы убедимся, что алгоритм бинарного поиска обладает именно такими свойствами. Напомним, что при бинарном поиске массив делится пополам, а затем поиск продолжается в одной из полученных половин массива. Обычно логарифмические алгоритмы решают задачу, сводя ее к задаче меньшего размера. Основание логарифма не влияет на сложность алгоритма, поэтому его можно не указывать. Доказательство этого факта предоставляется читателям в качестве упражнения 6, помещенного в конце главы
n	Время выполнения **линейного алгоритма** (linear algorithm) прямо пропорционально размеру задачи. Если размер задачи возводится в квадрат, объем времени увеличивается точно так же
$n*\log_2 n$	Время выполнения алгоритма, имеющего сложность $O(n*\log_2 n)$ растет быстрее, чем у линейного алгоритма. Такие алгоритмы обычно разбивают задачи на подзадачи и решают их по отдельности. Пример такого алгоритма — сортировка слиянием — рассматривается далее

[3] Функция $f(n)=1$ на рисунке не показана, поскольку она не соответствует выбранному масштабу. Ее график представляет собой линию, проходящую через точку $y=1$ параллельно оси x.

n^2 Время выполнения **квадратичного алгоритма** (quadratic algorithm) быстро возрастает с увеличением размера задачи. В алгоритмах такого типа часто используются два вложенных цикла. Такие алгоритмы следует применять лишь для решения небольших задач. Далее в этой главе мы изучим несколько квадратичных алгоритмов сортировки

n^3 Время выполнения **кубического алгоритма** (qubic algorithm) еще быстрее возрастает с увеличением размера задачи по сравнению с квадратичным. Алгоритмы, использующие три вложенных цикла, часто оказываются кубическими. Такие алгоритмы следует применять лишь для решения небольших задач

2^n С увеличением размера задачи время выполнения **экспоненциального алгоритма** (exponential algorithm) обычно резко возрастает, поэтому на практике такие алгоритмы применяются редко

Если сложность алгоритма *A* пропорциональна функции $f(n)$, а сложность алгоритма *B* пропорциональна функции g, которая растет медленнее, чем функция f, то совершенно очевидно, что алгоритм *B* эффективнее алгоритма *A*, если размер решаемой задачи достаточно велик. Сложность алгоритма является решающим фактором при оценке его эффективности.

Для упрощения анализа алгоритмов будем использовать некоторые математические свойства обозначения O-большое. При этом следует иметь в виду, что запись $O(f(n))$ означает "порядка $f(n)$" или "имеет порядок $f(n)$". Символ O — это не функция.

Некоторые свойства функций

1. **При оценке сложности алгоритма можно учитывать только старшую степень.** Например, если алгоритм имеет сложность $O(n^3 + 4*n^2 + 3*n)$, он имеет порядок $O(n^3)$. Из таблицы, показанной на рис. 9.3, *a*, видно, что слагаемое n^3 намного больше, чем слагаемые $4*n^2$ и $3*n$, особенно при больших значениях n, когда порядок функции $n^3 + 4*n^2 + 3*n$ совпадает с порядком функции n^3. Иначе говоря, эти функции имеют одинаковый порядок роста. Итак, даже если сложность алгоритма равна $O(n^3 + 4*n^2 + 3*n)$, можно говорить, что он имеет порядок просто $O(n^3)$. Как правило, алгоритмы имеют сложность $O(f(n))$, где функцией $f(n)$ является одна из функций, перечисленных на рис. 9.3.
2. **При оценке сложности алгоритма можно игнорировать множитель при старшей степени.** Например, если алгоритм имеет сложность $O(5*n^3)$, можно говорить, что он имеет порядок $O(n^3)$. Это утверждение следует из определения величины $O(f(n))$, если положить $k = 5$.
3. **O(f(n))+O(g(n))=O(f(n)+g(n)).** Функции, описывающие сложность алгоритма, можно складывать. Например, если алгоритм имеет сложность $O(n^2)+O(n)$, то говорят, что он имеет сложность $O(n^2+n)$. В соответствии с п.1, это можно записать просто как $O(n^2)$. Аналогичные правила выполняются для умножения.

Из указанных выше свойств следует, что при оценке эффективности алгоритма нужно оценить лишь порядок его сложности. Точная формула, описывающая сложность алгоритма, зачастую весьма сложна, а иногда и просто невозможна.

При решении задач одинаковой размерности время выполнения алгоритма может изменяться

Наихудший и средний варианты. При решении конкретных задач одинаковой размерности время выполнения алгоритма может оказаться разным. Например, время, необходимое

для поиска n элементов, может зависеть от природы самих элементов. Обычно оценивается максимальное время, необходимое для решения задачи размера n, т.е. наихудший вариант. **Анализ наихудшего варианта** (worts-case analysis) приводит к оценке O($f(n)$), если при решении задачи, имеющей размер n, в наихудшем случае алгоритм выполняется не более чем за $k*f(n)$ единиц времени для всех значений n, за исключением их конечного числа. Хотя анализ наихудшего варианта приводит к пессимистическим оценкам, это не означает, что алгоритм всегда будет работать медленно. Следует иметь в виду, что наихудший вариант на практике встречается редко.

а)

	n					
Функция	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

б)

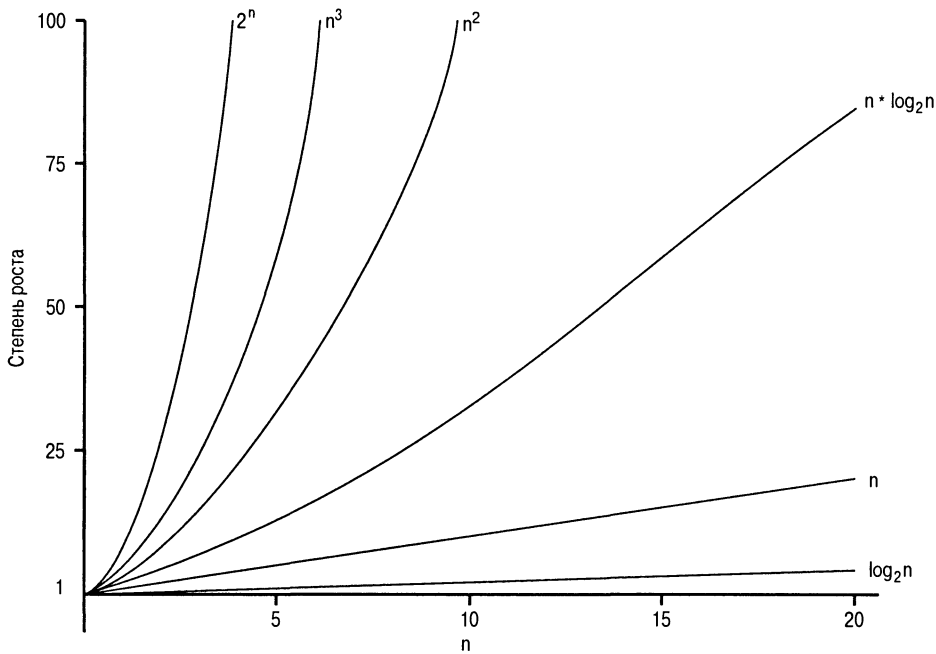


Рис. 9.3. Сравнение сложности алгоритмов: а) в табличном виде; б) в графическом виде

Анализ среднего варианта (average-case analysis) позволяет оценить среднее время выполнения алгоритма при решении задачи размера n. Говорят, что среднее время выполнения алгоритма A равно $O(f(n))$, если при решении задачи размера n оно не превышает величины $k * f(n)$ для всех значений n, за исключением их конечного числа. Как правило, анализ среднего варианта выполнить намного сложнее, чем анализ наихудшего варианта. Одна из трудностей заключается в определении вероятностей появления разных задач одинаковой размерности. Вторая трудность заключается в вычислении распределений разных значений. Анализ наихудшего варианта легче поддается вычислениям и поэтому выполняется намного чаще.

Перспективы

Сложность операции retrieve при реализации абстрактного списка в виде массива равна O(1)

Прежде чем перейти к оценкам порядка величин, характеризующих сложность конкретных алгоритмов, имеет смысл остановиться на перспективах. В качестве примера рассмотрим абстрактный список, состоящий из n элементов. Ранее мы уже видели, что при реализации абстрактного списка в виде массива операция `retrieve` имеет прямой доступ к i-му элементу списка независимо от значения n. На извлечение 1-го и 100-го элементов списка операция `retrieve` затрачивает одинаковое количество времени. Следовательно, сложность операции `retrieve` при реализации абстрактного списка в виде массива равна O(1).

Сложность операции retrieve при реализации абстрактного списка в виде связанного списка равна O(n)

Однако при реализации абстрактного списка в виде связанного списка операция `retrieve` выполняет n шагов, прежде чем найдет n-й элемент. Следовательно, ее сложность равна $O(n)$.

Анализируя алгоритмы, следует иметь в виду, что нас интересуют только *существенные* различия в оценках эффективности. Можно ли считать существенными описанные выше различия при оценке эффективности операции `retrieve`? При возрастании размера связанного списка для извлечения искомого элемента потребуется все больше времени, хотя время доступа к элементам массива постоянно. Итак, по мере увеличения длины списка различие между временем выполнения операции `retrieve` в разных реализациях рано или поздно станет существенным. В нашем примере оценки эффективности операции `retrieve` начинают различаться, если список достаточно велик. Если количество элементов списка не превышает 25, разницы между оценками эффективности операции `retrieve` в разных реализация абстрактного списка вообще нет.

Выбирая реализацию абстрактного типа данных, пытайтесь оценить, насколько часто в конкретном приложении выполняется та или иная операция

Рассмотрим конкретное приложение — например, проверку орфографии в текстовом процессоре, — которое часто извлекает элементы из списка, однако редко вставляет их туда или удаляет. Поскольку операция `retrieve` с массивами работает быстрее, чем со связанными списками, в этом случае следует предпочесть реализацию абстрактного списка в виде массива. С другой стороны, если в приложении часто выполняются операции вставки и удаления элементов, следует выбрать связанный список. Выбор реализации абстрактного типа данных сильно зависит от того, насколько часто в данном приложении выполняется та или иная операция. В следующей главе мы будем часто сталкиваться с такими ситуациями.

Редкие, но важные операции должны быть эффективными

Время выполнения некоторых операций над абстрактным типом данных может оказаться крайне важным, несмотря на то что эти опера-

ции могут выполняться относительно редко. Например, в системе управления полетами может быть предусмотрена аварийная операция, предотвращающая столкновение двух самолетов. Очевидно, эта операция должна выполняться как можно быстрее, даже если она применяется очень редко. Итак, выбирая реализацию абстрактного типа данных, необходимо проанализировать выполняемые операции, а также оценить частоту и время их выполнения.

Вскоре мы сравним два алгоритма поиска, один из которых имеет эффективность $O(n)$, а другой — $O(\log_2 n)$. Несмотря на то что алгоритм, имеющий сложность $O(\log_2 n)$, с большими массивами работает быстрее, на небольших массивах ($n < 25$) их эффективность различить невозможно. В принципе вполне возможно, что алгоритм, имеющий сложность $O(n)$, с маленькими массивами работает быстрее, поскольку в определение его эффективности входит константа k. Однако значительное различие в быстродействии этих алгоритмов проявляется лишь при решении больших задач. Это явление продемонстрировано на рис. 9.1.

Если размер задачи невелик, эффективностью алгоритма можно пренебречь

Итак, если максимальный размер задачи невелик, время выполнения алгоритмов разной сложности не будет значительно отличаться. Если заранее известно, что размер задачи никогда не будет больше, анализ эффективности алгоритмов можно не проводить. В таких случаях следует выбрать наиболее простой алгоритм, запрограммировать его и протестировать.

Следует стремиться к равновесию между быстродействием алгоритма и объемом занимаемой им памяти

Довольно часто при оценке эффективности алгоритмов нужно отыскать компромисс между быстродействием и занимаемым объемом памяти. Редко удается определенно сказать: "Этот метод является наилучшим способом решения данной задачи." Решение, которое выполняется относительно быстро, часто выдвигает завышенные требования к памяти компьютера. Иногда невозможно определенно сказать, что один алгоритм работает быстрее другого. Одни операции алгоритм *A* может выполнять быстрее, чем алгоритм *B*, а другие — наоборот. В силу этих причин, анализируя эффективность алгоритмов, нужно ориентироваться на конкретное приложение.

Сравнение стиля и эффективности алгоритмов

Итак, стиль и эффективность алгоритма одинаково важны. При анализе следует концентрироваться только на значительных различиях эффективности и не прибегать к программистским трюкам ради экономии нескольких миллисекунд. Более тщательный анализ эффективности связан с вопросами программирования, которые не следует рассматривать на этапе разработки алгоритма. Если вам удалось разработать метод решения задачи, эффективность которого значительно превышает эффективность остальных алгоритмов, смело выбирайте его, если размер задачи относительно велик. Если размер задачи невелик, возможно, лучшим окажется не самый эффективный алгоритм. Иными словами, если задача невелика, на первый план выступают другие факторы, например простота алгоритма.

Анализ сложности алгоритмов ориентируется на большие задачи

Фактически при анализе сложности алгоритма неявно предполагается, что он будет применяться для решения больших задач. Это предположение позволяет сосредоточиться только на порядке сложности алгоритма, пренебрегая другими факторами, поскольку при решении больших задач менее сложный алгоритм выполняется быстрее.

Эффективность алгоритмов поиска

В качестве еще одного примера рассмотрим два алгоритма поиска: последовательный и бинарный поиск элемента в массиве.

Последовательный поиск. Наихудший вариант O(n), средний вариант O(n), наилучший вариант O(1)

Последовательный поиск. При последовательном поиске элемента в массиве, имеющем длину n, элементы просматриваются по очереди, начиная с первого, пока не обнаружится искомый, либо не будет достигнут конец массива. В наилучшем случае искомым элементом является первый. Для его обнаружения понадобится только одно сравнение. Следовательно, в наилучшем случае сложность алгоритма последовательного поиска равна O(1). В наихудшем случае искомый элемент является последним. Для того чтобы его найти, понадобится n сравнений. Следовательно, в наихудшем случае сложность алгоритма последовательного поиска равна O(n). В среднем случае искомый элемент находится в средней ячейке массива и обнаруживается после $n/2$ сравнений.

Каков порядок алгоритма, если элемент не найден? Зависит ли его порядок от того, упорядочены элементы массива или нет? Эти вопросы для самопроверки читатели должны рассмотреть самостоятельно.

Бинарный поиск. Является ли бинарный поиск более эффективным, чем последовательный? Алгоритм бинарного поиска, описанный в главе 2, предназначен для поиска элемента в упорядоченном массиве и основан на повторяющемся делении частей массива пополам. Алгоритм определяет, в какой из двух частей находится элемент, если он действительно хранится в массиве, а затем повторяет процедуру деления пополам. Итак, в ходе бинарного поиска возникает несколько массивов меньшего размера, причем каждый раз размер очередного массива уменьшается вдвое по сравнению с предыдущим.

В ходе очередного разбиения массива алгоритм выполняет сравнения. Сколько сравнений выполняет алгоритм при поиске элемента в массиве, имеющем длину n? Точный ответ на этот вопрос, разумеется, зависит от позиции искомого элемента в массиве. Однако можно вычислить максимальное количество сравнений, т.е. наихудший вариант. Допустим, что $n = 2^k$, где k — некоторое натуральное число. Алгоритм поиска выполняет следующие шаги.

1. Проверяет среднюю ячейку массива, имеющего длину n.
2. Проверяет среднюю ячейку массива, имеющего длину $n/2$.
3. Проверяет среднюю ячейку массива, имеющего длину $n/2^2$ и т.д.

Чтобы проверить среднюю ячейку массива, сначала нужно поделить массив пополам. После того как массив, состоящий из n элементов, поделен пополам, делится пополам одна из его половин. Эти деления продолжаются до тех пор, пока не останется только один элемент. Для этого потребуется выполнить k разбиений массива. Это возможно, поскольку $n/2^k=1$. (Напомним, что $n = 2^k$.) В наихудшем случае алгоритм выполнит k разбиений и, следовательно, k сравнений. Поскольку $n = 2^k$, получаем, что

$k = \log_2 n$.

Что произойдет, если число n не будет степенью двойки? Легко найти наименьшее число k, удовлетворяющее условию

$2^{k-1} < n < 2^k$.

(Например, если n равно 30, то k=5, поскольку $2^4 = 16 < 30 < 32 < 2^5$.) Алгоритм по-прежнему выполнит по меньшей мере k разбиений, пока не возникнет массив, состоящий из одного элемента. Итак, получаем следующие оценки.

$k-1 < \log_2 n < k$,

$k < 1+\log_2 n < k+1$,

$k \approx 1+\log_2 n$.

В наихудшем случае сложность бинарного поиска равна $O(\log_2 n)$

Следовательно, в наихудшем случае сложность бинарного поиска оценивается величиной $O(\log_2 n)$, если $n \neq 2^k$. В принципе сложность алгоритма в наихудшем случае имеет порядок $O(\log_2 n)$ для любого значения n.

Можно ли утверждать, что бинарный поиск лучше последовательного? Намного лучше! Например, $\log_2 1000000 = 19$, поэтому алгоритм последовательного поиска выполнит миллион сравнений, в то время как алгоритм бинарного поиска — не более 20. Если массивы велики, бинарный поиск намного эффективнее последовательного.

Однако следует иметь в виду, что условие упорядоченности массива приводит к дополнительным затратам, которые могут стать существенными. В следующем разделе мы попробуем их оценить.

Алгоритмы сортировки и их эффективность

Сортировка (sorting) — это процесс упорядочения набора элементов в возрастающем или убывающем[4] порядке. Сортировка необходима во многих ситуациях. Например, иногда нужно упорядочить данные, прежде чем включить их в отчет. Однако чаще всего сортировка выполняется в качестве первого шага некоторых алгоритмов. Например, поиск данных является одной из наиболее распространенных задач, выполняемых компьютерами. Если набор данных достаточно велик, необходимо применять эффективный метод, например алгоритм бинарного поиска. Однако для применения алгоритма бинарного поиска необходимо, чтобы массив был упорядочен. Следовательно, если исходный набор не был упорядочен, сортировка данных должна предшествовать бинарному поиску.

В главе рассматривается только внутренняя сортировка

Алгоритмы сортировки разделяются на две категории. При выполнении **внутренней сортировки** (internal sort) предполагается, что все данные находятся в оперативной памяти компьютера. В этой главе рассматриваются только алгоритмы внутренней сортировки. При выполнении **внешней сортировки** (external sort) данные могут храниться на вспомогательных запоминающих устройствах, например на жестком диске. Внешняя сортировка рассматривается в главе 14.

Упорядочивать можно целые числа, строки символов и даже объекты. Легко представить себе результаты сортировки набора целых чисел или строк. Однако для набора объектов эта операция непривычна. Если каждый объект содержит только одну переменную-член, то их сортировка ничем не отличается от сортировки целых чисел. Однако если в объектах содержатся несколько данных-членов, нужно указать, какая переменная определяет порядок следования объектов. Эта переменная-член называется **ключом сортировки** (sort key). Например, если объекты хранят информацию о людях, их можно упорядочить по имени, возрасту или почтовому индексу. Независимо от выбора ключа сортировки алгоритм сортировки упорядочивает все объекты по значению этой переменной-члена.

[4] Мы не будем запрещать дубликаты, поэтому *возрастающий* порядок означает не убывающий, а *убывающий* — не возрастающий.

Для простоты в этой главе будем предполагать, что сортировка применяется к числам или символам. Алгоритмы сортировки объектов рассмотрены в упражнениях. Все алгоритмы в этой главе ориентируются на возрастающий порядок. Изменить порядок на убывающий довольно просто. В каждом примере предполагается, что данные хранятся в массиве.

Сортировка методом выбора

Представьте себе данные, которые можно проверять все сразу. Для их упорядочения можно было бы выбрать наибольший элемент, поставить его на свое место, затем найти следующий наибольший элемент, поставить его на свое место т.д. Карточным игрокам этот процесс напоминает перетасовку карт в определенном порядке. Этот интуитивный алгоритм формализуется с помощью **сортировки методом выбора** (selection sort).

Выбор наибольшего элемента

Чтобы упорядочить массив в возрастающем порядке, нужно выбрать наибольший элемент. Поскольку наибольший элемент нужно поставить в самый конец массива, его нужно поменять местами с последним элементом, даже если эти элементы идентичны. Теперь, игнорируя последний (наибольший) элемент массива, выполним поиск наибольшего элемента в оставшейся части массива и поменяем его местами с предпоследним элементом исходного массива. Этот процесс продолжается до тех пор, пока не будут найдены и переставлены n-1 элемент из n элементов массива. Оставшийся элемент, стоящий первым, не нарушает порядок и поэтому не рассматривается.

На рис. 9.4 показан пример сортировки методом выбора. Среди пяти целых чисел выбирается наибольшее — число 37, — которое меняется местами с последним элементом массива — числом 13. (Числа, стоящие на правильных местах, выделены полужирным шрифтом. Это соглашение принято для всех остальных рисунков этой главы.) Затем среди оставшихся четырех чисел снова выбирается наибольшее — число 29, — которое меняется местами с предпоследним элементом — числом 13. Обратите внимание, что следующий выбор — число 14 — уже стоит на правильном месте, однако алгоритм игнорирует этот факт и выполняет фиктивную перестановку числа 14 на одном и том же месте. В принципе намного эффективнее выполнять фиктивные перестановки, чем каждый раз проверять, нужна перестановка или нет. В заключение выбирается число 13, которое меняется местами со вторым элементом массива — числом 10. Теперь массив упорядочен по возрастанию.

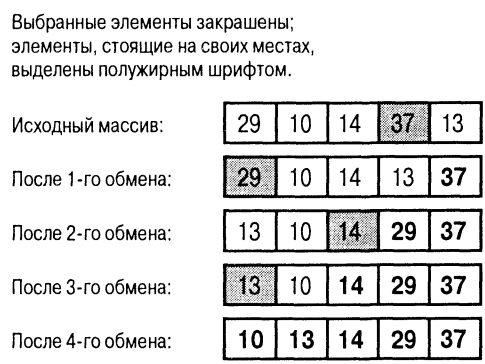


Рис. 9.4. Сортировка массива, состоящего из пяти целых чисел, методом выбора

Рассмотрим функцию на языке C++, выполняющую сортировку массива *theArray* методом выбора.

```
typedef тип-элемента-массива DataType;
void selectionSort(DataType theArray[], int n)
// ------------------------------------------------
// Упорядочивает элементы массива по возрастанию.
// Предусловие: массив theArray состоит из n элементов.
// Постусловие: массив theArray упорядочен по возрастанию;
// число n остается без изменения.
// Вызываемые функции: indexOfLargest, swap.
// ------------------------------------------------
{
   // last = индекс последнего элемента в подмассиве,
   // подлежащем сортировке,
   // largest = индекс найденного наибольшего элемента

   for (int last = n-1; last >= 1; --last)
   {
     // Инвариант: массив theArray[last+1..n-1] упорядочен,
     // а его размер превышает размер массива theArray[0..last]

     // Выбираем наибольший элемент в массиве theArray[0..last]
     int largest = indexOfLargest(theArray, last+1);

     // Меняем местами элементы theArray[largest] и theArray[last]
     swap(theArray[largest], theArray[last]);
   } // Конец оператора for
} // Конец функции selectionSort
```

Функция *selectionSort* вызывает две функции: *indexOfLargest* и *swap*.

```
int indexOfLargest(const DataType theArray[], int size)
// ------------------------------------------------
// Находит наибольший элемент массива.
// Предусловие: размер массива theArray задается аргументом
// size, причем size >= 1.
// Постусловие: возвращает индекс наибольшего элемента массива. //
Аргументы не изменяются.
// ------------------------------------------------
{
   int indexSoFar = 0; // Индекс наибольшего элемента,
                       // найденного до сих пор.
   for (int currentIndex = 1; currentIndex < size;
                              ++currentIndex)
   {
       // Инвариант: theArray[indexSoFar] >=
       // theArray[0..currentIndex-1]
       indexSoFar = currentIndex;
   } // Конец оператора for

   return indexSoFar; // Индекс наибольшего элемента
} // Конец функции indexOfLargest

void swap(DataType& x, DataType& y)
// ------------------------------------------------
```

```
// Обмен двух элементов.
// Предусловие: аргументы x и y — элементы, подлежащие обмену.
// Постусловие: содержимое ячейки x находится в ячейке y, и наоборот.
// ---------------------------------------------------
{
   DataType temp = x;
   x = y;
   y = temp;
} // Конец функции swap
```

Анализ. Как следует из описания алгоритма, сортировка сводится к сравнениям, обменам и перестановкам элементов. Для начала подсчитаем количество этих операций. Как правило, такие операции более дорогостоящи, чем операции управления счетчиком цикла или манипуляции с индексами массива, особенно если в массиве хранятся не числа или символы, а более сложные объекты. Поэтому в нашем анализе мы будем пренебрегать второстепенными операциями. Убедитесь, что это не влияет на окончательный результат (упражнение 7).

Очевидно, цикл *for* в функции *selectionSort* выполняется n-1 раз. Таким образом, функция *selectionSort* n-1 раз вызывает функции *indexOfLargest* и *swap*. При каждом вызове функции *indexOfLargest* ее цикл выполняется *last* раз (т.е. *size*-1 раз, где *size* равно *last*+1). Итак, при n-1 вызове функции *indexOfLargest* для значений переменной *last* от n-1 до 1 общее количество итераций цикла равняется

$$(n-1)+(n-2)+ \ldots +1 = n*(n-1)/2.$$

Поскольку при каждой итерации в функции *indexOfLargest* выполняется одно сравнение, их общее количество равно

$$n*(n-1)/2.$$

В результате для n-1 вызова функции *swap* выполняется n-1 обменов. Для каждого обмена нужно выполнить три присваивания. Следовательно, общее количество операций присваивания равно

$$3*(n-1).$$

В сумме алгоритм сортировки методом выбора выполняет

$$n*(n-1)/2+3*(n-1) = n^2/2 + 5*n/2-3$$

основных операций.

Сложность алгоритма сортировки методом выбора равна $O(n^2)$

Применяя свойства обозначения О-большое, можем отбросить слагаемые с младшими степенями. В итоге получим величину $O(n^2/2)$. Игнорируя множитель 1/2, получаем окончательную оценку $O(n^2)$. Итак, сложность алгоритма сортировки методом выбора равна $O(n^2)$.

Хотя алгоритм сортировки методом выбора не зависит от первоначального расположения данных, что можно отнести к преимуществам этого метода, его можно применять только для небольших массивов, поскольку величина $O(n^2)$ довольно быстро растет. Хотя алгоритм выполняет $O(n^2)$ сравнений, в ходе сортировки осуществляется только $O(n)$ перестановок. Алгоритм сортировки методом выбора хорош, когда перестановки представляют собой затратные операции, а сравнения — нет. Это может произойти, когда каждый элемент данных достаточно велик, а ключ сортировки мал. Разумеется, хранение данных в связанном списке позволяет эффективно выполнять перестановки элементов любого алгоритма.

Сортировка методом пузырька

Возможно, этот метод вам уже знаком. Именно по этой причине мы его рассматриваем, хотя на практике он не слишком хорош. **Алгоритм сортировки методом пузырька** (bubble sort) сравнивает между собой соседние элементы и меняет их местами, если они нарушают порядок. Для этого приходится несколько раз просматривать одни и те же элементы. Во время первого прохода сравниваются два первых элемента массива; если они нарушают порядок, их меняют местами. Затем сравнивается другая пара, т.е. 2-й и 3-й элементы. Если они нарушают порядок, их меняют местами. Просмотр, сравнение и обмен двух элементов выполняется до тех пор, пока не будет достигнут конец массива.

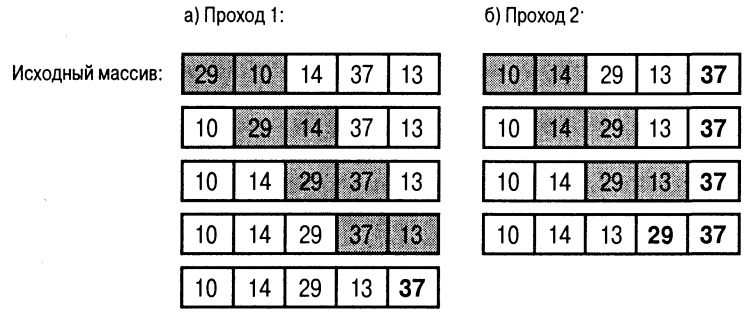


Рис. 9.5. Первые два прохода при сортировке массива, состоящего из пяти целых чисел, методом пузырька: а) первый проход; б) второй проход

На рис. 9.5, *а* показаны результаты первого прохода алгоритма сортировки методом пузырька на примере массива, содержащего пять целых чисел. Сначала сравниваются между собой элементы первой пары — числа 29 и 10. Они нарушают заданный порядок, поэтому их меняют местами. Затем сравниваются элементы второй пары — числа 29 и 14, поэтому их также меняют местами. После этого сравниваются элементы третьей пары — числа 29 и 37. Они не нарушают установленный порядок, поэтому остаются на своих местах. В заключение меняются местами элементы последней пары — числа 37 и 13.

В ходе проверок последовательных пар элементов наибольший элемент "всплывает вверх" (в конец массива), как пузырек на поверхность воды

Хотя после первого прохода массив остается неупорядоченным, наибольший элемент оказывается в конце массива, "всплывая", как пузырек на поверхность воды. Во время второго прохода нужно вернуться к началу массива и обработать его точно так же, как и в первый раз, останавливая обработку на предпоследнем элементе. Таким образом, при втором проходе просматриваются n-1 элемент массива. После второго прохода второй наибольший элемент окажется на предпоследнем месте, как показано на рис. 9.5, *б*. Теперь, игнорируя два последних элемента, которые уже поставлены в нужном порядке, следует продолжить обработку массива, пока он весь не будет упорядочен.

При сортировке массива методом пузырька массив необходимо пройти несколько раз

Несмотря на то что алгоритм сортировки методом пузырька состоит из n-1 прохода, в некоторых случаях удается обойтись меньшим количеством шагов. Таким образом, процесс можно прекратить, если в ходе проверки не выполнено ни одной перестановки. В приведенной ниже функции `bubbleSort`, написанной на языке C++, для сигнализации о перестановке используется булева переменная `sorted`. Функция `bubbleSort` использует функцию `swap`, описанную ранее.

```
void bubbleSort(DataType theArray[], int n)
// ------------------------------------------------
// Упорядочивает элементы массива в возрастающем порядке.
// Предусловие: массив theArray состоит из n элементов.
// Постусловие: массив theArray упорядочен по возрастанию;
// число n остается без изменения.
// Вызываемая функция: swap.
// ------------------------------------------------
{
   bool sorted = false; // Если выполняется перестановка,
                        // принимает значение false
   for (int pass = 1; (pass < n) && !sorted; ++pass)
   {
     // Инвариант: массив theArray[n+1-pass..n-1] упорядочен,
     // а его размер больше размера массива theArray[0..n-pass]
     sorted = true; // Массив упорядочен

     for (int index = 0; index < n-pass; ++index)
     {
         // Инвариант: размер массива theArray[0..index-1]
         // не превышает размера массива theArray[index]
         int nextIndex = index + 1;
         if (theArray[index] > theArray[nextIndex])
         {
             // Переставляем элементы
             swap(theArray[index], theArray[nextIndex]);
             sorted = false; // Признак перестановки
         } // Конец оператора if
     } // Конец оператора for

       // Диагностическое утверждение: размер массива
       // theArray[0..n-pass-1] меньше размера массива
       // theArray[n-pass]
     } // Конец оператора for
} // Конец функции bubbleSort
```

Анализ. Как указывалось выше, количество проходов при использовании метода пузырька не превышает n-1. При первом проходе выполняется n-1 сравнение и не больше n-1 перестановок. При втором проходе выполняется n-2 сравнения и не больше n-2 перестановок. В общем, при i-м проходе выполняется n-i сравнений и не больше n-i перестановок. Следовательно, в худшем случае при сортировке методом пузырька будет выполнено

$$(n\text{-}1)+(n\text{-}2)+\ldots+1 = n*(n\text{-}1)/2$$

сравнений и столько же перестановок. Напомним, что при каждой перестановке выполняется три присваивания. Таким образом, общее количество основных операций в худшем случае равно

$$2 * n * (n\text{-}1) = 2 * n^2 - 2*n.$$

Следовательно, в худшем случае сложность алгоритма сортировки методом пузырька равна $O(n^2)$.

Сортировка методом пузырька. Наихудший вариант $O(n^2)$, наилучший вариант O(n)

Лучшим считается вариант, когда исходные данные уже упорядочены. В этом случае алгоритм сортировки методом пузырька сделает только один проход, выполнив n-1 сравнений, и ни одной перестановки.

Сортировка методом вставок

Представьте себе колоду карт, из которой каждый раз вынимается и вставляется на указанное место одна карта. Такой способ упорядочения называется **сортировкой методом вставок** (insertion sort). Этот алгоритм уже был описан в главе 4 в контексте связанного списка. Мы создавали упорядоченный связанный список, считывая данные из файла, содержащего целые числа, записанные в произвольном порядке. Для этого мы многократно вызывали функцию, предназначенную для вставки целого числа в нужное место упорядоченного связанного списка.

> Извлеките элементы из неупорядоченной части массива и вставьте их в нужное место упорядоченной части

Описанную выше стратегию можно применить для сортировки массива. В этом случае массив делится на две части: упорядоченную и неупорядоченную, как показано на рис. 9.6. Вначале весь массив неупорядочен. На каждом шаге метода вставок из неупорядоченной части извлекается первый элемент, который затем вставляется в нужное место упорядоченной части. Первый шаг тривиален: переместить элемент `theArray[0]` из неупорядоченной части в упорядоченную. Для этого даже не нужно переставлять элементы массива. Следовательно, этот шаг можно пропустить, считая, что элемент `theArray[0]` уже принадлежит упорядоченной части, а неупорядоченной частью массива является отрезок `theArray[1...n-1]`. Тот факт, что элементы в упорядоченной части расположены в порядке возрастания, является инвариантом алгоритма. Поскольку на каждом шаге размер упорядоченной части увеличивается на единицу, а размер неупорядоченной части, соответственно, на единицу уменьшается, в момент окончания алгоритма весь массив окажется упорядоченным.

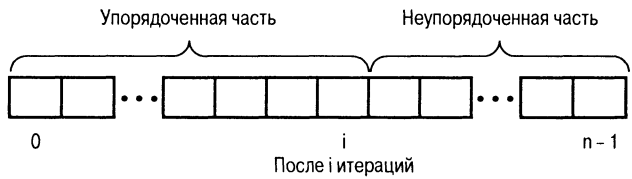


Рис. 9.6. Сортировка методом вставок разбивает массив на две части

На рис. 9.7 показаны результаты сортировки массива, состоящего из пяти целых чисел, методом вставок. В исходном положении упорядоченная часть массива состоит из единственного элемента `theArray[0]`, равного 29, а к неупорядоченной части относятся все остальные элементы массива. Извлечем из неупорядоченной части массива ее первый элемент — число 10 — и вставим в соответствующее место упорядоченной части. Для этого понадобится сдвинуть элементы массива, чтобы освободить место для вставляемого числа. Снова извлечем из вновь образованной неупорядоченной части массива ее первый элемент — число 14 — и вставим в соответствующее место упорядоченной части и т.д.

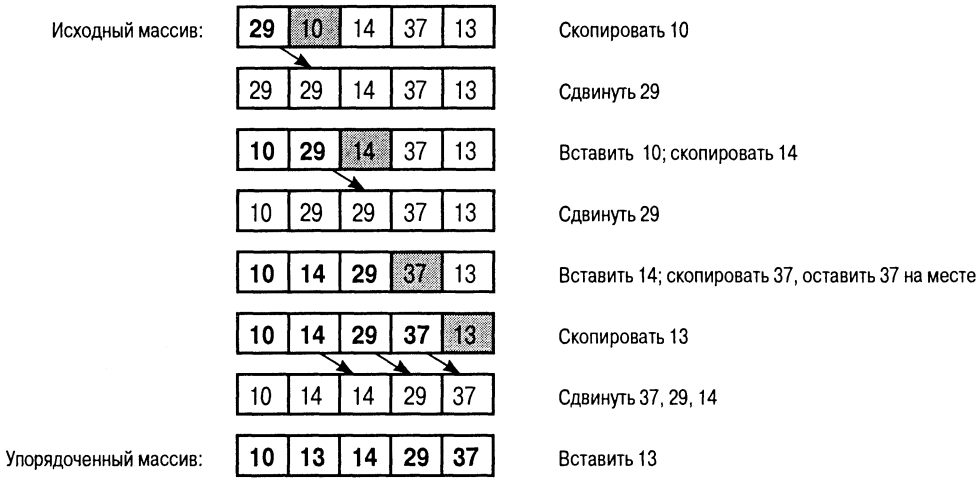


Рис. 9.7. Сортировка массива, состоящего из пяти целых чисел, методом вставок

Рассмотрим функцию на языке C++, выполняющую сортировку массива, состоящего из *n* элементов, методом вставок.

```
void insertionSort(DataType theArray[], int n)
// ------------------------------------------------
// Упорядочивает элементы массива в возрастающем порядке.
// Предусловие: массив theArray состоит из n элементов.
// Постусловие: массив theArray упорядочен  по возрастанию;
// число n остается без изменения.
// ------------------------------------------------
{
// unsorted = индекс первого элемента неупорядоченной части;
// loc = индекс ячейки упорядоченной части, в которую
// производится вставка;
// nextItem = следующий элемент неупорядоченной части.
// В исходном положении упорядоченная часть состоит
// из единственного элемента theArray[0],
// неупорядоченной частью массива является отрезок
// theArray[1..n-1]. В общем случае упорядоченной частью
// массива является отрезок theArray[0..unsorted-1],
// а неупорядоченной — отрезок theArray[unsorted..n-1]

   for (int unsorted = 1; unsorted < n; ++unsorted)
   {
       // Инвариант: отрезок theArray[0..unsorted-1] упорядочен

       // Находим индекс нужной ячейки (loc) в отрезке
       // theArray[0..unsorted] для вставки элемента
       // theArray[unsorted], являющегося первым элементом
       // неупорядоченной части; при необходимости выполняем
       // сдвиг элементов, освобождая место для вставки.

       DataType nextItem = theArray[unsorted];
       int loc = unsorted;

       for (;(loc > 0) && (theArray[loc-1 ]> nextItem);--loc)
```

```
            // Сдвигаем отрезок theArray[loc-1] вправо
            theArray[loc] = theArray[loc-1];

            // Диагностическое утверждение: ячейка theArray[loc]
            // содержит элемент nextItem

            // Вставляем элемент nextItem в упорядоченную часть
            theArray[loc] = nextItem;
        } // Конец цикла for
    } // Конец функции insertionSort
```

Анализ. Внешний цикл в функции *insertionSort* выполняется n-1 раз. Этот цикл содержит внутренний цикл, который выполняется не больше чем *unsorted* раз для значений переменной *unsorted*, изменяющихся от 1 до n-1. Таким образом, в худшем случае алгоритм выполняет

$$1 + 2 + \ldots + (n-1) = n*(n-1)/2$$

сравнений. Кроме того, в худшем случае столько же раз внутренний цикл сдвигает элементы.

Во внешнем цикле перемещение элементов на каждой итерации выполняется дважды, т.е. в сумме $2*(n-1)$ раз. Итак, подведем итог: в наихудшем варианте выполняется

$$n * (n-1) + 2 * (n-1) = n^2 + n - 2$$

основных операций.

В наихудшем случае сложность алгоритма сортировки методом вставок равна $O(n^2)$

Следовательно, в худшем случае сложность алгоритма сортировки методом вставок равна $O(n^2)$. Для небольших массивов, скажем, содержащих не более 25 элементов, алгоритм сортировки методом вставки предпочтительнее, поскольку он понятнее остальных алгоритмов. Однако для больших массивов этот метод совершенно неэффективен.

Сортировка слиянием

Разделяй и властвуй

Два важных алгоритма сортировки, основанных на принципе "разделяй и властвуй", сортировка слиянием и быстрая сортировка, имеют элегантное рекурсивное воплощение и чрезвычайно эффективны. В этом разделе мы рассмотрим сортировку массивов методом слияний, однако в главе 14 будет показано, что этот алгоритм можно обобщить на внешние файлы. Формулируя алгоритм, будем пользоваться обозначением отрезка массива *theArray[firat...last]*.

Алгоритм делит массив пополам, рекурсивно упорядочивает обе половины, а затем объединяет их в одно целое

Алгоритм сортировки методом слияний является рекурсивным. Его эффективность не зависит от порядка следования элементов в исходном массиве. Допустим, что мы разделили массив пополам, рекурсивно упорядочили обе половины, а затем объединили их в одно целое, как показано на рис. 9.8. На рисунке показано, что части массива <1, 4, 8> и <2, 3> объединяются в массив <1, 2, 3, 4, 8>. В ходе слияния элементы, стоящие в разных частях массива, попарно сравниваются друг с другом, и меньший элемент отправляется во временный массив. Этот процесс продолжается до тех пор, пока не будет использована одна из двух частей массива. Теперь достаточно просто скопировать оставшиеся элементы во временный массив. В заключение содержимое временного массива копируется обратно в исходный массив.

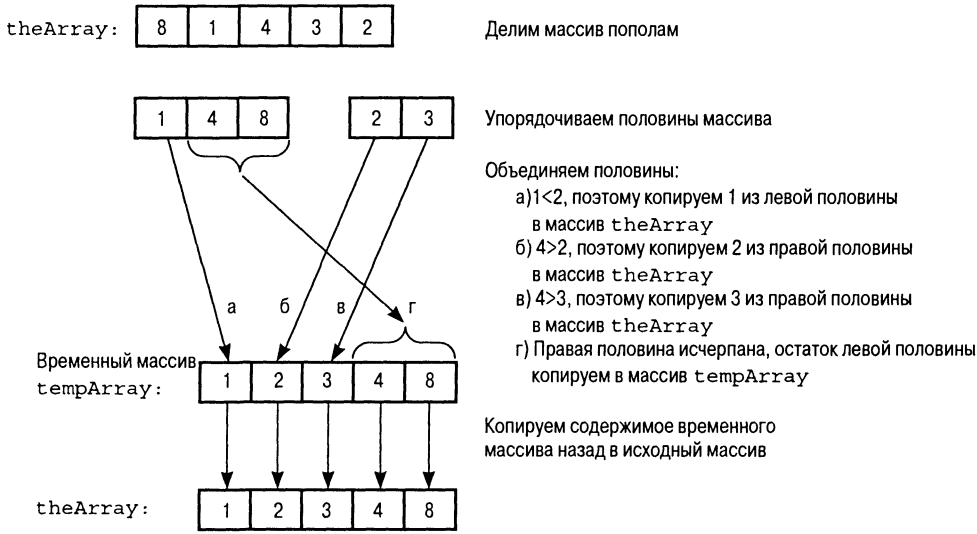


Рис. 9.8. Сортировка методом слияния с помощью вспомогательного массива

Хотя в результате слияния возникает упорядоченный массив, остается неясным, как выполняется сортировка на предыдущих этапах. Сортировка слиянием выполняется рекурсивно. Ее псевдокод имеет следующий вид.

```
mergesort(inout theArray:ItemArray,
          in first:integer, in last:integer)
// Упорядочивает отрезок theArray[first..last],
// 1) сортируя первую половину массива;
// 2) сортируя вторую половину массива;
// 3) объединяя две упорядоченные половины массива.

   if (first < last)
   {
      mid = (first + last)/2 // Определяем середину
     // Сортируем отрезок theArray[first..mid]
     mergesort(theArray, first, mid)
     // Сортируем отрезок theArray[mid+1..last]
     mergesort(theArray, mid + 1, last)
     // Объединяем упорядоченные отрезки theArray[first..mid]
     // и theArray[mid+1..last]
     merge(theArray, first, mid, last)
   } // Конец оператора if
// Если first >= last, операции завершены
```

Совершенно ясно, что основные операции этого алгоритма выполняются на этапе слияния, и все же, почему в результате возникает упорядоченный массив? Рекурсивные вызовы продолжают разделять части массива пополам, пока в них не останется только по одному элементу. Очевидно, что массив, состоящий из одного элемента, является упорядоченным. Затем алгоритм объединяет фрагменты массива, пока не образуется один упорядоченный массив. Рекурсивные вызовы функции *mergesort* и результаты слияний проиллюстрированы на рис. 9.9 на примере массива, состоящего из шести целых чисел.

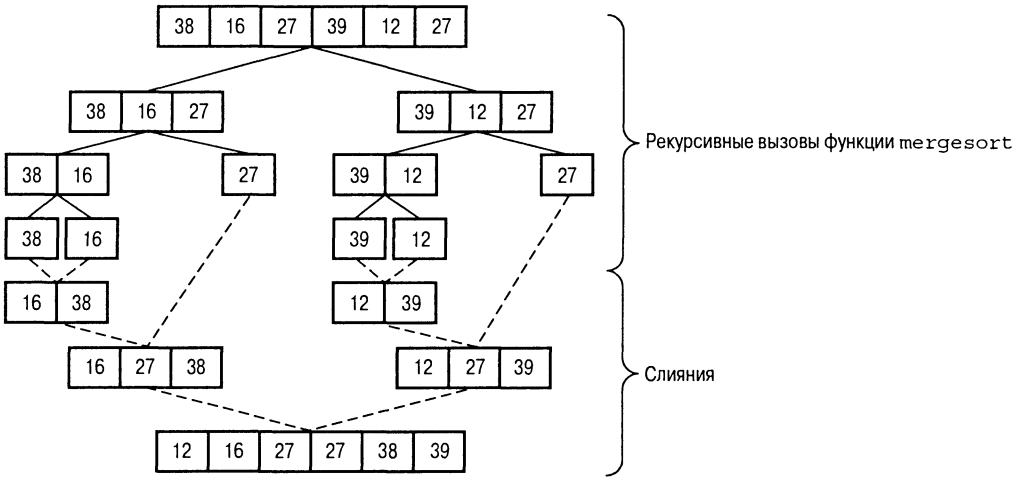


Рис. 9.9. Сортировка методом слияния массива, состоящего из шести целых чисел

Ниже приведена функция на языке C++, реализующая алгоритм сортировки методом слияний. Для того чтобы упорядочить массив *theArray*, состоящий из *n* элементов, выполняется рекурсивный вызов *mergesort(theArray, 0, n-1)*.

```
const int MAX_SIZE = максимальное-количество-элементов-массива;
void merge(DataType theArray[], int first, int mid, int last)
// ----------------------------------------------------------
// Объединяет два упорядоченных отрезка theArray[first..mid] и //
theArray[mid+1..last] в один  упорядоченный массив.
// Предусловие: first <= mid <= last. Оба подмассива
// theArray[first..mid] и theArray[mid+1..last] упорядочены
// по возрастанию.
// Постусловие: отрезок theArray[first..last] упорядочен.
// Замечание о реализации: функция выполняет слияние двух
// подмассивов во временный массив, а затем копирует его
// содержимое в исходный массив theArray.
// ----------------------------------------------------------
{
   DataType tempArray[MAX_SIZE]; // Временный массив

   // Инициализируем локальные индексы, выделяя подмассивы
   int first1 = first;    // Начало первого подмассива
   int last1 = mid;       // Конец первого подмассива
   int first2 = mid + 1; // Начало второго подмассива
   int last2 = last;      // Конец второго подмассива

   // Пока оба подмассива не пусты, копируем меньший элемент
   // во временный массив
   int index = first1;    // Следующая свободная ячейка
                          // массива tempArray
   for (; (first1 <= last1) && (first2 <= last2); ++index)
   {
      // Инвариант: отрезок tempArray[first1..index-1]
      // упорядочен.
      if(theArray[first1] < theArray[first2])
      {
```

```
            tempArray[index] = theArray[first1];
            ++first1;
         }
         else
         {
            tempArray[index] = theArray[first2];
            ++first2;
         } // Конец оператора if
      } // Конец оператора for

      // Скопировать подмассив, оставшийся непустым

      // Скопировать первый подмассив, если необходимо
      for (; first1 <= last1; ++first1, ++index)
         // Инвариант: отрезок tempArray[first1..index-1]
         // упорядочен.
         tempArray[index] = theArray[first1];

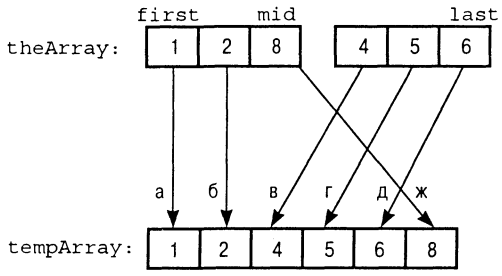
      //Скопировать второй подмассив, если необходимо
      for (; first2 <= last2; ++first2, ++index)
         // Инвариант: отрезок tempArray[first1..index-1]
         // упорядочен
         tempArray[index] = theArray[first2];

      // Копируем результат обратно в исходный массив
      for (index = first; index <= last; ++index)
         theArray[index] = tempArray[index];
   } // Конец функции merge

   void mergesort(DataType theArray[], int first, int last)
   // ----------------------------------------------------------
   // Упорядочивает элементы массива в возрастающем порядке.
   // Предусловие: отрезок theArray[first..last] — массив.
   // Постусловие: массив theArray[first..last] упорядочен
   // по возрастанию.
   // Вызываемые функции: merge.
   // ----------------------------------------------------------
   {
      if (first < last)
      {
         // Упорядочиваем каждую из частей массива
         int mid = (first + last)/2; // Индекс среднего элемента
         // Упорядочиваем левую половину theArray[first..mid]
         mergesort(theArray, first, mid);
         // Упорядочиваем правую половину theArray[mid+1..last]
         mergesort(theArray, mid+1, last);
         // Объединяем две половины
         merge(theArray, first, mid, last);
      } // Конец оператора if
   } // Конец функции mergesort
```

Анализ. Поскольку основные операции в этом алгоритме выполняются на этапе слияния, начнем анализ с него. На каждом шаге происходит объединение подмассивов *theArray[first...mid]* и *theArray[mid+1...last]*. На рис. 9.10 показан пример, в котором требуется выполнить максимальное количество сравнений. Если общее количество элементов объединяемых отрезков массива равно n, то при их слиянии потребуется выполнить n-1 сравнений. (Например, на

рис. 9.10 показан массив, состоящий из шести элементов, следовательно, выполняется пять сравнений.) Кроме того, после сравнений осуществляется копирование n элементов временного массива в исходный. Таким образом, на каждом шаге слияния выполняется $3*n-1$ основных операций.



Слияние половин:
а) 1<4, поэтому копируем 1 из подмассива `theArray[first..mid]` в массив `tempArray`
б) 2<4, поэтому копируем 2 из подмассива `theArray[first..mid]` в массив `tempArray`
в) 8>4, поэтому копируем 4 из подмассива `theArray[mid+1..last]` в массив `tempArray`
г) 8>5, поэтому копируем 5 из подмассива `theArray[mid+1..last]` в массив `tempArray`
д) 8>6, поэтому копируем 6 из подмассива `theArray[mid+1..last]` в массив `tempArray`
ж) Подмассив `theArray[mid+1..last]` исчерпан, поэтому копируем 8 в массив `tempArray`

Рис. 9.10. Наихудший случай на этапе слияния

В функции *mergesort* выполняются два рекурсивных вызова. Как показано на рис. 9.11, если исходный вызов функции *mergesort* принадлежит нулевому уровню, то на первом уровне возникают два рекурсивных вызова. Затем каждый из этих вызовов порождает еще два рекурсивных вызова второго уровня и т.д. Сколько уровней рекурсии возникнет? Попробуем их подсчитать.

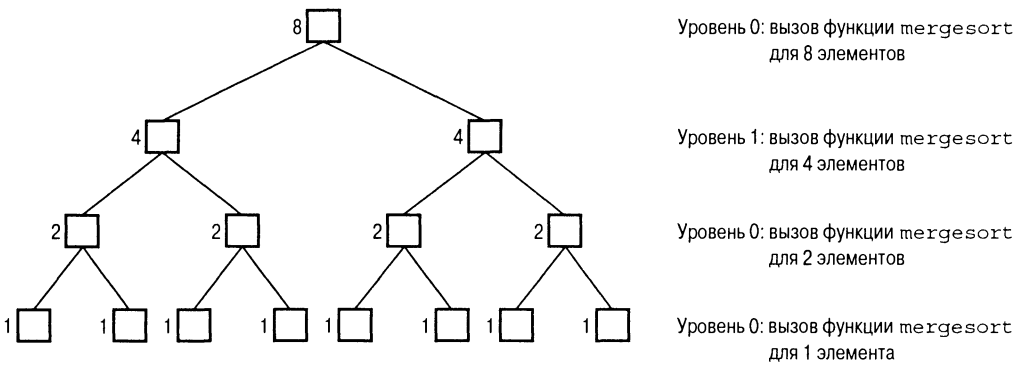


Рис. 9.11. Уровни рекурсивных вызовов функции mergesort при сортировке массива, состоящего из восьми элементов

Каждый вызов функции *mergesort* делит массив пополам. На первом этапе исходный массив оказывается разделенным на две части. При следующем рекурсивном вызове функции *mergesort* каждая из этих частей снова делится пополам, образуя четыре части исходного массива. При следующем рекурсивном вызове каждая из этих четырех частей опять делится пополам, образуя восемь частей массива, и т.д. Рекурсивные вызовы продолжаются до тех пор, пока части массива не станут содержать только по одному элементу, иными словами, пока исходный массив не будет разбит на n частей, что соответствует количеству его элементов, если число n является степенью двойки ($n=2^k$), глубина рекурсии равна $k=\log_2 n$. Например, как показано на рис. 9.11, если исходный массив содержит восемь эле-

ментов ($8=2^3$), то глубина рекурсии равна 3. Если число n является степенью двойки, глубина рекурсии равна $1+\log_2 n$ (округленное значение).

Сложность алгоритма сортировки слиянием равна $O(n*\log_2 n)$

Исходный вызов функции *mergesort* (уровень 0) обращается к функции *merge* только один раз. Затем функция *merge* осуществляет слияние n элементов, выполняя $3*n-1$ операций. На первом уровне рекурсии выполняются два вызова функции *mergesort* и, следовательно, функции *merge*. Каждый из этих двух вызовов приводит к слиянию $n/2$ элементов и требует выполнения $3*(n/2)-1$ операций. Таким образом, на этом уровне выполняется $2*(3*(n/2)-1)=3*n-2$ операций. На m-м уровне рекурсии выполняются 2^m вызовов функции *merge*. Каждый из этих вызовов приводит к слиянию $n/2^m$ элементов, а общее количество операций равно $3*(n/2^m)-2$. В целом, 2^m рекурсивных вызова функции *merge* порождает $3*n-2^m$ операций. Таким образом, на каждом уровне рекурсии выполняется $O(n)$ операций. Поскольку количество уровней рекурсии равно $\log_2 n$ или $\log_2 n+1$, в наихудшем и среднем вариантах функция *mergesort* имеет сложность $O(n*\log_2 n)$. Посмотрите на рис. 9.3 и еще раз убедитесь, что величина $O(n*\log_2 n)$ растет намного быстрее, чем величина $O(n^2)$.

Для сортировки слиянием необходим вспомогательный массив, размер которого совпадает с размером исходного массива

Хотя алгоритм сортировки слиянием имеет чрезвычайно высокое быстродействие, у него есть один недостаток. Для выполнения операции

```
Объединить упорядоченные подмассивы
theArray[first...mid] и
theArray[mid+1...last]
```

необходим вспомогательный массив, состоящий из n элементов. Если объем доступной памяти ограничен, это требование может оказаться неприемлемым.

Быстрая сортировка

Еще один алгоритм, основанный на принципе "разделяй и властвуй"

Рассмотрим два первых шага решения задачи о поиске k-го наименьшего элемента массива *theArray[fist...last]*, описанного в главе 2.

```
Выберите в массиве theArray[first...last]опорный элемент p
Поделите массив theArray[first...last] относительно элемента p
```

Алгоритм быстрой сортировки разбивает массив на две части: в первую часть входят элементы меньше опорного, а во вторую — остальные

Разбиение, показанное на рис. 9.12, характеризуется тем, что все элементы множества S_1=*theArray[first...pivotIndex-1]* меньше опорного элемента p, а множество S_2=*theArray[pivotIndex+1...last]* состоит из элементов, больших или равных опорному. Хотя из этого свойства не следует, что массив упорядочен, из него вытекает чрезвычайно полезный факт: если массив упорядочен, элементы, стоящие на позициях от *first* до *pivotIndex-1*, остаются на своих местах, хотя их позиции относительно друг друга могут измениться. Аналогичное утверждение выполняется и для элементов, стоящих на позициях от *pivotIndex+1* до *last*. Опорный элемент в полученном упорядоченном массиве останется на своем месте.

В результате быстрой сортировки опорный элемент останется на своем месте

Такое разбиение массива определяет рекурсивный характер алгоритма. Разбиение массива относительно опорного элемента p порождает две задачи сортировки меньшего размера — сортировка левой (S_1) и правой (S_2) частей массива. Решив эти две задачи, мы полу-

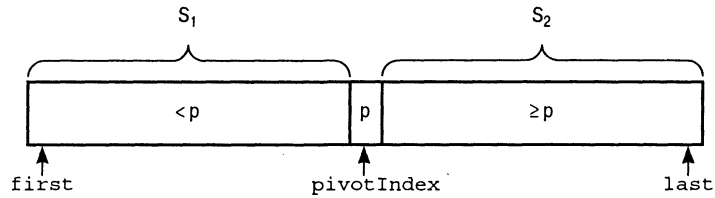


Рис. 9.12. Разбиение относительно опорного элемента

чим решение исходной задачи. Иными словами, разбиение массива перед рекурсивными вызовами оставляет опорный элемент на своем месте и гарантирует, что левый и правый отрезки массива окажутся упорядоченными. Кроме того, алгоритм быстрой сортировки конечен: размеры левого и правого отрезка массива меньше размера исходного массива, причем каждый шаг рекурсии приближает нас к базису, когда массив состоит из одного элемента. Это следует из того факта, что опорный элемент p не принадлежит ни одному из массивов S_1 и S_2.

Псевдокод алгоритма быстрой сортировки выглядит следующим образом.

```
quicksort(inout theArray:ItemArray,
          in first:integer, in last:integer)
// Упорядочивает массив theArray[first..last]

   if (first < last)
      Выбрать опорный элемент p из массива theArray[first..last]
      Разбить массив theArray[first..last] относительно
         опорного элемента p
      // Разбиение имеет вид theArray[first..pivotIndex..last]

      // Упорядочиваем массив S1
      quicksort(theArray, first, pivotIndex-1)
      // Упорядочиваем массив S2
      quicksort(theArray, pivotIndex+1, last)
   } // Конец оператора if
   // если first >= last, ничего не делаем
```

Имеет смысл сравнить функцию *quickSort* с псевдокодом функции, предназначенной для поиска k-го наименьшего элемента, описанной в главе 2.

```
kSmall(in k:integer, in theArray:ItemArray,
       in first:integer, in last:integer):ItemType
// Возвращает значение k-го наименьшего элемента массива
// theArray[first..last].

  Выбрать в массиве theArray[first..last] опорный элемент p
  Разбить массив theArray[first..last] относительно элемента p
   if (k < pivotIndex - first + 1)
      return kSmall(k, theArray, first, pivotIndex-1)
   else if (k == pivotIndex - first + 1)
      return p
   else
      return kSmall(k-(pivotIndex-first+1), theArray,
                    pivotIndex+1, last)
```

Различия между функциями kSmall и quickSort

Функция *kSmall* вызывается рекурсивно, только если одна из частей массива содержит искомый элемент. Если этим элементом является опорный, функция не вызывается вообще. В то же время функция *quickSort* вызывается рекурсивно для обеих частей массива. Различия между этими двумя функциями проиллюстрировано на рис. 9.13.

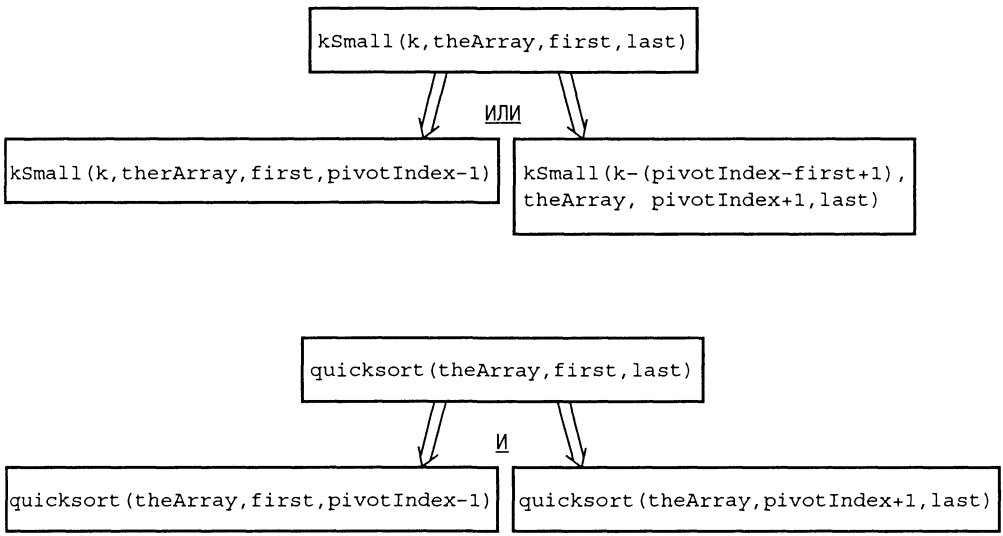


Рис. 9.13. Сравнение функций kSmall и quickSort

Использование инварианта в алгоритме разбиения. Рассмотрим функцию разбиения массива, которая должна вызываться функциями *kSmall* и *quickSort*. В обоих алгоритмах именно разбиение массива представляет собой наиболее трудную задачу.

Функция, предназначенная для разбиения массива, получает в качестве аргумента отрезок *theArray[first..last]*. Функция должна распределить элементы массива, руководствуясь следующим правилом: в множество S_1 включаются элементы, меньшие опорного, а в множество S_2 — остальные. Как показано на рис. 9.12, множество S_1 является отрезком массива *theArray[first..pivotIndex-1]*, а множество S_2 — отрезком массива *theArray[pivotIndex+1..last]*.

Перед разбиением поместите выбранный опорный элемент в ячейку theArray[first]

Как выбрать опорный элемент? Если элементы массива записаны в произвольном порядке, в качестве опорного можно выбрать любой элемент, например *theArray[first]*. (Более детально процедура выбора опорного элемента будет рассмотрена позднее.) При разбиении массива опорный элемент удобно помещать в ячейку *theArray[first]*, независимо от того, какой именно элемент выбран в качестве опорного.

Часть массива, в которой находятся элементы, еще не распределенные по отрезкам S_1 и S_2, называется неопределенной. Итак, рассмотрим массив, изображенный на рис. 9.14. Индексы *first*, *lastS1*, *firstUnknown* и *last* разделяют массив на три части. Отношения между опорным элементом и элементами неопределенной части *theArray[firstUnknown..last]* неизвестны.

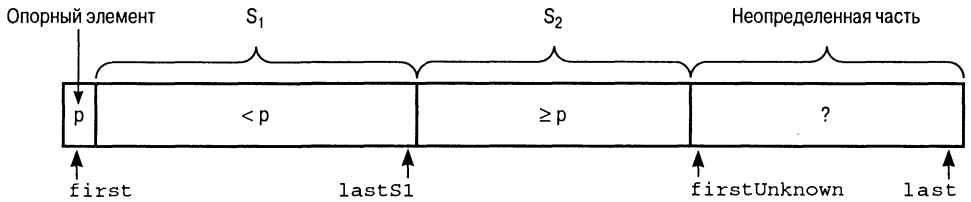


Рис. 9.14. Инвариант алгоритма разбиения

В процессе разбиения массива должно выполняться следующее условие.

Инвариант алгоритма разбиения

Элементы множества S_1 должны быть меньше опорного элемента, а элементы множества S_2 — больше или равны ему.

Это утверждение является инвариантом алгоритма разбиения. Для того чтобы в начале алгоритма выполнялся его инвариант, необходимо проинициализировать индексы массива так, чтобы весь массив, кроме опорного элемента, считался неопределенным.

```
lastS1 = first
firstUnknown = first + 1
```

В исходном положении все элементы, за исключением элемента theArray[first], относятся к неопределенной части

Исходное состояние массива изображено на рис. 9.15.

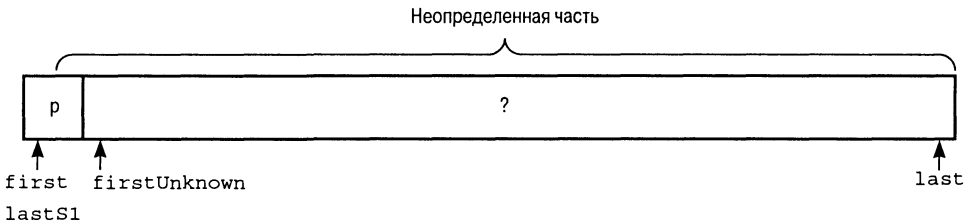


Рис. 9.15. Исходное состояние массива

На каждом шаге алгоритма разбиения проверяется один элемент из неопределенной части. В зависимости от его значения он помещается в множество S_1 или S_2. Таким образом, на каждом шаге размер неопределенной части уменьшается на единицу. Алгоритм останавливается, когда размер неопределенной части становится равным нулю, т.е. выполняется условие *firstUnknown* > *last*.

Рассмотрим псевдокод этого алгоритма.

```
partition(inout theArray:ItemArray,
          in first:integer, in last:integer,
          out pivotIndex:integer)
// Разделяет массив theArray[first..last]

   // Инициализация
   Выбрать опорный элемент и поменять его местами
      с элементом theArray[first]
   p = theArray[first]      // p — опорный элемент

   // Задаем пустые множества S1 и S2, а неопределенную
   // часть массива инициализируем отрезком
```

```
// theArray[first+1..last]
lastS1 = first
firstUnknown = first + 1

// Определяем множества S1 и S2
while (firstUnknown <= last)
{
   // Вычисляем индекс самого левого элемента
   // неопределенной части массива
   if (theArray[firstUnknown] < p)
      Поместить элемент theArray[firstUnknown] в S1
   else
      Поместить элемент theArray[firstUnknown] в S2
} // Конец оператора while

// Ставим опорный элемент между множествами S1 и S2
// и запоминаем его новый индекс
Поменять местами theArray[first] и theArray[lastS1]
pivotIndex = lastS1
```

Алгоритм достаточно прост, но операция перемещения требует разъяснения. Рассмотрим два возможных действия, которые необходимо выполнить на каждой итерации цикла *while*.

- **Поместить элемент *theArray[firstUnknown]* в множество S_1.** Множество S_1 и неопределенная часть, как правило, не являются смежными. Обычно между ними располагается множество S_2. Однако эту операцию можно выполнить более эффективно. Элемент *theArray[firstUnknown]* можно поменять местами с первым элементом множества S_2, т.е. с элементом *theArray[lastS1+1]*, как показано на рис. 9.16. Как быть с элементом множества S_2, который был помещен в ячейку *theArray[firstUnknown]*? Если увеличить индекс *firstUnknown* на единицу, этот элемент становится самым правым в множестве S_2. Таким образом, для переноса элемента *theArray[firstUnknown]* в массив S_1 необходимо выполнить следующие шаги.

```
Поменять местами элементы theArray[firstInknown]
   и theArray[lastS1+1]
Увеличить индекс lastS1 на единицу
Увеличить индекс firstUnknown на единицу
```

- Эта стратегия остается верной, даже если множество S_2 пусто. В этом случае величина *lastS1+1* равна индексу *firstUnknown*, и элемент просто остается на своем месте. *Инвариант при этом не нарушается.*
- **Поместить элемент *theArray[firstUnknown]* в множество S_2.** Эту операцию легко выполнить. Напомним, что индекс крайнего правого элемента множества S_2 равен *firstUnknown-1*, т.е. множество S_2 и неизвестная часть являются смежными (рис. 9.17). Таким образом, чтобы переместить элемент *theArray[firstUnknown]* в множество S_2, нужно просто увеличить индекс *firstUnknown* на единицу, расширяя множество S_2 вправо. *Инвариант при этом не нарушается.*

После переноса всех элементов из неопределенной части в множества S_1 и S_2 остается решить последнюю задачу. Нужно поместить опорный элемент между множествами S_1 и S_2. Обратите внимание, что элемент *theArray[lastS1]* явля-

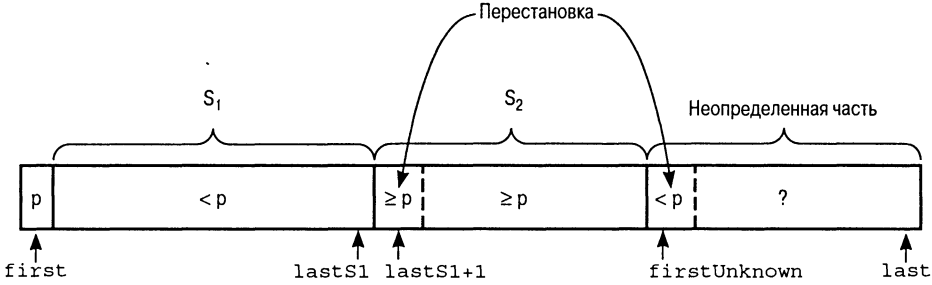


Рис. 9.16. Перенос элемента theArray[firstUnknown] в множество S_1 путем перестановки с элементом theArray[lastS1+1] с последующим увеличением индексов lastS1 и firstUnknown на единицу

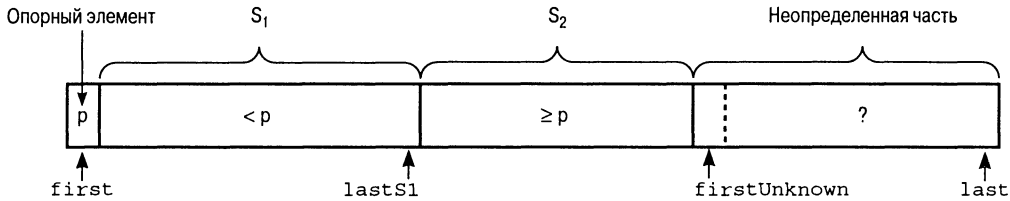


Рис. 9.17. Перенос элемента theArray[firstUnknown] в множество S_2 после увеличения индекса firstUnknown на единицу

ется крайним правым элементом множества S_1. Если поменять его местами с опорным элементом, тот станет на правильное место. Следовательно, оператор

```
pivotIndex = lastS1
```

позволяет определить индекс опорного элемента. Этот индекс можно использовать в качестве границы между множествами S_1 и S_2. Результаты трассировки алгоритма разбиения массива, состоящего из шести целых чисел, когда опорным является первый элемент, показаны на рис. 9.18.

Прежде чем приступить к реализации алгоритма быстрой сортировки, проверим корректность алгоритма разбиения, используя его инварианты. Инвариант цикла, входящего в алгоритм, имеет следующий вид.

Все элементы множества S_1 (theArray[first+1..lastS1]) меньше опорного, а все элементы множества S_2 (theArray[lastS1..firstUnknown-1]) больше или равны опорному

Напомним, что для определения правильности алгоритма с помощью его инвариантов, необходимо выполнить четыре шага.

1. Инвариант должен быть истинным с самого начала, до выполнения цикла. В алгоритме разбиения опорным элементом является *theArray[first]*, неизвестной частью — отрезок массива *theArray[first+1..last]*, а множества S_1 и S_2 пусты. Очевидно, что при этих условиях инвариант является истинным.
2. Итерации цикла не должны нарушать инвариант. Иными словами, если инвариант был истинным перед определенной итерацией цикла, он должен оставаться истинным и после ее выполнения. В алгоритме разбиения каж-

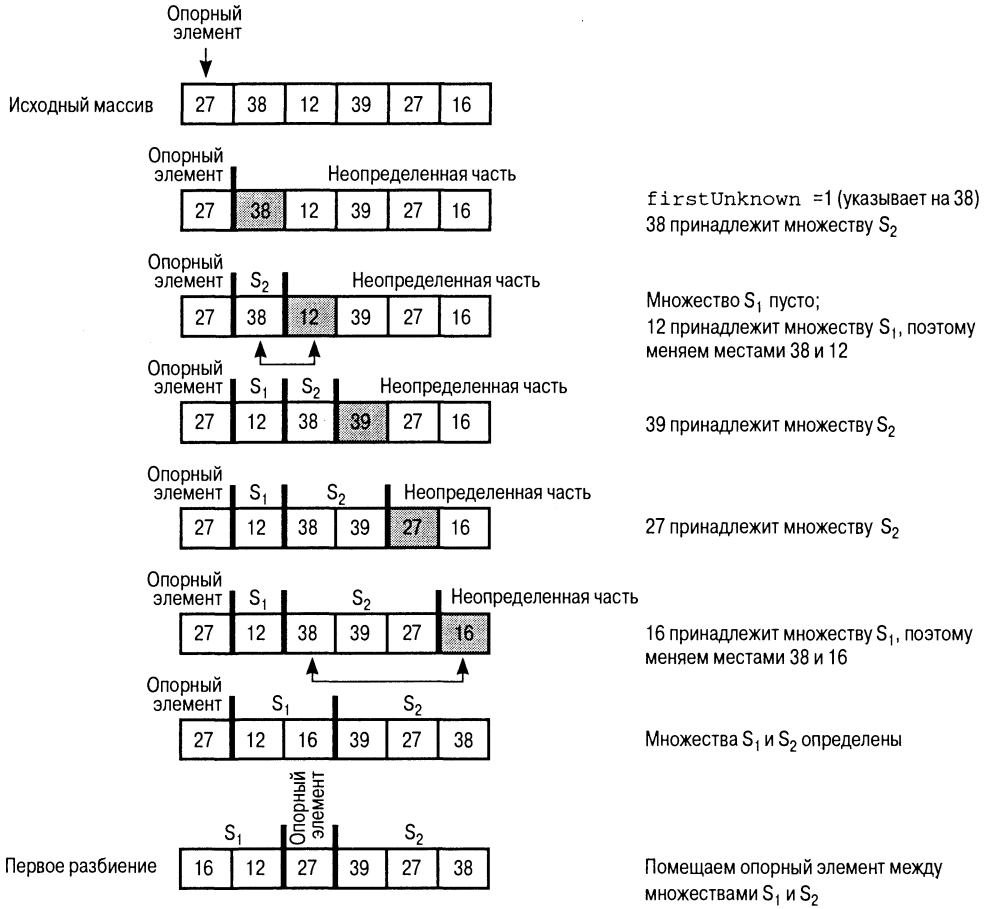


Рис. 9.18. Первое разбиение массива, когда опорным является первый элемент

дая итерация цикла переносит один элемент из неизвестной части в множество S_1 или S_2, в зависимости от его значения по сравнению с опорным. Итак, если до переноса инвариант был истинным, он должен сохраняться и после переноса.

3. Инвариант должен определять корректность алгоритма. Иными словами, из истинности инварианта должна следовать корректность алгоритма. Выполнение алгоритма разбиения прекращается, когда неопределенная область становится пустой. В этом случае каждый элемент отрезка `theArray[first+1..last]` должен принадлежать либо множеству S_1, либо множеству S_2. В любом случае из корректности инварианта следует, что алгоритм достиг своей цели.

4. Цикл должен быть конечным. Иными словами, нужно показать, что выполнение цикла завершится после конечного числа итераций. В алгоритме разбиения размер неопределенной части на каждой итерации уменьшается на единицу. Следовательно, после выполнения конечного количества итераций неопределенная часть становится пустой, и цикл завершается.

Рассмотрим функцию на языке C++, реализующую алгоритм *quickSort*. В ней для выбора опорного элемента используется функция *choosePivot*, а функция *swap* работает, как и раньше, в алгоритме *selectionSort*. Для упорядочения массива *theArray*, состоящего из *n* элементов, выполняется вызов *quickSort(theArray, 0, n-1)*.

```
void choosePivot(DataType theArray[], int first, int last);
// ----------------------------------------------------------
// Выбирает опорный элемент для алгоритма быстрой сортировки.
// Меняет его местами с первым элементом массива.
// Предусловие: отрезок theArray[first..last] — массив;
// first <= last.
// Постусловие: элемент theArray[first] является опорным.
// ----------------------------------------------------------
// Реализация этой функции предоставляется читателям.

void partition(DataType theArray[],
               int first, int last, int& pivotIndex)
// ----------------------------------------------------------
// Разбивает массив для быстрой сортировки.
// Предусловие: отрезок theArray[first..last] — массив;
// first <= last.
// Постусловие: массив theArray[first..last] разбит
// следующим образом:
// S1 = theArray[first..pivotIndex-1] < pivot
// theArray[pivotIndex] == pivot
// S2 = theArray[pivotIndex+1..last] >= pivot
// Вызываемые функции: choosePivot и swap.
// ----------------------------------------------------------
{
   // Помещаем опорный элемент в ячейку theArray[first]
   choosePivot(theArray, first, last);

    DataType pivot = theArray[first]; // Копируем
                                      // опорный элемент

   // В исходном положении все элементы, кроме опорного,
   // принадлежат неопределенной части массива
   int lastS1 = first; // Индекс последнего элемента
                       // множества S1
   int firstUnknown = first + 1; // Индекс первого элемента
                                 // неопределенной части

   // Переносим элементы один за другим,
   // пока неопределенная часть массива не станет пустой
   for (; firstUnknown <= last; ++firstUnknown)
   {
       // Инвариант: theArray[first+1..lastS1] < pivot
       // theArray[lastS1+1..firstUnknown-1] >= pivot

       // Переносим элемент из неопределенной части
       // в множество S1 или S2
       if (theArray[firstUnknown] < pivot)
       {
          // Элемент принадлежит множеству S1
          ++lastS1;
          swap(theArray[firstUnknown], theArray[lastS1]);
       } // Конец оператора if
```

```
            // Иначе элемент принадлежит множеству S2
        } // Конец оператора for

        // Поставить опорный элемент на соответствующее место
        // и запомнить его индекс
        swap(theArray[first], theArray[lastS1]);
        pivotIndex = lastS1;
    } // Конец функции partition

    void quicksort(DataType theArray[], int first, int last)
    // ----------------------------------------------------------
    // Упорядочивает элементы массива в возрастающем порядке.
    // Предусловие: отрезок theArray[first..last] - массив.
    // Постусловие: массив theArray[first..last] упорядочен.
    // Вызываемая функция: partition.
    // ----------------------------------------------------------
    {
        int pivotIndex;
        if (first < last)
        {
            // Создаем разбиение: S1, опорный элемент, S2
            partition(theArray, first, last, pivotIndex);

            // Упорядочиваем множества S1 и S2
            quicksort(theArray, first, pivotIndex-1);
            quicksort(theArray, pivotIndex+1, last);
        } // Конец оператора if
    } // Конец функции quicksort
```

Дальнейший анализ покажет, что желательно избегать такого выбора опорного элемента, при котором множество S_1 или S_2 оказываются пустыми. Лучше всего выбирать опорный элемент поближе к медиане массива. Этот выбор опорного элемента рассматривается в упражнении 17.

Алгоритмы *quickSort* и *mergeSort* "близки по духу", хотя в алгоритме *quickSort* основная работа выполняется до рекурсивных вызовов, а в алгоритме *mergseSort* — после.

Схему псевдокода алгоритма *quickSort* можно записать так.

```
quicksort(inout theArray:ItemArray, in first:integer, in last:integer)

    if (first < last)
    {
        Подготовить массив theArray к рекурсивным вызовам
        quicksort(отрезок S1 массива theArray)
        quicksort(отрезок S2 массива theArray)
    } // Конец оператора if
```

В то же время общая схема алгоритма *mergeSort* выглядит следующим образом.

```
mergesort(inout theArray:ItemArray, in first:integer, in last:integer)

    if (first < last)
    {
        mergesort(левая часть массива theArray)
        mergesort(правая часть массива theArray)
        Собрать массив, полученный после рекурсивных вызовов
    } // Конец оператора if
```

Перед вызовом функции *quickSort* выполняется разбиение массива на части S_1 и S_2. Затем алгоритм упорядочивает отрезки S_1 и S_2 независимо друг от друга, поскольку любой элемент отрезка S_1 находится левее любого элемента отрезка S_2. В функции *mergeSort*, наоборот, перед рекурсивными вызовами никакая работа не выполняется. Алгоритм упорядочивает каждую из частей массива, постоянно учитывая отношения между элементами обеих частей. По этой причине алгоритм должен объединять две половины массива после выполнения рекурсивных вызовов.

Анализ. Основная работа в алгоритме *quickSort* выполняется на этапе разбиения массива. Анализируя каждый элемент, принадлежащий неопределенной части, необходимо сравнивать элемент *theArray[firstunknown]* с опорным и помещать его либо в отрезок S_1, либо в отрезок S_2. Один из отрезков S_1 или S_2 может быть пустым; например, если опорным элементом является наименьший элемент отрезка, множество S_1 останется пустым. Это происходит в наихудшем случае, поскольку размер отрезка S_2 при каждом вызове функции *quickSort* уменьшается только на единицу. Таким образом, в этой ситуации будет выполнено максимальное количество рекурсивных вызовов функции *quickSort*.

Посмотрим, что произойдет, если исходный массив окажется заранее упорядоченным, а в качестве опорного выбран наименьший элемент. На рис. 9.19 показаны результаты первого вызова функции *partition*. Опорным является наименьший элемент массива, и множество S_1 остается пустым. При разбиении массива, состоящего из n элементов, функция *partition* выполняет n-1 сравнение.

Если исходный массив упорядочен и в качестве опорного выбран наименьший элемент, алгоритм quickSort работает медленно

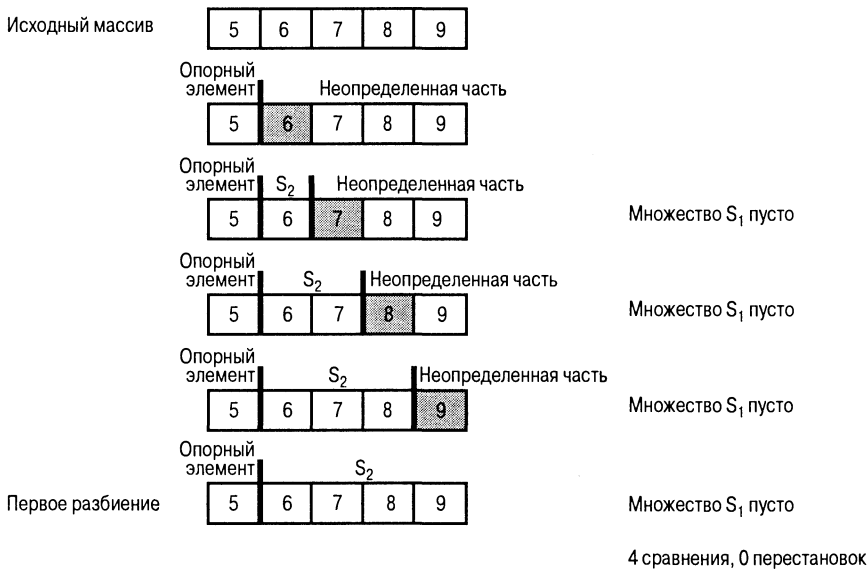


Рис. 9.19. Разбиение массива в алгоритме quickSort в наихудшем варианте

При следующем рекурсивном вызове функции *quickSort* функция *partition* просмотрит n-1 элемент. Чтобы распределить их по отрезкам, понадобится n-2 сравнений. Поскольку размер отрезка, рассматриваемого функцией *quickSort*,

на каждом уровне рекурсии уменьшается только на единицу, возникнет n-1 уровней рекурсии. Следовательно, функция `quickSort` выполняет

$$1 + 2 + \ldots + (n-1) = n * (n-1)/2$$

сравнений. Напомним, однако, что при переносе элемента в множество S_2 выполнять перестановку элементов не обязательно. Для этого достаточно лишь изменить индекс `firstUnknown`.

Аналогично, если множество S_2 при каждом рекурсивном вызове остается пустым, потребуется $n * (n-1)/2$ сравнений. Кроме того, в этом случае для переноса каждого элемента из неизвестной части в множество S_1 придется выполнять перестановку элементов. Таким образом, понадобится $n * (n-1)/2$ перестановок. (Напомним, что каждая перестановка выполняется с помощью трех операций присваивания.) Итак, в худшем случае сложность алгоритма `quickSort` равна $O(n^2)$.

Для контраста на рис. 9.20 продемонстрирован пример, когда множества S_1 и S_2 состоят из одинакового количества элементов. В среднем случае, когда множества S_1 и S_2 состоят из одинакового — или приблизительно одинакового — количества элементов, записанных в произвольном порядке, рекурсивных вызовов функции `quickSort` потребуется меньше. Как и при анализе алгоритма `mergeSort`, легко показать, что глубина рекурсии в алгоритме `quickSort` равна $\log_2 n$ или $\log_2 n+1$. При каждом вызове функции `quickSort` выполняется m сравнений и не больше, чем m перестановок, где m — количество элементов в подмассиве, подлежащем сортировке.

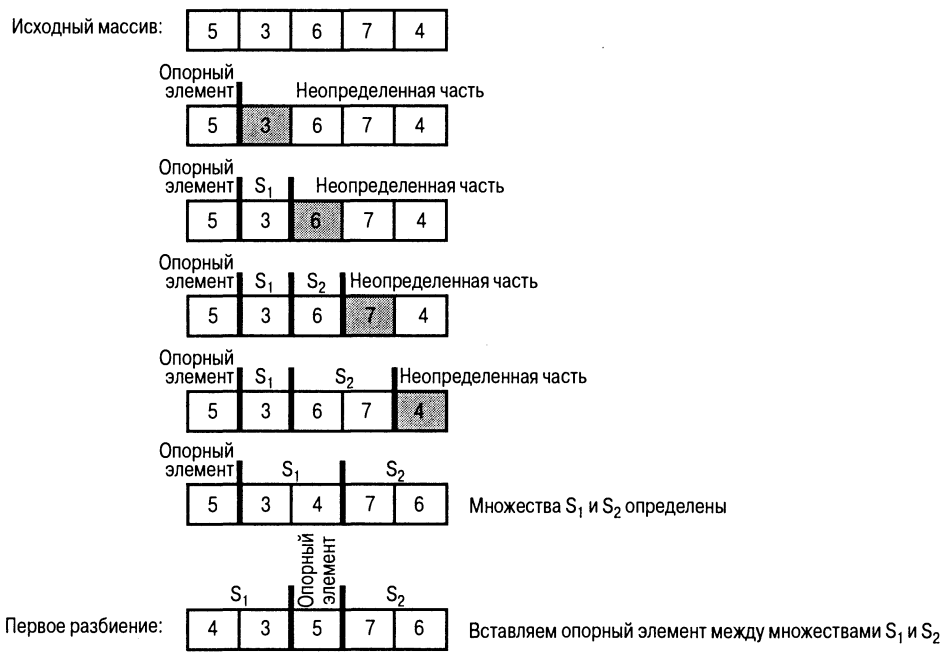


Рис. 9.20. Разбиение массива в алгоритме quickSort в среднем варианте

Формальный анализ алгоритма `quickSort` для среднего варианта показывает, что его сложность является величиной $O(n*\log n)$. Таким образом, с большими массивами алгоритм

Быстрая сортировка: наихудший вариант $O(n^2)$, средний вариант $O(n*\log n)$

quickSort работает значительно быстрее, чем алгоритм *insertionSort*, хотя в наихудшем варианте они оба имеют приблизительно одинаковое быстродействие.

Алгоритм *quickSort* часто используется для сортировки больших массивов. Причина его популярности заключается в исключительном быстродействии, несмотря на обескураживающие оценки наихудшего варианта. Дело в том, что этот вариант встречается крайне редко, и на практике алгоритм *quickSort* отлично работает с относительно большими массивами.

Значительное различие между оценками сложности в среднем и наихудшем вариантах выделяет алгоритм быстрой сортировки среди остальных алгоритмов, рассмотренных в данной главе. Если порядок записи элементов в исходном массиве является "случайным", алгоритм *quickSort* работает по крайней мере не хуже любого другого алгоритма, использующего сравнения элементов. Если исходный массив совершенно не упорядочен, алгоритм *quickSort* работает лучше всех.

Алгоритм *mergeSort* имеет приблизительно такую же эффективность. В некоторых случаях быстрее работает алгоритм *quickSort*, в других — алгоритм *mergeSort*. Несмотря на то что оценка сложности алгоритма *mergeSort* в наихудшем варианте имеет тот же порядок, что и оценка сложности алгоритма *quickSort* в среднем варианте, в большинстве случаев алгоритм *quickSort* работает несколько быстрее. Однако в наихудшем варианте быстродействие алгоритма *quickSort* намного ниже.

Поразрядная сортировка

Последний алгоритм сортировки, рассматриваемый в данной главе, совершенно отличается от остальных.

Вспомните, когда вы в последний раз тасовали карты. Возьмите колоду и вытаскивайте карты одну за другой, располагая их по группам: 2, 3, ..., 10, J, Q, K, A. Таким образом, получится 13 групп. Объедините эти группы и положите карты лицом вверх, так чтобы двойки были наверху, в тузы — внизу. Теперь берите карты по одной и раскладывайте их на четыре группы в соответствии с мастью: трефы, бубны, черви и пики. Собрав их вместе, вы получите упорядоченную колоду карт.

Поразрядная сортировка (radix sort) использует эту идею для образования групп с их последующим объединением в упорядоченный набор данных. В процессе сортировки каждый элемент рассматривается как строка символов. В качестве первого примера поразрядной сортировки рассмотрим набор строк, состоящих из трех символов.

ABC, XYZ, BWZ, AAC, RLT, JBX, RDT, KLT, AEO, TIJ

Группы строк, упорядоченные по последнему символу

Сортировка начинается с упорядочения данных по последнему символу (наименее значимому). Символами A и B не заканчивается ни одна строка, а символом C заканчиваются две строки. Поместим эти две строки в отдельную группу. Продолжая упорядочение по алфавиту, получим следующие группы.

(ABC, AAC) (TLJ) (AEO) (RLT, RDT, KLT) (JBX) (XYZ, BWZ)

Строки, объединенные в одну группу, заканчиваются одним и тем же символом, а сами группы упорядочены по этому символу. Строки внутри группы записываются в том же порядке, в котором они были указаны в исходном списке.

Объединенная группа

Теперь объединим эти группы в одну. Для этого возьмем строки из первой группы, не меняя их порядка, запишем после них строки из второй группы и т.д., получится объединенная группа.

ABC, AAC, TLJ, AEO, RLT, RDT, KLT, JBX, XYZ, BWZ

Затем создадим новые группы, используя в качестве критерия средний символ, а не последний.

Группы строк, объединенных по второму символу

(AAC) (ABC, JBX) (RDT) (AEO) (TLJ, RLT, KLT) (BWZ) (XYZ)

Теперь в каждой группе собраны строки, имеющие одинаковый средний символ. Сами группы упорядочены по алфавиту. Порядок внутри группы сохранен прежним.

Объединим эти группы в одну, сохраняя относительный порядок следования строк.

Результат объединения групп

AAC, ABC, JBX, RDT, AEO, TLJ, RLT, KLT, BWZ, XYZ

Теперь образуем новые группы, используя в качестве критерия первый символ.

Группы строк, объединенных по первому символу

(AAC, ABC, AEO) (BWZ) (JBX) (KLT) (RDT, RLT) (TLJ) (XYZ)

В заключение объединим все группы, сохраняя порядок, установленный в каждой из них.

Упорядоченные строки

AAC, ABC, AEO, BWZ, JBX, KLT, RDT, RLT, TLJ, XYZ

Теперь строки упорядочены.

В этом примере все строки имели одинаковую длину. Если строки имеют разную длину, их можно дополнить пробелами и упорядочить, как показано выше.

Для упорядочения чисел алгоритм поразрядной сортировки рассматривает их как строки символов. Числа можно выравнивать, приписывая слева незначащие нули, так чтобы все строки имели одинаковую длину. Затем формируются группы в соответствии с последней цифрой, далее эти группы объединяются в одну, затем создаются группы чисел, имеющих одинаковую предпоследнюю цифру, затем они объединяются в одну группу и т.д., как в предыдущем примере. Результат сортировки восьми целых чисел показан на рис. 9.21.

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Исходные целые числа
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Группы по четвертой цифре
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Объединенная группа
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Группы по третьей цифре
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Объединенная группа
(00**0**4, 10**6**1) (01**2**3, 21**5**0, 21**5**4) (02**2**2, 02**8**3) (15**6**0)	Группы по второй цифре
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Объединенная группа
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Группы по первой цифре
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Объединенная (упорядоченная) группа

Рис. 9.21. Результат сортировки восьми целых чисел

Ниже приведен псевдокод алгоритма поразрядной сортировки n десятичных целых чисел, состоящих из d цифр.

```
radixSort(inout theArray:ItemArray,
          in n:integer, in d:integer)
// Упорядочивает массив theArray, содержащий n целых чисел,
// состоящих из d цифр

   for (j = d вниз до 1)
   {
     Инициализировать 10 пустых групп
     Инициализировать нулем счетчик каждой группы

     for (i = 0 до n-1)
     {
         k = j-я цифра элемента theArray[i]
         Поместить элемент theArray[i] в конец группы k
         Увеличить k-й счетчик на 1
     } // Конец цикла по i
    Записать в массив theArray элементы группы 0,
      затем элементы группы 1 и т.д.
} // Конец цикла по j
```

Анализ. Рассматривая псевдокод алгоритма поразрядной сортировки, легко увидеть, что в нем при каждом разбиении на группы выполняется m перемещений, а при объединении групп — n перемещений. Эти $2*n$ перемещений алгоритм выполняет d раз. Следовательно, при поразрядной сортировке выполняется $2*n*d$ перемещений n строк, состоящих из d символов. Однако при этом не требуется выполнять никаких сравнений. Итак, сложность алгоритма поразрядной сортировки оценивается величиной O(n).

Несмотря на высокую оценку эффективности, алгоритм поразрядной сортировки имеет некоторые недостатки, которые не позволяют считать его универсальным. Например, для поразрядной сортировки строк, состоящих из прописных букв, необходимо предусмотреть 27 групп — одну для пробела и по одной для каждой буквы (в английском алфавите. — *Прим. ред.*). Если исходный набор данных состоит из n строк, каждая группа должна вмещать n строк. Для больших значений n, если входные данные и результат хранятся в массивах, это требование становится невыполнимым. Однако применение связных списков позволяет сэкономить память. Таким образом, поразрядная сортировка больше подходит для упорядочения связанных списков, а не массивов.

> Несмотря на то что сложность алгоритма поразрядной сортировки оценивается величиной O(n), его нельзя назвать универсальным

Сравнение алгоритмов сортировки

На рис. 9.22 показаны приближенные оценки сложности алгоритмов сортировки, рассмотренных в главе, в наихудшем и среднем вариантах. В таблицу включены древовидная и пирамидальная сортировки, которые будут рассмотрены в главах 10 и 11 соответственно.

	Наихудший вариант	Средний вариант
Сортировка методом выбора	n^2	n^2
Сортировка методом пузырька	n^2	n^2
Сортировка методом вставок	n^2	n^2
Сортировка слиянием	n * log n	n * log n
Быстрая сортировка	n^2	n * log n
Поразрядная сортировка	n	n
Древовидная сортировка	n^2	n * log n
Пирамидальная сортировка	n * log n	n * log n

Рис. 9.22. Приближенные оценки сложности алгоритмов сортировки

Резюме

1. Оценка порядка величины и обозначение О-большое позволяют представить сложность алгоритма в виде функции, зависящей от размера задачи. Этот подход позволяет анализировать эффективность алгоритма независимо от быстродействия компьютера и мастерства программиста.
2. Сравнение эффективности алгоритмов сводится к оценке их сложности при решении больших задач. При этом учитываются лишь значительные различия.
3. При анализе наихудшего варианта оценивают максимальный объем работы, необходимый для решения задачи заданного размера, а при анализе среднего варианта — вероятный объем работы.
4. Оценка сложности алгоритма позволяет правильно выбрать реализацию абстрактного типа данных. Если приложение часто требует выполнения определенных операций, реализация абстрактного типа данных должна быть эффективной по крайней мере по отношению к ним.
5. Алгоритмы сортировки методом выбора, сортировки методом пузырька и методом вставок имеют сложность $O(n^2)$. Хотя в некоторых случаях они работают быстрее других, при решении больших задач они не эффективны.
6. Рекурсивные алгоритмы быстрой сортировки и сортировки слиянием чрезвычайно эффективны. В среднем, быстрая сортировка считается одним из самых эффективных из существующих алгоритмов сортировки. Однако в наихудшем случае этот алгоритм работает намного медленнее, чем сортировка слиянием. К счастью, на практике наихудший вариант встречается редко. Среднее быстродействие алгоритма сортировки слиянием ниже, чем у быстрой сортировки, однако его эффективность одинаково хороша во всех вариантах. Единственным недостатком этого алгоритма является завышенные требования к объему памяти, поскольку нужно хранить вспомогательный массив, равный по объему исходному.

Предупреждения

1. Как правило, следует избегать оценки эффективности алгоритма только по *скорости выполнения его конкретной реализации*. На *быстродействие* программы влияют множество факторов, например, стиль программирования, мощность компьютера и даже особенности входных данных.
2. Сравнивая эффективность разных решений, следует учитывать только существенные различия.

3. Работая с обозначением О-большое, следует помнить, что величина O($f(n)$) означает неравенство. Это не функция, а просто обозначение, имеющее смысл "величина порядка $f(n)$" или "имеет порядок $f(n)$".
4. Если размер задачи невелик, не стоит слишком подробно анализировать сложность алгоритма. В этом случае основным фактором является его простота. Например, для сортировки небольшого массива — не более 25 элементов — вполне подходит простой алгоритм сортировки вставками, хотя его сложность имеет порядок O(n^2).
5. При сортировке очень большого массива алгоритмы, сложность которых оценивается величиной O(n^2), не эффективны.
6. Алгоритм быстрой сортировки следует выбирать, только если элементы массива записаны в произвольном порядке. Хотя в худшем случае сложность алгоритма быстрой сортировки оценивается величиной O(n^2), на практике такая ситуация встречается редко.

Вопросы для самопроверки

1. Сколько сравнений элементов массива выполняется внутри следующего цикла?

```
for (j = 1; j <= n-1; ++j)
{
   i = j + 1;
   do
   {
      if (theArray[i] < theArray[j])
          swap(theArray[i], theArray[j]);
          ++i;
   } while (i <= n);
} // Конец оператора for
```

2. Повторите предыдущее задание, заменив оператор `i = j + 1` оператором `i = j`.
3. Какой порядок имеют следующие функции.

   3.1. $8 * n^3 - 9 * n$

   3.2. $7 * \log_2 n + 20$

   3.3. $7 * \log_2 n + n$
4. Проанализируйте алгоритм последовательного поиска элемента в массиве, имеющем длину n.

   4.1. Как проверить, принадлежит ли искомый элемент массиву, не выполняя n сравнений, если элементы упорядочены по возрастанию?

   4.2. Какой порядок имеет сложность алгоритма последовательного поиска, если искомого элемента в массиве нет? Оцените сложность алгоритма для упорядоченных и неупорядоченных данных; рассмотрите наихудший, средний и наилучший варианты.

   4.3. Докажите, что если искомый элемент принадлежит массиву, сложность алгоритма не зависит от порядка следования данных.
5. Выполните трассировку алгоритма сортировки методом выбора, если нужно упорядочить в порядке возрастания следующий массив: 20 80 40 25 60 30.

6. Выполните упражнение 5, упорядочив массив в порядке убывания.
7. Выполните трассировку алгоритма сортировки методом выбора, если нужно упорядочить в порядке возрастания следующий массив: 23 30 20 80 40 60.
8. Выполните трассировку алгоритма сортировки вставками, если нужно упорядочить в порядке возрастания массив из предыдущего задания.
9. Докажите, что алгоритм сортировки слиянием удовлетворяет четырем критериям рекурсии, описанным в главе 2.
10. Выполните трассировку алгоритма разбиения массива, являющего частью быстрой сортировки, при упорядочении следующего массива: 38 16 40 39 12 27.
11. Допустим, нужно упорядочить большой массив, состоящий из целых чисел, используя сортировку слиянием. Затем в этом массиве выполняется бинарный поиск заданного целого числа. В заключение на экран выводятся все целые числа, принадлежащие уже упорядоченному массиву.
    11.1. Какой алгоритм в принципе быстрее: сортировка слиянием или бинарный поиск? Запишите это утверждение, пользуясь обозначением О-большое.
    11.2. Какой алгоритм в принципе быстрее: бинарный поиск или вывод на экран? Запишите это утверждение, пользуясь обозначением О-большое.

Упражнения

1. Оцените сложность решения каждой из указанных ниже задач в наихудшем случае.
    1.1. Вычисление суммы первых *n* целых чисел с помощью цикла `for`.
    1.2. Вывод на экран всех *n* целых чисел, содержащихся в массиве.
    1.3. Вывод на экран всех *n* целых чисел, содержащихся в упорядоченном связанном списке.
    1.4. Вывод на экран всех *n* целых чисел, содержащихся в кольцевом связанном списке.
    1.5. Вывод на экран одного элемента, принадлежащего массиву.
    1.6. Вывод на экран последнего элемента, принадлежащего связанному списку.
    1.7. Поиск конкретного числа в массиве, состоящем из *n* чисел, с помощью алгоритма бинарного поиска.
    1.8. Упорядочение в порядке убывания массива, состоящего из *n* целых чисел, с помощью алгоритма сортировки слиянием.
    1.9. Добавление элемента в стек, состоящий из *n* элементов.
    1.10. Добавление элемента в очередь, состоящую из *n* элементов.
2. Допустим, что реализация конкретного алгоритма на языке C++ выглядит следующим образом.

```
for (int pass = 1; pass <= n; ++pass)
   {
     for (int index = 0; index < n; ++index)
     {
         for (int count = 1; count < 10; ++count)
         {
```

```
            . . .
        } // Конец оператора for
    } // Конец оператора for
} // Конец оператора for
```

Здесь пропущены операторы, выполняющие конкретные вычисления. Будем считать, что они не зависят от числа n. Оцените порядок этого алгоритма. Обоснуйте свой ответ.

3. Проанализируйте функцию *f*, приведенную ниже. Она вызывает функцию *swap*. Предположим, что функция *swap* существует и просто выполняет перестановку двух аргументов. Назначение функции *f* в данном упражнении не важно.

```
void f(int theArray[], int n)
   {
     for (int j = 0; j < n; ++j)
     {
         int i = 0;
         while (i <= j)
         {
            if (theArray[i] < theArray[j])
                swap(theArray[i], theArray[j]);
                ++i;
         } // Конец оператора while
     } // Конец оператора for
} // Конец функции f
```

Сколько сравнений выполняет функция *f*?

4. Сравните быстродействие алгоритмов сортировки методом выбора и сортировки методом вставок при упорядочении большого массива и в наихудшем случае. Обоснуйте свой ответ.

5. Докажите, что полином $f(x)=c_n x^n+c_{n-1}x^{n-1}+\ldots+c_1 x+c_0$ имеет порядок $O(x_n)$.

6. Докажите, что для любых констант a, $b > 1$ функция $f(n)$ имеет порядок $O(\log_a n)$ тогда и только тогда, когда $f(n)$ имеет порядок $O(\log_b n)$. Следовательно, основание логарифма можно игнорировать. *Подсказка*: используйте тождество $\log_a n=\log_b n/\log_a b$ при любых константах a, $b > 1$.

7. При анализе алгоритма методом выбора мы игнорировали операции цикла и манипуляции с индексами массива. Проанализируйте снова этот алгоритм, учитывая *все* операции, и докажите, что его сложность по-прежнему оценивается величиной $O(n^2)$.

8. Выполните трассировку алгоритма сортировки методом вставок, если нужно упорядочить в порядке возрастания следующий массив: 20 80 40 25 60 40.

9. Примените алгоритмы сортировки методом выбора, методом пузырька и методом ставок для упорядочения следующих массивов.

   9.1. Массив, записанный в обратном порядке: 8 6 4 2.

   9.2. Упорядоченный массив: 2 4 6 8.

10. Подберите массив, при упорядочении которого алгоритм сортировки методом пузырька работает хуже всего.

11. Модифицируйте функцию *selectionSort* так, чтобы она упорядочивала массив структур по их целочисленному полю, которое считается ключом сортировки. Повторите это упражнение для массива, состоящего из объек-

тов некоторого класса. Предположите, что этот класс содержит функцию-член *sortKey*, возвращающую целочисленный ключ сортировки.

12. Напишите рекурсивную версию функций *selectionSort*, *bubbleSort* и *insertionSort*.

13. Выполните трассировку алгоритма сортировки слиянием при упорядочении массива, указанного ниже, в возрастающем порядке. Перечислите вызовы функций *mergeSort* и *merge* в порядке их появления.

    20 80 40 25 60 30

14. Выполните сортировку массива методом слияния.

    14.1. Зависят ли рекурсивные вызовы функции *mergeSort* от элементов массива или их количества, или от обоих факторов сразу? Обоснуйте свой ответ.

    14.2. На каком этапе выполнения функции *mergeSort* осуществляются фактические перестановки элементов массива (т.е. упорядочение массива). Обоснуйте свой ответ.

15. Выполните трассировку алгоритма быстрой сортировки при упорядочении массива, указанного ниже, в возрастающем порядке. Перечислите вызовы функций *quickSort* и *partition* в порядке их появления.

    20 80 40 25 10 15

16. Допустим, вы удалили вызовы функции *merge* из алгоритма *mergeSort* и получили в результате следующую функцию.

```
mystery(inout theArray:ItemArray, in n:integer)
// Таинственный алгоритм, работающий с массивом
// theArray[0..n-1]
   if (n > 1)
   {
       Mystery(lefthalf(theArray))
       Mystery(righthalf(theArray))
   } // Конец оператора if
```

    Для чего предназначен новый алгоритм?

17. При быстрой сортировке массива в качестве опорного можно выбрать любой элемент. Для этого достаточно записать любой элемент в ячейку *theArray[first]*. Один из способов выбора опорного элемента заключается в вычислении среднего значения трех величин: *theArray[first]*, *theArray[first+lats]/2]* и *theArray[last]*. Сколько рекурсивных вызовов необходимо выполнить для сортировки массива, состоящего из n элементов, если опорный элемент всегда выбирается описанным выше способом.

18. Алгоритм разбиения, использующийся при быстрой сортировке, перемещает элементы из неопределенной части массива в множества S_1 и S_2. Если элемент, подлежащий переносу, принадлежит множеству S_1, а множество S_2 пусто, алгоритм выполнит фиктивную перестановку элемента (он останется на месте). Модифицируйте алгоритм разбиения так, чтобы исключить ненужные перестановки.

19. Используя инварианты, докажите, что функция *selectionSort* работает корректно.

20. Опишите итеративное решение функции *mergeSort*. Определите подходящий инвариант и докажите правильность вашего алгоритма.

21. Одним из критериев оценки алгоритма сортировки является его надежность. Алгоритм сортировки называется **надежным** (stable), если он не выполняет перестановку элементов, имеющих одинаковые ключи сортировки. Итак, элементы, имеющие одинаковые ключи сортировки (но, возможно, отличающиеся другими свойствами), не изменяют взаимного расположения. Например, массив записей о студентах можно упорядочивать по имени и году выпуска. Используя надежный алгоритм, выполните сортировку этого массива по году выпуска так, чтобы студенты, имеющие одинаковый год выпуска, оставались упорядоченными по имени. В некоторых приложениях применение надежного алгоритма является обязательным условием. Какие из алгоритмов, рассмотренных в главе, являются устойчивыми?

22. При описании алгоритма поразрядной сортировки мы упоминали колоду карт, упорядоченных сначала по рангу, а потом по масти. Для реализации такой сортировки каждую карту необходимо представлять с помощью двух символов, а десятке можно поставить в соответствие символ T. Например, S2 — это двойка пик, а HT — червонная десятка.

    22.1. Выполните трассировку алгоритма на описанном выше примере.

    22.2. Допустим, что для представления десятки используется не символ T, а просто число 10. Например, H10 означает червонную десятку, а строки, состоящие из двух символов, дополняются справа пробелами. Как применить поразрядную сортировку в этом случае?

Задания по программированию

1. Добавьте в функции *insertionSort* и *mergeSort* счетчики, отслеживающие количество выполненных сравнений. Примените обе функции к массивам, имеющим разные размеры. При каком размере массива количество сравнений, выполняемых в этих функциях, значительно отличается друг от друга? Как согласуется этот размер с порядком сложности алгоритма?

2. Функция *quickSort* вызывает функции *choosePivot* для выбора опорного элемента и помещает его в первую ячейку массива. Реализуйте функцию *choosePivot* двумя способами.

    2.1. В качестве опорного всегда выбирается первый элемент.

    2.2. Опорный элемент выбирается способом, описанным в упражнении 17.

    Добавьте в функцию *partition* счетчик, отслеживающий количество выполненных сравнений. Примените алгоритм *quickSort* для сортировки массивов, имеющих разные размеры, каждый раз иначе выбирая опорный элемент. При каком размере массива количество сравнений, выполняемых в этих функциях, значительно отличается друг от друга? При какой стратегии выбора опорного элемента количество сравнений, выполняемых в этих функциях, значительно отличается друг от друга?

3. Выполните следующие задания.

    3.1. Модифицируйте алгоритм разбиения массива в алгоритме *quickSort* так, чтобы множества S_1 и S_1 никогда не были пустыми.

    3.2. Существует иная стратегия разбиения массива. Пусть индекс *low* пробегает по отрезку *theArray[first..last]* от индекса *first* до индекса

last и останавливается на первом элементе, превышающем опорный. Аналогично, второй индекс, *high*, пробегает по отрезку *theArray[first..last]* от индекса *last* до индекса *first* и останавливается на первом элементе, меньшем опорного. Затем выполняется перестановка этих двух элементов, увеличение индекса *low* на единицу и уменьшение индекса *index* на единицу. Процесс продолжается до тех пор, пока индексы *high* и *low* не встретятся где-то посередине. Реализуйте эту версию алгоритма разбиения на языке C++. Благодаря чему множества S_1 и S_1 никогда не будут пустыми?

3.3. Существуют несколько вариантов стратегии разбиения, описанной выше. Предложите собственную стратегию. Сравните ее с существующими.

4. Выполните поразрядную сортировку массива, используя для представления каждой группы абстрактную очередь.

5. Выполните поразрядную сортировку связанного списка, состоящего из целых чисел.

6. Выполните следующие задания.

   6.1. Отсортируйте связанный список, состоящий из целых чисел, с помощью сортировки слиянием.

   6.2. Примените другие алгоритмы сортировки, подходящие для связанного списка.

7. При сортировке массива целых чисел, изменяющихся от 1 до 100, можно использовать массив *count*, в котором записано, сколько раз встречается каждое из чисел. Опишите детали этого алгоритма, называемого **блочной сортировкой** (bucket sort), и напишите соответствующую функцию на языке C++. Какова сложность блочной сортировки? Почему этот алгоритм нельзя считать универсальным?

8. Сортировка Шелла (названная так в честь своего изобретателя Дональда Шелла (Donald Shell)) представляет собой усовершенствованный вариант сортировки методом вставок. Вместо того чтобы переставлять только соседние элементы — как при сортировке методов вставок, — сортировка Шелла меняет местами удаленные друг от друга элементы. Массив сортируется таким образом, что каждый h-й элемент образует упорядоченный подмассив для каждого значения h. Величины h образуют убывающую последовательность. Например, если число h равно 5, то каждый пятый элемент формирует упорядоченный массив. Если значение h равно 1, весь массив оказывается полностью упорядоченным.

   Последовательность величин h может начинаться с числа $n/2$ и последовательно уменьшать число n вдвое, пока величина h не станет равной единице. С помощью этой последовательности, заменяя в алгоритме *insertionSort* число 1 величиной h, а число 0 — величиной h-1, можно получить следующую функцию, реализующую сортировку Шелла.

```
void shellsort(DataType theArray[], int n)
{
   for (int h = n/2; h > 0; h = h/2)
   {
      for (int unsorted = h; unsorted < n; ++unsorted)
      {
          DataType nextItem = theArray[unsorted];
          int loc = unsorted;
```

```
            loc = loc - h)
            theArray[loc] = theArray[loc-h];
            theArray[loc] = nextItem;
        } // Конец оператора for
    } // Конец оператора for
} // Конец функции shellsort
```

Добавьте в функции *insertionSort* и *shellSort* счетчики сравнений. Примените эти функции к массивам, имеющим разные размеры. При каком размере массива разница между количеством сравнений становится значительной?

Деревья

В этой главе ...

Терминология

Абстрактное бинарное дерево
- Обход бинарного дерева
- Способы представления бинарного дерева
- Реализация абстрактного бинарного дерева в виде связанного списка

Абстрактное бинарное дерево поиска
- Алгоритмы, реализующие операции над абстрактным бинарным деревом поиска
- Реализация абстрактного бинарного дерева поиска с помощью указателей
- Эффективность операций над бинарными деревьями поиска
- Древовидная сортировка
- Запись бинарного дерева поиска в файл

Деревья общего вида

Резюме

Предупреждения

Вопросы для самопроверки

Задания по программированию

Введение. Структуры данных, рассмотренные в предыдущих главах, были линейными, поскольку их элементы следовали друг за другом. В данной главе рассматриваются нелинейные, иерархические структуры данных, в которых каждый элемент может иметь несколько преемников. В частности, в главе изучаются спецификации, способы реализации и относительная эффективность абстрактного бинарного дерева и абстрактного бинарного дерева поиска. Именно этим трем абстрактным типам данных посвящены следующие три главы.

Основные категории операций управления данными

В предыдущих главах рассматривались абстрактные типы данных, операции над которыми относились к одной из следующих категорий.

- Операции вставки элемента в набор данных.
- Операции удаления элемента из набора данных.
- Операции поиска элемента в наборе данных.

Виды операций над позиционно-ориентированными абстрактными типами данных

Абстрактные список, стек и очередь представляют собой **позиционно-ориентированные типы данных** (position-oriented data types). Операции над ними формулировались следующим образом.

- Вставить элемент в *i*-ю *позицию* набора данных.
- Удалить элемент из *i*-й *позиции* набора данных.
- Найти элемент, стоящий на *i*-й *позиции* набора данных.

Абстрактный список не накладывает никаких ограничений на значение *i*, в то время как абстрактные стек и очередь не позволяют обращаться к произвольной позиции. Например, операции над абстрактным стеком сводятся лишь к вставке, удалению и просмотру вершины стека. Следовательно, несмотря на различную гибкость операций, списки, стеки и очереди устанавливают соответствие между своими элементами и их *позициями*.

Виды операций над абстрактными типами данных, ориентированными на значения

Абстрактный упорядоченный список **ориентирован на значения элементов** (value oriented). Операции над ним имеют следующий вид.

- Вставить элемент, имеющий *значение x*.
- Удалить элемент, имеющий *значение x*.
- Найти элемент, имеющий *значение x*.

Хотя эти операции, как и позиционно-ориентированные операции, разделяются на три категории — вставка, удаление и поиск элемента, — они основаны на *значениях* элементов, а не на их *позициях*.

В главе обсуждаются два основных абстрактных типа данных: бинарное дерево и бинарное дерево поиска. Как мы увидим, бинарное дерево является позиционно-ориентированным абстрактным типом данных, но не линейным, как списки, стеки и очереди. Следовательно, на элементы бинарного дерева нельзя ссылаться, указывая их позицию. Изучив бинарные деревья, мы перейдем к более полезному типу данных — бинарному дереву поиска, ориентированному на значения элементов.

В следующей главе мы встретимся с еще двумя абстрактными типами данных, ориентированными на значения элементов, — таблицей и очередью с приоритетами. Реализация этих абстрактных типов основана на идеях, изложенных в этой главе.

Терминология

Деревья (trees) используются для представления отношения. В предыдущих главах мы уже использовали деревья для неформального изображения отношений между вызовами в рекурсивном алгоритме. Например, диаграмма рекурсивных вызовов алгоритма `rabbit` из главы 2, изображенная на рис. 10.1, фактически представляет собой дерево. Каждый вызов алгоритма `rabbit` изображается блоком, или **узлом** (node), или **вершиной** (vertex). Линии, соединяющие узлы (блоки), называются **ребрами** (edges). В данном дереве ребра означают рекурсивные вызовы. Например, ребра, выходящие из вершины `rabbit(7)` к вершинам `rabbit(6)` и `rabbit(5)`, означают, что подзадача `rabbit(7)` сводится к решению подзадач `rabbit(6)` и `rabbit(5)`, для решения которых вновь применяется алгоритм `rabbit`.

Деревья являются иерархическими структурами данных

Все деревья являются **иерархическими** (hierarchical) по своей сути. Интуитивное значение этого термина состоит в том, что между узлами дерева существует отношение "родительский-дочерний". Если ребро соединяет узел *n* и узел *m*, причем узел *n* находится выше узла *m*, то узел *n* считается **родителем** (parent) узла *m*, а узел *m* — его **дочерним узлом** (child) . В дереве, изображенном на рис. 10.1, узлы *B* и *C* являются дочерними по отношению к узлу *A*. Дочерние узлы, имеющие одного и того же родителя, — например, узлы *B* и *C* — называются **братьями** (siblings) . Каждый узел дерева имеет по крайней мере одного родителя, причем в дереве существует только один узел, не имеющий предков. Такой узел называется **корнем дерева** (root) . На рис. 10.1 корнем является узел *A*. Узел, у которого нет дочерних узлов, называется **листом дерева** (leaf). Листьями дерева, изображенного на рис. 10.1, являются узлы *C*, *D*, *E* и *F*.

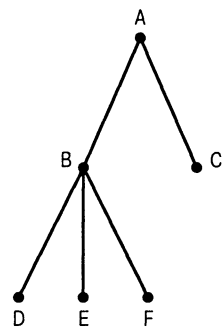


Рис. 10.1. Дерево общего вида

Отношения между родительским и дочерним узлом с абстрактной точки зрения является отношением между **предком** (ancestor) и **потомком** (descendant). На рис. 10.1 узел *A* является предком узла *D*, следовательно, узел *D* является потомком узла *A*. Не все узлы могут быть связаны отношениями "предок-потомок": узлы *B* и *C*, например, не связаны родственными отношениями. Однако корень любого дерева является предком каждого его узла. **Поддеревом** (subtree) называется любой узел дерева со всеми его потомками. **Поддеревом узла** *n* является поддерево, корнем которого является узел *n*. Например, на рис. 10.2 показано поддерево дерева, изображенного на рис. 10.1. Корнем этого поддерева является узел *B*, а само поддерево является поддеревом узла *A*.

Благодаря иерархической природе деревьев, их можно использовать для представления информации, имеющей иерархическую структуру, например, структуры организации (рис. 10.3, *а*) или генеалогического древа (рис. 10.3, *б*). При этом обнаруживаются странные вещи! Узлы генеалогического древа на рис. 10.3, *б*, представляющие родителей Каролины (Джона и Жаклин), являются дочерними по отношению к узлу, представляющему саму Каролину! Иными словами, узлы, представляющие предков Каролины, являются потомками узла, представляющего ее саму. Неудивительно, что программистов считают людьми со странностями.

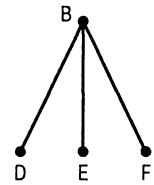


Рис. 10.2. Поддерево дерева, изображенного на рис. 10.1

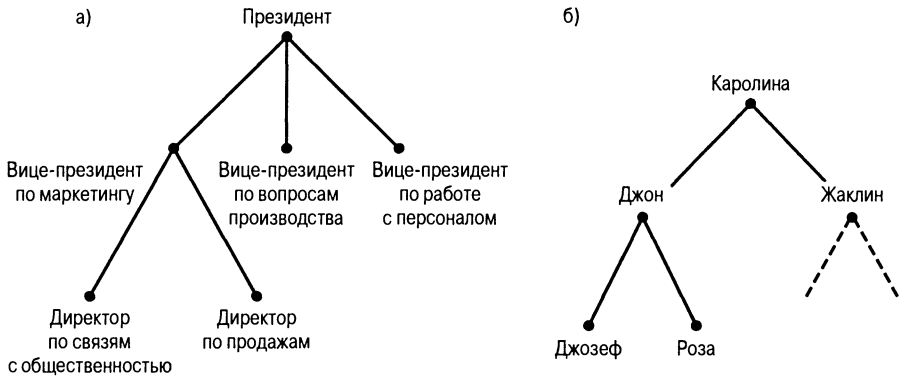


Рис. 10.3. Изображение иерархических структур в виде деревьев: а) структура организации; б) генеалогическое древо

С формальной точки зрения, **дерево общего вида** (general tree) T представляет собой множество, состоящее из одного или нескольких узлов, принадлежащих непересекающимся подмножествам, указанным ниже.

- Отдельный узел r, корень.
- Множество поддеревьев корня r.

Таким образом, деревья, изображенные на рис. 10.1 и 10.3, *а* являются общими.

Формальное определение бинарного дерева

Основное внимание в этой главе уделяется бинарным деревьям. С формальной точки зрения, **бинарное дерево** (binary tree) — это множество узлов T, таких что

- множество T пусто, или
- множество T распадается на три непересекающихся подмножества:
  - отдельный корень r, корень;
  - два возможно пустых поддерева бинарного дерева, которые называются **левым и правым поддеревьями** (leaf and right subtrees) корня r, соответственно.

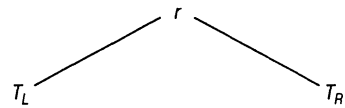
Деревья, изображенные на рис. 10.1 и 10.3, *б*, являются бинарными. Обратите внимание, что каждый узел бинарного дерева имеет не более двух дочерних узлов. Бинарное дерево не является разновидностью дерева общего вида, поскольку бинарное дерево может быть пустым, а дерево общего вида — нет.

Иногда полезно пользоваться интуитивным определением бинарного дерева.

Интуитивное определение бинарного дерева

Дерево T называется бинарным, если выполняется одно из двух условий:

- дерево T не имеет узлов, или
- дерево T имеет вид, показанный на рисунке ниже.



Здесь r обозначает узел, а T_L и T_R — бинарные деревья.

Обратите внимание, что это формальное определение хорошо согласуется с интуитивным: если узел r является корнем дерева T, то бинарное дерево T_L — его левое поддерево, а бинарное дерево T_R, соответственно, правое. Если дерево T_L не пусто, то его корень является **левым дочерним узлом** (left child) узла r. Аналогично, если дерево T_R не пусто, то его корень является **правым дочерним узлом** (right child) узла r. Если оба поддерева узла пусты, он является листом.

В качестве иллюстрации использования бинарных деревьев для представления данных в иерархическом виде рассмотрим рис. 10.4. На этом рисунке бинарные деревья представляют алгебраические выражения, содержащие бинарные операторы +, -, * и /. Для представления выражения $a-b$ оператор помещается в корневой узел, а операнды a и b — в его левый и правый дочерние узлы, соответственно (рис. 10.4). На рис. 10.4, *б* представлено выражение $a-b/c$, где поддерево представляет подвыражение b/c. Аналогичная ситуация показана на рис. 10.4, *в*, где изображено выражение $(a-b)*c$. В узлах этих деревьев хранятся операнды выражений, а в остальных узлах — операторы. Скобки в этих деревьях не представлены. Бинарное дерево создает иерархию операций, т.е. дерево однозначно определяет порядок вычисления выражения.

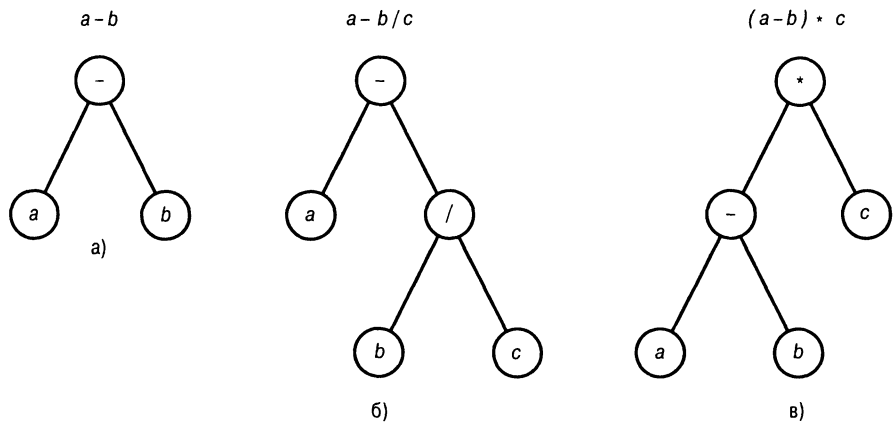


Рис. 10.4. Бинарные деревья, представляющие алгебраические выражения

Свойства бинарного дерева поиска

Обычно в узлах деревьев содержатся некие значения. **Бинарное дерево поиска** (binary search tree) — это бинарное дерево, некоторым образом упорядоченное в соответствии со значениями, содержащимися в его узлах. Каждый узел n бинарного дерева поиска обладает следующими тремя свойствами.

- Значение узла n больше всех значений, содержащихся в левом поддереве T_L.
- Значение узла n меньше всех значений, содержащихся в правом поддереве T_R.
- Деревья T_L и T_R являются деревьями бинарного поиска.

На рис. 10.5 показан пример бинарного дерева поиска. Как следует из его названия, это дерево обеспечивает возможности поиска конкретного элемента. Позднее мы рассмотрим эту структуру данных более подробно.

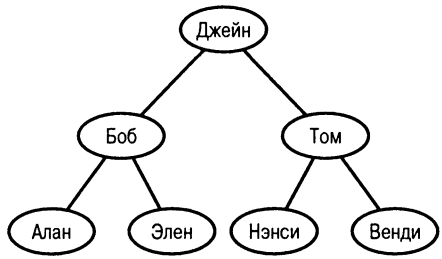


Рис. 10.5. Дерево бинарного поиска, содержащее имена

Высота деревьев. Деревья могут иметь разную форму. Например, хотя деревья, изображенные на рис. 10.6, состоят из одинаковых узлов, они имеют совершенно разную структуру. Каждое из этих деревьев содержит семь узлов, но некоторые деревья "выше" других. **Высота** (height) дерева — это количество узлов, расположенных на самом длинном пути от корня к листу. Например, высота деревьев, показанных на рис. 10.6, равна 3, 5 и 7, соответственно. Интуитивное представление о высоте деревьев может привести к заключению, что их высота равна 2, 4 и 6, соответственно. Действительно, многие авторы используют именно интуитивное определение высоты дерева (подсчитывая количество ребер на самом длинном пути от корня до дерева. — *Прим. ред.*). Однако принятое нами определение позволяет более четко формулировать многие алгоритмы и свойства деревьев.

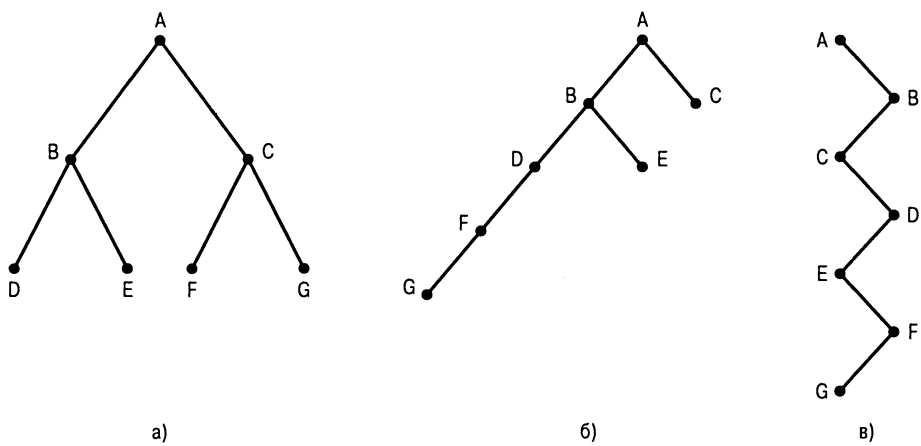


Рис. 10.6. Бинарные деревья поиска, имеющие одинаковые узлы, но разную высоту

Высоту дерева можно определить несколькими эквивалентными способами. Например, можно ввести понятие **уровня узла** n (level of the node).

Уровень узла

- Если узел n является корнем дерева T, он принадлежит первому уровню.
- В противном случае уровень узла на единицу больше, чем уровень его предка.

Например, узел A на рис. 10.6, *а* находится на первом уровне, уровень B — на втором, а узел D — на третьем.

Высота дерева, выраженная в терминах уровней узлов

Пользуясь понятием уровня узла, высоту дерева можно определить следующим образом.

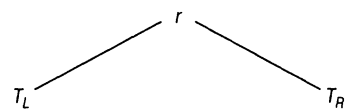
- Высота пустого дерева равна 0.
- Высота непустого дерева равна максимальному уровню его узлов.

Применяя это определение к деревьям, изображенным на рис. 10.6, легко установить, что их высота равна, 3, 5 и 7, соответственно, как было показано ранее.

Рекурсивное определение высоты дерева

Для бинарных деревьев часто бывает удобно пользоваться рекурсивным определением высоты дерева.

- Если дерево T пусто, его высота равна 0.
- Если дерево T — непустое бинарное дерево, то благодаря его форме



- высота дерева T на единицу больше, чем высота его самого высокого поддерева, т.е. $высота(T) = 1 + max\{высота(T_L)+высота(T_R)\}$.

Позднее, обсуждая эффективность алгоритмов поиска в бинарном дереве поиска, нам придется определять максимальную и минимальную высоту бинарного дерева, состоящего из n узлов.

Полное, совершенное и сбалансированное дерево. В **полном бинарном дереве** (full binary tree), имеющем высоту h, все узлы, расположенные на уровнях, меньших уровня h, имеют по два дочерних узла. На рис. 10.7 представлено полное бинарное дерево, имеющее высоту, равную 3. Каждый узел полного бинарного дерева имеет левое и правое поддерево, имеющие одинаковую высоту. Среди всех бинарных деревьев, высота которых равна h, полное бинарное дерево имеет максимально возможное количество листьев, и все они расположены на уровне h. Проще говоря, в полном бинарном дереве нет пропущенных узлов.

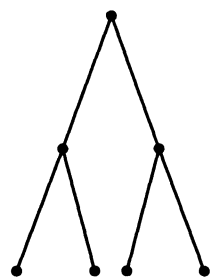


Рис. 10.7. Полное бинарное дерево, имеющее высоту равную 3

Доказывая свойства полных бинарных деревьев, подсчитывая количество их узлов, удобно пользоваться рекурсивным определением.

Полное бинарное дерево

- Если дерево T пусто, то оно является полным бинарным деревом, имеющим высоту, равную 0.
- Если дерево T не пусто, и его высота равна $h > 0$, то дерево T является полным бинарным деревом, если поддеревья его корня являются полными бинарными деревьями, имеющими высоту h-1.

  Это определение хорошо отражает рекурсивную природу бинарного дерева.

Совершенное бинарное дерево (complete binary tree), имеющее высоту h, — это бинарное дерево, которое является полным вплоть до уровня h-1, а уровень h заполнен слева направо так, как показано на рис. 10.8.

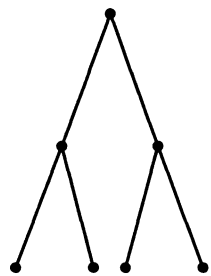


Рис. 10.8. Совершенное бинарное дерево

Совершенное бинарное дерево

Говоря более строго, бинарное дерево T, имеющее высоту h, является совершенным, если выполняются следующие условия.

1. Все узлы, начиная с уровня h-2 и выше, имеют по два дочерних узла.
2. Если узел, находящийся на уровне h-1, имеет дочерние узлы, то все узлы, находящиеся на этом же уровне слева от него, имеют по два дочерних узла.
3. Если узел, находящийся на уровне h-1, имеет один дочерний узел, он является его левым дочерним узлом.

Полные бинарные деревья являются совершенными

Пункты 2 и 3 этого определения формализуют требование, заключающееся в том, что уровень h должен быть заполнен слева направо. Очевидно, что полное бинарное дерево является совершенным.

Совершенные бинарные деревья являются сбалансированными

Бинарное дерево называется **сбалансированным по высоте** (height balanced), или просто **сбалансированным** (balanced), если высота правого поддерева любого его узла отличается от высоты левого поддерева не больше, чем на 1. Бинарные деревья, изображенные на рис. 10.8 и 10.6, *а*, являются сбалансированными, а деревья, представленные на рис. 10.6, *б* и 10.6, *в* — нет. Совершенное бинарное дерево является сбалансированным.

Итак, повторим кратко введенные понятия.

ОСНОВНЫЕ ПОНЯТИЯ

Терминология, связанная с деревьями

Дерево общего вида	Множество, состоящее из одного или нескольких узлов, разделенных на корень и подмножества, представляющие собой деревья общего вида
Родитель узла *n*	Узел дерева, находящийся непосредственно над узлом *n*
Дочерний узел узла *n*	Узел дерева, находящийся непосредственно под узлом *n*
Корень	Единственный узел дерева, не имеющий родителей
Лист	Узел, не имеющий дочерних узлов
Братья	Узлы, имеющие общих родителей
Предок узла *n*	Узел, расположенный на пути от корня к узлу *n*
Потомок узла *n*	Узел, расположенный на пути от узла *n* к листу
Поддерево узла *n*	Дерево, состоящее из дочернего узла (если он существует) узла *n* и его потомков
Высота	Количество узлов, расположенных на самом длинном пути от корня к листу
Бинарное дерево	Множество узлов, которое либо пусто, либо разделено на корень и одно или два подмножества, представляющих собой бинарные деревья
Левый (правый) дочерний узел узла *n*	Узел бинарного дерева, расположенный непосредственно ниже и левее (правее) узла *n*
Левое (правое) поддерево узла *n*	Левый (правый) дочерний узел (если он существует) узла *n* плюс его потомки (в бинарном дереве)
Бинарное дерево поиска	Бинарное дерево, в котором значение каждого узла *n* больше значения каждого узла левого поддерева узла *n*, но меньше значения каждого узла его правого поддерева
Пустое бинарное дерево	Бинарное дерево, не имеющее узлов
Полное бинарное дерево	Бинарное дерево, имеющее высоту *h*, в котором ни один узел не пропущен. Все листы, расположенные на уровне *h*, а также остальные узлы имеют по два дочерних узла
Совершенное бинарное дерево	Бинарное дерево, имеющее высоту *h*, которое является полным вплоть до уровня *h* - 1, причем уровень *h* заполнен слева направо
Сбалансированное бинарное дерево	Бинарное дерево, в котором высота левого и правого поддерева каждого узла не отличается больше, чем на 1

Абстрактное бинарное дерево

Будучи абстрактным типом данных, бинарное дерево предусматривает операции добавления и удаления узлов и поддеревьев. Используя эти основные операции, можно построить любое бинарное дерево. Остальные операции, касающиеся записи и извлечения данных из корня дерева, позволяют определить, пусто ли дерево.

Для бинарного дерева характерными являются операции обхода, которые *посещают* каждый узел. "Посещение" узла означает "выполнение какой-нибудь операции" над данным узлом. В главе 4 было рассмотрено понятие обхода свя-

занного списка: начиная с первого узла, мы проходили последовательно узел за узлом, пока не достигали конца списка. Однако в бинарном дереве существует несколько способов обхода. Стандартными являются три порядка обхода дерева: прямой, симметричный и обратный. Все они описываются в следующем разделе.

Над абстрактным бинарным деревом выполняются следующие операции.

ОСНОВНЫЕ ПОНЯТИЯ

Операции над абстрактным бинарным деревом

1. Создать пустое бинарное дерево.
2. Создать бинарное дерево, содержащее один узел, по заданному элементу.
3. Создать бинарное дерево по заданному корню и двум бинарным поддеревьям этого корня.
4. Уничтожить бинарное дерево.
5. Определить, пусто ли бинарное дерево.
6. Определить или изменить данные, записанные в корне бинарного дерева.
7. Присоединить к корню бинарного дерева левый или правый дочерний узел.
8. Присоединить к корню бинарного дерева левое или правое поддерево.
9. Отсоединить от корня бинарного дерева левое или правое поддерево.
10. Вернуть копию левого или правого поддерева корня бинарного дерева.
11. Обойти узлы бинарного дерева в прямом, симметричном или обратном порядке.

Более подробно эти операции описаны в псевдокоде, приведенном ниже, а также в UML-диаграмме класса бинарных деревьев, изображенной на рис. 10.9.

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операций над абстрактным бинарным деревом

```
// TreeItemType — это тип элементов, записанных в бинарном дереве

   +createBinaryTree()
   // Создает пустое бинарное дерево.

   +createBinaryTree(in rootItem:TreeItemType)
   // Создает бинарное дерево, состоящее из одного узла,
   // корень которого содержит элемент rootItem.

   +createBinaryTree(in rootItem:TreeItemType,
                     inout leftTree:BinaryTree,
                     inout rightTree:BinaryTree)
   // Создает бинарное дерево, корень которого содержит элемент
   // rootItem, а также имеет левое и правое поддеревья leftTree
   // и rightTree, соотвественно. Поддеревья leftTree и rightTree
   // создаются пустыми, поэтому их нельзя использовать для доступа к
   новому дереву.

   +destroyBinaryTree()
   // Уничтожает бинарное дерево.

   +isEmpty():boolean {query}
   // Определяет, пусто ли дерево.
```

```
+getRootData():TreeItemType throw TreeException
// Возвращает элемент, записанный в корне непустого бинарного
// дерева. Если дерево пусто, генерирует исключительную ситуацию
// TreeException.

+setRootData(in newItem:TreeItemType) throw TreeException
// Если дерево не пусто, заменяет элемент, содержащийся в корне
// бинарного дерева, элементом newItem. Если дерево пусто,
// создает корневой узел, содержащий элемент newItem, и вставляет
// новый узел в дерево. Если новый узел создать невозможно,
// генерируется исключительная ситуация TreeException.

+attachLeft(in newItem:TreeItemType) throw TreeException
// Присоединяет к корню бинарного дерева левый дочерний узел,
// содержащий элемент newItem. Если новый дочерний узел
// невозможно разместить в памяти, генерируется исключительная
// ситуация TreeException. Если бинарное дерево пусто
//(нет корня, к которому можно было бы присоединить дочерний
// узел) или левое поддерево уже существует (сначала его нужно
// отсоединить), также генерируется исключительная ситуация
// TreeException.

+attachRight(in newItem:TreeItemType) throw TreeException
// Присоединяет к корню бинарного дерева правый дочерний узел,
// содержащий элемент newItem. Если новый дочерний узел
// невозможно разместить в памяти, генерируется исключительная
// ситуация TreeException. Если бинарное дерево пусто
//(нет корня, к которому можно было бы присоединить дочерний
// узел) или правое поддерево уже существует (сначала его нужно
// отсоединить), также генерируется исключительная ситуация
// TreeException.

+attachLeftSubtree(inout leftTree:BinaryTree)
      throw TreeException
// Присоединяет дерево leftTree к корню в качестве левого
// поддерева и делает это дерево пустым, так что его невозможно
// использовать для доступа к новому дереву. Если бинарное дерево
// пусто (нет корня, к которому можно было бы присоединить
// левое поддерево) или левое поддерево уже существует
// (сначала его нужно отсоединить), генерируется
// исключительная ситуация TreeException.

+attachRightSubtree(inout rightTree:BinaryTree)
      throw TreeException
// Присоединяет дерево rightTree к корню в качестве правого
// поддерева и делает это дерево пустым, так что его невозможно
// использовать для доступа к новому дереву. Если бинарное дерево
// пусто (нет корня, к которому можно было бы присоединить
// правое поддерево) или правое поддерево уже существует
// (сначала его нужно отсоединить), генерируется
// исключительная ситуация TreeException.

+detachLeftSubtree(out leftTree:BinaryTree)
throw TreeException
```

```
// Отсоединяет от корня его левое поддерево и хранит его в
// переменной getLeftTree. Если бинарное дерево пусто
// (нет корня, от которого можно было бы отсоединить
// поддерево), генерируется исключительная ситуация TreeException.

 +detachRightSubtree(out rightTree:BinaryTree)
throw TreeException
// Отсоединяет от корня его правое поддерево и хранит его в
// переменной getLeftTree. Если бинарное дерево пусто
// (нет корня, от которого можно было бы отсоединить
// поддерево), генерируется исключительная ситуация TreeException.

+leftSubtree():BinaryTree
// Возвращает копию левого поддерева корня бинарного дерева,
// не отсоединяя его. Если бинарное дерево пусто, возвращает
// пустое бинарное дерево

+rightSubtree():BinaryTree
// Возвращает копию правого поддерева корня бинарного дерева,
// не отсоединяя его. Если бинарное дерево пусто, возвращает
// пустое бинарное дерево

+preorderTraverse(in visit:FunctionType)
// Выполняет прямой обход бинарного дерева и для каждого узла
// один раз вызывает функцию visit().

+inorderTraverse(in visit:FunctionType)
// Выполняет симметричный обход бинарного дерева
// и для каждого узла один раз вызывает функцию visit().

+postorderTraverse(in visit:FunctionType)
// Выполняет обратный обход бинарного дерева и для каждого узла
// один раз вызывает функцию visit().
```

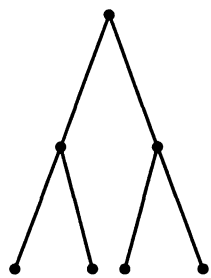


Рис. 10.9. UML-диаграмма класса BinaryTree

Эти операции можно использовать, например, для создания бинарного дерева, изображенного на рис. 10.6, *а*, где метки узлов являются символами. Псевдокод, приведенный ниже, создает дерево на основе поддеревьев `tree1` с корнем в узле `'F'`, поддерева `tree2` с корнем в узле `'D'`, поддерева `tree3` с корнем в узле `'B'` и поддерева `tree4` с корнем в узле `'C'`. В исходном положении все поддеревья пусты.

Применение операций над абстрактным бинарным деревом для создания конкретного бинарного дерева

```
tree1.setRootData('F')
tree1.attachLeft('G')
tree2.setRootData('D')
tree2.attachLeftSubtree(tree1)
tree3.setRootData('B')
tree3.attachLeftSubtree(tree2)
tree3.attachRight('E')
tree4.setRootData('C')
// Дерево, изображенное на рис.10.6, б
binTree.createBinaryTree('A', tree3, tree4)
```

Обратите внимание, что объект

```
binTree.getLeftSubtree()
```

представляет собой копию поддерева *tree1* (с корнем в узле *'B'*), а операция

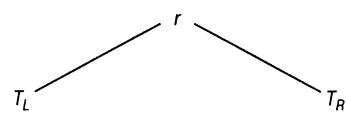
```
binTree.detachLeftSubtree(leftTree)
```

отсоединяет это поддерево от дерева *binTree* и называет его именем *leftTree*.

Обход бинарного дерева

Алгоритм обхода бинарного дерева посещает каждый узел. При этом над каждым узлом выполняется какая-нибудь операция, например, вывод его содержимого на экран или изменение данных. Для определенности будем считать, что при посещении узла на экран просто выводятся записанные в нем данные.

Используя рекурсивное определение бинарного дерева, можно создать рекурсивный алгоритм его обхода. Из определения следует, что бинарное дерево T либо является пустым, либо имеет вид, приведенный на рисунке ниже.



Если дерево T пусто, алгоритм обхода ничего не предпринимает — пустое дерево представляет собой базис рекурсии. Если дерево T не пусто, алгоритм обхода должен выполнить три задания: вывести данные, хранящиеся в корне r, и обойти деревья TL и TR, каждое из которых представляет собой бинарное дерево, размер которого меньше, чем у дерева T.

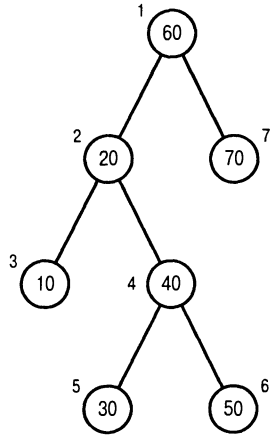
Общий вид рекурсивного алгоритма обхода

Итак, рекурсивный алгоритм обхода выглядит следующим образом.

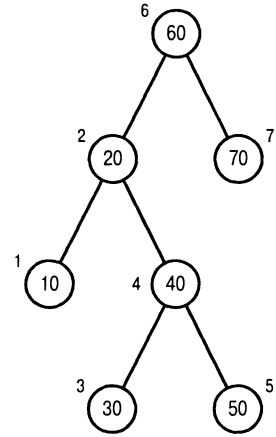
```
traverse(in binTree:BinaryTree)
// Traverses the binary tree binTree.

   if (дерево binTree не пусто)
   {
      traverse(левое поддерево корня дерева binTree)
      traverse(правое поддерево корня дерева binTree)
   }
```

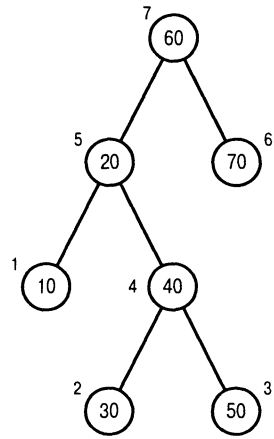
Однако этот алгоритм не полон. Здесь не указаны инструкции, выполняющие вывод на экран данных, содержащихся в корне. При обходе любого бинарного дерева алгоритм имеет три возможности при посещении корня r. Он может посетить корень r перед обходом обоих поддеревьев, после обхода левого поддерева T_L, но перед обходом правого поддерева T_R, или после обхода обоих поддеревьев.



а) Прямой обход: 60, 20, 10, 40, 30, 50, 70



б) Симметричный обход 10, 20, 30, 40, 50, 60, 70



в) Обратный обход: 10, 30, 50, 40, 20, 70, 60 (числа возле узлов обозначают порядок обхода)

Рис. 10.10. Обход бинарного дерева: а) прямой; б) симметричный; в) обратный

Такой порядок обхода называется **прямым** (preorder), **симметричным** (inorder) и **обратным** (postorder), соответственно. Результаты обхода конкретного бинарного дерева показаны на рис. 10.10.

Алгоритм прямого обхода выглядит следующим образом.

Прямой порядок обхода

```
preorder(in binTree:BinaryTree)
// Прямой обход бинарного дерева binTree.
// "Посещение узла" означает вывод на экран данных,
// которые в нем содержатся.

   if (дерево binTree не пусто)
   {
      Вывести на экран данные, содержащиеся в корне
      preorder(левое поддерево корня дерева binTree)
      preorder(правое поддерево корня дерева binTree)
   }
```

Прямой порядок обхода дерева, изображенного на рис. 10.10, порождает следующую последовательность чисел: 60, 20, 10, 40, 30, 50, 70. Если прямой порядок обхода применить к бинарному дереву, представляющему алгебраические операции, например, к дереву, изображенному на рис. 10.4, и вывести на экран данные, содержащиеся в его узлах, получим префиксную форму записи выражения1.

Симметричный обход

Алгоритм симметричного обхода имеет следующий вид.

```
inorder(in binTree:BinaryTree)
// Симметричный обход бинарного дерева binTree.
// "Посещение узла" означает вывод на экран данных,
// которые в нем содержатся.

   if (дерево binTree не пусто)
   {
      inorder(левое поддерево корня дерева binTree)
      Вывести на экран данные, содержащиеся в корне
      preorder(правое поддерево корня дерева binTree)
   }
```

Результатом симметричного обхода дерева, изображенного на рис. 10.10, *б*, является последовательность 10, 20, 30, 40, 50, 60, 70. Если алгоритм симметричного обхода применяется к бинарному дереву поиска, узлы посещаются в порядке следования их значений. Эта ситуация показана на рис. 10.10, *в*.

Обратный обход

В заключение рассмотрим алгоритм обратного обхода.

```
postorder(in binTree:BinaryTree)
// Обратный обход бинарного дерева binTree.
// "Посещение узла" означает вывод на экран данных,
// которые в нем содержатся.

   if (дерево binTree не пусто)
   {
      postorder(левое поддерево корня дерева binTree)
      postorder(правое поддерево корня дерева binTree)
      Вывести на экран данные, содержащиеся в корне
   }
```

В результате обратного обхода дерева, изображенного на рис. 10.10, *в*, на экран выводится следующая последовательность чисел: 10, 30, 50, 40, 20, 70, 60. Если применить алгоритм обратного обхода к бинарному дереву, представляющему алгебраическое выражение, например, к дереву, изображенному на рис. 10.4, и вывести на экран значения всех его узлов, получится постфиксная форма записи[2].

Сложность алгоритма обхода оценивается величиной O(n)

При каждом обходе каждый узел бинарного дерева посещается только один раз. Следовательно, при обходе дерева, состоящего из n узлов, будет выполнено n посещений. При каждом визите над узлами выполняется одна и та же операция, сложность которой не зависит от числа n, т.е. оценивается величиной O(1). Итак, сложность алгоритма обхода дерева в любом порядке оценивается величиной O(n).

---

[1] Префиксная форма записи такова: а) *-ab*; б) *-a/bc*; в) **-abc*.

[2] Постфиксная форма записи имеет вид: а) *ab-*; б) *abc/-*; в) *ab-c**.

Несмотря на то что обход дерева означает просто последовательное посещение каждого узла, выполнить его бывает чрезвычайно трудно, особенно если над узлом выполняется операция, более сложная, чем тривиальный вывод на экран. Например, данные, содержащиеся в узлах, можно копировать в другие структуры или изменять. Детали алгоритма обхода сильно зависят от конкретной реализации. Это очень затрудняет абстрагирование операции обхода дерева. Существует две возможности. Во-первых, для каждой операции, выполняемой над узлом дерева, можно предусмотреть отдельный алгоритм обхода, например, *preorderTraversAndDisplay*, *preorderTraversAndCopy* и т.д. Во-вторых, операция обхода может вызывать функцию, определенную клиентом, и передавать ей каждый узел дерева в качестве аргумента.

Например, операция над абстрактным бинарным деревом

```
binTree.preorderTraverse(visit)
```

получает функцию *visit* как аргумент. Если для вывода данных на экран предназначена функция *display*, ее можно передать операции обхода в качестве аргумента

Функция, определенная клиентом, которую вызывает операция обхода дерева, определяет смысл "посещения" узла

```
binTree.preorderTraverse(display)
```

В этом случае операция обхода дерева будет вызывать ее каждый раз при посещении нового узла.

Несмотря на то что операция обхода абстрактного дерева вызывает функцию, определенную клиентом, стена между программой и реализацией абстрактного типа данных остается невредимой. Поскольку функция *display* находится на клиентской стороне от стены, она может обращаться к данным только с помощью операций над абстрактным типом.

Детали реализации алгоритма обхода будут рассмотрены в следующих разделах.

Способы представления бинарного дерева

Для представления бинарного дерева в языке C++ есть три возможности. Две из них основаны на использовании массивов, но обычно для реализации бинарного дерева используются указатели. В любом случае выбранная структура данных содержится как закрытая переменная-член в классе бинарных деревьев.

Для иллюстрации этих трех подходов реализуем бинарное дерево имен. Каждый узел в этом дереве содержит имя, причем, поскольку дерево является бинарным, каждый узел имеет по крайней мере двух потомков.

Представление бинарного дерева в виде массива. Если для определения узла дерева используется класс, все дерево можно представить в виде массива объектов этого класса. Каждый узел дерева содержит данные — в нашем случае имя — и два индекса, по одному на каждый дочерний узел. Рассмотрим определение соответствующего класса на языке C++.

```
const int MAX_NODES = 100; // Максимальное количество узлов
typedef string TreeItemType;

class TreeNode // Узел дерева
{
private:
   TreeNode();
   TreeNode(const TreeItemType& nodeItem,
            int left, int right)
```

```
    TreeItemType item; // Данные
    int leftChild;     // Индекс левого дочернего узла
    int rightChild;    // Индекс правого дочернего узла

    // Дружественный класс - имеет доступ к закрытым разделам
    friend class BinaryTree;
}; // Конец класса TreeNode

TreeNode[MAX_NODES] tree; // Массив узлов дерева
int root; // Индекс корня
int free; // Индекс свободного списка
```

Все члены класса *TreeNode* объявлены закрытыми, однако класс *BinaryTree* указан в качестве дружественного класса, поэтому его функции-члены имеют непосредственный доступ ко всем членам класса *TreeNode*. Класс *TreeNode* используется только для реализации класса *BinaryTree*, остальные классы не имеют доступа к его членам.

Переменная *root* является индексом корня дерева в массиве *tree*. Если дерево пусто, значение переменной *root* равно -1. Переменные *leftChild* и *rightChild* в классе *TreeNode* являются индексами дочерних узлов. Если у данного узла нет левого дочернего узла, значение переменной *leftChild* равно -1. Если узел не имеет правого дочернего узла, значение переменной *rightChild* также равно -1.

Свободный список отслеживает доступные узлы

Поскольку изменения в дереве происходят вследствие выполнения операций вставки и удаления узлов, его узлы не обязательно примыкают друг к другу в массиве. Следовательно, при реализации дерева необходимо создать список доступных узлов, называемый **свободным списком** (free list). Чтобы вставить в дерево новый узел, сначала нужно получить доступный узел из свободного списка. Чтобы удалить узел из дерева, его следует поместить в свободный список, получая возможность использовать его в дальнейшем. Переменная *free* является индексом первого узла, содержащегося в свободном списке, а переменная *rightChild*, принадлежащая каждому узлу из этого списка, — индексом следующего доступного элемента.[3] На рис. 10.11 показано бинарное дерево и его реализация в виде массива.

Если переменная *root* в этой реализации бинарного дерева представляет собой индекс его корня, то элемент *tree[root].leftChild* задает индекс корня левого поддерева корня *r*, а *tree[root].rightChild* — индекс правого поддерева корня *r*.

Реализация совершенного дерева в виде массива. Предыдущая реализация ориентировалась на произвольное бинарное дерево, хотя дерево, изображенное на рис. 10.11, является совершенным. Если заранее известно, что бинарное дерево является совершенным, можно использовать более простую и экономную реализацию, основанную на массивах. Как указывалось ранее, совершенное дерево, имеющее высоту h, является полным вплоть до уровня h-1, а на уровне h заполняется слева направо.

На рис. 10.12 показано совершенное бинарное дерево, изображенное ранее на рис. 10.11, *а*. Теперь его узлы пронумерованы по уровням. Корень имеет номер 0, его дочерние узлы (лежащие на следующем уровне) — 1 и 2. Узлы, принадлежащие следующему уровню, нумеруются слева направо, т.е. их номера равны

[3] Иными словами, свободный список представляет собой связанный список, реализованный в виде массива, как показано в задании 9 из главы 4.

3, 4 и 5, соответственно. Эти узлы записываются в массив `tree` в ячейки, соответствующие их номерам. Иными словами, ячейка `tree[i]` содержит число `i`, как показано на рис. 10.13. Теперь по номеру `tree[i]` можно легко определить его дочерние и родительский узлы: индекс левого дочернего узла (если он существует) хранится в ячейке `tree[2*i+1]`, индекс правого дочернего узла (если он существует) — в ячейке `tree[2*i+2]`, а индекс родительского узла (если узел `tree[i]` не является корнем дерева) — в ячейке `tree[(i-1)/2)]`.

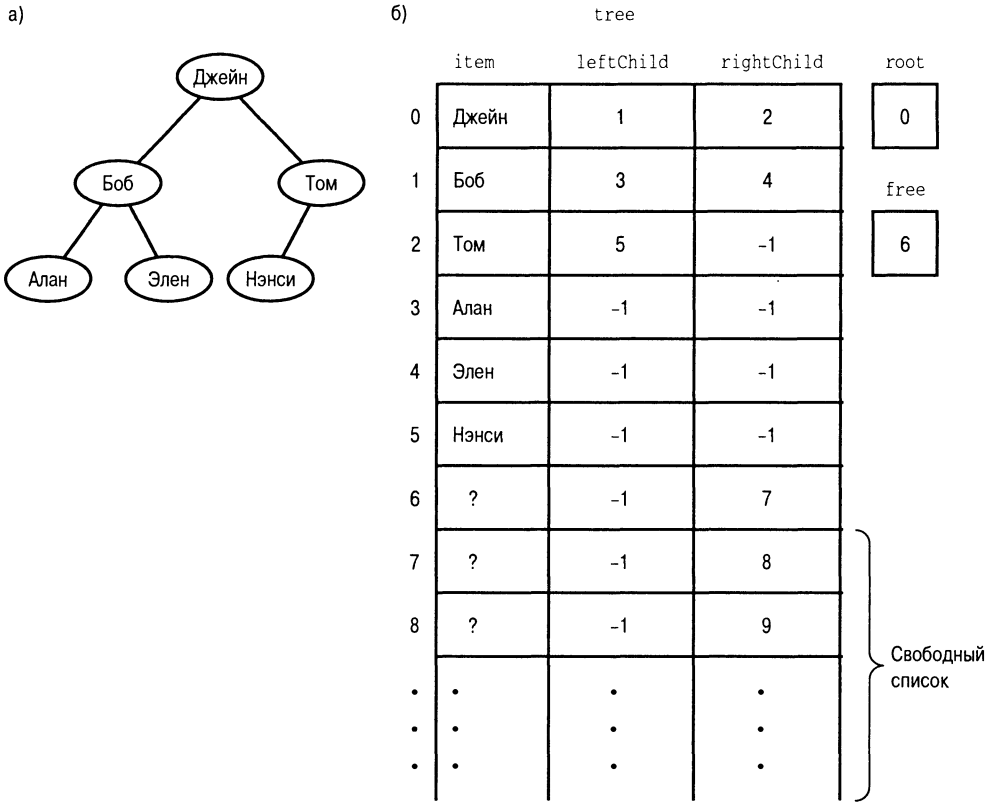


Рис. 10.11. Бинарное дерево и его реализация в виде массива: а) бинарное дерево имен; б) его реализация в виде массива

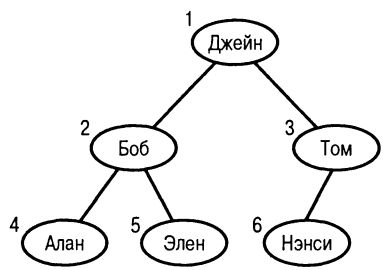


Рис. 10.12. Нумерация совершенного бинарного дерева по уровням

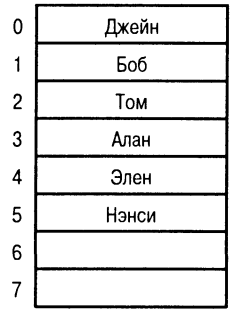


Рис. 10.13. Реализация совершенного бинарного дерева, изображенного на рис. 10.12, в виде массива

В этой реализации необходимо, чтобы бинарное дерево было совершенным. Если в середине дерева пропустить какой-нибудь элемент, схема нумерации нарушится, и отношение "родительский-дочерний" между узлами дерева станет неоднозначным. Следовательно, необходимо, чтобы при любых изменениях дерево оставалось совершенным.

Как мы увидим в следующей главе, представление бинарного дерева в виде массива является весьма полезным инструментом при реализации абстрактной очереди с приоритетами.

Представление бинарного дерева в виде связанного списка. Для связывания узлов дерева можно применить указатели, предусмотренные в языке C++. Таким образом, с помощью следующего фрагмента программы дерево можно представить в виде связанного списка.

```
typedef string TreeItemType;

class TreeNode // Узел дерева
{
private:
   TreeNode() {};
   TreeNode(const TreeItemType& nodeItem,
            TreeNode *left = NULL,
            TreeNode *right = NULL):
   item(nodeItem),leftChildPtr(left),
   rightChildPtr(right) {}
   TreeItemType item; // Данные
   TreeNode *leftChildPtr; // Указатель на левый дочерний узел
   TreeNode *rightChildPtr; // Указатель на правый дочерний узел
   friend class BinaryTree;
}; // Конец класса TreeNode
```

Внешний указатель *root* ссылается на корень дерева. Если дерево пусто, указатель *root* равен константе *NULL*. Эта реализация показана на рис. 10.14.

Корень непустого бинарного дерева имеет левое и правое поддеревья, каждое из которых является бинарным деревом. В реализации, основанной на применении указателей, указатель *root* ссылается на корень бинарного дерева, указатель *root->leftChildPtr* — на корень левого поддерева корня *r*, а указатель *root->rightChildPtr* — на корень правого поддерева корня *r*.

Детали реализации абстрактного бинарного дерева с помощью указателей описаны в следующем разделе.

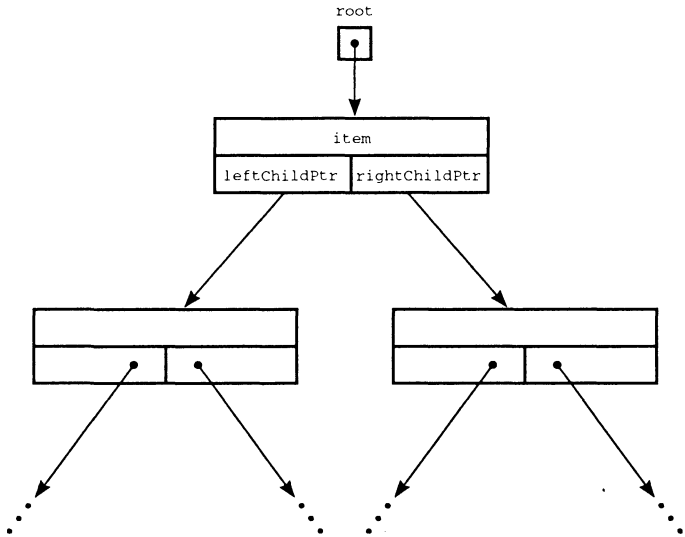


Рис. 10.14. Реализация совершенного бинарного дерева, изображенного на рис. 10.12, в виде связанного списка

Реализация абстрактного бинарного дерева в виде связанного списка

Ниже приведены заголовочный файл и файл реализации абстрактного бинарного дерева в виде связанного списка. Детали, касающиеся этой программы, обсуждаются в дальнейших разделах. Формулировка пред- и постусловий предоставляется читателям в качестве самостоятельного упражнения.

```
// ********************************************************
// Заголовочный файл TreeException.h
// абстрактного бинарного дерева.
// ********************************************************
#include <exception>
#include <string>
using namespace std;

class TreeException : public exception
{
public:
   TreeException(const string & message =""):
      exception(message.c_str()) { }
}; // Конец исключительной ситуации TreeException

// Заголовочный файл

// ********************************************************
// Заголовочный файл BinaryTree.h
// абстрактного бинарного дерева.
// ********************************************************
#include "TreeException.h"
#include "TreeNode.h" // Содержит определения классов TreeNode
                      // и TreeItemType
```

```
typedef void (*FunctionType)(TreeItemType& anItem);

class BinaryTree
{
public:
   // Конструкторы и деструктор:
   BinaryTree();
   BinaryTree(const TreeItemType& rootItem);
   BinaryTree(const TreeItemType& rootItem,
              BinaryTree& leftTree,
              BinaryTree& rightTree);
   BinaryTree(const BinaryTree& tree);
   virtual ~BinaryTree();[4]
// Бинарные операции над деревом:
   virtual bool isEmpty() const;

   virtual TreeItemType getRootData() const
      throw(TreeException);
   virtual void setRootData(const TreeItemType& newItem)
      throw(TreeException);

   virtual void attachLeft(const TreeItemType& newItem)
      throw(TreeException);
   virtual void attachRight(const TreeItemType& newItem)
      throw(TreeException);

   virtual void attachLeftSubtree(BinaryTree& leftTree)
      throw(TreeException);
   virtual void attachRightSubtree(BinaryTree& rightTree)
      throw(TreeException);

   virtual void detachLeftSubtree(BinaryTree& leftTree)
      throw(TreeException);
   virtual void detachRightSubtree(BinaryTree& rightTree)
      throw(TreeException);

   virtual BinaryTree getLeftSubtree() const;
   virtual BinaryTree getRightSubtree() const;

   virtual void preorderTraverse(FunctionType visit);
   virtual void inorderTraverse(FunctionType visit);
   virtual void postorderTraverse(FunctionType visit);

// Перегруженный оператор:
   virtual BinaryTree& operator=(const BinaryTree& rhs);
//
// Защищенный конструктор
//
protected:
   BinaryTree(TreeNode *nodePtr); // constructor
```

[4] Виртуальные функции-члены и защищенные члены обсуждались в главе 8. Если вы не читали эту главу, можете пропустить ключевое слово *virtual* в этой и последующих реализациях. Кроме того, замените ключевое слово *protected* на *private*. Это не повлияет на остальную часть реализации.

```
   void copyTree(TreeNode *treePtr,
                 TreeNode *& newTreePtr) const;
   // Копирует дерево с корнем, на который ссылается указатель
   // treePtr, в дерево, на которое ссылается указатель
   // newTreePtr. Если копирование невозможно, генерируется
   // исключительная ситуация TreeException.

   void destroyTree(TreeNode *& treePtr);
   // Освобождает память, занятую деревом.
   // Следующие две функции извлекают и задают значение
   // закрытого члена корневого узла.

   TreeNode *rootPtr() const;
   void setRootPtr(TreeNode *newRoot);

   // Следующие две функции извлекают и задают значения
   // указателей на левый и правый дочерний узлы
   // корневого узла.

   void getChildPtrs(TreeNode *nodePtr,
                     TreeNode *& leftChildPtr,
                     TreeNode *& rightChildPtr) const;
   void setChildPtrs(TreeNode *nodePtr,
                     TreeNode *leftChildPtr,
                     TreeNode *rightChildPtr);
//
// Защищенные функции-члены, выполняющие рекурсивный обход
//

   void preorder(TreeNode *treePtr, FunctionType visit);
   void inorder(TreeNode *treePtr, FunctionType visit);
   void postorder(TreeNode *treePtr, FunctionType visit);

private:
   TreeNode *root; // Указатель на корень дерева
}; // Конец класса
// Конец заголовочного файла.

// ********************************************************
// Файл реализации BinaryTree.cpp
// абстрактного бинарного дерева.
// ********************************************************
#include "BinaryTree.h" // Заголовочный файл
#include <cstddef> // Определение константы NULL
#include <cassert> // Определение макроса assert()
BinaryTree::BinaryTree() : root(NULL)
{
} // Конец конструктора по умолчанию

BinaryTree::BinaryTree(const TreeItemType& rootItem)
{
   root = new TreeNode(rootItem, NULL, NULL);
   assert(root != NULL);
} // Конец конструктора
```

```
BinaryTree::BinaryTree(const TreeItemType& rootItem,
                       BinaryTree& leftTree,
                       BinaryTree& rightTree)
{
   root = new TreeNode(rootItem, NULL, NULL);
   assert(root != NULL);

   attachLeftSubtree(leftTree);
   attachRightSubtree(rightTree);
} // Конец конструктора

BinaryTree::BinaryTree(const BinaryTree& tree)
{
   copyTree(tree.root, root);
} // end Конец конструктора

BinaryTree::BinaryTree(TreeNode *nodePtr):
   root(nodePtr)
{
} // Конец защищенного конструктора

BinaryTree::~BinaryTree()
{
   destroyTree(root);
} // Конец деструктора

bool BinaryTree::isEmpty() const
{
   return (root == NULL);
} // Конец функции isEmpty

TreeItemType BinaryTree::getRootData() const
{
   if (isEmpty())
       throw TreeException("TreeException: дерево пусто");
       return root->item;
} // Конец функции getRootData

void BinaryTree::setRootData(const TreeItemType& newItem)
{
   if (!isEmpty())
      root->item = newItem;
   else
   {
      root = new TreeNode(newItem, NULL, NULL);
      if (root == NULL)
         throw TreeException(
                  "TreeException: невозможно выделить память");
   } // Конец оператора if
} // Конец функции setRootData

   void BinaryTree::attachLeft(const TreeItemType& newItem)
   {
      if (isEmpty())
        throw TreeException("TreeException: дерево пусто");
      else if (root->leftChildPtr != NULL)
```

```
            throw TreeException(
           "TreeException: переписать левое поддерево невозможно");
         else // Диагностическое утверждение: дерево не пусто;
              // левый дочерний узел отсутствует
         {
            root->leftChildPtr = new TreeNode(newItem, NULL, NULL);
            if (root->leftChildPtr == NULL)
               throw TreeException(
                     "TreeException: невозможно выделить память ");
         } // Конец оператора if
   } // Конец функции attachLeft

   void BinaryTree::attachRight(const TreeItemType& newItem)
   {
      if (isEmpty())
         throw TreeException("TreeException: дерево пусто");
         else if (root->rightChildPtr != NULL)
            throw TreeException(
            "TreeException: невозможно переписать правое поддерево ");
         else // Диагностическое утверждение: дерево не пусто;
              // правый дочерний узел отсутствует
            root->leftChildPtr = new TreeNode(newItem, NULL, NULL);
            if (root->rightChildPtr == NULL)
               throw TreeException(
                     "TreeException: невозможно выделить память");
      } // Конец оператора if
   } // Конец функции attachRight

   void BinaryTree::attachLeftSubtree(BinaryTree& leftTree)
   {
      if (isEmpty())
         throw TreeException("TreeException: дерево пусто");
         {
            else if (root->leftChildPtr != NULL)
               throw TreeException(
               "TreeException: невозможно переписать левое поддерево");
            else // Диагностическое утверждение: дерево не пусто;
                 // левый дочерний узел отсутствует
            {
               root->leftChildPtr = leftTree.root;
               leftTree.root = NULL;
            }
   } // Конец функции attachLeftSubtree

   void BinaryTree::attachRightSubtree(BinaryTree& rightTree)
   {
      if (isEmpty())
         throw TreeException("TreeException: дерево пусто");
         else if (root->rightChildPtr != NULL)
            throw TreeException(
            "TreeException: невозможно переписать левое поддерево ");
         else // Диагностическое утверждение: дерево не пусто;
              // правый дочерний узел отсутствует
         {
            root->rightChildPtr = rightTree.root;
            rightTree.root = NULL;
```

```
      } // Конец оператора if
} // Конец функции attachRightSubtree

void BinaryTree::detachLeftSubtree(BinaryTree& leftTree)
{
   if (isEmpty())
      throw TreeException("TreeException: дерево пусто");
   else
   {
      leftTree = BinaryTree(root->leftChildPtr);
      root->leftChildPtr = NULL;
   } // Конец оператора if
} // Конец функции detachLeftSubtree

void BinaryTree::detachRightSubtree(BinaryTree& rightTree)
{
   if (isEmpty())
      throw TreeException("TreeException: дерево пусто");
   else
   {
      rightTree = BinaryTree(root->rightChildPtr);
      root->rightChildPtr = NULL;
   } // Конец оператора if
} // Конец функции detachRightSubtree

BinaryTree BinaryTree::getLeftSubtree() const
{
   TreeNode *subTreePtr;
   if (isEmpty())
      return BinaryTree();
   else
   {
      copyTree(root->leftChildPtr, subTreePtr);
      return BinaryTree(subTreePtr);
   } // Конец оператора if
} // Конец функции getLeftSubtree

BinaryTree BinaryTree::getRightSubtree() const
{
   TreeNode *subTreePtr;
   if (isEmpty())
      return BinaryTree();
   else
   {
      copyTree(root->rightChildPtr, subTreePtr);
      return BinaryTree(subTreePtr);
   } // Конец оператора if
} // Конец функции getRightSubtree

void BinaryTree::preorderTraverse(FunctionType visit)
{
   preorder(root, visit);
} // Конец функции preorderTraverse
```

```
void BinaryTree::inorderTraverse(FunctionType visit)
{
   inorder(root, visit);
} // Конец функции inorderTraverse

void BinaryTree::postorderTraverse(FunctionType visit)
{
   postorder(root, visit);
} // Конец функции postorderTraverse

BinaryTree& BinaryTree::operator=(const BinaryTree& rhs)
{
   if (this != &rhs)
   {
      destroyTree(root); // Освободить память, занятую
                         // объектом, стоящим слева
      copyTree(rhs.root, root); // Скопировать объект,
                                // стоящий справа
   } // Конец оператора if
   return *this;
} // Конец функции operator=

void BinaryTree::copyTree(TreeNode *treePtr,
                          TreeNode *& newTreePtr) const
{
   // Прямой обход
   if (treePtr != NULL)
   {
       // Копируем узел
       newTreePtr = new TreeNode(treePtr->item, NULL, NULL);

       if (newTreePtr == NULL)
          throw TreeException(
                "TreeException: невозможно выделить память");
       copyTree(treePtr->leftChildPtr, newTreePtr->leftChildPtr);
       copyTree(treePtr->rightChildPtr, newTreePtr->rightChildPtr);
   }
   else
       newTreePtr = NULL; // Копируем пустое дерево
} // Конец функции copyTree

void BinaryTree::destroyTree(TreeNode *& treePtr)
{
   // Обратный обход
   if (treePtr != NULL)
   {
       destroyTree(treePtr->leftChildPtr);
       destroyTree(treePtr->rightChildPtr);
       delete treePtr;
       treePtr = NULL;
   } // Конец оператора if
} // Конец функции destroyTree

TreeNode *BinaryTree::rootPtr() const
{
   return root;
} // Конец функции rootPtr
```

```
void BinaryTree::setRootPtr(TreeNode *newRoot)
{
   root = newRoot;
} // Конец функции setRoot

void BinaryTree::getChildPtrs(TreeNode *nodePtr,
                              TreeNode *& leftPtr,
                              TreeNode *& rightPtr) const
{
   leftPtr = nodePtr->leftChildPtr;
   rightPtr = nodePtr->rightChildPtr;
} // Конец функции getChildPtrs

void BinaryTree::setChildPtrs(TreeNode *nodePtr,
                              TreeNode *leftPtr,
                              TreeNode *rightPtr)
{
   nodePtr->leftChildPtr = leftPtr;
   nodePtr->rightChildPtr = rightPtr;
} // Конец функции setChildPtrs


void BinaryTree::preorder(TreeNode *treePtr,
                          FunctionType visit)
{
   if (treePtr != NULL)
   {
       visit(treePtr->item);
       preorder(treePtr->leftChildPtr, visit);
       preorder(treePtr->rightChildPtr, visit);
    } // Конец оператора if
} // Конец функции preorder

void BinaryTree::inorder(TreeNode *treePtr,
                         FunctionType visit)
{
   if (treePtr != NULL)
   {
       inorder(treePtr->leftChildPtr, visit);
       visit(treePtr->item);
       inorder(treePtr->rightChildPtr, visit);
   } // Конец оператора if
} // Конец функции inorder

void BinaryTree::postorder(TreeNode *treePtr,
                  FunctionType visit)
{
   if (treePtr != NULL)
   {
       postorder(treePtr->leftChildPtr, visit);
       postorder(treePtr->rightChildPtr, visit);
       visit(treePtr->item);
   } // Конец оператора if
} // Конец оператора postorder
// Конец файла реализации.
```

В классе *BinaryTree* предусмотрено больше конструкторов, чем в классах, рассмотренных нами ранее. Это позволяет определить бинарные деревья для самых разных ситуаций.

- Дерево может быть пустым.
- Дерево может состоять лишь из корня.
- Дерево может состоять из корня и двух его поддеревьев.

Например, в приведенном ниже фрагменте вызываются три конструктора.

Пример использования открытых конструкторов

```
BinaryTree tree1;
BinaryTree tree2(root2);
BinaryTree tree3(root3);
BinaryTree tree4(root4, tree2, tree3);
```

Здесь объект *tree1* является пустым бинарным деревом, а объекты *tree2* и *tree3* состоят из одного узла. В их корнях записаны значения переменных *root2* и *root3*, соответственно. Корень бинарного дерева *tree4* содержит значение переменной *root4* и имеет два поддерева: *tree2* и *tree3*. Обратите внимание, что объекты *tree2* и *tree3* являются экземплярами класса *BinaryTree*, а не указателями на дерево.

Некоторые функции-члены не должны быть открытыми

Класс также содержит защищенный конструктор, создающий дерево с помощью указателя на его корень. Например, вызов

```
BinaryTree tree5(nodePtr);
```

создает дерево *tree5*, корень которого является узлом, на который ссылается указатель *nodePtr*. Хотя этот конструктор вызывается функциями-членами *getLeftSubtree* и *getRightSubtree*, он не должен быть доступным клиентам класса, поскольку они не имеют доступа к указателям на узлы дерева. Следовательно, этот конструктор не должен быть открытым. В то же время этот конструктор и не является закрытым, так что производные классы могут его вызывать. Это относится и к другим защищенным функциям-членам класса *BinaryTree*.

Элементы бинарного дерева часто бывают объектами других классов. Некоторые функции-члены класса *BinaryTree* получают их в качестве аргументов. Чтобы избежать копирования этих объектов, которое может затрачивать много времени и памяти, их следует передавать по константной ссылке, а не по значению.

Операции рекурсивного обхода дерева следует реализовывать очень тщательно, чтобы они не нарушили защиту абстрактного типа данных. Например, аргументом функции *indorder*, имеющей объявление

```
void inorder(TreeNode *treePtr, FunctionType visit);
```

является указатель *treePtr*, поочередно ссылающийся на каждый узел дерева. Поскольку этот аргумент очевидным образом зависит от способа реализации дерева, его не следует объявлять открытым. Функция-член *inorder* также не должны быть открытой. В классе функция-член *inorder* объявлена защищенной. Ее вызывает открытая функция-член *inorderTraverse*.

Обход следует реализовывать так, чтобы функция visit оставалась на клиентской стороне от стен, окружающих абстрактный тип данных

Функция *visit* является формальным аргументом функций *inorder* и *inorderTraverse*. Эта функция имеет тип *FunctionType*, который в заголовочном файле определяется так.

```
typedef void (*FunctionType)(TreeItemType& anItem);
```

Обратите внимание, что функция *visit* получает элемент дерева по ссылке. Это позволяет клиенту не только просматривать элемент, но и модифицировать его.

Чтобы вызвать функцию *inorderTraverse*, клиент сначала должен определить функцию, которая будет "посещать" каждый узел дерева. В соответствии с определением класса *FunctionType*, эта функция должна получать аргумент типа *TreeItemType*. Затем клиент передает ее функции *inorderTraverse* в качестве аргумента, соответствующего функции *visit*.

Например, если клиент хочет вывести на экран данные, записанные в узлах дерева, он может написать функцию, имеющую следующее объявление.

```
void display(TreeItemType& anItem);
```

Обход дерева *tree4* в прямом порядке выполняется следующим оператором:

```
tree4.inorderTraverse(display);
```

В заключение отметим, что этот класс содержит виртуальные функции-члены, которые можно перегружать в производных классах, изменяя их смысл, как описано в главе 8.

Чтобы продемонстрировать использование класса *BinaryTree*, построим и обойдем дерево, изображенное на рис. 10.10.

Пример программы

```
#include "BinaryTree.h" // Операции над бинарным деревом
#include <iostream>
using namespace std;

void display(TreeItemType& anItem);

int main()
{
   BinaryTree tree1, tree2, left; // Пустые деревья
   BinaryTree tree3(70); // Деревья, состоящие из корня,
                         // в котором записано число 70
   // Создаем дерево, изображенное на рис. 10.10
   tree1.setRootData(40);
   tree1.attachLeft(30);
   tree1.attachRight(50);

   tree2.setRootData(20);
   tree2.attachLeft(10);
   tree2.attachRightSubtree(tree1);

   // Дерево, изображенное на рис. 10.10
   BinaryTree binTree(60, tree2, tree3);

   binTree.inorderTraverse(display);
   binTree.getLeftSubtree().inorderTraverse(display);
   binTree.detachLeftSubtree(left);
   left.inorderTraverse(display);
   binTree.inorderTraverse(display);
   return 0;
} // Конец функции main
```

Дерево *binTree* показано на рис. 10.10. При его обходе в симметричном порядке на экран выводятся числа: 10, 20, 30, 40, 50, 60, 70. Симметричный обход левого поддерева корня дерева *binTree* (поддерево, корень которого содержит число 20) порождает следующую последовательность чисел: 10, 20, 30, 40, 50. Симметричный обход поддерева *left* приводит к таким же результатам. Поскольку поддерево *left* на самом деле отсоединяется от дерева *binTree*, заключительный обход дерева *binTree* приведет к выводу на экран чисел 60 и 70.

Конструктор копирования и деструктор неявно используют обход дерева. Защищенная функция-член *copyTree*, которую вызывает конструктор, для копирования каждого узла дерева, использует рекурсивный обход в прямом порядке. Копируя, т.е. посещая каждый узел при обходе дерева функция *copyTree* создает точную копию исходного дерева. Аналогично, защищенная функция-член *destroyTree*, вызываемая деструктором, использует рекурсивный обход в обратном порядке, чтобы удалить каждый элемент дерева. Обратный обход хорошо подходит для этой цели, поскольку любой элемент можно удалить лишь после обхода и удаления обоих его поддеревьев.

Итеративный обход. Прежде чем закончить обсуждение способов обхода деревьев, попробуем разработать итеративный алгоритм обхода, чтобы проиллюстрировать отношения между стеками и рекурсией, о которых мы уже говорили в главе 6. В частности, мы разработаем итеративный алгоритм симметричного обхода бинарного дерева, реализованного с помощью указателей.

При разработке итеративного алгоритма обхода принципиально трудной задачей является определение следующего узла, который нужно посетить. Чтобы решить эту задачу, посмотрим, как работает рекурсивная функция *inorder*.

Рекурсивные вызовы из точек 1 и 2

```
void BinaryTree::inorder(TreeNode
*treePtr, FunctionType visit)
{
   if (treePtr != NULL)
   {
      inorder(treePtr->leftChildPtr, visit);  // Точка 1
      visit(treePtr->item);
      inorder(treePtr->rightChildPtr, visit); // Точка 2
   } // Конец оператора if
} // Конец функции inorder
```

Эта функция рекурсивно вызывается из точек 1 и 2.

При выполнении этой функции значение указателя *treePtr* отмечает текущую позицию в дереве. Каждый раз, когда функция *inorder* делает рекурсивный вызов, алгоритм обхода переходит к другому узлу. Это означает, что при каждом вызове функции *inorder* в стек, неявно связанный с рекурсивной функцией, заталкивается новое значение указателя *treePtr*, т.е. указатель на новый текущий узел. В любой фиксированный момент времени стек содержит указатели на узлы, лежащие на пути от корня дерева до текущего узла *n*, причем указатель на узел *n* находится на вершине стека, а указатель на корень дерева — на его дне. Обратите внимание, что узел *n* может быть пустым, т.е. указатель *treePtr*, находящийся на вершине стека, может иметь значение *NULL*.

Чтобы разработать итеративный алгоритм обхода дерева, следует изучить стек, неявно связанный с рекурсивной функцией inorder

На рис. 10.15 показаны результаты частичной трассировки функции *inorder* и содержимое неявного стека. Первые четыре шага трассировки показывают содержимое стека, когда указатель *treePtr* поочередно ссылается на число 60, затем на 20, затем на 10, а затем становится равным константе *NULL*.

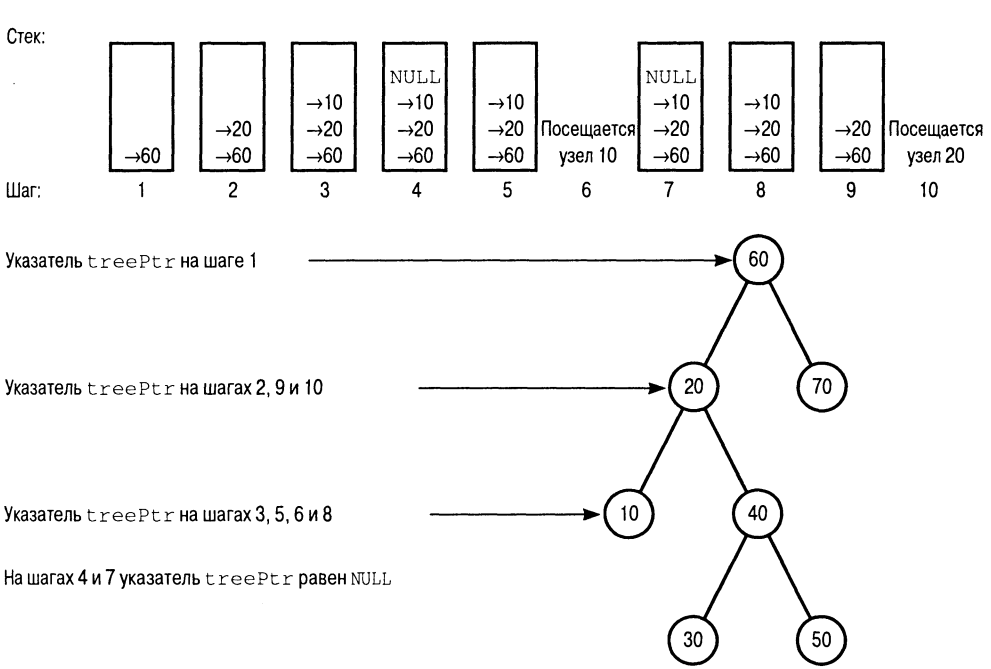
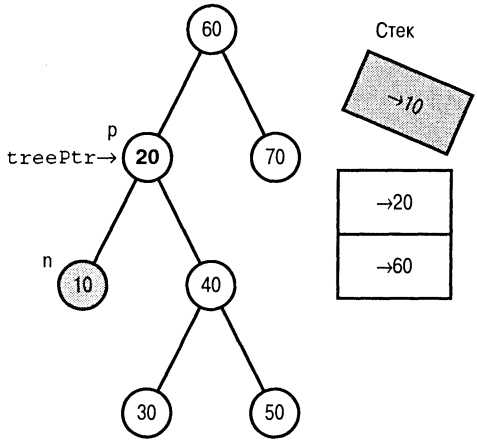


Рис. 10.15. Содержимое неявного стека при рекурсивном симметричном обходе дерева

Рассмотрим теперь, что произойдет, если функция `inorder` вернет управление после рекурсивного вызова. Указатель `treePtr` вернется из узла *n* в его родительский узел *p*, из которого был сделан рекурсивный вызов. При этом указатель на узел *n* выталкивается из стека, а указатель на узел *p* заталкивается на вершину стека. Это соответствует пятому шагу трассировки, результаты которой показаны на рис. 10.15. (В этом случае узел *n* становится пустым, поэтому константа *NULL* выталкивается из стека.)

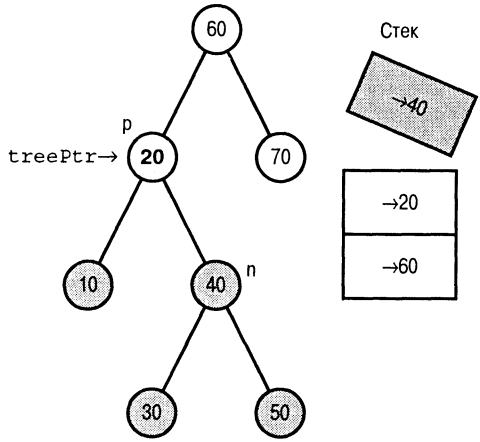
Что произойдет далее, зависит от того, какие поддеревья узла *p* были пройдены до этого момента. Если перед этим было пройдено левое дерево узла *p* (т.е. если узел *n* является левым дочерним узлом узла *p* и, следовательно, возврат управления производится в точку 1 функции `inorder`), то управление будет возвращено оператору, который выводит на экран содержимое узла *p*. Это соответствует шестому и десятому шагам трассировки, результаты которой показаны на рис. 10.15. Рис. 10.16, *а* иллюстрирует шаги 9 и 10 более детально.

После вывода на экран содержимого числа *p* выполняется рекурсивный вызов из точки 2 и обход правого поддерева узла *p*. Однако, как показано на рис. 10.16, *б*, правое поддерево узла *p* уже было пройдено (иными словами, узел *n* является правым дочерним узлом узла *p* и, следовательно, возврат управления производится в точку 2), поэтому управление передается в конец функции. Вследствие этого, указатель на узел *p* выталкивается из стека, и алгоритм обхода возвращается к родителю узла *p*, из которого был сделан предыдущий рекурсивный вызов. В этом случае содержимое узла *p* на экран не выводится — оно уже было выведено перед рекурсивным вызовом, сделанным из точки 2.



а)

Пройдено левое поддерево узла 20.
Ссылка на узел 10 выталкивается из стека.
Посещается узел 20.



б)

Пройдено правое поддерево узла 20.
Ссылка на узел 40 выталкивается из стека.

Рис. 10.16. Обход поддеревьев узла, содержащего число 20: а) обход левого поддерева; б) обход правого поддерева

Итак, анализ рекурсивной версии функции `inorder` позволяет выявить два факта.

Действия, выполняемые после возврата управления из рекурсивных вызовов функции inorder

- Для поиска узла *p*, в который должен перейти алгоритм обхода, используется неявный рекурсивный стек.
- Как только алгоритм вернулся к узлу *p*, он либо посещает его (например, выводит его содержимое на экран), либо пропускает. Узел *p* посещается, если перед этим было пройдено его левое поддерево. Узел *p* пропускается, если перед этим было пройдено его правое поддерево. Выбор действия зависит от того, в какой точке был выполнен рекурсивный вызов — в точке 1 или 2.

Разумеется, можно просто имитировать эти действия, используя итеративную функцию и явный стек, чтобы отслеживать пройденные поддеревья. Однако этого можно избежать. Рассмотрим дерево, изображенное на рис. 10.17. После завершения обхода поддерева корня *R* нет необходимости возвращаться в узлы *C* и *B*, поскольку правое поддерево уже было пройдено. Вместо этого можно вернуться прямо к узлу *A*, ближайшему предку узла *R*, чье правое дерево еще не было пройдено.

Эту стратегию легко реализовать: указатель на узел нужно помещать в стек только после обхода его левого, а не правого поддерева. Итак, вернемся к рис. 10.17. Допустим, что мы находимся в узле *R*, стек содержит указатели на узлы *A* и *R*, причем указатель на узел *R* находится на вершине. Указателей на узлы *B* и *C* в стеке нет, поскольку они уже были пройдены, и в настоящее время выполняется обход их правых поддеревьев. Указатель на узел *A* находится в стеке, поскольку в данный момент выполняется обход его левого поддерева. Вернувшись из узла *R*, следует пропустить узлы *B* и *C*, поскольку их правые поддеревья уже пройдены и нет никакой необходимости возвращаться в эти узлы.

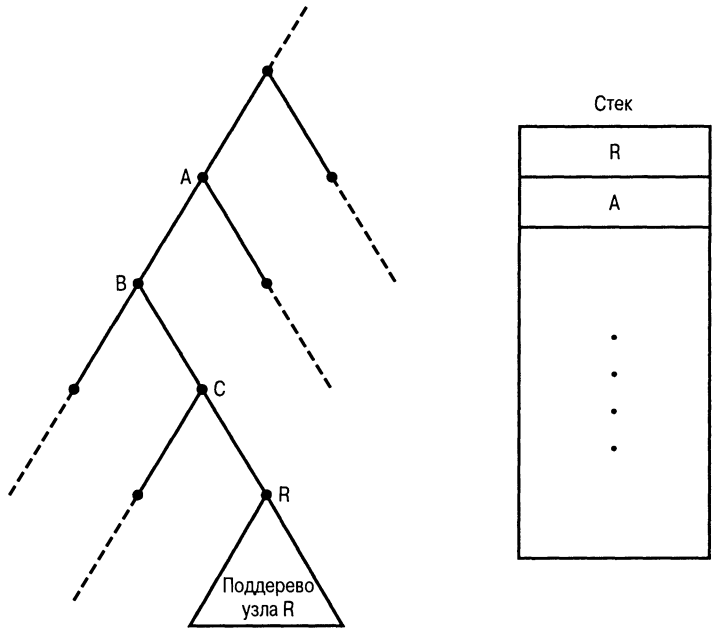


Рис. 10.17. Возврата в узлы B и C можно избежать

Итак, из стека выталкивается указатель на узел *R*, и алгоритм переходит непосредственно к узлу *A*, левое дерево которого только что было пройдено. Затем выполняется посещение узла *A*, указатель на него выталкивается из стека и осуществляется обход его правого поддерева.

Эта итеративная стратегия обхода описывается псевдокодом, приведенным ниже. Предполагается, что дерево реализуется с помощью указателей. Трассировка этого алгоритма на примере дерева, изображенного на рис. 10.15, предоставляется читателям в виде упражнения 14, приведенного в конце главы.

Итеративный симметричный обход

```
traverse(in visit:FunctionType)
// Итеративный симметричный обход
// бинарного дерева.
   // Инициализация
   Создать пустой стек s
   cur = rootPtr() // Обход начинается с корня
   done = false

   while (!done)
   {
      if (cur != NULL)
      {
         // Помещаем указатель на узел в стек перед обходом
         // его левого поддерева
         s.push(cur)

         // Обходим левое поддерево
        cur = cur->leftChildPtr
      }
```

```
    else // Откат от пустого стека. Посещаем узел,
         // на который ссылается вершина стека.
         // Если стек  пуст, обход завершается.
    {
       if (!s.isEmpty())
          {
             s.getTop(cur)
             visit(cur->item)
             s.pop()

             // Обходим правое поддерево
             // узла, посещенного только что
             cur = cur->rightChildPtr
          }

          else
             done = true
       } // Конец отката
    } // Конец оператора while
```

В общем случае избежать применения рекурсии намного сложнее, чем в указанном примере. Однако эта тема выходит за рамки нашей книги.

Абстрактное бинарное дерево поиска

Абстрактное бинарное дерево плохо подходит для поиска конкретного элемента. Этого недостатка лишено бинарное дерево поиска, в котором данные организованы в соответствии с их значениями. Напомним, что каждый узел n бинарного дерева поиска удовлетворяет трем условиям.

- Значение узла n больше всех значений, содержащихся в левом поддереве T_L.
- Значение узла n меньше всех значений, содержащихся в правом поддереве T_R.
- Деревья T_L и T_R являются деревьями бинарного поиска.

Такая организация данных позволяет использовать бинарное дерево поиска для поиска конкретного элемента по его значению, а не по позиции. Как мы убедимся, такой поиск достаточно эффективен.

Бинарное дерево поиска часто оказывается еще более полезным, когда элементами дерева являются экземпляры некоего класса. Например, каждый элемент бинарного дерева поиска может содержать имя человека, его идентификационный номер (ID), адрес, телефонный номер и т.п. Такие элементы называются **записями** (records). Чтобы определить, принадлежит ли дереву конкретная запись, нужно проверить ее компоненты, или **поля** (fields) . Однако обычно для поиска используется только одно поле, например поле ID. Итак, запрос

Найти запись о человеке, имеющем идентификационный номер 123456789

вполне реален, если идентификационный номер уникален. Выполняя этот запрос, можно не только ответить на вопрос, принадлежит ли данная запись указанному дереву, но и получить доступ к другим данным об интересующем нас человеке.

Поле, по которому производится поиск, называется **поисковым ключом** (search key), или просто **ключом**, поскольку оно позволяет идентифицировать искомую запись. Запись, или элемент в дереве может быть экземпляром класса на языке C++.

Данные, хранящиеся в бинарном дереве поиска, содержат специальный поисковый ключ

```
class KeyedItem
{
public:
   KeyedItem() {};
   KeyedItem(const KeyType& keyValue)
      :searchKey(keyValue) { }
   KeyType getKey() const // returns search key
   {
      return searchKey;
   } // Конец функции getKey
private:
   KeyType searchKey;
   ... и возможно другие данные-члены
};
```

Иногда одного поля недостаточно, чтобы точно идентифицировать запись. Например, запрос

Найти запись о Джоне Брауне

трудно выполнить, поскольку дерево может содержать несколько записей о человеке по имени Джон Браун. Этот запрос можно было бы модифицировать, записав

Найти записи обо всех людях по имени Джон Браун

или

Найти запись о Джоне Брауне, имеющем номер телефона 401-555-1212

Для простоты будем считать, что поисковый ключ, состоящий из одного поля, позволяет однозначно идентифицировать запись в бинарном дереве поиска. В этом случае можно дать новую формулировку рекурсивного определения бинарного дерева поиска.

Для каждого узла N бинарное дерево поиска удовлетворяет следующим трем условиям.

Рекурсивное определение бинарного дерева поиска

- Значение поискового ключа узла N больше всех значений поисковых ключей, содержащихся в левом поддереве T_L.
- Значение поискового ключа узла N меньше всех значений поисковых ключей, содержащихся в правом поддереве T_R.
- Деревья T_L и T_R являются деревьями бинарного поиска.

Поскольку бинарное дерево поиска является абстрактным типом данных, на него распространяются все операции вставки, удаления и извлечения элементов, рассмотренные в предыдущих главах. Абстрактные типы данных, изученные нами ранее, также позволяют применять записи в качестве своих элементов. В реализациях позиционно-ориентированных абстрактных списков, стеков и очередей их элементы могут быть объектами, а операции могут применяться без каких-либо модификаций. Поскольку бинарное дерево поиска ориентируется на значения своих элементов, тот факт, что оно может содержать объекты, приобретает особый смысл. Вставка, удаление и извлечение элементов бинарного дерева поиска выполняется не по их позиции, а по значению поискового ключа.

Операции обхода бинарного дерева применяются к бинарному дереву поиска без модификаций, поскольку бинарное дерево поиска является разновидностью бинарного дерева.

ОСНОВНЫЕ ПОНЯТИЯ

Операции над бинарным деревом поиска

1. Создать пустое бинарное дерево поиска.
2. Уничтожить бинарное дерево поиска.
3. Определить, пусто ли бинарное дерево.
4. Вставить новый элемент в бинарное дерево поиска.
5. Удалить элемент из бинарного дерева поиска по заданному ключу.
6. Извлечь элемент из бинарного дерева поиска по заданному ключу.
7. Обойти узлы бинарного дерева поиска в прямом, симметричном или обратном порядке.

Детали этих операций уточняются в приведенном ниже псевдокоде. UML-диаграмма класса бинарных деревьев поиска показана на рис. 10.18.

На рис. 10.19 показано бинарное дерево поиска *nameTree*, содержащее имена. Каждый узел в этом дереве представляет собой запись, в которой хранится имя человека. Если поисковым ключом является имя, то на экран будет выведено лишь оно одно.

Например, операция

```
nameTree.searchTreeRetrieve("Нэнси", nameRecord)
```

извлекает в переменную *nameRecord* запись о Нэнси. Если с помощью операции

```
nameTree.searchTreeInsert(HalRecord)
```

в бинарное дерево поиска вставить запись, описывающую Хэла, позднее обе записи о Нэнси и Хэле можно будет извлечь обратно. Если из бинарного дерева поиска удалить запись о Джейн с помощью операции

```
nameTree.searchTreeDelete("Джейн")
```

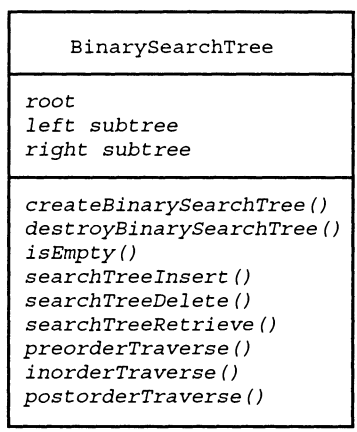


Рис. 10.18. UML-диаграмма класса BinarySearchTree

то записи о Нэнси и Хэле останутся доступными. В заключение отметим функцию *displayName*, выводящую имя из записи. Например, операция

```
nameTree.inorderTraverse(displayName)
```

выводит на экран имена людей, представленных в объекте *nameTree* в алфавитном порядке.

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операций над абстрактным бинарным деревом поиска

```
// TreeItemType — это тип элементов, записанных в бинарном
// дереве поиска. Он должен быть производным от типа KeyedItem,
// содержащего поле поискового ключа, имеющее тип KeyType.

+createSearchTree()
// Создает пустое бинарное дерево поиска.

+destroySearchTree()
// Уничтожает бинарное дерево поиска.

+isEmpty():boolean {query}
// Определяет, пусто ли бинарное дерево поиска.

+searchTreeInsert(in newItem:TreeItemType) throw TreeException
// Вставляет в бинарное дерево поиска элемент newItem.
// Поисковые ключи остальных элементов дерева должны отличаться
// от поискового ключа элемента newItem. Если вставка невозможна, //
генерируется исключительная ситуация TreeException.

+searchTreeDelete(in searchKey:KeyType) throws TreeException
// Удаляет из бинарного дерева поиска элемент, поисковый ключ
// которого совпадает со значением переменной searchKey. Если
// такого элемента нет, генерируется исключительная ситуация
// TreeException.

+searchTreeRetrieve(in searchKey:KeyType,
out treeItem:TreeItemType) throw TreeException
// Извлекает из бинарного дерева поиска элемент, поисковый ключ
// которого совпадает со значением переменной searchKey. Если
// такого элемента нет, генерируется исключительная ситуация
// TreeException.

+preorderTraverse(in visit:FunctionType)
// Выполняет прямой обход бинарного дерева поиска, и для каждого
// узла один раз вызывает функцию visit().

+inorderTraverse(in visit:FunctionType)
// Выполняет симметричный обход бинарного дерева поиска,
// и для каждого узла один раз вызывает функцию visit().

+postorderTraverse(in visit:FunctionType)
// Выполняет обратный обход бинарного дерева поиска,
// и для каждого узла один раз вызывает функцию visit().
```

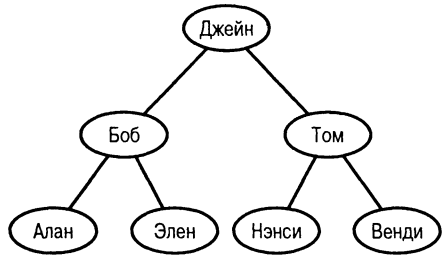


Рис. 10.19. Бинарное дерево поиска

Алгоритмы, реализующие операции над абстрактным бинарным деревом поиска

Рассмотрим еще раз бинарное дерево поиска, изображенное на рис. 10.19. Каждый узел этого дерева содержит данные о конкретном человеке. Поисковым ключом является имя, поэтому на рисунке показаны только имена. Узел дерева описывается следующим классом на языке C++.

```
#include <string>
using namespace std;

typedef string KeyType;

class KeyedItem
{
public:
   KeyedItem() {};
   KeyedItem(const KeyType& keyValue)
      :searchKey(keyValue) { }
   KeyType getKey() const
{
   return searchKey;
} // Конец функции getKey

private:
   KeyType searchKey;
}; // Конец класса

// Элементы бинарного дерева поиска могут быть
// экземплярами некоего класса

class Person : public KeyedItem
{
public:
   Person() {}
   Person(const string& name,
          const string& id,
          const string& phone):
      KeyedItem(name), idNum(id),
      phoneNumber(phone) { }
```

```
private:
   // Поисковым ключом является имя человека
   string idNum;
   string phoneNumber;
   //... и другие данные о человеке
}; // Конец класса
```

Поскольку бинарное дерево поиска рекурсивно по своей природе, естественно сформулировать рекурсивные алгоритмы, выполняющие операции над деревом. Допустим, в бинарном дереве поиска необходимо найти запись об Элен (см. рис. 10.19). Корень дерева содержит информацию о Джейн, поэтому, если запись об Элен содержится в дереве, она должна принадлежать левому поддереву корня, поскольку поисковые ключи упорядочиваются по алфавиту. В соответствии с рекурсивным определением, левое поддерево узла, содержащего запись о Джейн, само является бинарным деревом поиска, поэтому к нему можно применить ту же самую стратегию поиска записи об Элен. В корне этого бинарного дерева поиска содержится запись о Бобе. Поскольку поисковый ключ Элен больше поискового ключа Боба, запись об Элен должна находиться в правом поддереве узла, содержащего запись о Бобе. Это правое поддерево также является бинарным деревом поиска, причем его корень содержит запись об Элен. Итак, искомая запись найдена.

Описанная выше стратегия реализуется следующим псевдокодом.

Алгоритм поиска в бинарном дереве поиска

```
search(in binTree:BinarySearchTree,
       in searchKey:KeyType)
// Поиск элемента, имеющего поисковый ключ searchKey
// в бинарном дереве поиска binTree.

   if (дерево binTree пусто)
      Искомый элемент не найден

   else if (searchKey == поисковый ключ корня)
      Искомый элемент найден

   else if (searchKey < поисковый ключ корня)
      search(левое поддерево дерева binTree, searchKey)
   else
      search(правое поддерево дерева binTree, searchKey)
```

Имена Алан, Боб, Элен, Джейн, Нэнси, Том и Венди могут содержаться в разных бинарных деревьях поиска. Например, кроме дерева, изображенного на рис. 10.19, каждое из деревьев, представленных на рис. 10.20, также является вполне корректным бинарным деревом поиска. Хотя эти деревья имеют разную форму, это не влияет на корректность алгоритма *search*. В этом алгоритме важно лишь, чтобы дерево было бинарным деревом поиска, а его форма не имеет значения.

Одним и тем же данным могут соответствовать разные бинарные деревья поиска

Заметим, однако, что алгоритм *search* для некоторых конкретных видов деревьев оказывается более эффективным. Например, прежде, чем найти узел, содержащий запись о Венди, алгоритм *search* перебирает все узлы дерева, изображенного на рис. 10.20, *в*. Фактически это бинарное дерево по своей структуре не отличается от линейного упорядоченного связанного списка и не дает выигрыша в эффективности поиска. В противоположность ему, в полном дереве, представленном на рис. 10.19, алгоритм *search* просматривает лишь узлы, со-

держащие имена Джейн, Том и Венди. Эти имена в точности совпадают с именами, просматриваемыми в упорядоченном массиве, изображенном на рис. 10.21. Позднее мы более подробно изучим вопрос, какие виды бинарных деревьев поиска обеспечивают более высокую эффективность поиска и как это влияет на операции вставки и удаления элементов.

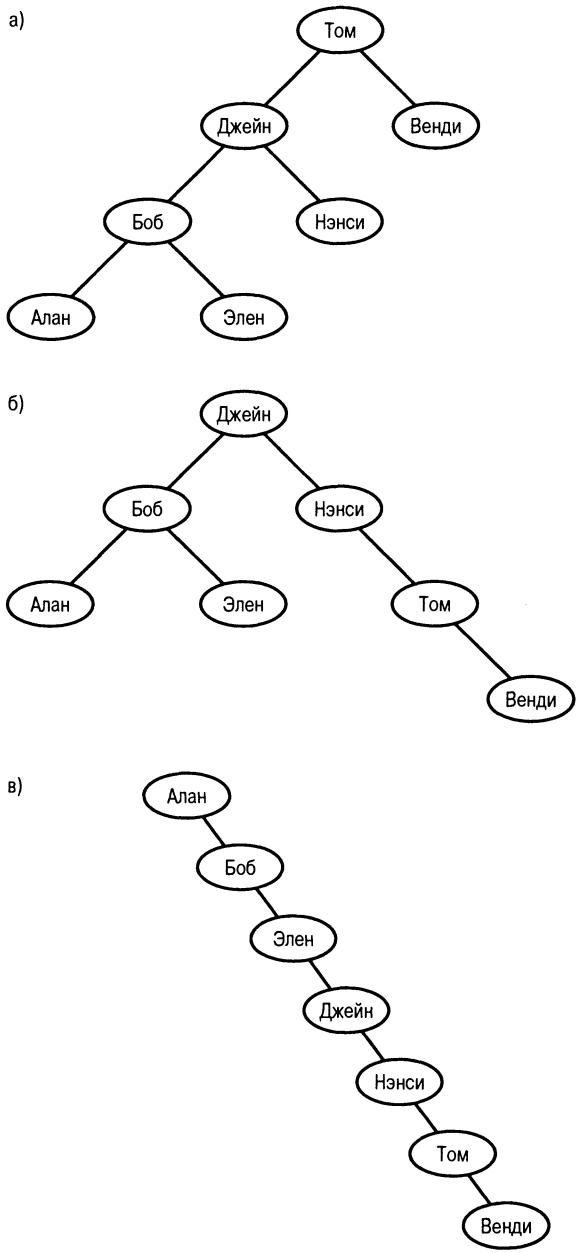


Рис. 10.20. Бинарные деревья поиска, содержащие данные, показанные на рис. 10.19

Алан	Боб	Элен	Джейн	Нэнси	Том	Венди
0	1	2	3	4	5	6

Рис. 10.21. Упорядоченный массив имен

Алгоритмы, реализующие операции вставки, удаления, извлечения и обхода, основанные на реализации бинарного дерева с помощью указателей, были рассмотрены нами в предыдущих разделах. С небольшими изменениями их можно применять и к другим видам бинарных деревьев. При этом следует иметь в виду, что поисковый ключ должен быть уникальным.

Вставка. Допустим, нам нужно вставить запись о Фрэнке в бинарное дерево поиска, изображенное на рис. 10.19. Для начала представьте себе, что нам нужно *найти* узел, содержащий запись о Фрэнке. Алгоритм `search` сначала проверяет дерево, корнем которого является запись о Джейн, затем — дерево, корнем которого является запись о Бобе, а потом — дерево, корнем которого является запись об Элен. Затем он проверяет правое поддерево узла, содержащего запись об Элен. Поскольку это дерево пусто, как показано на рис. 10.22, алгоритм `search` достигает базиса и сообщает, что узла, содержащего запись о Фрэнке, в дереве нет. Почему алгоритм `search` просматривает правое поддерево узла, содержащего запись об Элен? Если бы узел, содержащий запись о Фрэнке, был правым дочерним узлом узла, содержащего запись об Элен, он был бы найден.

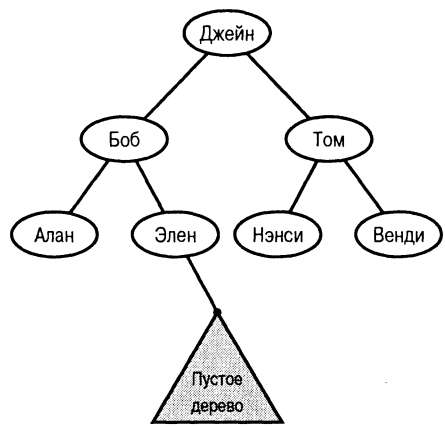


Рис. 10.22. Пустое поддерево, на котором останавливается алгоритм search

Это наблюдение позволяет выбрать подходящее место для вставки узла, содержащего запись о Фрэнке. Он должен быть правым дочерним узлом по отношению к записи об Элен. Поскольку у этой записи нет правых дочерних узлов, вставка не представляет никаких трудностей. Для этого достаточно, чтобы указатель `rightChildPtr` в узле, содержащем запись об Элен, ссылался на узел, содержащий запись о Фрэнке. Очень важно, что алгоритм `search` будет искать его именно в этом месте. Вставка записи о Фрэнке именно в это место не нарушает свойства бинарного дерева поиска. Поскольку алгоритм `search` будет просматривать правый дочерний узел записи об Элен, это самое подходящее место для записи о Фрэнке.

Использование алгоритма search для определения места вставки

Использование алгоритма `search` для определения места вставки намного облегчает эту операцию. Независимо от значения нового эле-

мента, алгоритм *search* всегда найдет для него свободное место, остановившись на пустом поддереве. Итак, алгоритм *search* предлагает вставлять новый элемент в качестве нового листа дерева. Поскольку для добавления листа достаточно изменить значение указателя в его родительском узле, сложность вставки практически совпадает со сложностью поиска.

Рассмотрим псевдокод, описывающий процесс вставки.

Первое приближение алгоритма вставки

```
insertItem(in treePtr:TreeNodePtr,
           in newItem:TreeItemType)
// Вставка элемента newItem в бинарное дерево поиска,
// на которое ссылается указатель treePtr.

   Будем считать, что объект parentNode является родительским
   узлом пустого поддерева, на котором останавливается
   алгоритм search при поиске элемента newItem

   if (алгоритм search остановился на родительском узле
       левого поддерева)
      Установить указатель leftChildPtr на узел parentNode
         на элемент newItem
   else
      Установить указатель rightChildPtr в узле parentNode
         на элемент newItem
```

Соответствующий указатель узла *parentNode* — *leftChildPtr* или *rightChildPtr* — должен быть установлен на новый узел. Рекурсивная природа алгоритма *search* позволяет элегантно изменять значения этих указателей, передавая указатель *treePtr* по ссылке. Итак, псевдокод можно уточнить следующим образом.

Уточненный алгоритм вставки

```
insertItem(inout treePtr:TreeNodePtr,
           in newItem:TreeItemType)
// Вставка элемента newItem в бинарное дерево поиска,
// на которое ссылается указатель treePtr.

   if (указатель treePtr равен NULL)
   {
     Создать новый узел и установить на него указатель treePtr
     Скопировать элемент newItem в новый узел
     Присвоить указателям нового узла константу NULL
   }

   else if (newItem.getKey() < treePtr->item.getKey())
      insertItem(treePtr->leftChildPtr, newItem)

   else
      insertItem(treePtr->rightChildPtr, newItem)
```

Как этот рекурсивный алгоритм устанавливает указатели *leftChildPtr* и *rightChildPtr* на новый узел? Ситуация аналогична рекурсивной вставке в упорядоченный связанный список, описанной в главе 4. Если перед вставкой дерево пусто, внешний указатель на его корень должен быть равным константе *NULL*, и функция не должна выполнять рекурсивный вызов. Поскольку указатель *treePtr*, ссылающийся на новый узел, передается по ссылке, фактический аргумент — внешний указатель на корень дерева — также ссылается на новый узел. Вставка в пустое дерево показана на рис. 10.23, *а*.

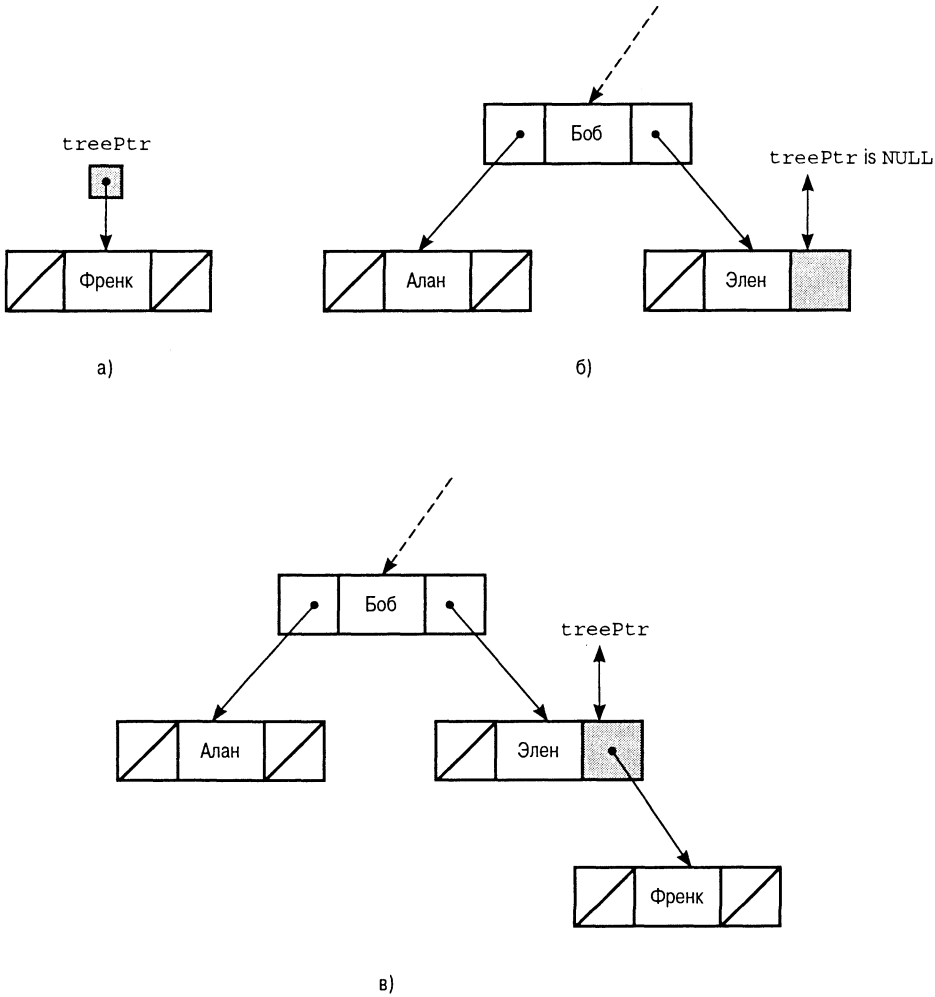


Рис. 10.23. Вставка узла в дерево: а) вставка в пустое дерево; б) поиск прекращается на листе; в) вставка листа

В общем случае функция `insertItem` работает аналогично. Когда формальный аргумент `treePtr` становится равным константе `NULL`, соответствующий фактический аргумент является указателем `leftChildPtr` или `rightChildPtr` в родительском узле пустого поддерева. Иными словами, значение этого указателя равно константе `NULL`. Указатель передается функции `insertItem` с помощью одного из следующих рекурсивных вызовов:

```
insertItem(treePtr->leftChildPtr, newItem)
```

или

```
insertItem(treePtr->rightChildPtr, newItem)
```

Итак, когда указатель `treePtr` ссылается на новый узел, фактический аргумент, — соответствующий указатель родительского узла — также ссылается на новый узел. Общий случай вставки показан на рис. 10.23, *б* и 10.23, *в*.

Функцию *insertItem* можно использовать для создания бинарного дерева поиска, например, начиная с пустого дерева, можно вставлять в него имена Алан, Боб, Элен, Джейн, Нэнси, Том и Венди в порядке, показанном на рис. 10.19. Интересно, что эти имена соответствуют прямому порядку обхода дерева, представленного на рис. 10.19. Итак, если выбрать прямой порядок обхода и применить функцию *insertItem* для создания бинарного дерева поиска, можно создать его копию. Это не удивительно, поскольку в конструкторе копирования абстрактного дерева используется именно прямой обход.

Чтобы скопировать дерево, нужно выполнить прямой или симметричный обход

Вставляя имена в другом порядке, можно создать другое бинарное дерево поиска. Например, вставляя указанные выше имена в алфавитном порядке, можно получить бинарное дерево поиска, показанное на рис. 10.20, *в*.

Удаление. Операция удаления немного сложнее, чем операция вставки. Сначала для поиска удаляемого элемента по его ключу применяется алгоритм *search*. Затем, если искомый элемент найден, он удаляется из дерева. Первое приближение алгоритма имеет следующий вид.

Первое приближение алгоритма удаления

```
deleteItem(inout treePtr:TreeNodePtr,
           in searchKey:KeyType)
           throw TreeException
// Удаляет из бинарного дерева поиска, на которое ссылается
// указатель treePtr, элемент, поисковый ключ которого
// совпадает со значением аргумента searchKey. Если такого
// элемента в дереве нет, генерируется исключительная ситуация
// TreeException.

   Найти (используя алгоритм search) элемент i,
      поисковый ключ которого равен значению searchKey

   if (элемент i найден)
      Удалить элемент i из дерева
   else
      Генерировать исключительную ситуацию TreeException
```

Основная работа здесь выполняется оператором

```
Удалить элемент i из дерева
```

Предположим, что функция *deleteItem* обнаруживает элемент *i* в конкретном узле *N* и рассмотрим три следующих варианта.

Три варианта удаления узла N

- Узел *N* является листом.
- Узел *N* имеет только один дочерний узел.
- Узел *N* имеет два дочерних узла.

Первый вариант проще всех. Чтобы удалить лист, содержащий элемент *i*, достаточно присвоить указателю его родительского узла константу *NULL*. Второй вариант немного сложнее. Если узел *N* имеет только один дочерний узел, возникают две возможности.

Вариант 1: присвоить указателю в родительском узле листа константу NULL

- Узел *N* имеет только левый дочерний узел.
- Узел *N* имеет только правый дочерний узел.

Вариант 2: две возможности, возникающие, если узел N имеет только один дочерний узел

Родитель узла N усыновляет его дочерние узлы

Эти две возможности симметричны, поэтому достаточно рассмотреть первую из них. На рис. 10.24, *а* узел *L* является левым дочерним узлом по отношению к узлу *N*, а узел *P* — родителем узла *N*. Узел *N*, в свою очередь, может быть левым или правым дочерним узлом по отношению к узлу *P*. Если удалить его из дерева, узел *L* останется без родителя, а узел *P* — без одного из своих дочерних узлов. Допустим, что узел *L* заменяет собой узел *N* и становится дочерним по отношению к узлу *P*, как показано на рис. 10.24, *б*. Сохранятся ли при этом свойства бинарного дерева поиска?

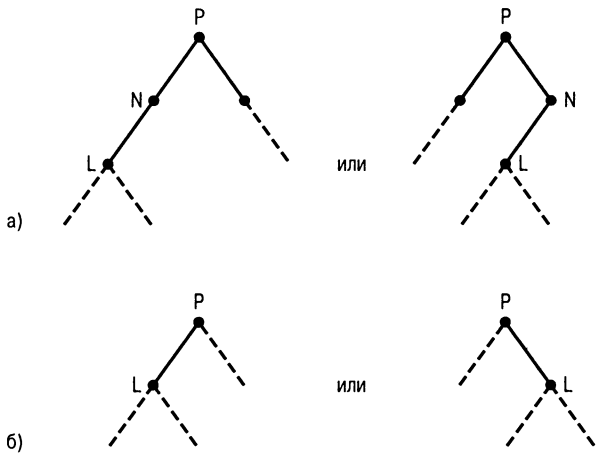


Рис. 10.24. Удаление узла: а) узел N имеет только левый дочерний узел — узел N может быть левым или правым дочерним узлом по отношению к узлу P; б) после удаления узла N

Например, если узел *N* является левым дочерним узлом по отношению к узлу *P*, все поисковые ключи в поддереве, корнем которого является узел *N*, меньше, чем поисковый ключ узла *P*. Таким образом, все поисковые ключи в поддереве, корнем которого является узел *L*, также меньше поискового ключа узла *P*. Следовательно, после удаления узла *N* и усыновления узла *L* узлом *P* все поисковые ключи левого поддерева узла *P* остаются меньше поискового ключа узла *P*. Эта стратегия удаления позволяет сохранить свойства бинарного дерева поиска. Если узел *N* является правым дочерним узлом по отношению к узлу *P*, все рассуждения остаются в силе, и, следовательно, свойства бинарного дерева поиска сохраняются в любом случае.

Вариант 3: узел N имеет два дочерних узла

Наиболее трудная ситуация возникает, когда удаляемый элемент принадлежит узлу *N*, имеющему два дочерних узла, как показано на рис. 10.25. Если бы узел *N* имел один дочерний узел, то его можно было бы заменить этим узлом. Однако когда узел *N* имеет два дочерних узла, они не могут оба заменить узел *N*: место есть только для одного из них. Очевидно, здесь нужно применить другую стратегию.

Фактически узел *N* в этом случае можно совсем не удалять. Можно найти другой узел, который легче удалить, чем узел *N*, и удалить его вместо узла *N*. Это уже похоже на мошенничество! Ведь программист, написавший оператор

```
nameTree.searchTreeDelete(searchkey);
```

ожидает, что из абстрактного бинарного дерева поиска будет удален элемент, поисковый ключ которого совпадает со значение аргумента `searchKey`. Однако обратите внимание, что программист хочет удалить лишь *элемент* дерева, а не *узел*, который защищен стеной, возведенной вокруг реализации абстрактного типа данных.

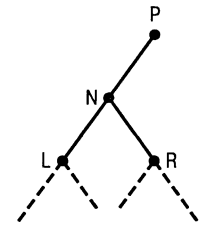


Рис. 10.25. Узел N имеет два дочерних узла

Рассмотрим альтернативную стратегию. Чтобы удалить из бинарного дерева поиска элемент, хранящийся в узле *N*, имеющем два дочерних узла, нужно выполнить следующие действия.

Удаление элемента, находящегося в узле, имеющем два дочерних узла

1. Найти другой узел *M*, который легче удалить, чем узел *N*.
2. Скопировать элемент узла *M* в узел *N*, тем самым удаляя из дерева данные, хранившиеся в узле *N*.
3. Удалить узел *M* из дерева.

Какие элементы легче удалить, чем узел *N*? Узел *M* можно выбрать среди элементов, имеющих не больше одного дочернего узла. Однако при этом следует проявлять осторожность. Можно ли выбрать произвольный узел и скопировать его данные в узел *N*? Нет, поскольку при этом должны сохраняться свойства бинарного дерева поиска. Например, если в дереве, изображенном на рис. 10.26, данные, содержащиеся в узле *M*, скопировать в узел *N*, оно перестанет быть бинарным деревом поиска.

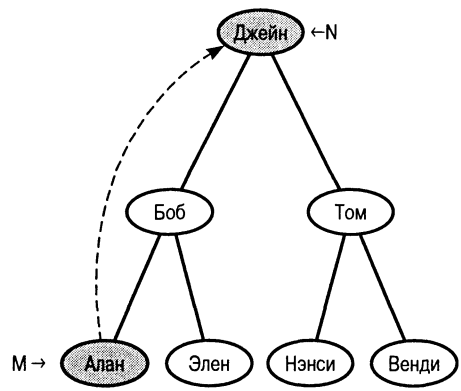


Рис. 10.26. Для удаления подходит не всякий узел

Какой узел выбрать, чтобы при копировании его данных в узел *N* сохранялись свойства бинарного дерева? Как известно, все поисковые ключи левого поддерева узла *N* меньше поискового ключа узла *N*, а все поисковые ключи правого поддерева узла *N* больше поискового ключа узла *N*. Заменяя поисковый ключ *x* узла *N* новым поисковым ключом *y*, мы не должны нарушить это условие. Существуют две возможности выбрать поисковый ключ *y*: можно выбрать поисковый ключ, непосредственно следующий за ключом *x* или непосредственно предшествующий ему по величине. Если ключ *y* следует сразу за ключом *x*, то очевидно, что все поисковые ключи левого поддерева узла *N* меньше *y*, поскольку все они

меньше x, как показано на рис. 10.27. Далее, все поисковые ключи правого поддерева узла N больше или равны y, поскольку все они больше x и, в соответствии с предположением, между ключами x и y других ключей нет. Аналогичными рассуждениями можно доказать, что если ключ y непосредственно предшествует ключу x, то он не меньше всех поисковых ключей в левом поддереве узла N и меньше всех поисковых ключей в правом поддереве этого узла.

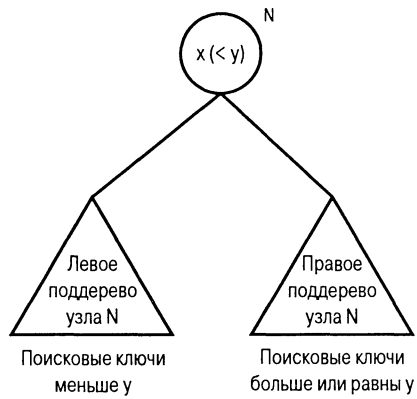


Рис. 10.27. Поисковый ключ x можно заменить ключом y

Симметричным преемником поискового ключа узла N является самый левый узел в правом поддереве узла N

Таким образом, в узел N можно копировать данные из узлов, поисковые ключи которых либо непосредственно предшествуют[5] поисковому ключу узла N, либо непосредственно следуют за ним. Допустим, что мы выбрали ключ y, непосредственно следующий за поисковым ключом x узла N. Этот поисковый ключ называется **симметричным преемником** (inorder successor).[6] Как найти этот узел? Поскольку узел N имеет два дочерних узла, симметричный преемник поискового ключа является самым левым узлом в правом поддереве узла N. Иными словами, чтобы найти узел, содержащий ключ y, нужно проследовать за указателем `rightChildPtr` узла N к его дочернему узлу R, который обязательно существует, поскольку узел N имеет два дочерних узла. Затем нужно спуститься вниз по дереву, корнем которого является узел R, придерживаясь левых ветвей, пока не обнаружится узел M, у которого нет левого дочернего узла. Элемент, содержащийся в узле M, копируется в узел N, а затем узел M можно удалить, поскольку у него нет левого дочернего узла (рис. 10.28).

Более детальное описание алгоритма удаления приведено в следующем псевдокоде.

Второе приближение алгоритма удаления

```
deleteItem(inout treePtr:TreeNodePtr,
           in searchKey:KeyType)
           throw TreeException
// Удаляет из бинарного дерева поиска, на которое ссылается
// указатель treePtr, элемент, поисковый ключ которого
// совпадает со значением аргумента searchKey. Если такого
```

[5] Поисковый ключ узла N является поисковым ключом элемента, содержащегося в узле N.

[6] Мы также используем термин *симметричный преемник узла N*, подчеркивая, что выбирается симметричный преемник поискового ключа элемента, содержащегося в узле N.

```
// элемента в дереве нет, генерируется исключительная ситуация
// TreeException.

   Найти (используя алгоритм search) элемент i,
      поисковый ключ которого равен значению searchKey

   if (элемент i найден)
      Удалить элемент i из дерева
   else
      Генерировать исключительную ситуацию TreeException

deleteNodeItem(inout N:TreeNode)
// Удаляет элемент, содержащийся в узле N, из бинарного
// дерева поиска
   if (узел N является листом)
      Удалить узел N из дерева
   else if (узел N имеет только один дочерний узел C)
   {
       if (узел N является левым дочерним узлом узла P)
          Сделать узел C левым дочерним узлом узла P
       else
          Сделать узел C правым дочерним узлом узла P
   }
   else // Узел имеет два дочерних узла
   {
      Найти узел M, содержащий симметричного преемника узла N
      Скопировать элемент, содержащийся в узле M, в узел N
      Удалить узел M из дерева, используя алгоритм удаления
      листа или узла, имеющего только один дочерний узел
   }
// Конец оператора if
```

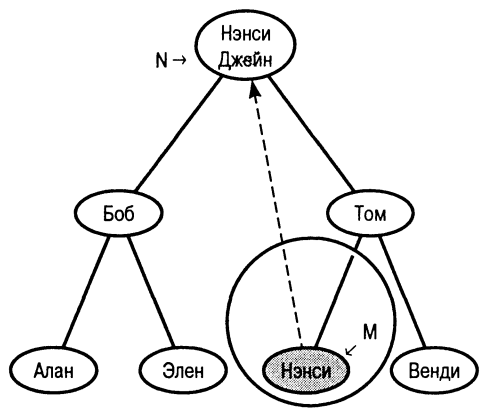


Рис. 10.28. Копирование элемента, поисковый ключ которого является симметричным преемником поискового ключа узла N

В процессе уточнения алгоритм *search* включается непосредственно в алгоритм *deleteItem*. Кроме того, функция *deleteItem* использует функцию *processLeftMost* для поиска узла *M*, содержащего симметричного преемника узла

N. Функция *processLeftMost* возвращает элемент, содержащийся в узле *M*, а затем удаляет этот узел из дерева. Возвращаемый элемент заменяет собой элемент, содержащийся в узле *N*, тем самым удаляя его из бинарного дерева поиска.

Окончательная версия алгоритма удаления

```
deleteItem(inout treePtr:TreeNodePtr,
           in searchKey:KeyType)
           throw TreeException
// Удаляет из бинарного дерева поиска, на которое ссылается
// указатель treePtr, элемент, поисковый ключ которого
// совпадает со значением аргумента searchKey. Если такого
// элемента в дереве нет, генерируется исключительная ситуация
// TreeException.

   if (treePtr == NULL)
      throw TreeException // Элемент не найден
   else if (searchKey == treePtr->item.getKey())
      // Элемент находится в корне поддерева
      deleteNodeItem(treePtr) // Удалить элемент

   else if (searchKey < treePtr->item.getKey())
      // Поиск левого поддерева
      deleteItem(treePtr->leftChildPtr, searchKey)

   else // Поиск правого поддерева
      deleteItem(treePtr->rightChildPtr, searchKey)

deleteNodeItem(inout nodePtr:TreeNodePtr)
// Удаляет элемент из узла N, на который ссылается
// указатель nodePtr.

   if (если узел N является листом)
   {
      // Удаляем лист из дерева
      delete nodePtr
      nodePtr = NULL
   }
   else if (узел N имеет только один дочерний узел C)
   {
      // Заменяем узел N узлом C, делая его дочерним узлом
      // родителя узла N
      delPtr = nodePtr
      if (узел C является левым дочерним узлом узла N)
         nodePtr = nodePtr->leftChildPtr
      else
         nodePtr = nodePtr->rightChildPtr
      delete delPtr
   }
   else // Узел N не имеет дочерних узлов
   {
      // Ищем симметричного преемника поискового ключа
      // узла N. Им является самый левый узел правого
      // поддерева узла N

      Поместить элемент replacementItem в узел N
   } // Конец оператора if
```

```
processLeftmost(inout nodePtr:TreeNodePtr,
                out treeItem:TreeItemType)
// Извлекаем и присваиваем переменной treeItem
// элемент самого левого потомка узла, на который
// ссылается указатель nodePtr.
// Удаляем этот узел.
if (nodePtr->leftChildPtr == NULL)
{
   // Искомый узел найден; у него нет левого
   // дочернего узла, но может существовать
   // правое поддерево
   treeItem = nodePtr->item
   delPtr = nodePtr

   // Фактический аргумент, соответствующий указателю
   // nodePtr, является указателем на дочерний узел
   // узла, на который ссылается указатель nodePtr.
   // Следовательно, оператор, приведенный ниже,
   // "удаляет" правое поддерево.
   nodePtr = nodePtr->rightChildPtr
   delete delPtr
}
else
processLeftmost(nodePtr->leftChildPtr, treeItem)
```

Обратите внимание, что, как и в функции *insertItem*, фактический аргумент, соответствующий указателю *treePtr*, либо является указателем на родителя узла *N*, как показано на рис. 10.29, либо представляет собой внешний указатель на корень дерева, если узел *N* является корнем исходного дерева. Итак, любое изменение указателя *treePtr* при вызове функции *deleteNodeItem* с фактическим аргументом *treePtr* приводит к изменениям значения указателя на родителя узла *N*. Рекурсивная функция *processLeftmost*, вызываемая функцией *deleteNodeItem*, если узел N имеет два дочерних узла, также использует эту стратегию для удаления симметричного преемника узла, содержащего удаляемый элемент.

В упражнении 27, приведенном в конце главы, описывается более простой алгоритм удаления. Однако этот алгоритм приводит к увеличению высоты дерева, что, в свою очередь, влечет за собой снижение эффективности поиска.

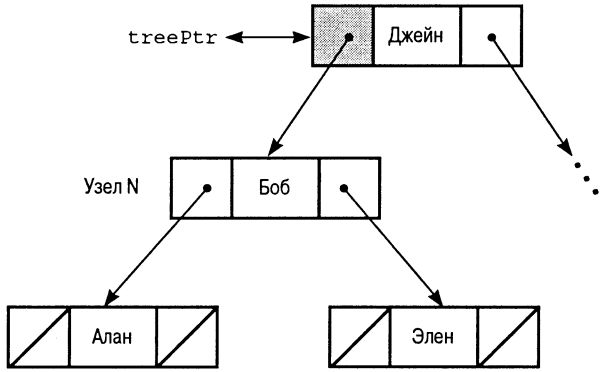
Извлечение. Уточнив алгоритм *search*, можно реализовать операцию извлечения элемента из дерева. Напомним, что алгоритм *search* выглядит так.

```
search(in bst:BinarySearchTree,
       in searchKey:KeyType)
// Поиск элемента, имеющего поисковый ключ searchKey
// в бинарном дереве поиска bst.

   if (дерево bst пусто)
      Искомый элемент не найден

   else if (searchKey == поисковый ключ корня)
      Искомый элемент найден

   else if (searchKey < поисковый ключ корня)
      search(левое поддерево дерева bst, searchKey)
   else
      search(правое поддерево дерева bst, searchKey)
```



Любое изменение указателя `treePtr` при удалении узла N (Боб) приводит к изменению указателя `leftChildPtr` в узле, содержащем запись о Джейн

Рис. 10.29. Рекурсивное удаление узла N

Операция извлечения должна возвращать элемент, имеющий искомый поисковый ключ, если тот существует; в противном случае операция генерирует исключительную ситуацию *TreeException*. Следовательно, алгоритм извлечения элемента из бинарного дерева поиска должен выглядеть так.

Алгоритм retrieveItem представляет собой усовершенствованный вариант алгоритма search

```
// Извлекает из бинарного дерева поиска,
// на которое ссылается указатель
// treePtr, элемент, поисковый ключ
// которого совпадает со значением
// аргумента searchKey, и присваивает его переменной treeItem.
// Если такого элемента в дереве нет, генерируется исключительная
// ситуация TreeException.

   if (treePtr == NULL)
      Генерировать исключительную ситуацию,
        означающую, что искомого элемента в дереве нет
   else if (searchKey == treePtr->item.getKey())
       // Элемент принадлежит корню некоего поддерева
       treeItem = treePtr->item

   else if (searchKey < treePtr->item.getKey())
       // Поиск левого поддерева
       retrieveItem(treePtr->leftChildPtr,
                    searchKey, treeItem)

   else // Поиск правого поддерева
       retrieveItem(treePtr->rightChildPtr,
                    searchKey, treeItem)
```

Обход. Обход бинарного дерева поиска выполняется точно так же, как и обход обычного бинарного дерева. Однако следует иметь в виду, что симметричный обход бинарного дерева поиска обходит узлы дерева в порядке, определенном поисковыми ключами. Перед доказательством этого утверждения напомним алгоритм симметричного обхода.

```
inorder(in bst:BinaryTree)
// Симметричный обход бинарного дерева поиска bst.

   if (дерево bst не пусто)
   {
      inorder(левое поддерево корня дерева bst)
      Вывести на экран данные, содержащиеся в корне
      preorder(правое поддерево корня дерева bst)
   }
```

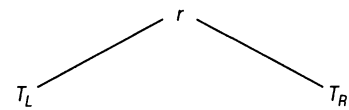
Теорема 10.1. Алгоритм симметричного обхода бинарного дерева поиска T посещает узлы в порядке, определенном их поисковыми ключами.

Доказательство. Воспользуемся методом математической индукции по h, где h — высота дерева T.

Базис: $h = 0$. Если дерево T пусто, алгоритм не посещает ни один узел. Формально это соответствует порядку следования имен в пустом множестве.

Индуктивное предположение: допустим, что теорема верна для всех k, $0 < k < h$. Иными словами, предположим, что для всех k ($0 < k < h$) алгоритм симметричного обхода посещает узлы, упорядоченные по поисковым ключам.

Индуктивное заключение: необходимо доказать, что теорема верна для $k = h > 0$. Дерево T имеет вид, показанный на рисунке ниже.



Поскольку дерево T является бинарным деревом поиска, все поисковые ключи в левом поддереве T_L меньше поискового ключа корня r, а все поисковые ключи в правом поддереве T_R больше или равны поисковому ключу корня r. Алгоритм симметричного обхода посещает все узлы левого поддерева T_L, затем посещает узел r и, в заключение, обходит правое поддерево T_R. Следовательно, нужно лишь убедиться, что алгоритм обходит левое и правое поддеревья в порядке следования поисковых ключей их узлов. Поскольку дерево T является бинарным деревом поиска и имеет высоту h, каждое из его поддеревьев также является бинарным деревом поиска, высота которого меньше h. Следовательно, по индуктивному предположению, алгоритм *inorder* обходит каждое из поддеревьев T_L и T_R в правильном порядке следования их поисковых ключей, что и требовалось доказать.

Из теоремы следует, что алгоритм *inorder* посещает симметричного преемника узла сразу после самого узла.

> Для обхода бинарного дерева перехода в порядке следования его поисковых ключей следует применять алгоритм симметричного обхода

Реализация абстрактного бинарного дерева поиска с помощью указателей

Ниже приводится реализация абстрактного бинарного дерева поиска с помощью указателей на языке C++. Обратите внимание на защищенные функции-члены, реализующие рекурсивные алгоритмы. Эти функции не объявлены открытыми, поскольку клиенты не должны иметь доступ к указателям на узлы дерева. Эти функции можно было бы объявить закрытыми, но защищенные функции позволяют производным классам использовать их непосредственно.

```
// ********************************************************
// Заголовочный файл file KeyedItem.h
// абстрактного бинарного дерева поиска.
// ********************************************************
typedef тип-поискового-ключа KeyType;

class KeyedItem
{
public:
   KeyedItem() {};
   KeyedItem(const KeyType& keyValue)
             :searchKey(keyValue) {}
   KeyType getKey() const
   {
      return searchKey;
   } // Конец функции getKey

private:
   KeyType searchKey;
   //... и другие данные
}; // Конец класса

// ********************************************************
// Заголовочный файл file TreeNode.h
// абстрактного бинарного дерева поиска.
// ********************************************************
#include "KeyedItem.h"
typedef KeyedItem TreeItemType;

class TreeNode // Узел дерева
{
private:
   TreeNode() { }
   TreeNode(const TreeItemType& nodeItem,
            TreeNode *left = NULL, TreeNode *right = NULL)
      :item(nodeItem), leftChildPtr(left),
      rightChildPtr(right) { }

   TreeItemType item; // Элементы, хранящиеся в дереве

   // Указатели на дочерние узлы
   TreeNode *leftChildPtr, *rightChildPtr;

   // Дружественный класс - имеет доступ к закрытым разделам
   friend class BinarySearchTree;
}; // Конец класса

// ********************************************************
// Заголовочный файл BST.h
// абстрактного бинарного дерева поиска.
// Предположение: в каждый момент времени дерево содержит по
//                крайней мере один элемент с заданным
//                поисковым ключом.
// ********************************************************
#include "TreeNode.h"
typedef void (*FunctionType)(TreeItemType& anItem);
```

```
class BinarySearchTree
{
public:
// Конструкторы и деструктор:
   BinarySearchTree();
   BinarySearchTree(const BinarySearchTree& tree);
   virtual ~BinarySearchTree();

// Операции над бинарным деревом поиска:
// Предусловие всех методов: в бинарном дереве поиска
// нет двух элементов, имеющих одинаковый поисковый ключ.

   virtual bool isEmpty() const;
   // Определяет, пусто ли бинарное дерево поиска.
   // Постусловие: если дерево пусто, возвращает значение true,
   // в противном случае возвращает значение false.

   virtual void searchTreeInsert(const TreeItemType& newItem);
   // Вставляет в бинарное дерево поиска новый элемент.
   // Предусловие: элемент, подлежащий вставке, задается
   // аргументом newItem.
   // Постусловие: элемент newItem вставлен в соответствующее
   // место бинарного дерева поиска.

   virtual void searchTreeDelete(KeyType searchKey)
      throw(TreeException);
   // Удаляет из бинарного дерева поиска элемент,
   // содержащий заданный поисковый ключ.
   // Предусловие: поисковый ключ удаляемого элемента
   // задается аргументом searchKey.
   // Постусловие: если в дереве есть элемент, поисковый ключ
   // которого совпадает со значением аргумента searchKey,
   // он оттуда удаляется. В противном случае дерево остается
   // без изменения, и генерируется исключительная ситуация

   virtual void searchTreeRetrieve(KeyType searchKey,
      TreeItemType& treeItem) const throw(TreeException);
   // Извлекает из бинарного дерева поиска элемент,
   // содержащий заданный поисковый ключ.
   // Предусловие: поисковый ключ удаляемого элемента
   // задается аргументом searchKey.
   // Постусловие: если в дереве есть элемент, поисковый ключ
   // которого совпадает со значением аргумента searchKey,
   // он оттуда извлекается и присваивается переменной
   // treeItem. В противном случае генерируется исключительная
   // ситуация TreeException.

   virtual void preorderTraverse(FunctionType visit);
   // Выполняет обход бинарного дерева поиска в прямом
   // порядке, один раз вызывая для каждого элемента
   // функцию visit()
   // Предусловие: функция visit() существует вне
   // реализации класса.
   // Постусловие: функция visit() выполнена по одному
   // разу для каждого узла дерева.
   // Замечание: функция visit() может изменять дерево.
```

```
   virtual void inorderTraverse(FunctionType visit);
   // Выполняет обход бинарного дерева поиска в симметричном
   // порядке, один раз вызывая для каждого элемента
   // функцию visit()

   virtual void postorderTraverse(FunctionType visit);
   // Выполняет обход бинарного дерева поиска в обратном
   // порядке, один раз вызывая для каждого элемента
   // функцию visit()

// Перегруженный оператор:
   virtual BinarySearchTree& operator=(
      const BinarySearchTree& rhs);

protected:
   void inserItem(TreeNode *& treePtr,
                  const TreeItemType& newItem);
   // Рекурсивно вставляет элемент в бинарное дерево поиска.
   // Предусловие: указатель treePtr ссылается на бинарное
   // дерево поиска. Вставке подлежит элемент newItem.
   // Постусловие: такое же, как и у функции searchTreeInsert.

   void deleteItem(TreeNode *& treePtr, KeyType searchKey);
      throw(TreeException);
   // Рекурсивно удаляет элемент из бинарного дерева поиска.
   // Предусловие: указатель treePtr ссылается на бинарное
   // дерево поиска, аргумент searchKey задает поисковый ключ
   //  удаляемого элемента.
   // Постусловие: такое же, как и у функции searchTreeInsert.

   void deleteNodeItem(TreeNode *& nodePtr);
   // Удаляет элементы из корня указанного дерева.
   // Предусловие: указатель nodePtr ссылается на корень
   // бинарного дерева поиска; nodePtr != NULL.
   // Постусловие: элемент, содержащийся в корне указанного
   // дерева, удален оттуда.

   void processLeftmost(TreeNode *& nodePtr,
                        TreeItemType& treeItem);
   // Извлекает и удаляет из бинарного дерева поиска
   // самый левый потомок указанного узла.
   // Предусловие: указатель nodePtr ссылается на корень
   // бинарного дерева поиска; nodePtr != NULL.
   // Постусловие: аргумент treeItem содержит элемент,
   // хранящийся в самом левом потомке узла, на который
   // ссылается указатель nodePtr. Сам потомок удален.

   void retrieveItem(TreeNode *treePtr, KeyType searchKey,
                     TreeItemType& treeItem) const
      throw(TreeException);
   // Рекурсивно извлекает элемент из бинарного дерева поиска.
   // Предусловие: указатель treePtr ссылается на бинарное
   // дерево поиска. Аргумент searchKey задает поисковый ключ
   // извлекаемого элемента.
   // Постусловие: такое же, как и у функции searchTreeRetrieve.
```

```
    // Следующие 9 методов совпадают с методами абстрактного
    // бинарного дерева, поэтому их спецификации не указываются.

    void copyTree(TreeNode *treePtr, TreeNode *& newTreePtr) const;
    void destroyTree(TreeNode *& treePtr);
    void preorder(TreeNode *treePtr, FunctionType visit);
    void inorder(TreeNode *treePtr, FunctionType visit);
    void postorder(TreeNode *treePtr, FunctionType visit);
                   TreeNode *rootPtr() const;
    void setRootPtr(TreeNode *newRoot);
    void getChildPtrs(TreeNode *nodePtr,
                      TreeNode *& leftChildPtr,
                      TreeNode *& rightChildPtr) const;
    void setChildPtrs(TreeNode *nodePtr,
                      TreeNode *leftChildPtr,
                      TreeNode *rightChildPtr);
private:
    TreeNode *root; // Указатель на корень дерева
}; // Конец класса
// Конец заголовочного файла.

// ********************************************************
// Файл реализации BST.cpp.
// ********************************************************
#include "BST.h" // Заголовочный файл
#include <cstddef> // Определение константы NULL
BinarySearchTree::BinarySearchTree() : root(NULL)
{
} // Конец конструктора по умолчанию

BinarySearchTree::BinarySearchTree(
    const BinarySearchTree& tree)
{
    copyTree(tree.root, root);
} // Конец конструктора копирования

BinarySearchTree::~BinarySearchTree()
{
    destroyTree(root);
} // Конец деструктора

bool BinarySearchTree::isEmpty() const
{
    return (root == NULL);
} // Конец функции searchTreeIsEmpty

void BinarySearchTree::searchTreeInsert(
    const TreeItemType& newItem)
{
    insertItem(root, newItem);
} // Конец функции searchTreeInsert

void BinarySearchTree::searchTreeDelete(KeyType searchKey)
{
    deleteItem(root, searchKey);
} // Конец функции searchTreeDelete
```

```
void BinarySearchTree::searchTreeRetrieve(KeyType searchKey,
   TreeItemType& treeItem) const
{
  // Если функция retrieveItem генерирует исключительную
  // ситуацию TreeException, она игнорируется, и управление
  // передается в точку вызова функции searchTreeRetrieve.
   retrieveItem(root, searchKey, treeItem);
} // Конец функции searchTreeRetrieve

void BinarySearchTree::preorderTraverse(FunctionType visit)
{
   preorder(root, visit);
} // Конец функции preorderTraverse

void BinarySearchTree::inorderTraverse(FunctionType visit)
{
   inorder(root, visit);
} // Конец функции inorderTraverse

void BinarySearchTree::postorderTraverse(FunctionType visit)
{
   postorder(root, visit);
} // Конец функции postorderTraverse

void BinarySearchTree::insertItem(TreeNode *& treePtr,
   const TreeItemType& newItem)
{
   if (treePtr == NULL)
   {
     // Найдена позиция вставки;
     // производится вставка после листа

     // Создать новый узел
     treePtr = new TreeNode(newItem, NULL, NULL);

     // Правильно ли выделена память?
     if (treePtr == NULL)
        throw TreeException(
        "TreeException: вставку выполнить невозможно");
   }
   // В противном случае ищем место для вставки
   else if (newItem.getKey() < treePtr->item.getKey())
   // Поиск в левом поддереве
   insertItem(treePtr->leftChildPtr, newItem);

   else // Поиск в правом поддереве
   insertItem(treePtr->rightChildPtr, newItem);
} // Конец функции insertItem

void BinarySearchTree::deleteItem(TreeNode *& treePtr,
   KeyType searchKey)
// Вызываемые функции: deleteNodeItem.
{
   if (treePtr == NULL)
      throw TreeException(
      "TreeException: удалить элемент невозможно"); // Дерево пусто
```

```
    else if (searchKey == treePtr->item.getKey())
    // Элемент принадлежит корню некоего поддерева
    deleteNodeItem(treePtr); // Удаляем элемент

    // В противном случае выполняем поиск заданного элемента
    else if (searchKey < treePtr->item.getKey())
    // Поиск в левом поддереве
    deleteItem(treePtr->leftChildPtr, searchKey);

    else // Поиск в правом поддереве
    deleteItem(treePtr->rightChildPtr, searchKey);
} // Конец функции deleteItem

void BinarySearchTree::deleteNodeItem(TreeNode *& nodePtr)
    // Замечания об алгоритме: существуют четыре варианта.
    // 1. Корень является листом.
    // 2. У корня нет левого дочернего узла.
    // 3. У корня нет правого дочернего узла.
    // 4. Корень имеет два дочерних узла.
    // Вызываемая функция: processLeftmost.
{
    TreeNode *delPtr;
    TreeItemType replacementItem;
   // Проверка первого варианта
   if ( (nodePtr->leftChildPtr == NULL) &&
      (nodePtr->rightChildPtr == NULL) )
   {
      delete nodePtr;
      nodePtr = NULL;
    } // Конец проверки первого варианта

    // Проверка второго варианта
    else if (nodePtr->leftChildPtr == NULL)
    {
      delPtr = nodePtr;
      nodePtr = nodePtr->rightChildPtr;
      delPtr->rightChildPtr = NULL;
      delete delPtr;
    } // Конец проверки второго варианта

    // Проверка третьего варианта
    else if (nodePtr->rightChildPtr == NULL)
    {
      delPtr = nodePtr;
      nodePtr = nodePtr->leftChildPtr;
      delPtr->leftChildPtr = NULL;
      delete delPtr;
    } // Конец проверки третьего варианта

    // Проверка четвертого варианта:
    // извлечь и удалить симметричного преемника
    else
    {
       processLeftmost(nodePtr->rightChildPtr,
       replacementItem);
       nodePtr->item = replacementItem;
    } // Конец проверки четвертого варианта
```

```
} // Конец функции deleteNodeItem

void BinarySearchTree::processLeftmost(TreeNode *& nodePtr,
{
   if (nodePtr->leftChildPtr == NULL)
   {
     treeItem = nodePtr->item;
     TreeNode *delPtr = nodePtr;
     nodePtr = nodePtr->rightChildPtr;
     delPtr->rightChildPtr = NULL; // Защита
     delete delPtr;
   }
   else
     processLeftmost(nodePtr->leftChildPtr, treeItem);
} // Конец функции processLeftmost

void BinarySearchTree::retrieveItem(TreeNode *treePtr,
   KeyType searchKey,TreeItemType& treeItem) const
{
   if (treePtr == NULL)
      throw TreeException(
      "TreeException: ключ searchKey не найден");
      TreeItemType& treeItem)

   else if (searchKey == treePtr->item.getKey())
      // Элемент содержится в корне некоего поддерева
      treeItem = treePtr->item;

   else if (searchKey < treePtr->item.getKey())
      // Поиск в левом поддереве
      retrieveItem(treePtr->leftChildPtr,
      searchKey, treeItem);

   else // Поиск в правом поддереве
      retrieveItem(treePtr->rightChildPtr,
                   searchKey, treeItem);
} // Конец функции retrieveItem

// Реализации функций-членов copyTree, destroyTree, preorder,
// inorder, postorder, setRootPtr, rootPtr, getChildPtrs,
// setChildPtrs и перегруженных операторов присваивания
// совпадают с реализациями, предусмотренными в классе
// абстрактных бинарных деревьев.
// Конец файла реализации.
```

Сравнивая реализации абстрактного бинарного дерева и абстрактного бинарного дерева поиска, можно заметить некоторую избыточность кода. Обе структуры данных и их реализации предусматривают несколько фактически идентичных методов (см. комментарии в конце файла реализации). Поскольку бинарное дерево поиска является разновидностью бинарного дерева, эта избыточность не удивительна. Ее можно избежать, определив класс *BinarySearchTree* производным от класса *BinaryTree*. Это задание читатели могут выполнить самостоятельно (см. упражнение 29).

Эффективность операций над бинарными деревьями поиска

Мы изучили разные формы бинарных деревьев поиска. Например, несмотря на то что все бинарные деревья поиска, изображенные на рис. 10.19 и 10.20, *в*, содержат по семь узлов, их высота и форма совершенно отличаются друг от друга. Скажем, чтобы обнаружить запись о Венди в дереве, показанном на рис. 10.20, *в*, необходимо проверить все семь узлов, а в дереве, изображенном на рис. 10.19, для этого достаточно проверить только три узла (записи о Джейн, Томе и самом Венди). Рассмотрим теперь связь между высотой бинарного дерева поиска и эффективностью операций извлечения, вставки и удаления.

Максимальное количество сравнений, необходимых для выполнения операций извлечения, вставки и удаления, равно высоте бинарного дерева поиска

Каждая из этих операций сравнивает заданное значение `searchKey` с поисковыми ключами узлов, находящихся в дереве на некотором **пути** (path). Этот путь всегда начинается с корня дерева и для каждого узла *n* проходит либо по левой, либо по правой ветви, в зависимости от результата сравнения значения `searchKey` с поисковым ключом узла *n*. Путь заканчивается узлом, содержащим значение `searchKey`, или пустым поддеревом, если этот ключ не найден. Таким образом, количество сравнений при каждой операции извлечения, вставки или удаления равно количеству узлов на этом пути. Это значит, что максимальное количество сравнений, необходимых для выполнения каждой операции, равно количеству узлов на самом длинном пути, существующем в дереве. Иными словами, *максимальное количество сравнений, необходимых для выполнения этих операций, равно высоте бинарного дерева поиска*. Каковы же максимальная и минимальная высота бинарного дерева поиска, состоящего из *n* узлов?

Максимальная и минимальная высота бинарного дерева поиска. Достичь максимальной высоты бинарного дерева поиска, состоящего из *n* узлов, очень просто. Для этого достаточно сделать так, чтобы каждый внутренний узел (не лист) имел только один дочерний узел, как показано на рис. 10.30. В результате получится дерево, высота которого равна *n*. Если высота бинарного дерева поиска, состоящего из *n* узлов, равна *n*, то это дерево представляет собой линейный связанный список.

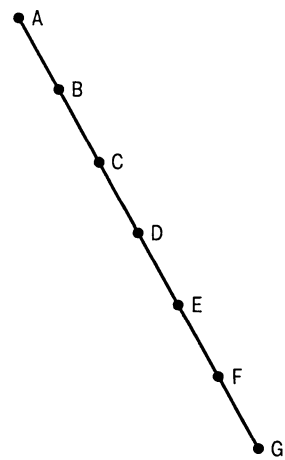


Рис. 10.30. Максимальная высота бинарного дерева поиска, состоящего из семи узлов

Оценить минимальную высоту бинарного дерева поиска, состоящего из n узлов, немного сложнее. На первом шаге рассмотрим количество узлов, которое может иметь бинарное дерево поиска, высота которого равна h. Например, если $h = 3$, то возможное бинарное дерево поиска может быть таким, как показано на рис. 10.31. Итак, бинарное дерево поиска, высота которого равна 3, может иметь от 3 до 7 узлов. Кроме того, на рис. 10.31 видно, что число 3 — это минимальная высота бинарного дерева, состоящего из 4, 5, 6 или 7 узлов. Аналогично, высота бинарных деревьев, состоящих более чем из 7 узлов, больше 3.

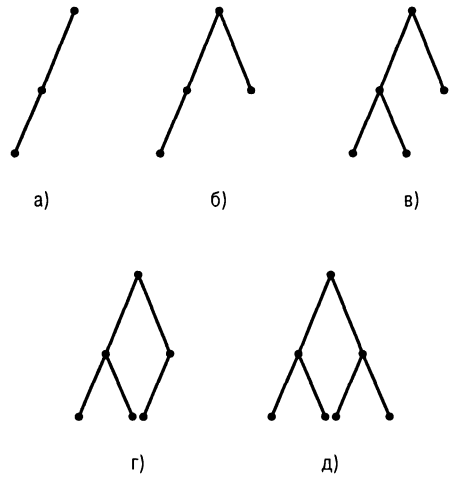


Рис. 10.31. Бинарное дерево поиска, высота которого равна 3

Интуитивно ясно, что для минимизации высоты бинарного дерева поиска, состоящего из n узлов, нужно как можно больше заполнить каждый уровень дерева. Этому условию соответствует совершенное бинарное дерево (хотя при этом не имеет значения, что последний узел такого дерева заполняется слева направо). Фактически деревья, изображенные на рис. 10.31, *б–г*, являются совершенными. Чтобы совершенное бинарное дерево, имеющее высоту h, содержало максимально возможное количество узлов, оно должно быть полным (как на рис. 10.31, *д*). На рис. 10.32 показаны результаты подсчета узлов на каждом уровне такого дерева. Сформулируем эти результаты в виде теоремы.

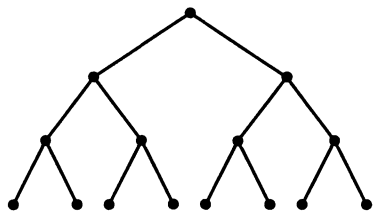
Теорема 10.2. Полное бинарное дерево поиска, высота которого равна $h \geq 0$, состоит из $2^h - 1$ узлов.

Формальное доказательство этой теоремы с помощью метода математической индукции читатели могут провести самостоятельно.

Теорема 10.3. Максимальное количество узлов, которое может содержать бинарное дерево, имеющее высоту h, равно $2^h - 1$.

В полное бинарное дерево невозможно добавить новые узлы, не изменив его высоту. Формальное доказательство этой теоремы, очень похожее на доказательство теоремы 10.2, предоставляется читателям в качестве упражнения.

Приведенная ниже теорема использует теоремы 10.2 и 10.3 для вычисления минимальной высоты бинарного дерева, содержащего заданное количество узлов.



Уровень	Количество узлов на заданном уровне	Количество узлов на заданном и предыдущих уровнях
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
•	•	•
•	•	•
•	•	•
h	2^{h-1}	$2^h - 1$

Рис. 10.32. Подсчет узлов в полном бинарном дереве поиска, высота которого равна h

Теорема 10.4 Минимальная высота бинарного дерева, имеющего высоту h, равна $[\log_2(n+1)]$.[7]

Доказательство. Пусть h — наименьшее целое число, удовлетворяющее условию $n \le 2^h - 1$. Чтобы найти минимальную высоту бинарного дерева, состоящего из n узлов, сначала докажем следующие утверждения.

1. *Количество узлов бинарного дерева, высота которого не превышает h-1, меньше n.*

   По теореме 10.3 бинарное дерево высоты h-1 имеет не более $2^{h-1}-1$ узлов. Если возможно неравенство $n \le 2^{h-1}-1<2^h-1$, то число h не является наименьшим целым числом, удовлетворяющим условию $n \le 2^h-1$. Следовательно, число n должно быть больше, чем $2^{h-1}-1$, или, что эквивалентно, $2^{h-1}-1<n$. Итак, поскольку бинарное дерево, высота которого равна h-1, состоит не более чем из $2^{h-1}-1$ узлов, количество его узлов не превышает числа n.

2. *Существует совершенное бинарное дерево, имеющее высоту h и состоящее из n узлов.*

   Рассмотрим полное бинарное дерево, высота которого равна h-1. По теореме 10.2 оно состоит из $2^{h-1}-1$ узлов. Как мы только что доказали, $n>2^{h-1}-1$, поскольку число h выбрано так, что $n \le 2^h-1$. Следовательно, мы можем добавить в полное дерево новые узлы, размещая их слева направо, пока их общее количество не станет равным числу n, как показано на рис. 10.33. Поскольку $n \le 2^h-1$, а бинарное дерево высоты h не может содержать больше 2^h-1 узлов, общее количество узлов становится равным n, когда последний уровень h оказывается полностью заполненным.

[7] Квадратные скобки [X] обозначают наименьшее целое число, превосходящее X (ceiling of X), например [6]=6, [6.1]=7, [6.8]=[7].

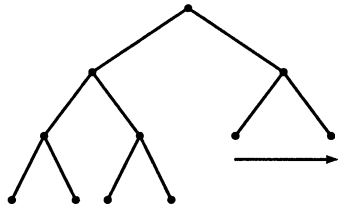


Рис. 10.33. Заполнение последнего уровня дерева

3. *Минимальная высота бинарного дерева, состоящего из n узлов, равна наименьшему целому числу, удовлетворяющему условию* $n \le 2^h - 1$.

   Если h является наименьшим целым числом, удовлетворяющим условию $n \le 2^h - 1$, а высота бинарного дерева не превышает $h-1$, то вследствие утверждения 1, оно состоит не больше чем из n узлов. По утверждению 2, существует бинарное дерево, имеющее высоту h и состоящее из n узлов.

Из доказанных утверждений следуют следующие неравенства.

$2^{h-1} - 1 < n \le 2^h - 1$

$2^{h-1} < n+1 \le 2^h$

$h - 1 < \log_2(n+1) \le h$

Если $\log_2(n+1) = h$, теорема доказана. В противном случае из неравенства $h-1 < \log_2(n+1) < h$ следует, что число $\log_2(n+1)$ не может быть целым. Следовательно, округление этого числа сверху равно числу h.

Итак, $h = [\log_2(n+1)]$ — минимальная высота бинарного дерева, состоящего из n узлов, что и требовалось доказать.

Совершенные и полные деревья имеют минимальную высоту

Итак, мы доказали, что совершенные и полные деревья, состоящие из n узлов, имеют высоту $[\log_2(n+1)]$. Это число является минимальным среди всех теоретически возможных значений высоты. Оказывается, минимальная высота бинарного дерева равна максимальному количеству сравнений, выполняемых при бинарном поиске элемента в массиве, состоящем из n элементов. Таким образом, если бинарное дерево поиска является совершенным и, следовательно, сбалансированным, время поиска значения в этом дереве приблизительно равно времени бинарного поиска элемента в массиве.

Высота бинарного дерева, состоящего из n узлов, изменяется от $[\log_2(n+1)]$ до n

Однако если перейти от сбалансированных деревьев к деревьям с линейной структурой, их высота становится равной количеству узлов n. Это число равно максимальному количеству сравнений, которые необходимо выполнить при поиске элемента в связанном списке, состоящем из n узлов.

Вставка по ключу порождает бинарное дерево поиска максимальной высоты

Однако эффективность операций над бинарным деревом поиска зависит от предположения, что его высота равна $[\log_2(n+1)]$. Какой будет высота бинарного дерева поиска на практике? Это зависит от порядка, в котором выполняются операции вставки и удаления элементов этого дерева. Напомним, что если, начиная с пустого дерева, мы будем вставлять имена в порядке Алан, Боб, Элен, Джейн, Нэнси, Том, Венди, то получим бинарное дерево поиска, имеющее максимальную высоту, как показано на рис. 10.20, *в*. С другой стороны, если вставлять имена в порядке Джейн, Боб, Том, Алан, Элен, Нэнси, Венди, получится бинарное дерево поиска, имеющее минимальную высоту, как показано на рис. 10.19.

Какие из этих ситуаций могут возникнуть в реальных приложениях? Можно строго математически доказать, что, если операции вставки и удаления выполняются в произвольном порядке, высота бинарного дерева поиска будет довольно близкой к числу $[\log_2(n+1)]$. Следовательно, анализ, выполненный нами ранее, не был чрезмерно оптимистичным. Однако насколько реалистичным является предположение, что операции вставки и удаления выполняются в произвольном порядке? Во многих приложениях это действительно так. И все же бывают ситуации, в которых это предположение оказывается сомнительным. Например, человек, вставляющий указанные выше имена в бинарное дерево поиска, может упорядочить имена по алфавиту, решив, что так будет "лучше". Как было показано выше, такой порядок имен приводит к максимальной высоте дерева. Итак, хотя в большинстве приложений бинарное дерево поиска демонстрирует максимальную эффективность, необходимо учитывать особенности конкретного приложения.

Вставка в произвольном порядке порождает бинарное дерево поиска, имеющее почти минимальную высоту

Что предпринять, если операции не выполняются в произвольном порядке? А что делать, если количество элементов чрезвычайно велико? Как убедиться, что высота дерева действительно близка к величине $[\log_2(n+1)]$? Ответам на эти вопросы посвящена глава 12.

Оценки сложности операций извлечения, вставки, удаления и обхода абстрактного бинарного дерева поиска, реализованного с помощью указателей, приведены на рис. 10.34.

Операция	Средний вариант	Наихудший вариант
Извлечение	O(log n)	O(n)
Вставка	O(log n)	O(n)
Удаление	O(log n)	O(n)
Обход	O(n)	O(n)

Рис. 10.34. Порядок сложности операций извлечения, вставки, удаления и обхода абстрактного бинарного дерева поиска, реализованного с помощью указателей

Древовидная сортировка

Абстрактное бинарное дерево поиска можно применять для сортировки массива записей по ключу. Однако для простоты мы, как и в главе 9, рассмотрим сортировку массива целых чисел в порядке возрастания.

Основная идея алгоритма проста.

Древовидная сортировка использует бинарное дерево поиска

```
treesort(inout anArray:ArrayType,
         in n:integer)
// Сортирует n целочисленных элементов массива anArray
// в возрастающем порядке.

   Вставить элемент массива anArray в бинарное дерево
      поиска bTree

   Выполнить симметричный обход дерева bTree. При посещении
      узлов дерева копировать содержащиеся там числа
      в последовательные ячейки массива anArray
```

При симметричном обходе бинарного дерева поиска *bTree* целые числа, содержащиеся в его узлах, записываются в порядке возрастания.

Древовидная сортировка. Средний вариант O(n*log n), наихудший вариант $O(n^2)$

Алгоритм древовидной сортировки довольно эффективен. Как показано на рис. 10.34, при каждой вставке элемента в бинарное дерево поиска в среднем выполняются O(log *n*) операций, а в худшем случае — O(*n*). Следовательно, чтобы вставить *n* элементов в бинарное дерево поиска, как предусмотрено алгоритмом *treeSort*, необходимо выполнить в среднем O(*n**log *n*) операций, а в худшем случае — $O(n^2)$ операций. При обходе дерева выполняется одна операция копирования для каждого из *n* элементов. Таким образом, общее число таких операций оценивается величиной O(*n*). Поскольку порядок величины O(*n*) меньше порядка величин O(*n**log *n*) и $O(n^2)$, средняя сложность алгоритма *treeSort* имеет порядок O(*n**log *n*), а худшая — $O(n^2)$.

Запись бинарного дерева поиска в файл

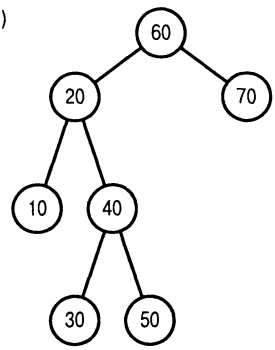
Представьте себе программу, позволяющую хранить имена, адреса и номера телефонов ваших друзей и родственников. Пока программа работает, вы можете вводить имена и получать адреса и телефонные номера ваших приятелей. Однако по завершении работы программы базу данных необходимо записать в форме, позволяющей использовать ее в следующий раз.

Если для представления этой базы данных программа использует бинарное дерево поиска, его элементы необходимо записать в файл, чтобы иметь возможность в будущем извлечь их оттуда. Для записи и считывания бинарного дерева поиска существуют два разных алгоритма. Первый алгоритм позволяет считать бинарное дерево поиска в его исходном виде, а второй — приводит его в сбалансированную форму.

Для записи и считывания бинарного дерева поиска в его исходном виде используются алгоритм прямого обхода и операция searchTreeInsert

Сохранение бинарного дерева поиска и его восстановление в исходном виде. Первый алгоритм восстанавливает бинарное дерево поиска в той форме, в которой оно было записано. Рассмотрим в качестве примера дерево, изображенное на рис. 10.35. Если сохранить это дерево в прямом порядке, мы получим последовательность чисел 60, 20, 10, 40, 30, 50, 70. Если затем применить операцию *searchTreeInsert*, чтобы вставить эти значения в пустое дерево, мы восстановим исходное дерево.

а)



б)

```
bst.searchTreeInsert(60);
bst.searchTreeInsert(20);
bst.searchTreeInsert(10);
bst.searchTreeInsert(40);
bst.searchTreeInsert(30);
bst.searchTreeInsert(50);
bst.searchTreeInsert(70);
```

Рис. 10.35. Пример бинарного дерева поиска: а) бинарное дерево поиска bst; б) последовательность вставок, результатом которых является это дерево

Сохранение бинарного дерева поиска и его восстановление в сбалансированном виде. Можно ли улучшить описанный выше алгоритм? Так ли уж необходимо восстанавливать бинарное дерево поиска в его первоначальном виде? Напомним, что элементы бинарного дерева поиска можно организовывать по-разному. Хотя результат операций над абстрактным бинарным деревом поиска не зависит от его формы, она влияет на их эффективность. Сбалансированное бинарное дерево поиска позволяет повысить эффективность выполняемых операций.

Сбалансированное бинарное дерево поиска позволяет повысить эффективность операций

Алгоритм, восстанавливающий бинарное дерево поиска в сбалансированном виде, удивительно прост. Фактически он даже позволяет минимизировать высоту восстановленного дерева, а это условие намного важнее простой сбалансированности. Чтобы разобраться в решении этой задачи, рассмотрим полное дерево, поскольку оно, по определению, является сбалансированным. Если записать его, применяя алгоритм симметричного обхода, файл будет содержать упорядоченные числа, как показано на рис. 10.36. Средний элемент полного дерева, имеющего высоту h и содержащего ровно $n = 2^h-1$ узлов, находится в его корне. Левые и правые поддеревья корня сами являются полными деревьями, состоящими из $2^{h-1}-1$ узлов каждое (т.е. из $n/2$ узлов). Таким образом, для создания полного бинарного дерева, содержащего n узлов, можно использовать следующий рекурсивный алгоритм. Предполагается, что число n известно заранее.

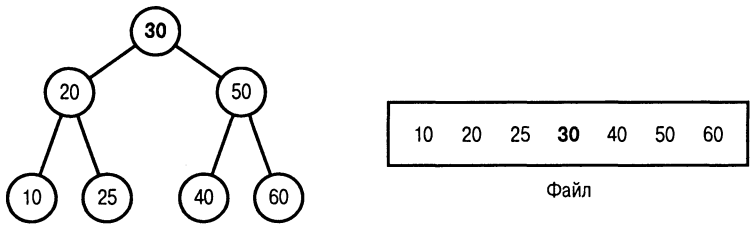


Рис. 10.36. Полное дерево, записанное в файл с помощью алгоритма симметричного обхода

Создание полного бинарного дерева поиска

```
readFull(out treePtr:TreeNodePtr,
         in n:integer)
// Создает полное бинарное дерево
// на основе n упорядоченных чисел, записанных в файле.
// Указатель treePtr ссылается на корень дерева.
   if (n > 0)
   {
      // Создаем левое поддерево
      treePtr = указатель на новый узел,
                в котором указатели на дочерние узлы
                равны константе NULL
      readFull(treePtr->leftChildPtr, n/2)

      // Считываем значение, содержащееся в корне
      Считать элемент из файла в ячейку treePtr->item

      // Создаем правое поддерево
      readFull(treePtr->rightChildPtr, n/2)
  } // Конец оператора if
```

Удивительно, но факт: мы можем создать дерево, просто последовательно считав упорядоченные данные, записанные в файле.

Алгоритм построения полного бинарного дерева поиска прост, но что делать, если восстанавливаемое дерево не является полным (т.е. не содержит $n = 2^h - 1$ узлов при высоте h)? Первое, что приходит в голову: восстанавливаемое дерево должно быть совершенным, т.е. полным вплоть до последнего уровня, который заполнен слева направо. Поскольку мы стремимся минимизировать высоту восстанавливаемого дерева, порядок заполнения элементов последнего уровня не важен, как показано на рис. 10.37.

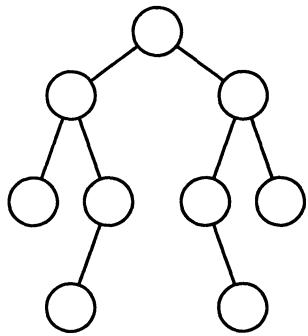


Рис. 10.37. Несовершенное дерево, имеющее минимальную высоту

Функция *readFull* остается правильной, даже если дерево не полно. Однако, вычисляя размеры левого и правого поддеревьев, теперь нужно быть осторожным. Если число n нечетно, то оба поддерева, как и прежде, имеют высоту $n/2$. (Корень учитывается автоматически.) Однако, если число n *четно, корень нужно* учитывать явно, причем одно поддерево будет содержать на один узел больше другого. В этом случае, для определенности, "лишний" узел можно произвольно помещать в левое поддерево.

Создание бинарного дерева поиска, имеющего минимальную высоту

```
readTree(out treePtr:TreeNodePtr,
         in n:integer)
// Создает бинарное дерево поиска,
// имеющее минимальную высоту,
// из n упорядоченных чисел, записанных в файле.
// Указатель treePtr ссылается на корень дерева.

   if (n > 0)
   {
      // Создаем левое поддерево
      treePtr = указатель на новый узел,
                в котором указатели на дочерние узлы
                равны константе NULL
      readTree(treePtr->leftChildPtr, n/2)

     // Считываем значение, содержащееся в корне
     Считать элемент из файла в переменную treePtr->item

     // Создаем правое поддерево
     readTree(treePtr->rightChildPtr, (n-1)/2)
   } // Конец оператора if
```

Выполните трассировку этого алгоритма и убедитесь, что он правильно работает при четных и нечетных значениях *n*.

Итак, бинарное дерево поиска легко восстановить в сбалансированном виде, если данные в файле упорядочены, т.е. записаны туда при симметричном обходе дерева, а число узлов *n* известно. Число *n* должно быть известно заранее, поскольку от его значения зависит выбор среднего элемента и, следовательно, количество узлов, содержащихся в левом и правом поддереве. Для этого достаточно просто подсчитать количество узлов при обходе дерева и записать их в файл, а при восстановлении дерева считать их оттуда.

Обратите внимание, что функцию `readTree` иногда удобно делать защищенным членом класса `BinarySearchTree`. В этом случае для ее вызова в классе необходимо предусмотреть соответствующую открытую функцию, играющую роль посредника.

Деревья общего вида

Завершим главу кратким обсуждением деревьев общего вида и их связей с бинарными деревьями. Рассмотрим дерево общего вида, изображенное на рис. 10.38. Три узла, *B*, *C* и *D*, дочерних по отношению к узлу *A*, являются братьями. Левый дочерний узел *B* называется **старшим** (oldest child), или **первым** (first child). Для реализации этого дерева можно воспользоваться структурой узла, которую мы уже применяли при работе с бинарными деревьями. Таким образом, каждый узел имеет два указателя: левый указатель ссылается на старший дочерний узел, а правый — на следующий дочерний узел. Для реализации дерева, представленного на рис. 10.38, можно применить структуру, изображенную на рис. 10.39. Обратите внимание, что эта структура соответствует также бинарному дереву, изображенному на рис. 10.40.

Бинарное дерево, каждый узел которого может иметь не более *n* дочерних узлов, называется **n-арным деревом** (*n*-ary tree). Дерево, изображенное на рис. 10.38, является тернарным (*n*=3). Разумеется, описанный выше способ реализации можно применять для создания произвольного *n*-арного дерева. Однако, поскольку количество дочерних узлов для каждого узла известно заранее, указатели каждого узла можно установить непосредственно на дочерние. Такая реализация *n*-арного бинарного дерева, изображенного на рис. 10.38, показана на рис. 10.41. Такое дерево оказалось короче дерева, показанного на рис. 10.40.

Дальнейшие свойства деревьев общего вида обсуждаются в упражнении 32, приведенном в конце главы.

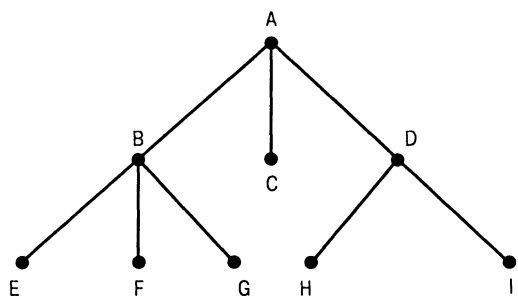


Рис. 10.38. Дерево общего вида

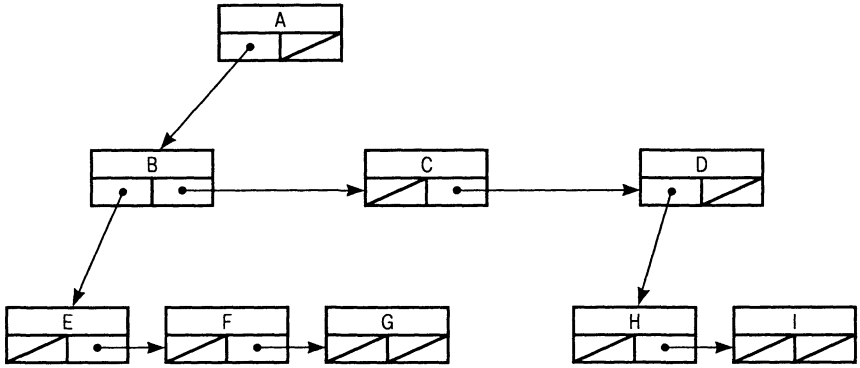


Рис. 10.39. Реализация дерева общего вида, изображенного на рис. 10.38, с помощью указателей

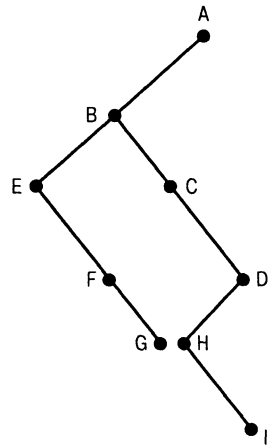


Рис. 10.40. Бинарное дерево, представленное структурой, изображенной на рис. 10.39

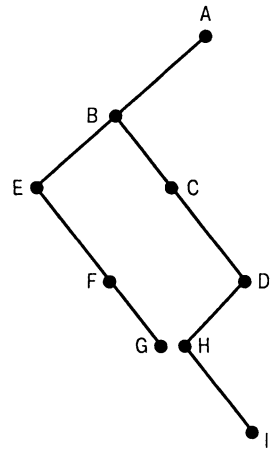


Рис. 10.41. Реализация n-арного бинарного дерева, изображеного на рис. 10.38

Резюме

1. Бинарные деревья обеспечивают иерархическую организацию данных, играющую важную роль в различных приложениях.
2. Как правило, бинарные деревья реализуются с помощью указателей. Если бинарное дерево является совершенным, его можно эффективно реализовать в виде массива.
3. Обход бинарного дерева — очень полезная операция. Интуитивно ясно, что обход бинарного дерева означает посещение каждого узла дерева. Поскольку смысл слова "посещение" зависит от конкретного приложения, операция обхода получает функцию `visit()` от клиента.
4. Бинарное дерево поиска позволяет применять для поиска заданного элемента алгоритмы, подобные алгоритму бинарного поиска.
5. Бинарные деревья поиска могут принимать разные формы. Высота бинарного дерева поиска, состоящего из n узлов, изменяется от $[\log_2(n+1)]$ до n. Эффективность операций над бинарным деревом поиска зависит от его формы. Чем больше сбалансировано бинарное дерево поиска, тем ближе эффективность алгоритма `search` к эффективности алгоритма бинарного поиска (и дальше от эффективности алгоритма линейного поиска).
6. При симметричном обходе бинарного дерева поиска узлы посещаются в порядке следования их поисковых ключей.
7. Алгоритм древовидной сортировки позволяет эффективно упорядочивать массивы, используя операции вставки элемента в бинарное дерево поиска и его обхода.
8. Если при записи данных, содержащихся в узлах бинарного дерева поиска, выполняется симметричный обход, при восстановлении дерево будет иметь наименьшую высоту. Если при записи данных, содержащихся в узлах бинарного дерева поиска, выполняется прямой обход, при восстановлении дерево будет иметь исходную форму.

Предупреждения

1. Реализуя совершенное бинарное дерево в виде массива, убедитесь, что после выполнения операций вставки и удаления дерево остается совершенным.
2. Операции над бинарным деревом поиска могут быть довольно эффективными. Однако в наихудшем случае, — когда дерево имеет линейную структуру, — производительность операций падает и становится сравнимой с операциями над линейным связанным списком. Для того чтобы избежать снижения эффективности операций, следует применять методы балансировки деревьев, описанные в главе 12.

Вопросы для самопроверки

1. Проанализируйте дерево, изображенное на рис. 10.42. Укажите следующие узлы дерева.
   1.1. Корень дерева.
   1.2. Родительские узлы.
   1.3. Дочерние узлы.
   1.4. Братья.
   1.5. Предки узла 50.
   1.6. Потомки узла 50.
   1.7. Листья.

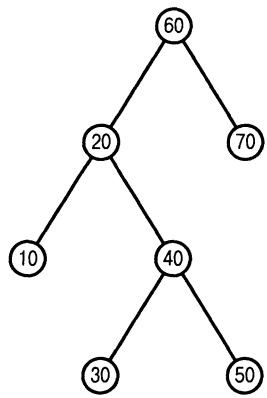


Рис. 10.42. Дерево, упоминающееся в вопросах 1, 3 и 7, а также в упражнениях 6 и 11

2. Сколько уровней имеют деревья, изображенные на следующих рисунках.
   2.1. Рис. 10.6, *б*.
   2.2. Рис. 10.6, *в*.
3. Чему равна высота дерева, изображенного на рис. 10.42?
4. Проанализируйте бинарные деревья, изображенные на рис. 10.31. Какое из них является совершенным? Полным? Сбалансированным?

5. Укажите порядок прямого, симметричного и обратного обхода бинарного дерева, представленного на рис. 10.6, *а*.
6. Как из пустого бинарного дерева поиска, пользуясь операцией вставки, получить дерево, содержащее значения J, N, B, A, W, E, T в указанном порядке?
7. В каком порядке следует вставлять элементы в пустое бинарное дерево поиска, чтобы получить дерево, изображенное на рис. 10.42?
8. Представьте в виде массива полное бинарное дерево поиска, показанное на рис. 10.36.
9. Какое совершенное бинарное дерево представляет массив, изображенный на рис. 10.43?

5	1	2	8	6	10	3	9	4	7
0	1	2	3	4	5	6	7	8	9

Рис. 10.43. Массив, упоминающийся в вопросе 9

10. Является ли дерево, изображенное на рис. 10.44, бинарным?

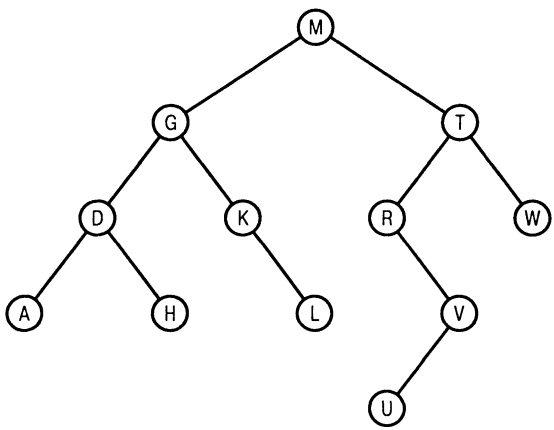


Рис. 10.44. Дерево, упоминающееся в вопросе 10 и упражнении 2.1

11. Используя бинарное дерево поиска, представленное на рис. 10.42, выполните трассировку алгоритма поиска элемента по ключу при разных значениях.

    11.1. 30

    11.2. 15

    Для каждого варианта перечислите узлы в порядке их посещения.
12. Выполните трассировку алгоритма сортировки массива в возрастающем порядке. Массив имеет следующий вид: 20 80 40 25 60 30.
13. Выполните следующие задания.

    13.1. Какое бинарное дерево получится при выполнении операции `readTree`, если в файле записаны числа 2, 4, 6, 8, 10, 12?

    13.2. Имеет ли это дерево минимальную высоту? Является ли оно совершенным? Является ли оно полным?

Упражнения

1. Напишите пред- и постусловия операций над абстрактным бинарным деревом.
2. Укажите порядок прямого, симметричного и обратного обхода бинарных деревьев, изображенных на следующих рисунках.

   2.1. Рис. 10.44.

   2.2. Рис. 10.6, *б*.

   2.3. Рис. 10.6, *в*.
3. Проанализируйте бинарное дерево поиска, изображенное на рис. 10.45. Номера на рисунке являются лишь метками узлов, но не их содержанием.

   3.1. Какой узел содержит симметричный преемник значения, содержащегося в корне? Обоснуйте свой ответ.

   3.2. В каком порядке алгоритм симметричного обхода посещает узлы этого дерева? Перечислите метки этих узлов в порядке их посещения.

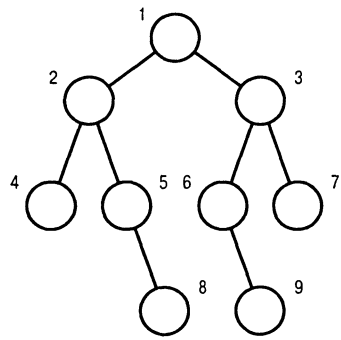


Рис. 10.45. Бинарное дерево поиска из упражнения 3

4. Как из пустого бинарного дерева поиска, пользуясь операциями вставки, получить дерево, содержащее указанные символы?

   4.1. W, T, N, J, E, B, A

   4.2. W, T, N, A, B, E, J

   4.3. A, B, W, J, N, T, E
5. Организуйте узлы, содержащие символы A, C, E, F, L, V и Z, в два бинарных дерева поиска: одно из них должно иметь максимальную высоту, а второе — минимальную.
6. Проанализируйте бинарное дерево поиска, изображенное на рис. 10.42.

   6.1. Какое дерево получится после вставки узлов 80, 65, 75, 45, 5 и 25 в указанном порядке?

   6.2. Какое дерево получится, если после вставки указанных выше узлов, из него удалить узлы 50 и 20?
7. Изменится ли форма бинарного дерева поиска, если узел удалить из него и вставить обратно?
8. Какое дерево или деревья порождаются следующей последовательностью операторов?

```
typedef int TreeItemType; // Элемент, содержащийся в узле
BinaryTree tree1, tree2;

tree2.setRootData(9);
tree2.attachLeft(10);
tree2.attachRight(8);
tree2.getLeftSubtree().attachLeft(2);
tree2.getLeftSubtree().getLeftSubtree().attachLeft(5);
tree2.getLeftSubtree().getLeftSubtree().attachRight(3);
tree2.getRightSubtree().attachLeft(6);
tree2.getRightSubtree().attachRight(7);
tree1.setRootData(1);
tree1.attachLeft(2);
tree1.attachRight(3);

BinaryTree bTree(4, tree2, tree1);
```

9. Рассмотрите функцию *isLeaf()*, возвращающую значение *true*, если экземпляр класса *BinaryTree* состоит из единственного узла, т.е. из листа, и значение *false* — в противном случае.

9.1. Добавьте объявление функции *isLeaf* в класс *BinaryTree*, так чтобы функция стала доступной клиентам класса.

9.2. Напишите реализацию функции *isLeaf* внутри реализации класса *BinaryTree*.

9.3. Если бы функция *isLeaf* не была членом класса *BinaryTree*, мог бы клиент этого класса реализовать ее сам? Обоснуйте свой ответ.

10. Операция *searchTreeReplace(in replacementItem:KeyType):boolean* находит в бинарном дереве поиска элемент, поисковый ключ которого совпадает со значением аргумента *replacementItem*. Если дерево содержит такой элемент, операция заменяет его элементом *replacementItem*. Таким образом, соответствующая запись обновляется полностью.

10.1. Добавьте операцию *searchTreeReplace* в реализацию абстрактного бинарного дерева поиска с помощью указателей. Операция должна заменять элемент, не изменяя структуры дерева.

10.2. Реализуйте операцию *searchTreeReplace* в качестве клиента класса *BinarySearchTree*. Изменится ли при этом форма бинарного дерева?

11. Допустим, что во время обхода бинарного дерева поиска, изображенного на рис. 10.42, данные, содержащиеся в каждом посещенном узле, записываются в файл. Затем эти данные будут считаны из файла, и с помощью операции *seacrhtreeInsert* будет создано новое бинарное дерево поиска. В каком порядке следует записывать данные в файл, чтобы новое дерево имело прежнюю форму и содержало те же узлы? Что будет записано в файл после обхода дерева?

12. Изучите реализацию бинарного дерева поиска *bst* в виде массива. Конкретный пример такой реализации показан на рис. 10.11.

12.1. Изобразите массив, реализующий бинарное дерево, показанное на рис. 10.20, *а*.

12.2. Опишите результат каждой из указанных ниже операций. Для простоты можете считать, что элементами узлов являются имена, а не записи.

```
bst.searchTreeInsert("Дуг");
bst.searchTreeDelete("Нэнси", Success);
bst.searchTreeDelete("Боб", Success);
bst.searchTreeInsert("Сара");
```

12.3. Повторите задания 12.1 и 12.2 для дерева, изображенного на рис. 10.20, *б*.

12.4. Напишите алгоритм симметричного обхода бинарного дерева поиска, реализованного в виде массива.

13. Дубликатами абстрактного типа данных называются либо полностью идентичные элементы, либо элементы, записи которых содержат одинаковые поисковые ключи, а остальные поля могут быть разными. Если в бинарном дереве поиска допускаются дубликаты, нужно принять какое-то соглашение об их взаимосвязи. Элементы, являющиеся дубликатами корня, должны принадлежать либо левому поддереву, либо правому поддереву, причем это свойство должно выполняться для всех поддеревьев.

13.3. Почему это соглашение сильно влияет на эффективное использование бинарных деревьев поиска?

13.4. В главе утверждается, что элемент бинарного дерева поиска можно удалить, просто заменив его элементом, поисковый ключ которого либо непосредственно предшествует, либо непосредственно следует за поисковым ключом удаляемого элемента. Однако, если в дереве могут существовать дубликаты, выбор симметричного предшественника и симметричного преемника не может быть произвольным. Как указанное выше соглашение о дубликатах влияет на этот выбор?

14. Завершите трассировку итеративного алгоритма симметричного обхода, начало которой показано на рис. 10.15. Опишите содержание неявного стека в ходе обхода.

15. Реализуйте на языке C++ итеративный алгоритм симметричного обхода бинарного дерева (см. упражнение 14).

16. Учитывая рекурсивную природу бинарного дерева, сначала следует написать рекурсивное определение задачи, а затем переходить к написанию функции, работающей с бинарным деревом. В этом случае реализация функции на языке C++ становится очевидной.

Напишите рекурсивные определения следующих задач, касающихся произвольных бинарных деревьев. Реализуйте эти определения на языке C++. Должны ли эти функции быть членами класса *BinaryTree*? Для простоты можете считать, что элементами дерева являются целые числа и что в нем нет дубликатов.

16.1. Подсчитайте количество узлов дерева. (Подсказка: если дерево пусто, количество узлов равно 0. Если дерево не пусто, количество узлов равно 1 плюс количество узлов в левом поддереве корня, плюс количество узлов в правом поддереве корня.)

16.2. Вычислите высоту дерева.

16.3. Найдите максимальный элемент.

16.4. Найдите сумму элементов.

16.5. Найдите среднее значение элементов.

16.6. Найдите заданный элемент.

16.7. Определите, является ли один из элементов предком другого (т.е. принадлежит ли он поддереву другого элемента).

16.8. Определите, заполнен ли верхний уровень дерева, подсчитав максимально возможное количество элементов в верхнем уровне (см. упражнение 22.)

17. Рассмотрим непустое бинарное дерево, содержащее элементы двух типов: **максимальные** и **минимальные**. Пользуясь приведенными ниже инструкциями, можно определить значение минимаксного дерева.

- Если корень дерева является минимальным узлом, то значение дерева равно *минимуму* среди следующих чисел.
- Целое число, записанное в корне.
- Значение левого поддерева (если оно не пусто).
- Значение правого поддерева (если оно не пусто).
- Если корень дерева является максимальным числом, то значение дерева равно *максимуму* среди всех указанных выше чисел.

17.1. Вычислите значение минимаксного дерева, изображенного на рис. 10.46. Каждый узел этого дерева помечен его исходным значением.

17.2. Напишите на языке C++ программу, позволяющую представлять минимаксные деревья и вычислять их значения.

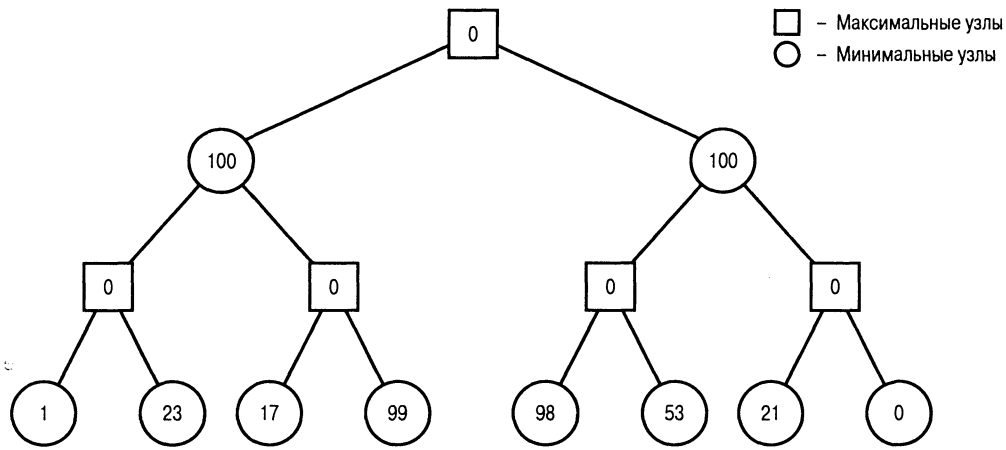


Рис. 10.46. Минимаксное дерево из упражнения 3

18. Определению бинарного дерева поиска соответствует несколько структур. Существует ли хотя бы одно бинарное дерево поиска, прямой порядок обхода которого совпадал бы с порядком элементов, перечисленных в заданном списке? Может ли существовать несколько таких деревьев?

19. Сколько разных форм может иметь n-арное дерево? Сколько разных форм может иметь бинарное дерево, состоящее из n узлов? (Напишите рекурсивные определения.)

20. Напишите псевдокод функции, определяющей диапазон бинарного дерева поиска. Иными словами, функция должна посетить все элементы, содержащие поисковый ключ, принадлежащий заданному диапазону значений (например, все числа от 100 до 1000).

21. Докажите теоремы 10.2 и 10.3 с помощью метода математической индукции.

22. Какое максимальное количество узлов может находиться на n-м уровне бинарного дерева? Обоснуйте свой ответ с помощью метода математической индукции. Используйте этот факт для выполнения следующих заданий.

    22.1. Перепишите формальное определение совершенного дерева, имеющего высоту h.

    22.2. Выведите замкнутый вид формулы $\sum_{i=1}^{h} 2^{i-1}$. В чем смысл этой формулы?

23. Докажите методом математической индукции, что бинарное дерево, состоящее из n узлов, содержит ровно $n+1$ пустых поддеревьев (или, в терминах языка C++, $n+1$ указателей, равных константе *NULL*).

24. Бинарное дерево называется **строго бинарным** (strictli binary), если каждый узел, не являющийся листом, имеет ровно два дочерних узла. Докажите, пользуясь индукцией по количеству листьев, что строго бинарное дерево, содержащее n листьев, имеет ровно $2n-1$ узлов.

25. Рассмотрим два итеративных алгоритма обхода бинарных деревьев, использующих внешний абстрактный тип данных для регистрации узлов. Оба алгоритма имеют следующий вид.

```
Поместить корень дерева в список регистрации
while (список регистрации не пуст)
{
   Удалить узел из списка регистрации и обозначить его числом n
   Посетить узел n
   if (узел n имеет левый дочерний узел)
      Поместить дочерний узел в список регистрации
   if (узел n имеет правый дочерний узел)
   Поместить дочерний узел в список регистрации
} // Конец оператора while
```

Разница между этими двумя алгоритмами заключается в способе выбора узла n, подлежащего удалению из абстрактного списка регистрации.

Алгоритм 1: удалить из списка регистрации узел, помещенный туда позже всех.

Алгоритм 2: удалить из списка регистрации узел, помещенный туда раньше всех.

25.1. В каком порядке каждый из этих алгоритмов обходит дерево, изображенное на рис. 10.19?

25.2. Опишите абстрактный тип данных, подходящий для регистрации узлов в каждом из этих алгоритмов. Какой абстрактный тип данных следует выбрать? Старайтесь экономить память. Кроме того, учтите, что дерево при обходе изменяться не должно.

26. Опишите способ записи бинарного дерева в файл, при котором восстанавливается его исходная форма. Сравните эффективность операций чтения и записи бинарного дерева и бинарного дерева поиска.

27. Реализуйте новый алгоритм удаления элементов из бинарного дерева поиска. Этот алгоритм должен отличаться от остальных в ситуации, когда узел N имеет два дочерних узла. Алгоритм заменяет узел N его правым дочерним узлом, как это происходит с узлами, имеющими лишь один дочерний узел. Затем дочерний узел удаленного узла N (вместе со всеми его подде-

ревьями) в качестве левого дочернего узла присоединяется к узлу, содержащему симметричного преемника поискового ключа узла *N*.

28. Напишите итеративные функции, реализующие вставку и удаление элементов бинарного дерева поиска.

29. Используя открытое наследование, создайте класс, производный от класса `BinarySearchTree`. Достаточно ли для решения этой задачи описания класса `BinaryTree` или его нужно модифицировать?

30. Если заранее известно, что заданный элемент при обходе бинарного дерева поиска встречается несколько раз, его поиск можно ускорить. Для этого можно предусмотреть внешний список регистрации, т.е. отслеживать указатель на узел, к которому в последний раз применялась какая-либо из операций над абстрактным бинарным деревом поиска. Реализовав такую регистрацию, можно перед выполнением операции проверить поисковый ключ элемента, посещаемого наиболее часто.

    Модифицируйте реализацию абстрактного бинарного дерева поиска, добавив в его класс новый член `lastAccessed`.

31. Дважды связанные списки вводились для того, чтобы вставлять и удалять элементы, не выполняя обхода списка. Для бинарных деревьев поиска аналогичная ситуация возникает, когда в их узлах хранятся указатели на родительские узлы. Иными словами, каждый узел, кроме корня, содержит указатель на своего родителя. Напишите операции вставки и удаления элементов для этих деревьев.

32. Узел дерева общего вида, например, изображенного на рис. 10.38, может иметь произвольное количество дочерних узлов.

    32.1. Опишите реализацию на языке C++ дерева общего вида, в котором каждый узел содержит массив указателей на дочерние узлы. Напишите рекурсивный метод прямого обхода дерева общего вида. Какие преимущества и недостатки имеет эта реализация?

    32.2. Проанализируйте реализацию дерева общего вида, описанную в главе. Каждый узел этого дерева содержит два указателя: левый указатель ссылается на старший дочерний узел, а правый — на следующий дочерний узел. Напишите рекурсивный метод прямого обхода такого дерева.

    32.3. Проанализируйте бинарное дерево *T*, в котором каждый узел имеет не больше двух дочерних узлов. Сравните реализацию дерева *T*, описанную в предыдущем задании, с представлением бинарного дерева, приведенным в главе. Упрощает ли какое-либо из этих представлений реализацию операций над абстрактным бинарным деревом? Одинаковы ли эти представления?

33. Реализуйте операцию симметричного обхода бинарного дерева поиска, позволяющую удалять посещенный элемент.

34. Напишите шаблонные классы `BinaryTree` и `BinarySearchTree`.

35. Добавьте в класс `BinaryTree` перегруженный оператор `==`.

Задания по программированию

1. Напишите реализацию абстрактного бинарного дерева поиска в виде динамического массива. Используйте структуру, изображенную на рис. 10.11.

2. Выполните задание 1 для совершенного бинарного дерева.

3. Напишите на языке C++ программу, обучающуюся по ответам “да” и “нет”. Например, программа может приобретать знания о животных, используя приведенный ниже диалог с пользователем. (Ответы пользователя выделены прописными буквами.)

```
Задумайте животное и я его отгадаю.
У него есть ноги? ДА
Это кошка? ДА
Я победила! Играем дальше? ДА

Задумайте животное и я его отгадаю.
У него есть ноги? НЕТ
Это змея? ДА
Я победила! Играем дальше? ДА

Задумайте животное и я его отгадаю.
У него есть ноги? НЕТ
Это змея? НЕТ
Я сдаюсь! Что это? ЧЕРВЯК
Пожалуйста, наберите на клавиатуре вопрос с положительным
ответом для червяка и отрицательным для змеи:
ОНО ЖИВЕТ ПОД ЗЕМЛЕЙ?
Играем дальше? ДА

Задумайте животное и я его отгадаю.
У него есть ноги? НЕТ
Оно живет под землей? НЕТ
Это змея? НЕТ
Я сдаюсь! Что это? РЫБА
Пожалуйста, наберите на клавиатуре вопрос с положительным
ответом для рыбы и отрицательным для змеи:
ОНО ЖИВЕТ В ВОДЕ?
Играем дальше? НЕТ

До свидания.
```

Программа начинает свою работу, практически ничего не зная о животных. Она знает лишь, что у кошки есть лапы, а у змеи — нет. Когда в следующий раз программа неправильно угадает змею, она попросить ввести вопрос, позволяющий различать змей и червяков.

Программа создает бинарное дерево вопросов и ответов. Ответ “ДА” записывается в левый дочерний узел по отношению к узлу, содержащему вопрос, а ответ “НЕТ” — в правый.

4. Напишите программу, позволяющую работать с фамилиями, именами, адресами и номерами телефонов ваших друзей и родственников, т.е. имитирующую вашу записную книжку. Программа должна предусматривать операции вставки, удаления, модификации и поиска данных. Поисковым ключом считается фамилия человека, причем она должна быть уникальной. Записи, содержащиеся в записной книжке, должны храниться в файле.

Разработайте класс записей о людях в записной книжке, а также класс, описывающий саму записную книжку. Класс должен содержать бинарное дерево поиска, хранящее информацию о знакомых и родственниках.

Поставленную задачу можно усложнить, добавив в базу данных дни рождения и предусмотрев операцию, позволяющую вывести список людей, удов-

летворяющих конкретному запросу. Например, программа должна выводить на экран список людей, родившихся в заданном месяце или живущих в конкретном городе. Кроме того, следует предусмотреть возможность вывода на экран всех записей, содержащихся в базе данных.

5. Напишите программу, позволяющую записывать и извлекать номера телефонов. Разработайте пользовательский интерфейс, предусмотрев следующие операции.
   - Вставка: добавить в телефонную книгу указанные имя и номер.
   - Удаление: удалить из телефонной книги имя и номер по указанному имени.
   - Найти: отыскать в телефонной книге имя и номер по указанному имени.
   - Изменить: изменить номер телефона, пользуясь указанным именем и новым номером телефона.
   - Выйти: прекратить работу программы, сохранив телефонную книгу в файле.
   - Разработайте и реализуйте класс *Person*, представляющий имя и номер телефона. Экземпляры этого класса будут храниться в телефонной книге.
   - Разработайте и реализуйте класс *Book*, имитирующий телефонную книгу. Этот класс должен содержать бинарное дерево поиска, в котором хранятся имена и телефонные номера.
   - Добавьте функции-члены, записывающие информацию в текстовый файл и считывающие их оттуда.
   - Разработайте и реализуйте класс *UserInterface*, предоставляющий возможности пользовательского интерфейса.
   - В начале своей работы программа должна считывать данные из текстового файла, а перед завершением — записывать данные обратно.

Таблицы и очереди с приоритетами

В этой главе ...

Абстрактная таблица
- Выбор способа реализации
- Реализация абстрактной таблицы в виде упорядоченного массива
- Реализация абстрактной таблицы в виде бинарного дерева поиска

Абстрактная очередь с приоритетами: вариант абстрактной таблицы
- Кучи
- Реализация абстрактной очереди с приоритетами в виде кучи
- Пирамидальная сортировка

Резюме
Предупреждения
Вопросы для самопроверки
Упражнения
Задания по программированию

Введение. В этой главе рассматриваются абстрактные таблицы, предназначенные для управления данными по значению. Представлено несколько реализаций таблиц в виде массивов, связанных списков и бинарных деревьев поиска, описаны их преимущества и недостатки.

Чтобы правильно выбрать способ реализации абстрактного типа данных, нужно оценить его эффективность. В главе проведен сравнительный анализ эффективности реализаций таблиц в виде массивов и с помощью указателей. Показано, что во многих приложениях эти способы недостаточно эффективны, поэтому в главе рассмотрен более сложный способ реализации таблиц — в виде бинарных деревьев поиска.

В главе рассматривается также важная разновидность абстрактной таблицы — очередь с приоритетами. Этот абстрактный тип данных позволяет легко извлекать и удалять элемент, имеющий максимальное значение. Несмотря на то что абстрактную очередь с приоритетами можно реализовать с помощью бинарного дерева поиска, существует более простая структура данных, называемая кучей, которая больше подходит для этой цели.

Абстрактная таблица

В предыдущей главе мы уже встречались с абстрактными типами данных, ориентированными на значения, когда выполняли следующие операции.

- Вставить элемент, содержащий элемент x.
- Удалить элемент, содержащий элемент x.
- Найти элемент, содержащий элемент x.

Приложения, в которых необходимо выполнять операции, ориентированные на значение, встречаются намного чаще, чем можно себе представить. Например, для решения указанных ниже задач нужно выполнять именно такие операции.

- *Найти номер телефона Джона Смита.*
- *Удалить всю информацию о сотруднике под номером 12908.*

Название абстрактного типа данных часто относится к целому семейству, обладающему схожими свойствами. Например, имя "стек" может относиться к стопке тарелок. А какие ассоциации вызывает слово "таблица"? Если бы этот вопрос был задан до того, как вы стали читать эту книгу, возможно, вы бы ответили: "Мой любимый кофейный столик из красного дерева" (слово *table* переводится и как "стол", и как "таблица". — *Прим. ред.*). Однако, прочитав большую половину этой книги, вы должны ответить: "Таблица главных городов мира", изображенная на рис. 11.1.

В этой таблице содержится информация о перечисленных городах. Ее структура позволяет просматривать эту информацию. Например, если мы хотим узнать, сколько людей живет в Лондоне, можно просмотреть столбец, в котором перечисляются названия городов сверху вниз, пока не увидим название "Лондон". Поскольку города перечислены в алфавитном порядке, можно имитировать бинарный поиск. Найдем середину таблицы, определим, в какой половине находится Лондон, и рекурсивно применим бинарный поиск к этой половине таблицы. Как мы уже знаем, бинарный поиск намного эффективнее, чем просмотр всей таблицы от начала до конца.

Однако, если мы хоти найти самый большой город Италии, у нас нет другого выбора, кроме просмотра всей таблицы. Алфавитный порядок перечисления названий городов нам ничем помочь не может. Структура таблицы позволяет легко найти город по его названию, но любой другой запрос потребует полного просмотра всей таблицы.

Город	Страна	Население
Афины	Греция	2 500 000
Барселона	Испания	1 800 000
Каир	Египет	9 500 000
Лондон	Англия	9 400 000
Нью-Йорк	США	7 300 000
Париж	Франция	2 200 000
Рим	Италия	2 800 000
Торонто	Канада	3 200 000
Венеция	Италия	300 000

Рис. 11.1. Обычная таблица городов

Для идентификации своих элементов абстрактная таблица использует ключи поиска

Абстрактная таблица (ADT table), или **словарь** (dictionary), также позволяет легко просматривать информацию и предусматривает для этого специальную операцию. Обычно элементами абстрактной таблицы являются записи, содержащие несколько полей. Применение поисковых ключей позволяет намного облегчить извлечение элементов из таблицы. Например, в таблице городов поисковым ключом можно считать поле `City`. Можно изобрести реализации таблиц, позволяющие быстро извлекать элементы, поисковые ключи которых совпадают с указанным значением. Однако, если нужно извлечь информацию, основываясь на значении поля, которое не является ключом поиска, придется просмотреть всю таблицу. Следовательно, при выборе поискового ключа нужно придерживаться следующего правила.

Необходимо упорядочить данные так, чтобы облегчить поиск элемента по заданному значению его поискового ключа.

Основные операции над абстрактной таблицей перечислены ниже.

ОСНОВНЫЕ ПОНЯТИЯ

Операции над абстрактной таблицей

1. Создать пустую таблицу.
2. Уничтожить таблицу.
3. Определить, пуста ли таблица.
4. Определить количество элементов в таблице.
5. Вставить в таблицу новый элемент.
6. Удалить из таблицы элемент, поисковый ключ которого совпадает с заданным значением.
7. Извлечь из таблицы элемент, поисковый ключ которого совпадает с заданным значением.
8. Обойти элементы таблицы в порядке следования их поисковых ключей.

Для простоты будем предполагать, что все элементы в таблице имеют разные поисковые ключи. Следовательно, вставка нового элемента, поисковый ключ которого совпадает с поисковым ключом одного из элементов таблицы, невозможна. Операции над абстрактной таблицей, содержащей элементы с разными поисковыми ключами, описаны в следующем псевдокоде. UML-диаграмма класса `Table` приведена на рис. 11.2.

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операций над абстрактной таблицей

```
// TableItemType — тип элементов, хранящихся в таблице

+createTable()
// Создает пустую таблицу.

+destroyTable()
// Уничтожает таблицу.

+tableIsEmpty():boolean {query}
// Определяет, пуста ли таблица.

+tableLength():integer {query}
// Определяет количество элементов в таблице.

+tableInsert(in newItem:TableItemType) throw TableException
// Вставляет в таблицу элемент newItem. Поисковые ключи элементов
// этой таблицы должны отличаться от поискового ключа элемента
// newItem. Если вставка невозможна, генерируется исключительная
// ситуация TableException.

+tableDelete(in searchKey:KeyType) throw TableException
// Удаляет из таблицы элемент, поисковый ключ которого совпадает
// со значением аргумента searchKey. Если такого элемента нет,
// функция генерирует исключительную ситуацию TableException.

+tableRetrieve(in searchKey:KeyType, out tableItem:TableItemType)
                throw TableExeption {query}
// Извлекает из таблицы элемент, поисковый ключ которого
// совпадает со значением аргумента searchKey. Если такого
// элемента нет, функция генерирует исключительную ситуацию
// TableException.

+traverseTable(in visit:FunctionType)
// Обходит таблицу в порядке следования поисковых ключей
// и один раз вызывает для каждого элемента функцию visit.
```

Возможны разные типы операций над таблицей

Следует иметь в виду, что этот набор данных представляет собой только один из многих вариантов операций над таблицей. Клиент может определять новые операции или модифицировать прежние в зависимости от потребностей конкретного приложения.

В рассмотренном примере предполагалось, что элементы таблицы имеют разные поисковые ключи

Например, в перечисленных выше операциях предполагается, что в таблице не могут содержаться два элемента, имеющие одинаковые поисковые ключи. Однако во многих приложениях это условие не выполняется.

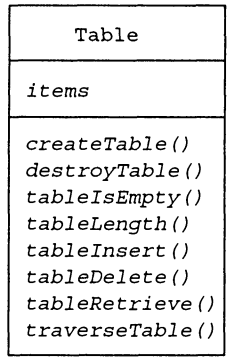


Рис. 11.2. UML-диаграмма класса Table

В других таблицах элементы могут иметь одинаковые поисковые ключи

Если элементы могут иметь одинаковые поисковые ключи, следует модифицировать операции, чтобы устранить неоднозначность. Например, какой элемент должна возвращать операция *tableRetrieve*, если в таблице есть несколько элементов с одинаковыми ключами поиска? Чтобы ответить на этот вопрос, нужно уточнить определение операции *tableRetrieve*.

Хотя в некоторых приложениях можно обойтись лишь операциями *tableInsert*, *tableDelete* и *tableRetrieve*, в большинстве случаев их оказывается недостаточно. Например, с их помощью нельзя вывести на экран все элементы таблицы, поскольку невозможно извлечь элемент, не зная значения его поискового ключа. Как известно, обход таблицы определяется порядком следования поисковых ключей ее элементов, поэтому вывести на экран ее содержимое оказывается невозможно.

Операция tableTraverse посещает все элементы таблицы в указанном порядке

Операция *tableTraverse* посещает все элементы таблицы по одному разу. Определяя эту операцию, следует задать порядок обхода элементов. Обычно порядок обхода таблицы определяется порядком следования поисковых ключей ее элементов, но иногда он может быть произвольным. Как мы увидим в дальнейшем, операция *tableTraverse* может повлиять на выбор способа реализации таблицы.

Как и операции обхода, описанные в предыдущих главах, операция *tableTraverse* получает в качестве аргумента функцию *visit*. В зависимости от действий, выполняемых этой функцией, смысл операции *tableTraverse* может изменяться. Проиллюстрируем эту изменчивость тремя небольшими примерами, связанными с таблицей городов, рассмотренной нами выше.

Для реализации таблицы очень важно правильно выбрать поисковый ключ. Необходимо, чтобы он всегда оставался постоянным. После изменения значения поискового ключа элемент, содержащийся в таблице, можно больше никогда не найти. Таким образом, модификацию поискового ключа следует запретить. Представим элемент таблицы в виде класса, содержащего в качестве своих членов поисковый ключ и методы доступа к нему. Этот класс уже рассматривался нами в главе 10

```
#include <string>
using namespace std;
typedef string KeyType;

class KeyedItem
```

```
{
public:
   KeyedItem() {};
   KeyedItem( const KeyType& keyValue):
      searchKey(keyValue) { }
   KeyType getKey() const
   {
     return searchKey;
   } // end getKey
private:
   KeyType searchKey;
}; // Конец класса
```

Обратите внимание, что потомки класса *KeyedItem* унаследуют лишь конструктор, позволяющий инициализировать поисковый ключ. Следовательно, значение поискового ключа уже созданного элемента модифицировать невозможно, что и требовалось доказать.

Допустим, что элементами таблицы являются экземпляры следующего класса.

```
#include <string>
using namespace std;

class City
{
public:
. . .
   // Функции, обеспечивающие доступ к закрытым членам класса
private:
   string cityName; // Название города
   string country;  // Название страны
   int pop; // Численность населения, живущего в городе
}; // Конец класса
```

Сформулируем следующие задачи.

Задачи, при решении которых используется класс City

- *Вывести на экран в алфавитном порядке названия городов и численность их населения.*
- *Увеличить на 10% численность населения каждого города.*
- *Удалить из таблицы все города, численность населения которых меньше 1000000 человек.*

При решении каждой задачи предполагается, что поисковым ключом является поле *cityName*. Класс *City* содержит всю информацию о городе, включая его название (возвращаемое унаследованным методом *key*), страну и численность населения. Вот как выглядит его определение.

```
class City : public KeyedItem
{
public:
   City() { }
   City(const string& name,
        const string& ctry,
        const int& num)
        :KeyedItem(name), country(ctry), pop(num) { }
   string cityName() const;
   int getPopulation() const;
```

```
    void setPopulation(int newPop);
private:
    // Поисковым ключом является название города
    string country; // Название страны
    int pop; // Численность населения города
}; // Конец класса
```

В первой задаче требуется вывести на экран в алфавитном порядке все названия городов. Следовательно, функция *traverseTable* должна посетить все элементы таблицы в алфавитном порядке. Для решения первой задачи функции *traverseTable* в качестве аргумента передается функция *displayItem*.

Первая задача

```
displayItem(in anItem:TableItemType)

   Display anItem.cityName()
   Display anItem.getPopulation()
```

Порядок обхода, установленный функцией *traverseTable*, при решении второй и третьей задач не имеет никакого значения. Для решения второй задачи функции *traverseTable* в качестве аргумента передается функция *updatePopulation*.

Вторая задача

```
updatePopulation(
          inout anItem:TableItemType)

   anItem.setPopulation(1.1 * anItem.getPopulation())
```

Для решения третьей задачи функции *traverseTable* в качестве аргумента передается функция *deleteSmall*.

Третья задача

```
deleteSmall(inout t:Table,
          in anItem::TableItemType)

   if (anItem.getPopulation() < 1,000,000)
       t.tableDelete(anItem)
```

Однако эта задача не так проста, как может показаться. Удаляя элемент, мы изменяем таблицу в процессе обхода. Какой элемент функция *tableTraverse* посетит следующим? Очевидно, что она должна посетить элемент, следующий непосредственно за удаленным. Но не пропустит ли функция *tableTraverse* этот элемент? Задача обхода таблицы с одновременным удалением ее элементов представляет собой довольно сложное упражнение, которое мы оставляем читателям.

Выбор способа реализации

В предыдущих главах для реализации абстрактных типов данных мы выбирали массивы или пользовались указателями. Иными словами, элементы абстрактной структуры данных хранились либо в массиве, либо в связанном списке. Такие реализации называются **линейными** (linear), поскольку элементы в этих структурах следуют один за другим. Они напоминают собой список, изображенный на рис. 11.1.

Четыре категории линейных реализаций

Линейные реализации таблицы разделяются на четыре категории.

- Неупорядоченный массив.
- Неупорядоченный связанный список.
- Упорядоченный (по ключу) массив.
- Упорядоченный (по ключу) связанный список.

Элементы в неупорядоченных реализациях сохраняются в произвольном порядке; их можно вставлять в любое место. Однако вставка элемента в упорядоченную реализацию осуществляется по значению ключа поиска. Базовая структура линейных реализаций таблицы в виде массива и связанного списка показана на рис. 11.3. В обеих реализациях предусматривается счетчик, отслеживающий количество элементов в таблице. Как мы увидим далее, упорядоченные и неупорядоченные реализации абстрактной таблицы имеют свои преимущества и недостатки.

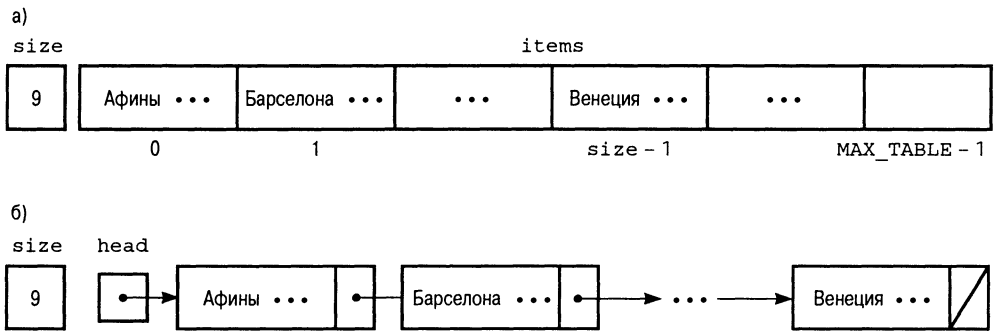


Рис. 11.3. Элементы двух упорядоченных реализаций абстрактной таблицы, содержащей данные, показанные на рис. 11.1: а) в виде массива; б) в виде связанного списка

Реализация таблицы в виде бинарного дерева поиска является нелинейной

Существуют и другие способы реализации таблицы. Например, абстрактную таблицу можно реализовать в виде абстрактного списка, абстрактного упорядоченного списка или бинарного дерева поиска. Реализация таблицы в виде бинарного дерева поиска, как показано на рис. 11.4, является **нелинейной**, что дает ей определенное преимущество над линейными реализациями. К ее преимуществам относится возможность повторного применения абстрактного дерева поиска, описанного в главе 10. Реализации, основанные на абстрактном списке и абстрактном упорядоченном списке, также обладают этим свойством. Читатели могут сами в этом убедиться.

Какие операции нужны в конкретном приложении

Основная цель этой главы — показать, как особенности конкретного приложения влияют на выбор способа реализации абстрактной таблицы. В ходе обсуждения этой темы мы разовьем тезисы, изложенные в главе 9 в разделе "Перспективы". В некоторых приложениях достаточно предусмотреть операции над абстрактными таблицами, определенные выше, другие могут использовать лишь их подмножество или, наоборот, нуждаться в дополнительных операциях. Прежде чем выбрать реализацию абстрактной таблицы, необходимо тщательно проанализировать, какие операции потребуются в конкретном приложении. Разумеется, довольно заманчиво предусмотреть сразу все возможные операции над таблицами, однако эта стратегия неверна, поскольку в разных приложениях разные операции выполняются с разной эффективностью. Следовательно, если включить в приложение операцию, которая никогда не используется, общая производительность программы снизится.

Как часто выполняется каждая из операций

Определив, какие операции необходимы для работы данного приложения, нужно приближенно оценить, как часто выполняется каждая из них. Одни приложения одинаково часто выполняют все операции, другие — наоборот. Например, при работе с таблицей главных городов мира, изображенной на рис. 11.1, операция извлечения выполняется намного чаще, чем опера-

ции вставки и удаления. Следовательно, если вставка элементов выполняется редко, с относительной неэффективностью этой операции можно смириться, поскольку другие операции обладают высоким быстродействием. Разумеется, как указывалось в главе 9, если от результата выполнения операции зависит очень многое, она должна быть эффективной, даже если используется редко. Необходимые операции, ожидаемая частота и длительность их выполнения представляют собой факторы, влияющие на выбор способа реализации абстрактного типа данных для конкретного приложения. Однако не следует забывать и о других факторах, не имеющих отношения к эффективности (см. главу 9).

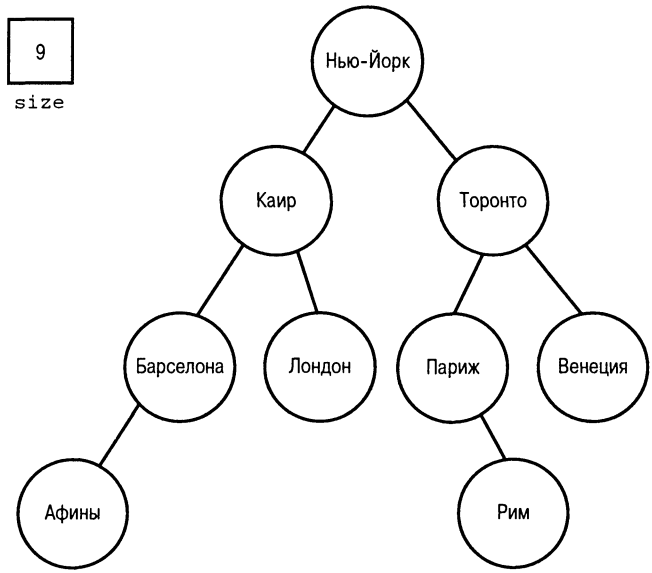


Рис. 11.4. Элементы бинарного дерева поиска, реализующего абстрактную таблицу, содержащую данные, показанные на рис. 11.1

Рассмотрим несколько сценариев, в каждом из которых требуются разные операции над таблицами. Анализ разных реализаций абстрактной таблицы проиллюстрирует несколько основных понятий, связанных с анализом алгоритмов. Мы рассмотрим критерии выбора реализаций таблиц для конкретных приложений, в которых поддерживается высокая эффективность выполняемых операций.

Сценарий А: вставка и обход в произвольном порядке. Женский клуб, в котором состоит Мэри, планирует собрать деньги на благотворительность. Утомившись от предыдущих благотворительных кампаний, Мэри собрала совещание, чтобы выработать новую стратегию сбора денег. Члены клуба высказывали свои идеи, а Мэри записывала их в таблицу, чтобы потом распечатать их на принтере. Допустим, что структура этого отчета значения не имеет — элементы могут быть как упорядочены, так и не упорядочены. Кроме того, допустим, что операции извлечения, удаления и обхода таблицы в определенном порядке либо не выполняются вообще, либо выполняются очень редко и поэтому не влияют на выбор ее реализации.

В данном приложении упорядоченность элементов никаких преимуществ не дает. На самом деле, если элементы таблицы записаны в произвольном порядке, операция `tableInsert`

Произвольный порядок записей позволяет эффективно выполнять операции

может быть довольно эффективной. Вставка элемента в неупорядоченную таблицу может выполняться в любое место, например в последнюю ячейку массива `item[size]`. Результат этой операции показан на рис. 11.5, *а*. Работая со связанным списком, новый элемент можно вставлять в его начало. Как показано на рис. 11.5, *б*, указатель на голову списка ссылается на новый элемент, а указатель нового элемента ссылается на элемент, который до вставки был первым элементом массива. Итак, вставка нового элемента в неупорядоченную реализацию таблицы выполняется достаточно быстро, причем сложность операции `tableInsert` оценивается величиной O(1), т.е. не зависит от размера таблицы.

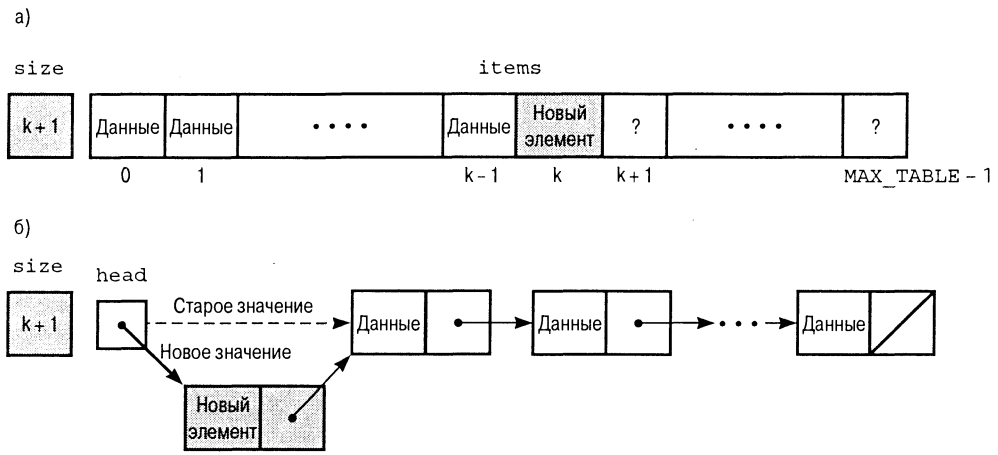


Рис. 11.5. Вставка элемента в неупорядоченную линейную реализацию таблицы: а) в массив; б) в связанный список

Сравнение массива и связанного списка

Что предпочесть, реализуя абстрактную таблицу, — массив или связанный список? Если размер таблицы невозможно оценить заранее, следует использовать динамическую память. Таблица, в которую Мэри записывает предложения членов клуба, относится именно к такой категории. Однако если размер таблицы не может быть слишком большим[1], выбор становится делом вкуса. Массив экономнее использует память, чем связанный список, поскольку для него не нужен внешний указатель. Однако размер этого указателя по сравнению с размером всей структуры данных настолько незначителен, что во многих ситуациях им можно пренебречь.

Не следует ли выбрать бинарное дерево поиска для реализации нашей таблицы? Для этого понадобилось бы упорядочить элементы таблицы, что в данном случае совершенно излишне. Как показано в главе 10, вставка элемента в бинарное дерево в среднем оценивается величиной O(log *n*).

Сценарий Б: извлечение. Используя тезаурус текстового процессора для поиска синонима какого-либо слова, мы применяем операцию извлечения. Если абстрактная таблица представляет собой тезаурус, то каждый ее элемент является записью, содержащей само слово (поисковый ключ) и его синоним. Поскольку операция извлечения слова из тезауруса производится часто, необходимо, чтобы

[1] В разделе "Реализации списка в виде массива и на основе указателей" главы 4 показано, как оценка среднего и максимального количества элементов влияет на реализацию абстрактного типа данных в виде массива.

реализация таблицы позволяла осуществлять эффективный поиск элемента по заданному ключу. Обычно тезаурус изменить нельзя, поэтому операции вставки и удаления не нужны.

Реализация таблицы в виде упорядоченного массива позволяет применить бинарный поиск записи. Однако если применить связанный список, то при извлечении записи придется проходить тезаурус от начала до искомого слова. Учитывая, что бинарный поиск намного эффективнее последовательного, следует отдать предпочтение массиву. В связи с этим возникают два вопроса.

Реализация таблицы в виде упорядоченного массива позволяет применить бинарный поиск

1. Возможен ли бинарный поиск в связанном списке?
2. Насколько бинарный поиск элемента в упорядоченном массиве эффективнее, чем последовательный поиск в связанном списке?

Вопросы

Можно ли выполнить бинарный поиск элемента в связанном списке? Да, но этот алгоритм слишком неэффективен. Рассмотрим самый первый шаг этого алгоритма.

Бинарный поиск в связанном списке слишком неэффективен

Найти "середину" таблицы

Как найти середину связанного списка, состоящего из n элементов? Можно обойти список с самого начала, подсчитывая количество пройденных элементов, пока их количество не станет равным $n/2$. Однако, как мы убедимся, ответив на второй вопрос, уже этот первый шаг потребует больше времени, чем весь алгоритм бинарного поиска элемента в упорядоченном массиве. Более того, та же проблема поиска "среднего" элемента будет возникать на каждом шаге рекурсивного алгоритма. Таким образом, алгоритм бинарного поиска элемента в связанном списке слишком неэффективен. Этот вывод чрезвычайно важен.

Однако в массиве `items`, состоящем из n элементов, средний элемент находится в ячейке $n/2$, причем к нему существует прямой доступ. Следовательно, алгоритм бинарного поиска элемента в массиве выполняется намного быстрее алгоритма, в котором необходимо просматривать каждый элемент таблицы. Что значит "намного быстрее"? Если бинарный поиск выполнить невозможно, придется просматривать каждый элемент таблицы, пока не будет обнаружен искомый элемент, содержащий заданный ключ, либо пока не выяснится, что такого элемента в массиве нет. Иными словами, если таблица содержит n элементов, возможно, придется просмотреть все n элементов, т.е. сложность такого поиска оценивается величиной $O(n)$. Напомним, что сложность бинарного поиска в худшем случае имеет порядок $O(\log_2 n)$, а алгоритмы, имеющие сложность $O(\log_2 n)$, намного эффективнее алгоритмов, эффективность которых оценивается величиной $O(n)$. Например, $\log_2 1024=10$, а $\log_2 1048576=20$. Для больших таблиц преимущество бинарного поиска становится чрезвычайно важным.

Поскольку тезаурус может быть большим, следует выбрать реализацию, которая допускает эффективный бинарный поиск элемента. Это сразу исключает из рассмотрения связанный список. Следовательно, лучше выбрать упорядоченный массив.

Если максимальный размер таблицы известен заранее, для эффективного выполнения операции извлечения элемента подходит упорядоченный массив

Для приложений, в которых часто выполняется операция извлечения, хорошо подходит реализация таблицы в виде бинарного дерева поиска. Если дерево сбалансировано, для обна-

Если максимальный размер таблицы заранее не известен, используйте бинарное дерево поиска

ружения элемента в бинарном дереве поиска понадобится выполнить O(log n) операций. Поскольку тезаурус изменяться не может, его лучше реализовать в виде сбалансированного дерева, тем самым гарантируя эффективный поиск элемента. Хотя указатели на бинарное дерево поиска занимают определенную часть памяти, их размер ничтожен по сравнению с размером тезауруса.

Сценарий В: вставка, удаление, извлечение и обход в определенном порядке. Представим себе электронный каталог библиотечных книг. Для доступа к его записям читатели применяют операцию извлечения. В свою очередь, библиотекари обновляют каталог, используя операции вставки и удаления, а при записи каталога в файл осуществляют его полный обход. Совершенно ясно, что операции извлечения записей выполняются чаще других, однако остальные операции также нельзя полностью игнорировать. (В противном случае получился бы сценарий Б!)

Чтобы вставить в таблицу элемент, ключ которого имеет значение *X*, сначала нужно определить его место. Аналогично, чтобы удалить из таблицы элемент, ключ которого имеет значение *X*, его сначала нужно найти. Итак, для выполнения операций `tableInsert` и `tableDelete` необходимо выполнить следующие действия.

1. Найти соответствующую ячейку в таблице.
2. Вставить (или удалить) эту ячейку.

Операции вставки и удаления выполняют два действия

Выполнение шага 1 становится намного эффективнее, если таблица реализована в виде массива, а не связанного списка. В этом случае, для того чтобы найти ячейку, в которую следует вставить элемент *X*, или элемент, который нужно удалить, можно применить алгоритм бинарного поиска. Однако мы уже знаем, что для связанного списка алгоритм бинарного поиска неэффективен. Кроме того, при анализе сценария Б было показано, что бинарный поиск элемента в массиве осуществляется намного быстрее.

Для выполнения первого шага используйте массив

Итак, благодаря тому, что массив позволяет применять бинарный поиск элемента, он лучше всего подходит для выполнения шага 1 в операции `tableInsert` и `tableDelete`. Однако для выполнения шага 2, связанного с фактическими операциями вставки и удаления элементов, предпочтительнее применять связанный список.

Для выполнения второго шага используйте связанный список

Если таблица реализуется в виде упорядоченного массива, при вставке и удалении приходится сдвигать ее элементы, чтобы освободить место для нового элемента (см. рис. 11.6, *а*) или заполнить образовавшуюся брешь, причем в худшем случае придется выполнить сдвиг каждого элемента массива. Если таблица представляет собой связанный список, для выполнения второго шага понадобится лишь изменить значения не более двух указателей, как показано на рис. 11.6, *б*.

При вставке и удалении элементов упорядоченного массива приходится сдвигать данные

Анализируя шаги 1 и 2, можно убедиться, что в случае упорядоченного массива и упорядоченного связанного списка операции `tableInsert` и `tableDelete` выполняются за одинаковое время, поскольку их сложность оценивается величиной O(n). Приходится признать, что ни одна из этих реализаций особой эффективностью не отличается. В то же время реализация таблицы в виде бинарного дерева поиска объединяет в себе лучшие особенности обеих линейных реализаций. Поскольку она использует указатели, сдвига данных удается избежать, и размер таблицы может динамически изменяться по мере надобности. Кроме того, операция извлечения элементов из бинарного дерева поиска также выполняется достаточно эффективно.

Эффективность упорядоченных линейных реализаций практически одинакова, но не приемлема

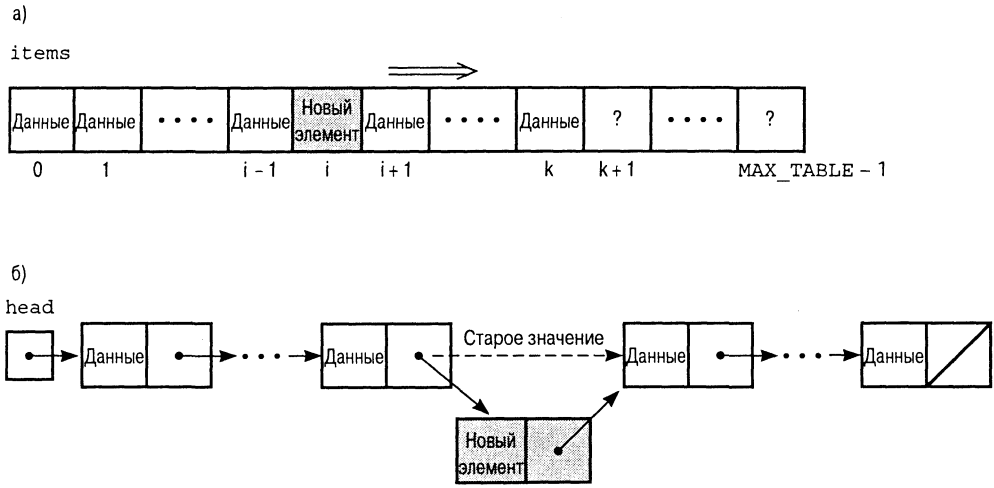


Рис. 11.6. Вставка элемента в упорядоченную линейную реализацию: а) в массив; б) в связанный список

Итоги. Реализация абстрактной таблицы в виде неупорядоченного массива позволяет эффективно выполнять операцию вставки элемента в конец массива. Однако удаление, как правило, вынуждает сдвигать элементы, чтобы в массиве не было дыр. Поскольку элементы неупорядочены, операция извлечения сводится к последовательному перебору.

Реализация абстрактной таблицы в виде упорядоченного массива вынуждает осуществлять сдвиг элементов при выполнении каждой операции вставки и удаления. В то же время, операция извлечения элементов позволяет применить эффективный бинарный поиск.

Реализация абстрактной таблицы в виде упорядоченного связанного списка требует выполнения последовательного поиска, но при этом не нужно сдвигать элементы при вставке и удалении. При извлечении элемента также осуществляется последовательный поиск.

Несмотря на определенные недостатки, для реализации таблицы вполне можно применять линейные структуры данных

Хотя линейные структуры довольно примитивны и малоэффективны, они оказываются полезными во многих приложениях. Поскольку линейные структуры данных просты и понятны, их можно применять для реализации таблиц, состоящих из небольшого количества элементов. В таких ситуациях эффективность не имеет большого значения, и на первый план выходят простота и ясность реализации. Даже если таблица велика, линейные структуры данных оказываются вполне приемлемыми, если таблица не упорядочена, а операция удаления применяется редко.

Наилучшей реализацией таблицы является бинарное дерево поиска

И все же самой лучшей реализацией таблицы является нелинейное бинарное дерево поиска. Если бинарное дерево поиска, содержащее n узлов, имеет минимальную высоту, т.е. ее высота равна $[\log_2(n+1)]$, оно позволяет эффективно реализовывать абстрактные таблицы в тех случаях, когда линейные структуры абсолютно неприемлемы. Эффективность первого шага операций извлечения, вставки и удаления в такой реализации сравнима с эффективностью бинарного поиска. Кроме того, связанные списки позволяют легко изменять размеры таблиц. Такая реализация позволят эффективно выполнять и

второй шаг операций вставки и удаления элементов: фактически для этого нужно лишь изменить значения нескольких указателей (и выполнить переход к симметричному преемнику, если удаленный узел имел два дочерних узла). При этом никакого сдвига данных делать не нужно. Таким образом, реализация таблицы в виде бинарного дерева поиска объединяет в себе лучшие особенности обеих линейных реализаций, устраняя их недостатки.

Сбалансированное дерево поиска позволяет увеличить эффективность операций над абстрактной таблицей

Однако, как показано в предыдущей главе, высота бинарного дерева поиска зависит от порядка, в котором выполняются операции вставки и удаления элементов, и может быть сравнима с величиной n. Если операции вставки и удаления выполняются в произвольном порядке, высота бинарного дерева близка к минимальной. Однако следить за изменением высоты дерева и соответствующим снижением эффективности операций не стоит. Вместо этого лучше применить методы балансировки бинарного дерева поиска, описанные в главе 12, которые позволяют сохранять высоту дерева близкой к величине $\log_2 n$.

На рис. 11.7 приведены средние оценки эффективности операций вставки, удаления, извлечения и обхода для разных реализаций абстрактной таблицы.

	Вставка	Удаление	Извлечение	Обход
Неупорядоченный массив	O(1)	O(n)	O(n)	O(n)
Неупорядоченный связанный список	O(1)	O(n)	O(n)	O(n)
Упорядоченный массив	O(n)	O(n)	O(log n)	O(n)
Упорядоченный связанный список	O(n)	O(n)	O(n)	O(n)
Бинарное дерево поиска	O(log n)	O(log n)	O(log n)	O(n)

Рис. 11.7. Средние оценки эффективности операций над абстрактной таблицей при разных реализациях

Реализация абстрактной таблицы в виде упорядоченного массива

Причины для изучения линейных реализаций: перспективы применения, эффективность и мотивация

Если бинарное дерево поиска настолько эффективно реализует абстрактную таблицу, зачем рассматривать ее линейные реализации? Существуют три причины. Первая и главная — перспективы применения. В главе 9 мы уже предостерегали читателей от излишне строгого анализа задач. Если размер задачи невелик, эффективность ее возможных решений практически одинакова. В частности, если размер таблицы мал, следует предпочесть ясную и понятную линейную реализацию.

Вторая причина — эффективность. В некоторых ситуациях линейные структуры данных вполне эффективны. Например, для сценария А, в котором доминируют операции вставки и обхода в произвольном порядке, лучше всего подходит именно линейная реализация. Для сценария Б, в котором доминирует операция извлечения, вполне приемлем упорядоченный массив, если максимальное количество элементов таблицы заранее известно. В этих ситуациях на первый план выходят простота и ясность реализации, которой характеризуются именно линейные структуры.

Третья причина — мотивация. Анализируя сценарии, в которых линейные реализации оказываются неэффективными, мы вынуждены придумывать новые структуры и рассматривать другие способы реализации, например бинарное де-

рево поиска. Фактически сравнение разных способов реализации позволяет лучше понять их особенности.

Ниже приведена программа, реализующая абстрактную таблицу в виде упорядоченного массива. Предполагается, что в таблице нет двух элементов с одинаковыми ключами поиска. Снять эти ограничения читатели смогут, выполнив упражнения 7 и 8, сформулированные в конце главы.

```
// ********************************************************
// Заголовочный файл TableException.h
// ********************************************************
#include <exception>
#include <string>
using namespace std;

class TableException : public exception
{
public:
   TableException(const string & message ="")
      : exception(message.c_str())
{ }
}; // Конец класса TableException

// ********************************************************
// Заголовочный файл TableA.h абстрактной таблицы ADT.
// Реализация в виде упорядоченного массива.
// Предположение: в каждый момент времени таблица содержит
//                не более одного элемента с заданным значением
//                поискового ключа.
// ********************************************************
#include "KeyedItem.h" // Определение классов KeyedItem
                       // и KeyType
#include "TableException.h"
const int MAX_TABLE = максимальный размер таблицы;

typedef KeyedItem TableItemType;
typedef void (*FunctionType)(TableItemType& anItem);

class Table
{
public:
   Table(); // Конструктор по умолчанию
   // Конструктор копирования и деструктор генерируются компилятором

// Операции над таблицей:
// Предусловие для всех операций:
// В таблице нет двух элементов с одинаковым поисковым ключом.
// Элементы таблицы упорядочены по ключу.

virtual bool tableIsEmpty() const;
// Определяет, пуста ли таблица.
// Постусловие: если таблица пуста, возвращает значение true;
// в противном случае возвращает значение false.

virtual int tableLength() const;
// Определяет размер таблицы.
// Постусловие: возвращает количество элементов,
// содержащихся в таблице.
```

```
virtual void tableInsert(const TableItemType& newItem)
   throw(TableException);
// Вставляет в таблицу новый элемент,
// оставляя ее упорядоченной.
// Предусловие: в таблицу вставляется элемент newItem,
// поисковый ключ которого отличается от поисковых ключей
// всех остальных элементов, содержащихся в таблице.
// Постусловие: если вставка произведена успешно,
// элемент newItem находится в соответствующем месте таблицы.
// Исключительная ситуация: если вставить элемент невозможно,
// генерируется исключительная ситуация TableException.

virtual void tableDelete(KeyType searchKey)
   throw(TableException);
// Удаляет из таблицы элемент с заданным ключом.
// Предусловие: поисковый ключ удаляемого элемента задается
// аргументом searchKey.
// Постусловие: если в таблице есть элемент с поисковым
// ключом, значение которого равно аргументу searchKey,
// он из нее удаляется
// Исключительная ситуация: если удаляемого элемента в таблице
// нет, генерируется исключительная ситуация TableException.

virtual void tableRetrieve(KeyType searchKey,
   TableItemType& tableItem) const
   throw(TableException);
// Извлекает из таблицы элемент с заданным ключом.
// Предусловие: ключ искомого элемента задается
// аргументом searchKey.
// Постусловие: если в таблице есть элемент с поисковым
// ключом, значение которого равно аргументу searchKey,
// он присваивается переменной tableItem.
// Исключительная ситуация: если искомого элемента в таблице
// нет, генерируется исключительная ситуация TableException.

virtual void traverseTable(FunctionType visit);
// Выполняет обход таблицы в порядке следования ключей поиска,
// вызывая функцию visit() по одному разу для каждого элемента.
// Предусловие: функция, соответствующая аргументу visit(),
// определяется вне реализации абстрактной таблицы.
// Постусловие: функция visit() выполнена для каждого элемента
// по одному разу.
// Замечание: функция visit() может изменять таблицу.

protected:
   void setSize(int newSize);
   // Присваивает закрытому члену size значение newSize.

   void setItem(const TableItemType& newItem, int index);
   // Присваивает элементу items[index] значение newItem.

   int position(KeyType searchKey) const;
   // Находит позицию заданного элемента или точку вставки.
   // Предусловие: значение поискового ключа searchKey должно
   // принадлежать одному из элементов таблицы.
```

```
    // Предусловие: возвращает индекс (от 0 до size - 1)
    // элемента таблицы, поисковый ключ которого равен аргументу
    // searchKey. Если такого элемента нет, возвращает номер
    // позиции, которую он мог бы занимать после вставки.
    // Таблица остается без изменения.

private:
    TableItemType items[MAX_TABLE]; // Элементы таблицы
    int size; // Размер таблицы

    int keyIndex(int first, int last, KeyType searchKey) const;
    // Находит отрезок закрытого массива по заданному ключу,
    // используя бинарный поиск.
    // Предусловие: 0 <= first, last < MAX_TABLE,
    // где MAX_TABLE = максимальный размер массива,
    // а массив items[first..last] упорядочен по возрастанию.
    // Постусловие: если поисковый ключ searchKey принадлежит
    // одному из элементов массива, возвращает его индекс;
    // в противном случае возвращает номер (от first
    // до last) позиции, которую он мог бы занимать после
    // вставки.  Таблица остается без изменения.
}; // Конец класса Table
// Конец заголовочного файла.
```

Поскольку большая часть описанного выше класса вполне очевидна, осталось только привести фрагмент файла реализации.

```
// ********************************************************
// Фрагмент файла реализации TableA.cpp.
// Реализация в виде упорядоченного массива.
// ********************************************************
#include "TableA.h" // Заголовочный файл

void Table::tableInsert(const TableItemType& newItem)
// Замечание: если таблица заполнена полностью,
// т.е. содержит MAX_TABLE элементов, вставка невозможна.
// Вызываемая функция: position.
{
    if (size == MAX_TABLE)
        throw TableException("TableException: таблица переполнена");

    // Есть место для вставки;
    // Найти позицию элемента newItem
    int spot = position(newItem.getKey());

    // Сдвинуть элементы, чтобы освободить место
    for (int index = size-1; index >= spot; --index)
    items[index+1] = items[index];

    // Выполнить вставку
    items[spot] = newItem;
    ++size;
} // Конец функции tableInsert

void Table::tableDelete(KeyType searchKey)
// Вызываемая функция: position.
{
```

```
    // Найти позицию, занимаемую элементом
    // с поисковым ключом searchKey
    int spot = position(searchKey);

    // Существует ли в таблице элемент с ключом searchKey?
    if ((spot > size) || (items[spot].getKey() != searchKey))

    // В таблице нет элемента с ключом searchKey
    throw TableException(
       "TableException: удаляемого элемента в таблице нет");
    else
    {
       // В таблице есть элемент с ключом searchKey
       --size; // Удалить этот элемент

      // Сдвинуть элементы, чтобы заполнить пробел
      for (int index = spot; index < size; ++index)
      items[index] = items[index+1];
    } // Конец оператора if
} // Конец функции tableDelete

void Table::tableRetrieve(KeyType searchKey,
                          TableItemType& tableItem) const
// Вызываемая функция: position.
{
    // Найти позицию, занимаемую элементом
    // с поисковым ключом searchKey
    int spot = position(searchKey);

    // Существует ли в таблице элемент с ключом searchKey?
    if ((spot > size) || (items[spot].getKey() != searchKey))

    // В таблице нет элемента с ключом searchKey
    throw TableException(
       "TableException: извлекаемого элемента в массиве нет");
    else
       tableItem = items[spot]; // Элемент существует;
                                // извлечь его
} // Конец функции tableRetrieve

void Table::traverseTable(FunctionType visit)
{
    for (int index = 0; index < size; ++index)
    visit(items[index]);
} // Конец функции traverseTable
// Конец файла реализации.
```

Реализация абстрактной таблицы в виде бинарного дерева поиска

Несмотря на то что в некоторых приложениях линейные реализации таблиц вполне приемлемы, в общем случае они неэффективны.

Ниже приведена программа реализации абстрактной таблицы в виде нелинейного бинарного дерева поиска. Класс *Table* содержит бинарное дерево поиска в качестве одного из своих данных-членов. Таким образом, класс *Table* исполь-

зует класс *BinarySerachTree*, описанный в предыдущей главе. Для экономии места пред- и постусловия функций-членов пропущены, поскольку они полностью совпадают с предыдущей реализацией.

```
// ********************************************************
// Заголовочный файл TableB.h абстрактной таблицы.
// Реализация в виде бинарного дерева поиска.
// Предположение: в каждый момент времени таблица содержит
// по крайней мере один элемент с заданным ключом поиска.
// ********************************************************
#include "BST.h" // Операции над бинарным деревом поиска
#include "TableException.h"

typedef TreeItemType TableItemType;

class Table
{
public:
   Table(); // Конструктор по умолчанию
   // Конструктор копирования и деструктор
   // генерируются компилятором

// Операции над таблицей:
   virtual bool tableIsEmpty() const;
   virtual int tableLength() const;
   virtual void tableInsert(const TableItemType& newItem)
      throw(TableException);
   virtual void tableDelete(KeyType searchKey)
      throw(TableException);
   virtual void tableRetrieve(KeyType searchKey,
      TableItemType& tableItem) const
      throw(TableException);
   virtual void traverseTable(FunctionType visit);

protected:
   void setSize(int newSize);

private:
   BinarySearchTree bst; // Бинарное дерево поиска,
                         // содержащее элементы таблицы
   int size; // number of items in the table
}; // Конец класса Table
// Конец заголовочного файла.
```

Как и в предыдущем случае, приведем лишь фрагмент файла реализации.

```
// ********************************************************
// Отрывки из файла реализации TableB.cpp.
// Реализация в виде бинарного дерева поиска.
// ********************************************************
#include "TableB.h" // Заголовочный файл.

void Table::tableInsert(const TableItemType& newItem)
{
   try
   {
      bst.searchTreeInsert(newItem);
```

```
      ++size;
   } // Конец блока try
   catch (TreeException e)
   {
      throw TableException(
      "TableException: невозможно вставить элемент");
   } // Конец блока catch
} // Конец функции tableInsert

void Table::tableDelete(KeyType searchKey)
{
   try
   {
      bst.searchTreeDelete(searchKey);
   } // Конец блока try
   catch (TreeException e)
   {
      throw TableException(
      "TableException: удаляемый элемент не найден");
   } // Конец блока catch
} // Конец функции tableDelete

void Table::tableRetrieve(KeyType searchKey,
                          TableItemType& tableItem) const
   {
     try
     {
        bst.searchTreeRetrieve(searchKey, tableItem);
     } // Конец блока try
     catch (TreeException e)
     {
        throw TableException(
        "TableException: искомый элемент не найден");
     } // Конец блока catch
} // Конец функции tableRetrieve

void Table::traverseTable(FunctionType visit)
{
   bst.inorderTraverse(visit);
} // Конец функции traverseTable
// Конец файла реализации.
```

Ниже приведены операторы, демонстрирующие применение этих файлов в программе, работающей с таблицей.

```
#include <iostream>
#include "TableB.h"
using namespace std;

void displayKey(TableItemType& anItem)
{
   cout << anItem.getKey() << endl;
} // Конец функции displayKey

int main()
{
   Table chart;
```

```
TableItemType anItem;

cin >> anItem;
chart.tableInsert(anItem);
. . .

chart.traverseTable(displayKey); // Обход в заданном порядке
```

Абстрактная очередь с приоритетами: вариант абстрактной таблицы

Абстрактная таблица позволяет организовывать данные по ключу, облегчая поиск конкретного элемента по заданному значению. Следовательно, абстрактную таблицу следует применять, когда поиск в базе данных производится не по позиции, а по значению записи. Рассмотрим теперь абстрактный тип данных, тесно связанный с таблицей, который может оказаться еще более удобным.

Представьте себе человека, пришедшего в приемный покой больницы. Когда в больницу поступает новый пациент, регистратор вносит в базу данных запись об этом человеке. В дальнейшем эта запись будет извлечена медсестрами и врачами. Кроме того, регистратор должен следить за пациентами, поступившими в приемный покой, и решать, кому из них нужна помощь.

Данные можно организовывать по приоритету

Для создания базы данных общего назначения можно применить таблицу. А какой абстрактный тип данных следует выбрать для регистрационного журнала в приемном покое? Таблица позволяет хранить записи о больных либо в алфавитном порядке, либо в порядке возрастания их идентификационных номеров. Для помощи больным по мере их поступления можно было бы применить очередь. Однако в этом случае больной Я., поступивший с приступом острого аппендицита, был бы вынужден ждать, пока у больного А. вынут занозу. Совершенно ясно, что регистратор обязан присвоить каждому пациенту определенный **приоритет** (priority). Тогда, освободившись, доктор может подойти к пациенту, имеющему наивысший приоритет. Абстрактный тип данных, необходимый для описания такой ситуации, должен указывать пациента, больше всех нуждающегося в помощи.

Рассмотрим еще один пример. Представьте себе список дел, которые вам предстоит выполнить на этой неделе. Допустим, это список состоит из таких пунктов.

Люди часто записывают задания в порядке их важности

- Послать поздравление с днем рождения тете Соне.
- Начать работать над диссертацией по всемирной истории.
- Закончить чтение главы 11.
- Написать распорядок дня на воскресенье.

Сверяясь с этим списком, вы, конечно, уделите больше внимания наиболее важному заданию.

Приоритет указывает, например, порядок обслуживания пациентов в приемном покое или очередность выполнения заданий. Какую величину можно использовать для вычисления приоритета? Для этого есть масса возможностей, начиная с простой нумерации. Для определенности будем считать, что наибольшее значение этой величины соответствует наивысшему приоритету. В этом случае

величина приоритета становится частью записи, представляющей элемент какой-либо абстрактной структуры данных. Записывать в эту структуру можно любой элемент, а вот извлекать — только элемент, имеющий наивысший приоритет.

На рис. 11.8 показана UML-диаграмма класса очередей с приоритетами. Детали операций над этим абстрактным типом данных описаны в приведенном ниже псевдокоде.

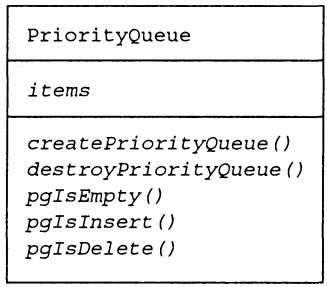


Рис. 11.8. UML-диаграмма класса PriorityQueue

ОСНОВНЫЕ ПОНЯТИЯ

Операции над абстрактной очередью с приоритетами

1. Создать пустую очередь с приоритетами.
2. Уничтожить пустую очередь с приоритетами.
3. Определить, пуста ли очередь с приоритетами.
4. Вставить новый элемент в очередь с приоритетами.
5. Извлечь, а затем удалить из очереди элемент, имеющий наивысший приоритет.

Очередь с приоритетами содержит элементы, упорядоченные по приоритету

Такой абстрактный тип данных называется **очередью с приоритетами** (priority queue). Говоря более формально, очередь с приоритетами — это абстрактный тип данных, предусматривающий следующие операции.

Очередь с приоритетами отличается от таблицы операцией pqDelete

Эти операции напоминают операции над абстрактной таблицей. Основное различие заключается в операции `pqDelete`. В то время как операции `tableInsert` и `tableDelete` позволяли вставлять и удалять элемент таблицы, соответствующий заданному значению поискового ключа, операция `pqDelete` позволяет извлекать и удалять элемент, имеющий наивысший приоритет. Обратите внимание, что операция `pqDelete`, в отличие от операций `tableRetrieve` и `tableDelete`, не запрашивает никаких значений. Поскольку величина наивысшего приоритета заранее не известна, операции `tableRetrieve` и `tableDelete` применить нелегко. Однако операцию `pqDelete` невозможно применить для извлечения и удаления произвольного элемента очереди с приоритетами по заданному значению.

Возможные реализации

Абстрактная очередь с приоритетами и абстрактная таблица и похожи, и не похожи друг на друга. Этот факт отражается на их реализациях. Для начала реализуем очередь с приоритетами с помощью таблицы. Если количество элементов в очереди с приоритетами мало, можно применить упорядоченную линейную структуру.

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операций над абстрактной очередью с приоритетами

```
// PQItemType представляет собой тип элементов, хранящихся в очереди
с приоритетами

+createPQueue()
// Создает пустую очередь с приоритетами.

+destroyPQueue()
// Уничтожает очередь с приоритетами.

+pqIsEmpty():boolean
// Определяет, пуста ли очередь с приоритетами.

+pqInsert(in newItem:PQItemType) throw PQException
// Вставляет элемент newItem в очередь с приоритетами. Если
// очередь переполнена, генерируется исключительная ситуация
// PQException.

+pqDelete(out priorityItem:PQItemType) throw PQException
// Извлекает в переменную priorityItem, а затем удаляет из
// очереди элемент, имеющий наивысший приоритет. Если очередь
// пуста, генерируется исключительная ситуация PQException.
```

Динамический массив позволяет хранить элементы в порядке возрастания их приоритетов, так что наивысший приоритет всегда имеет последний элемент массива, как показано на рис. 11.9, *а*. Следовательно, операция *pqDelete* будет просто возвращать элемент *items[size-1]* и уменьшать значение переменной *size* на единицу. Однако операция *pqInsert*, определив подходящее место для вставки с помощью алгоритма бинарного поиска, должна сдвигать элементы массива, чтобы освободить место для нового элемента.

Линейный связанный список, изображенный на рис. 11.9, *б*, содержит элементы, упорядоченные по убыванию, так что элемент, имеющий наивысший приоритет, находится в начале. Следовательно, операция *pqDelete* просто вернет элемент, на который ссылается указатель *pqHead*, а затем установит его на следующий элемент. Однако операция *pqInsert* должна обойти список с самого начала, прежде чем обнаружит позицию, подходящую для вставки. Таким образом, линейные реализации очереди с приоритетами "страдают" от тех же недостатков, что и линейные реализации абстрактной таблицы.

В качестве альтернативной реализации очереди с приоритетами рассмотрим бинарное дерево поиска, показанное на рис. 11.9, *в*. Хотя операция *pqInsert* совпадает с операцией *tableInsert*, операция *pqDelete* не имеет прямого аналога среди операций над таблицами. Она должна определить элемент, имеющий наивысший приоритет, не зная заранее значение этого приоритета. Однако это не сложно, поскольку этот элемент всегда является крайним правым элементом дерева. (Почему?) Следовательно, нам нужно лишь следовать за указателем *rightChildPtr*, пока не обнаружится узел, у которого значение указателя *rightChildPtr* равно константе *NULL*. (Эту задачу может выполнить функция, аналогичная функции *processLeftmost*.) Удалить этот узел из дерева очень легко, поскольку он имеет только один дочерний узел.

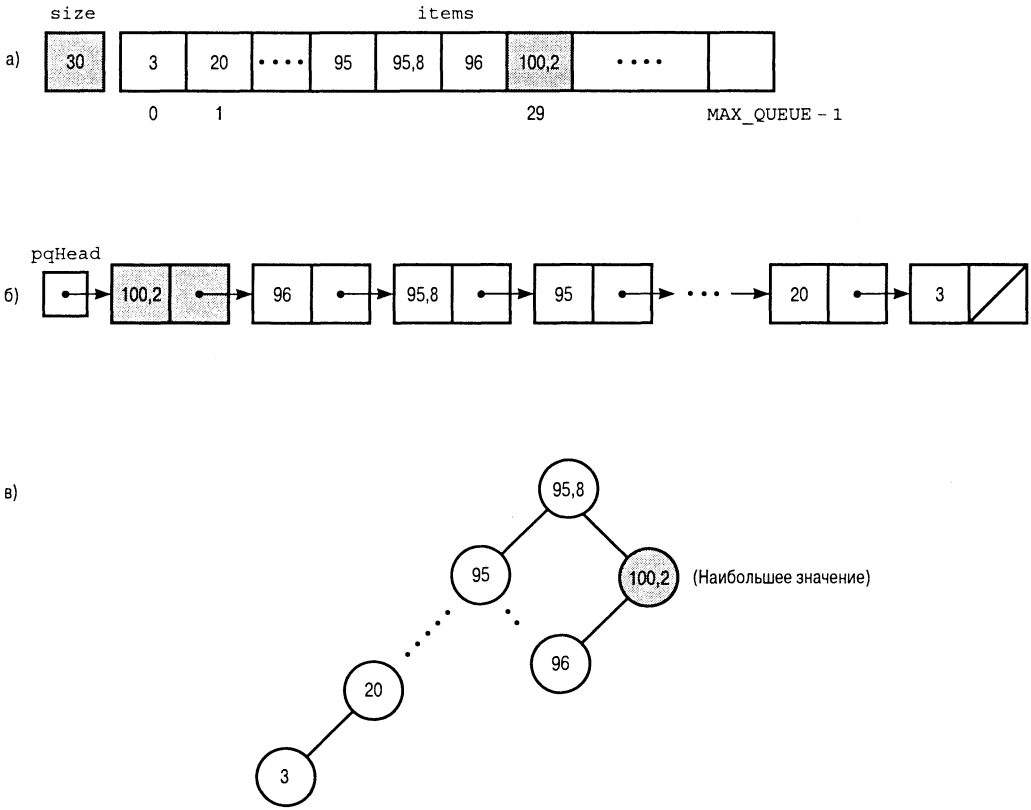


Рис. 11.9. Некоторые реализации абстрактной очереди с приоритетами: а) в виде массива; б) в виде бинарного дерева поиска

Итак, бинарное дерево поиска одинаково хорошо реализует как таблицу, так и очередь с приоритетами. Однако эти абстрактные типы данных используются в совершенно разных задачах. Некоторые приложения, использующие таблицы, в основном выполняют операции извлечения и обхода и поэтому не изменяют сбалансированности бинарного дерева поиска. В то же время, очередь с приоритетами не предусматривает операций извлечения и обхода, поэтому все операции вставки и удаления элементов изменяют форму соответствующего бинарного дерева поиска. Методы балансировки бинарного дерева поиска обсуждаются в главе 12. Однако, если размер очереди с приоритетами известен заранее, имеет смысл выбрать реализацию **кучи** (heap) в виде массива, описанную в следующем разделе. Эта реализация очень часто используется для представления очереди с приоритетами, но она не всегда подходит для работы с таблицами.

Кучи

Куча — это абстрактный тип данных, очень похожий на бинарное дерево поиска, но отличающийся от него двумя важными свойствами.

> Куча отличается от бинарного дерева поиска двумя особенностями

Во-первых, бинарное дерево поиска считается упорядоченным, а порядок элементов в куче имеет совершенно другой смысл. Однако этого достаточно, чтобы эффективно реализовать операции, предусмотренные для очереди с приоритета-

ми. Во-вторых, бинарное дерево поиска может иметь разную форму, а кучи всегда являются совершенными бинарными деревьями.

Куча — это полное бинарное дерево, обладающее следующими свойствами.

Куча — это специфический вид совершенного бинарного дерева

- Она пуста

или

- ключ, содержащийся в ее корне, больше или равен ключу каждого его дочернего узла и
- поддеревья корня являются кучами.

В нашем определении корень содержит элемент, имеющий наибольший ключ поиска. Такую кучу также называют **максимальной** (maxheap). В **минимальной куче** (minheap) корень содержит элемент, имеющий наименьший ключ. Более подробно минимальная куча рассматривается в упражнении 16.

UML-диаграмма класса *Heap* показана на рис. 11.10.

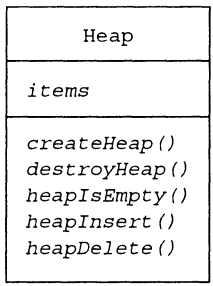


Рис. 11.10. UML-диаграмма класса Heap

ОСНОВНЫЕ ПОНЯТИЯ

Псевдокод операций над абстрактной кучей

```
// HeapItemType представляет собой тип элементов, хранящихся в куче

+createHeap()
// Создает пустую кучу

+destroyHeap()
// Уничтожает кучу.

+heapIsEmpty():boolean {query}
// Определяет, пуста ли куча.

+heapInsert(in newItem:HeapItemType) throw HeapException
// Вставляет в кучу элемент newItem. Если куча переполнена,
// генерируется исключительная ситуация HeapException

+heapDelete(out rootItem:HeapItemType) throw Heap Exception
// Извлекает, а затем удаляет элемент из корня кучи.
// Этот элемент имеет наибольший ключ поиска. Если куча пуста,
// генерируется исключительная ситуация HeapException
```

Куча является совершенным бинарным деревом. Поэтому, если максимальный размер кучи известен заранее, для ее реализации можно применять массив, как показано в главе 10. Пример такой реализации показан на рис. 11.11. Ключ произвольного узла кучи больше или равен ключам всех его дочерних узлов. Кроме того, между ключами дочерних ключей нет никаких соотношений, т.е. заранее неизвестно, какой дочерний узел содержит больший ключ.

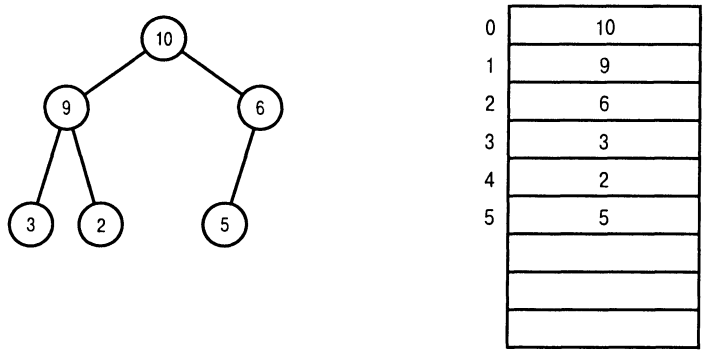


Рис. 11.11. Реализация кучи в виде массива

Реализация кучи в виде массива. Представим кучу с помощью следующих данных-членов:

Реализация кучи в виде массива содержит массив и счетчик

- `items` — массив элементов кучи;
- `size` — количество элементов, содержащихся в куче.

Массив `items` соответствует реализации дерева. (Для простоты будем считать, что куча содержит целые числа.)

Операция heapDelete. Сначала рассмотрим операцию `heapDelete`. В каком узле находится элемент, содержащий наибольший поисковый ключ? Поскольку поисковый ключ каждого ключа больше или равен поисковым ключам своих дочерних узлов, наибольший поисковый ключ должен находиться в корне дерева. Следовательно, первый шаг операции `heapDelete` должен быть таким.

Первый шаг операции heapDelete

```
// Возвращает элемент, содержащийся
// в корне
rootItem = items[0]
```

Это было легко, но теперь нам нужно удалить корень. Сделав это, мы получим две разъединенные кучи, как показано на рис. 11.12, *а*. Следовательно, оставшиеся узлы необходимо объединить в новую кучу. Для этого нужно взять последний узел дерева и поместить его в корень, как показано в следующем фрагменте псевдокода.

На втором шаге операции heapDelete создается полукуча

```
// Копируем в корень последний
// элемент дерева
items[0] = items[size-1]

// Удаляем последний узел
--size;
```

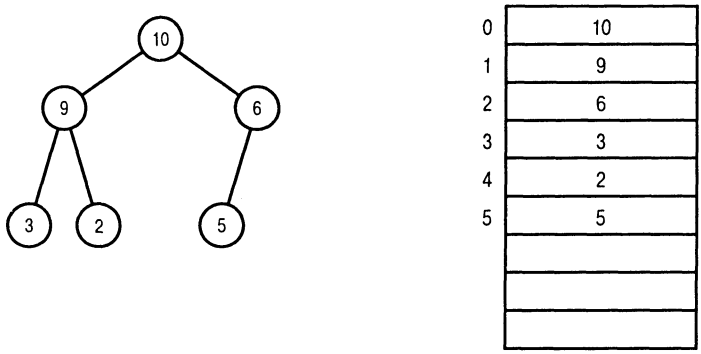


Рис. 11.12. Разновидности кучи: а) разъединенные кучи; б) полукуча

Как следует из рис. 11.12, *б*, в результате такой операции *не* обязательно возникнет новая куча. Однако полученное дерево будет совершенным, а его левое и правое поддеревья будут кучами. Единственная проблема — элемент, содержащийся в корне, как правило, находится не на своем месте. Такая структура называется **полукучей** (semiheap). Таким образом, нам нужен способ, позволяющий преобразовать полукучу в настоящую кучу. Одна из стратегий решения этой задачи заключается в том, что элемент, содержащийся в корне, *стекает* (trickle down) по дереву, пока не достигнет узла, в котором он должен был бы находиться. Иными словами, элемент останется в первом же узле, поисковый ключ которого больше или равен поисковым ключам своих дочерних узлов. Для того чтобы осуществить этот план, сначала нужно сравнить поисковый ключ корня полукучи с поисковыми ключами его дочерних узлов. Если поисковый ключ корня меньше, чем поисковые ключи его дочерних узлов, элемент, содержащийся в корне нужно поменять местами с элементом, находящимся в наибольшем дочернем узле. (Наибольшим дочерним узлом называется дочерний узел с наибольшим ключом.)

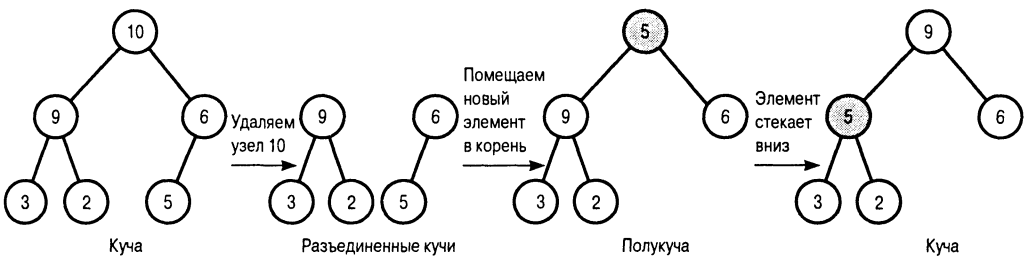


Рис. 11.13. Удаление элемента из кучи

Операция `heapDelete` продемонстрирована на рис. 11.13. Хотя в данном примере значение 5 стекает по дереву в правильный узел после первой же перестановки, обычно для этого требуется намного больше перестановок. Фактически, как только элементы, содержащиеся в корне и наибольшем дочернем узле `C`, поменялись местами, узел `C` становится корнем полукучи. (Обратите внимание, что сам узел `C` никуда не перемещается; изменяется только его значение.) Эта стратегия воплощается следующим алгоритмом.

На последнем шаге операции heapDelete полукуча превращается в кучу

```
heapRebuild(inout items:ArrayType,
            in root:integer,
            in size:integer)
// Преобразовывает полукучу в кучу.
// Элемент, содержащийся в корне, рекурсивно стекает по дереву
// в соответствующую позицию, меняясь местами с элементами,
// находящимися в наибольших дочерних узлах, если ключ
// дочернего узла больше ключа самого элемента.
// Если элемент принадлежит листу, алгоритм завершается.

   if (корень не является листом)
   {
      // Корень должен иметь левый дочерний узел
      child = 2 * root + 1 // Индекс левого дочернего узла

     if (корень имеет правый дочерний узел)
     {
       rightChild = child + 1 // Индекс правого дочернего узла
       if (items[rightChild].getKey() > items[child].getKey())
           child = rightChild // larger child index
     } // Конец оператора if

     // Если ключ элемента, содержащегося в корне,
     // меньше ключа его наибольшего дочернего узла,
     // выполняется перестановка этих элементов

     if (items[root].getKey() < items[child].getKey())
     {
        Меняем местами items[root] и items[child]

       // Преобразовываем полукучу с корнем child в кучу
       heapRebuild(items, child, size)
     } // end if
   } // end if

// Если корень является листом, алгоритм завершается
```

Рекурсивные вызовы операции *heapRebuild* проиллюстрированы на рис. 11.14.

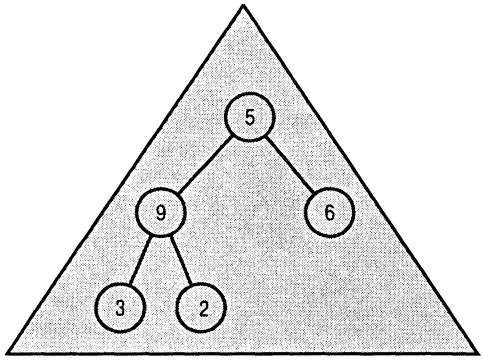
Операция *heapDelete* следующим образом использует операцию *heapRebuild*.

```
// Возвращаем элемент, содержащийся в корне
rootItem = items[0]

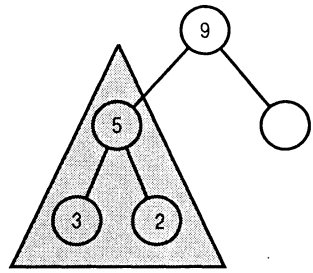
// Копируем элемент из последнего узла в корнеь
items[0] = items[size-1]

// Удаляем последний корень
--size

// Преобразовываем полукучу в кучу
heapRebuild(items, 0, size)
```



Первая полукуча, передаваемая операции **heapRebuild**



Вторая полукуча, передаваемая операции **heapRebuild**

Рис. 11.14. Рекурсивные вызовы операции heapRebuild

Эффективность операции heapDelete

Оценим эффективность операции *heapDelete*. Поскольку дерево хранится в массиве, для удаления узла придется переставлять элементы массива, а не просто изменять значения нескольких указателей. Однако это не значит, что алгоритм неэффективен. Сначала попробуем определить, сколько элементов массива понадобится переставить в худшем случае? После того как операция *heapDelete* скопирует элемент из последнего узла в корень, операция *heapRebuild* будет перемещать этот элемент вниз по дереву, пока не найдет для него подходящее место. Следовательно, количество переставляемых элементов массива не может превосходить высоту дерева. Высота совершенного бинарного дерева, состоящего из n узлов, всегда равна $[\log_2(n+1)]$. Каждая перестановка сводится к трем операторам присваивания. Следовательно, операция *heapDelete* потребует выполнения $3 * [\log_2(n+1)] + 1$ операций. Таким образом, эффективность операции *heapDelete* имеет порядок O(log n).

Эффективность операции heapDelete имеет порядок O(log n)

Операция heapInsert. Стратегия, положенная в основу операции *heapInsert*, противоположна стратегии операции *heapDelete*. Новый элемент вставляется внизу дерева и просачивается наверх на нужное место, как показано на рис. 11.15. Реализовать эту стратегию довольно легко, поскольку родитель узла, хранящегося в узле *items[i]* (если он не является корнем), всегда находится в ячейке *items[(i-1)/2)*. Псевдокод операции *heapInsert* имеет следующий вид.

Стратегия вставки

```
// Вставляем элемент newItem
// в основание дерева
   items[size] = newItem

   // Продвигаем новый элемент вверх, пока не обнаружим
   // подходящий узел
   place = size
   parent = (place-1)/2
   while ( (parent >= 0) and (items[place] > items[parent]) )
   {
      Меняем местами элементы items[place] и items[parent]
      place = parent
      parent = (place-1)/2
   } // Конец оператора while

   Увеличить переменную size на единицу
```

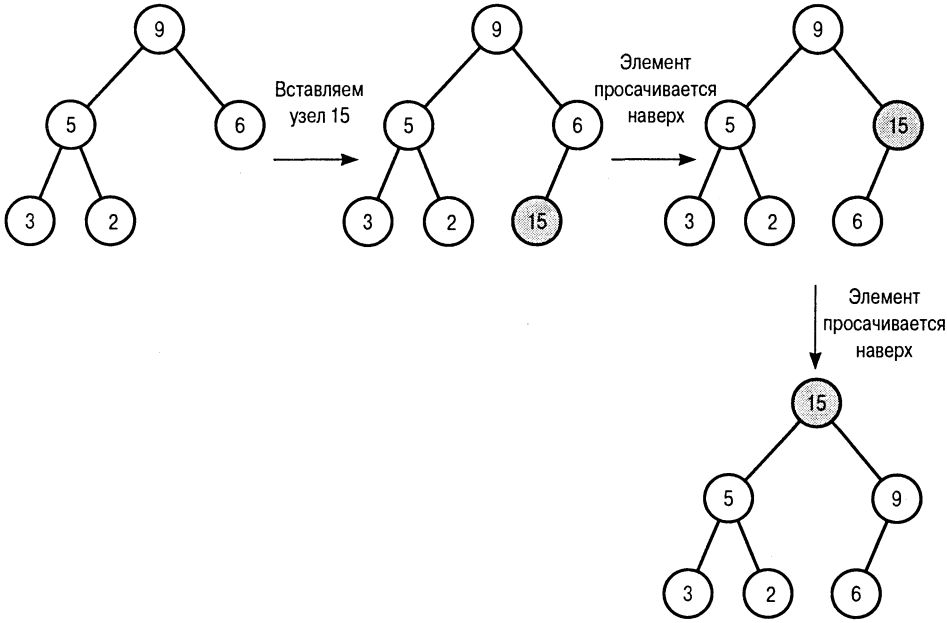


Рис. 11.15. Вставка элемента в кучу

Рассмотрим файл, содержащий реализацию абстрактной кучи в виде массива.

```
// ********************************************************
// Заголовочный файл Heap.h абстрактной кучи.
// ********************************************************
const int MAX_HEAP = максимальный размер кучи;
#include "KeyedItem.h" // Определение класса KeyedItem

typedef KeyedItem HeapItemType;

class Heap
{
public:
   Heap(); // Конструктор по умолчанию
   // Конструктор копирования и деструктор генерируются
   // компилятором

// Операции над кучей:
   virtual bool heapIsEmpty() const;
   // Определяет, пуста ли куча.
   // Предусловие: нет.
   // Постусловие: если куча пуста, возвращает значение true;
   // в противном случае возвращает значение false.

   virtual void heapInsert(const HeapItemType& newItem)
      throw(HeapException);
   // Вставляет в кучу новый элемент.
   // Предусловие: вставляемый элемент задается аргументом newItem.
   // Постусловие: если куча не полна, элемент newItem
   // занимает соответствующую позицию; в противном случае
   // генерируется исключительная ситуация HeapException.
```

```
   virtual void heapDelete(HeapItemType& rootItem)
      throw(HeapException);
   // Извлекает и удаляет элемент из корня кучи.
   // Данный элемент имеет наибольшее значение ключа.
   // Предусловие: нет.
   // Постусловие: если куча не пуста, из кучи извлекается
   // и удаляется элемент rootItem. Если куча пуста,
   // удаление невозможно, и генерируется исключительная
   // ситуация HeapException.

protected:
   void heapRebuild(int root);
   // Преобразовывает полукучу в кучу.

private:
   HeapItemType items[MAX_HEAP]; // Массив элементов кучи
   int size; // Количество элементов кучи
}; // Конец класса
// Конец заголовочного файла.

// ********************************************************
// Файл реализации Heap.cpp абстрактной кучи.
// ********************************************************
#include "Heap.h" // Заголовочный файл класса Heap
Heap::Heap() : size(0)
{
} // конец конструктора по умолчанию

bool Heap::heapIsEmpty() const
{
   return bool(size == 0);
} // Конец функции heapIsEmpty

void Heap::heapInsert(const HeapItemType& newItem)
// Метод: вставляет новый элемент за последним узлом кучи
// и перемещает его вверх  по дереву на соответствующую
// позицию. Куча считается полной, если она содержит
// MAX_HEAP элементов.
{
   if (size > MAX_HEAP)
      throw HeapException("HeapException: куча полна");
   // Размещаем новый элемент в конце кучи
   items[size] = newItem;

   // Перемещаем новый элемент на подходящую позицию
   int place = size;
   int parent = (place - 1)/2;
   while ( (parent >= 0) &&
         (items[place].getKey() > items[parent].getKey()) )
   {
       // Меняем местами элементы items[place] и items[parent]
       HeapItemType temp = items[parent];
       items[parent] = items[place];
       items[place] = temp;
       place = parent;
       parent = (place - 1)/2;
   } // Конец оператора while
```

```
   ++size;
} // Конец функции heapInsert

void Heap::heapDelete(HeapItemType& rootItem)
// Метод: меняет местами последний элемент кучи с корнем
// и перемещает его вниз по дереву, пока не будет обнаружена
// подходящая позиция.
{
   if (heapIsEmpty())
      throw HeapException("HeapException: куча пуста");
   else
   {
      rootItem = items[0];
      items[0] = items[--size];
      heapRebuild(0);
   } // Конец оператора if
} // Конец оператора heapDelete

void Heap::heapRebuild(int root)
{
   // Если корень не является листом и ключ корня меньше ключей
   // его дочерних узлов
   int child = 2 * root + 1; // Индекс левого дочернего узла
                             // корня, если он существует

   if ( child < size )
   {
      // Корень не является листом, поэтому имеет левый
      // дочерний узел

      int rightChild = child + 1; // Индекс правого дочернего
                                  // узла, если он существует

      // если корень имеет правый дочерний узел,
      // найти его наибольший дочерний узел
      if ( (rightChild < size) &&
         (items[rightChild].getKey() > items[child].getKey()) )
         child = rightChild; // Индекс наибольшего
                             // дочернего узла корня

      // Если ключ корня меньше ключа его наибольшего
      // узла, меняем эти элементы местами
      if ( items[root].getKey() < items[child].getKey() )
      {
         HeapItemType temp = items[root];
         items[root] = items[child];
         items[child] = temp;

         // Преобразовываем новое поддерево в кучу
         heapRebuild(child);
      } // Конец оператора if
   } // Конец оператора if
   // Если корень является листом, алгоритм завершен
} // Конец функции heapRebuild
// Конец файла реализации.
```

Реализация абстрактной очереди с приоритетами в виде кучи

Операции над очередью с приоритетами и операции над кучей аналогичны

От реализации абстрактной кучи легко перейти к реализации абстрактной очереди с приоритетами, поскольку оба этих типа предусматривают одинаковые операции. Значение приоритета в очереди с приоритетами соответствует значению поискового ключа в куче. Таким образом, в реализации очереди с приоритетами можно использовать класс *Heap*. Иными словами, класс *PriorityQueue* содержит экземпляр класса *Heap* в качестве переменной-члена. Рассмотрим реализацию абстрактной очереди с приоритетами подробнее.

```
// ********************************************************
// Заголовочный файл PQ.h абстрактной очереди с приоритетами.
// Реализация в виде кучи.
// ********************************************************
#include "Heap.h" // Абстрактные операции над кучей
typedef HeapItemType PQueueItemType;

class PriorityQueue
{
public:
   // Конструктор по умолчанию, конструктор копирования
   // и деструктор генерируются компилятором

// Операции над очередью с приоритетами:
   virtual bool pqIsEmpty() const;
   virtual void pqInsert(const PQueueItemType& newItem)
      throw(PQueueException);
   virtual void pqDelete(PQueueItemType& PQueueItemType)
      throw(PQueueException);
//
// Реализация очереди с приоритетами содержит кучу
// в качестве переменной-члена
//
private:
   Heap h;
}; // Конец класса PriorityQueue
// Конец заголовочного файла.

// ********************************************************
// Файл реализации PQ.cpp абстрактной очереди с приоритетами.
// Реализация очереди с приоритетами в виде кучи.
// ********************************************************
#include "PQ.h" // Заголовочный файл очереди с приоритетами
bool PriorityQueue::pqIsEmpty() const
{
   return h.heapIsEmpty();
} // Конец функции pqIsEmpty

void PriorityQueue::pqInsert(const PQueueItemType& newItem)
{
   try
   {
      h.heapInsert(newItem);
   } // Конец блока try
```

```
    catch (HeapException e)
    {
       throw PQueueException(
       "PQueueException: очередь с приоритетами переполнена");
    } // Конец блока catch
} // Конец функции pqInsert

void PriorityQueue::pqDelete(PQueueItemType& priorityItem)
{
    try
    {
       h.heapDelete(priorityItem);
    } // Конец блока try
    catch (HeapException e)
    {
       throw PQueueException(
       "PQueueException: очередь с приоритетами пуста");
    } // Конец блока catch
} // Конец функции pqDelete
// Конец файла реализации.
```

Для реализации очереди с приоритетами в виде кучи необходимо знать заранее ее максимальный размер

Какая из реализаций очереди с приоритетами лучше: в виде кучи или в виде бинарного дерева поиска? Если максимальный размер очереди известен заранее, реализация в виде кучи более эффективна.

Поскольку куча является совершенным деревом, она всегда сбалансирована, что является ее несомненным преимуществом. Если бинарное дерево поиска сбалансировано, то эффективность обеих реализаций одинакова: для очереди, состоящей из n элементов, она имеет порядок O(log n). Однако высота бинарного дерева поиска в процессе вставок и удалений может увеличиваться, намного превышая величину $O(\log_2 n)$, в худшем случае снижая эффективность реализации до величины O(n). Реализация очереди с приоритетами в виде кучи лишена этого недостатка.

В следующей главе мы рассмотрим способы балансировки бинарных деревьев поиска, однако применяемые для этого операции намного сложнее операций над кучей. Не следует, однако, думать, что куча может заменить бинарное дерево поиска при реализации таблицы. Как указывалось ранее, куча для этого не подходит. Если вы сомневаетесь, попробуйте применить к куче операцию *tableRetrieve*, либо обойти кучу в порядке следования ключей поиска.

Конечные и разные значения приоритетов. Если в очереди используется конечное число разных приоритетов, например, целые числа от 1 до 20, многие элементы могут иметь одинаковый приоритет. Такие элементы можно размещать в порядке их появления.

Куча очередей

Для этой ситуации подходит куча, состоящая из очередей, в которой каждой очереди соответствует отдельное значение приоритета. Для того чтобы вставить элемент в очередь с приоритетами, можно добавить в кучу очередь, соответствующую заданному значению приоритета. Затем новый элемент вставляется в соответствующую очередь. Чтобы удалить элемент из очереди с приоритетами, нужно удалить элемент, стоящий в начале очереди, соответствующей наивысшему приоритету в куче. Если после удаления очередь становится пустой, ее следует удалить из кучи.

Другие виды очереди с приоритетами рассматриваются в задании 6 в этой главе.

Пирамидальная сортировка

Как следует из ее имени (heapsort), алгоритм пирамидальной сортировки использует кучу для упорядочения массива *anArray*. На первом шаге алгоритм преобразует массив в кучу. Для этого можно последовательно вставить элементы массива в кучу, используя функцию *heapInsert*.

Однако существует более эффективный способ превращения массива в кучу. Допустим, что в исходном положении массив *anArray* состоит из элементов, изображенных на рис. 11.16, *а*. Сначала представим массив *anArray* в виде бинарного дерева, присваивая его элементы узлам дерева, начиная с корня и перемещаясь по дереву вниз и слева направо. В результате получится дерево, изображенное на рис. 11.16, *б*. Затем мы трансформируем это дерево в кучу, вызывая несколько раз функцию *heapRebuild*. При каждом вызове функции *heapRebuild* полукуча — дерево, поддеревья которого являются кучами, но корень может находиться не на своем месте, — превращается в кучу. К каким полукучам следует применить функцию *heapRebuild*? Хотя дерево, изображенное на рис. 11.16, *б*, полукучей не является, оно содержит в себе полукучи — его листья, которые также являются полукучами. (Фактически каждый лист является кучей, но для простоты мы этот факт игнорируем.) Сначала мы вызываем функцию *heapRebuild* для каждого листа слева направо. Затем мы перемещаем дерево вверх, зная, что поддеревья узла *s* являются кучами, и, следовательно, функция *heapRebuild* превратит полукучу с корнем *s* в кучу.

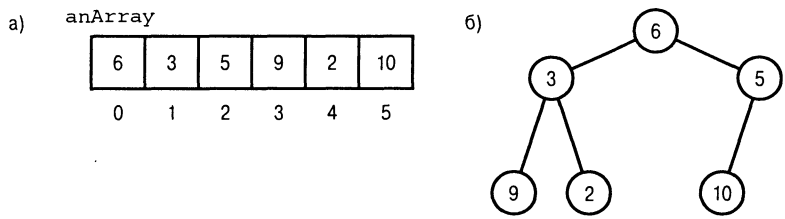


Рис. 11.16. Превращение массива в кучу: а) исходное состояние массива anArray; б) бинарное дерево, соответствующее массиву anArray

Приведенный ниже алгоритм превращает массив *anArray*, состоящий из *n* элементов, в кучу, выполняя первый шаг пирамидальной сортировки.

Создание кучи из массива элементов

```
for (index = n - 1 вниз до to 0)
   // Диагностическое утверждение:
   // дерево  с корнем  index является полукучей
   heapRebuild(anArray, index, n)
   // Диагностическое утверждение: дерево с корнем index
   // является кучей
```

Значение *n*-1 в операторе *for* можно заменить величиной *n*/2. Почему это возможно, вы узнаете, выполнив упражнение 20. Результаты трассировки этого алгоритма на примере массива, изображенного на рис. 11.16, *а*, показаны на рис. 11.17.

После преобразования массива в кучу алгоритм пирамидальной сортировки разделяет массив на две части — область кучи и упорядоченную область, как показано на рис. 11.18. Область кучи — это отрезок массива *anArray[0..last]*, а упорядоченная область — отрезок *anArray[last+1..n-1]*. В исходном положении область кучи распространяется на весь массив, а упорядоченная область пуста, т.е. переменная *last* равна *n*-1.

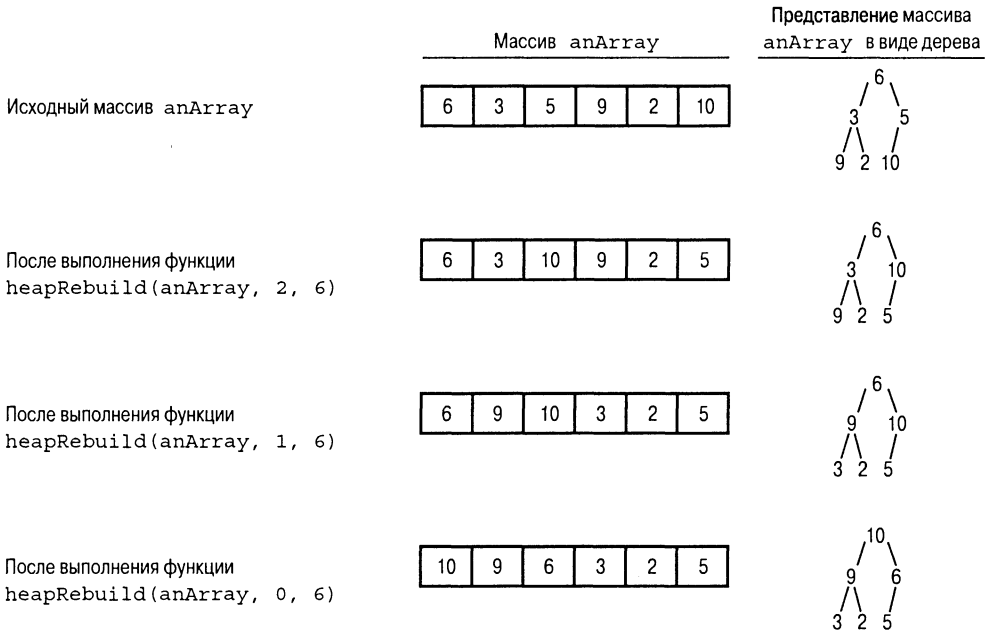


Рис. 11.17. Превращение массива anArray в кучу

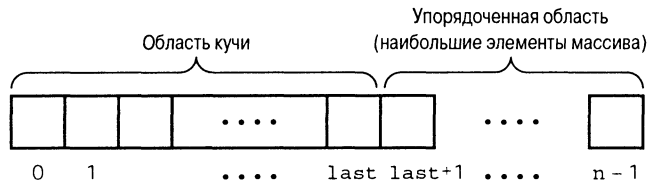


Рис. 11.18. Алгоритм пирамидальной сортировки разделяет массив на две части

На каждом шаге алгоритма элемент *I* из области кучи перемещается в упорядоченную область массива. Инвариант алгоритма пирамидальной сортировки формулируется так.

Инвариант алгоритма пирамидальной сортировки

- После *k*-го шага упорядоченная часть содержит *k* наибольших элементов массива *anArray*, следующих в таком порядке: *anArray[n-1]* — наибольший элемент массива, *anArray[n-2]* — второй наибольший элемент и т.д.
- Элементы другой части массива образуют кучу.

Для того чтобы инвариант выполнялся, элемент *I* должен иметь наибольшее значение в области кучи и, следовательно, быть ее корнем. Для перемещения элемента нужно поменять местами корень кучи и ее последний элемент, т.е. элементы *anArray[0]* и *anArray[last]*, а затем уменьшить значение переменной *last* на единицу. В результате элемент, который перешел из корня в ячейку *anArray[last]*, становится наименьшим элементом упорядоченной части, зани-

мая ее первую ячейку. После перемещения область кучи нужно преобразовать в кучу, поскольку ее корень находится не на своем месте. Это преобразование выполняется функцией *healRebuild*, позволяющей переместить элемент в корень так, чтобы область кучи стала настоящей кучей.

Итак, алгоритм пирамидальной сортировки формулируется следующим образом.

```
heapSort(inout anArray: ArrayType, in n:integer)
// Упорядочиваем массив anArray[0..n-1].

   // Создаем исходную кучу
   for (index = n - 1 down to 0)
   {
      // Инвариант: дерево с корнем index является полукучей
      heapRebuild(anArray, index, n)
      // Диагностическое утверждение: дерево с корнем index
      // является кучей
   } // Конец оператора for
   // Диагностическое утверждение: элемент anArray[0]
   // является наибольшим элементом кучи anArray[0..n-1]

   // Инициализируем области массива
   last = n - 1
   // Инвариант: отрезок anArray[0..last] является кучей,
      отрезок anArray[last+1..n-1] упорядочен по возрастанию

   for (step = 1 до n)
   {
      // Переместить наибольший элемент кучи, т.е. элемент
      // anArray[0], в начало упорядоченной части, поменяв
      // элементы местами
      Поменять местами элементы anArray[0] и anArray[last]

      // Увеличить упорядоченную область,
      // уменьшив область кучи
      Уменьшить значение переменной last на единицу

      // Снова создать кучу из элементов области кучи
      heapRebuild(anArray, 0, last)
   } // Конец оператора for
```

На рис. 11.19 показаны результаты трассировки алгоритма пирамидальной сортировки кучи, изображенной на рис. 11.17. Реализация этого алгоритма на языке C++ предоставляется читателям в качестве упражнения.

Эффективность пирамидальной сортировки имеет порядок O(n*logn)

Анализ эффективности пирамидальной сортировки практически совпадает с анализом сортировки слиянием, проведенным в главе 9. Эффективность обоих алгоритмов имеет порядок $O(n*\log n)$. Пирамидальная сортировка имеет преимущество над сортировкой слиянием, поскольку для ее выполнения не нужен второй массив. В среднем быстрая сортировка также имеет эффективность порядка $O(n*\log n)$, но в худшем случае ее эффективность оценивается величиной $O(n^2)$. И все же, несмотря на низкую эффективность в худшем случае, алгоритм быстрой сортировки считается лучшим методом.

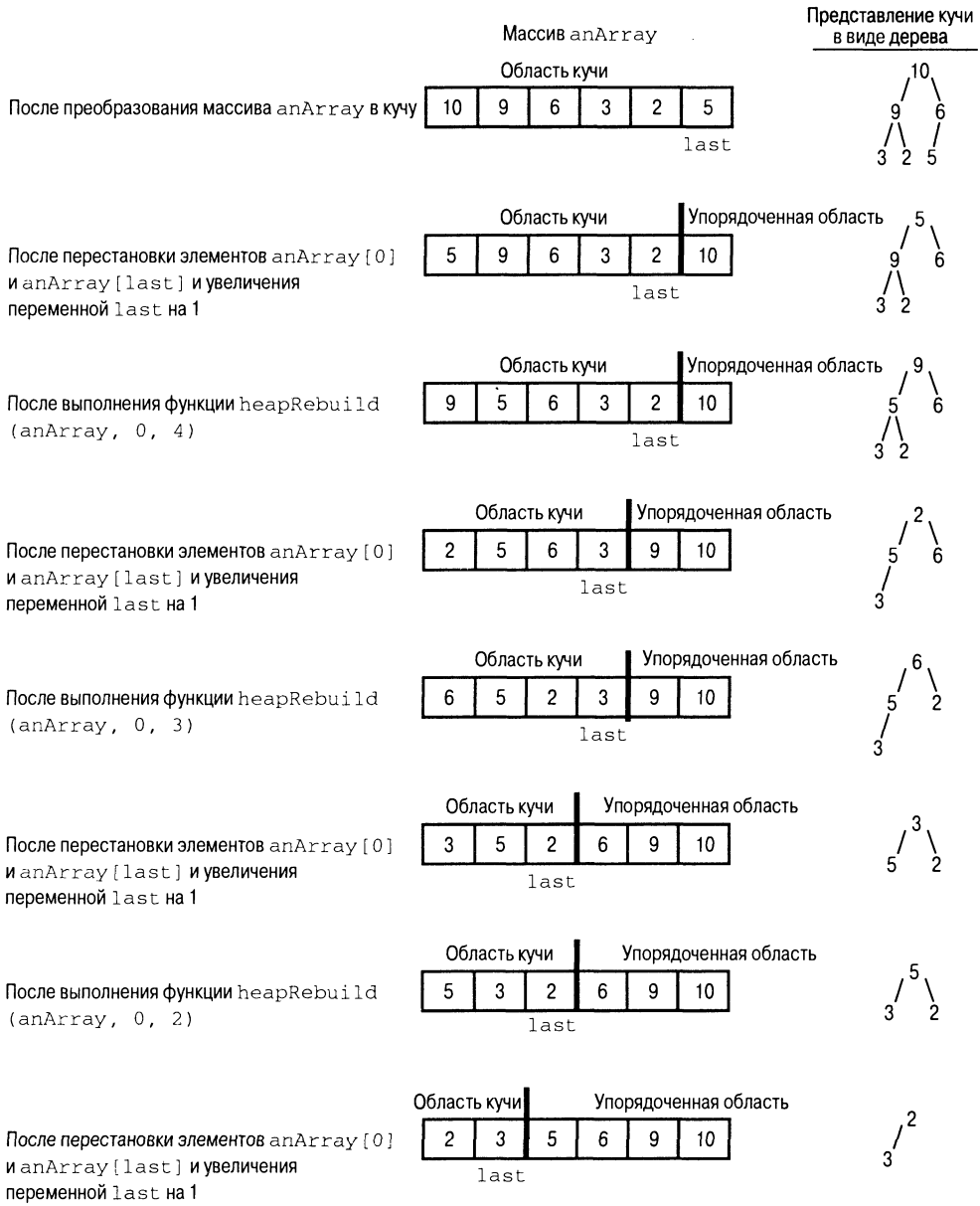

Массив anArray
Представление кучи в виде дерева
Область кучи
После преобразования массива anArray в кучу
10 9 6 3 2 5
last
Область кучи
Упорядоченная область
После перестановки элементов anArray[0] и anArray[last] и увеличения переменной last на 1
5 9 6 3 2 10
last
Область кучи
Упорядоченная область
После выполнения функции heapRebuild (anArray, 0, 4)
9 5 6 3 2 10
last
Область кучи
Упорядоченная область
После перестановки элементов anArray[0] и anArray[last] и увеличения переменной last на 1
2 5 6 3 9 10
last
Область кучи
Упорядоченная область
После выполнения функции heapRebuild (anArray, 0, 3)
6 5 2 3 9 10
last
Область кучи
Упорядоченная область
После перестановки элементов anArray[0] и anArray[last] и увеличения переменной last на 1
3 5 2 6 9 10
last
Область кучи
Упорядоченная область
После выполнения функции heapRebuild (anArray, 0, 2)
5 3 2 6 9 10
last
Область кучи
Упорядоченная область
После перестановки элементов anArray[0] и anArray[last] и увеличения переменной last на 1
2 3 5 6 9 10
last



Рис. 11.19. Трассировка алгоритма пирамидальной сортировки кучи, изображенной на рис. 11.17

Резюме

1. Абстрактная таблица поддерживает операции, ориентированные на значения, например:

   Извлечь всю информацию о Джоне Брауне

2. Линейные реализации таблицы (в виде массива или связанного списка) подходят лишь для немногих ситуаций, когда таблица невелика или ориентирована на узкий круг операций. В этих случаях простота линейных реализаций становится их основным преимуществом. Однако, как правило, линейные реализации таблицы не применяются.
3. Нелинейная реализация абстрактной таблицы (в виде бинарного дерева поиска) воплощает в себе лучшие особенности линейных структур. Реализация таблицы в виде связанного списка позволяет динамически изменять ее размеры и легко вставлять и удалять элементы, изменяя значения нескольких указателей и не сдвигая элементы таблицы. Кроме того, бинарное дерево поиска позволяет использовать алгоритм бинарного поиска для извлечения элемента по его значению. Во многих приложениях эти свойства дают нелинейной реализации таблицы большие преимущества над линейными.
4. Очередь с приоритетами является разновидностью абстрактной таблицы. Она позволяет извлекать и удалять элемент, имеющий наивысший приоритет.
5. Куча, использующая представление совершенного бинарного дерева в виде массива, является хорошей реализацией очереди с приоритетами, если максимальное количество ее элементов известно заранее.
6. Пирамидальная сортировка, как и сортировка слиянием, хорошо работает в среднем и худшем случаях, однако оценка его средней эффективности уступает быстрой сортировке. Преимуществом пирамидальной сортировки является то, что для ее выполнения не нужен второй массив.

Предупреждения

1. Выбирая абстрактный тип данных для конкретного приложения, не предусматривайте лишние операции, так как это может привести к снижению общей производительности приложения.
2. Если абстрактная таблица реализуется в виде упорядоченного массива, при вставке и удалении придется сдвигать его элементы. Сдвиг элементов снижает эффективность приложения, особенно при работе с большими таблицами.
3. Хотя реализация абстрактной таблицы в виде связанного списка позволяет избежать сдвига ее элементов, она не повышает эффективность операций вставки и удаления, поскольку в связанном списке трудно выполнить бинарный поиск.
4. Бинарное дерево поиска обеспечивает достаточно высокую эффективность операций над абстрактной таблицей. Однако в худшем случае, когда дерево имеет линейную форму, эффективность операций над таблицей сравнима с эффективностью операций над связанным списком. Если эффективность приложения должна быть высокой, следует применять реализации таблицы, описанные в главе 12.
5. Хотя куча является хорошей реализацией очереди с приоритетами, она не подходит для реализации таблиц. В частности, куча не поддерживает эффективное выполнение операций `tableInsert` и `traverseTable`.

Вопросы для самопроверки

1. Используя операции над абстрактной таблицей, напишите псевдокод операции `tableReplace`, заменяющей элемент таблицы, ключ которого равен x, элементом, ключ которого тоже равен x.
2. Является ли кучей массив, представленный на рис. 11.20?

5	1	2	8	6	10	3	9	4	7
0	1	2	3	4	5	6	7	8	9

Рис. 11.20. Массив, упоминающийся в вопросах 2, 7 и упражнении 21

3. Является ли полное бинарное дерево, изображенное на рис. 10.36, полукучей?
4. Проанализируйте кучу, представленную на рис. 11.10. Нарисуйте кучу, полученную после вставки и удаления числа 12.
5. Допустим, что в исходном положении куча h пуста. Какие элементы она будет содержать после выполнения следующих операций?

```
h.heapInsert(2)
h.heapInsert(3)
h.heapInsert(4)
h.heapInsert(1)
h.heapInsert(9)
h.heapDelete(item)
h.heapInsert(7)
h.heapInsert(6)
h.heapDelete(item)
h.heapInsert(5)
```

6. Какую кучу представляет собой очередь с приоритетами *pQueue* после выполнения следующих операций, если в начальный момент времени она была пуста?

```
pQueue.pqInsert(5)
pQueue.pqInsert(9)
pQueue.pqInsert(6)
pQueue.pqInsert(7)
pQueue.pqInsert(3)
pQueue.pqInsert(4)
pQueue.pqDelete(item)
pQueue.pqInsert(9)
pQueue.pqInsert(2)
pQueue.pqDelete(item)
```

7. Выполните оператор

```
for (index = n-1 вниз до 0)
   heapRebuild(anArray, index, n)
```

для массива, изображенного на рис. 11.20.

Упражнения

1. Завершите реализацию абстрактной таблицы в виде упорядоченного массива.
2. Операция *tableReplace(replacementItem)* позволяет найти в таблице элемент, ключ которого равен значению аргумента *replacementItem*. Если такой элемент в таблице есть, он заменяется аргументом *replacementItem*. Итак, заменяемый элемент полностью изменяется, за исключением ключа поиска.

   2.1. Напишите функцию *tableReplace* для пяти реализаций (четырех линейных и одной нелинейной) абстрактной таблицы.

   2.2. При каких условиях функция *tableReplace* заменяет элемент бинарного дерева поиска, не изменяя его структуры? (См. упражнение 10 из главы 10.)
3. Представьте себе программу, имитирующую толковый словарь. Пользователь набирает на клавиатуре слово, а программа выводит на экран его определение. В этом словаре нужна лишь операция извлечения. Какая реализация абстрактной таблицы в этом случае становится наиболее эффективной?
4. Программа проверки орфографии сравнивает слово, введенное пользователем, со словами из словаря. По мере необходимости в словарь можно добавлять новые слова. Следовательно, в таблице часто выполняются операции извлечения и редко — операции вставки. Какая реализация абстрактной таблицы в этом случае становится наиболее эффективной?
5. Для хранения идентификаторов, используемых в программе, компилятор применяет **таблицу символов** (symbol table). Обнаружив идентификатор, он производит поиск в таблице символов, чтобы проверить, не встречался ли этот идентификатор раньше. Если идентификатор встречается впервые, он вставляется в таблицу. Таким образом, для таблицы символов нужны лишь операции вставки и извлечения. Какая реализация абстрактной таблицы в этом случае становится наиболее эффективной?
6. Сделайте класс *Table* шаблонным.

7. Реализации абстрактной таблицы, рассмотренные в главе, используют следующее предположение: в любой момент времени таблица содержит не больше одного элемента с заданным ключом. Почему лучше проверять таблицы на наличие дубликатов, чем запрещать их?

   Модифицируйте реализации таблицы так, чтобы они проверяли — и запрещали — наличие дубликатов. Какие операции для этого нужны? Как это повлияет на неупорядоченные линейные реализации абстрактной таблицы?

8. Хотя в некоторых случаях имеет смысл просто запретить дубликаты (см. упражнение 7), в других случаях этого делать не следует.

   8.1. Какие последствия вызывает вставка дубликатов? Какие последствия возникают при удалении и извлечении дубликатов?

   8.2. Какие последствия вызывает вставка элементов с одинаковыми ключами? В частности, как выполняются в этом случае операции *tableInsert*, *tableDelete* и *tableReretrieve*?

9. Предположим, нужно предусмотреть удаление элементов, имеющих два разных ключа поиска (например, *tableDeleteN* для удаления элемента по имени и *tableDeleteS* — для удаления элемента по номеру карточки социального страхования). Опишите эффективную реализацию.

10. Повторите упражнение 9, удаляя элементы по имени из бинарного дерева поиска и по номеру — из упорядоченного связанного списка.

11. Реализуйте операцию *traverseTable*, позволяющую функции *visit()* удалять посещенный элемент. Аналогичная задача рассмотрена в упражнении 33 из главы 10.

12. Докажите, что корень кучи содержит наибольший поисковый ключ.

13. Влияет ли порядок вставки элементов в кучу на форму кучи? Обоснуйте свой ответ.

14. Модифицируйте реализации операций *heapInsert* и *heapRebuild* так, чтобы фактические перестановки элементов стали излишними.

15. Допустим, два элемента имеют одинаковый приоритет. Как порядок вставки элементов в очередь с приоритетами влияет на порядок их удаления оттуда? Как быть, если элементы, имеющие одинаковый приоритет, должны обрабатываться по принципу "первым пришел, первым обработан"?

16. В главе описана максимальная куча, содержащая в корне максимальный элемент дерева. Эта структура данных хорошо подходит для реализации очереди с приоритетами, поскольку ее операция *pqDelete* удаляет элемент, имеющий наивысший приоритет. Допустим, что операция *pqDelete* должна удалять из очереди элемент, имеющий наименьший приоритет. Преобразуйте реализацию максимальной кучи в реализацию минимальной кучи.

17. Допустим, нам нужно знать индекс элемента, имеющего наименьший приоритет в максимальной куче. Иными словами, в дополнение к операции *RemoveMax* нам нужна операция *retrieveMin*. Трудно ли реализовать операцию *retrieveMin* с помощью операций *pqInsert* и *removeMax*?

18. Предположим, что после размещения в очереди с приоритетами нескольких элементов, нам нужно изменить приоритет одного из них. Например, важность одной из задач, перечисленных в очереди, могла измениться. Как при этом изменится куча?

19. Сделайте классы *Heap* и *PriorityQueue* шаблонными.

20. Покажите, что в псевдокоде функции *heapSort* оператор

for (index = n-1 вниз до 0)

можно заменить оператором

for (index = n/2 вниз до 0)

21. Выполните трассировку функции heapSort для массива, изображенного на рис. 11.20.
22. Реализуйте функцию *heapSort* на языке C++.
23. Модифицируйте функцию *heapSort* так, чтобы она упорядочивала массив по убыванию.

Задания по программированию

1. Напишите реализации абстрактной таблицы в виде упорядоченного связного списка, неупорядоченного массива и неупорядоченного связного списка.
2. Напишите упорядоченные и неупорядоченные реализации абстрактной таблицы, используя абстрактный список и абстрактный упорядоченный список.
3. Выполните задание 4 из главы 10, используя для реализации адресной книги абстрактную таблицу.
4. Реализуйте таблицу символов, описанную в упражнении 5, используя класс *Table*.
5. Как показано на рис. 11.9, для реализации абстрактной очереди с приоритетами можно использовать структуры данных, отличающиеся от кучи.

   5.1. Напишите класс для реализации очереди с приоритетами в виде связанного списка.

   5.2. Напишите класс для реализации очереди с приоритетами в виде бинарного дерева поиска.
6. Допустим, нам нужно реализовать очередь, приоритеты элементов которой изменяются от 1 до 20.

   6.1. Реализуйте очередь с приоритетами в виде кучи, состоящей из очередей.

   6.2. Реализуйте очередь с приоритетами в виде массива, состоящего из очередей.
7. Проанализируйте любой набор данных, допускающий разные способы организации. Например, список сотрудников можно упорядочить по их фамилиям или по номерам, а список книг — по фамилиям авторов или заглавиям. В базе данных может содержаться и другая информация о сотрудниках или книгах, но она не должна влиять на способ ее организации. Предполагается, что ключ поиска (например, название книги или фамилия автора) уникален и представляет собой строку. Таким образом, номер сотрудника следует задавать строкой, а не числом, а для одного автора хранится запись только об одной его книге. Выберите любой набор данных, удовлетворяющий этим условиям, и запишите его в текстовый файл.

   Функционирование программы. При запуске программа должна считывать содержимое текстового файла. Предусмотрите некоторые типичные операции над базами данных, а также интерфейс пользователя. Например, обеспечьте операции вставки, удаления, извлечения и вывода на экран определенного элемента, а также операцию вывода на экран всех элементов, упорядоченных по ключу. Для удаления или вывода элемента на экран предусмотрите использование одного из двух ключей.

Замечание о реализации. Элементами базы данных должны быть объекты, содержащие два поисковых ключа и дополнительные данные. Таким образом, нужно разработать и реализовать класс таких объектов.

Хотя программа может создавать две таблицы объектов — по одному из ключей поиска (например, по фамилиям сотрудников и по их номерам), — это может привести к дополнительным затратам памяти, вследствие дублирования информации.

Лучше модифицировать абстрактную таблицу и предусмотреть операции, выполняемые по двум ключам поиска. Например, можно предусмотреть удаление записи по фамилии сотрудника или по его номеру. В основу такой реализации следует положить бинарное дерево поиска. Фактически для разной организации данных потребуется создать два бинарных дерева поиска: например по имени и по номеру.

Чтобы избежать дублирования информации, храните данные в абстрактном списке и разрешите каждому элементу бинарного дерева поиска хранить указатель на элемент списка, а не сам элемент.

Программа должна настраиваться на определенный тип данных (сотрудники, книги и т.д.). Например, можно указать описание поискового ключа с помощью пользовательского интерфейса.

8. Напишите интерактивную программу, отслеживающую поступление пациентов в приемный покой больницы. Программа должна учитывать как поступление, так и выписку пациента из больницы, а также выводить информацию о заданном пациенте. Кроме того, программа должна управлять графиком дежурств медицинского персонала в трех приемных покоях больницы. Доктор должен делать запрос о пациенте по его фамилии и указывать его приоритет, отражающий тяжесть состояния здоровья больного. Пациенты выбираются по приоритету, причем пациенты, имеющие одинаковый приоритет, обслуживаются по принципу "первым поступил, первым осмотрен".

   Пользователь должен использовать либо однобуквенные, либо однословные команды. Попробуйте идентифицировать основные операции (извините за каламбур), а затем выберите подходящую структуру данных для их реализации. Этот подход позволит вам возвести стену между основной частью программы и реализациями данных. Интересно было бы сделать такую программу событийно-управляемой.

Эффективные реализации таблиц

В этой главе ...

Сбалансированные деревья поиска
- 2-3 деревья
- 2-3-4 деревья
- Красно-черные деревья
- AVL-деревья

Хэширование
- Функции хэширования
- Разрешение конфликтов
- Эффективность хэширования
- Чем отличается хорошая функция хэширования
- Обход таблицы: неэффективная операция при хэшировании

Одновременное применение нескольких структур данных

Резюме

Предупреждения

Вопросы для самопроверки

Упражнения

Задания по программированию

Введение. Реализация абстрактной таблицы в виде бинарного дерева поиска обладает явными преимуществами, однако, если дерево несбалансированно, эффективность этой реализации падает. В главе описываются несколько новых реализаций таблицы. Описываются различные деревья поиска, остающиеся сбалансированными в любых ситуациях, и, таким образом, обеспечивающие эффективность, сравнимую с бинарным деревом поиска.

Затем в главе рассматривается другой способ реализации абстрактной таблицы, который во многих приложениях оказывается еще более эффективным, чем бинарное дерево поиска. Алгоритм хэширования позволяет определять положение элемента на основе вычисления его ключа, не прибегая к поиску.

В заключение в главе рассматриваются структуры данных, поддерживающие несколько разных видов операций одновременно. Например, данные можно организовать по принципу FIFO, одновременно требуя, чтобы они были упорядочены.

Сбалансированные деревья поиска

Как следует из предыдущей главы, эффективность реализации абстрактной таблицы в виде бинарного дерева поиска зависит от его высоты. Выполнение операций `tableRetrieve`, `tableInsert` и `tableDelete` связано с обходом дерева от корня до искомого элемента (или до узла, который станет родительским для нового элемента). Посещая каждый узел, встреченный на пути, мы сравниваем заданное значение с ключом поиска и определяем, к какому поддереву перейти на следующем шаге. Поскольку максимальное количество узлов, встреченных на этом пути, равно высоте дерева, максимальное количество сравнений, которые необходимо выполнить, также равно этому числу.

Как известно, высота бинарного дерева поиска, состоящего из N элементов, изменяется от $[\log_2(N+1)]$ до N. Следовательно, для обнаружения искомого элемента в бинарном дереве поиска понадобится выполнить от $[\log_2(N+1)]$ до N сравнений. Таким образом, поиск элемента в бинарном дереве поиска может быть настолько же неэффективен, как последовательный поиск в связанном списке, и настолько же эффективен, как бинарный поиск в массиве. Именно эффективность была главной причиной, побуждающей выбрать бинарное дерево поиска в качестве реализации абстрактной таблицы. Наша цель — достичь эффективности, сравнимой с эффективностью бинарного поиска в массиве. Таким образом, мы стремимся достичь наилучшего функционирования бинарного дерева поиска.

Высота бинарного дерева поиска сильно зависит от порядка выполнения операций вставки и удаления элементов

Какие факторы влияют на высоту бинарного дерева поиска? Как мы видели в главе 10, высота бинарного дерева поиска сильно зависит от порядка выполнения операций вставки и удаления элементов. Рассмотрим, например, бинарное дерево поиска, содержащее элементы[1] 10, 20, 30, 40, 50, 60 и 70. Если элементы были вставлены в дерево в возрастающем порядке, мы получим бинарное дерево поиска максимальной высоты, как показано на рис. 12.1, *а*. Если порядок вставки элементов был иным, а именно: 40, 20, 60, 10, 30, 50 и 70, то получится сбалансированное бинарное дерево поиска, приведенное на рис. 12.1, *б*, имеющее минимальную высоту.

[1] Как и в главе 10, элементы дерева представляют собой записи, содержащие поисковый ключ. На диаграммах деревьев, приведенных в этой главе, указываются только значения ключей, а обсуждение ведется так, будто записи, кроме ключей, больше ничего не содержат.

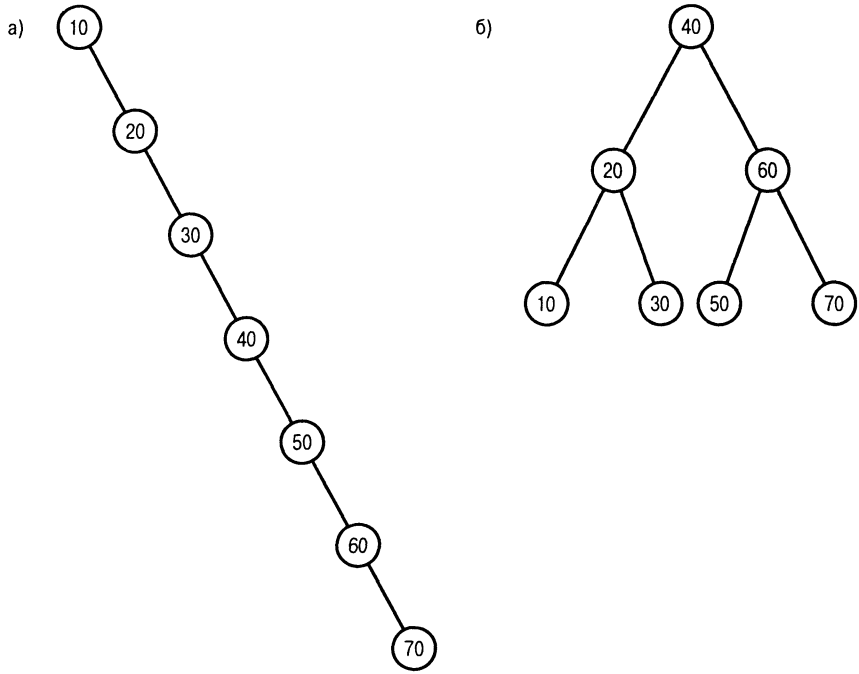


Рис. 12.1. Примеры бинарного дерева поиска: а) бинарное дерево поиска, имеющее максимальную высоту; б) бинарное дерево поиска, имеющее минимальную высоту

Различные деревья поиска могут сохранять баланс независимо от порядка выполнения операций вставки и удаления

Если применить к бинарному дереву поиска алгоритмы вставки и удаления, описанные в главе 10, дерево может стать несбалансированным и принять линейную форму. Такое дерево ничем не лучше связанного списка. Поэтому во многих приложениях желательно использовать вариации основного бинарного дерева поиска. Такие деревья не теряют баланса при выполнении операций вставки и удаления, поэтому их высота близка к минимальной. Кроме того, эффективность поиска элемента в таких деревьях близка к максимальной. Попрежнему будем предполагать, что каждый элемент дерева содержит уникальный ключ, т.е. дубликаты запрещены.

2-3 деревья

Дерево называется **2-3 деревом** (2-3 tree), если каждый внутренний узел (не лист) имеет либо два, либо три дочерних узла. Например, на рис. 12.2 показано 2-3 дерево, высота которого равна 3. Узел, имеющий два дочерних узла, называется **двухместным узлом** (2-node) . Все узлы бинарного дерева являются двухместными. Соответственно, узел, имеющий три дочерних узла, называется **трехместным узлом** (3-node).

2-3 дерево не является бинарным

2-3 дерево не является бинарным, поскольку в нем существует узел, имеющий три дочерних узла. Тем не менее, 2-3 дерево напоминает полное бинарное дерево. Если конкретное бинарное дерево не содержит трехместных узлов, что в принципе возможно, оно становится похожим на полное бинарное дерево, поскольку все его

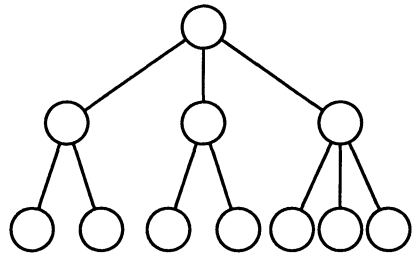


Рис. 12.2. 2-3 дерево, высота которого равна 3

внутренние узлы имеют по два дочерних узла, а все листья находятся на одном и том же уровне. Однако если какой-то из внутренних узлов 2-3 дерева имеет три дочерних узла, дерево может содержать больше узлов, чем полное бинарное дерево той же высоты. Следовательно, количество узлов в 2-3 дереве, высота которого равна h, всегда больше или равна количеству узлов в полном бинарном дереве той же высоты, т.е. всегда содержит по крайней мере 2^h-1 узлов. Иными словами, высота 2-3 дерева, содержащего N узлов, не может быть больше $[\log_2(N+1)]$, т.е. минимальной высоты бинарного дерева, состоящего из N узлов.

Высота 2-3 дерева никогда не превышает минимальную высоту бинарного дерева

Таким образом, 2-3 деревья могут оказаться полезными для реализации абстрактной таблицы. Действительно, если узлы 2-3 дерева упорядочены, его можно использовать вместо бинарного дерева поиска. Рекурсивное определение 2-3 дерева выглядит следующим образом.[2]

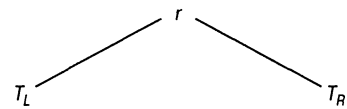
2-3 дерево

Дерево T называется 2-3 деревом, имеющим высоту h, если:

дерево T пусто (2-3 дерево, имеющее высоту 0)

или

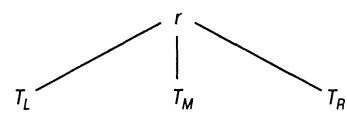
дерево T имеет вид, показанный на рисунке ниже.



Здесь r — это узел, содержащий элемент данных, а T_L и T_R — 2-3 деревья, имеющие высоту $h-1$. Поисковый ключ, содержащийся в узле r, должен быть больше всех ключей, содержащихся в левом поддереве T_L, и меньше всех поисковых ключей, содержащихся в правом поддереве T_R;

или

дерево T имеет такой вид.



[2] Как и в случае бинарных деревьев, следует различать 2-3 дерево и "2-3 дерево поиска". Предыдущее определение относилось к обычному 2-3 дереву, а приведенное ниже — к 2-3 дереву поиска. Однако, как правило, два этих понятия не различаются. В дальнейшем мы также будем считать эти понятия эквивалентными.

Здесь r — это узел, содержащий два элемента данных, а T_L, T_M и T_R — 2-3 деревья, имеющие высоту h-1. Наименьший поисковый ключ, содержащийся в узле r, должен быть больше всех ключей, содержащихся в левом поддереве T_L, и меньше всех поисковых ключей, содержащихся в **среднем** поддереве T_M. Наибольший поисковый ключ, содержащийся в узле r, должен быть больше всех ключей, содержащихся в среднем поддереве T_M, и меньше всех поисковых ключей, содержащихся в правом поддереве T_K.

Из этого определения следуют следующие правила размещения данных в узлах 2-3 деревьев.

ОСНОВНЫЕ ПОНЯТИЯ

Правила размещения данных в узлах 2-3 деревьев

1. Двухместный узел, имеющий два дочерних узла, должен содержать один элемент, поисковый ключ которого должен быть больше, чем ключи левого дочернего узла, и меньше, чем ключи правого дочернего узла, как показано на рис. 12.3, *а*.
2. Трехместный узел, имеющий три дочерних узла, должен содержать два элемента, ключи которых S и L, соответственно, удовлетворяют следующим соотношениям: ключ S больше ключей левого дочернего узла и меньше ключей среднего дочернего узла; ключ L больше ключей среднего дочернего узла и меньше ключей правого дочернего узла, как показано на рис. 12.3, *б*.
3. Лист может содержать один или два элемента.

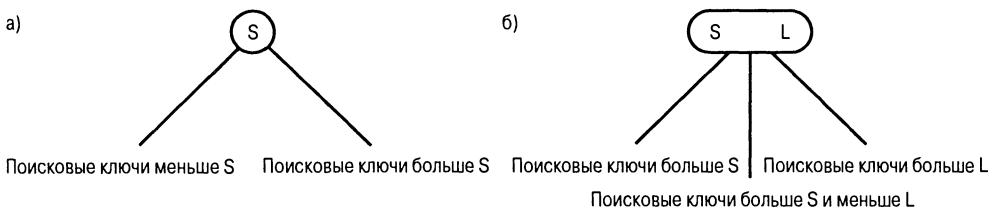


Рис. 12.3. Узлы в 2-3 дереве: а) двухместный узел; б) трехместный узел

Таким образом, элементы, содержащиеся в 2-3 дереве, упорядочены по ключу. Например, дерево, изображенное на рис. 12.4, является 2-3 деревом.

Элементы, содержащиеся в 2-3 дереве, упорядочены

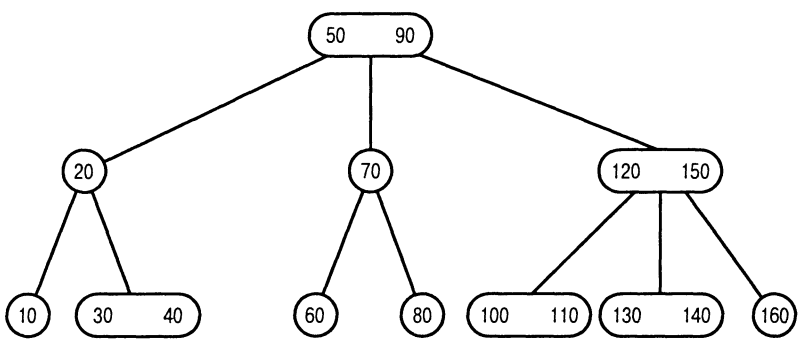


Рис. 12.4. 2-3 дерево

Узлы 2-3 дерева можно описать на языке C++ с помощью следующих операторов[3].

Узел 2-3 дерева

```
class TreeNode
{
private:
   TreeItemType smallItem, largeItem;
   TreeNode *leftChildPtr, *midChildPtr, *rightChildPtr;

   // Дружественный класс - имеет доступ к закрытым разделам
   friend class TwoThreeTree;
}; // Конец класса TreeNode
```

Если узел содержит только один элемент, его можно записать в переменную *smallItem*, а указатели *leftChildPtr* и *midChildPtr* использовать для ссылки на дочерние узлы. В целях безопасности указателю *rightChildPtr* следует присвоить константу *NULL*.

Рассмотрим операции обхода, извлечения, вставки и удаления элементов 2-3 дерева. Алгоритмы этих операций являются рекурсивными. Если выбрать в качестве базиса этих рекурсивных алгоритмов лист, а не пустое поддерево, можно упростить анализ их реализаций. Итак, будем считать, что алгоритм не получает пустое дерево в качестве своего аргумента.

Обход 2-3 дерева. 2-3 дерево можно обойти в порядке следования поисковых ключей, что соответствует симметричному обходу бинарного дерева.

Обход в порядке следования поисковых ключей

```
inorder(in ttTree:TwoThreeTree)
// Обходит непустое 2-3 дерево ttTree в
// порядке следования поисковых ключей.

   if (корень r дерева ttTree является листом)
      Посещаем элемент(ы)
   else if (корень r содержит два элемента)
   {
      inorder(левое поддерево корня)
      Посещаем первый элемент
      inorder(среднее поддерево корня)
      Посещаем второй элемент
      inorder(правое поддерево корня)
   }
   else // Корень r содержит один элемент
   {
      inorder(левое поддерево корня)
      Посещаем элемент
      inorder(правое поддерево корня)
   } // end if
```

Поиск в 2-3 дереве. Порядок следования элементов 2-3 дерева аналогичен порядку следования элементов бинарного дерева поиска. Это позволяет эффективно выполнять поиск конкретного элемента. Операция извлечения элемента из 2-3 дерева практически совпадает с операцией извлечения элемента из бинарного дерева поиска.

Поиск элемента в 2-3 дереве выполняется эффективно

[3] Для простоты конструктор пропущен.

```
retrieveItem(in ttTree:TwoThreeTree,
             in searchKey:KeyType,
             out treeItem:TreeItemType):boolean
// Извлекает из непустого 2-3 дерева ttTree и присваивает
// переменной treeItem элемент, поисковый ключ которого
// равен значению аргумента searchKey. Если такого
// элемента нет, операция не выполняется. Если элемент
// найден, функция возвращает значение true. В противном
// случае функция возвращает значение false.

  if (ключ searchKey принадлежит корню r дерева ttTree)
  {
     // Элемент найден
     treeItem = данные, содержащиеся в узле r
     return true
   }

   else if (корень r является листом)
      return false // Отказ

    // Иначе выполняем поиск в соответствующем поддереве
    else if (корень r содержит два элемента)
    {
      if (searchKey < меньшего поискового ключа узла r)
          return retrieveItem(левое поддерево узла r,
                              searchKey, treeItem)
      else if (searchKey <  большего поискового ключа узла r)
          return retrieveItem(среднее поддерево узла r,
                              searchKey, treeItem)
      else
          return retrieveItem(правое поддерево узла r,
                              searchKey, treeItem)
    }
    else // Корень r содержит один элемент
    {
      if (searchKey < поискового ключа узла r)
         return retrieveItem(левое поддерево узла r,
                             searchKey, treeItem)
      else
         return retrieveItem(правое поддерево узла r,
                             searchKey, treeItem)
    } // Конец оператора if
```

Можно ли с помощью 2-3 деревьев достичь большего, чем с помощью бинарных деревьев поиска? Эффективность поиска элемента в 2-3 дереве и бинарном дереве поиска, имеющем минимальную высоту, приблизительно одинакова, поскольку:

- высота бинарного дерева поиска, содержащего N узлов, не может быть меньше $[\log_2(N+1)]$;
- высота 2-3 дерева, содержащего N узлов, не может быть больше $[\log_2(N+1)]$;
- каждый узел в 2-3 дереве имеет по крайней мере 2 дочерних узла.

Эффективность поиска в 2-3 дереве оценивается величиной O(logN)

Однако эффективность поиска в 2-3 дереве *не превышает* эффективности поиска в бинарном дереве поиска. Это довольно неожиданно, поскольку, помимо всего прочего, узлы в 2-3 дереве могут иметь три дочерних узла, значит, высота 2-3 дерева может быть меньше самого низкого бинарного дерева поиска. Это преимущество в высоте компенсируется дополнительным временем, затрачиваемым на сравнение заданного значения с двумя ключами вместо одного. Иными словами, хотя общее количество посещаемых узлов при поиске элемента в 2-3 дереве может быть меньше, чем в бинарном дереве поиска, для каждого узла приходится выполнять больше сравнений. Вследствие этого, количество сравнений, необходимых для поиска заданного элемента в 2-3 дереве, *приближенно равно* количеству сравнений, необходимых при поиске в *сбалансированном* бинарном дереве поиска. Это число приближенно равно $\log_2 N$.

Если эффективность поиска в 2-3 дереве и бинарном дереве поиска приблизительно одинаковы, зачем вообще нужно 2-3 дерево? Дело в том, что в бинарном дереве поиска довольно трудно поддерживать баланс при выполнении операций вставки и удаления, а 2-3 дерево сохраняет форму без особых усилий. Рассмотрим два дерева, изображенных на рис. 12.5. На рис. 12.5, *а* показано бинарное дерево поиска, а на рис. 12.5, *б* — 2-3 дерево. Оба дерева содержат одинаковые элементы. Бинарное дерево поиска максимально сбалансировано, следовательно, эффективность поиска в обоих деревьях приблизительно одинакова. Однако если мы выполним последовательность операций вставки в бинарное дерево поиска, используя алгоритм, описанный в главе 10, то дерево быстро потеряет баланс, как показано на рис. 12.6, *а*. Как мы вскоре убедимся, если ту же самую последовательность операций применить к 2-3 дереву, оно не потеряет свою форму, как показано на рис. 12.6, *б*.

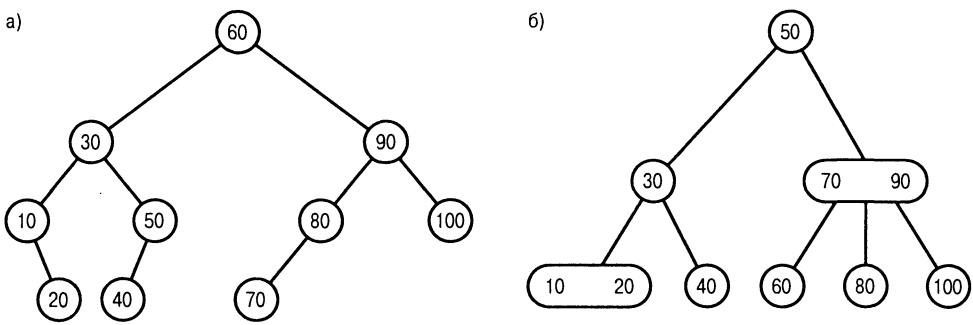


Рис. 12.5. Сравнение бинарного дерева поиска и 2-3 дерева: а) сбалансированное бинарное дерево поиска; б) 2-3 дерево с теми же элементами

Новые значения (от 32 до 39), вставленные в бинарное дерево поиска, изображенное на рис. 12.5, *а*, располагаются вдоль длинного пути (см. рис. 12.6, *а*). Операции вставки увеличивают высоту бинарного дерева поиска с 4 до 12 — высота увеличилась на 8. Новые значения, вставленные в 2-3 дерево, распределяются более равномерно (см. рис. 12.6, *б*). Следовательно, высота полученного дерева увеличится лишь на 1.

Вставка в 2-3 дерева. Поскольку узлы 2-3 дерева могут иметь два или три дочерних узла и содержать одно или два значения, элементы в такое дерево можно вставлять, не нарушая его формы. Рассмотрим, как вставляются в 2-3 дерево элементы, показанные на рис. 12.6, *б*.

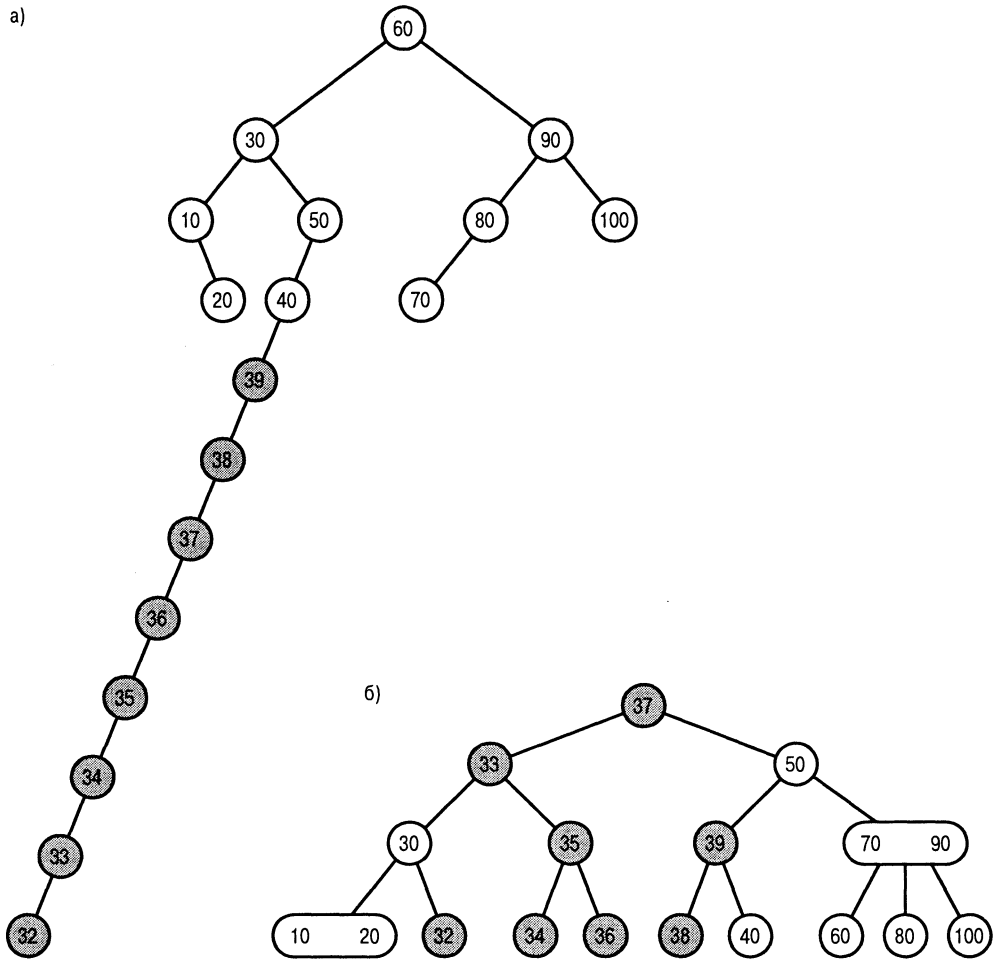


Рис. 12.6. Бинарное дерево поиска и 2-3 дерево после вставки элементов: а) бинарное дерево поиска, изображенное на рис. 12.5, а после выполнения последовательности операций вставки; б) 2-3 дерево, показанное на рис. 12.5, б, после тех же операций

Вставка в двухместный лист затруднений не вызывает

Вставка числа 39. Как и для бинарного дерева поиска, на первом шаге операции вставки в 2-3 дерево выполняется поиск узла, подходящего для размещения числа 39. Для этого можно применить стратегию алгоритма `retrieveItem`, описанную выше. Безуспешный поиск всегда прекращается на листе. Для дерева, представленного на рис. 12.5, *б*, поиск узла 39 прекратится на листе <40>.[4] Поскольку этот узел содержит только один элемент, новое значение можно просто вставить в него. Результат показан на рис. 12.7.

Вставка числа 38. Аналогично, в дереве, изображенном на рис. 12.7, можно найти место для числа 38. Поиск завершится на узле <39 40>. Остается лишь записать число 38 в этот узел, как показано на рис. 12.8, *а*.

[4] Угловые скобки означают узел и его содержимое.

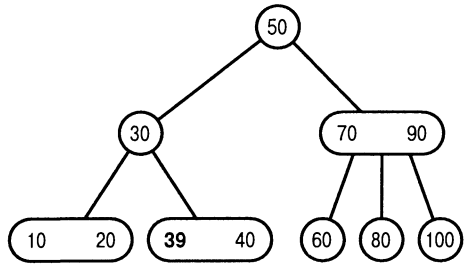
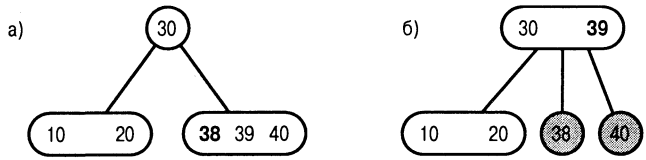


Рис. 12.7. После вставки числа 39



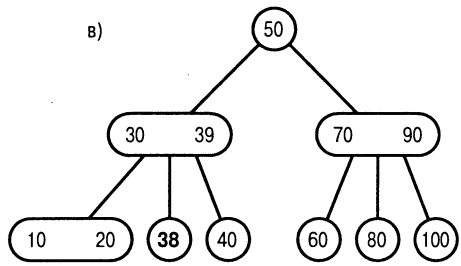


Рис. 12.8. Вставка элемента в 2-3 дерево: а) вставка элемента в двухместный лист; б) разделение переполненного узла; в) дерево, полученное в результате

Вставка в трехместный узел вынуждает разделить его на части

Однако такую операцию выполнить невозможно, поскольку узел не может содержать три значения. Однако мы можем разделить его на три новых узла, записав туда наименьшее (38), среднее (39) и наибольшее (40) значения. Затем среднее значение (39) можно переместить в родительский узел *p*, а оставшиеся, 38 и 40, разделить между двумя узлами, присоединив их к узлу *p* в качестве дочерних, как показано на рис. 12.8, *б*. Поскольку в родительский узел перенесено среднее значение, содержавшееся в узле <38 39 40>, оставшиеся числа правильно распределяются по дочерним узлам, т.е. число 38 меньше 39, которое в свою очередь меньше 40. В результате получится 2-3 дерево, изображенное на рис. 12.8, *в*.

Вставка числа 37. Вставка числа 37 в дерево, представленное на рис. 12.8, *в*, также не вызывает затруднений, поскольку число 37 принадлежит листу, который в данный момент содержит лишь одно число 38. В результате получится 2-3 дерево, изображенное на рис. 12.9.

Вставка числа 36. Стратегия поиска позволяет прийти к выводу, что число 36 принадлежит узлу <37 38> дерева, представленного на рис. 12.9. Запишем его в этот узел, как показано на рис. 12.10, *а*.

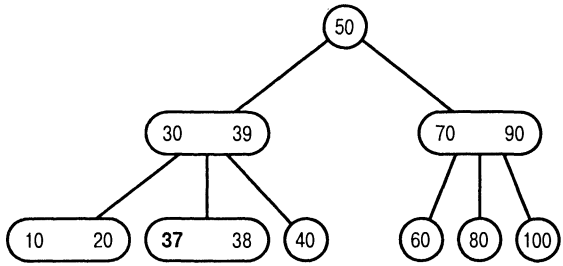


Рис. 12.9. После вставки числа 37

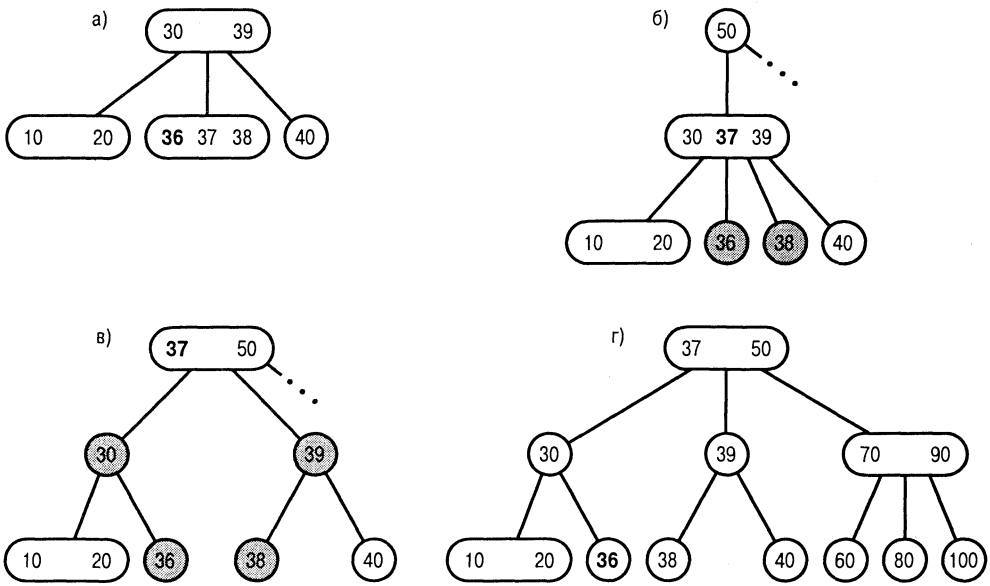


Рис. 12.10. Вставка элемента в 2-3 дерево: а) вставка в двухместный узел; б) разделение переполненного узла; в) вид дерева после вставки числа 37; г) дерево, полученное в результате

Поскольку узел <36 37 38> теперь содержит три числа, разделим его, как показано выше, на наименьшее (36), среднее (37) и наибольшее (38) значения. Затем среднее значение (37) переместим в родительский узел *p*, а оставшиеся, 36 и 38, разделим между двумя узлами, присоединив их к узлу *p* в качестве дочерних, как показано на рис. 12.10, *б*.

Однако это еще не все. В дереве остался узел <30 37 39>, содержащий три значения и имеющий четыре дочерних узла. Этот узел перенаселен и к тому же не является листом. Разделим его на наименьшее (30), среднее (37) и наибольшее (39) значения. Затем среднее значение (30) переместим в родительский узел *p*. Поскольку на это раз мы разделили на части внутренний узел, нужно как-то распределить его дочерние узлы, т.е. решить, что делать с узлами <10 20>, <36>, <38> и <40>. Левую пару дочерних узлов (<10 20> и <36>) следует присоединить к наименьшему значению (30), а правую пару (<38> и <40>) — к наибольшему значению (39), как показано на рис. 12.10, *в*. Окончательный результат показан на рис. 12.10, *г*.

Вставка чисел 35, 34 и 33. Эти числа вставляются в дерево аналогично. Результат выполненных вставок показан на рис. 12.11.

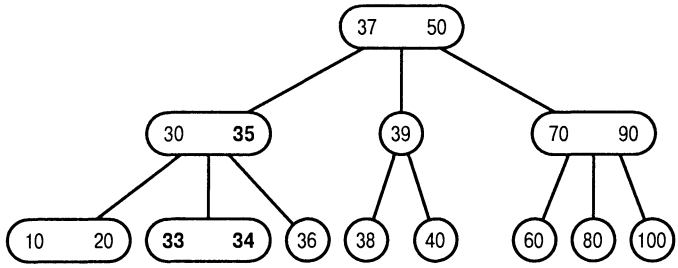


Рис. 12.11. Дерево, полученное в результате вставки чисел 35, 34 и 33

Перед последней вставкой числа 32 рассмотрим общую стратегию вставки элементов в 2-3 дерево.

Если лист содержит три элемента, его нужно разделить на два узла

Алгоритм вставки. Чтобы вставить элемент I в 2-3 дерево, сначала нужно найти лист, на котором прекращается поиск элемента I. Затем элемент I вставляется в этот лист, причем если лист содержит только два элемента, все в порядке. Однако если лист содержит три элемента, его следует **разделить** (split) на два узла: n_1 и n_2. Как показано на рис. 12.12, наименьший элемент[5] S записывается в узел n_1, наибольший L — в узел n_2, а средний элемент M перемещается в родительский узел исходного листа. Затем узлы n_1 и n_2 присоединяются к этому узлу в качестве дочерних. Если у родительского узла есть только три дочерних (и он содержит два элемента), разделение закончено. Однако если у родительского узла оказалось четыре дочерних (и он содержит три элемента), его также следует разделить.

Если внутренний узел содержит три элемента, его следует разделить на два узла, а дочерние узлы присоединить к другим узлам

Внутренний узел n, содержащий три элемента, разделяется так же, как и лист, за исключением одной детали: теперь нужно позаботиться о его четырех дочерних узлах. Как показано на рис. 12.13, узел n разделяется на узлы n_1 и n_2. Наименьший элемент S записывается в узел n_1, и к нему присоединяются два левых дочерних узла. Наибольший L — в узел n_2, и к нему присоединяются два правых дочерних узла. Средний элемент M перемещается в родительский узел исходного узла n.

Процесс разделения узлов и перемещения элементов продолжается рекурсивно, пока перед вставкой не будет обнаружен узел, содержащий только один элемент. Следовательно, после вставки он будет содержать только два элемента. Обратите внимание, что в описанной выше последовательности вставок высота дерева ни разу не увеличилась. Как правило, вставка не увеличивает высоту дерева, поскольку в нем на пути от корня к листу всегда существует по крайней мере один узел, в который можно записать новый элемент. Таким образом, стратегия вставки нового элемента в 2-3 дерево предотвращает увеличение высоты дерева намного эффективнее, чем стратегия вставки элемента в бинарное дерево поиска.

Если корень содержит три элемента, его следует разделить на два узла и создать новый корень

Рост 2-3 дерева происходит на его вершине. Высота 2-3 дерева увеличивается, лишь когда каждый узел на пути от корня к листу, в который производится вставка, содержит по два элемента. В этом случае рекурсивный процесс разделения узлов и перемещения элемен-

[5] Термин *наименьший элемент* означает элемент, содержащий наименьший поисковый ключ. Аналогичный смысл имеют термины *средний элемент* и *наибольший элемент*.

тов достигнет корня r. В этом случае корень r придется разделить на узлы r_1 и r_2, как если бы он был внутренним. Однако теперь нужно создать новый узел, который будет содержать средний элемент корня r и станет родителем узлов r_1 и r_2. Таким образом, этот новый узел станет новым корнем дерева, как показано на рис. 12.14.

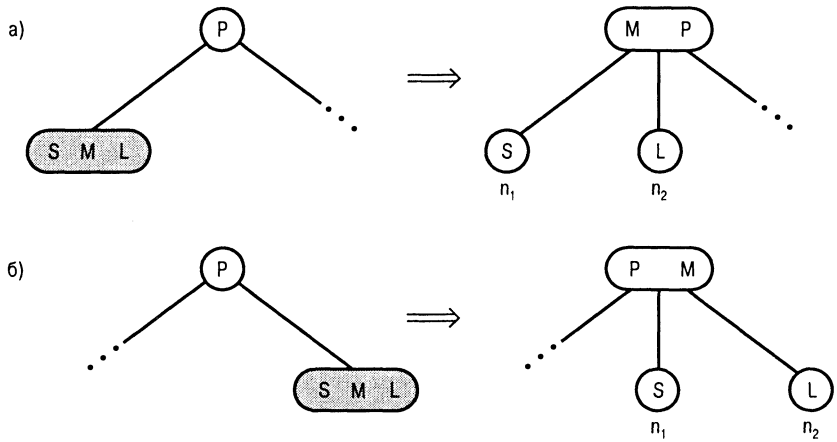


Рис. 12.12. Разделение листа в 2-3 дереве

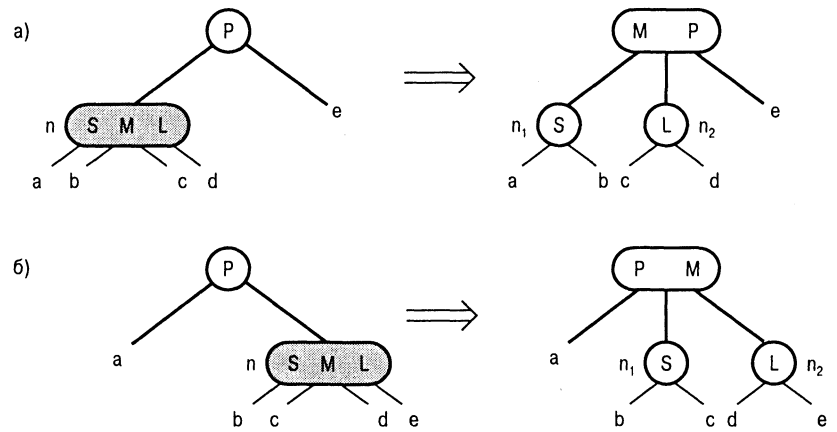


Рис. 12.13. Разделение внутреннего узла 2-3 дерева

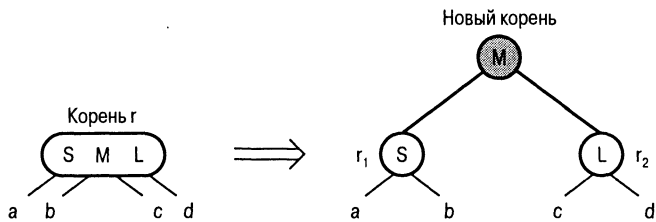


Рис. 12.14. Разделение корня 2-3 дерева

Опишем стратегию вставки в виде алгоритма.

```
insertItem(in ttTree:TwoThreeTree, in newItem:TreeItemType)

// Вставляет элемент newItem в 2-3 дерево ttTree,
// элементы которого имеют уникальные ключи, отличающиеся
// от ключа элемента newItem.

   Присвоить переменной sKey поисковый ключ элемента newItem
   Найти лист leafNode, которому принадлежит ключ sKey
   Добавить элемент newItem в лист leafNode

   if (узел leafNode содержит три элемента)
      split(leafNode)

split(inout n:TreeNode)
// Разделяет узел n, содержащий 3 элемента. Замечание:
// если узел n не является листом, он имеет четыре
// дочерних узла. Если узел не обнаружен, генерируется
// исключительная ситуация TreeException.

   if (узел n является корнем)
      Создает новый узел p (если это невозможно,
         генерируется исключительная ситуация)
   else
      Назначить узел p родителем узла n
      Заменить узел n двумя узлами, n1 и n2,
         так чтобы узел p был их родителем

      Записать в узел n1 наименьший элемент узла n
      Записать в узел n2 наибольший элемент узла n

   if (узел n не является листом)
   {
      Узел n1 становится родителем левых дочерних узлов
      Узел n2 становится родителем правых дочерних узлов
   }

   Переместить в узел p средний элемент узла n

   if (узел p содержит три элемента)
      split(p)
```

Вставка числа 32. Чтобы убедиться, что вы хорошо разобрались в алгоритме вставки, попробуйте вставить число 32 в 2-3 дерево, изображенное на рис. 12.11. В результате должно получиться дерево, представленное на рис. 12.6, *б*.

Еще раз сравните это дерево с бинарным деревом поиска, изображенным на рис. 12.6, *а*, и обратите внимание на значительное превосходство стратегии вставки элемента в 2-3 дерево.

Удаление элемента из 2-3 дерева. Стратегия удаления элемента из 2-3 дерева прямо противоположна стратегии вставки. Как мы видели, стратегия вставки элемента в 2-3 дерево приводила к разделению узлов и насыщению дерева. В противоположность ей, стратегия удаления выполняет слияние пустых узлов. В качестве иллюстрации рассмотрим процесс удаления чисел 70, 100 и 80 из дерева, изображенного на рис. 12.5, *б*.

Удаление числа 70. Выполняя поиск, обнаруживаем, что число 70 принадлежит узлу <70 90>. Поскольку процесс удаления всегда должен начинаться с листа, на первом шаге нужно поменять местами число 70 с его симметричным преемником — значением, которое следует за ним при симметричном обходе дерева. Поскольку число 70 меньше обоих значений, содержащихся в этом узле, его симметричный преемник (число 80) представляет собой наименьшее значение в среднем поддереве данного узла. (Симметричным преемником внутреннего узла всегда является лист.) После перестановки дерево принимает вид, показанный на рис. 12.15, *а*. Число 80 расположено правильно, поскольку оно больше, чем все числа, записанные в левом поддереве, и меньше всех чисел, записанных в правом поддереве данного узла. В то же время число 70 расположено неправильно, но это не имеет значения, поскольку на следующем шаге оно будет удалено из листа.

Удаляемое значение меняется местами со своим симметричным преемником

Как правило, после удаления числа из листа, в нем может остаться следующее значение (поскольку лист перед удалением содержал два значения). В этом случае алгоритм завершается, поскольку листья 2-3 дерева могут содержать одно значение. Однако в нашем примере узел остается пустым, как показано на рис. 12.15, *б*.

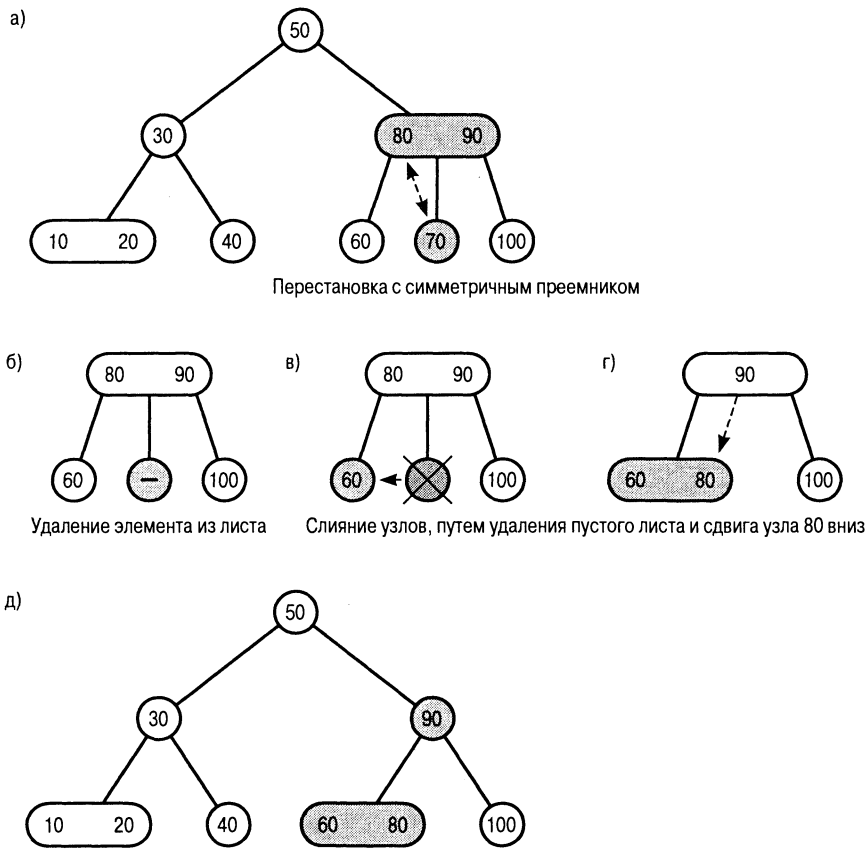


Рис. 12.15. Удаление числа 70 из 2-3 дерева: а) перестановка элементов; б) пустой узел; в) удаление пустого узла; г) перемещение элемента по дереву; д) дерево, полученное в результате

Удаление пустого узла продемонстрировано на рис. 12.15, *в*. Теперь родитель удаленного узла содержит два значения (80 и 90), имея два дочерних узла (60 и 100). Эта ситуация для 2-3 дерева не допускается (см. правило 1). Эта проблема решается просто: меньшее значение (число 80) перемещается из родительского узла в левый дочерний узел, как показано на рис. 12.15, *г*. Удаление листа и перемещение значения вниз на уровень листа называется **слиянием** (merging) листа с его братом.

Слияние узлов

Дерево, полученное в результате этого процесса удаления, показано на рис. 12.15, *д*.

Удаление числа 100. Стратегия поиска обнаруживает, что число 100 принадлежит узлу <100> дерева, изображенного на рис. 12.15, *д*. Это значение удаляется из листа, и узел становится пустым, как показано на рис. 12.16, *а*. Однако в этом случае выполнять слияние узлов не требуется, поскольку брат <60 80> содержит запасное значение. Поскольку в 2-3 дереве лист может содержать одно значение, слияние выполнять не обязательно. Если число 80 просто переместить в пустой узел, как показано на рис. 12.16, *б*, дерево окажется разрушенным: число, записанное в правом дочернем узле узла 90, оказывается меньше, в то время как оно должно было быть больше числа 90. Для того чтобы исправить ситуацию, нужно перераспределить значения среди пустых узлов, его братьев и их родителем. Для этого можно переместить большее значение (число 80) из дочернего узла в родительский, а число 90 записать в пустой узел, на место числа 80, как показано на рис. 12.16, *в*. Это позволяет сохранить порядок следования узлов и завершить процедуру удаление элемента. Дерево, полученное в результате, показано на рис. 12.16, *г*.

Перераспределение значений

Удаление числа 80. Число 80 находится во внутреннем узле дерева, изображенного на рис. 12.16, *г*. Следовательно, нужно число 80 поменять местами с его симметричным преемником, как показано на рис. 12.17. После удаления

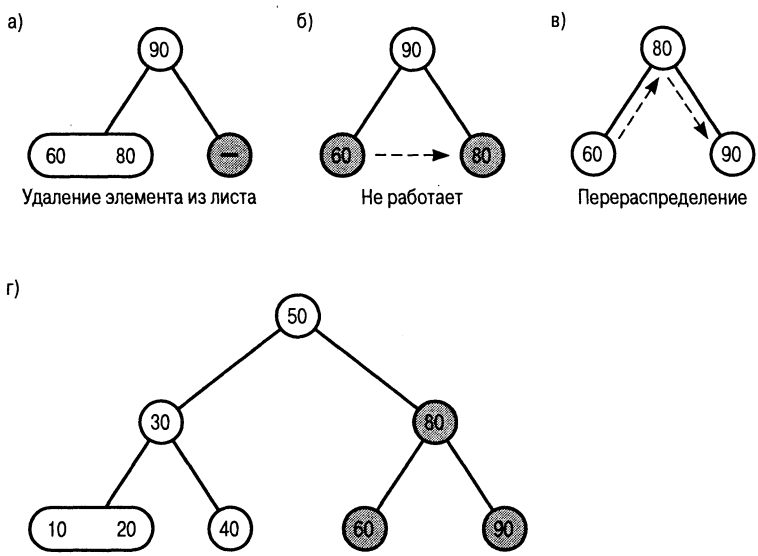


Рис. 12.16. Удаление числа 100 из 2-3 дерева: а) пустой узел; б) запрещенное перемещение элемента по дереву; в) разрешенное перемещение элемента по дереву; г) дерево, полученное в результате

числа 80 лист становится пустым. (См. рис. 12.17, *б*.) Поскольку брат пустого узла содержит только один элемент, перераспределение элементов, подобное описанному выше, невозможно. Вместо этого придется объединить узлы, перенеся число 90 из родительского узла и удалив пустой лист, как показано на рис. 12.17, *в*.

Однако это еще не все, поскольку родитель теперь не содержит никаких значений и имеет только один дочерний узел. Теперь нужно рекурсивно применить стратегию удаления к внутреннему узлу, не содержащему никаких значений. Во-первых, нужно проверить, имеет ли брат листа запасной элемент. Поскольку узел <30> содержит единственное число, перераспределение невозможно — придется выполнять слияние узлов. Слияние двух внутренних узлов производится точно так же, как и слияние листьев, за исключением дочернего узла <60 90>,

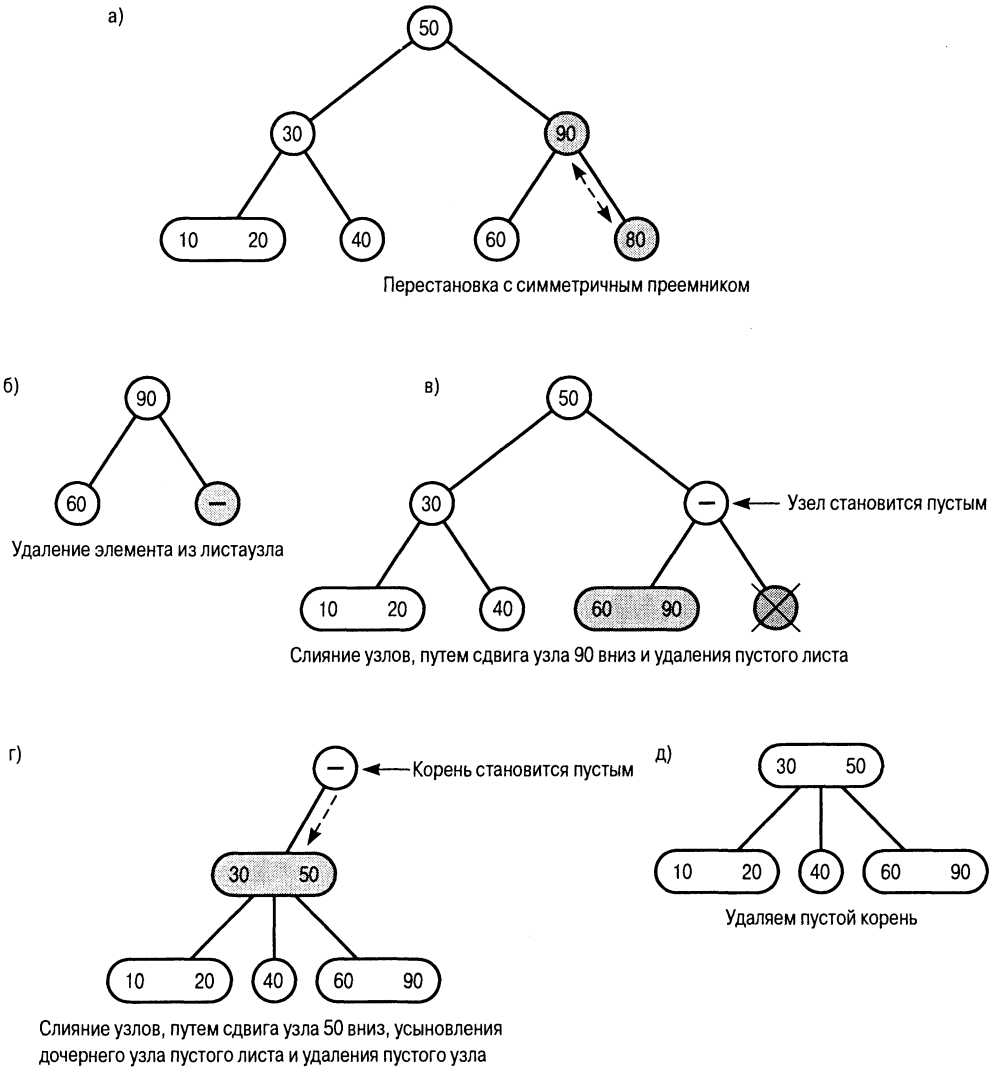


Рис. 12.17. Шаги при удалении числа 80

который нужно присоединить к новому родителю. Поскольку брат пустого узла содержит только одно значение (и, следовательно, может иметь только два дочерних узла, как утверждает правило 1), он может стать родителем узла <60 90>, только если число 50 перенести вниз. В результате возникает дерево, изображенное на рис. 12.17, *г*. Обратите внимание, что эта операция сохраняет свойства дерева.

Теперь родитель объединившихся узлов не содержит никаких значений, имеющих лишь один дочерний узел. Как правило, к такому узлу можно применить ту же рекурсивную стратегию удаления, но в данном случае ситуация другая — узел является корнем. Поскольку корень пуст, имея лишь один дочерний узел, его можно просто удалить, назначив корнем узел <30 50>, как показано на рис. 12.17, *д*. Вследствие этого удаления высота дерева уменьшится на 1. Итак, мы удалили числа 70, 100 и 80 из 2-3 дерева, представленного на рис. 12.5, *б*, и получили 2-3 дерево, изображенное на рис. 12.18, *б*. В противоположность этому, после удаления чисел 70, 100 и 80 из сбалансированного бинарного дерева поиска, показанного на рис. 12.5, *а*, мы получим дерево, изображенное на рис. 12.18, *а*. Обратите внимание, что удаление оказывает влияние лишь на одну часть бинарного дерева поиска, нарушая его баланс. На левое поддерево удаление не действует вообще, поэтому общая высота дерева не уменьшается.

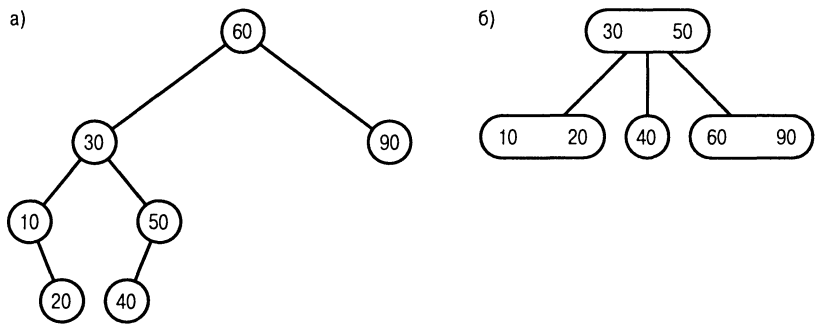


Рис. 12.18. Деревья, полученные в результате удаления чисел 70, 100 и 80: а) бинарное дерево поиска (рис. 12.5, а) после удаление чисел; б) 2-3 дерево (рис. 12.5, б) после удаления чисел

Алгоритм удаления. Итак, чтобы удалить элемент *I* из 2-3 дерева, нужно сначала найти узел *n*, которому он принадлежит. Если узел *n* является листом, мы находим симметричного преемника элемента *I* и меняем их местами. В результате перестановки удаление всегда начинается с листа. Если, кроме элемента *I*, в листе содержатся и другие элементы, нужно просто удалить элемент *I* и завершить алгоритм. Однако если элемент *I* является единственным элементом листа, то после удаления лист останется пустым. В этом случае нужно выполнить перераспределение значений.

Перераспределение значений

Сначала нужно проверить содержание братьев опустевшего листа. Если какой-нибудь брат содержит два элемента, их нужно распределить между братьями, пустым узлом и родителем, как показано на рис. 12.19, *а*. Если у листа нет братьев, содержащих два элемента, его нужно объединить с соседним братом, переместив значение из родительского узла вниз; поскольку брат содержит только один элемент, в нем есть место и для второго. Затем пустой лист удаляется из дерева. Результат этой процедуры показан на рис. 12.19, *б*.

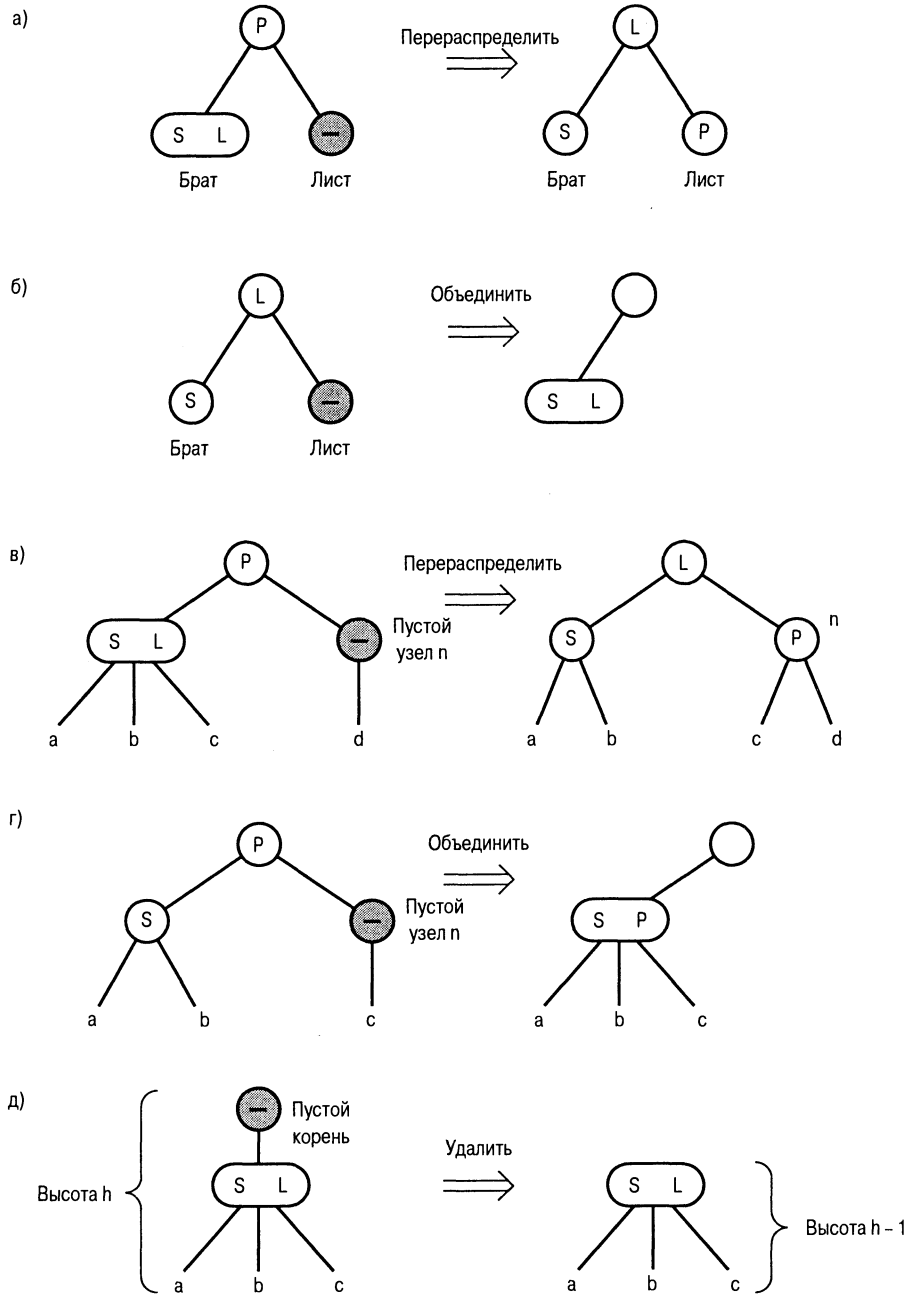


Рис. 12.19. Перераспределение элементов дерева: а) перераспределение значения; б) слияние листьев; в) перераспределение значений и дочерних узлов; г) слияние внутренних узлов; д) удаление корня

После перемещения элемента вниз по дереву узел *n* может остаться пустым, имея один дочерний узел. В этом случае к нему нужно применить рекурсивный алгоритм удаления. Таким образом, если узел *n* имеет братьев с двумя элементами (и тремя дочерними узлами), значения перераспределяются между узлом *n*, его братьями и родителем. К узлу *n* можно также присоединить один из дочерних узлов его братьев, как показано на рис. 12.19, *в*.

Слияние узлов

Если узел *n* не имеет братьев, содержащих два элемента, его следует объединить с братом, как показано на рис. 12.19, *г*. Другими словами, элемент перемещается вниз из родительского узла в дочерний, а к брату присоединяется узел, дочерний по отношению к узлу *n*. (О том, что брат имел только одно значение и два дочерних узла, было известно заранее.) Затем пустой лист удаляется из дерева. Если после слияния родительский узел остается без элементов, к нему рекурсивно применяется алгоритм удаления.

Если в результате слияний оказалось, что корень остался без элементов, имея лишь один дочерний узел, он просто удаляется, при этом высота дерева уменьшается на 1, как показано на рис. 12.19, *д*.

Алгоритм удаления элемента из 2-3 дерева имеет следующий вид.

```
deleteItem(in ttTree:TwoThreeTree, in searchKey:KeyType)
           throw TwoThreeTreeException
// Удаляет из 2-3 дерева ttTree элемент, поисковый ключ
// которого равен значению аргумента searchKey.
// Если такого элемента нет, генерируется исключительная
// ситуация TwoThreeTreeException.

   Найти позицию элемента theItem, поисковый ключ
      которого равен значению аргумента searchKey

   if (элемент theItem существует)
   {
      if (элемент theItem не является листом)
         Меняем местами элемент theItem с его симметричным
            преемником, принадлежащим узлу leafNode

   // Удаление всегда начинается с листа
   Удаляем элемент theItem из листа leafNode

   if (лист leafNode теперь пуст)
      fix(leafNode)
   }
   else
      Генерируется исключительная ситуация TwoThreeTreeException

fix(in n:TreeNode)
// Выполняет удаление, если узел n пуст, либо удаляя корень,
// либо перераспределяя значения, либо объединяя узлы.
// Замечание: если узел n является внутренним, он имеет
// один дочерний узел.

   if (узел n является корнем)
      Удалить корень
   else
      {
       Назначить узел p родителем узла n
```

```
        if (некий брат узла n содержит два элемента)
        {
            Перераспределить элементы между узлом n,
               его братьями и узлом p
            if (узел n является внутренним)
               Переместить соответствующий дочерний узел
                  от брата к узлу n
        }
        else // Слияние узлов
        {
            Выбрать соседнего брата s узла n
            Перенести соответствующий элемент вниз из узла p в узел s

            if (узел n является внутренним)
               Присоединить дочерний узел к узлу s

            Удалить узел n

            if (узел p теперь пуст)
               fix(p)
        } // Конец оператора if
    } // Конец оператора if
```

Детали реализации этого алгоритма на языке C++ читатели смогут изучить сами, выполнив задание 2.

2-3 дерево всегда сбалансировано

Возникает вопрос, насколько велики издержки алгоритмов вставки элементов в 2-3 дерево и удаления их оттуда. После обнаружения подходящей позиции алгоритмы вставки и удаления иногда выполняют дополнительную работу, например, разделение и слияние узлов. Однако эта работа незначительно влияет на общее быстродействие алгоритмов. Строгий математический анализ показывает, что объем этой работы незначителен. Иными словами, анализируя эффективность алгоритмов *insertItem* и *deleteItem*, достаточно ограничиться оценкой эффективности поиска позиции элемента.

Эффективность алгоритма поиска в 2-3 дереве имеет порядок O(logN)

Поскольку 2-3 дерево всегда сбалансировано, алгоритм поиска имеет логарифмическую эффективность. Таким образом, реализация абстрактной таблицы в виде 2-3 дерева гарантирует эффективное выполнение операций. Хотя сбалансированное дерево поиска минимизирует объем работы, необходимой для выполнения операций над абстрактной таблицей, поддерживать его баланс довольно трудно. Здесь на помощь приходит 2-3 дерево. Хотя эффективность поиска элемента в таком дереве несколько ниже эффективности, его баланс обеспечивается достаточно просто.

2-3-4 деревья

Если 2-3 дерево такое замечательное, то не окажутся ли деревья, узлы которых могут иметь более трех дочерних узлов, еще лучше? В определенном смысле, да. **2-3-4 дерево** похоже на 2-3 дерево, но может содержать **четырехместные узлы** (4-nodes) , т.е узлы, имеющие четыре дочерних узла и три элемента данных. Например, на рис. 12.20 представлено 2-3-4 дерево, высота которого равна 3. Оно состоит из тех же элементов, что и дерево, изображенное на рис. 12.6, *б*. Легко убедиться, что для выполнения операций вставки элементов в 2-3-4 дерево и удаления их оттуда требуется меньше шагов, чем при выполнении тех же операций для 2-3 дерева.

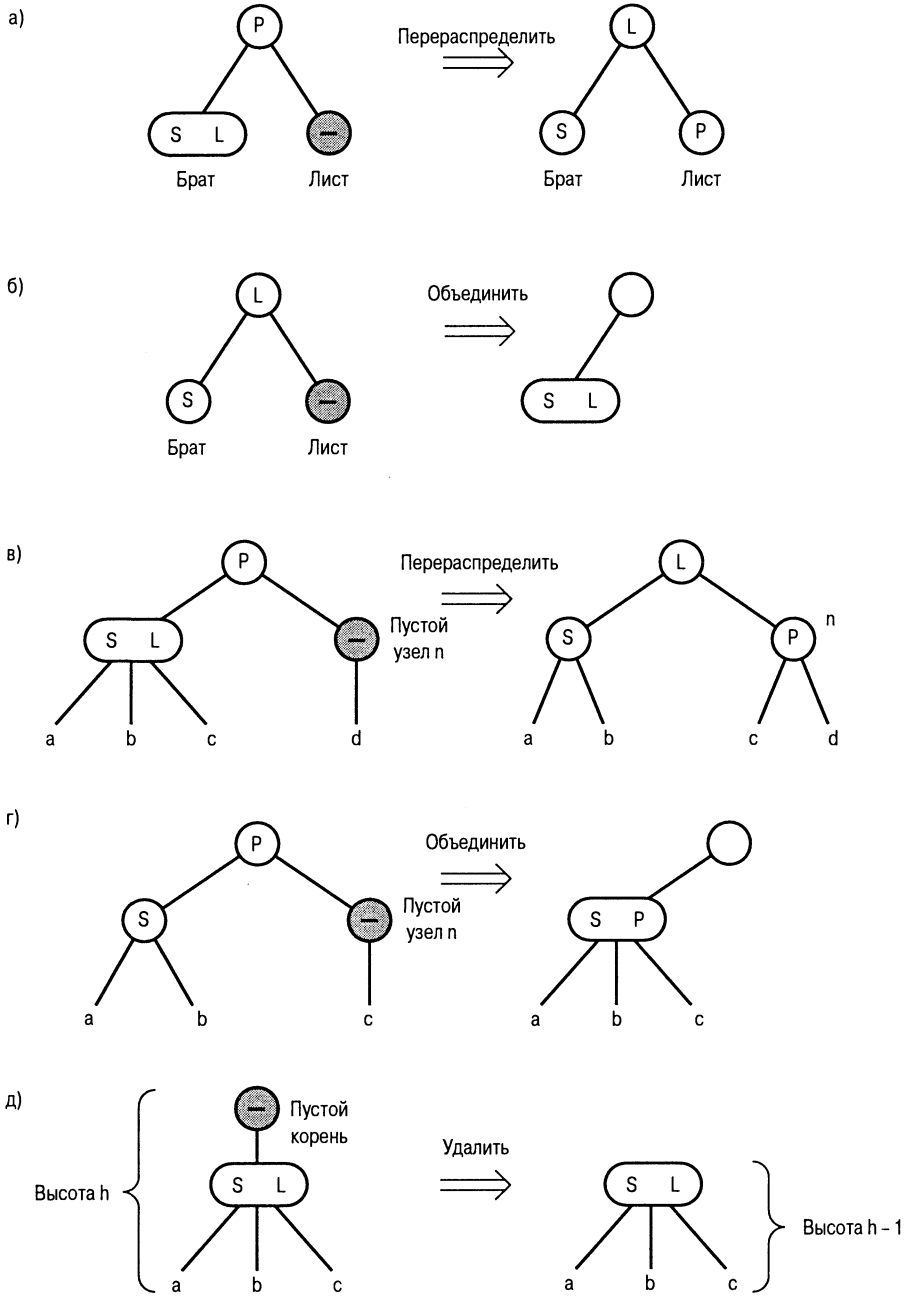


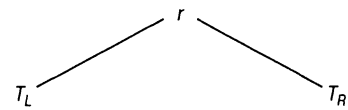
Рис. 12.20. 2-3-4 дерево, содержащее те же элементы, что и дерево, изображенное на рис. 12.6, б

Дерево T называется 2-3-4 деревом, имеющим высоту h, если: | 2-3-4 дерево

дерево T пусто (2-3-4 дерево, высота которого равна 0)

или

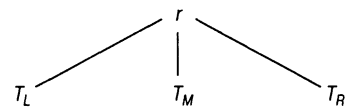
дерево T имеет вид, показанный на рисунке ниже.



Здесь r — это узел, содержащий элемент данных, а T_L и T_R — 2-3-4 деревья, имеющие высоту h-1. Поисковый ключ, содержащийся в узле r, должен быть больше всех ключей, содержащихся в левом поддереве T_L, и меньше всех поисковых ключей, содержащихся в правом поддереве T_R;

или

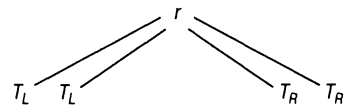
дерево T имеет вид, показанный на рисунке ниже.



Здесь r — это узел, содержащий два элемента данных, а T_L, T_M и T_R — 2-3-4 деревья, имеющие высоту h-1. Наименьший поисковый ключ, содержащийся в узле r, должен быть больше всех ключей, содержащихся в левом поддереве T_L, и меньше всех поисковых ключей, содержащихся в **среднем** поддереве T_M. Наибольший поисковый ключ, содержащийся в узле r, должен быть больше всех ключей, содержащихся в среднем поддереве T_M, и меньше всех поисковых ключей, содержащихся в правом поддереве T_K;

или

дерево T имеет вид, показанный на рисунке ниже.



Здесь r — это узел, содержащий три элемента данных, а T_L, T_{ML}, T_{MR} и T_R — 2-3-4 деревья, имеющие высоту h-1. Наименьший поисковый ключ, содержащийся в узле r, должен быть больше всех ключей, содержащихся в левом поддереве T_L, и меньше всех поисковых ключей, содержащихся в **левом среднем поддереве** T_{ML} (middle-left subtree). Средний поисковый ключ узла r должен быть больше все ключей, содержащихся в левом среднем поддереве T_{ML}, и меньше все ключей, содержащихся в **правом среднем поддереве** T_{MR} (middle-right subtree). Наибольший поисковый ключ, содержащийся в узле r, должен быть больше всех ключей, содержащихся в правом среднем поддереве T_{MR}, и меньше всех поисковых ключей, содержащихся в правом поддереве T_K.

Из этого определения следуют следующие правила размещения данных в узлах 2-3 деревьев.

ОСНОВНЫЕ ПОНЯТИЯ

Правила размещения данных в узлах 2-3-4 деревьев

1. Двухместный узел, имеющий два дочерних узла, должен содержать один элемент, поисковый ключ которого должен быть больше, чем ключи левого дочернего узла, и меньше, чем ключи правого дочернего узла, как показано на рис. 12.3, *а*.
2. Трехместный узел, имеющий три дочерних узла, должен содержать два элемента, ключи которых S и L, соответственно, удовлетворяют следующим соотношениям: ключ S больше ключей левого дочернего узла и меньше ключей среднего дочернего узла; ключ L больше ключей среднего дочернего узла и меньше ключей правого дочернего узла, как показано на рис. 12.3, *б*.
3. Четырехместный узел, имеющий четыре дочерних узла, должен содержать три элемента, подчиняющихся соотношениям, показанным на рис. 12.21: ключ S больше ключей левого дочернего узла и меньше ключей левого среднего дочернего узла; ключ M больше ключей левого среднего дочернего узла и меньше ключей правого среднего дочернего узла; ключ L больше ключей правого среднего дочернего узла и меньше ключей правого дочернего узла.
4. Лист может содержать один, два или три элемента.

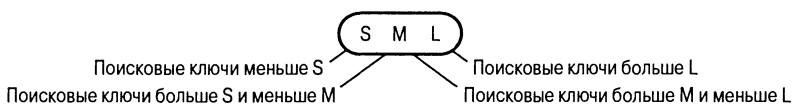


Рис. 12.21. Четырехместный узел в 2-3-4 дереве

Несмотря на то что операции вставки элементов в 2-3-4 дерево и удаления их оттуда выполняются эффективнее, чем для 2-3 деревьев, 2-3-4 деревья требуют больше памяти для хранения своих элементов, поскольку в их четырехместных узлах содержится больше элементов.

2-3-4 деревья требуют больше памяти для хранения своих элементов, чем 2-3 деревья

Узлы 2-3-4 деревьев описываются следующим классом.

```
class TreeNode
{
private:
   TreeItemType smallItem, middleItem, largeItem;
   TreeNode *leftChildPtr, *lMidChildPtr,
            *rMidChildPtr, *rightChildPtr;

   friend class TwoThreeFourTree;
}; // Конец класса TreeNode
```

Однако, как мы позднее убедимся, 2-3-4 дерево можно преобразовать в бинарное дерево специального вида, что позволяет более экономно использовать память.

Поиск элемента и обход 2-3-4 дерева. Алгоритм поиска элемента и обхода 2-3-4 дерева представляет собой расширение соответствующего алгоритма 2-3 дерева. Например, для поиска в дереве, изображенном на рис. 12.20, элемента, содержащего ключ 31, хорошо было бы обследовать левое поддерево корня, поскольку число 31 меньше, чем 37. Затем обследуется среднее поддерево узла <30 35>, поскольку число 31 лежит между 30 и 35. Поиск заканчивается указателем на левый дочерний узел узла <32 33 34>, поскольку число 31 меньше, чем 32. В результате приходим к выводу, что в дереве нет элемента, содержащего ключ 31. Детали алгоритмов поиска и обхода 2-3-4 дерева описываются в упражнении 5.

Вставка в 2-3-4 дерево. Алгоритм вставки элемента в 2-3-4 дерево, аналогично алгоритму вставки в 2-3 дерево, разбивает узел и перемещает один из его элементов в родительский узел. В дереве 2-3 алгоритм поиска проходил по пути от корня до листа, а затем возвращался обратно, разделяя узлы. Чтобы избежать этого возвращения, алгоритм вставки элемента в 2-3-4 дерево разделяет четырехместные узлы сразу при обнаружении на пути от корня к листу. Для того чтобы этот алгоритм правильно работал, родительский узел четырехместного узла не должен быть четырехместным. Это позволяет размещать в нем дополнительный элемент при разделении дочернего четырехместного узла.

Четырехместные узлы разделяются сразу после обнаружения

Рассмотрим дерево, изображенное на рис. 12.22. Это дерево, состоящее из одного узла, возникло в результате вставки в пустое 2-3-4 дерево чисел 60, 30 и 10.

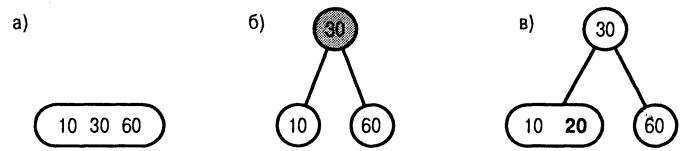


Рис. 12.22. Вставка числа 20 в 2-3-4 дерево, состоящее из одного узла

Вставка числа 20. Поиск позиции для вставки начинается с корня, представляющего собой четырехместный узел <10 30 60>. Перемещая число 30 вверх, разделяем его на три части. Поскольку этот узел является корнем, нужно создать новый корень, поместить в него число 30 и присоединить к нему два дочерних узла, как показано на рис. 12.22, *б*. Продолжаем поиск числа 20, проверяя левое поддерево корня, поскольку число 20 меньше 30. Результат показан на рис. 12.22, *в*.

Вставка 50 и 40. Для вставки чисел 50 и 40 разделять узлы не нужно. Дерево, полученное в результате, изображено на рис. 12.23.

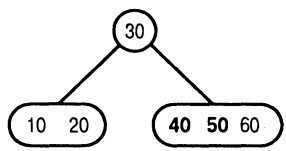


Рис. 12.23. После вставки чисел 50 и 40

Вставка числа 70. Выполняя поиск позиции для вставки числа 70 в дерево, представленное на рис. 12.23, мы обнаруживаем четырехместный узел <40 50 60>, поскольку число 70 больше 30. Перемещая число 50 вверх в родительский узел <30>, получаем дерево, изображенное на рис. 12.24, *а*. Затем вставляем число <70> в лист <30>, как показано на рис. 12.24, *б*.

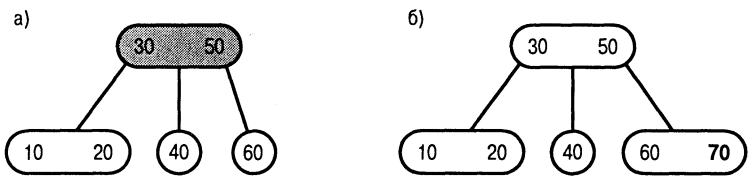


Рис. 12.24. Вставка числа 70

Вставка чисел 80 и 15. Для вставки этих чисел разделять узлы не нужно. Дерево, полученное в результате этих операций, изображено на рис. 12.25.

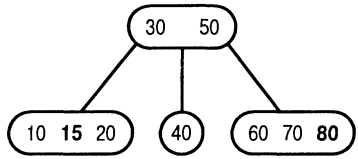


Рис. 12.25. Вставка числа 80

Вставка числа 90. Выполняя поиск позиции для вставки числа 90 в дерево, представленное на рис. 12.25, мы обнаруживаем четырехместный узел <40 50 60>, поскольку число 90 больше 50. Разделяя это узел пополам и перемещая число 70 в корень, получаем дерево, изображенное на рис. 12.26, *а*. В заключение, поскольку число 90 больше 70, вставляем число 90 в лист <80> и получаем дерево, представленное на рис. 12.26, *б*.

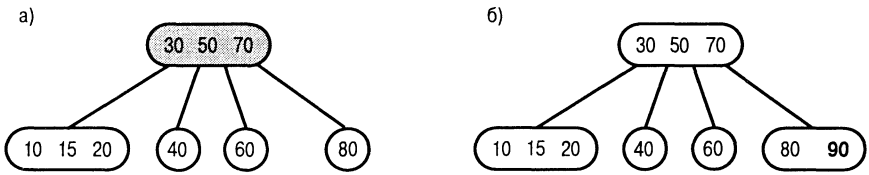


Рис. 12.26. Вставка числа 90

Вставка числа 100. Выполняя поиск позиции для вставки числа 100 в дерево, представленное на рис. 12.26, мы сразу же обнаруживаем четырехместный корень. Разделим его надвое и переместим число 50 вверх в новый корень, как показано на рис. 12.27, *а*. Затем продолжаем поиск и вставляем число 100 в узел <80 90>, получая дерево, изображенное на рис. 12.27, *б*.

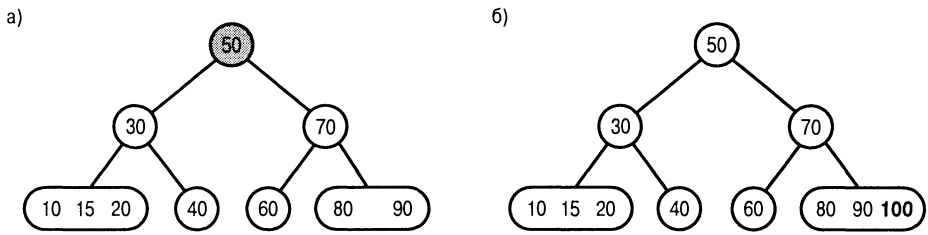


Рис. 12.27. Вставка числа 100

Разделение четырехместных узлов при вставке. Как мы видели, четырехместные узлы разделяются немедленно после их обнаружения. Четырехместный узел характеризуется следующими свойствами.

- Он может быть корнем.
- У него может быть три дочерних узла и два элемента.
- У него может быть четыре дочерних узла и три элемента.

Процедура разделения четырехместного корня проиллюстрирована на рис. 12.28. Мы уже сталкивались с такой ситуацией, разделяя узлы <10 30 60> в дереве, изображенном на рис. 12.22, *а*, и <30 50 70> в дереве, представленном на рис. 12.26, *б*. В результате получались деревья, показанные на рис. 12.22, *б* и 12.27, *а*, соответственно.

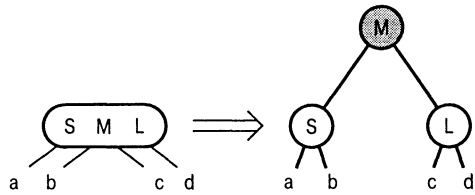


Рис. 12.28. Разделение четырехместного узла при вставке нового элемента

На рис. 12.29 продемонстрированы две возможные ситуации, в которых разделяется четырехместный узел, имеющий двухместного родителя. Например, разделяя узел <40 50 60> при вставке числа 70 в дерево, изображенное на рис. 12.23, мы получим дерево, показанное на рис. 12.24, *а*.

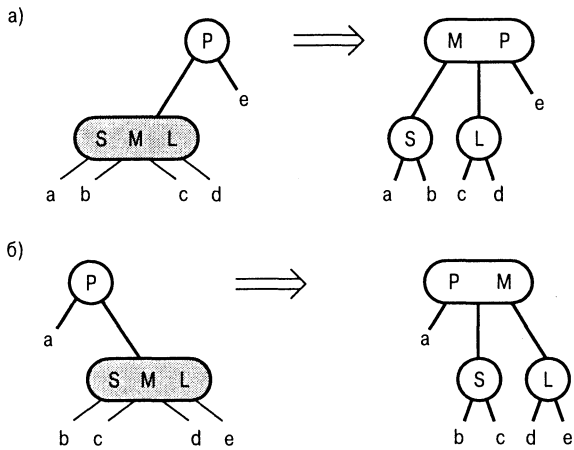


Рис. 12.29. Разделение четырехместного узла, родитель которого является двухместным

Рис. 12.30 иллюстрирует три возможные ситуации, в которых разделяется четырехместный узел, имеющий трехместного родителя. Например, разделяя узел <60 70 80> при вставке числа 90 в дерево, изображенное на рис. 12.25, мы получим дерево, показанное на рис. 12.26, *а*.

Удаление узла из 2-3-4 дерева. Алгоритм удаления узла из 2-3-4 дерева начинается точно так же, как и алгоритм удаления узла из 2-3 дерева. Сначала выполняется поиск узла *n*, содержащего заданный элемент *I*. Затем обнаруживается его симметричный преемник, который меняется местами с элементом *I*, так чтобы удаление всегда выполнялось из листа. Если лист является трех- или четырехместным, мы просто удаляем из него элемент *I*. Если у нас есть гарантии, что элемент *I* не принадлежит двухместному узлу, удаление можно выполнить за один проход дерева от корня до листа. В 2-3 дереве это сделать было невозможно. Иными словами, в 2-3-4 дереве нам не нужно возвращаться в корень и перестраивать дерево.

Чтобы достичь этой цели, нужно преобразовать каждый двухместный узел, встреченный в ходе поисков элемента *I*, в трех- или четырехместный. Здесь возможны разные варианты,

> Следует преобразовать каждый двухместный узел в трех- или четырехместный

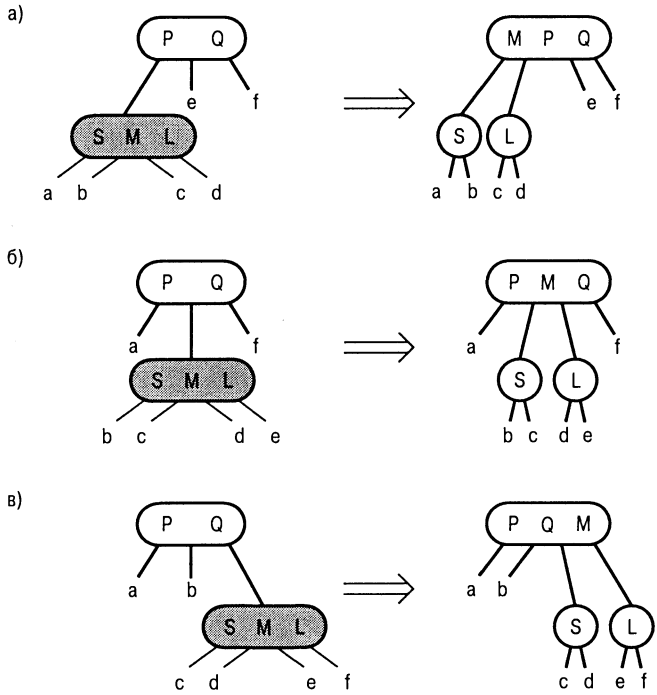


Рис. 12.30. Разделение четырехместного узла, родитель которого является трехместным

зависящие от конфигураций, образованных двухместным родителем и его ближайшими братьями. Для определенности будем считать, что ближайший брат узла является левым (если сам узел не является в свою очередь левым дочерним узлом, поскольку в таком случае его ближайший брат может быть только правым). Иными словами, либо родительский узел, либо его братья могут быть двух-, трех- или четырехместными. Например, если следующий встреченный узел оказался двухместным, а его родитель и ближайшие братья также являются двухместными, то применяется преобразование, показанное на рис. 12.28, но в обратном порядке. Если родитель узла является трехместным, в обратном порядке применяется преобразование, проиллюстрированное на рис. 12.29. Если родитель узла является четырехместным, применяется преобразование, показанное на рис. 12.30, также в обратном порядке.

Детали этого алгоритма описаны в упражнении 5.

2-3 и 2-3-4 деревья довольно удобны, поскольку легко сохраняют баланс

Заключительные замечания. Преимущество 2-3 и 2-3-4 деревьев заключается в том, что они легко сохраняют баланс, а вовсе не в их относительно небольшой высоте. Даже если 2-3 дерево ниже сбалансированного бинарного дерева поиска, разница в высоте компенсируется возросшим количеством сравнений, выполняемых алгоритмом поиска при посещении каждого узла.

Алгоритмы вставки и удаления для 2-3-4 деревьев выполняются за меньшее количество шагов, чем для 2-3 деревьев

Аналогичная ситуация наблюдается и для 2-3-4 дерева, но в этом случае алгоритмы вставки и удаления выполняются за один проход, что намного проще, чем для 2-3 дерева. По этой причине 2-3-4 деревья привлекательнее 2-3 деревьев.

Нужно ли рассматривать деревья, узлы которых могут иметь больше четырех дочерних узлов? Разумеется, высота дерева, узлы которого могут иметь 100 дочерних узлов, была бы меньше высоты 2-3-4 дерева, но алгоритм поиска элемента в таком дереве потребовал бы намного больше сравнений, выполняемых при посещении каждого узла для уточнения поддерева поиска. Следовательно, как правило, рассматривать деревья, узлы которых могут иметь больше четырех дочерних узлов, не имеет смысла. Однако их можно применять, если такие деревья реализованы на внешнем запоминающем устройстве, в котором перенос элементов из узла в узел оказывается достаточно затратной операцией. В таких случаях дерево поиска должно иметь минимальную высоту, даже за счет дополнительных сравнений, выполняемых для каждого узла. Этой теме посвящена глава 14.

Рассматривать деревья, узлы которых могут иметь больше четырех дочерних узлов, не имеет смысла

Красно-черные деревья

2-3-4 деревья привлекательны, поскольку они сбалансированы, а операции вставки и удаления выполняются за один проход от корня до листа. Однако для хранения 2-3-4 дерева требуется больше памяти, чем для хранения бинарного дерева поиска, содержащего те же самые элементы, поскольку узлы 2-3-4 дерева могут содержать до трех элементов. Обычное бинарное дерево поиска часто оказывается малопригодным, поскольку оно может быть не сбалансированным.

Для хранения 2-3-4 дерева требуется больше памяти, чем для хранения бинарного дерева поиска

Для представления 2-3-4 деревьев можно использовать **красно-черные деревья** (red-black trees), обладающие теми же преимуществами, но при этом требующее для своего хранения меньше памяти. Идея, лежащая в основе этой структуры данных, заключается в следующем: каждый трех- и четырехместный узел 2-3-4 дерева представляется эквивалентным бинарным деревом. Для того чтобы отличать двухместные узлы, принадлежащие исходному 2-3-4 дереву, и двухместные узлы, созданные из его трех- и четырехместных узлов, используются красные и черные указатели на дочерние узлы. Назовем все указатели на дочерние узлы исходного 2-3-4 дерева черными, а красными указателями будем обозначать связи с двухместными узлами, порожденными при разделении трех- и четырехместных узлов.

Красно-черное дерево обладает преимуществами 2-3-4 дерева, но при этом требует для своего хранения меньше памяти

На рис. 12.31 и 12.32 показаны способы представления трех- и четырехместных узлов в виде бинарных деревьев. Поскольку существует два способа представить трехместный узел в виде бинарного дерева, красно-черное представление 2-3-4 дерева не является единственным. Красно-черное представление дерева, изображенного на рис. 12.20, показано на рис. 12.33. На всех рисунках пунктирные линии означают красные указатели, а сплошные — черные.

Узел красно-черного дерева похож на узел бинарного дерева, но в нем должен храниться признак цвета указателя. Рассмотрим соответствующий фрагмент программы на языке C++.

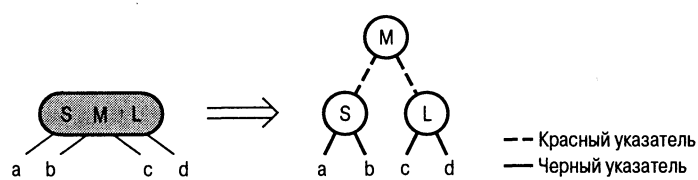


Рис. 12.31. Красно-черное представление четырехместного узла

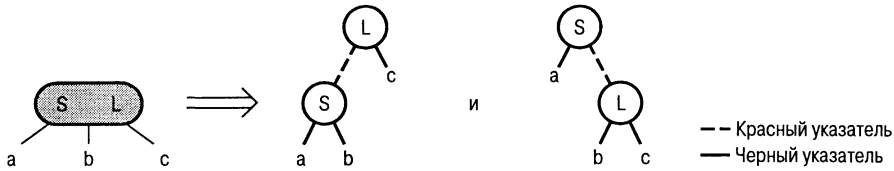


Рис. 12.32. Красно-черное представление трехместного узла

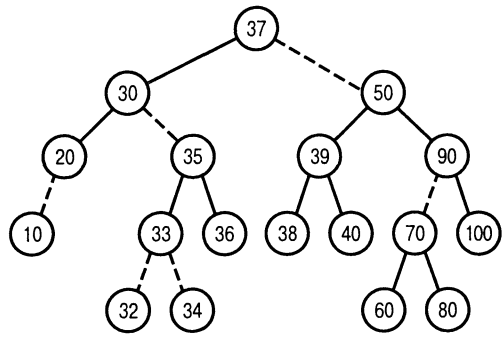


Рис. 12.33. Красно-черное представление дерева, изображенного на рис. 12.20

Узел красно-черного дерева

```
enum Color {RED, BLACK};

class TreeNode
{
private:
   TreeItemType Item;
   TreeNode *leftChildPtr, *rightChildPtr;
   Color leftColor, rightColor;

   friend class RedBlackTree;
}; // Конец класса TreeNode
```

Несмотря на то что узел красно-черного дерева должен дополнительно хранить признак цвета указателей, он занимает меньше памяти, чем узел 2-3-4 дерева. (Почему? См. упражнение 6.) Учтите, что преобразования, показанные на рис. 12.31 и 12.32, приводят к изменениям в структуре узла.

Поиск элемента и обход красно-черного дерева. Благодаря структуре красно-черного дерева, к нему можно применять алгоритмы поиска и обхода, предусмотренные для бинарного дерева поиска. Цвет указателей при этом можно просто игнорировать.

Вставка элемента в красно-черное дерево и удаление его оттуда. Поскольку красно-черное дерево на самом деле представляет 2-3-4 дерево, нужно просто настроить алгоритмы, предусмотренные для 2-3-4 деревьев, чтобы учесть цвет указателей. Напомним, что при поиске элемента в 2-3-4 дереве каждый четырехместный узел разделяется сразу после обнаружения, поэтому достаточно переформулировать эту процедуру в терминах красно-черного представления. Пример красно-черного представления четырехместного узла показан на рис. 12.31. Таким образом, чтобы идентифицировать четырехместный узел в красно-черном виде, нужно просто найти узел, содержащий два красных указателя.

Допустим, что четырехместный узел является корнем 2-3-4 дерева. На рис. 12.28 показано, как разделить его на двухместные узлы. Сравнив рис. 12.28 и 12.31, легко увидеть, что аналогичную операцию можно выполнить и над красно-черным деревом, просто изменив цвет указателей корня на черный, как показано на рис. 12.34.

Для разделения красно-черного эквивалента четырехместного узла нужно просто изменить цвет указателей

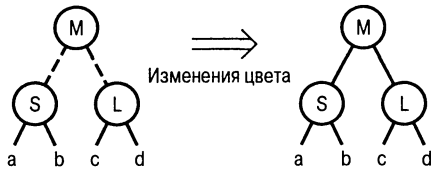


Рис. 12.34. Разделение красно-черного представления четырехместного корня

На рис. 12.29 показано, как разделить четырехместный узел, имеющий двухместного родителя. Переформулировав эту процедуру в терминах красно-черного представления (рис. 12.31 и 12.32), получим рис. 12.35. Обратите внимание, что в этом случае также нужно лишь поменять цвета указателей в красно-черном дереве.

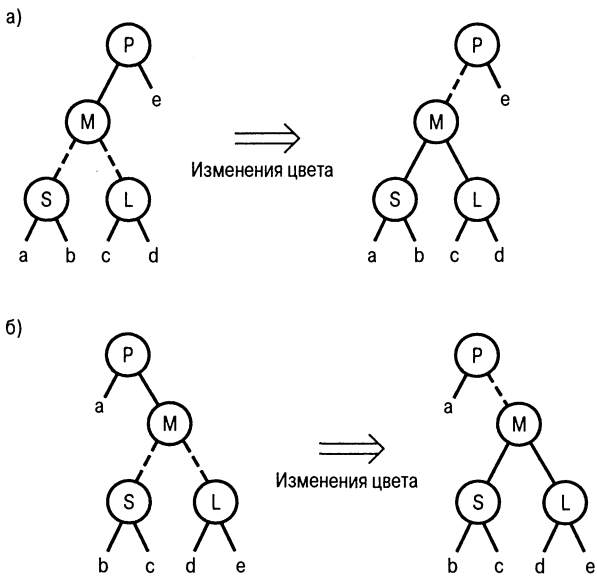
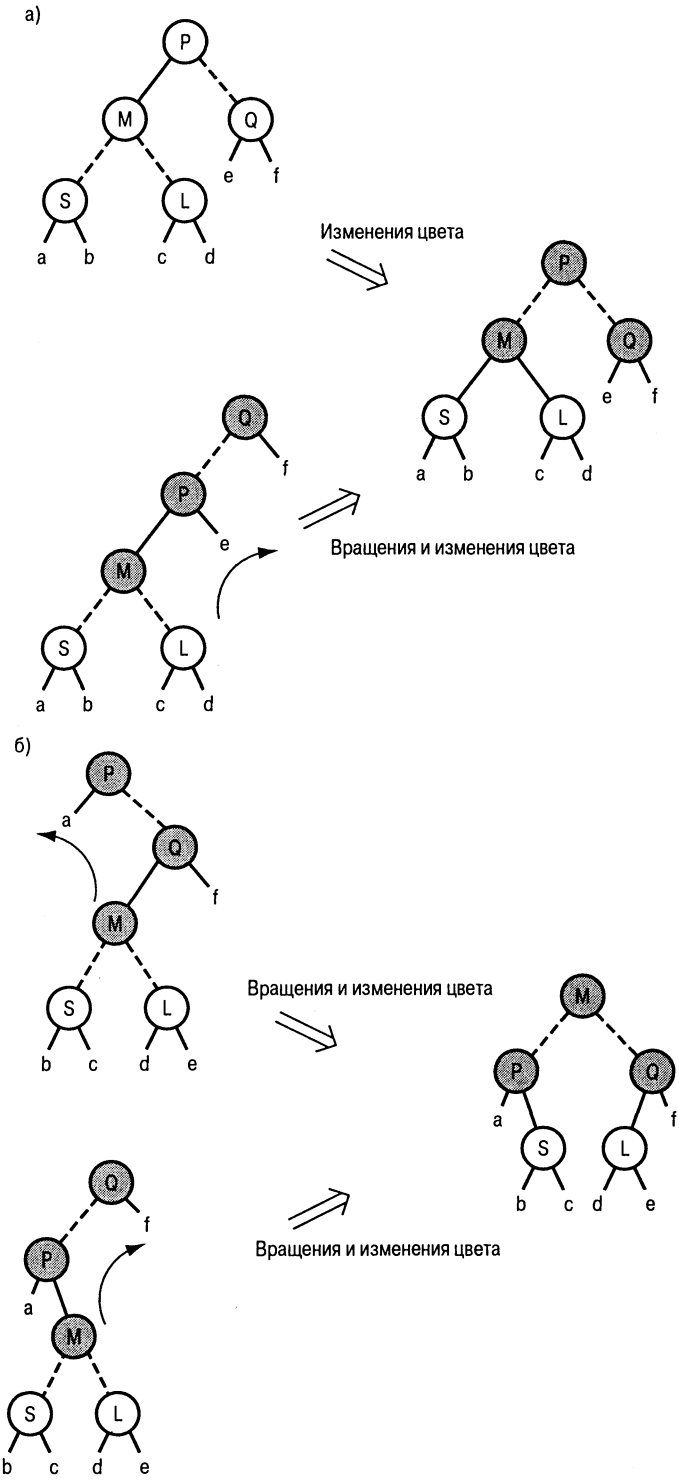


Рис. 12.35. Разделение красно-черного представления четырехместного корня, имеющего двухместного родителя

В заключение обратимся к рис. 12.30, на котором показано, как разделить четырехместный узел, имеющий трехместного родителя. Как показано на рис. 12.36, каждая конфигурация, возникающая перед разделением узла, изображенного на рис. 12.30, имеет свое красно-черное представление. (Примените к рис. 12.30 преобразования, показанные на рис. 12.31 и 12.32.) Как следует из рис. 12.36, каждая пара представлений преобразуется в одну и ту же красно-черную конфигурацию.


а)
P
M
Q
S
L
e
f
a
b
c
d
Изменения цвета
Вращения и изменения цвета
б)
Вращения и изменения цвета
Вращения и изменения цвета

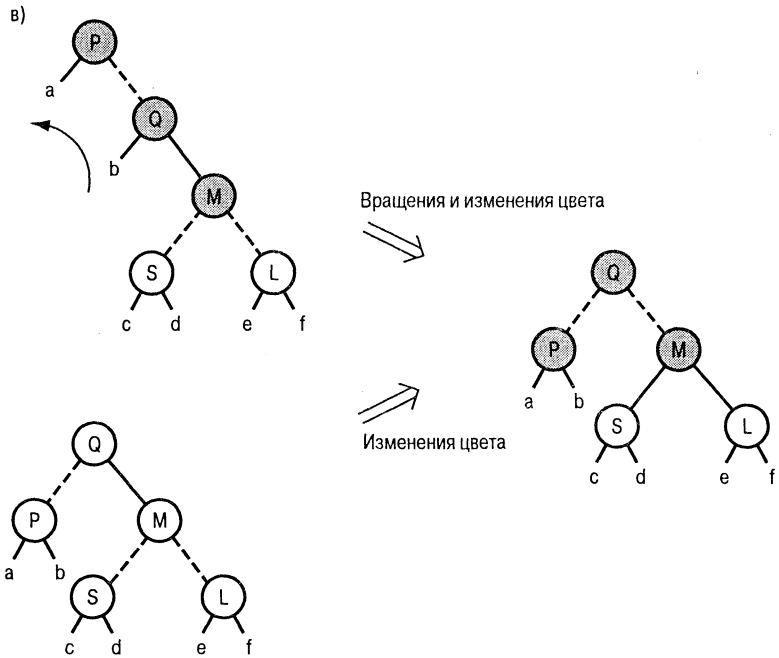


Рис. 12.36. Разделение красно-черного представления четырехместного корня, имеющего трехместного родителя

Изменения указателей, называемые вращениями, приводят к более короткому дереву

Из шести возможностей, изображенных на рис. 12.36, только две можно осуществить с помощью изменения цвета указателей. Остальные преобразования требуют изменения самих указателей. В результате таких преобразований, называемых **вращениями** (rotations), возникает более короткое дерево.

Алгоритм удаления элемента из красно-черного дерева аналогичен алгоритму удаления элемента из 2-3-4 дерева. Поскольку вставка и удаление элемента красно-черного дерева часто сводится к простому изменению цвета указателя, они более эффективны, чем соответствующие операции над 2-3-4 деревом.

Детали алгоритма вставки и удаления элемента красно-черного дерева описаны в упражнении 8.

AVL-деревья

AVL-дерево — это сбалансированное бинарное дерево поиска

AVL-дерево, названное так в честь своих изобретателей, Адельсона-Вельски (Adel'son-Vel'skii) и Ландиса (Landis), — это сбалансированное бинарное дерево поиска. Поскольку высота левого и правого поддеревьев любого узла сбалансированного бинарного дерева поиска могут отличаться не более чем на 1, поиск в AVL-дереве практически так же эффективен, как и в бинарном дереве поиска, имеющем минимальную высоту. В этом разделе вводятся лишь основные понятия, связанные с AVL-деревьями, одной из самых старых форм сбалансированных бинарных деревьев. Подробное описание этих деревьев выходит за рамки нашей книги.

Любое бинарное дерево поиска, содержащее N узлов, можно перестроить так, чтобы получить бинарное дерево поиска, имеющее минимально возможную высоту $[\log_2(N+1)]$. Напомним, что алгоритмы, описанные в главе 10, для хранения и считывания бинарного дерева поиска использовали файл. Возьмем произвольное бинарное дерево поиска, запишем его значения в файл, а затем создадим из них новое бинарное дерево поиска, имеющее минимальную высоту. Хотя этот подход вполне годился для реализации таблиц, которые записывались и считывались достаточно редко, при вставке и удалении листов несбалансированного дерева операции считывания и записи в файл оказываются слишком неэффективными. Стоимость постоянной перестройки дерева может оказаться слишком высокой и перевесить выгоды, получаемые от его минимальной высоты.

AVL-дерево минимизирует свою высоту

Метод AVL-деревьев предлагает компромисс. Он позволяет свести высоту дерева *почти* до минимума, выполнив при этом намного меньший объем работы, чем это потребовалось бы для достижения точного минимума. Основная стратегия этого метода заключается в постоянном отслеживании формы бинарного дерева поиска. Операции вставки и удаления элементов ничем не отличаются от соответствующих операций над элементами бинарного дерева поиска, но после каждой такой операции выполняется проверка, сохранило ли дерево свои свойства. Иными словами, после каждой вставки или удаления выполняется проверка, имеет ли каждый узел дерева левое и правое поддерево, высота которых отличается не более чем на 1. Допустим, что бинарное дерево поиска, изображенное на рис. 12.37, возникло в результате выполнения последовательности вставок и удалений. Высота левого и правого поддеревьев корня 30 отличается на 2. Восстановить свойства AVL-деревьев — т.е. баланс — можно, переупорядочив узлы.

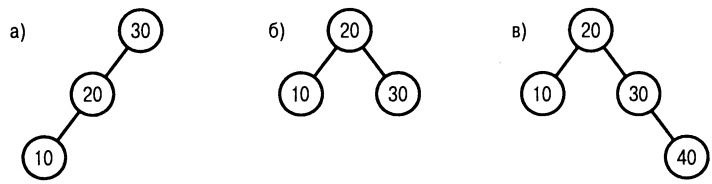


Рис. 12.37. Восстановление баланса AVL-дерева: а) несбалансированное бинарное дерево поиска; б) сбалансированное дерево после вращения; в) сбалансированное дерево после вставки

Вращения восстанавливают баланс

Например, можно **повернуть** (rotate) дерево так, чтобы узел 20 стал корнем, имеющим левый дочерний узел 10 и правый дочерний узел 30, как показано на рис. 12.37, *б*. Обратите внимание, что узлы дерева нельзя переставлять в произвольном порядке, поскольку, восстанавливая баланс, нужно соблюдать их порядок следования.

Вращения необязательно выполнять после каждой вставки или удаления. Например, в AVL-дерево, изображенное на рис. 12.37, *б*, можно вставить узел 40, и при этом свойства дерева не изменятся. (См. рис. 12.37, *в*.) Существует два вида вращений, которые необходимо выполнять для восстановления баланса. Рассмотрим каждый из них.

Допустим, после вставки или удаления узла возникло дерево, изображенное на рис. 12.38. (Например, это дерево могло возникнуть после вставки в AVL-дерево числа 60.) В узле 20 обнаруживается дисбаланс — высота левого и правого поддеревьев узла 20 отличается больше чем на 1. Для восстановления баланса необходимо выполнить **одно вращение** (single rotation). В результате мы получим дерево, изображенное на рис. 12.38, *б*. Узел 40 становится родителем

узла 20, который, в свою очередь, присоединяет к себе узел 30 в качестве правого дочернего узла. Более общий вид этого вращения представлен на рис. 12.39. В частности, там показано, что перед вращением высота левого и правого поддеревьев узла 40 равнялась h и $h+1$ соответственно. После вращения дерево оказалось сбалансированным, причем в данном конкретном случае его высота уменьшилась с $h+3$ до $h+2$. На рис. 12.40 и 12.41 показаны примеры одиночного вращения против часовой стрелки, восстанавливающего баланс дерева, но не изменяющего его высоту. Аналогичное вращение по часовой стрелке привело бы к зеркальному отражению этих примеров.

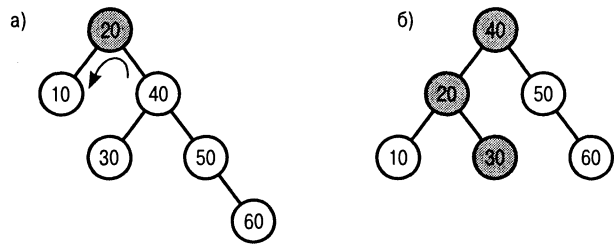


Рис. 12.38. Одиночное вращение AVL-дерева: а) несбалансированное бинарное дерево поиска; б) сбалансированное дерево после одного вращения против часовой стрелки

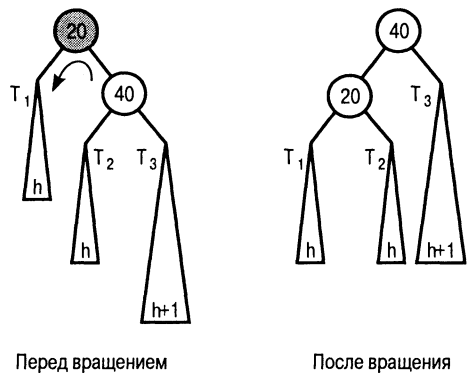


Рис. 12.39. До и после одиночного вращения против часовой стрелки, уменьшающего высоту дерева

В некоторых ситуациях могут понадобиться более сложные вращения. В качестве примера рассмотрим дерево, представленное на рис. 12.42, *а*, которое возникло в результате вставок и удаления элементов AVL-дерева. Высота левого и правого поддеревьев узла 20 отличается больше чем на 1. Для восстановления баланса нужно выполнить двойное вращение (double rotate). Результат вращения против часовой стрелки вокруг узла 20 показан на рис. 12.42, *б*, а результат вращения по часовой стрелке вокруг узла 40 — на рис. 12.42, *в*. На рис. 12.43 изображен более общий вариант двойного вращения. С помощью других двойных вращений можно было бы получить зеркальное отражение полученных результатов.

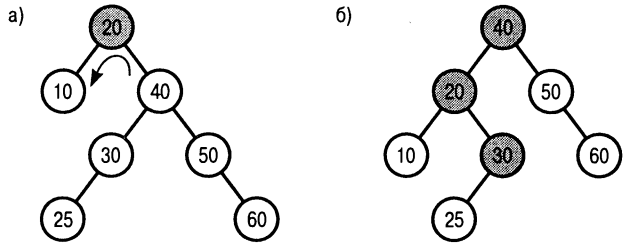


Рис. 12.40. Одиночное вращение AVL-дерева, не изменяющее его высоту; а) несбалансированное бинарное дерево поиска; б) сбалансированное дерево после одного вращения против часовой стрелки

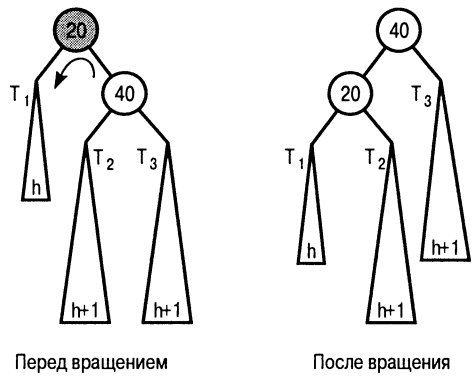


Рис. 12.41. До и после одиночного вращения против часовой стрелки, не влияюще

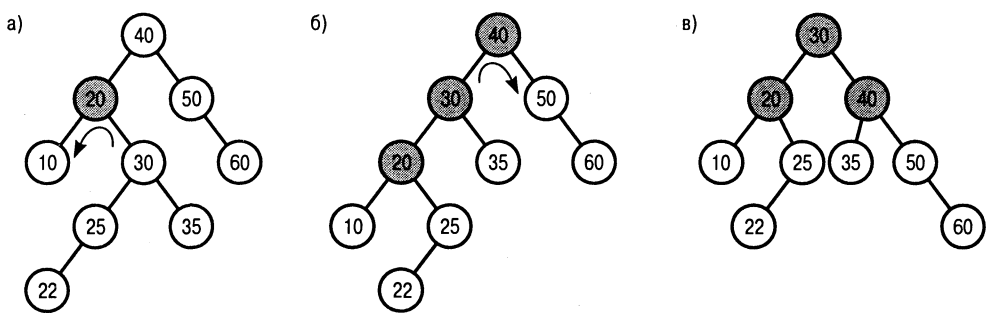


Рис. 12.42. Двойное вращение AVL-дерева: а) перед выполнением вращения; б) во время вращения; в) после двойного вращения

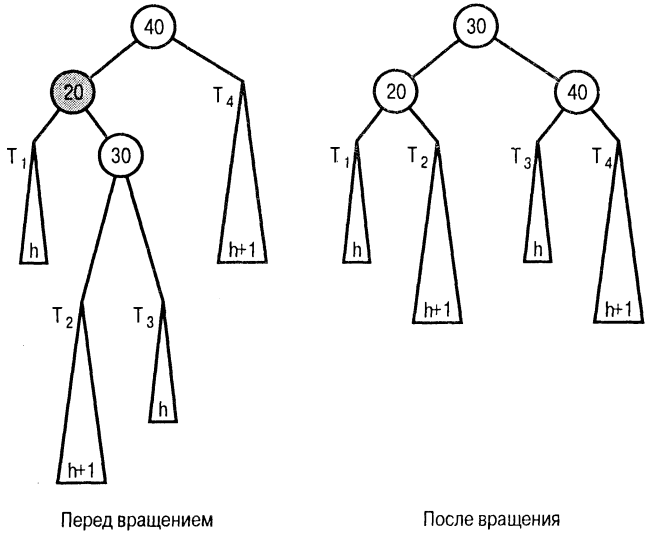


Рис. 12.43. До и после двойного вращения, уменьшающего высоту дерева

Реализовать таблицу в виде AVL-дерева труднее всего

Можно доказать, что высота AVL-дерева, содержащего N узлов, всегда очень близка к теоретическому минимуму $[\log_2(N+1)]$. Следовательно, реализация таблицы в виде AVL-дерева могла бы обеспечить эффективность, сравнимую с эффективностью бинарного дерева поиска. Однако, как правило, реализации, использующие красно-черные и 2-3-4 деревья, оказываются проще.

Хэширование

Бинарное дерево поиска и его сбалансированные варианты, такие как 2-3, 2-3-4, красно-черное и AVL-деревья, позволяют очень эффективно реализовывать абстрактную таблицу. Если, например, таблица содержит 10000 элементов, операции `tableRetrieve`, `tableInsert` и `tableDelete` выполняются за приблизительно $\log_2 10000 = 13$ шагов. Несмотря на эту впечатляющую эффективность, встречаются ситуации, когда реализация таблицы в виде дерева поиска не подходит.

Как мы знаем, время является чрезвычайно важным фактором. Например, если человек звонит 911, вызывая "Скорую помощь", система определяет телефонный номер звонившего и его домашний адрес. Аналогично, система управления воздушным движением ищет в базе данных информацию о самолете по его бортовому номеру. Совершенно очевидно, что поиск в таких базах данных должен производиться как можно быстрее.

Операции над таблицей, не предусматривающие поиск

Для немедленного определения позиции элемента (например, для вставки или удаления) необходима совершенно иная стратегия. Представьте себе массив `table`, состоящий из N элементов, в котором каждая ячейка может хранить отдельный элемент таблицы. Теперь вообразите, что у вас есть волшебная палочка под названием "механизм вычисления адреса". Получив элемент, который нужно вставить в таблицу, механизм вычисления адреса сообщит вам, в какую ячейку массива его следует поместить. Этот сценарий продемонстрирован на рис. 12.44.

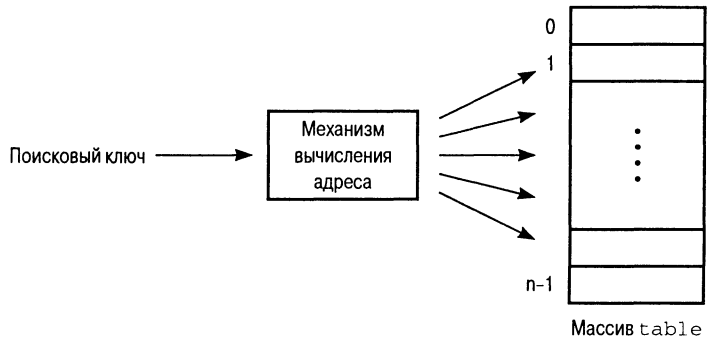


Рис. 12.44. Механизм вычисления адреса

Теперь вставка элемента в массив не вызывает затруднений.

```
tableInsert(in newItem:TableItemType)

i = индекс массива, определенный механизмом вычисления
      адреса по ключу элемента newItem
table[i] = newItem
```

Сложность операции вставки оценивается величиной O(1), т.е. время выполнения этой операции является постоянным.

Операции *tableRetrieve* и *tableDelete* также используют механизм вычисления адреса. Если из массива нужно извлечь элемент по заданному ключу, достаточно просто спросить у механизма вычисления адреса, где находится этот элемент. Поскольку вполне возможно, что этот элемент был включен в таблицу ранее с помощью операции *tableInsert*, он окажется именно на том месте, на которое укажет механизм вычисления адреса.

Таким образом, псевдокод операции *tableRetrieve* можно сформулировать следующим образом.

```
tableRetrieve(in searchKey:KeyType,
              out tableItem:TableItemType)
              throw TableException

   i = индекс массива, определенный механизмом вычисления
         адреса для элемента, ключ которого равен значению
         аргумента searchKey

   if (table[i].getKey() != searchKey)
      Генерировать исключительную ситуацию TableException
   else
      tableItem = table[i]
```

Совершенно аналогично можно описать псевдокод операции удаления.

```
tableDelete(in searchKey:KeyType)
            throw TableException

   i = индекс массива, определенный механизмом вычисления
         адреса для элемента, ключ которого равен значению
         аргумента searchKey

   if (table[i].getKey() != searchKey)
```

```
        Генерировать исключительную ситуацию TableException
    else
        Удалить элемент из ячейки table[i]
```

Таким образом, если бы у нас был механизм вычисления адреса элемента, операции *tableInsert*, *tableRetrieve* и *tableDelete* выполнялись бы практически мгновенно. Элемент вообще не пришлось бы искать. Вместо этого было бы достаточно обратиться к механизму вычисления адреса. Быстродействие этой операции постоянно и зависит только от скорости вычислений, выполняемых этим механизмом.

Функция хэширования определяет, где находится элемент массива, называемого "таблица хэширования"

Разумеется, для реализации такой схемы нужно создать механизм вычисления адреса, который определял бы местонахождение элемента, выполняя очень небольшой объем работы. На самом деле механизм вычисления адреса вовсе не такой волшебный, как кажется. Существует много устройств, которые работают приблизительно так же, такие устройства называются **функциями хэширования** (hash fubction). Описанная выше схема относится к идеальному методу **хэширования** (hashing). Массив *table*, упомянутый в этом описании, называется **таблицей хэширования** (hash table).

Чтобы понять, как работает функция хэширования, вернемся к системе скорой помощи 911, описанной выше. Допустим, для каждого человека система содержит запись, ключом которой является телефонный номер. Эти записи можно было бы хранить в дереве поиска. Хотя поиск в дереве бывает довольно быстрым, мы смогли бы еще быстрее найти нужную запись, если бы она хранилась в массиве *table*, как описано ниже. Поместим запись о человеке, имеющем телефонный номер *t*, в ячейку *table[t]*. Извлечение этой записи по ключу *t* происходит практически мгновенно. Например, запись о человеке с телефонным номером 123–4567 можно хранить в ячейке *table[1234567]*. Если бы можно было бы поместить в массив *table* десять миллионов телефонных номеров, все было бы прекрасно. Однако в таком экстравагантном способе хранения записей нет никакой необходимости. Система 911 является местной, поэтому номер телефона можно сократить, например, номер 123–4567 можно записать в ячейку *table[4567]* и работать с массивом, содержащим 10 тысяч, а не 10 миллионов ячеек.

Преобразование номера 123–4567 в индекс 4567 представляет собой простой пример функции хэширования. Функция хэширования h должна получать произвольное целое число x и ставить ему в соответствие индекс массива. В нашем примере такие индексы изменяются от 0 до 9999.

Функция хэширования отображает целое число в индекс массива

Итак, функция h задается следующим образом:

$h(x) = i$, где i — целое число, изменяющееся в диапазоне от 0 до 9999.

Поскольку база данных содержит номера всех телефонов, обслуживаемых конкретной телефонной станцией, массив *table* полностью заполнен. В этом смысле описанный выше пример хэширования не типичен и служит лишь иллюстрацией общих идей. Что, если массив будет хранить меньшее количество записей? Рассмотрим, например, систему управления воздушным движением, в которой хранятся четырехзначные номера самолетов, находящихся в данный момент в полете. Можно было бы поместить запись о рейсе 4567 в ячейку *table[4567]*, но для этого пришлось бы разместить в памяти массив, состоящий из 10000 ячеек, хотя в воздухе одновременно может находиться не более 50 самолетов.

Экономно использовать память позволяет другая функция хэширования. Например, если выделить память лишь для 101 самолета, то индексы массива `table` стали бы изменяться от 0 до 100. В таком случае функция хэширования должна ставить в соответствие каждому четырехзначному номеру самолета целое число в диапазоне от 0 до 100.

Если бы такая функция *h* существовала, можно было бы легко написать все операции над таблицей. Например, шаг алгоритма `tableRetrieve`

```
i = индекс массива, определенный механизмом вычисления
          адреса для элемента, ключ которого равен значению
          аргумента searchKey
```

можно было бы реализовать с помощью оператора

```
i = h(searchKey)
```

Например, в системе управления воздушным движением ключ `secrhKey` должен задавать четырехзначный номер самолета.

На первый взгляд, операции над таблицами выполняются мгновенно. Но так ли это на самом деле? Если бы это было правдой, другие реализации таблицы стали бы не нужны. Хэширование вытеснило бы их!

Почему хэширование не так просто, как кажется? Во-первых, поскольку хэширование основано на использовании массива, оно должно иметь уже известные нам недостатки, связанные с фиксированным размером. Кроме того, таблица хэширования должна быть достаточно большой, чтобы в ней уместились все элементы, подлежащие хранению. Однако это требование не вызывает особых затруднений, поскольку, как мы увидим далее, существуют несколько методов, позволяющих динамически увеличивать размер таблицы хэширования. Эта реализация имеет основной скрытый недостаток, даже если количество элементов массива никогда не превышает размера массива.

Идеальная функция хэширования ставит в соответствие каждому поисковому ключу единственную ячейку таблицы хэширования

В идеале функция хэширования должна каждому числу *x* ставить в соответствие единственное целое число *i*. Такая функция называется **идеальной функцией хэширования** (prefect hash function). Если бы нам были известны все возможные ключи, которые *действительно* находятся в таблице, можно было бы на самом деле создать идеальную функцию хэширования. Для системы скорой помощи 911 это условие выполняется, а для системы управления воздушным движением — нет. Обычно поисковые ключи заранее не известны.

Идеальная функция хэширования возможна, если все поисковые ключи известны заранее

На практике функция хэширования может быть неоднозначной. Разным ключам *x* и *y* она может ставить в соответствие *одно и то же* целое число. Иными словами, функция хэширования отвечает, что в ячейке `table[i]` хранятся несколько элементов. Такая ситуация называется **конфликтом** (collision).

Конфликты возникают, когда функция хэширования разным элементам ставит в соответствие одну и ту же ячейку массива

Итак, даже если количество элементов в таблице хэширования `table[0..100]` не превышает 101, функция *h* может разным элементам ставить в соответствие один и тот же индекс. Например, если два элемента имеют ключи 4567 и 7597 и

$h(4567) = h(7597) = 22,$

функция *h* отвечает, что в ячейке `table[22]` записаны два элемента. Иными словами, ключи 4567 и 7597 вступают в конфликт.

Даже если количество элементов, хранящихся в массиве в каждый момент времени, невелико, единственный способ избежать конфликтов — увеличить размер таблицы хэширования так, чтобы любой ключ мог иметь свою собственную ячейку. Например, если поисковым ключом является номер карточки социального страхования, номер ячейки должен изменяться от 000000000 до 999999999. Для этого понадобится очень много памяти! Поскольку такой подход совершенно не практичен, необходимо разработать схемы, позволяющие предотвратить конфликты. Обычно такие схемы требуют, чтобы функция хэширования равномерно распределяла элементы в таблице хэширования.

Итак, сформулируем основные требования, которые предъявляются к функции хэширования.

Требования, которые предъявляются к функции хэширования

- Функция хэширования должна быстро и легко вычисляться.
- Функция хэширования должна равномерно распределять элементы по массиву.

Обратите внимание, что размер таблицы хэширования влияет на способность функции хэширования равномерно распределять элементы. Более детально мы обсудим эту тему позднее.

Рассмотрим теперь несолько функций хэширования и **схем, предотвращающих конфликты** (collision-resolution schemes).

Функции хэширования

Функция хэширования должна оперировать целыми числами

Достаточно рассмотреть функции хэширования, аргументом которых является целое число. Почему? Если поисковый ключ не является целым числом, ему можно поставить в соответствие какое-нибудь целое число, а затем применить хэширование. В конце раздела мы рассмотрим способ преобразования строк в целые числа.

Существует много способов преобразования произвольного целого числа в целое число, *лежащее в заданном диапазоне, например от 0 до 100. Следовательно,* есть много способов создания функций хэширования. Однако многие из этих функций не удобны. Рассмотрим несколько примеров функций хэширования, оперирующих положительными целыми числами.

Выбор цифр. Если поисковым ключом является идентификационный номер сотрудника 001364825, из него можно извлечь четвертую и последнюю цифры, образовав число 35 — индекс элемента в таблице хэширования.

h(001364825) = 35 (*выбираем четвертую и последнюю цифры*).

Следовательно, элемент с ключом 001364825 можно хранить в ячейке `table[35]`.

Выбирая цифры из поискового ключа, следует быть осторожным. Например, первые три цифры номера карточки социального страхования кодируют географический регион. Если для хэширования выбрать только первые три цифры, всем людям, проживающим в определенном штате, будет соответствовать одна и та же ячейка таблицы хэширования.

Выбор цифр не обеспечивает равномерного распределения элементов в таблице хэширования

Функция хэширования, основанная на выборе цифр, быстро и легко вычисляется, однако она не обеспечивает равномерного распределения элементов в таблице хэширования. Следовательно, правильная функция хэширования должна использовать весь ключ.

Свертка. Улучшить хэширование можно с помощью суммирования всех цифр поискового ключа. Например, можно сложить все цифры числа 001364825 и получить следующий результат.

0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 = 29 (*складываем все цифры*).

Таким образом, для элемента с ключом 001364825 предназначена ячейка `table[29]`. Обратите внимание, что если сложить все цифры девятизначного числа, значение функции хэширования будет изменяться в следующих пределах:

0 ≤ *h*(поисковый ключ) ≤ 81.

Иными словами, для хранения элементов, имеющих девятизначный поисковый ключ, можно использовать таблицу хэширования `table[0..81]`. Чтобы модифицировать этот метод или увеличить размер таблицы хэширования, можно сгруппировать цифры поискового ключа и складывать группы цифр. Например, на основе ключа 001364825 можно сформировать три группы, состоящие из трех цифр, а затем сложить их.

001 + 364 + 825 = 1190.

Значения этой функции хэширования изменяются в более широких пределах:

0 ≤ *h*(поисковый ключ) ≤ 3*999 = 2997.

Очевидно, если число 2997 превышает размер таблицы хэширования, цифры можно сгруппировать иначе.

Можно применять несколько функций хэширования, хотя, на первый взгляд, это может показаться не очевидным. Например, можно выбрать определенные цифры из поискового ключа, а затем сложить их, или выбрать цифры из полученного ранее результата 2997 или применить свертку, сложив еще раз 29 и 97.

Для того чтобы поисковый ключ был единственным, необходимо применять несколько функций хэширования

Модульная арифметика. Простые и эффективные функции хэширования можно создать с помощью модульной арифметики, которую мы будет теперь применять до конца главы. Рассмотрим функцию[6]

$h(x) = x \bmod tableSize$,

где таблица хэширования `table` имеет размер *tableSize*. В частности, если значение *tableSize* равно 101, функция $h(x)=x \bmod 101$ каждому целому числу x ставит в соответствие целое число из интервала от 0 до 100, например *h*(001364825)=12.

Если $h(x) = x \bmod tableSize$, несколько чисел x отображаются в ячейку `table[0]`, несколько — в ячейку `table[0]` и т.д., иными словами, возникают конфликты. Однако, выбрав в качестве значения переменной *tableSize* простое число, можно равномерно распределить элементы по таблице, предотвратив тем самым возможные конфликты. Например, число 101 из предыдущего примера является простым. Способ выбора размера таблицы будет рассмотрен позднее. Следует иметь в виду, что число 101 приведено просто в качестве примера, как правило, размеры таблиц намного больше.

Размер таблицы должен быть простым числом

Преобразование строки символов в целое число. Если поисковый ключ является строкой символов, например, именем, то, перед тем как применить функцию хэширования *h*(*x*), его следует преобразовать в целое число. Для этого мож-

[6] Для операции деления по модулю в книге используется обозначение "mod". В языке C++ эта операция обозначается символом %.

но было бы закодировать каждый символ целым числом. Например, строке "NOTE" соответствуют ASCII-коды 78, 79, 84 и 69, которые кодируют символы N, O, T и E соответственно. Если буквам от A до Z поставить в соответствие числа от 1 до 26, то буква N окажется закодированной числом 14, буква O — числом 15, буква T — числом 20, а буква E — числом 5.

Если просто сложить эти числа, то получится новое целое число, которое может соответствовать сразу нескольким строкам. Например, строка "TONE" будет закодирована точно так же. Для того чтобы этого не случилось, нужно записать каждый символ в двоичной системе счисления и конкатенировать полученные результаты. Если буквы от A до Z закодированы числами от 1 до 26, то строке "NOTE" соответствует следующий код.

Буква N кодируется числом 14, или 01110 в двоичной системе.

Буква O кодируется числом 15, или 01111 в двоичной системе.

Буква T кодируется числом 20, или 10100 в двоичной системе.

Буква E кодируется числом 5, или 00101 в двоичной системе.

Конкатенируя двоичные величины, получаем новое двоичное число

01110011111010000101,

которое равно 474757 в десятичной системе счисления. К этому числу можно применить функцию хэширования x mod *tableSize*.

Рассмотрим теперь более эффективный способ вычисления числа 474757. Вместо преобразования двоичного числа в десятичное, можно вычислить выражение

$$14 * 32^3 + 15 * 32^2 + 20 * 32^1 + 5 * 32^0.$$

Это вполне возможно, поскольку тем самым мы представляем каждый символ 5-битовым двоичным числом, а 2^5=32.

Правило Горнера минимизирует количество вычислений

Факторизация этого выражения позволяет минимизировать количество арифметических операций. Этот прием называется **правилом Горнера** (Horner's rule). Итак, можно переписать выражение следующим образом.

$$((14 * 32 + 15) * 32 + 20) * 32 + 5.$$

Хотя оба эти выражения равны одному и тому же числу, в первом случае в ходе вычислений могут возникнуть очень большие числа, вызывающие переполнение памяти компьютера.

Поскольку для хэширования мы собираемся применить функцию

$h(x) = x$ mod *tableSize*,

применение правила Горнера позволяет избежать переполнения памяти. Реализация этого алгоритма предоставляется читателям в качестве упражнения.

Разрешение конфликтов

Рассмотрим проблему, порожденную конфликтом. Допустим, нам нужно вставить в таблицу хэширования `table` элемент, ключ которого равен 4567. Значение функции хэширования $h(x) = x$ mod 101 равно 22. Это означает, что новый элемент нужно поместить в ячейку `table[22]`. Предположим, что в этой ячейке ранее уже был записан элемент, как показано на рис. 12.45. Если в ячейке `table[22]` хранится число 7597, поскольку 7597 mod 101 = 22, куда же записать новый элемент? Очевидно, это не связано с заполненностью таблицы: конфликт может возникнуть даже тогда, когда в таблице хранится только один элемент!

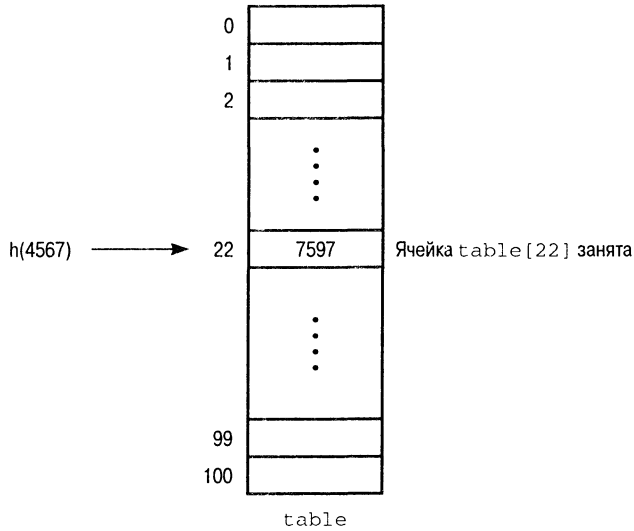


Рис. 12.45. Конфликт

Два подхода к разрешению конфликтов

Существует два общих подхода к разрешению конфликтов. Во-первых, можно найти другую ячейку *внутри* таблицы хэширования и поместить туда новый элемент. Во-вторых, можно изменить структуру таблицы хэширования, так чтобы каждая ячейка `table[i]` могла хранить несколько элементов. Рассмотрим каждую из этих возможностей.

Подход 1: открытая адресация. Если при попытке записи элемента в ячейку выяснилось, что она занята, **зондируется** (probe) другая пустая, или открытая ячейка, в которую можно записать новый элемент. Последовательность проверяемых ячеек называется **зондируемой последовательностью** (probe sequence).

Эта схема называется **открытой адресацией** (open addressing). Разумеется, проблема заключается в том, чтобы выполнить эффективный поиск свободной ячейки. Иными словами, операции `tableDelete` и `tableRetrieve` должны предусматривать эффективное последовательное зондирование, использованное операцией `tableInsert`.

Различие между разными схемами открытой адресации заключается в методе зондирования пустой ячейки. Рассмотрим три таких метода.

Последовательный поиск свободной ячейки, начиная с ячейки хэширования

Линейное зондирование. В этой простой схеме выполняется последовательное зондирование свободных ячеек, начиная с ячейки хэширования. Говоря конкретнее, если ячейка `table[h(searchKey)]` занята, проверяются ячейки `table[h(searchKey)+1]`, `table[h(searchKey)+2]` и т.д., пока на обнаружится свободная ячейка. Обычно при необходимости выполняется *сворачивание* поиска, начиная от последней ячейки таблицы и заканчивая ее первой ячейкой.

Если бы не было операции удаления, реализация операции `tableRetrieve` была бы совсем простой. Нужно было бы только повторить последовательное зондирование, которое выполнила функция `tableInsert`, найти искомый элемент, обнаружить свободную ячейку или просмотреть каждую ячейку таблицы.

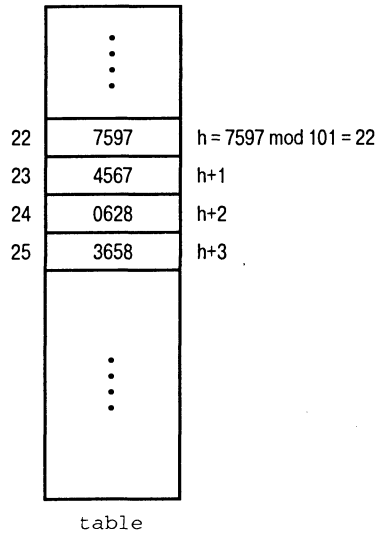


Рис. 12.46. Линейное зондирование с функцией хэширования h(x) = x mod 101

Три состояния ячейки: занятая, пустая и удаленная

Однако удаление элементов все немного усложняет. Сама по себе операция `tableDelete` затруднений не вызывает. Выполняя ее, мы просто находим искомый элемент, как и при осуществлении операции `tableRetrieve`, и удаляем его из ячейки. А что произойдет, если выполнить операцию `tableRetrieve` после удаления элемента? Новые пустые ячейки, созданные операцией `tableDelete` во время последовательного зондирования, могут привести к преждевременному завершению операции `tableRetrieve`, имитируя фиктивный отказ. Эту проблему можно разрешить, если предусмотреть для каждой ячейки три возможных состояния: занята (используется в данный момент), пуста (еще не использовалась) и удалена (была занята, но теперь свободна). Теперь можно модифицировать операцию `tableRetrieve`, так чтобы при последовательном зондировании она искала удаленные ячейки. Аналогично, операцию `tableInsert` нужно модифицировать так, чтобы она выполняла вставку элемента либо в пустую, либо в удаленную ячейку.

Кластеризация может породить проблемы

Одна из проблем, возникающих в схеме линейного зондирования, заключается в том, что элементы таблицы хэширования стремятся образовать **кластер** (cluster). Иными словами, в таблице возникают группы занятых ячеек, расположенных последовательно. Это явление называется **первичной кластеризацией** (primary clusterinn). Кластеры могут располагаться близко друг к другу, сливаясь в один более крупный кластер. Такие кластеры стремятся слиться с другими, образуя новый, еще более крупный кластер ("Деньги — к деньгам.") Следовательно, одна часть таблицы может быть довольно плотной, а другие — разреженными. Первичная кластеризация тормозит процесс линейного зондирования и снижает общую эффективность хэширования.

Квадратичное пробирование. Уточнив схему линейного зондирования, можно избежать первичной кластеризации. Вместо зондирования последовательных ячеек исходной таблицы хэширования `table[h(searchKey)]` можно проверять ячейки `table[h(searchKey)+1`^{2}`]`, `table[h(searchKey)+2`^{2}`]`, `table[h(searchKey)+3`^{2}`]` и

т.д., пока не обнаружится свободная ячейка. Эта схема открытой адресации, называемая квадратичным зондированием (quadratic probing), изображена на рис. 12.47.

К сожалению, если два элемента хэшируются в одну и ту же ячейку, при квадратичном зондировании проверяется одна и та же последовательность ячеек. Это явление, называемое **вторичной кластеризацией** (secondary clustering), тормозит разрешение конфликта. Хотя анализ квадратичного зондирования в настоящее время еще не завершен, оказалось, что вторичная кластеризация не представляет большой опасности.

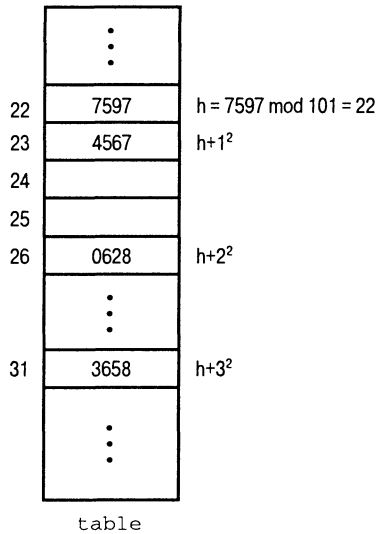


Рис. 12.47. Квадратичное зондирование с функцией хэширования h(x) = x mod 101

Двойное хэширование. Двойное хэширование представляет собой еще один вид открытой адресации. Последовательность ячеек, проверяемых при линейном и квадратичном зондировании, *не зависит от ключа*. Например, при линейном зондировании ячейки таблицы проверяются последовательно, независимо от того, какой ключ хэширования при этом используется. В противоположность этому, при двойном хэшировании последовательность проверяемых ячеек *зависит от значения ключа*. В этой схеме зондирование, как и раньше, проводится в линейном порядке, начиная с ячейки $h_1(key)$. Однако теперь размер шага задается функцией h_2.

При двойном хэшировании последовательность зондирования зависит от адреса хэширования и шага

Функцию h_1 можно выбирать как обычно, а при выборе функции h_2 должны выполняться следующие условия:

$h_2(key) \neq 0$;

$h_2 \neq h_1$.

Принципы выбора функции h_2, задающей размер шага зондирования

Первое условие очевидно — шаг зондирования не может равняться нулю. Второе условие заключается в том, что функции h_1 и h_2 должны быть разными, чтобы избежать кластеризации.

Допустим, что функции h_1 и h_2 являются первичной и вторичной функциями хэширования, определенными следующими формулами:

Первичная и вторичная функции хэширования

$h_1(key) = key \bmod 11$,

$h_2(key) = 7 - (key \bmod 7)$.

Предполагается, что таблица хэширования содержит не более 11 элементов, поэтому влияние этих функций на таблицу хэширования легко проследить. Если *key*=58, функция h_1 хэширует ключ в ячейку 3 (поскольку 58 mod 11 = 3), а функция h_2 вычисляет размер шага зондирования, равный 5 (поскольку 7 - 58 mod 7 = 5). Иными словами, последовательность зондируемых ячеек такова: 3, 8, 2 (сворачивание), 7, 1 (сворачивание), 6, 0, 5, 10, 4, 9. Если *key*=14, функция h_1 хэширует ключ в ячейку 3 (поскольку 14 mod 11 = 3), а функция h_2 вычисляет размер шага зондирования, равный 7 (поскольку 7 - 14 mod 7 = 7). Итак, последовательность зондируемых ячеек выглядит иначе: 3, 10, 6, 2, 9, 5, 1, 8, 4, 1.

При каждом зондировании проверяются *все* ячейки таблицы. Это происходит тогда, когда размер таблицы и размер шага зондирования являются взаимно простыми числами, т.е. их наибольший общий делитель равен 1. Поскольку размер таблицы хэширования, как правило, равен простому числу, при любом размере шага будут просмотрены все ячейки.

На рис. 12.48 показан результат вставки чисел 58, 14 и 91 в пустую таблицу хэширования. Поскольку $h_1(58)$ равен 3, число 58 записывается в ячейку `table[3]`. Затем вычисляется значение $h_1(14)$, которое тоже равно 3. Чтобы избежать конфликта, вычисляется шаг $h_2(14)$=7, и число 14 записывается в ячейку `table[3+7]`, т.е. в ячейку `table[10]`. В заключение вычисляются значения $h_1(91)$=3 и $h_2(91)$=7. Поскольку ячейка `table[3]` занята, проверяется ячейка `table[10]`, которая тоже оказывается занятой. В итоге число 91 оказывается в ячейке `table[(10+7)%11]`, т.е. в ячейке `table[6]`.

Применение нескольких функций хэширования называется **повторным хешированием** (rehashing). Поскольку часто двух функций хэширования оказывается недостаточно, такая схема будет довольно сложной для реализации.

Увеличение размера таблицы хэширования. В любой из схем открытой адресации при полной таблице хэширования увеличивается вероятность конфликтов.

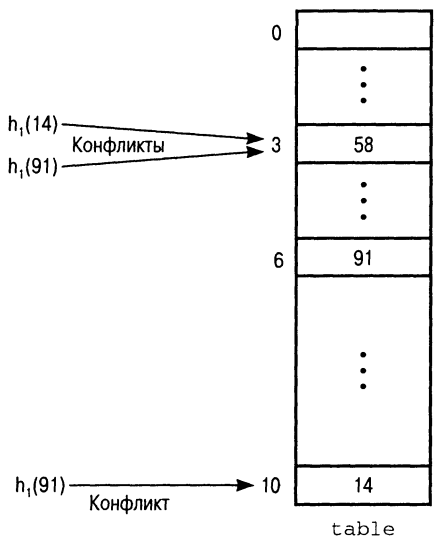


Рис. 12.48. Двойное хэширование при вставке чисел 58, 14 и 91

В этом случае возникает необходимость увеличить таблицу хэширования. Если для хранения таблицы хэширования используется динамический массив, его размер можно увеличивать каждый раз, когда таблица оказывается полностью заполненной.

Во-первых, размер таблицы хэширования нельзя просто увеличить вдвое, как мы делали в предыдущих главах, поскольку он должен оставаться простым числом. Во-вторых, элемент исходной таблицы не следует слепо копировать в новую таблицу хэширования. Если используется функция хэширования *x* mod *tableSize*, изменение значения *tableSize* повлияет и на нее. Следовательно, к каждому элементу старой таблицы хэширования перед копированием в новую таблицу нужно применить новую функцию хэширования.

Каждая ячейка таблицы хэширования может содержать несколько элементов

Подход 2: перестройка таблицы хэширования. Другой способ разрешения конфликтов — изменение структуры таблицы хэширования `table` так, чтобы каждая ячейка таблицы хэширования могла содержать несколько элементов. Опишем два способа изменения таблицы хэширования.

Блоки. Если каждая ячейка таблицы хэширования `table[i]` сама является массивом, называемым **блоком** (bucket), элементы, которые хэшируются в эту ячейку, можно записывать непосредственно в блок. Разумеется, остается невыясненным вопрос, как выбирать размер *B* каждого блока. Если этот размер слишком мал, он лишь отодвинет момент возникновения конфликта, пока в блок не будут хэшированы *B*+1 элемент. Если задать размер *B* достаточно большим, то каждый блок сможет хранить больше элементов, чем потенциально возможно. Это приведет к неэффективным затратам памяти.

Каждый элемент таблицы хэширования представляет собой связанный список

Отдельное связывание. Более эффективным считается представление таблицы хэширования в виде массива связанных списков. В этом методе разрешения конфликтов, называемым **отдельным связыванием** (separate chaining), каждая ячейка `table[i]` хранит указатель на связанный список элементов — **цепочку** (chain), — которым функция хэширования поставила в соответствие ячейку `table[i]`, как показано на рис. 12.49. Рассмотрим классы, которые используются для реализации абстрактной таблицы с помощью таблицы хэширования и отдельного связывания.

```
// ********************************************************
// Заголовочный файл TableH.h абстрактной таблицы.
// Реализация в виде таблицы хэширования.
// Предположение: в каждый момент времени таблица содержит по крайней
//                мере один элемент, имеющий заданный ключ.
// ********************************************************
#include "ChainNode.h"

typedef KeyedItem TableItemType;

class HashTable
{
public:
// Конструкторы и деструктор:
   HashTable();
   HashTable(const HashTable& table);
   ~HashTable();

// Операции над таблицей:
   virtual bool tableIsEmpty() const;
```

```
    virtual int tableGetLength() const;
    virtual void tableInsert(const TableItemType& newItem)
       throw(HashTableException);
    virtual bool tableDelete(KeyType searchKey);
    virtual bool tableRetrieve(KeyType searchKey,
                               TableItemType& tableItem) const;

protected:
    int hashIndex(KeyType searchKey); // Функция хэширования

private:
    enum {HASH_TABLE_SIZE = 101}; // Размер таблицы хэширования
    typedef ptrType HashTableType[HASH_TABLE_SIZE];

    HashTableType table; // Таблица хэширования
    int size; // размер абстрактной таблицы
}; // Конец класса HashTable
// Конец заголовочного файла.

// ********************************************************
// Заголовочный файл KeyedItem.h.
// Является основой для классов, которым необходим
// поисковый ключ.
// ********************************************************
typedef тип-поискового-ключа KeyType;

class KeyedItem
{
public:
    KeyedItem() {};
    KeyedItem(const KeyType& keyValue) : searchKey(keyValue) {}
    KeyType getKey() const // Возвращает поисковый ключ
    {
       return searchKey;
    } // Конец функции getKey

private:
    KeyType searchKey;
}; // Конец класса KeyedItem

// ********************************************************
// Заголовочный файл ChainNode.h.
// Определение узла цепочки для таблицы хэширования.
// ********************************************************
#include "KeyedItem.h"

class ChainNode
{
private:
    ChainNode();
    ChainNode(const KeyedItem & nodeItem,
              ChainNode *nextNode = NULL)
             :item(nodeItem), next(nextNode) {}

    KeyedItem item;
    ChainNode *next;
    friend class HashTable;
}; // Конец класса ChainNode
```

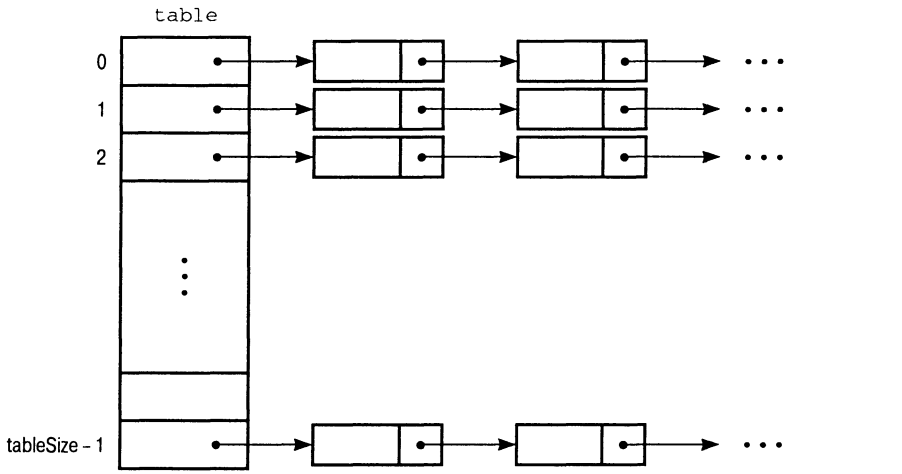


Рис. 12.49. Отдельное связывание

Класс *KeyedItem* можно использовать в качестве базового для элементов, хранящихся в таблице хэширования. Впервые класс *KeyedItem* был представлен в главе 10, где он описывал поля данных для поискового ключа. Этот ключ используется методом *hashIndex* в классе *HashTable* для генерации индекса хэширования.

Вставляя в таблицу новый элемент, мы просто записываем его в начало связанного списка, на который указывает функция хэширования. Рассмотрим псевдокод этой операции.

```
tableInsert(in newItem:TableItemType)

   searchKey = поисковый ключ элемента newItem
   i = hashIndex(searchKey)
   p = указатель на новый узел
   Если память выделить невозможно, генерируется
      исключительная ситуация HashTableException
   p->item = newItem
   p->next = table[i]
   table[i] = p
```

Если нужно извлечь элемент, выполняется поиск в связанном списке, на который указывает функция хэширования. Псевдокод этого алгоритма имеет следующий вид.

```
tableRetrieve(in searchKey:KeyType,
              in tableItem:TableItemType)
              throw TableException

   i = hashIndex(searchKey)
   p = table[i]

   while ( (p != NULL) &&
         (p->item.getKey() != searchKey) )
      p = p->next
      if (p == NULL)
```

```
        Генерируется исключительная ситуация TableException
    else
        tableItem = p->item
```

Алгоритм удаления практически не отличается от извлечения и описан в упражнении 11.

Отдельное связывание успешно предотвращает конфликты

Итак, отдельное связывание — это удачный метод разрешения конфликтов. Он позволяет динамически изменять размер абстрактной таблицы, который может превышать размер таблицы хэширования, поскольку каждый связанный список может быть сколь угодно длинным. Как мы увидим в следующем разделе, длина этих связанных списков влияет на эффективность операций извлечения и удаления элементов.

Эффективность хэширования

Коэффициент загрузки измеряет степень заполненности таблицы хэширования

Коэффициент загрузки (load factor), позволяющий оценить среднюю эффективность хэширования, представляет собой отношение количества элементов, записанных в таблице, к ее максимальному размеру:

$$\alpha = \frac{N}{tableSize}.$$

Коэффициент α оценивает степень заполненности массива `table`. По мере заполнения массива `table` коэффициент α увеличивается, при этом возрастает вероятность конфликтов и время поиска свободной ячейки. Таким образом, при увеличении коэффициента α эффективность хэширования снижается.

В отличие от других реализаций абстрактной таблицы, эффективность хэширования не зависит только от количества элементов *N*. При фиксированном размере таблицы `tableSize` с увеличением количества элементов *N* эффективность хэширования уменьшается. В то же время при фиксированном числе *N*, выбирая значение `tableSize`, эффективность хэширования можно повысить. Таким образом, определяя число `tableSize`, нужно оценить максимально возможное число N и выбрать размер таблицы так, чтобы коэффициент загрузки α был маленьким. Как мы вскоре убедимся, его значение не должно превышать 2/3.

Безуспешный поиск в среднем занимает больше времени, чем успешный

Эффективность хэширования при осуществлении поиска конкретного элемента зависит также от того, насколько успешным был поиск. Безуспешный поиск в среднем занимает больше времени, чем успешный. Анализ, проведенный ниже[7], позволяет сравнить разные способы разрешения конфликтов.

Линейное зондирование. При линейном зондировании среднее количество сравнений, выполняемых при поиске свободной ячейки, приближенно равно $\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$, если поиск был успешным, и $\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]^2$, если поиск был безуспешным.

[7] Кнут Д. Искусство программирования, т.3. Поиск и сортировка.— М.: "Издательский дом Вильямс", 2001.

По мере возрастания количества конфликтов последовательность зондируемых ячеек увеличивается, что приводит к возрастанию времени поиска. Например, если таблица заполнена на две трети (α=2/3), в среднем при безуспешном поиске может потребоваться до пяти сравнений, в то время как для успешного поиска достаточно всего двух. Чтобы эффективность не снижалась, таблица не должна быть слишком плотно заполненной.

Таблица хэширования не должна быть слишком заполненной

Квадратичное зондирование и двойное хэширование. Эффективность квадратичного зондирования и двойного хэширования оценивается величиной $\frac{-\ln(1-\alpha)}{\alpha}$, если поиск был успешным, и $\frac{1}{1-\alpha}$ в противном случае. В среднем оба метода используют меньше сравнений, чем линейное зондирование. Например, для таблицы, заполненной на две трети, при безуспешном поиске понадобится выполнить примерно три сравнения, в то время как при успешном поиске — не более двух. Однако, поскольку все три метода относятся к схемам открытой адресации, при их использовании невозможно предсказать, какое количество вставок и удалений потребуется выполнить. Если размер таблицы хэширования слишком мал, эффективность поиска может снизиться.

Применяя схемы открытой адресации, следует точно оценивать количество операций вставки и удаления элементов

Отдельное связывание. Поскольку вставка нового элемента производится в начало связанного списка, его сложность оценивается величиной O(1). Однако операции `tableRetrieve` и `tableDelete` выполняются не так быстро. Для их осуществления нужно выполнить поиск узла в связанном списке, поэтому было бы хорошо, если бы эти списки были короткими.

Вставка выполняется мгновенно

При отдельном связывании значение `tableSize` задает количество связанных списков, а не максимальное количество элементов. Следовательно, вполне возможно, и даже желательно, чтобы текущее количество элементов таблицы N превышало число `tableSize`. Иными словами, коэффициент загрузки $\alpha = \frac{N}{tableSize}$ может быть больше 1. Поскольку число `tableSize` задает количество связанных списков, число α равно их средней длине.

Некоторые процедуры поиска в таблице хэширования могут оказаться безуспешными, поскольку соответствующий список может быть пустым. Такие процедуры действительно выполняются моментально. Однако, если поиск завершился безрезультатно, когда связанный список был не пустым, операции `tableRetrieve` и `tableDelete` вынуждены просматривать весь список, т.е. в среднем сравнивать α элементов. При успешном поиске также проверяется непустой связанный список. В среднем такой поиск завершается в середине списка. Иными словами, после того как мы определим, что связанный список не пуст, операция поиска пересмотрит $\alpha/2$ элементов.

Итак, эффективность операций извлечения и удаления элементов при отдельном связывании равна $1+\alpha/2$ при успешном поиске и α — при безуспешном.

Средняя эффективность операций извлечения и удаления

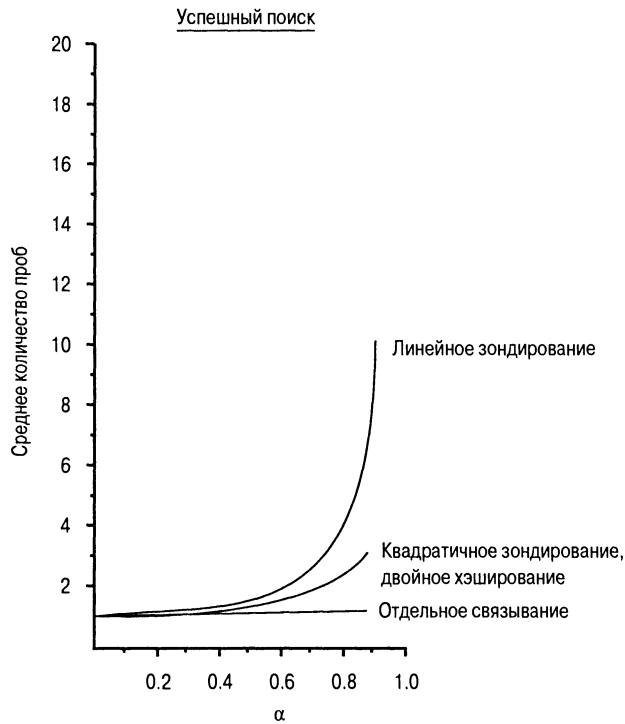
Даже если связанный список невелик, нужно оценить худший вариант. Если сильно недооценить величину `tableSize` или большинству элементов таблицы поставлено в соответствие одна и та же ячейка памяти, количество элементов

связанного списка может оказаться довольно большим. В худшем случае все N элементов таблицы могут оказаться в одном связанном списке!

Легко увидеть, что время, которое занимают операции извлечения и удаления элементов, может быть как весьма малым (если искомый связанный список оказался пуст или невелик), так и довольно большим (если всем элементам таблицы поставлена в соответствие одна ячейка и приходится просматривать связанный список).

Сравнение методов. На рис. 12.50 показана относительная эффективность четырех схем разрешения конфликтов. Если таблица хэширования `table` заполнена наполовину, т.е α=0.5, все методы имеют практически одинаковую эффективность. Если коэффициент загрузки таблицы α близок к единице, наиболее эффективным оказывается отдельное связывание. Означает ли это, что остальными методами можно пренебречь?

Нет. Эти результаты относятся к среднему варианту. Хотя реализации абстрактной таблицы, использующие хэширование, часто оказываются быстрее реализации в виде дерева поиска, в худшем случае оценка меняется на противоположную. Если в приложении допускаются относительно медленный поиск и большой размер `tableSize`, т.е. малое значение коэффициента α, то хэширование оказывается довольно удачным выбором. Однако если время поиска должно быть минимальным, реализация таблицы в виде дерева поиска позволяет по крайней мере получить точные оценки быстродействия в худшем случае.



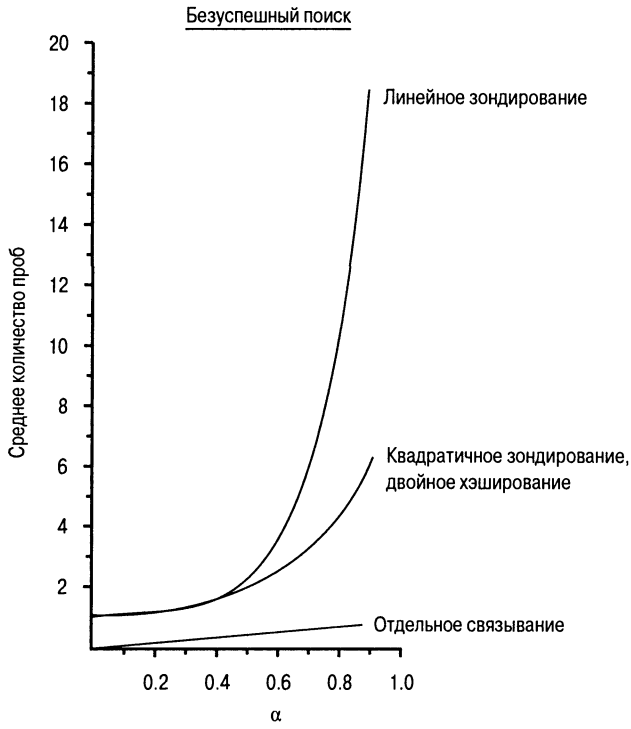


Рис. 12.50. Относительная эффективность четырех методов разрешения конфликтов

Более того, хотя отдельное связывание является наиболее эффективной схемой разрешения конфликтов с точки зрения быстродействия, для хранения указателей связанного списка приходится затрачивать дополнительную память. Если записи, хранящиеся в таблице, невелики, то дополнительные затраты памяти становятся значительными. В таких случаях следует применять более простые схемы разрешения конфликтов. Если записи велики, то объем памяти, требуемый для хранения указателей, становится пренебрежимо малым, и в этих ситуациях метод отдельного связывания вполне пригоден.

Чем отличается хорошая функция хэширования

Завершая введение в хэширование, рассмотрим более детально вопрос выбора функции хэширования, предназначенной для вычисления адреса в конкретном приложении. На эту тему написано много книг, причем большинство из них использует сложный математический аппарат, выходящий за рамки нашего курса. И все же попробуем кратко описать хотя бы основные понятия.

- **Функция хэширования должна легко и быстро вычисляться.** Если схема хэширования должна практически моментально и за постоянное время выполнять операции над таблицей, очевидно, функцию хэширования нужно вычислять как можно быстрее. При вычислении большинства распространенных функций хэширования нужно выполнить только одно деление (по модулю), одно умножение и некую побитовую операцию, применяемую к двоичному представлению поискового ключа. Во всех этих случаях функции вычисляются легко и быстро.

- **Функция хэширования должна равномерно распределять данные по таблице.** Если применяется не идеальная функция хэширования, которую практически невозможно создать, то полностью избежать конфликтов не удается. Например, чтобы достичь максимальной эффективности схемы отдельного связывания, каждый элемент `table[i]` должен содержать приблизительно одинаковое количество элементов. Иными словами, каждая цепочка должна содержать примерно *N/tableSize* элементов (а значит, ни одна цепочка не должна содержать намного больше, чем *N/tableSize* элементов). Чтобы достичь этой цели, функция хэширования должна равномерно распределять поисковые ключи по таблице.

Полностью избежать конфликтов невозможно

С равномерным распределением ключей по таблице связано еще два вопроса.

- **Насколько хорошо функция хэширования распределяет случайные данные?** Если каждый поисковый ключ встречается с одинаковой частотой, насколько равномерно они будут распределены? Рассмотрим следующую схему хэширования девятизначных идентификационных номеров:

  таблица хэширования — `table[0..39]`;

  функция хэширования — $h(x)$ = (две первых цифры числа x) mod 40.

  Допустим, что все идентификационные номера сотрудников равновероятны. Будет ли конкретный идентификационный номер x иметь равную вероятность хэширования в любую из сорока ячеек? Для данной функции хэширования — нет. Ячейка `table[19]` будет соответствовать только номерам, начинающимся с цифр 19, 59 и 99, а ячейка `table[20]` — номерам, начинающимся с цифр 20 и 60. Три разных *префикса* — две первые цифры идентификационного номера — отображаются в любую ячейку от 0 до 19, только если два разных префикса отображаются в ячейки от 20 до 39. Поскольку все идентификационные номера равновероятны — т.е. равновероятны все префиксы от 00 до 99, — вероятность попадания конкретного номера в одну из ячеек от 0 до 19 на 50 % выше, чем вероятность попадания в ячейки от 20 до 39. В результате каждая ячейка массива от 0 до 19 в среднем может содержать на 50 % больше элементов, чем ячейки от 30 до 39.

  Итак, функция хэширования

  $h(x)$ = (две первых цифры числа x) mod 40

  неравномерно распределяет данные по массиву `table[0..39]`. Однако можно показать, что функция хэширования

  $h(x) = x \bmod 40$

  на самом деле равномерно распределяет данные по массиву `table[0..100]`.

- **Насколько хорошо функция хэширования распределяет неслучайные данные?**

  Даже если функция хэширования равномерно распределяет случайные данные, она может неверно работать с неслучайными данными. Независимо от выбора функции хэширования, всегда существует вероятность, что данные будут распределены неравномерно. Хотя способа, который гарантировал бы равномерное распределение данных, не существует, вероятность этого события можно повысить.

В качестве примера рассмотрим следующую схему:

таблица хэширования — `table[0..99]`;

функция хэширования — $h(x)$ = две первых цифры числа x.

Если каждый идентификационный номер равновероятен, функция хэширования h равномерно распределит ключи по массиву. А что если эти номера имеют разную вероятность? Например, компания может присваивать идентификационные номера следующим образом:

10xxxxx (отдел сбыта);

20xxxxx (отдел снабжения);

...

90xxxxx (отдел обработки заказов).

В этих условиях все записи окажутся сконцентрированными только в 9 из 100 ячеек массива. Более того, ячейки, соответствующие отделам, имеющим наибольшие идентификационные номера (например, отделу продаж соответствует ячейка `table[10]`), будут содержать больше элементов, чем ячейки, соответствующие отделам с меньшими номерами. Очевидно, что эта схема не позволяет равномерно распределить данные по таблице. Для каждой разновидности данных нужно провести довольно обширные исследования, прежде чем обнаружится подходящая функция хэширования. Хотя эта тема и выходит за рамки нашей книги, подчеркнем два важных принципа.

Общие требования к функции хэширования

1. При вычислении функции хэширования *поисковый ключ должен использоваться полностью*. Следовательно, безопаснее делить по модулю весь поисковый ключ, а не только первые две цифры.

2. Если функция хэширования использует модульную арифметику, *ее основание должно быть простым числом*. Иными словами, если функция h имеет вид

   $h(x) = x \bmod tableSize$,

   то величина *tableSize* должна быть простым числом. Такой выбор величины *tableSize* предотвращает неприятности, связанные с различными нюансами, характерными для представления чисел (например, исключает поисковые ключи, цифры которых умножаются одна на другую). Хотя в каждом приложении используются свои собственные поисковые ключи, выбор простого числа `tableSize` позволяет легко преодолеть многие трудности, характерные для всех приложений.

Обход таблицы: неэффективная операция при хэшировании

Элементы, хэшированные в ячейки table[i] и table[i+1], не связаны отношением упорядоченности

Во многих приложениях хэширование оказывается наиболее эффективной реализацией абстрактной таблицы. И все же одна важная операция — обход в порядке следования ключей — при хэшировании выполняется плохо. Как указывалось ранее, хорошая функция хэширования равномерно распределяет случайные данные по массиву, поэтому элементы, хэшированные в ячейки `table[i]` и `table[i+1]`, не связаны отношением упорядоченности. В результате, если таблицу нужно обойти в по-

рядке следования ключей, их сначала нужно упорядочить. Если такая сортировка нужна достаточно часто, хэширование начинает проигрывать дереву поиска.

Обход таблицы в порядке следования ключей представляет собой лишь одну из целого класса операций, которые хэширование не способно эффективно поддерживать. Упорядоченность ключей нужна во многих операциях над таблицей. Рассмотрим в качестве примера поиск элемента, имеющего наименьший или наибольший ключ. Если используется реализация таблицы в виде дерева поиска, такие элементы являются крайними левыми и крайними правыми узлами дерева, соответственно. Однако, если применяется хэширование, заранее неизвестно, где искать эти элементы — они могут быть где угодно. К этому же классу относится операция **запроса в диапазоне значений** (range query), при выполнении которой нужно извлечь из таблицы элементы, ключи которых лежат в заданном диапазоне. Например, можно извлечь из таблицы элементы, поисковые ключи которых изменяются от 129 до 175. Эту задачу относительно легко решить с помощью дерева поиска (см. упражнение 3), но при хэшировании у нее нет эффективного решения.

Хэширование и сбалансированное дерево поиска

В общем, если в приложении нужно выполнять упорядоченные операции, следует выбрать дерево поиска. Хотя операции *tableInsert*, *tableRetrieve* и *tableDelete* при хэшировании более эффективны, чем в сбалансированном дереве поиска, во многих случаях разница в быстродействии оказывается незначительной (в то время как преимущество дерева поиска над хэшированием при выполнении упорядоченных операций весьма велико).

Однако все сказанное не относится к ситуации, когда данные хранятся на внешнем запоминающем устройстве. В этом случае разница в быстродействии операции *tableRetrieve* при хэшировании и в дереве поиска становится значительной, как показано в главе 14. В приложениях, использующих внешние запоминающие устройства, операция *tableRetrieve* при хэшировании и реализация упорядоченных операций над деревом поиска редко применяются одновременно.

Одновременное применение нескольких структур данных

Во многих приложениях необходимы структуры данных, предназначенные для решения разных задач. Рассмотрим список клиентов, а именно: очередь записей о клиентах. Предположим, что кроме стандартных операций над очередью *isEmpty*, *dequeue* и *getFront* в приложении нужно часто выводить на печать записи о клиентах. Этот список был бы намного полезнее, если бы клиенты в нем были указаны в алфавитном порядке. Следовательно, нужно предусмотреть операцию *traverse*, посещающую записи о клиентах в определенном порядке.

Этот сценарий порождает интересную проблему. Если записи о клиентах просто записать в очередь, они не будут, как правило, упорядочены по фамилии. Однако если записи хранятся в алфавитном порядке, нарушится принцип FIFO. Очевидно, для решения этой задачи данные нужно организовать двумя разными способами.

Например, можно предусмотреть две независимые структуры данных, одна из которых допускает обход в определенном порядке, а другая поддерживает операции над очередью. На рис. 12.51 показан упорядоченный связанный список записей о клиентах и реализация очереди в виде связанного списка. Структуры, основанные на использовании связанного списка, представляют собой удачный выбор, поскольку в них не требуется оценивать максимально возможное количество записей.

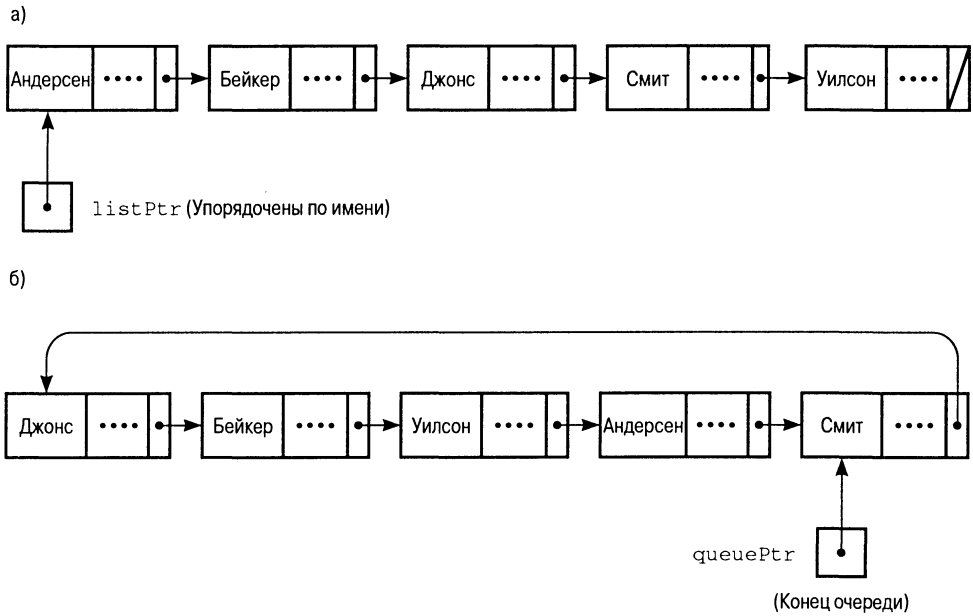


Рис. 12.51. Независимые структуры данных: а) упорядоченный связанный список; б) очередь в виде связанного списка

Независимые структуры данных занимают много памяти

Очевидный недостаток этой схемы связан с тем, что для этих структур придется хранить две копии каждой записи. Кроме того, не все операции над этими структурами реализуются достаточно эффективно.

Операции, при выполнении которых данные просто *извлекаются* — `traverse` и `getFront`, — реализуются легко. Упорядоченный список клиентов можно получить, обходя связанный список, сравнивая текущую запись с первой записью очереди и применяя операцию `getFront`. Однако операции `enqeque` и `dequeue` выполнить намного труднее, поскольку они *модифицируют* данные.

Операция `enqueue` выполняется за два шага.

1. Вставить копию новой записи в конец очереди. Для этого достаточно изменить значения нескольких указателей.
2. Вставить копию новой записи в соответствующую позицию связанного списка. Для этого нужно выполнить обход упорядоченного связанного списка.

Аналогично, `dequeue` выполняется за два шага.

1. Удалить запись из начала очереди, отложив копирование на следующий шаг. Для этого достаточно изменить значения нескольких указателей.
2. Найти в упорядоченном связанном списке запись, только что удаленную из очереди, и удалить ее из списка. Для этого нужно выполнить обход упорядоченного связанного списка.

Для нескольких независимых структур данных не все операции выполняются эффективно

Итак, хотя эта схема эффективно поддерживает выполнение операций `traverse` и `getFront`, операции `enqueue` и `dequeue` вынуждают обходить упорядоченный связанный список (в то время как в очереди они выполняются намного быстрее). Можно ли улучшить эту схему? Например, можно хранить записи в бинарном дереве поис-

ка, а не в упорядоченном связанном списке. Хотя этот подход позволяет намного эффективнее выполнять второй шаг операций *enqueue* и *dequeue*, общее количество работы, которую необходимо выполнить для их осуществления, остается слишком большим.

Если между структурами данных установить связь, возникает совершенно другая схема, которая поддерживает выполнение операции *dequeue* почти так же эффективно, как и очередь. Сначала продемонстрируем эту концепцию на примере упорядоченного связанного списка и очереди, а затем перейдем к более сложным структурам, таким как бинарное дерево поиска.

> Независимые структуры позволяют лучше организовать данные

В структуре данных, изображенной на рис. 12.52, упорядоченный связанный список продолжает хранить записи о клиентах, однако очередь теперь содержит только указатели на эти записи. Иными словами, каждый элемент очереди ссылается на соответствующую запись в упорядоченном списке. Очевидно, что этот способ позволяет значительно сократить расходы памяти, поскольку указатель, как правило, намного меньше, чем сама запись. Как мы вскоре убедимся, эта схема намного повышает эффективность выполнения операции *dequeue*.

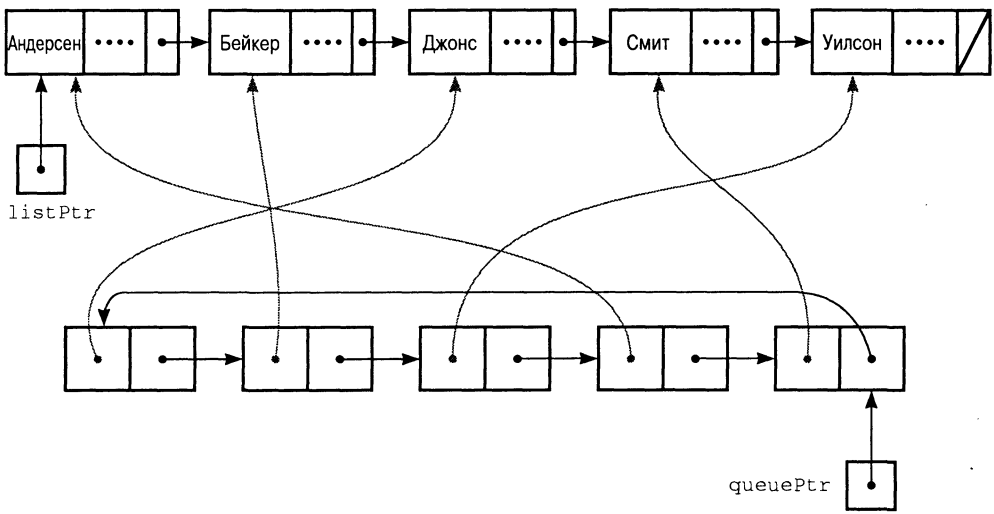


Рис. 12.52. Очередь, ссылающаяся на упорядоченный связанный список

Эффективность операций *traverse*, *getFront* и *enqueue* незначительно отличается от эффективности предыдущей схемы, представленной на рис. 12.51. Операция *traverse* по-прежнему выполняется путем обхода упорядоченного связанного списка. Однако псевдокод операций *getFront* и *enqueue* изменяется.

```
getFront(out queueFront:ItemType)

   Установить указатель p на голову очереди
      (указатель p ссылается на узел упорядоченного
      связанного списка, содержащий запись о клиенте,
      стоящем в начале очереди)

   queueFront = элемент узла, на который ссылается указатель p
```

```
enqueue(in newItem:ItemType)

    Найти позицию для элемента newItem в упорядоченном
       связанном списке
    Вставить в эту позицию узел, содержащий элемент newItem
    Вставить в конец очереди указатель на новый узел
```

Реальное преимущество этой схемы проявляется при выполнении операции *dequeue*.

```
dequeue()

    Удалить элемент из начала очереди и сохранить его значение
       p (указатель p ссылается на узел, содержащий удаленную
       запись о клиенте)

    Удалить из упорядоченного связанного списка узел,
      на который ссылается указатель p
```

Поскольку голова очереди всегда содержит указатель на удаляемую запись *R*, нет необходимости начинать поиск элемента в упорядоченном связанном списке. У нас есть указатель на эту запись, и все, что нужно сделать — просто удалить ее.

Применяйте дважды связанный список

Однако есть одна существенная трудность. Поскольку мы можем перейти к записи *R* непосредственно, не выполняя обхода связанного списка с самого начала, у нас нет указателя на запись, предшествующую записи *R*! Напомним, что для удаления записи *R*, нам нужно изменить указатель в предыдущей записи. Единственный способ найти предыдущую запись — выполнить обход списка с начала, т.е. именно то, от чего мы стремились избавиться! Но эту проблему можно решить, заменив односвязный список, изображенный на рис. 12.52, дважды связанным, как показано на рис. 12.53. (См. задание 8.)

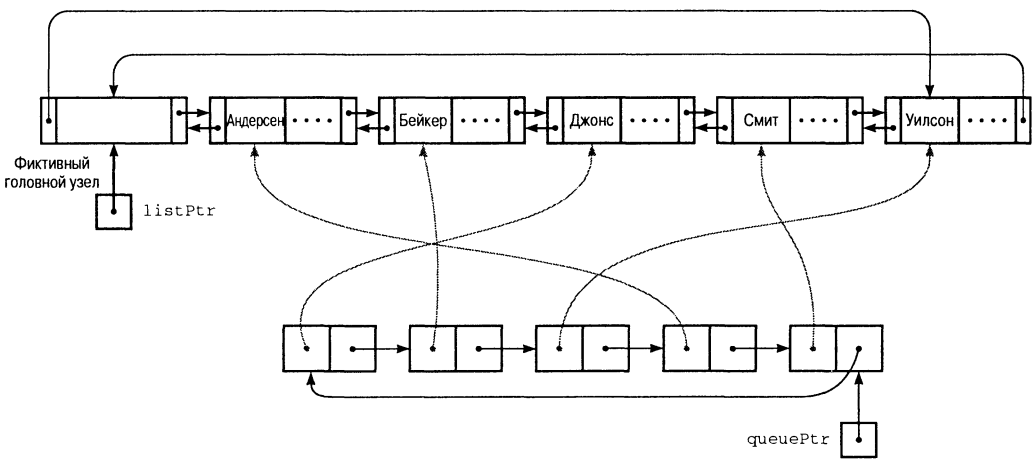


Рис. 12.53. Очередь, ссылающаяся на дважды связанный список

Итак, мы ознакомились с достаточно эффективной схемой реализации операций над очередью в комбинации с обходом упорядоченного связанного списка. Единственная операция, эффективность которой можно было бы значительно повысить, — операция *enqueue*, — поскольку в ней по-прежнему предполагается обход связанного списка в поисках подходящего места для вставки новой записи.

Линейный связанный список был выбран лишь для упрощения объяснений. Еще более эффективные схемы можно получить, объединяя очередь с бинарным деревом поиска, а не со связанным списком. Такие структуры данных позволяют выполнять операцию *enqueue* за логарифмическое время, если дерево остается сбалансированным. Однако для эффективного выполнения операции *dequeue* нужен дважды связанный список. Иными словами, каждый узел дерева должен ссылаться на своего родителя, так чтобы можно было легко удалить узел, на который ссылается очередь. Эта структура данных показана на рис. 12.54. Ее реализация довольно сложна и является предметом задания 9.

Как правило, для одного и того же множества данных можно применить несколько структур одновременно. Более подробно эта концепция обсуждается в главе 14 в контексте индексации внешнего запоминающего устройства.

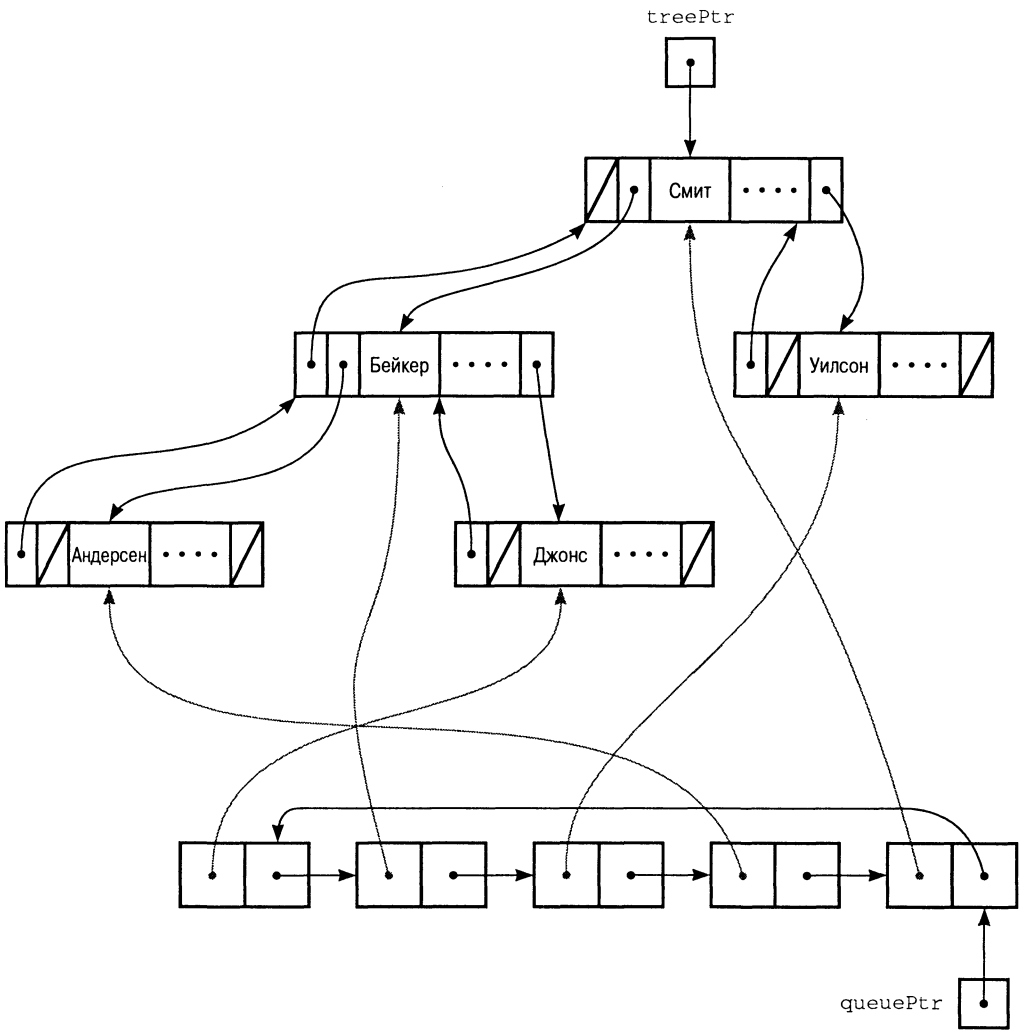


Рис. 12.54. Очередь, ссылающаяся на дважды связанное бинарное дерево поиска

Резюме

1. 2-3 и 2-3-4 деревья представляют собой варианты бинарного дерева поиска. Внутренние узлы 2-3 дерева могут иметь два или три дочерних узла. Внутренние узлы 2-3-4 дерева могут иметь два, три или четыре дочерних узла. Варьирование количества узлов облегчает поддержку баланса дерева при выполнении операции вставки и удаления узлов.
2. Алгоритм вставки и удаления узлов 2-3-4 дерева выполняется за один проход от корня до листа. Следовательно, они более эффективны, чем соответствующие алгоритмы для 2-3 дерева.
3. Красно-черное дерево является бинарным представлением 2-3-4 дерева, которое занимает меньше памяти. Операции вставки и удаления узлов красно-черного дерева выполняются более эффективно, чем соответствующие операции над 2-3-4 деревом.
4. AVL-дерево — это бинарное дерево поиска, гарантирующее сохранение баланса. Если дерево начинает терять баланс, при вставках и удалении его узлов выполняются вращения.
5. Хэширование — это реализация таблицы, при которой вычисляется адрес элемента. Это позволяет не выполнять процедуру поиска. Хэширование гарантирует эффективную реализацию операций извлечения, вставки и удаления.
6. Функция хэширования должна очень легко вычисляться — за несколько операций — и равномерно распределять данные по таблице хэширования.
7. Если два разных поисковых ключа хэшируются в одну и ту же ячейку массива, возникает конфликт. Для разрешения конфликтов применяются зондирование и связывание.
8. Хэширование неэффективно поддерживает операции, в которых данные должны быть упорядочены, например, обход таблицы в заданном порядке.
9. Если обход таблицы выполняется редко, максимальное количество элементов известно заранее и объем памяти не ограничен, хэширование эффективнее реализует таблицу, чем бинарное дерево поиска. Однако деревья более динамичны и не нуждаются в предварительной оценке максимального количества элементов таблицы.
10. Одно и то же множество данных можно представить в виде нескольких структур одновременно. Например, записи можно хранить в упорядоченном связанном списке, а принцип FIFO соблюдать с помощью очереди, содержащей указатели на эти записи.

Предупреждения

1. Несмотря на то что деревья поиска, узлы которых имеют больше двух дочерних узлов, короче бинарных деревьев поиска, поиск не становится эффективнее. В этом случае возрастает количество сравнений, которые необходимо выполнить для каждого узла, чтобы определить, в каком поддереве продолжать поиск.
2. Как правило, схема хэширования должна предусматривать способы разрешения конфликтов. Выбирайте функции хэширования, которые минимизируют количество конфликтов. Следует избегать функций хэширования, которые неравномерно распределяют данные по массиву.

3. Чтобы повысить производительность хэширования, нужно либо изменить функцию хэширования, либо увеличить размер таблицы. Не используйте сложные схемы разрешения конфликтов.
4. Если в таблице часто выполняются операции над упорядоченными данными, хэширование становится неэффективным. Например, если таблица часто просматривается в определенном порядке или нужно найти элемент с максимальным ключом, хэширование применять не следует.

Вопросы для самопроверки

1. Какое дерево получится в результате последовательных вставок элементов 5, 40, 10, 20, 15 и 30 в пустое 2-3 дерево? Обратите внимание, что вставка одного элемента в пустое 2-3 дерево создает отдельный узел, содержащий вставленный элемент.
2. Выполните следующие задания.
   2.1. Какое дерево получится в результате удаления элемента 10 из 2-3 дерева, созданного в упражнении 1?
   2.2. Какое дерево получится в результате вставки элементов 3 и 4 в 2-3 дерево, созданное в упражнении 1?
3. Выполните следующие задания.
   3.1. Повторите упражнение 1 для 2-3-4 дерева.
   3.2. Вставьте элементы 3 и 4 в дерево, созданное в упражнении 3.1.
4. Какое красно-черное дерево представляет 2-3-4 дерево, изображенное на рис. 12.27, *а*.
5. Если реализация абстрактной таблицы предусматривает лишь одну операцию извлечения, как, например, приложение, описанное в сценарии Б из главы 11, какое дерево позволит достичь наибольшей эффективности: сбалансированное дерево поиска, 2-3 дерево, 2-3-4 дерево или красно-черное дерево?
6. Почему для хранения узлов красно-черного дерева требуется меньше памяти, чем для узлов 2-3-4 дерева?
7. Напишите псевдокод операции `tableDelete`, если для реализации таблицы хэширования применяется линейное зондирование.
8. Какая последовательность зондируемых ячеек возникнет в результате следующего двойного хэширования:

   $h_1(key) = key \bmod 11$, $h_2(key) = 7 - (key \bmod 7)$ и $key = 19$.
9. Допустим $h(x)=x \bmod 7$ и для разрешения конфликтов применяется отдельное связывание. Как будет выглядеть таблица хэширования после следующей последовательности вставок: 8, 10, 24, 15, 32, 17? Предполагается, что каждый элемент таблицы содержит только один поисковый ключ.

Упражнения

1. Примените последовательность операций, указанную ниже, к пустой абстрактной таблице `table`, которая реализована в следующих вариантах.
   1.1. Бинарное дерево поиска.
   1.2. 2-3 дерево.

1.3. 2-3-4 дерево.

1.4. Красно-черное дерево.

1.5. AVL-дерево.

2. Изобразите деревья, полученные в результате выполнения каждой из следующих операций.

```
table.tableInsert(10)
table.tableInsert(100)
table.tableInsert(30)
table.tableInsert(80)
table.tableInsert(50)
table.tableDelete(10)
table.tableInsert(60)
table.tableInsert(70)
table.tableInsert(40)
table.tableDelete(80)
table.tableInsert(90)
table.tableInsert(20)
table.tableDelete(30)
table.tableDelete(70)
```

3. Какие преимущества имеет 2-3 дерево над бинарным деревом поиска при реализации абстрактной таблицы? Почему невозможно постоянно поддерживать баланс в бинарном дереве поиска?

4. Напишите псевдокод функции, выполняющей запрос по диапазону значений для 2-3 дерева. Функция должна посетить каждый узел, ключ которого лежит в заданном диапазоне (например, от 100 до 1000).

5. Предположим, что дерево, изображенное на рис. 12.5, *б*, является 2-3-4 деревом. Вставьте в него элементы 39, 38, 37, 36, 35, 34, 33 и 32. Какое 2-3-4 дерево получилось в результате?

6. Напишите псевдокод операций вставки, удаления, извлечения и обхода 2-3-4 дерева.

7. На рис. 12.33 показано красно-черное дерево, представляющее 2-3-4 дерево, изображенное на рис. 12.20. Нарисуйте другое красно-черное дерево, представляющее то же самое 2-3-4 дерево.

8. Какое 2-3-4 дерево представляет красно-черное дерево, изображенное на рис. 12.55?

9. Напишите псевдокод операций вставки, удаления, извлечения и обхода красно-черного дерева.

10. Напишите на языке C++ функцию, преобразующую 2-3-4 дерево в красно-черное дерево.

11. Напишите псевдокод операций `tableInsert`, `tableDelete` и `tableRetrieve`, если реализация таблицы использует хэширование и линейное зондирование.

12. Напишите псевдокод операции `tableDelete`, если реализация таблицы использует хэширование и отдельное связывание.

Успех реализации абстрактной таблицы с помощью хэширования связан с выбором хорошей функции хэширования. Эта функция должна легко вычисляться и равномерно распределять данные по массиву. Оцените приведенные ниже функции хэширования. Какие данные будут распределены по одинаковым ячейкам?

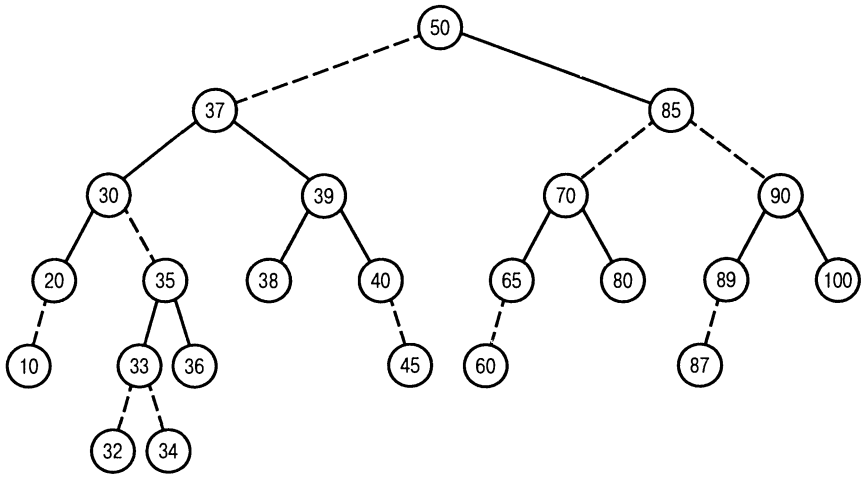


Рис. 12.55. Красно-черное дерево из упражнения 7

12.1. Размер таблицы хэширования равен 2048. Ключами поиска являются слова на английском языке. Функция хэширования задана следующим образом:

12.2. *h*(*key*) = (*сумма позиций, которые занимают в алфавите буквы ключа*) mod 2048.

12.3. Размер таблицы хэширования равен 2048. Ключами поиска являются строки, начинающиеся с буквы. Функция хэширования задана следующим образом:

12.4. *h*(*key*) = (*позиция, которую занимает в алфавите первая буква ключа*) mod 2048.

12.5. Таким образом, слово "BUT" будет преобразовано в число 2. Насколько эффективна эта функция, если строки являются случайными? Что изменится, если строки будут словами английского языка?

12.6. Размер таблицы хэширования равен 10000. Ключами поиска являются целые числа, изменяющиеся от 0 до 9999. Функция хэширования задана следующим образом:

12.7. *h*(*key*) = (*key* * *random*) округленное до целого числа,

12.8. где число *random* получено с помощью генератора случайных чисел, возвращающего значения от 0 до 1.

12.9. Размер таблицы хэширования равен 10000 (*HASH_TABLE_SIZE* равен 1000). Ключами поиска являются целые числа, изменяющиеся от 0 до 9999. Функция хэширования задана следующим фрагментом программы на языке C++:

```
int hashIndex(int x)
{
   for (int i = 1; i <= 1000000; ++i)
      x = (x * x) % HASH_TABLE_SIZE;
   return x;
} // Конец функции hashIndex
```

Задания по программированию

1. Реализуйте абстрактную таблицу в виде 2-3-4 дерева.
2. Реализуйте абстрактную таблицу в виде 2-3 дерева.
3. Реализуйте абстрактную таблицу в виде красно-черного дерева.
4. В упражнении 5 из главы 11 описана таблица символов, с помощью которой компилятор отслеживает идентификаторы программы. Напишите реализацию этой таблицы с помощью хэширования. Используйте функцию хэширования $h(x) = x \bmod tableSize$ и алгоритм преобразования переменной в целое число x, основанный на правиле Горнера. Для разрешения конфликтов примените отдельное связывание.

   Возрастет ли время выполнения операции вставки, если добавлять в таблицу только те элементы, которых в ней еще нет?
5. Повторите упражнение 4 при следующих условиях.

   5.1. Для разрешения конфликтов используется линейное зондирование.

   5.2. Для разрешения конфликтов используется двойное хэширование.

   5.3. Для разрешения конфликтов используется квадратичное зондирование.
6. Повторите упражнение 4, записывая таблицу в динамический массив. Если таблица хэширования заполняется больше, чем наполовину, увеличьте ее размер до ближайшего простого числа, превышающего число 2 * *tableSize*.
7. Повторите упражнение 4, экспериментируя с разными видами связывания. Например, вместо связанного списка примените бинарное дерево поиска или 2-3-4 дерево.
8. Реализуйте операции над абстрактной очередью, а также операцию упорядоченного обхода очереди, ссылающейся на дважды связанный список, показанный на рис. 12.53.
9. Реализуйте операции над абстрактной очередью, а также операцию упорядоченного обхода очереди, ссылающейся на дважды связанный список, показанный на рис. 12.54. Вам понадобятся операции вставки и удаления элементов бинарного дерева поиска, содержащие указатели на своих родителей, как указано в упражнении 31 из главы 10.
10. Повторите упражнение 4 из главы 10, используя АТД "Записная книжка". Для реализации таблицы примените сбалансированное бинарное дерево.
11. Реализуйте с помощью хэширования таблицу символов, описанную в упражнении 5 из главы 11.

Графы

В этой главе ...

Терминология
Графы как абстрактные типы данных
  Реализация графов
Алгоритмы обхода графа
  Поиск в глубину
  Поиск в ширину
Применения графов
  Топологическая сортировка
  Остовные деревья
  Минимальные остовные деревья
  Кратчайшие пути
  Простые цепи
  Некоторые трудные задачи
Резюме
Предупреждения
Вопросы для самопроверки
Упражнения
Задания по программированию

Введение. Графы — это важное математическое понятие, которое широко используется не только в компьютерных науках, но и в других областях знаний. Граф можно считать и математическим понятием, и структурой данных, и абстрактным типом данных. Эта глава представляет собой введение в теорию графов и позволяет изучать их с разных точек зрения. В ней также описаны основные операции над графами и использование графов в компьютерных науках.

Терминология

Вы, несомненно, уже знакомы с графами: графики, а также обычные и круговые диаграммы используются очень широко. На рис. 13.1 показан простой линейный граф: множество точек, соединенных линиями. Очевидно, графы позволяют иллюстрировать данные. Однако, кроме этого, графы могут выражать отношения между их элементами. Именно это свойство графов и будет нас интересовать.

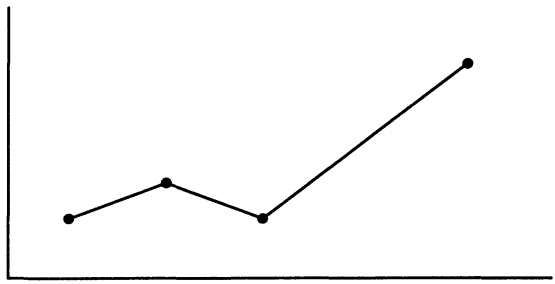


Рис. 13.1. Обычный линейный граф

G=(V, E), т.е. граф — это множество вершин ребер

Граф (graph) G состоит из двух множеств: множества **вершин** V и множества **ребер** E, соединяющих эти вершины. Например, карта студенческого городка, представленная на рис. 13.2, *а*, представляет собой граф, вершины которого изображают здания, а ребра — дорожки между ними. Такое определение графа носит более абстрактный характер, чем определение линейного графа. Фактически линейный граф, состоящий из точек и отрезков, представляет собой разновидность абстрактного графа.

Смежные вершины соединяются ребрами

Подграф (subgraph) состоит из подмножества вершин и ребер какого-либо графа. На рис. 13.2, *б* показан подграф графа, изображенного на рис. 13.2, *а*. Две вершины графа называются **смежными** (adjacent), если они соединены ребром. На рис. 13.2, *б* библиотека и штаб-квартира студенческого профсоюза являются смежными вершинами.

Путь между вершинами состоит из последовательности ребер

Путь между вершинами состоит из последовательности ребер, начинающихся в одной вершине и заканчивающихся в другой. Например, на рис. 13.2, *а* существует путь из общежития в библиотеку, а затем в штаб-квартиру студенческого профсоюза, а оттуда — обратно в библиотеку.

Простой путь проходит через вершину только один раз

Путь может проходить через одну и ту же вершину несколько раз, но существуют пути, которые этим свойством не обладают — они называются **простыми** (simple path). Путь из общежития через библиотеку в штаб-квартиру студенческого профсоюза является простым.

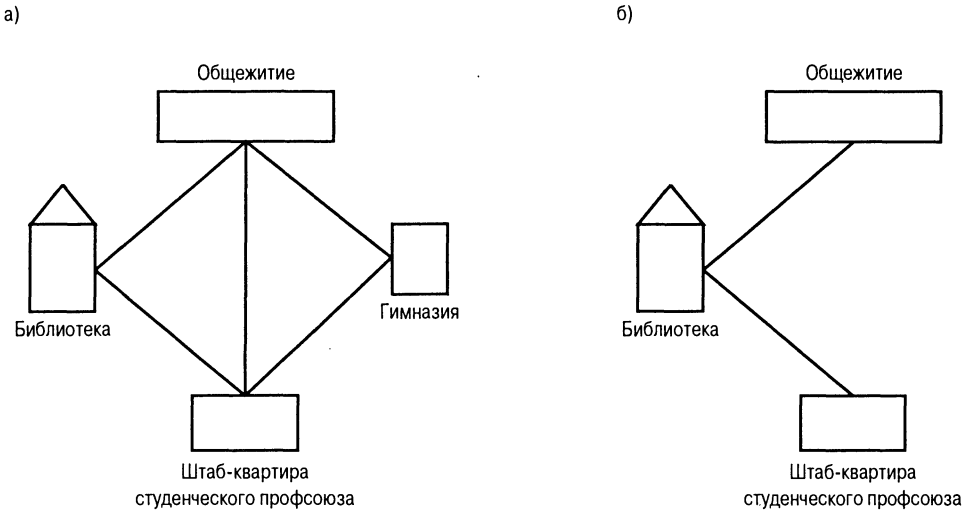


Рис. 13.2. Примеры графов: а) карта студенческого городка; б) подграф

В графе, изображенном на рис. 13.2, *а*, путь Библиотека – Штаб-квартира студенческого профсоюза – Гимназия – Общежитие – Библиотека является простым циклом.

> Цикл — это путь, который начинается и заканчивается в одной и той же вершине

Граф называется **связным** (connected) , если существует путь между любыми двумя вершинами. Иными словами, в связном графе можно выйти из любой вершины и, следуя по некоторому пути, оказаться в любой другой вершине. Пример связанного графа показан на рис. 13.2, *а*. Обратите внимание, что в связном графе любые две вершины не обязательно соединяются ребрами. Пример **несвязного** (disconnected) графа приведен на рис. 13.3, *а*.

> В связном графе существует путь между любыми двумя разными вершинами

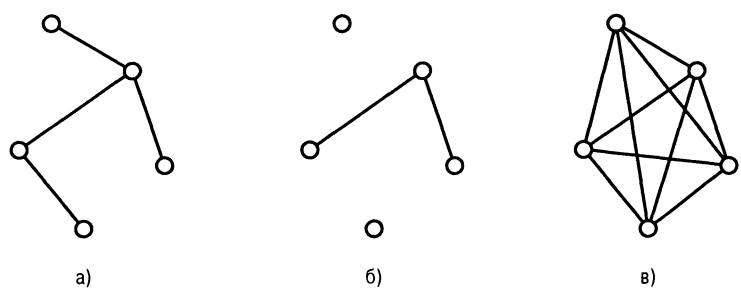


Рис. 13.3. Разновидности графов: а) связный; б) несвязный; в) совершенный

Граф называется **совершенным** (complete) , если каждая пара разных вершин соединена ребром. Совершенный граф показан на рис. 13.3, *в*. Очевидно, что совершенный граф является связным, однако обратное утверждение неверно. Обратите внимание на граф, изображенный на рис. 13.3, *а*, который является связным, но не совершенным.

> Совершенный граф является связным

Поскольку граф является *множеством* ребер, между двумя вершинами нельзя провести несколько ребер. Однако в **мультиграфе** (multigraph) , показанном на рис. 13.4, *а*, это допускается. Кроме того, в графе не существуют ребра, начинающиеся и заканчивающиеся в одной и той же вершине. Пример такого ребра, называющегося **петлей** (loop) , показан на рис. 13.4, *б*.

Мультиграф содержит множественные ребра и поэтому не является графом

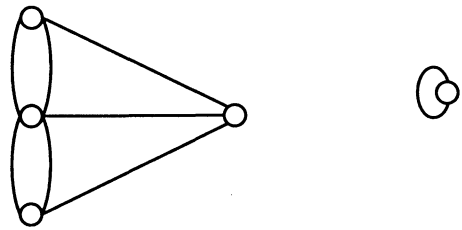


Рис. 13.4. Свойства мультиграфа: а) мультиграф не является графом; б) петли не допускаются

Ребра графа можно размечать. Если метками ребер являются числа, граф называется **взвешенным** (weighted graph) . На рис. 13.5 показан пример взвешенного графа, в котором числовые метки ребер означают расстояния между городами.

Ребра взвешенного графа имеют числовые метки

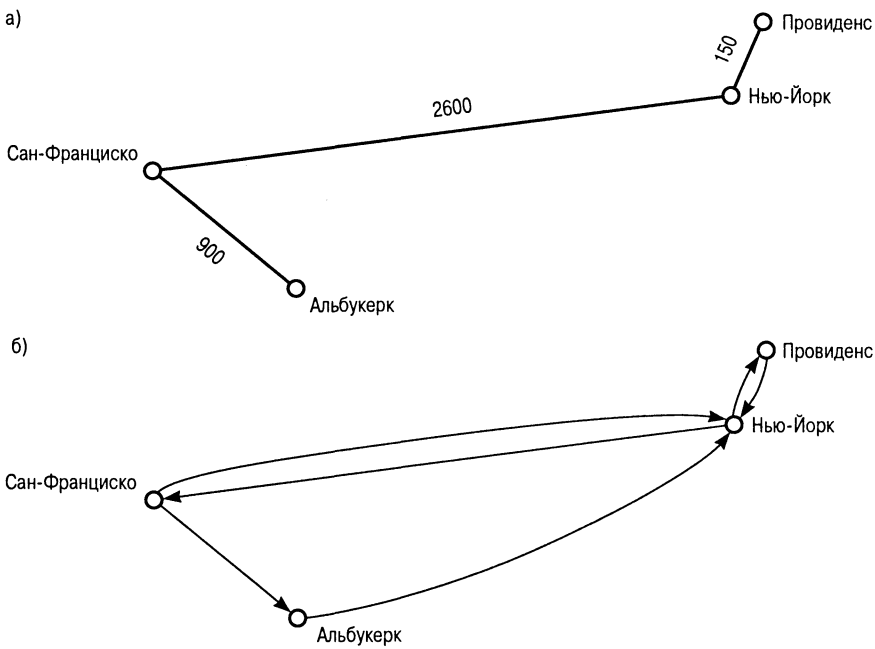


Рис. 13.5. Разновидности графов: а) взвешенный граф; б) ориентированный граф

Все графы, рассмотренные выше, были **неориентированными** (undirected) , поскольку их ребра не имели направления. Иными словами, по ребрам неориентированного графа можно перемещаться в обе стороны. В противоположность ему, каждое ребро **ориентированного графа** (directed graph) , или **орграфа** (digraph) имеет направление. Такие ребра называются **ориентированными** (directed edge) . В то время как в неориентированном графе любые две вершины соединяются только одним ребром, в ориентированном графе между такими вершинами можно провести два ребра, имеющих разное направление. В качестве примера на рис. 13.5, *б* изображена карта авиарейсов, являющаяся ориентированным графом. Между городами Провиденс и Нью-Йорк самолеты летают в обоих направлениях, но из Сан-Франциско в Альбукерк прилететь можно, а вернуться обратно — нельзя. Неориентированный граф можно преобразовать в ориентированный, заменив каждое ребро двумя новыми ребрами, имеющими противоположное направление.

Каждое ребро в ориентированном графе имеет направление

Определения, сформулированные выше для неориентированного графа, можно применить и к ориентированному графу, учтя направления ребер. Например, ориентированным путем (directed path) называется последовательность ориентированных ребер между двумя вершинами, как, например, на рис. 13.5, *б*, ориентированный путь начинается в Провиденсе, проходит через Нью-Йорк и заканчивается в Сан-Франциско. Однако определение смежных вершин в орграфе не так очевидно. Если вершина x соединяется с вершиной y ориентированным ребром, то вершина y называется смежной с вершиной x. (Соответственно, вершина y называется преемником вершины x (successor), а вершина x — предшественником вершины y.) Отсюда не следует, что вершина x является смежной по отношению к вершине y. Итак, на рис. 13.5, *б* Альбукерк является смежной вершиной с Сан-Франциско, но Сан-Франциско не является смежной вершиной по отношению к Альбукерку.

В ориентированном графе вершина y является смежной по отношению к вершине x, если существует ориентированное ребро из вершины x в вершину y

Графы как абстрактные типы данных

Графы можно рассматривать как абстрактные типы данных. Операции вставки и удаления вершин и ребер графа несколько отличаются от аналогичных операций над другими абстрактными типами данных. Абстрактный граф можно определять так, чтобы его вершины содержали или не содержали какие-либо значения. Графы, вершины которых не содержат никаких значений, представляют лишь отношения между вершинами. Такие графы встречаются достаточно часто, поскольку во многих ситуациях не нужно хранить в вершинах какие-либо значения. Однако в абстрактном графе, определение которого будет дано ниже, вершины содержат значения.

ОСНОВНЫЕ ПОНЯТИЯ

Операции над абстрактным графом

1. Создать пустой граф.
2. Уничтожить граф.
3. Определить, пуст ли граф.
4. Определить количество вершин в графе.
5. Определить количество ребер в графе.

6. Определить, существует ли ребро, соединяющее два заданных ребра.
7. Вставить вершину в граф. Поисковые ключи, хранящиеся в вершинах, должны отличаться от ключа новой вершины.
8. Вставить ребро, соединяющее две заданные вершины графа.
9. Удалить из графа указанную вершину, а также все ребра, соединяющие ее с другими вершинами.

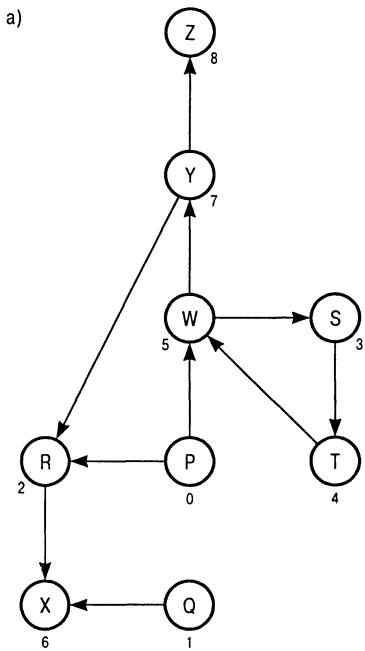
Возможны разные варианты абстрактного графа. Например, если граф является направленным, слово "ребра" можно заменить словосочетанием "ориентированные ребра". Кроме того, в список операций над абстрактным графом можно добавить операции обхода.

Реализация графов

Матрица смежности

Матрица смежности и список смежности представляют собой два наиболее распространенных способа реализации графов. **Матрица смежности** (adjacent matrix) графа, состоящего из n вершин, пронумерованных числами 0, 1,..., n-1, является массивом размера $n \times n$, в котором элемент `matrix[i][j]` равен 1 (`true`), если существует ребро, выходящее из вершины i в вершину j, и 0 (`false`) — в противном случае. Пример ориентированного графа и соответствующей матрицы смежности показан на рис. 13.6. Обратите внимание, что диагональные элементы этой матрицы `matrix[i][i]` равны 0, хотя иногда бывает полезно задать на диагонали единицу. Выбираемые значения элементов матрицы смежности зависят от конкретного приложения.

а)

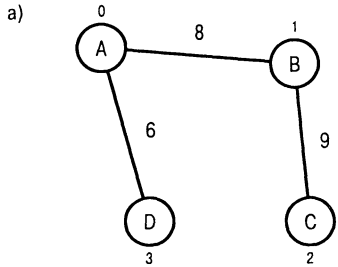


б)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Рис. 13.6. Пример ориентированного графа: а) ориентированный граф; б) его матрица смежности

Если граф является взвешенным, в ячейку `matrix[i][j]` можно записать вес ребра, выходящего из вершины i в вершину j, а не просто число 1. Если в графе нет ребра, соединяющего вершину i и вершину j, в ячейку `matrix[i][j]` записывается символ ∞. Соответствующий пример взвешенного неориентированного графа и его матрицы смежности приведен на рис. 13.7. Обратите внимание, что матрица смежности неориентированного графа является симметричной. Иными словами, элементы `matrix[i][j]` и `matrix[j][i]` равны между собой.



б)

		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Рис. 13.7. Пример взвешенного неориентированного графа: а) взвешенный неориентированный граф; б) его матрица смежности

Вершины могут содержать значения

В нашем определении матрицы смежности не упоминаются значения, которые могут храниться в вершинах. Если с каждой вершиной связано какое-то значение, нужно применить второй массив, `values`, в котором записаны n значений, хранящихся в вершинах. Массив `values` является одномерным, а элемент `values[i]` представляет собой значение, хранящееся в вершине i.

Список смежности

Список смежности (adjacent list) графа, состоящего из n вершин, пронумерованных числами 0, 1,..., n-1, состоит из n связанных списков. В i-м связанном списке содержится узел для вершины j тогда и только тогда, когда граф содержит ребро, соединяющее вершину i и вершину j. Этот узел может хранить значение, приписанное к вершине j. Если вершина не содержит никаких значений, в узле можно записать другую информацию, позволяющую идентифицировать вершину графа. Ориентированный граф и его список смежности показаны на рис. 13.8. Легко увидеть, что вершина 0(*P*) соединена ребрами с вершинами 2® и 5(*W*). Таким образом, первый связанный список содержит узлы, в которых записаны буквы *R* и *W*.

На рис. 13.9 приведен еще один пример взвешенного неориентированного графа и его списка смежности, в котором каждая вершина имеет два ребра с противоположной ориентацией. Итак, ребро, соединяющее вершины *A* и *B* на рис. 13.9, *а*, представлено в виде двух ребер, выходящих из *A* в *B*, и наоборот, как показано на рис. 13.9, *б*. Граф, изображенный на рис. 13.9, *а*, является взвешенным. Веса ребер можно записать в узлы списка смежности, представленного на рис. 13.9, *б*.

Какая из этих реализаций графа лучше — матрица смежности или список смежности? Ответ зависит от конкретного приложения, в котором используется граф.

Две операции над графами

Например, в приложениях часто выполняются две операции над графами.

1. Определить, существует ли ребро, соединяющее вершину i с вершиной j.
2. Найти все вершины, смежные с заданной вершиной i.

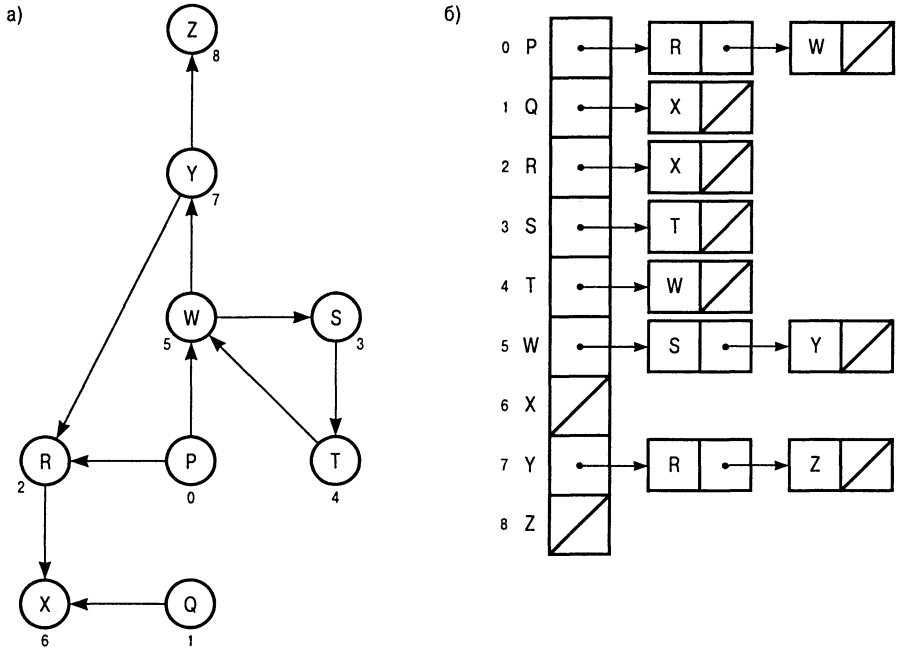


Рис. 13.8. Пример взвешенного ориентированного графа: а) взвешенный неориентированный граф; б) его список смежности

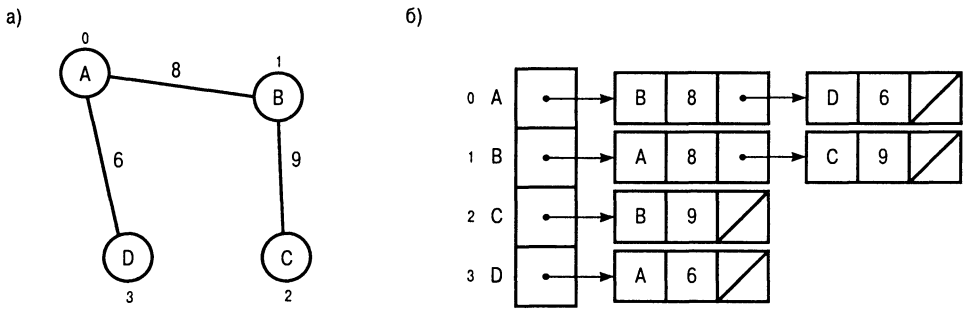


Рис. 13.9. Пример взвешенного неориентированного графа, имеющего ребра противоположной ориентации: а) взвешенный неориентированный граф; б) его список смежности

Матрица смежности позволяет эффективнее выполнять первую операцию

Матрица смежности позволяет эффективнее выполнять первую операцию, чем список смежности. Чтобы определить, существует ли ребро, соединяющее вершину *i* с вершиной *j*, достаточно проверить значение элемента `matrix[i][j]`. Если бы использовался список смежности, пришлось бы пройти *i*-й список, чтобы определить, содержит ли он искомый узел.

Однако если граф реализуется с помощью списка смежности, вторая операция выполняется эффективнее. Чтобы найти все вершины, смежные с заданной вершиной i, в матрице смежности пришлось бы просмотреть всю i-ю строку. В списке смежности для этого достаточно лишь пройти по i-му связанному списку. Если граф содержит n вершин, i-я матрица смежности всегда содержит n элементов, в то время как количество узлов i-го связанного списка равно количеству вершин, смежных с вершиной i. Обычно это число намного меньше n.

Список смежности позволяет эффективнее выполнять вторую операцию

Рассмотрим теперь, какие требования к памяти выдвигают описанные реализации. На первый взгляд, матрица смежности занимает меньше памяти, чем список смежности, поскольку каждый элемент матрицы является целым числом, а узел связанного списка содержит значение, идентифицирующее вершину и указатель. Одна матрица смежности всегда состоит из n^2 элементов, в то время как количество узлов в списке смежности равно количеству узлов ориентированного графа или удвоенному количеству узлов неориентированного графа. Даже с учетом того, что список смежности содержит n указателей, часто он занимает меньше памяти, чем матрица смежности.

Список смежности часто занимает меньше памяти, чем матрица смежности

Итак, выбирая реализацию графа для конкретного приложения, нужно определить, какие операции выполняются чаще и какое количество узлов может содержать граф. Например, в главе 6 была описана задача об авиарейсах, в которой требовалось определить, можно ли добраться из одного города в другой, пользуясь рейсами данной авиакомпании. Карта авиарейсов представляла собой ориентированный граф, изображенный на рис. 13.8, *а*. Матрица и список смежности, соответствующие этому графу, показаны на рис. 13.6, *б* и 13.8, *б*. Поскольку в этой задаче чаще всего выполняется операция, позволяющая найти все смежные города (вершины) по отношению к заданному городу (вершине), реализация карты авиарейсов на основе связанного списка оказывается эффективнее. Кроме того, список смежности занимает меньше памяти, чем матрица смежности. Доказать этот факт читатели могут самостоятельно.

Алгоритмы обхода графа

Решение задачи об авиарейсах, описанное в главе 6, сводилось к полному перебору вершин графа, изображенного на рис. 13.8, *а*. В результате обнаруживался ориентированный путь из пункта отправления в пункт назначения. Алгоритм `searchS` начинал свою работу с заданной вершины и выполнял обход всех ребер, соединяющих ее с другими вершинами, пока не обнаруживалась искомая вершина или не становилось ясно, что ориентированного пути между двумя вершинами не существует.

В отличие от стандартного алгоритма обхода графа, алгоритм `searchS` прекращал работу сразу после обнаружения вершины, соответствующей пункту назначения. **Алгоритм обхода графа** (grapf-traversal algorithm) не останавливается, пока не посетит *все доступные вершины*. Иначе говоря, обход графа, начинающийся с вершины v, пройдет через все вершины w, к которым существует путь.

При обходе графа посещаются все доступные вершины

В отличие от обхода дерева, в котором всегда посещаются *все* узлы, при обходе графа посещаются только доступные вершины. Следовательно, все вершины, независимо от стартовой точки, можно обойти только в связном графе (см. упражнение 15). Таким образом, обход графа может использоваться для доказательства его связности.

При обходе посещаются все вершины, только если граф является связным

Если граф не является связным, его обход, начинающийся с вершины v, коснется лишь подмножества вершин графа. Это подмножество называется **связным компонентом** (connected component), содержащим вершину v. Повторяя обход графа несколько раз, начиная с еще не посещенной вершины, можно обнаружить все его связные компоненты.

Связный компонент представляет собой подмножество вершин, посещаемых при обходе графа, начиная от заданной точки

Если граф содержит цикл, то алгоритм его обхода может зациклиться. Чтобы избежать этого, во время посещения нужно помечать вершину и никогда не проходить ее дважды.

Мы рассмотрим два основных алгоритма обхода графов. Они не зависят от того, является ли граф ориентированным или нет. Порядок посещения вершин у этих алгоритмов отличается, но если они начинаются с одной и той же точки, то пройдут через одно и то же множество вершин. Порядок обхода вершин, начинающийся с вершины x, показан на рис. 13.10.

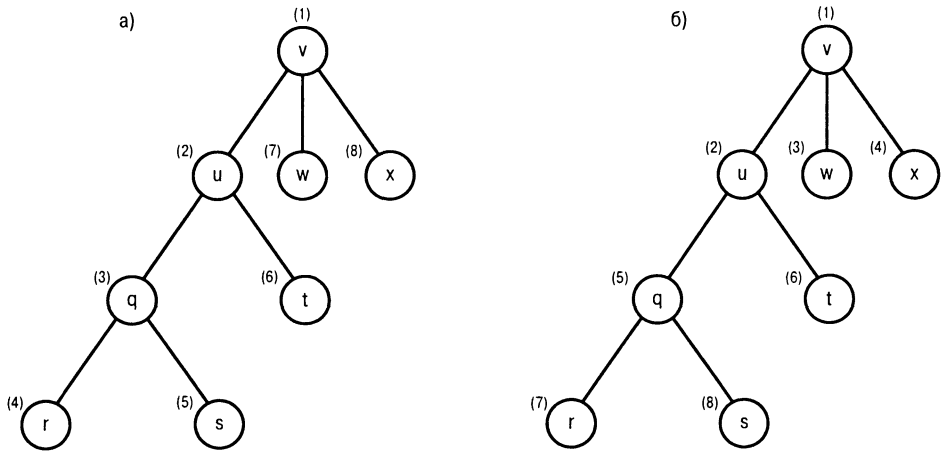


Рис. 13.10. Порядок обхода вершин: а) поиск в глубину; б) поиск в ширину

Поиск в глубину

Стратегия поиска в глубину (DFS — depth-first search) заключается в следующем: обход, начинающийся в вершине v, продолжается как можно дальше в глубину графа, а затем возвращается в исходную точку. Иными словами, после посещения любой вершины стратегия DFS пытается найти еще не посещенную смежную вершину.

Алгоритм поиска в глубину удаляется от вершины как можно дальше, а затем возвращается обратно

Стратегия DFS имеет простую рекурсивную форму.

Рекурсивный алгоритм обхода в глубину

```
dfs(in v:Vertex)
// Обход графа, начиная с вершины v, с помощью
// поиска в глубину: рекурсивная версия.

   Пометить вершину v как посещенную
   for (каждая не посещенная вершина u, смежная с вершиной v)
     dfs(u)
```

Алгоритм поиска в глубину не полностью задает порядок обхода вершин, смежных с вершиной *v*. Например, смежные вершины можно обходить в порядке возрастания их значений (алфавитном или возрастающем). Эта возможность вполне естественно появляется, если граф представлен матрицей смежности или узлы, содержащиеся в каждом связанном списке, хранящемся в списке смежности, упорядочены.

Выберите порядок, в котором посещаются смежные вершины

Как показано на рис. 13.10, *a*, алгоритм обхода в глубину помечает и посещает вершины *v*, *u*, *q* и *r*. Если алгоритм обхода достигает какой-либо вершины, например, вершины *r*, он возвращается обратно и посещает, если это возможно, новую смежную вершину. Таким образом, алгоритм возвращается в точку *q*, а затем переходит в вершину *s*. Продолжая в том же духе, алгоритм обходит все вершины в порядке, указанном на рисунке.

Используя стек, можно сформулировать итеративную стратегию DFS.

Итеративная стратегия обхода в глубину использует стек

```
dfs(in v:Vertex)
// Обход графа, начиная с вершины v,
// с помощью поиска в глубину:
// итеративная версия.

   s.createStack()

   // Заталкиваем вершину v в стек и помечаем ее
   s.push(v)
   Пометить вершину v как посещенную

   // Инвариант цикла: из вершины v, находящейся на дне стека,
   // существует путь в вершину графа, находящуюся
   // на вершине стека
   while (!s.isEmpty())
   {
     if (у вершины графа, находящейся на вершине стека,
         не осталось смежных не посещенных точек)
        s.pop() // Откат
     else
     {
        Выбрать не посещенную точку u, смежную с точкой,
          находящейся на вершине стека
        s.push(u)
        Пометить вершину u как посещенную
     } // Конец оператора if
   } // Конец оператора while
```

Алгоритм *dfs* аналогичен алгоритму *searchS* из главы 6, однако в алгоритме *searchS* оператор *while* прекращал работу, когда на вершине стека оказывался пункт назначения *destination*.

Рассмотрим еще один пример обхода в глубину, изображенный на рис. 13.11. На рис. 13.12 показано содержание стека при обходе в глубину, начинающемся с вершины *a*. Поскольку граф является связным, алгоритм DFS посетит каждую вершину. Порядок обхода имеет следующий вид: *a*, *b*, *c*, *d*, *g*, *e*, *f*, *h*, *i*.

Повторный обход в глубину начинается с вершины, посещенной последней. Эта стратегия называется "*последней посещена, первой исследуется*" (last visted, first explored). Она соответствует как явному стеку, который используется итеративной версией алгоритма *dfs*, так и неявному стеку вершин, который генерируется его рекурсивной версией.

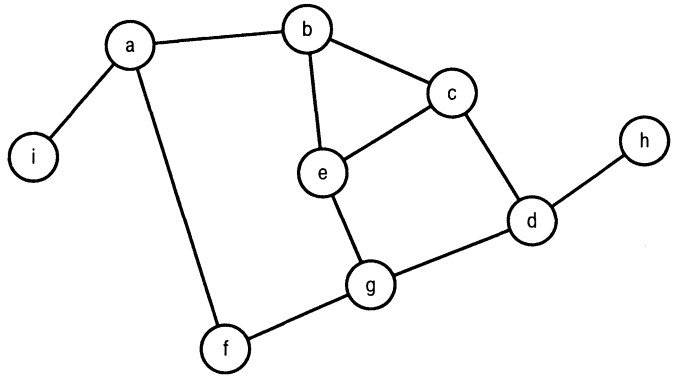


Рис. 13.11. Связный граф с циклами

Посещенный узел	Стек (снизу вверх)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(откат)	a b c d g
f	a b c d g f
(откат)	a b c d g
(откат)	a b c d
h	a b c d h
(откат)	a b c d
(откат)	a b c
(откат)	a b
(откат)	a
i	a i
(откат)	a
(откат)	*(пусто)*

Рис. 13.12. Результаты обхода в глубину графа, изображенного на рис. 13.11, начиная с вершины a

Поиск в ширину

Алгоритм поиска в ширину посещает все смежные вершины, а затем отправляется дальше

Посетив вершину v, алгоритм **поиска в ширину** (BFS — breadth-first search) обходит все смежные с ней вершины и только после этого переходит к следующей вершине. Как показано на рис. 13.10, *б*, алгоритм BFS помечает и посещает вершины u, w и x, смежные с

вершиной v. Поскольку других вершин, смежных с вершиной v, в графе нет, алгоритм начинает обход еще не посещенных вершин, смежных с вершиной u. Продолжая в том же духе, алгоритм посетит все вершины в порядке, указанном на рисунке.

Алгоритм поиска в ширину не применяется ни к одной из вершин, смежных с вершиной v, пока не посетит их все. В отличие от стратегии DFS, которая руководствовалась принципом "*последней посещена, первой исследуется*", алгоритм DFS следует правилу "*первой посещена, первой исследуется*". Следовательно, нет ничего удивительного, что для его реализации применяется очередь. Итеративная версия алгоритма BFS имеет следующий вид.

Итеративная версия алгоритма BFS использует очередь

```
bfs(in v:Vertex)
// Обход графа, начиная с вершины v,
// с помощью поиска в ширину:
// итеративная версия.

   q.createQueue()

   // Помещаем вершину v в очередь и помечаем ее
   q.enqueue(v)
   Пометить вершину v как посещенную

   while (!q.isEmpty())
   {
     q.dequeue(w)
     // Инвариант цикла: из вершины v существует путь
     // в любую вершину, содержащуюся в очереди q
     for (каждая не посещенная вершина u, смежная с вершиной w)
     {
        Пометить вершину u как не посещенную
        q.enqueue(u)
     } // Конец оператора for
   } // Конец оператора while
```

Рекурсивный алгоритм обхода в ширину возможен, но не прост

Рекурсивный алгоритм обхода в ширину не так прост, как рекурсивная версия алгоритма обхода в глубину. Причины этих трудностей описаны в упражнении 16.

На рис. 13.13 показано содержание очереди при обходе графа, изображенного на рис. 13.11, с помощью алгоритма *bfs*. Обход начинается с вершины a. При обходе в ширину посещаются те же точки, что и при обходе в глубину, но в другом порядке. Порядок обхода имеет следующий вид: a, b, f, i, c, e, g, d, h.

Применения графов

В этом разделе мы рассмотрим некоторые из наиболее распространенных применений графов.

Топологическая сортировка

Ориентированный граф, не содержащий циклов (рис. 13.14), упорядочен естественным образом. Например, вершина a предшествует вершине b, которая, в свою очередь, предшествует вершине c. Если вершины обозначают академические курсы, то графы описывают последовательность курсов, которые необходимо прослушать, прежде чем перейти к следующему. Например, курс a нужно изу-

Посещенный узел	Очередь (с начала до конца)
a	a
	(пусто)
b	b
f	bf
i	bfi
	fi
c	fic
e	fice
	ice
g	iceg
	ceg
	eg
d	egd
	gd
	d
	(пусто)
h	h
	(пусто)

Рис. 13.13. Результаты обхода в ширину графа, изображенного на рис. 13.11, начиная с вершины a

чить до курса *b*, причем оба они нужны для освоения курсов *c* и *e*. В каком порядке следует излагать все семь курсов, так чтобы были выполнены предварительные условия? Ответом на этот вопрос является линейный порядок вершин ориентированного графа без циклов, называемый **топологическим порядком** (topological order). В списке вершин, перечисленных в топологическом порядке, вершина *x* предшествует вершине *y*, если вершина *x* соединена с вершиной *y* ориентированным ребром.

Вершины данного графа могут иметь разный топологический порядок. Например, их можно перечислить как

a, *g*, *d*, *b*, *e*, *c*, *f*

или

a, *b*, *g*, *d*, *e*, *f*, *c*.

Если все вершины ориентированного графа линейно упорядочены, то все ребра имеют одинаковую ориентацию. На рис. 13.15 показаны два варианта графа, изображенного на рис. 13.14, в соответствии с разным топологическим порядком следования его узлов.

Упорядочение вершин графа в топологическом порядке называется **топологической сортировкой** (topological sorting). Существуют несколько простых алгоритмов топологической сортировки. Во-первых, можно найти вершину, у которой нет преемников. Удалим ее из графа вместе со всеми ведущими в нее ребрами и поместим в начало списка вершин. Теперь будем последовательно помещать в начало списка все вершины, не имеющие преемников. Когда граф станет пустым, все вершины в списке окажутся топологически упорядоченными. Этот алгоритм описывается следующим псевдокодом.

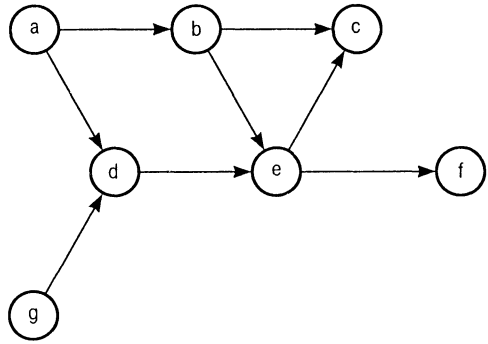


Рис. 13.14. Ориентированный граф, не содержащий циклов

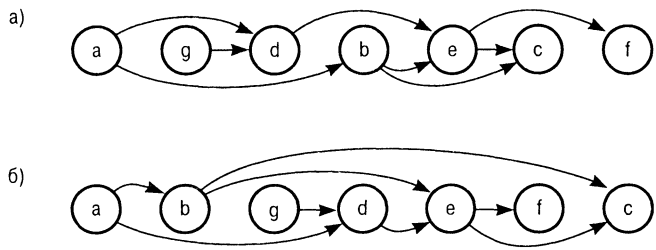


Рис. 13.15. Топологический порядок следования узлов графа, изображенного на рис. 13.14: а) a, g, d, b, e, c, f; б) a, b, g, d, e, f, c

Простой алгоритм топологической сортировки

```
topSort1(in theGraph:Graph,
         out aList:List)
// Упорядочивает вершины графа theGraph
// в топологическом порядке и помещает
// их в список aList

   n = количество вершин в графе theGraph
   for (step = 1 до n)
   {
      Выбрать вершину v, не имеющую преемников
      aList.insert(1, v)
      Удалить из графа theGraph вершину v и ее ребра
   } // Конец оператора for
```

По окончании обхода список вершин *aList* окажется топологически упорядоченным. На рис. 13.16 показаны результаты трассировки этого алгоритма на примере графа, изображенного на рис. 13.14. Полученный в итоге топологический порядок вершин показан на рис. 13.15, *а*.

Второй алгоритм представляет собой простую модификацию итеративного поиска в глубину. Сначала в стек заталкиваются все вершины, не имеющие предшественников. Затем каждый раз, когда из стека выталкивается очередная вершина, она помещается в начало списка. Ниже приводится псевдокод этого алгоритма.

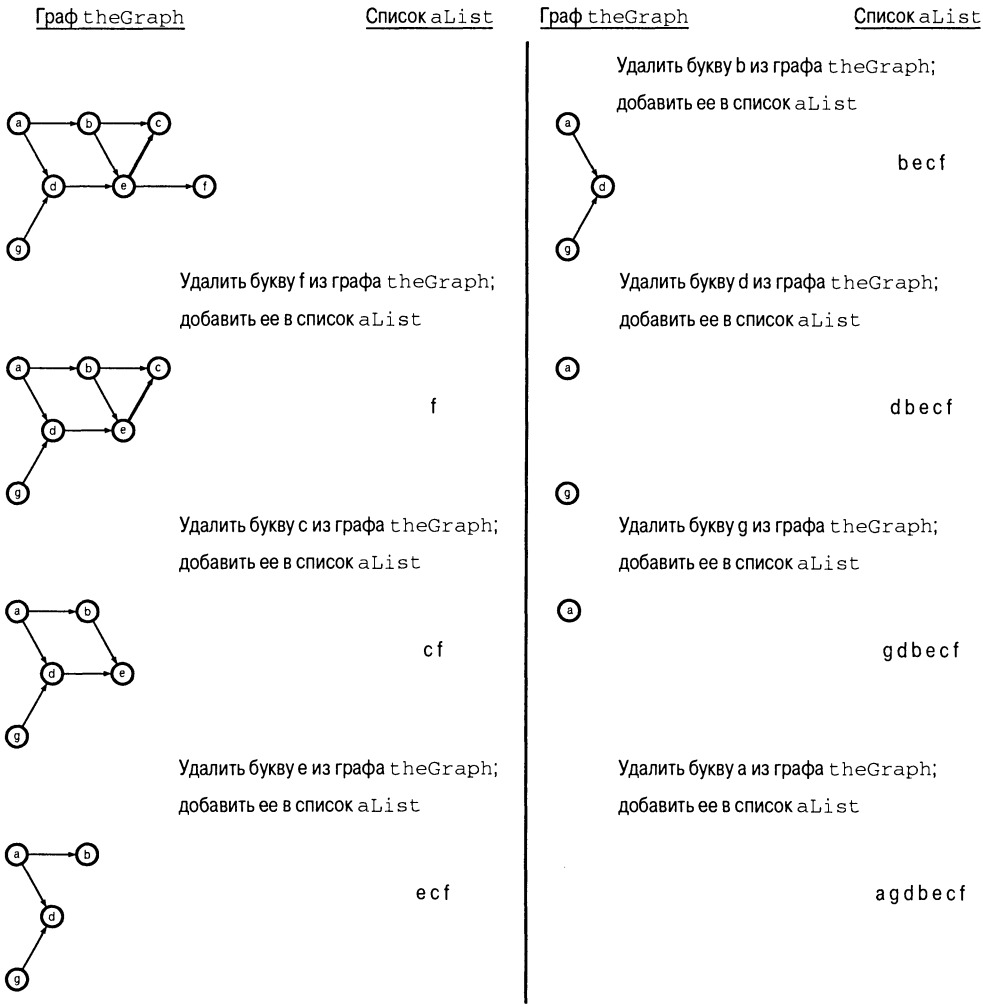


Рис. 13.16. Результаты трассировки алгоритма topSort1 на примере графа, изображенного на рис. 13.14

Алгоритм топологической сортировки на основе поиска в глубину

```
topSort2(in theGraph:Graph,
         out aList:List)
// Упорядочивает вершины графа theGraph
// в топологическом
// порядке и помещает их в список aList

   s.createStack()
   for (все вершины графа)
      if (вершина v не имеет предшественника)
      {
         s.push(v)
         Пометить вершину v как посещенную
      } // Конец оператора if/for
```

```
while (!s.isEmpty())
{
   if (все вершины, смежные с вершиной графа, находящейся
       на вершине стека, уже посещены)
   {
      s.pop(v)
      aList.insert(1, v)
   }
else
   {
      Выбрать не посещенную вершину u, смежную
         с вершиной графа, находящейся на вершине стека
      s.push(u)
      Пометить вершину u как не посещенную
   } // Конец оператора if
} // Конец оператора while
```

По окончании обхода список *aList* оказывается топологически упорядоченным. На рис. 13.17 показаны результаты трассировки этого алгоритма на примере графа, изображенного на рис. 13.14. Полученный в итоге топологический порядок вершин показан на рис. 13.15, *б*.

Действие	Стек (от дна до вершины)	Список aList (от начала до конца)
Вытолкнуть букву a	a	
Вытолкнуть букву g	ag	
Вытолкнуть букву d	agd	
Вытолкнуть букву e	agde	
Вытолкнуть букву c	agdec	
Вытолкнуть букву c из стека и добавить в список aList	agde	c
Вытолкнуть букву f	agdef	c
Вытолкнуть букву f из стека и добавить в список aList	agde	fc
Вытолкнуть букву e из стека и добавить в список aList	agd	efc
Вытолкнуть букву d из стека и добавить в список aList	ag	defc
Вытолкнуть букву g из стека и добавить в список aList	a	gdefc
Вытолкнуть букву b	ab	gdefc
Вытолкнуть букву b из стека и добавить в список aList	a	bgdefc
Вытолкнуть букву a из стека и добавить в список aList	(*пусто*)	abgdefc

Рис. 13.17. Результаты трассировки алгоритма topSort2 на примере графа, изображенного на рис. 13.14

Остовные деревья

Дерево — это разновидность неориентированного связного графа, но без циклов. Каждая вершина графа, изображенного на рис. 13.3, *а*, может быть корнем другого дерева. Несмотря

> Дерево представляет собой неориентированный связный граф без циклов

на то что все деревья являются графами, не все графы являются деревьями. Узлы (вершины) дерева иерархически упорядочены, а граф может не обладать таким свойством.

Остовное дерево (spanning tree) связного неориентированного графа G представляет собой подграф, содержащий все вершины графа G и ребра, образующие дерево. Например, на рис. 13.18 показано остовное дерево графа, изображенного на рис. 13.11. Пунктирными линиями обозначены ребра, которые не вошли в остовное дерево. Построить остовное дерево графа можно по-разному.

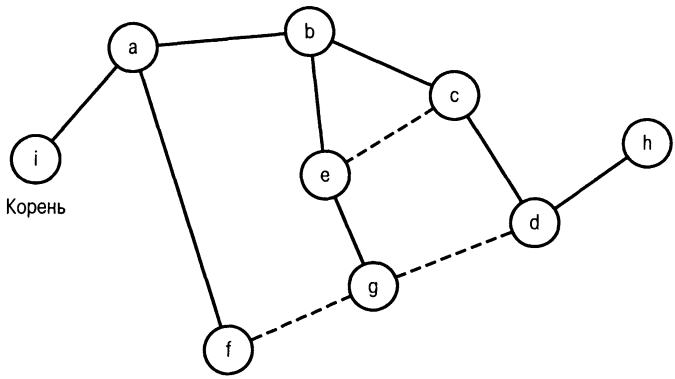


Рис. 13.18. Остовное дерево графа, изображенного на рис. 13.11

Если из связного неориентированного графа, содержащего циклы, выбрасывать ребра, пока циклы не исчезнут, получится его остовное дерево. Этот способ позволяет легко узнать, содержит ли данный граф циклы. Для этого достаточно проверить следующие свойства.

Свойства, которые позволяют обнаружить циклы в неориентированном графе

1. **Связный неориентированный граф, содержащий n вершин, должен иметь как минимум n–1 ребро.** Чтобы установить этот факт, напомним, что в связном графе существует путь между любыми двумя точками. Допустим, что вначале граф содержал n вершин. Выберем произвольную вершину и нарисуем ребро, соединяющее эту вершину с любой другой вершиной. Затем проведем ребро, соединяющее вторую вершину с любой другой еще не присоединенной вершиной. Если продолжать этот процесс, пока все вершины не окажутся соединенными друг с другом, мы получим связный граф (см. рис. 13.19). Если граф состоит из n вершин, то он содержит n-1 ребро. Кроме того, если из графа удалить какое-нибудь ребро, он станет несвязным.

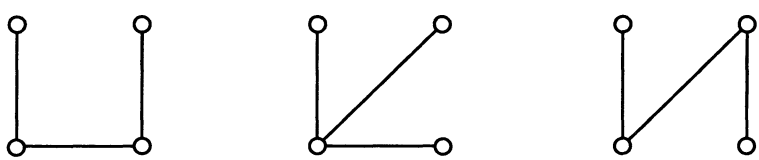


Рис. 13.19. Связные графы, состоящие из четырех вершин и трех ребер

2. **Связный неориентированный граф, состоящий из n вершин и n-1 ребра, не может содержать циклы.** Чтобы в этом убедиться, начнем с простого наблюдения: для того чтобы граф, содержащий n вершин, был связным,

он должен иметь по крайней мере n-1 ребер. Если бы связный граф содержал цикл, мы могли бы удалить любое ребро, входящее в этот цикл, и граф остался бы связным. Таким образом, если бы связный неориентированный граф, состоящий из n вершин и n-1 ребра, содержал цикл, удалив ребро, входящее в этот цикл, мы получили бы связный граф, содержащий только n-2 ребра, что невозможно в силу свойства 1.

3. **Связный неориентированный граф, состоящий из n вершин и более чем из n-1 ребра, должен содержать хотя бы один цикл.** Например, если в любой из графов, изображенных на рис. 13.19, добавить еще одно ребро, будет создан цикл. Этот факт мы оставим без доказательства (см. упражнение 14).

Итак, чтобы определить, содержит ли граф цикл, нужно просто подсчитать количество вершин и ребер.

Чтобы определить, содержит ли граф цикл, нужно просто подсчитать количество вершин и ребер

Следовательно, дерево, представляющее собой связный неориентированный граф без циклов, должно содержать n узлов и n-1 ребер. Итак, чтобы создать остовное дерево для связного графа, состоящего из n вершин, нужно удалять ребра, входящие в циклы, пока не останется ровно n-1 ребро.

Два алгоритма, позволяющих определить остовное дерево графа, основаны на алгоритмах обхода, описанных выше. Как правило, эти алгоритмы приводят к разным остовным деревьям, соответствующим одному и тому же графу.

Остовное дерево поиска в глубину. Чтобы определить остовное дерево графа, можно обойти его с помощью алгоритма поиска в глубину. Обходя граф, следует помечать его ребра. По завершении обхода вершины графа и помеченные ребра образуют **остовное дерево поиска в глубину** (depth-first search (DFS) spanning tree). (Чтобы получить остовное дерево, достаточно удалить из графа непомеченные ребра.) Алгоритм создания остовного дерева поиска в глубину получается путем простой модификации итеративной и рекурсивной версий алгоритма *dfs*. Рассмотрим его рекурсивный вариант.

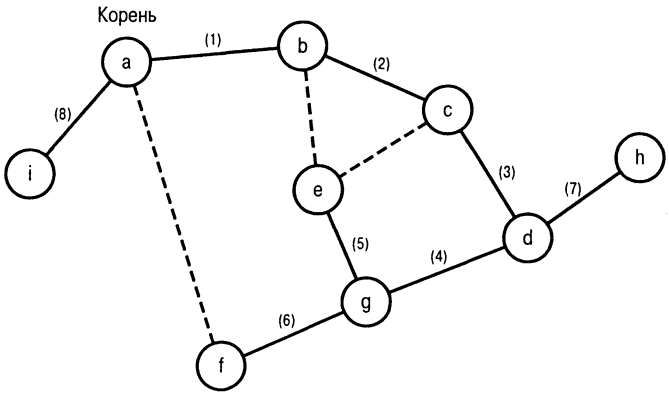
Алгоритм создания остовного дерева поиска в глубину

```
dfsTree(in v:Vertex)
// Формирует остовное дерево связного
// неориентированного графа, начиная
// с вершины v, с помощью поиска в глубину:
// рекурсивная версия.

   Пометить вершину v как посещенную
   for (каждая не посещенная вершина u, смежная с вершиной v)
   {
      Пометить ребро, соединяющее вершины u и v
      dfsTree(u)
   } // Конец оператора for
```

Применяя этот алгоритм к дереву, изображенному на рис. 13.11, можно получить остовное дерево поиска в глубину, корнем которого является вершина *a* (рис. 13.20). На рисунке показан также порядок, в котором алгоритм обходит вершины и помечает ребра. Попробуйте самостоятельно выполнить трассировку этого алгоритма.

Остовное дерево поиска в ширину. Есть еще один способ определения остовного дерева графа. Для этого его нужно обойти с помощью алгоритма поиска в ширину. Обходя граф, следует помечать его ребра. По завершении обхода вер-



Алгоритм создания основного дерева поиска в глубину посещает вершины в следующем порядке: a, b, c, d, g, e, f, h, i. Номера задают порядок, в котором алгоритм помечает ребра

Рис. 13.20. Остовное дерево поиска в глубину с корнем в вершине a для графа, изображенного на рис. 13.11

шины графа и помеченные ребра образуют **остовное дерево поиска в ширину** (breadth-first search (DFS) spanning tree). (Чтобы получить остовное дерево, достаточно удалить из графа непомеченные ребра.) Алгоритм создания остовного дерева поиска в ширину получается путем модификации итеративной версии алгоритма *bfs*.

```
bfsTree(in v:Vertex)
// Формирует остовное дерево связного неориентированного
// графа, начиная с вершины v, с помощью поиска в глубину:
// итеративная версия.

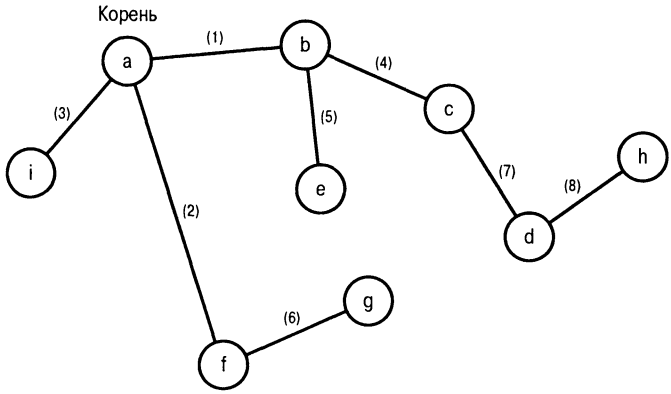
   q.createQueue()

   // Добавляем вершину v в очередь и помечаем ее
   q.enqueue(v)
   Пометить вершину v как посещенную

   while (!q.isEmpty())
   {
      q.dequeue(w)
      // Инвариант цикла: из вершины v существует путь
      // в любую вершину, содержащуюся в очереди q

     for (каждая не посещенная вершина u, смежная с вершиной w)
     {
        Пометить вершину u как посещенную
        Пометить ребро, соединяющее вершины w и u
        q.enqueue(u)
     } // Конец оператора for
   } // Конец оператора while
```

Применив этот алгоритм к дереву, изображенному на рис. 13.11, мы получим остовное дерево поиска в ширину, корнем которого является вершина *a* (рис. 13.21). На рисунке показан также порядок, в котором алгоритм обходит вершины и помечает ребра. Попробуйте самостоятельно выполнить трассировку этого алгоритма.



Алгоритм создания основного дерева поиска в ширину посещает вершины в следующем порядке: a, b, c, d, g, e, f, h, i. Номера задают порядок, в котором алгоритм помечает ребра

Рис. 13.21. Остовное дерево поиска в ширину с корнем в вершине a для графа, изображенного на рис. 13.11

Минимальные остовные деревья

Представьте себе, что некая развивающаяся страна пригласила вас для разработки телефонной системы, в которой из каждого города этой страны можно позвонить в любой другой город. Очевидно, для этого попарно можно соединить телефонными линиями все города. Однако гористая местность не позволяет протянуть телефонную линию между любыми двумя городами. Изобразим схему телефонных линий в виде взвешенного неориентированного графа, представленного на рис. 13.22. Вершины графа обозначают n городов. Ребро, соединяющее две вершины, обозначает телефонную линию, протянутую между двумя городами, а вес ребра — стоимость такой линии. Обратите внимание, что если бы этот граф оказался несвязным, соединить все города телефонными линиями оказалось бы невозможно. Однако граф, изображенный на рис. 13.22, является связным, что облегчает нашу задачу.

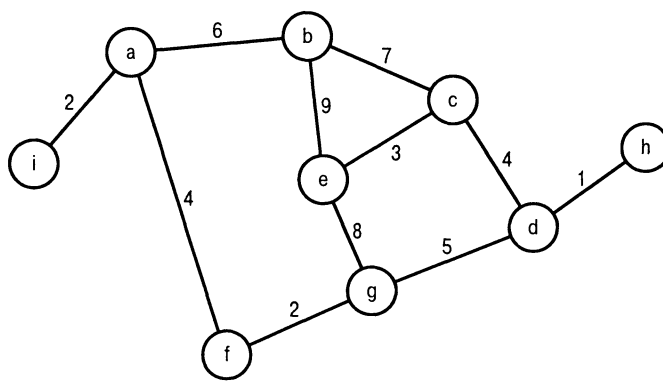


Рис. 13.22. Взвешенный связный неориентированный граф

Протянув телефонные линии, соответствующие ребрам графа, мы решили бы поставленную задачу, но решение было бы слишком дорогим. Учитывая свойство 1, сформулированное в предыдущем разделе, для соединения n вершин связного графа необходимо n-1 ребро. Следовательно, число n-1 — это минимальное количество телефонных линий, которыми можно соединить n городов.

Если стоимость всех телефонных линий одинакова, задачу можно свести к нахождению любого остовного дерева графа. Общая стоимость телефонной сети — т.е. стоимость основного дерева — равна сумме стоимостей каждого ребра, образующего остовное дерево. Однако, как показано на рис. 13.22, стоимость телефонных линий варьируется. Поскольку один и тот же граф может иметь несколько остовных деревьев, а стоимость телефонных линий является переменной величиной, нужно найти остовное дерево, имеющее минимальную стоимость. Иными словами, необходимо найти остовное дерево, сумма весов ребер которого является минимальной. Такое остовное дерево называется **минимальным**, и оно не единственно. Однако, несмотря на то что один и тот же граф может иметь несколько минимальных остовных деревьев, их стоимость одинакова.

Минимальное остовное дерево связного неориентированного графа имеет минимальную сумму весов своих ребер

Для нахождения минимального остовного дерева, начинающегося в любой вершине графа, существует простой алгоритм, называемый алгоритмом Прима (Prim). В исходном состоянии дерево состоит только из одной вершины. На каждом шаге алгоритма среди ребер, начинающихся в вершинах, принадлежащих дереву, и заканчивающихся в вершинах, не принадлежащих ему, выбирается ребро, имеющее наименьший вес. Затем соответствующая вершина и ребро, имеющее минимальный вес, добавляется в дерево. Рассмотрим псевдокод этого алгоритма.

Алгоритм нахождения минимального остовного дерева

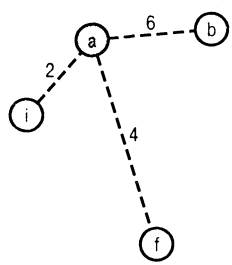
```
primsAlgorithm(in v:Vertex)
// Определяет минимальное остовное
// дерево взвешенного связного
// неориентированного графа, начинающееся
// в вершине v. Веса ребер неотрицательны.

   Пометить вершину v как посещенную и включить ее
      в минимальное остовное дерево

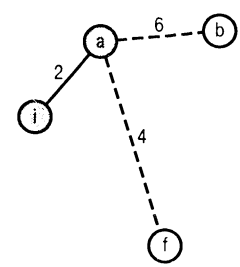
   while (остались не посещенные вершины)
   {
      Найти ребро (v, u) с минимальным весом, соединяющее
         посещенную вершину v и не посещенную вершину u
      Пометить вершину u как посещенную
      Добавить вершину u и ребро (v, u)
         в минимальное дерево поиска
   } // Конец оператора while
```

На рис. 13.23 показаны результаты трассировки алгоритма *primsAlgorithm* для графа, изображенного на рис. 13.22, начиная с вершины *a*. Ребра, входящие в дерево, изображены сплошными линиями, а проверяемые ребра — пунктиром.

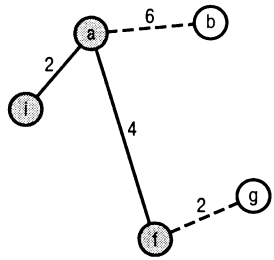
Доказательство утверждения, что алгоритм Прима позволяет найти именно минимальное остовное дерево, выходит за рамки нашей книги.



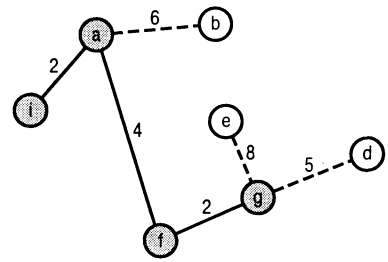
а) Пометить вершину a, проверить ребра, выходящие из вершины a



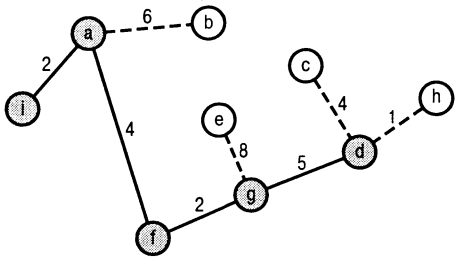
б) Пометить вершину i, добавить ребро (a, i)



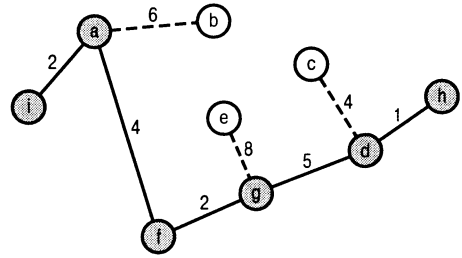
в) Пометить вершину f, добавить ребро (a, f)



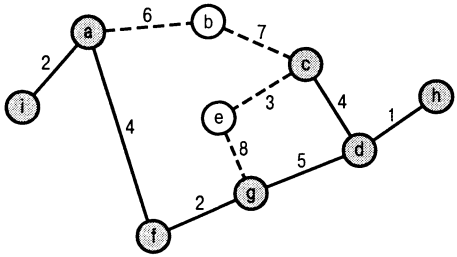
г) Пометить вершину g, добавить ребро (f, g)



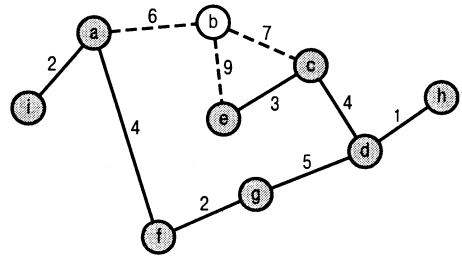
д) Пометить вершину d, добавить ребро (g, d)



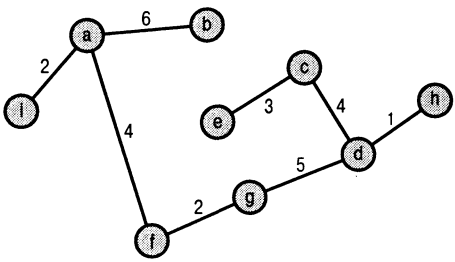
е) Пометить вершину h, добавить ребро (d, h)



ж) Пометить вершину c,добавить ребро (d, c)



з) Пометить вершину e, добавить ребро (c, e)



и) Пометить вершину b, добавить ребро (a, b)

Рис. 13.23. Результаты трассировки алгоритма primsAlgorithm для графа, изображенного на рис. 13.22, начиная с вершины a

Кратчайшие пути

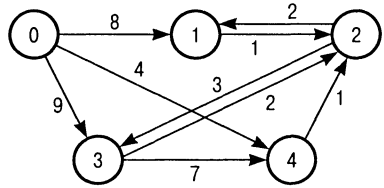
Вернемся к карте авиарейсов. Она представляет собой взвешенный ориентированный граф: города обозначаются вершинами, а перелеты из одного города в другой — ребрами. Вес ребра задает расстояние между городами (вершинами). Таким образом, веса не могут быть отрицательными. Например, можно объединить два графа, изображенных на рис. 13.5, и получить взвешенный ориентированный граф.

Кратчайший путь между двумя вершинами взвешенного графа состоит из ребер, имеющих минимальную сумму весов

Рассмотрим задачу о нахождении кратчайшего пути между двумя вершинами взвешенного ориентированного графа. Кратчайший путь между двумя вершинами взвешенного графа состоит из ребер, имеющих минимальную сумму весов. Используя термин "кратчайший", следует иметь в виду, что вес может обозначать не только расстояние, но и стоимость перелета, либо его продолжительность. Сумма весов ребер, образующих путь, называется **длиной пути** (length), или его **весом** (weight), или **стоимостью** (cost).

Например, кратчайший путь из вершины 0 в вершину 1 в графе, представленном на рис. 13.24, *а*, состоит не из ребра между 0 и 1 — его стоимость равна 8, — а из ребер, соединяющих вершины 0 – 4 – 2 – 1. Его общая стоимость равна 7. Для удобства стартовая точка обозначается числом 0, а другие вершины перенумеровываются числами от 1 до n-1. Обратите внимание на матрицу смежности, приведенную на рис. 13.24, *б*.

а) Начало



б)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Рис. 13.24. Задача о нахождении кратчайшего пути: а) взвешенный ориентированный граф; б) его матрица смежности

Определение кратчайшего пути, соединяющего вершину 0 со всеми другими вершинами

Сформулированный ниже алгоритм, приписываемый Э. Дейкстре (E. Dijkstra), позволяет найти кратчайший путь, соединяющий вершину 0 со *всеми* другими вершинами. Этот алгоритм использует множество выбранных вершин `vertexSet` и массив `wight`, где в ячейке `weight[i]` хранится вес кратчайшего пути из вершины 0 в вершину `v`, проходящий через вершины, принадлежащие множеству `vertexSet`.

Если вершина `v` принадлежит множеству `vertexSet`, кратчайший путь содержит только вершины из этого множества. В противном случае вершина `v` является единственной вершиной, лежащей на кратчайшем пути, которая не принадлежит множеству `vertexSet`. Иначе говоря, путь заканчивается ребром, соединяющим вершину, принадлежащую множеству `vertexSet`, и вершину `v`.

В исходном положении множество `vertexSet` состоит лишь из одной вершины 0, а массив `weight` содержит веса ребер, соединяющих вершину 0 со всеми остальными вершинами. Иными словами, элемент `weight[v]` равен элементу `matrix[0][v]` для всех `v`, где массив `matrix` является матрицей смежности.

Итак, в исходном положении массив *weight* является первой строкой массива *matrix*.

После инициализации обнаруживается вершина *v*, не принадлежащая множеству *vertexSet*, которой соответствует минимальный вес *weight[v]*. Добавим вершину *v* во множество *vertexSet*. Для всех (невыбранных) вершин *u*, не принадлежащих множеству *vertexSet*, проверяем значения элемента *weight[u]* и проверяем, не являются ли они минимальными. Иначе говоря, можно ли уменьшить значение *weight[u]* — вес пути от вершины 0 к вершине *u*, — пройдя через вновь выбранную вершину *v*?

Для того чтобы проверить это, разобьем путь от вершины 0 к вершине *u* на две части и вычислим их веса.

```
weight[v] = вес кратчайшего пути из вершины 0 к вершине v
matrix[v][u] = вес ребра вершины 0 к вершине v
```

Затем сравним числа *weight[u]* и *weight[v]+matrix[v][u]*. Найдем наименьшее из них и запишем в ячейку *weight[v]*.

```
weight[v] = меньшее из чисел weight[u] и weight[v]+matrix[v][u]
```

Псевдокод **алгоритма Дейкстры** имеет следующий вид.

Алгоритм нахождения кратчайшего пути

```
shortestPath(in theGraph:Graph, in
weight:WeightArray)
// Находит кратчайший путь от вершины 0
// ко всем остальным вершинам взвешенного
// ориентированного графа theGraph;
// все веса графа theGraph являются неотрицательными.

   // Шаг 1: инициализация
   Создать множество vertexSet, состоящее из единственной вершины 0
   n = количество вершин в графе theGraph
   for (v = 0 до n - 1)
      weight[v] = matrix[0][v]

   // Шаги от 2 до n
   // Инвариант: если вершина v, не принадлежит множеству
   // vertexSet, элемент weight[v] равен наименьшему весу
   // среди всех путей из вершины 0 в вершину v, проходящих
   // через вершины, принадлежащие множеству vertexSet.
   // Если вершина v принадлежит множеству vertexSet,
   // элемент weight[v] является наименьшим весом
   // среди всех путей из вершины 0 в вершину v, причем
   // кратчайший путь целиком лежит во множестве vertexSet.
   for (step = 2 до n)
   {
      Найти наименьший элемент weight[v], такой что
        вершина v не принадлежит множеству vertexSet
      Добавить вершину v в множество vertexSet

     // Проверить элемент weight[u] для всех вершин u,
     // не принадлежащих множеству vertexSet
     for (все вершины u, не принадлежащие множеству vertexSet)
       if (weight[u] > weight[v] + matrix[v][u])
           weight[u] = weight[v] + matrix[v][u]
   } // Конец оператора for
```

Инвариант цикла утверждает, что если вершина *v* помещена в множество *vertexSet*, элемент *weight[i]* равен весу самого короткого пути из вершины 0 в вершину *v* и уже не может измениться.

Результаты трассировки алгоритма на примере графа, изображенного на рис. 13.24, *а*, приведены на рис. 13.25. Алгоритм состоит из следующих шагов.

Шаг	v	vertexSet	wght[0]	wght[1]	wght[2]	wght[3]	wght[4]
1	-	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Рис. 13.25. Результаты трассировки алгоритма нахождения кратчайшего пути на примере графа, изображенного на рис. 13.24, а

Трассировка алгоритма Дейкстры

Шаг 1. В исходном положении множество *vertexSet* содержит вершину 0, а массив *weight* является первой строкой матрицы смежности графа, изображенной на рис. 13.24, *б*.

Шаг 2. Элемент *weight*[4]=4 является наименьшим числом в массиве *weight*, не считая элемента *weight[0]*, поскольку вершина 0 принадлежит множеству *vertexSet*. Следовательно, *v*=4, поэтому нужно добавить вершину 4 в множество *vertexSet*. Для вершин, не входящих в множество *vertexSet*, т.е. вершин 1, 2 и 3, следует проверить, короче ли путь из вершины 0 в вершину 4, а затем — в вершину *u*, чем ребро, соединяющее вершину 0 с вершиной *u*. Для вершин 1 и 3 путь через вершину 4 оказывается не короче. Однако *weight*[2]=∞>*weight*[4]+*matrix*[4][2]=4+1=5, поэтому элемент *weight*[2] нужно заменить числом 5. Справедливость этого утверждения можно проверить непосредственно, посмотрев на граф, изображенный на рис. 13.26, *а*.

Шаг 3. Элемент *weight*[2]=5 является наименьшим числом в массиве *weight*, не считая элементов *weight[0]* и *weight[4]*, поскольку вершины 0 и 4 принадлежат множеству *vertexSet*. Следовательно, *v*=2, поэтому нужно добавить вершину 2 в множество *vertexSet*. Для вершин, не входящих в множество *vertexSet*, т.е. вершин 1 и 3, следует проверить, короче ли путь из вершины 0 в вершину 2, а затем — в вершину *u*, чем ребро, соединяющее вершину 0 с вершиной *u*. (См. рис. 13.26, *б* и 13.26, *в*.)

weight[1]=8 >*weight*[2]+*matrix*[2][1]=5+2=7, поэтому элемент *weight*[1] нужно заменить числом 7.

weight[3]=9 >*weight*[2]+*matrix*[2][3]=5+3=8, поэтому элемент *weight*[3] нужно заменить числом 8.

Шаг 4. Элемент *weight*[1]=7 является наименьшим числом в массиве *weight*, не считая элементов *weight[0]*, *weight[2]* и *weight[4]*, поскольку вершины 0, 2 и 4 принадлежат множеству *vertexSet*. Следовательно, *v*=1, поэтому нужно добавить вершину 1 в множество

vertexSet. Для вершины 3, единственной, которая не входит в множество *vertex*, получаем, что *weight*[3]=8 <*weight*[1]+*matrix*[1][3]= 7+∞, как показано на рис. 13.26, *б*. Следовательно, *weight*[3] нужно оставить без изменения.

Шаг 5. Единственной вершиной, оставшейся за пределами множества *vertesSet*, является вершина 3. Добавляем ее в это множество и прекращаем выполнение алгоритма.

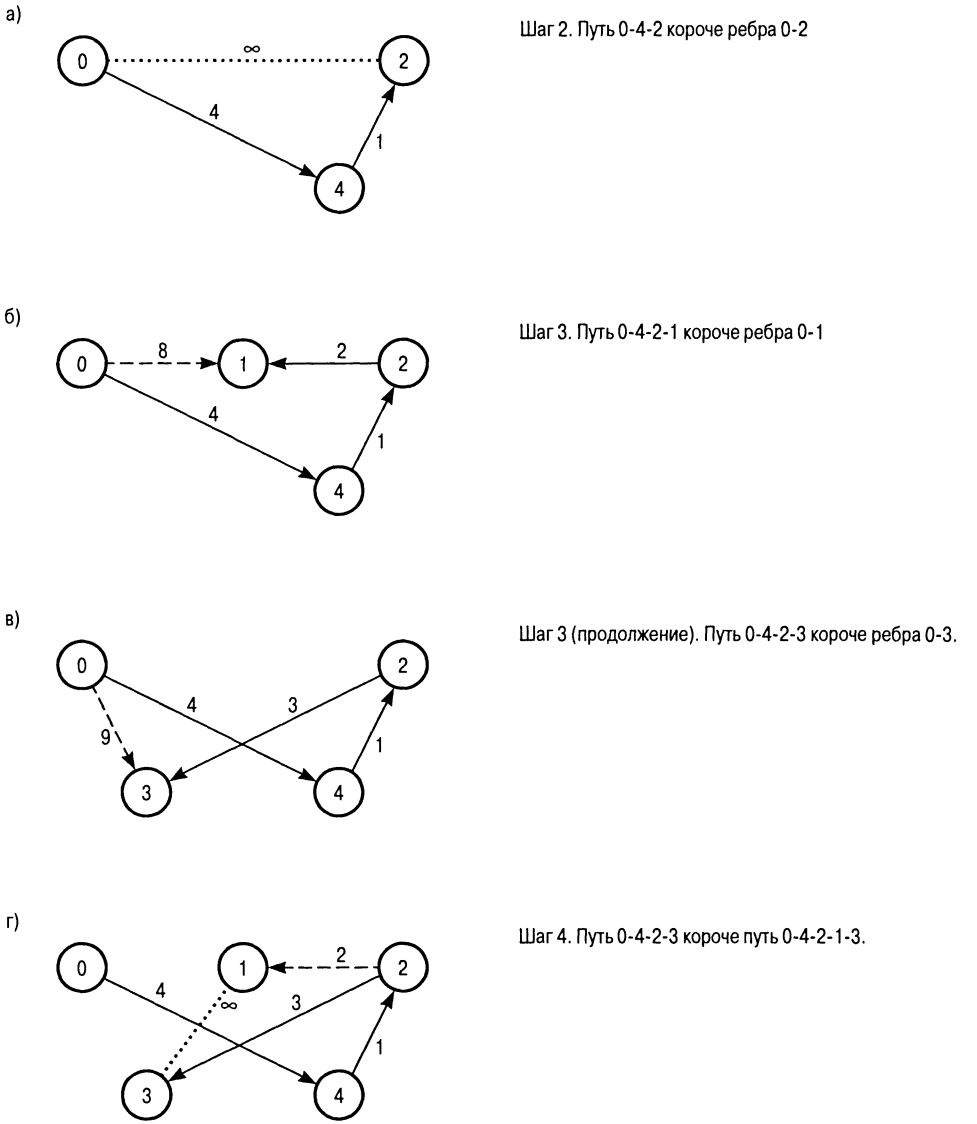


Рис. 13.26. Непосредственная проверка элемента weight[u]: а) значение weight[2] на шаге 2; б) значение weight[1] на шаге 3; в) значение weight[3] на шаге 4

Последние значения, записанные в массиве *weight*, представляют собой веса кратчайшего пути. Эти значения представлены в последней строке на рис. 13.25. Например, кратчайший путь из вершины 0 в вершину 1 имеет вес *weight*[1], т.е. равен 7. Этот результат согласуется с анализом рис. 13.24. Таким образом, кратчайшим является путь 0 – 4 – 2 – 1. Кроме того, кратчайший путь из вершины 0 в вершину 2 имеет вес *weight*[2], т.е. равен 5. Это путь 0 – 4 – 2.

Веса, записанные в массиве *weight*, являются наименьшими, поэтому инвариант цикла выполняется. Доказательство инварианта цикла методом индукции по номеру шага *step* является предметом упражнения 17.

Простые цепи

Простая цепь (circuit) — это всего лишь синоним цикла, который употребляется в формулировках некоторых задач. Напомним, что циклом в графе называется путь, который начинается и заканчивается в одной и той же вершине. Обычная простая цепь либо проходит по одному разу, либо каждую вершину, либо каждое ребро.

Вероятно, самое первое упоминание о графах относится к началу XVIII века, когда Эйлер сформулировал задачу о мостах. Два острова на реке соединены друг с другом и с берегами реки несколькими мостами, как показано на рис. 13.27, *а*. Мосты соответствуют ребрам мультиграфа, изображенного на рис. 13.27, *б*. Острова и берега реки изображаются вершинами. Задача заключается в следующем: можно ли выйти из вершины *v*, пройти по каждому ребру только один раз и снова оказаться в вершине *v*. Эйлер доказал, что существует такая конфигурация ребер и вершин, для которой эта задача не имеет решения.

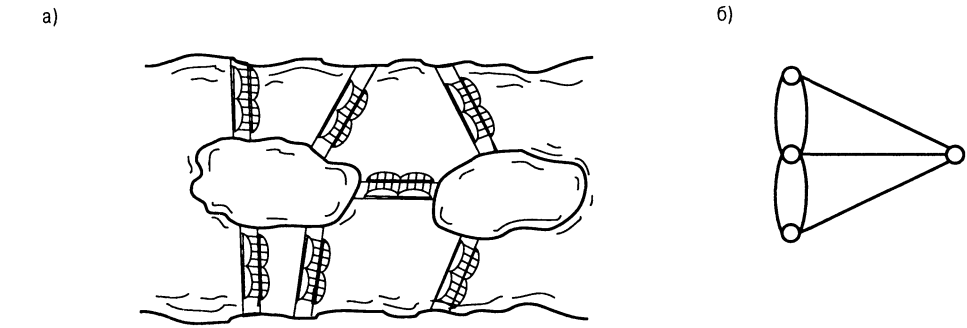


Рис. 13.27. Пример простой цепи: а) задача Эйлера о мостах; б) мультиграф

Простая цепь Эйлера начинается в вершине v, проходит по каждому ребру только один раз и заканчивается в вершине v

Для простоты рассмотрим не мультиграф, а неориентированный граф. Путь в неориентированном графе, начинающийся в вершине *v*, проходящий по каждому ребру только один раз и заканчивающийся в вершине *v*, называется **простой цепью Эйлера** (Euler circuit). Эйлер показал, что для существования такой простой цепи необходимо и достаточно, чтобы из каждой вершины выходило четное количество ребер. Интуитивно ясно, если мы пришли в вершину, пройдя по одному ребру, то выйти должны по другому. В противном случае достичь всех вершин не удается.

Отыскание простой цепи Эйлера равносильно рисованию диаграмм, изображенных на рис. 13.28, не отрывая карандаша от бумаги и не возвращаясь по проведенной линии. Для диаграммы, представленной на рис. 13.28, *а*, решения

не существует, однако можно довольно легко нарисовать диаграмму, показанную на рис. 13.28, *б*. На рис. 13.29 изображены неориентированные графы, основанные на диаграммах, приведенных на рис. 13.28. На рис. 13.29, *а* количество ребер, выходящих из вершин *h* и *i*, четно (равно 3), поэтому простая цепь Эйлера не существует. Однако из каждой вершины графа, представленного на рис. 13.29, *б*, выходит четное количество ребер, поэтому простая цепь Эйлера существует. Обратите внимание, что графы являются связными. Если граф является несвязным, обойти *все* его вершины в принципе невозможно.

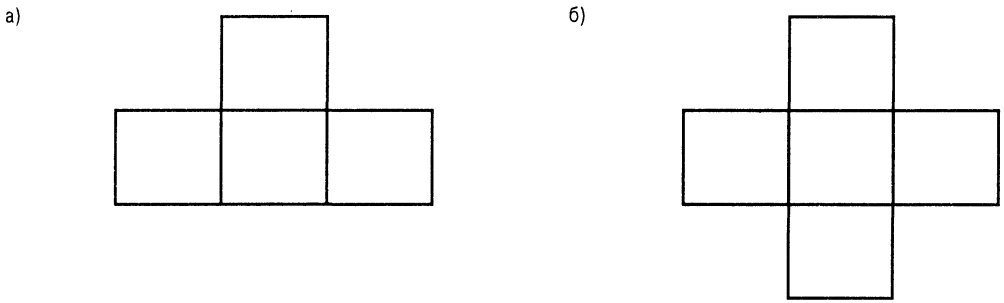


Рис. 13.28. Рисование диаграмм

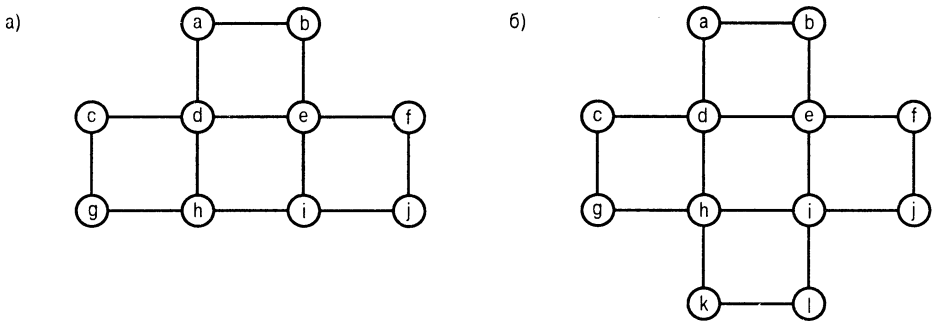


Рис. 13.29. Связные неориентированные графы, основанные на диаграммах, изображенных на рис. 13.28

Найдем простую цепь Эйлера на рис. 13.29, *б*, начиная с произвольной вершины *a*. Стратегия использует поиск в глубину, при котором помечаются не пройденные вершины, а ребра. Напомним, что при поиске в глубину алгоритм находит путь, ведущий из вершины *a* как можно более глубже. Отмечая ребра, мы вернемся в исходную вершину, т.е. найдем цикл. В нашем примере, если обходить вершины в алфавитном порядке, как показано на рис. 13.30, можно найти цикл, образованный вершинами *a*, *b*, *e*, *d*, *a*. Очевидно, этот цикл не является решением нашей задачи, поскольку мы не прошли по всем ребрам. Продолжим поиск.

Найдем первую вершину в цикле *a*, *b*, *e*, *d*, *a*, из которой выходит не пройденное ребро. В нашем примере этой вершиной оказалась вершина *e*. Применим модифицированный поиск в глубину, начиная с этой вершины. В результате обнаружится цикл *e*, *f*, *j*, *i*, *e*. Соединим его с циклом, найденным нами на предыдущем шаге. Иначе говоря, обнаружив вершину *e* в предыдущем цикле, мы проходим по второму циклу, а затем возвращаемся в первый. В результате возникает путь *a*, *b*, *e*, *f*, *j*, *i*, *e*, *d*, *a*, показанный на рис. 13.30, *б*.

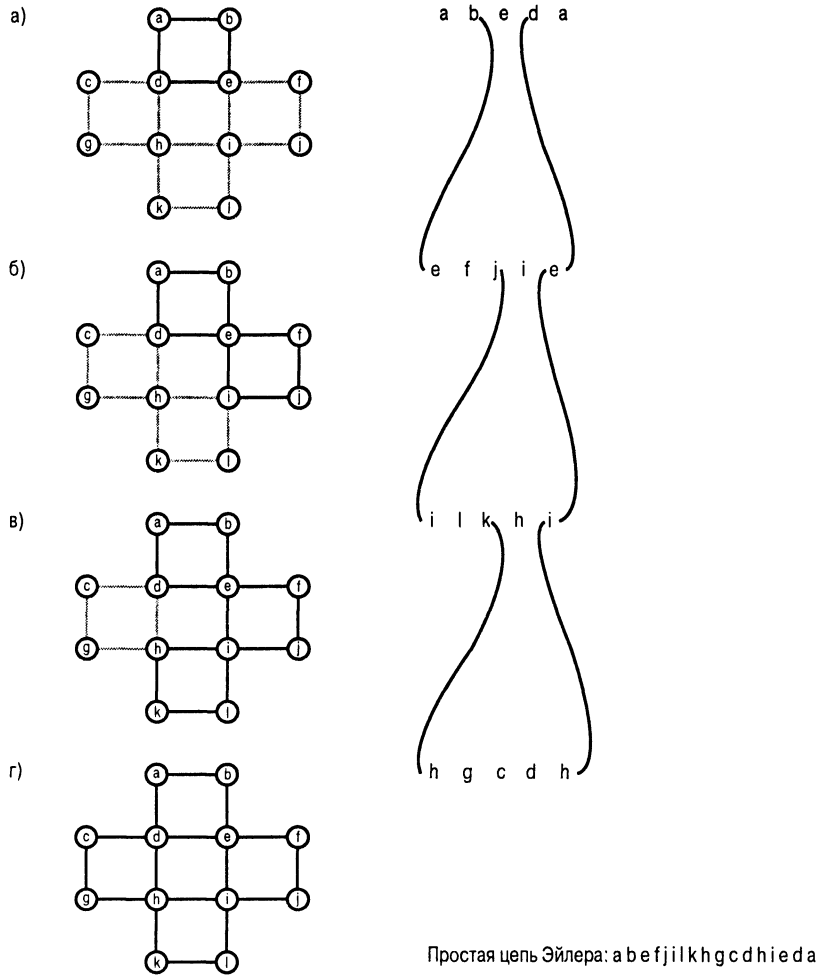


Рис. 13.30. Шаги, выполняемые при поиске простой цепи Эйлера в графе, изображенном на рис. 13.29, б

Первой вершиной, принадлежащей объединенному циклу, из которой выходит не пройденное ребро, является вершина *i*. Начиная с нее, пройдем по циклу *i*, *l*, *k*, *h*, *i*. Объединяя этот цикл с предыдущими, обнаружим путь *a*, *b*, *e*, *f*, *j*, *i*, *l*, *k*, *h*, *i*, *e*, *d*, *a*. (См. рис. 13.30, *в*.) Теперь первой вершиной, принадлежащей объединенному циклу, из которой выходит не пройденное ребро, является вершина *h*. Начиная с нее, обнаружим цикл *h*, *g*, *c*, *d*, *h*. Объединяя этот цикл с предыдущими, получим путь *a*, *b*, *e*, *f*, *j*, *i*, *l*, *k*, *h*, *g*, *c*, *d*, *h*, *i*, *e*, *d*, *a*. (См. рис. 13.30, *г*.)

Некоторые трудные задачи

Решения описанных ниже задач настолько трудны, что выходят за рамки нашей книги.

Задача о странствующем коммивояжере. Простая цепь Гамильтона (Hamilton circuit) представляет собой путь, начинающийся в вершине v, проходящий через каждую вершину графа только один раз и заканчивающийся в вершине v. Определить, содержит ли тот или иной граф простую цепь Гамильтона, бывает довольно трудно. Существует хорошо известный вариант этой задачи — задача о коммивояжере, — в которой рассматривается взвешенный граф, представляющий собой карту дорог. Каждое ребро имеет вес, выражающий расстояние между городами, или время, которое затрачивается на переезд из одного города в другой. Коммивояжер должен выехать из пункта отправления, объехать все города по одному разу и вернуться обратно. Однако путешествие по простой цепи должно быть как можно более дешевым.

> Простая цепь Гамильтона начинается в вершине v, проходит через каждую вершину графа только один раз и заканчивается в вершине v

К сожалению, эту задачу очень трудно решить. Несмотря на то что решение существует, алгоритм его нахождения выполняется довольно медленно, а лучшего пока не придумали.

Задача о трех услугах. Представьте себе три дома A, *B* и *C* и три услуги *X*, *Y* и *Z* (например, телефон, вода и электричество). Эта ситуация продемонстрирована на рис. 13.31. Если дома и услуги изобразить в виде вершин графа, можно ли соединить каждый дом с каждой услугой непересекающимися ребрами? Оказывается, нет.

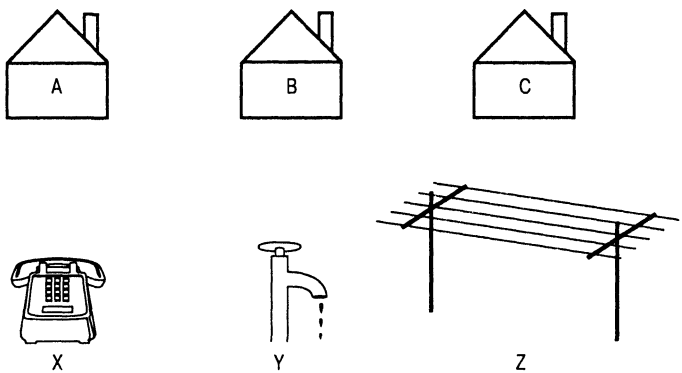


Рис. 13.31. Задача о трех услугах

Граф называется **планарным** (planar), если мы можем изобразить его на плоскости так, чтобы ни одна пара ребер не пересекалась. Задача о трех услугах является разновидностью более общей задачи: определить, является ли граф планарным. Эта задача имеет очень много приложений. Например, граф может имитировать электронную схему, в которой вершины представляют собой компоненты, а ребра — соединения между ними. Можно ли создать электронную схему, в которой соединения не пересекаются? Решение этой задачи выходит за рамки нашей книги.

Задача от четырех красках. Можно ли раскрасить планарный граф так, чтобы ни одна пара смежных вершин не была окрашена в одинаковый цвет? Например, граф, изображенный на рис. 13.11, является планарным, поскольку его ребра не пересекаются. Раскрасить этот граф так, чтобы выполнялись условия задачи, можно только тремя красками. Вершины *a*, *c* и *g* можно закрасить красным, *b*, *d*, *f* и *i* — синим, а вершину *e* — зеленым цветом.

Эта задача имеет положительное решение, но доказать это очень трудно. Она была поставлена более 100 лет назад и решена лишь с помощью компьютера в 1970-х годах.

Резюме

1. Графы реализуются в виде матрицы смежности или списка смежности. Каждая из этих реализаций имеет свои преимущества и недостатки. Выбор зависит от конкретного приложения.
2. Поиск в графе представляет собой важный пример использования стеков и очередей. Поиск в глубину является алгоритмом обхода графа, в котором используется стек. Он проходит в глубину графа, а затем выполняет откат. Поиск в ширину для отслеживания пройденных вершин использует очередь. Он проходит по всем смежным вершинам, а затем углубляется в граф.
3. Топологическая сортировка порождает линейный порядок следования вершин в направленном графе без циклов. Вершина x предшествует вершине y, если существует ориентированное ребро, выходящее из вершины x в вершину y.
4. Дерево представляет собой связный неориентированный граф без циклов. Остовное дерево связного неориентированного графа является подграфом, который содержит все вершины графа и те ребра, которые образуют дерево. Алгоритмы обхода графа в глубину и в ширину порождают остовные деревья поиска в глубину и остовные деревья поиска в ширину, соответственно (DFS и BFS spanning trees).
5. Минимальное остовное дерево взвешенного неориентированного графа является остовным деревом, в котором сумма ребер минимальна. Хотя граф может иметь несколько минимальных остовных деревьев, их общая сумма весов всегда одинакова.
6. Кратчайший путь между двумя вершинами во взвешенном направленном графе представляет собой путь, имеющий наименьшую сумму весов ребер.
7. Простая цепь Эйлера в неориентированном графе представляет собой цикл, начинающийся в вершине v, проходящий по каждому ребру графа только один раз и заканчивающийся в вершине v.
8. Простая цепь Гамильтона в неориентированном графе является циклом, начинающимся в вершине v, проходящим через каждую вершину графа только один раз и заканчивающимся в вершине v.

Предупреждения

1. Выполняя поиск в графе, имейте в виду, что алгоритм может зайти в тупик. Например, нужно исключить возможность зацикливания и предусмотреть откаты.

Вопросы для самопроверки

1. Опишите графы, изображенные на рис. 13.32. Являются ли они ориентированными? Связными? Совершенными? Взвешенными?
2. Используйте стратегию поиска в глубину и в ширину для обхода графа, представленного на рис. 13.32, *а*, начиная с вершины 0. Перечислите вершины в порядке их обхода.
3. Напишите матрицу смежности для графа, изображенного на рис. 13.32, *а*.

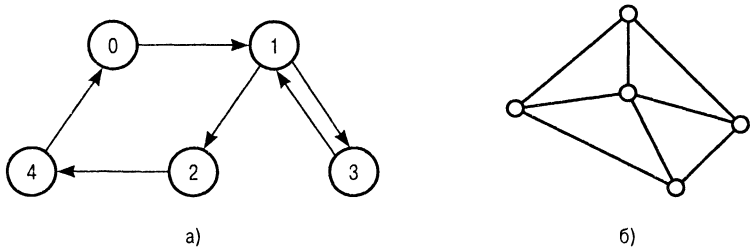


Рис. 13.32. Графы, упоминающиеся в вопросах 1, 2 и 3

4. Добавьте ребро в ориентированный граф, показанный на рис. 13.14, выходящее из вершины *d* в вершину *b*. Перечислите вершины нового графа в соответствии со всеми возможными вариантами топологического порядка.
5. Может ли граф, состоящий из 5 вершин и 4 четырех ребер, содержать простой цикл? Обоснуйте свой ответ.
6. Нарисуйте остовное дерево поиска в глубину с корнем в вершине 0 для графа, изображенного на рис. 13.33.

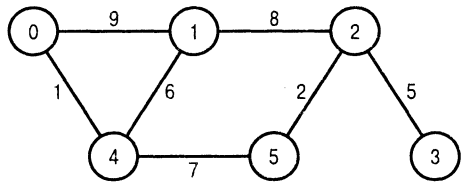


Рис. 13.33. Граф, упоминающийся в вопросах 6 и 7, а также в упражнениях 1 и 3

7. Нарисуйте минимальное остовное дерево с корнем в вершине 0 для графа, изображенного на рис. 13.33.
8. Какой путь из вершины 0 в каждую из вершин графа, изображенного на рис. 13.24, *а*, является кратчайшим? (Учтите веса этих путей, указанные на рис. 13.25.)

Упражнения

Алгоритм обхода графа должен выполняться в указанном порядке следования вершин. Если порядок не указан, читатели могут выбирать его произвольно.

1. Напишите матрицу смежности и список смежности для следующих графов.

   1.1. Взвешенный граф, изображенный на рис. 13.33.

   1.2. Ориентированный граф, изображенный на рис. 13.34.

2. Покажите, что список смежности, представленный на рис. 13.8, *б*, занимает меньше памяти, чем матрица смежности, показанная на рис. 13.6, *б*.
3. Примените стратегии поиска в глубину и в ширину для обхода графа, изображенного на рис. 13.33, начиная с вершины 0, а также для обхода графа, показанного на рис. 13.35, начиная с вершины *а*.
4. Напишите псевдокод модифицированного алгоритма поиска в глубину, позволяющий обнаруживать циклы.

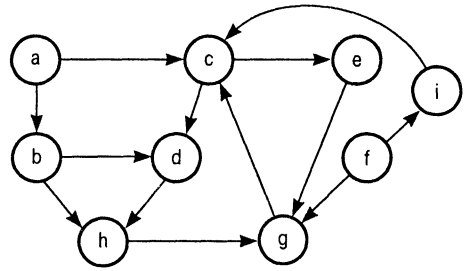


Рис. 13.34. Граф из упражнения 1

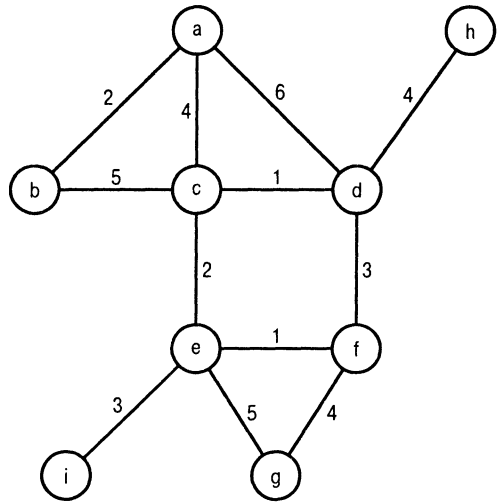


Рис. 13.35. Граф, упоминающийся в упражнениях 3 и 8

5. Выполните трассировку алгоритма топологической сортировки с помощью поиска в глубину `topSort2` и напишите, в каком топологическом порядке будут записаны вершины графа, изображенного на рис. 13.36.
6. Модифицируйте алгоритм топологической сортировки `topSort1`, удаляя не преемников, а предшественников. Выполните трассировку нового алгоритма на примере графа, изображенного на рис. 13.36.
7. Выполните трассировку алгоритмов построения остовных деревьев поиска в глубину и в ширину с корнем в вершине *a* для графа, представленного на рис. 13.11. Покажите, что полученные остовные деревья совпадают с деревьями, изображенными на рис. 13.20 и 13.21, соответственно.
8. Нарисуйте остовные деревья поиска в глубину и в ширину с корнем в вершине *a* для графа, представленного на рис. 13.35. Затем нарисуйте минимальное остовное дерево этого графа с корнем в вершине *a*.
9. Напишите псевдокод итеративного алгоритма, создающего остовное дерево поиска в глубину для заданного графа. Используйте в качестве основы алгоритм `dfs`.

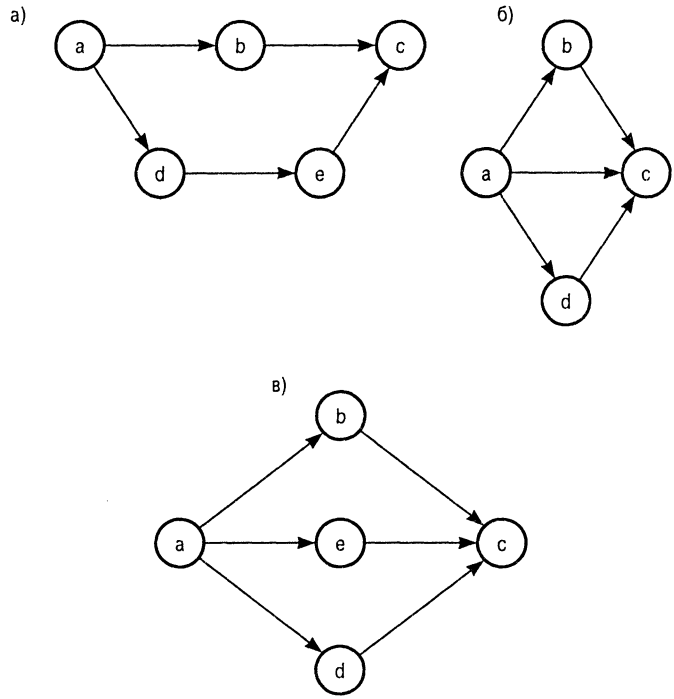


Рис. 13.36. Граф, упоминающийся в упражнениях 5 и 6

10. Нарисуйте минимальное остовное дерево графа, изображенного на рис. 13.22, начиная с указанной вершины.

    10.1. Вершина *g*.

    10.2. Вершина *c*.

11. Выполните трассировку алгоритма поиска кратчайшего пути в графе, изображенном на рис. 13.37, начиная с вершины 0.

12. Реализуйте алгоритм поиска кратчайшего пути в графе на языке C++/. Как модифицировать этот алгоритм, чтобы началом могла быть любая вершина?

13. Найдите простую цепь Эйлера в графе, изображенном на рис. 13.38. Почему это возможно?

14. Докажите, что связный неориентированный граф, состоящий из n вершин и более чем из $n-1$ ребра, должен содержать по крайней мере один цикл. (См. свойство 3 из раздела "Остовные деревья".)

15. Докажите, что алгоритм обхода графа посещает каждую вершину, только если граф является связным, независимо от вершины, с которой начинается обход.

16. Хотя алгоритм поиска в глубину имеет простую рекурсивную форму, рекурсивный алгоритм обхода в ширину не так очевиден.

    16.1. Докажите это утверждение.

    16.2. Напишите псевдокод рекурсивной версии алгоритма обхода в ширину.

17. Используя пошаговую индукцию алгоритма Дейкстры, докажите, что его инвариант цикла выполняется.

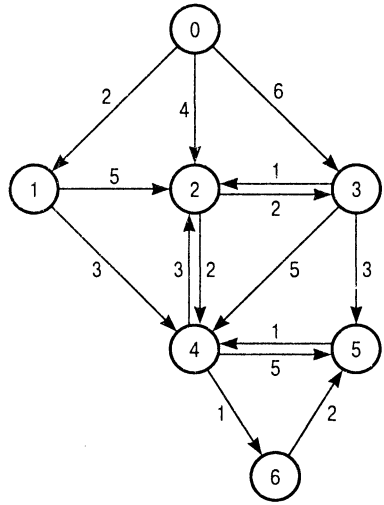


Рис. 13.37. Граф из упражнения 11

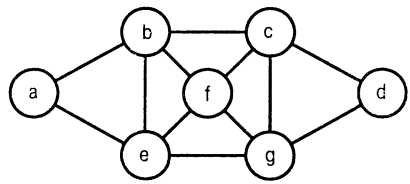


Рис. 13.38. Граф из упражнения 13

Задания по программированию

1. Реализуйте абстрактный граф в виде класса на языке C++, сначала используя матрицу смежности, а затем — список смежности. Учтите, что граф может быть ориентированным и не ориентированным, взвешенным и не взвешенным, связным и не связным. Включите в класс алгоритмы обхода в глубину и в ширину.
2. Дополните решение задания 1 абстрактными операциями `isConnected` и `hasCycle`. Кроме того, включите в класс операции, выполняющие топологическую сортировку направленного графа без циклов, создающие остовные деревья поиска в глубину и в ширину, а также формирующие минимальное остовное дерево для связного неориентированного графа.
3. Задача об авиарейсах рассматривалась в заданиях 9, 10 и 11 из главы 6. Модифицируйте эти постановки задач, реализовав АТД "Карта полетов" с помощью графа, и напишите соответствующий класс на языке C++.

Методы работы с внешними запоминающими устройствами

В этой главе ...

Внешние запоминающие устройства
Сортировка данных во внешнем файле
Внешние таблицы
- Индексирование внешнего файла
- Внешнее хэширование
- B-деревья
- Алгоритмы обхода
- Множественная индексация

Резюме
Предупреждения
Вопросы для самопроверки
Упражнения
Задания по программированию

Введение. Изучая все предыдущие реализации абстрактной таблицы, мы предполагали, что данные хранятся в оперативной памяти компьютера. Однако во многих практических приложениях таблицы бывают очень большими и не помещаются в оперативной памяти. В таких ситуациях их необходимо хранить на внешних запоминающих устройствах, например на диске.

В этой главе мы рассматриваем методы управления данными, хранящимися на внешнем запоминающем устройстве, используя файл прямого доступа в качестве модели. Описывается сортировка данных во внешнем файле, в частности метод сортировки слиянием, а также методы поиска, основанные на обобщенном хэшировании и применении деревьев поиска.

Внешние запоминающие устройства

Внешние запоминающие устройства сохраняют информацию после выполнения программы

Файл, из которого программа на языке C++ считывает данные и в который она записывает результаты вычислений, представляет собой типичный образец внешнего запоминающего устройства. Работая с текстовым процессором и выбирая, к примеру, опцию *Save*, вы записываете документ в файл. Это позволяет завершить работу программы и вернуться к документу в дальнейшем, прочитав его из файла. В этом заключается одно из основных преимуществ внешних запоминающих устройств: они сохраняют информацию и после завершения работы программы. В некотором смысле такие устройства можно назвать "постоянной" памятью компьютера, а оперативную память — временной.

Как правило, объем внешней памяти больше оперативной

Еще одно преимущество внешних запоминающих устройств заключается в том, что объем внешней памяти, как правило, намного превышает объем оперативной памяти компьютера. Если таблица состоит из миллиона записей среднего размера, она вряд ли целиком поместится в оперативной памяти, хотя ее легко разместить на внешнем диске. Следовательно, работая с такими большими таблицами, мы не можем просто целиком считывать их в оперативную память, обрабатывать и записывать обратно. Вместо этого нам нужны методы для работы с данными непосредственно на внешних запоминающих устройствах.

Как правило, файлы имеют последовательный или прямой доступ. Чтобы получить доступ к заданной позиции в **файле последовательного доступа** (sequential access file), курсор файла придется переместить по всем позициям, предшествующим заданной. Этим файлы последовательного доступа напоминают связанные списки. Для того чтобы обратиться к конкретному узлу списка, необходимо выполнить обход списка с самого начала. **Файл прямого доступа** (direct access file), наоборот, позволяет непосредственно обращаться к заданной позиции. Файл прямого доступа напоминает массив, обеспечивающий непосредственное обращение к данным, записанным в ячейке `data[i]`. Для этого не нужно перебирать все предшествующие элементы массива.

При работе с таблицами необходим прямой доступ

Без прямого доступа было бы невозможно обеспечить эффективное выполнение операций над таблицами, хранящимися на внешнем запоминающем устройстве. Многие языки программирования, включая язык C++, поддерживают как прямой, так и последовательный доступ к записям файлов. Однако, чтобы сохранить независимость от конкретного языка, мы сконструируем модель файлов прямого доступа и проиллюстрируем их реализацию в языке, который изначально этот механизм доступа не поддерживает. Эта модель упрощает реальные проблемы, но позволяет обсудить все важные детали.

Представим себе, что память компьютера разделена на две части: внутреннюю и внешнюю, как показано на рис. 14.1. Допустим, что сама программа и данные, не записанные в файл, хранятся в оперативной памяти компьютера. Постоянные файлы, необходимые для работы компьютера, находятся на внешнем запоминающем устройстве. Предположим также, что это устройство представляет собой диск (хотя некоторые системы используют другие механизмы).

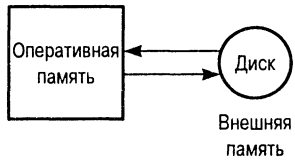


Рис. 14.1. Внутренняя и внешняя память

Файл состоит из **записей** (records). Записью может быть что угодно, начиная с обычного числа, например, целого, и заканчивая сложными структурами, скажем, учетными карточками сотрудников. Для простоты будем считать, что все записи, хранящиеся в файле, однотипны.

Записи файла группируются в **блоки** (blocks), как показано на рис. 14.2. Размер блока, т.е. количество бит, из которых состоят данные, определяется как конфигурацией, так и программным обеспечением компьютера. Как правило, пользовательские программы не управляют этим параметром. Следовательно, количество записей в блоке зависит от размера записей, хранящихся в файле. Например, в файле, состоящем из целых чисел, количество записей в блоке больше, чем в файле, в котором хранятся записи о сотрудниках.

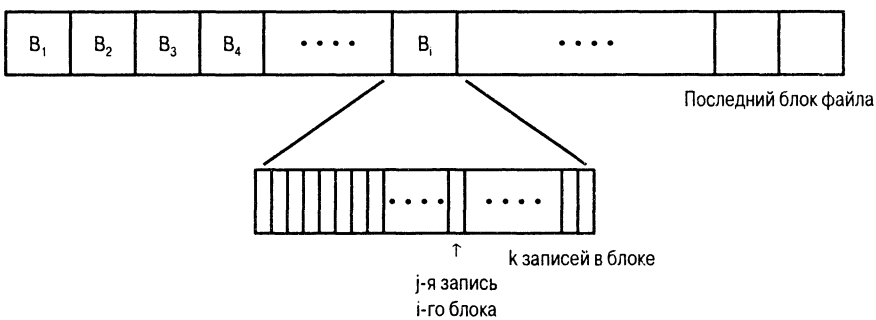


Рис. 14.2. Файл, разделенный на блоки записей

Аналогично элементам массива, блоки записей можно пронумеровать в порядке возрастания. Для чтения блока, хранящегося в файле прямого доступа, достаточно указать его номер. Аналогично выполняется операция записи блока в файл. В этом смысле файлы прямого доступа напоминают массивы массивов, в которых каждый блок подобен отдельному элементу массива, который, в свою очередь, сам является массивом, состоящим из нескольких записей.

При работе с файлами прямого доступа вводятся и выводятся блоки, а не записи

Модель прямого доступа характеризуется тем, что ввод и вывод выполняются на уровне блоков, а не записей. Иными словами, можно считать и записать блоки записей, но нельзя считать или записать отдельную запись. Операции чтения и записи блоков называются **блочным доступом** (block access).

В дальнейшем будем считать, что все операции выполняются над блоками. Оператор

Буфер предназначен для временного хранения данных

```
buf.readBlock(dataFile, i)
```

считывает *i*-й блок из файла *dataFile* и помещает его в объект *buf*. В этом объекте должно умещаться столько записей, сколько хранится в блоках файла *dataFile*. Например, если каждый блок содержит 100 учетных записей, то объект *buf* должен иметь размер, позволяющий хранить все эти записи. Объект *buf* называется буфером (buffer) и представляет собой место для временного хранения данных.

Поскольку операционная система позволяет считывать блок в буфер, программа может обрабатывать — например, проверять или модифицировать — записи блока. Кроме того, поскольку записи, хранящиеся в объекте *buf*, представляют собой лишь копии записей из файла *dataFile*, то модифицировав содержимое буфера, программа должна записать его обратно в файл. Будем считать, что оператор

```
buf.writeBlock(dataFile, i)
```

записывает содержимое объекта *buf* в *i*-й блок файла *dataFile*. Если файл *dataFile* состоит из *n* блоков, оператор

```
buf.writeBlock(dataFile, n+1)
```

добавит в файл *dataFile* новый блок. Таким образом, файл может динамически увеличиваться.

Напомним, что операции ввода и вывода относятся только к целым блокам, а не к отдельным записям. Следовательно, даже если нужно изменить одну-единственную запись, хранящуюся в файле, придется считать и записать целый блок. Допустим, зарплата сотрудника по фамилии Смит была повышена на 1000 долларов. Если запись о Смите находится в *i*-м блоке (как это узнать, мы объясним позднее), нужно выполнить следующие шаги.

Обновление записи, хранящейся в блоке

```
// Считываем i-й блок из файла dataFile
// в буфер buf
buf.readBlock(dataFile, i)

Найти элемент buf.getRecord(j), содержащий запись
   с поисковым ключом "Смит"

(buf.getRecod(j)).setSalary((buf.getRecord(j)).getSalary()+1000)

// Увеличиваем оклад в соответствующей записи.
// Записываем измененный блок обратно в файл dataFile

buf.writeBlock(dataFile, i)
```

Следует сократить количество обращений к блокам

Обычно операции чтения и записи блока выполняются намного дольше, чем его обработка в оперативной памяти.[1] Например, как правило, проверка каждой записи, хранящейся в буфере, выполняется быстрее, чем считывание блока в буфер. Так, в предыдущем псевдокоде прежде, чем записывать содержимое буфера в файл, следует обработать как можно больше за-

[1] Скорость загрузки и выгрузки буфера отличается от скорости обработки записей. (Поэтому буфер часто используют для синхронизации двух процессов, выполняющихся с разной скоростью.)

писей. Поскольку обработка данных, записанных в оперативную память, выполняется очень быстро, длительностью этого процесса можно пренебречь.

Интересно, что в нескольких языках программирования, в том числе в языке C++, существуют команды, имитирующие доступ к отдельным записям. Однако, как правило, на самом деле система выполняет операции чтения и записи блоков, скрывая этот факт от программы. Рассмотрим следующий пример.

```
rec.readRecord(dataFile, i)
// Считываем i-ю запись из файла dataFile в объект rec.
```

Вероятно, система сначала считает весь блок, содержащий *i*-ю запись, и лишь затем извлечет из него нужную информацию. Таким образом, наша модель ввода-вывода вполне соответствует реальности.

Время доступа к файлу является основным фактором, от которого зависит эффективность алгоритма

В большинстве приложений, связанных с использованием внешних запоминающих устройств, время доступа к блоку является доминирующим фактором. В остальной части главы мы рассмотрим алгоритмы сортировки и поиска данных на внешних запоминающих устройствах. Целью этих алгоритмов является сокращение количества обращений к блокам.

Сортировка данных во внешнем файле

В этом разделе мы рассмотрим задачу сортировки данных, хранящихся во внешнем файле.

Задача сортировки

Внешний файл содержит 1600 записей о сотрудниках. Необходимо упорядочить эти записи по номеру карточки социального страхования. Каждый блок содержит 100 записей, следовательно, файл содержит 16 блоков B_1, B_2, ..., B_{16}. Допустим, что оперативной памяти достаточно для одновременной обработки лишь 300 записей (т.е. трех блоков).

Сортировка файла, на первый взгляд, может показаться несложной задачей, поскольку мы уже рассмотрели несколько алгоритмов сортировки. Однако существует фундаментальное отличие, заключающееся в том, что файл слишком велик и не помещается в оперативной памяти целиком. Это ограничение создает определенные трудности, поскольку изученные нами алгоритмы сортировки основывались на предположении, что все данные одновременно находятся в оперативной памяти (например, заполняют массив). К счастью, эту трудность легко преодолеть с помощью алгоритма сортировки слиянием.

В основе алгоритма сортировки слиянием лежит предположение, что мы можем легко объединить два упорядоченных сегмента записей, например, два массива, в третий упорядоченный сегмент. Например, если обозначить через S_1 и S_2 два упорядоченных сегмента записей, то на первом шаге слияния следует сравнить первые записи каждого сегмента и выбрать запись, содержащую меньший поисковый ключ. Если выбрана запись из сегмента S_1, то на следующем шаге вторая запись сегмента S_1 сравнивается с первой записью сегмента S_2. Этот процесс продолжается до тех пор, пока не будут проверены все записи. Основная идея заключается в том, что на каждом шаге слияния проверяются только *первые элементы* (leading edges) каждого сегмента.

Это позволяет применять для упорядочения файлов слегка измененный алгоритм сортировки слиянием. Допустим, что 1600 записей, подлежащих сортировке, хранятся в файле *F*, причем сам файл изменять нельзя. Создадим два вспомогательных файла: F_1 и F_2. Один из этих файлов по окончании алгоритма будет со-

держать упорядоченные записи. Алгоритм состоит из двух этапов. На первом этапе сортируется каждый блок записей, а на втором — выполняется серия слияний.

Этап 1. Считываем блок из файла F в оперативную память, упорядочиваем его записи, пользуясь обычными алгоритмами сортировки, записываем упорядоченный блок в файл F_1, а затем считываем следующий блок из файла F. После обработки всех 16 блоков, содержащихся в файле F, файл F_1 содержит 16 **упорядоченных отрезков** (sorted runs), состоящих из блоков, в которых хранятся упорядоченные записи, как показано на рис. 14.3, *а*.

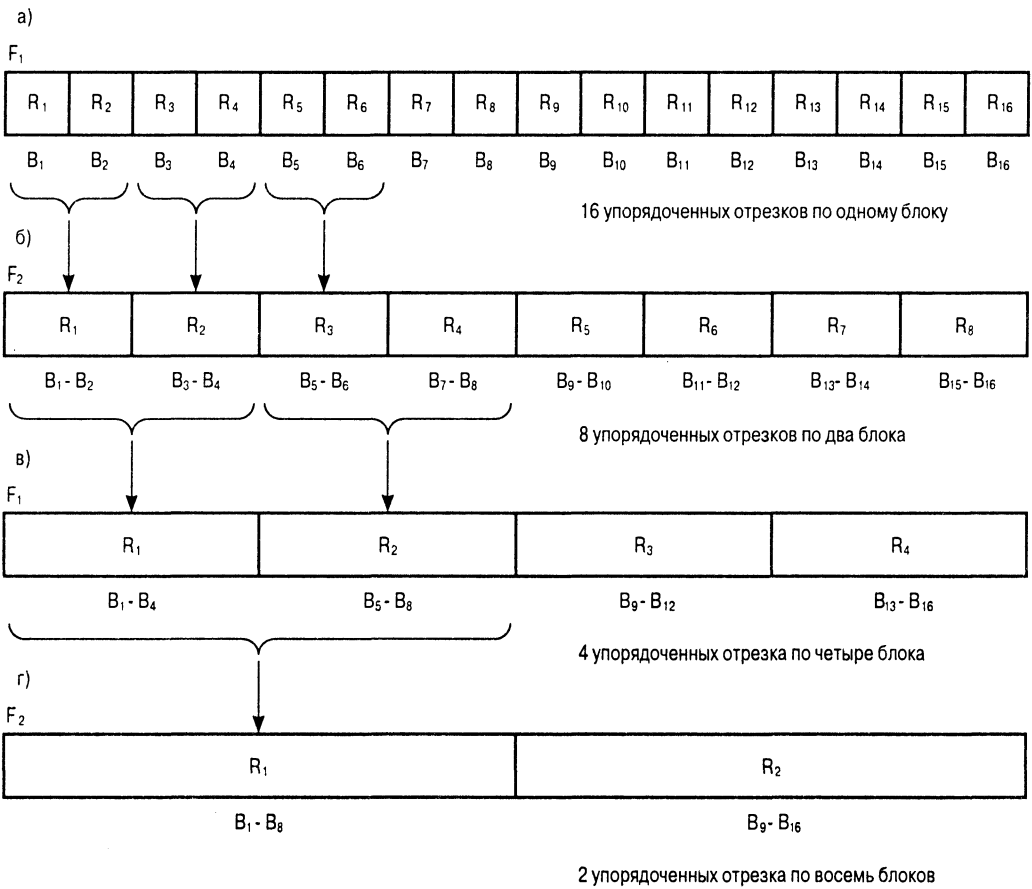


Рис. 14.3. Внешняя сортировка слиянием: а) файл F_1 содержит 16 упорядоченных отрезков, каждый из которых состоит из одного блока; б) файл F_2 содержит 8 упорядоченных отрезков, каждый из которых состоит из двух блоков; в) файл F_1 содержит 4 упорядоченных отрезка, каждый из которых состоит из четырех блоков; г) файл F_2 содержит 2 упорядоченных отрезка, каждый из которых состоит из восьми блоков

Этап 2. Второй этап представляет собой последовательность слияний. На каждом шаге объединяется пара упорядоченных отрезков и возникают более крупные упорядоченные отрезки. Количество блоков в каждом отрезке удваивается, а общее количество отрезков, соответственно, вдвое сокращается. Например, как показано на рис. 14.3, *б*, на первом шаге объединяются друг с другом восемь пар

упорядоченных отрезков, хранящихся в файле F_1 (R_1 — с R_2, R_3 — с R_4, ... , R_{15} — с R_{16}). Они образуют восемь новых упорядоченных отрезков, по два блока каждый. Эти отрезки записываются в файл F_2. На следующем шаге объединяются друг с другом четыре пары упорядоченных отрезков, хранящихся в файле F_2 (R_1 — с R_2, R_3 — с R_4, ... , R_7 — с R_8). Они образуют четыре новых упорядоченных отрезка, по четыре блока каждый. Эти отрезки записываются в файл F_1, как показано на рис. 14.3, *в*. Затем две пары упорядоченных отрезков, хранящихся в файле F_1, объединяются и образуют два новых упорядоченных отрезка. Эти отрезки записываются в файл F_2 (рис. 14.3, *г*). На последнем шаге два упорядоченных отрезка сливаются в один. Этот отрезок записывается в файл F_1. Теперь файл F_1 содержит упорядоченные записи исходного файла.

Слияние упорядоченных отрезков на втором этапе

Остается уточнить, каким образом осуществляется слияние упорядоченных отрезков на втором этапе. Оперативной памяти хватает для одновременной обработки всего 300 записей. Однако на последних шагах второго этапа отрезки становятся намного длиннее, поэтому их слияние выполняется по частям. Чтобы осуществить слияние, оперативную память нужно разделить на три массива *in1*, *in2* и *out*, каждый из которых способен хранить 100 записей (один блок). Блоки файла записываются в массивы *in1* и *in2*, а затем объединяются и копируются в массив *out*. Если массив *in1* или *in2* оказался исчерпанным, т.е. все его элементы скопированы в массив *out*, в него записывается новый блок из файла. Если массив *out* полон, упорядоченный отрезок, содержащийся в нем, выгружается в один из файлов.

Рассмотрим первый шаг слияния. Начнем с пары R_1 и R_2, т.е. с первого и второго блоков файла F_1 соответственно. (См. рис. 14.3, *а*.) Поскольку на первом шаге каждый отрезок состоит из одного блока, отрезок, полученный в результате слияния, вполне умещается в одном из массивов *in1* или *in2*. Следовательно, мы можем записать отрезки R_1 и R_2 в массивы *in1* и *in2*, а затем объединить их в массиве *out*. Однако, хотя результат слияния массивов *in1* и *in2* состоит из двух блоков (200 записей), массив *out* может хранить только один блок (100 записей). Таким образом, когда в процессе слияния массив *out* заполнится полностью, его содержимое будет скопировано в первый блок файла F_2, как показано на рис. 14.4. Затем процесс слияния массивов *in1* и *in2* возобновляется. Массив *out* вновь заполнится уже после того, как массивы *in1* и *in2* окажутся исчерпанными. Теперь содержимое массива *out* следует выгрузить во второй блок файла F_2. Остальные семь пар отрезков из файла *F* объединяются аналогично. Результат этих слияний записывается в файл F_2.

Первый шаг слияния в принципе немного проще, чем остальные, поскольку вначале отрезки состоят лишь из одного блока, и, следовательно, каждый из них может целиком поместиться в массивах *in1* или *in2*. Что делать в этих ситуациях? Рассмотрим, например, слияние отрезков, состоящих их четырех блоков, в отрезки, состоящие из восьми блоков. (См. рис. 14.3, *в*.) Первая пара этих отрезков состоит из блоков 1 – 4 и 5 – 8, записанных в файл F_1.

Алгоритм считывает первый блок отрезка R_1, т.е блок B_1, в массив *in1*, а затем считывает первый блок отрезка R_2, т.е. блок B_2, в массив *in2*, как показано на рис. 14.4, *б*. Затем, как и раньше, алгоритм объединяет массивы *in1* и *in2* в массив *out*. Сложность здесь заключается в том, что после перемещения всех записей из массива *in1* или *in2* необходимо считать следующий блок из соответствующего отрезка. Например, если сначала исчерпывается массив *in2*, нужно считать в массив *in2* блок из отрезка R_2, т.е. блок B_6, а затем продолжить слияние. Следовательно, алгоритм должен определять, какой из массивов *in1* или *in2* оказался исчерпанным, а также следить за процессом заполнения массива *out*.

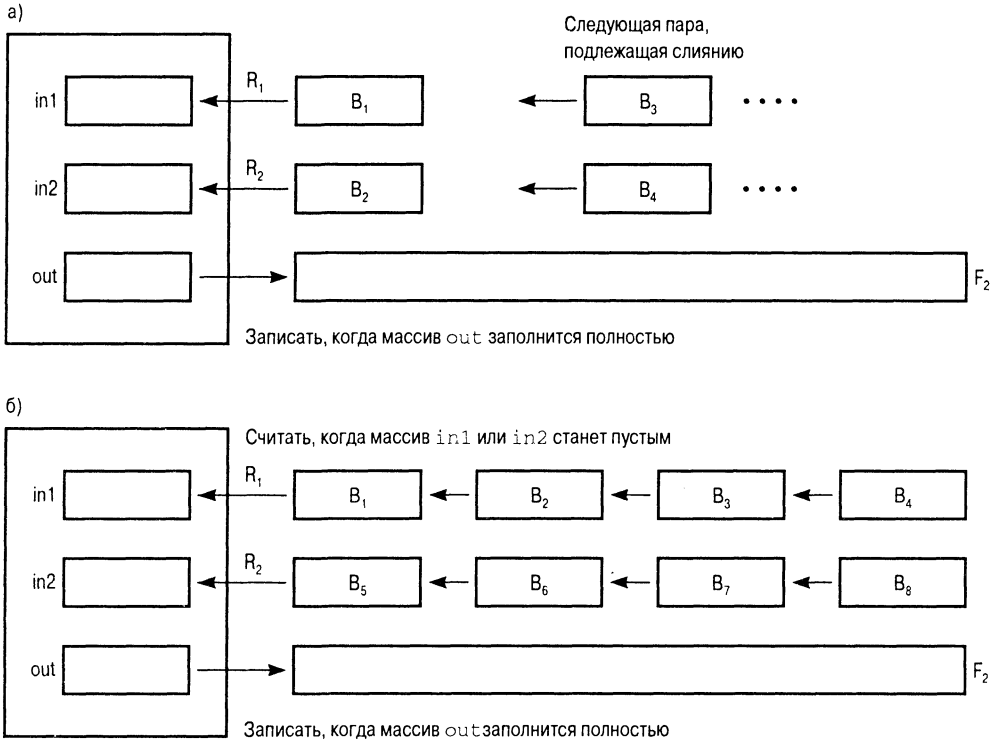


Рис. 14.4. Слияние упорядоченных отрезков: а) слияние отдельных блоков; б) слияние длинных отрезков

Рассмотрим описание алгоритма слияния произвольных отрезков R_i и R_j, хранящихся в файлах F_1 и F_2.

Псевдокод алгоритма слияния двух упорядоченных отрезков

```
Считать первый блок отрезка Ri в массив in1
Считать первый блок отрезка Rj в массив in2

while (либо массивы in1, либо in2 еще не исчерпаны)
{
   Выбрать наименьшую "ведущую" запись в массиве in1 или in2
     и поместить ее в следующую позицию массива out
     (если один из массивов исчерпан, выбрать ведущую
     запись из другого массива)

   if (массив out полон)
   Записать его содержимое в следующий блок файла F2

   if (массив in1 исчерпан, а в отрезке Ri остались блоки)
   Считать следующий блок в массив in1

   if (массив in2 исчерпан, а в отрезке Rj остались блоки)
   Считать следующий блок в массив in2

} // Конец оператора while
```

Ниже приведен псевдокод алгоритма внешней сортировки. Обратите внимание, что он использует операции *readBlock* и *writeBlock*, введенные в предыдущем разделе, а функция *copyFile* предназначена для копирования файла. Для простоты будем предполагать, что количество блоков в файле задается степенью двойки. Это позволяет образовывать пары на каждом шаге алгоритма слияния, избегая специальной проверки конца файла, которая лишь осложняет процесс. Кроме того, алгоритм использует два временных файла и копирует окончательный результат из временного файла в исходный.

Псевдокод функции сортировки

```
externalMergesort(
          in unsortedFileName:String,
          in sortedFileName:String)
// Упорядочивает файл, используя внешнее слияние.
// Предусловие: unsortedFileName — имя внешнего файла,
// подлежащего сортировке, sortedFileName — имя, которое
// функция присваивает упорядоченному файлу.
// Постусловие: вновь созданный файл с именем sortedFileName
// упорядочен. Исходный файл остается без изменений.
// Оба файла закрыты.
// Вызываемые функции: blockSort, mergeFile и copyFile.
// Упрощающие предположения: количество блоков
// в неупорядоченном файле равно степени двойки.

   Связать имя unsortedFileName с переменной inFile,
   а имя sortedFileName — с переменной outFile

   // Этап 1: сортируем блок за блоком и подсчитываем
   // их количество

   blockSort(inFile, tempFile1, numberOfBlocks)
   // Этап 2: слияние отрезков, имеющих размеры 1, 2, 4,
   // 8,..., numberOfBlocks/2 (используются два временных
   // файла и переключатель файлов toggle)

   toggle = 1
   for (size = 1 до numberOfBlocks/2 с шагом size)
   {
     if (toggle == 1)
        mergeFile(tempFile1, tempFile2, size, numberOfBlocks)
     else
        mergeFile(tempFile1, tempFile2, size, numberOfBlocks)

     toggle = -toggle
   } // Конец оператора for

   // Копируем временный файл в файл outFile
   if (toggle == 1)
      copyFile(tempFile1, outFile)
   else
      copyFile(tempFile2, outFile)
```

Обратите внимание, что функция *externalMergesort* вызывает функции *blockSort* и *mergeFile*, которые, в свою очередь, вызывают функцию *mergeRuns*. Псевдокод этих функций имеет следующий вид.

```
blockSort(in inFile:File, in outFile:File,
          in numberOfBlocks:integer)
// Упорядочивает каждый блок записей в файле.
// Предусловие: переменная inFile задает имя файла,
// подлежащего сортировке.
// Постусловие: файл, имя которого задано переменной outFile,
// содержит блоки файла inFile. Каждый блок упорядочен.
// Переменная numberOfBlocks задает количество обработанных
// блоков. Оба файла закрыты.
// Вызываемые функции: для обеспечения прямого доступа
// при вводе и выводе данных используются функции
// readBlock и writeBlock, функция sortBuffer применяется
// для сортировки массива.

   Подготовить файл inFile для ввода
   Подготовить файл outFile для вывода

   numberOfBlocks = 0
   while (в файле inFile остались непрочитанные блоки)
   {
      ++numberOfBlocks
      buffer.readBlock(inFile, numberOfBlocks)
      sortArray(buffer); // Внутренняя сортировка

      buffer.writeBlock(outFile, numberOfBlocks)
   } // Конец оператора while

   Close inFile and outFile
// Конец функции blockSort

mergeFile(in inFile:File, in outFile:File,
          in runSize:integer, in numberOfBlocks:integer)
// Объединяет блоки одного файла в другой.
// Предусловие: inFile — имя внешнего файла, содержащего
// numberOfBlocks упорядоченных блоков, объединенных
// в отрезки по runSize блоков в каждом.
// Постусловие: файл с именем outFile содержит объединенные
// отрезки файла inFile. Оба файла закрыты.
// Вызываемая функция: mergeRuns.

   Подготовить файл inFile для ввода
   Подготовить файл outFile для вывода

   for (next = 1 до numberOfBlocks с шагом 2*runsize
   {
      // Инвариант: отрезки файла outFile упорядочены
      mergeRuns(inFile, outFile, next, runSize)
   } // Конец оператора for
   закрыть файлы inFile и outFile
// Конец файла mergeFile

mergeRuns(in fromFile:File, in toFile:File,
          in start:integer, in size:integer)
// Объединяет два последовательно расположенных
// упорядоченных отрезка в новом файле.
```

```
// Предусловие: аргумент fromFile задает имя внешнего файла,
// состоящего из упорядоченных отрезков, аргумент tofile
// задает имя выходного файла. Переменная start задает
// количество блоков в первом отрезке файла fromFile,
// подлежащем сортировке. Этот отрезок состоит из
// size блоков.
//    Отрезок 1: от первого блока до блока start + size - 1.
//    Отрезок 2: от блока start + size до блока
//               start + (2 * size) - 1.
// Постусловие: отрезки файла fromFile, подлежащие
// объединению, добавлены в файл to toFile. Файлы
// остаются открытыми.

   // Инициализируем входные буферы для отрезков 1 и 2
   in1.readBlock(fromFile, первый блок отрезка 1)
   in2.readBlock(fromFile, первый блок отрезка 2)

   // Выполняем слияние, пока один из отрезков не закончится.
   // Как только входной буфер исчерпан, считываем следующий
   // блок. Как только выходной     буфер полностью заполнен,
   // выгружаем его в файл.
   while (ни один из отрезков не закончен)
   {
      Выбрать наименьший ведущий элемент в массивах in1 и in2
         и поместить его на следующую позицию массива out

      if (массив out полон)
          out.writeBlock(toFile, следующий блок файла toFile)

      if (массив in1 исчерпан, а в отрезке 1 остались
          непрочитанные блоки)
          in1.readBlock(fromFile, следующий блок отрезка 1)

      if (массив in2 исчерпан, а в отрезке 2 остались
          непрочитанные блоки)
          in2.readBlock(fromFile, следующий блок отрезка 2)
   } // Конец оператора while

   // Диагностическое утверждение: только один
   // из отрезков полон
   // Добавляем оставшиеся записи в буфер вывода
   // и выгружаем его в файл

   while (массив in1 не исчерпан)
   // Инвариант: массив out упорядочен
   Поместить следующий элемент массива in1 на следующую
      позицию массива out

   while (массив in2 не исчерпан)
   // Инвариант: массив out упорядочен.
   Поместить следующий элемент массива in2 на следующую
      позицию массива out

   out.writeBlock(toFile, следующий блок файла toFile)
```

```
    // Переписываем оставшиеся полные блоки

    while (в отрезке 1 остались непрочитанные блоки)
    {
       // Инвариант: каждый блок файла toFile упорядочен
       in1.readBlock(fromFile, следующий блок отрезка 1)
       in1.writeBlock(toFile, следующий блок файла toFile)
    } // Конец оператора while

    while (в отрезке 1 остались непрочитанные блоки)
    {
       // Инвариант: каждый блок файла toFile упорядочен
       in2.readBlock(fromFile, следующий блок отрезка 2)
       in2.writeBlock(toFile, следующий блок файла toFile)
    } // Конец оператора while
// Конец функции mergeRuns
```

Внешние таблицы

В этом разделе обсуждаются способы организации записей на внешнем запоминающем устройстве, позволяющие эффективно реализовать операции над абстрактной таблицей, например, извлечение, вставку, удаление элементов и обход таблицы. В разделе затрагиваются лишь самые поверхностные вопросы, причем некоторые проблемы, связанные с этой темой, 2-3 дерево и хэширование, уже рассматривались нами в главе 12.

Простая реализация внешней таблицы: записи хранятся в порядке следования ключей

Допустим, элементы таблицы, которыми являются записи, хранятся в файле прямого доступа. Этот файл разделен на блоки, как описано выше. Один из простейших способов организации таблицы заключается в записи ее элементов в порядке следования поисковых ключей, причем сортировка файла осуществляется с помощью алгоритма внешней сортировки слиянием, рассмотренного в предыдущем разделе. Поскольку файл упорядочен, его записи легко обойти с помощью следующего алгоритма.

Упорядоченный обход

```
traverseTable(in dataFile:File,
              in numberOfBlocks:integer,
              in recordsPerBlock:integer,
              in visit:function)
// Обходит упорядоченный файл dataFile в порядке следования ключей,
// вызывая функцию visit() по одному разу для каждого элемента.

    // Считать каждый блок файла dataFile во внутренний
    // буфер buf
    for (blockNumber = 1 до numberOfBlocks)
    {
       buf.readBlock(dataFile, blockNumber)
      // Посетить каждый элемент блока
      for (recordNumber = 1 до recordsPerBlock)
         Посетить запись buf.getRecord(recordNumber-1)
    } // Конец оператора for
```

Чтобы применить операцию *tableRetrieve* к упорядоченному файлу, можно использовать алгоритм бинарного поиска.

```
tableRetrieve(in dataFile:File, in recordsPerBlock:integer,
              in first:integer, in last:integer,
              in searchKey:KeyItemType,
              out tableItem:TableItemType):boolean
// Обойти блоки файла dataFile от начала до конца,
// скопировать в переменную tableItem запись, поисковый
// ключ которой равен значению аргумента searchKey,
// и вернуть значение true. Если такого элемента в файле
// нет, вернуть значение false.

   if (first > last или
       файл dataFile прочитан полностью)
       return false
   else
   {
       // Считать средний блок файла dataFile в массив buf
       mid = (first + last)/2
       buf.readBlock(dataFile, mid)

       if ( (searchKey >= (buf.getRecord(0)).getKey()) &&
            (searchKey <=
            (buf.getRecord(recordsPerBlock-1)).getKey()) )
       {
          // Искомый блок найден
          Найти в буфере buf запись buf.getRecord(j),
             поисковый ключ которой равен значению
             аргумента searchKey
          if (запись найдена)
          {
              tableItem = buf.getRecord(j)
              return true
          }
          else
              return false
       } // Конец оператора if

   // Иначе просмотреть соответствующую половину файла
   else if (searchKey < (buf.getRecord(0)).getKey())
      return tableRetrieve(dataFile, recordsPerBlock,
                           first, mid-1, searchKey, tableItem)
   else
      return tableRetrieve(dataFile, recordsPerBlock,
   } // Конец оператора if
```

Алгоритм *tableRetrieve* рекурсивно делит файл пополам и считывает средний блок во внутренний объект *buf*. Чтобы разделить сегменты файла, необходимо знать номера первого и последнего блоков сегмента. Эти значения вместе с именем файла можно передать функции *tableRetrieve* в качестве аргументов.

Считав средний блок файла в буфер *buf*, мы определим, в какой половине находится запись с искомым ключом. Для этого достаточно сравнить значение аргумента *searchKey* с наименьшим поисковым ключом, содержащимся в буфере *buf*, т.е. с записью *buf.getRecord(0)*, и наибольшим поисковым ключом, находящимся в записи *buf.getRecord(recordsPerBlock-1)*. Если значение аргумента *searchKey* не лежит в диапазоне от наименьшего до наибольшего ключа буфера, необходимо рекурсивно обойти одну из половин файла (выбор этой по-

ловины зависит от результата сравнения аргумента *searchKey* с поисковыми ключами, содержащимися в буфере). Если значение аргумента *searchKey* лежит в указанном диапазоне, искать запись нужно в буфере. Поскольку записи буфера упорядочены, можно применить бинарный поиск. Однако количество записей в буфере обычно невелико, поэтому время, необходимое для последовательного перебора, пренебрежимо мало по сравнению с временем считывания блока из файла. Вследствие этого, как правило, для поиска записи в буфере применяется именно последовательный перебор.

Операции tableInsert и tableDelete при реализации внешней таблицы могут оказаться неэффективными, поскольку они связаны со сдвигом элементов

Эта внешняя реализация абстрактной таблицы не очень отличается от внутренней реализации в виде упорядоченного массива. Следовательно, этой реализации присущи те же самые преимущества и недостатки. Ее основным преимуществом является то, что элементы таблицы последовательно упорядочены, поэтому можно применять алгоритм бинарного поиска. Основной недостаток заключается в том, что, как и при реализации в виде упорядоченного массива, операции *tableInsert* и *tableDelete* сопряжены со сдвигом элементов. Файл может содержать громадное количество записей, объединенных в несколько тысяч блоков. Следовательно, сдвиг элементов потребовал бы слишком большого количества обращений к блокам.

В качестве примера рассмотрим рис. 14.5. Если в блок *k* вставить новую запись, придется сдвинуть не только записи, содержащиеся в блоке *k*, но и записи каждого блока, расположенного за ним. В результате мы будем вынуждены перемещать записи через границы блоков. Итак, каждый такой блок придется считать в оперативную память, сдвинуть записи с помощью оператора

```
buf.setRecord(i+1, buf.getRecord(i))
```

и записать его в файл, сохранив внесенные изменения. Для этого потребуется слишком большое количество обращений к блоками. По этой причине внешняя реализация таблицы в виде упорядоченного массива применяется только тогда, когда операции вставки и удаления выполняются редко. (См. упражнение 1 в конце главы.)

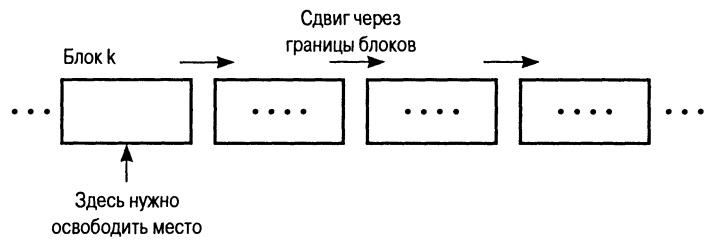


Рис. 14.5. Сдвиг элементов через границы блоков

Индексирование внешнего файла

Две наилучшие реализации внешних таблиц основаны на внутреннем хэшировании и деревьях поиска, соответственно. Основное различие между внутренними и внешними вариантами этих реализаций заключается в том, что внешние реализации используют **индексный файл**, а не сам файл данных. Индексный файл мало отличается от обычных предметных указателей. Рассмотрим, например, библиотечный каталог. Вместо того чтобы рыскать по всей библиотеке в поисках

нужной книги, мы можем просто найти ее карточку в каталоге. Каталог обычно упорядочен по алфавиту (по названиям или фамилиям авторов), поэтому найти нужную карточку очень легко. В этой карточке содержится запись о книге (например, ее номер), которая помогает найти ее на полке.

Преимущества библиотечного каталога

Используя каталог, мы получаем по крайней мере три преимущества.

- Поскольку каждая карточка намного меньше самой книги, весь каталог довольно большой библиотеки можно разместить в маленькой комнате. Это позволяет быстро обнаружить искомую книгу.
- Книги на полках могут стоять в произвольно установленном порядке. Чтобы найти конкретную книгу, ее нужно сначала отыскать в каталоге.
- Библиотека может создать несколько каталогов, в зависимости от вида поиска. Например, каталог можно организовать по фамилиям авторов, а можно — по названиям книг.

Теперь посмотрим, что общего у индексного файла и библиотечного каталога. Как показано на рис. 14.6, файл данных можно оставить неупорядоченным, если организовать соответствующий индексный файл. Если в файле данных нужно найти конкретную запись, достаточно найти в файле индексов ссылку на нее.

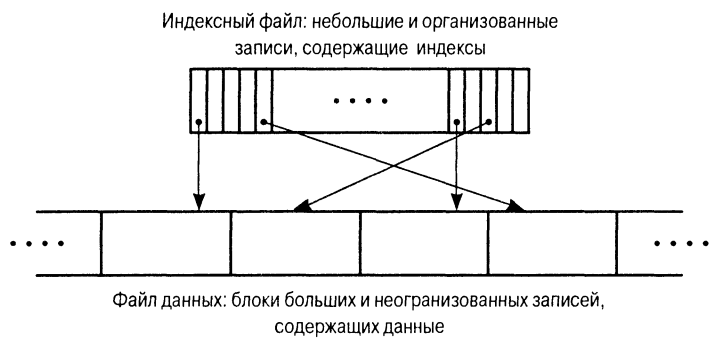


Рис. 14.6. Индексный файл и файл данных

Индексный файл

Итак, в индексном файле хранятся ссылки на записи. Каждая ссылка представляет собой **индексную запись** (index record), которая соответствует некоей записи из файла данных так же, как в каталоге каждая карточка соответствует какой-то книге. Индексная запись состоит из двух частей: **ключа** (key) , содержащего то же значение, что и поисковый ключ соответствующей записи в файле данных, и **указателя** (pointer) , ссылающегося на номер блока, в котором хранится запись с данным поисковым ключом. (Несмотря на похожее название, указатель индексной записи хранит не указатель языка C++, а обычное целое число — номер блока.) Итак, мы можем определить, в каком блоке находится искомая запись, содержащая ключ, равный значению `searchKey`.

Преимущества индексного файла

Индексный файл предоставляет те же преимущества, что и библиотечный каталог.

- Как правило, индексная запись намного меньше, чем запись, содержащая данные. В то время как запись данных может состоять из многочисленных компонентов, индексная запись содержит лишь ключ, который также является частью записи данных, и целочисленный указатель, задающий но-

мер блока. Таким образом, индексный файл намного меньше файла данных. Это позволяет уменьшить количество обращений к блокам при манипуляции с файлом данных.

- Поскольку файл данных может оставаться неупорядоченным, новую запись можно вставлять куда угодно, например, в конец файла данных. Это позволяет избежать сдвига данных при вставке и удалении элементов.
- Одному и тому же файлу данных можно ставить в соответствие несколько индексных файлов. Например, индексные файлы можно упорядочить по разным ключам. **Множественная индексация** (multiple indexing) кратко обсуждается в конце главы.

Упорядоченным является индексный файл, а не файл данных

Итак, файл данных можно не упорядочивать, однако индексный файл должен быть упорядочен, чтобы обеспечить быстрый поиск элементов. Прежде чем перейти к способам организации индексных файлов, рассмотрим более простую концепцию схемы. В частности, пусть индексный файл просто хранит последовательные индексные записи, упорядоченные по ключам, как показано на рис. 14.7.

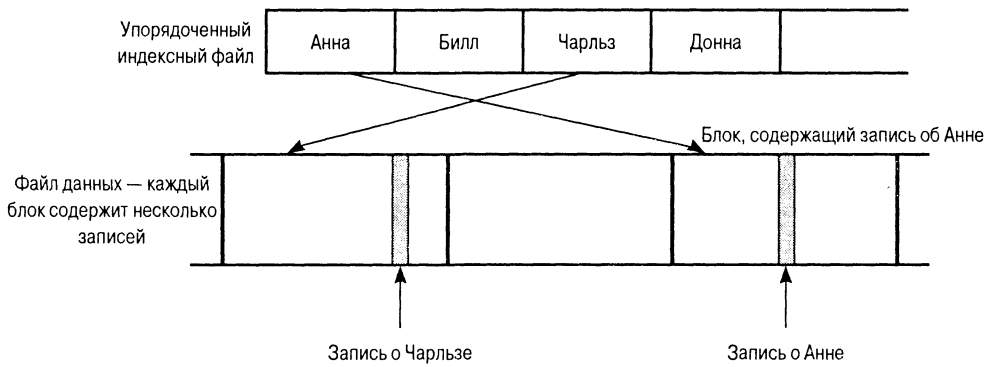


Рис. 14.7. Файл данных с упорядоченным индексным файлом

Для того чтобы, например, реализовать операцию `tableRetrieve`, необходимо применить бинарный поиск в индексном файле.

```
tableRetrieve(in tIndex:File, in tData:File,
              in searchKey:KeyItemType,
              out tableItem:TableItemType):boolean
// Извлекает из файла запись, содержащую поисковый ключ,
// равный значению аргумента searchKey, и присваивает ее
// переменной tableItem. Здесь аргумент tIndex задает имя
// индексного файла, а переменная tData — имя файла данных.
// Операция возвращает значение true, если искомая запись
// существует, в противном случае возвращается значение false.

   if (в файле tIndex не осталось непрочитанных блоков)
      return false
   else
   {
     // Считать средний блок индексного файла в объект buf
      mid = номер среднего блока индексного файла tIndex
      buf.readBlock(tIndex, mid)
```

```
    if ((searchKey >= (buf.getRecord(0)).getKey()) &&
       (searchKey <=
       (buf.getRecord(indexrecordsPerBlock-1)).getKey()))
    {
        // Искомый блок индексного файла найден
        Отыскать в массиве buf запись индексного файла
           buf.getRecord(j), поисковый ключ которой равен
           значению аргумента searchKey

        if (индексная запись buf.getRecord(j) найдена)
        {
           blockNum = номер блока в файле данных, на которую
                      ссылается запись buf.getRecord(j)
           data.readBlock(tData, blockNum)
           Найти запись данных data.getRecord(k),
              поисковый ключ которой равен значению
              аргумента searchKey
           tableItem = data.getRecord(k)
           return true
        }
        else
           return false
    }

    else if (в файле tIndex содержится только один блок)
        return false // Больше блоков в файле нет

    // Иначе — просмотреть соответствующую половину
    // индексного файла
    else if (searchKey < (buf.getRecord(0)).getKey())
        return tableRetrieve(первая половина файла tIndex,
                             tData, searchKey, tableItem)
    else
        return tableRetrieve(вторая половина файла tIndex,
                             tData, searchKey, tableItem)
} // Конец оператора if
```

Индексный файл позволяет сократить количество обращений к блокам при выполнении операций над таблицами

Поскольку индексные записи намного меньше записей в файле данных, индексный файл содержит намного меньше блоков. Например, если размер индексных записей равен одной десятой размера записей в файле данных, причем файл данных состоит из 1000 блоков, то для индексного файла потребуется лишь около 100 блоков. В результате использование индексов сокращает количество обращений к блокам при выполнении операции *tableRetrieve* с $\log_2 1000 = 10$ до $1+\log_2 100 = 8$. (Одно дополнительное обращение к файлу данных происходит при обнаружении соответствующей индексной записи.)

Более значительное сокращение обращений к блокам происходит при выполнении операций *tableInsert* и *tableDelete*. В реализации внешней таблицы, которую мы рассмотрели ранее, при вставке или удалении записи, принадлежащей первому блоку, нужно было сдвигать каждый блок, для чего потребовалось бы обратиться к каждому из 1000 блоков, хранящихся в файле данных. (См. рис. 14.5.)

Используя схему индексирования, при вставке и удалении элементов потребуется сдвинуть лишь индексные записи. Записи в файле данных могут храниться в произвольном порядке, поэтому вставку нового элемента можно производить в любое удобное место, например, в конец файла или позицию, освободившуюся при предыдущем удалении. В результате записи в файле данных сдвигать не потребуется вовсе. Однако придется сдвинуть индексные записи, чтобы сохранить порядок следования ключей. Поскольку индексный файл содержит намного меньше блоков (100 против 100), максимальное количество обращений к блокам намного сократится. Кроме того, на один сдвиг индексной записи потребуется намного меньше времени. Поскольку индексные записи имеют небольшие размеры, время, необходимое для выполнения оператора `buf.setRecord(i+1, buf.getRecord(i))`, уменьшится.

Сдвиг выполняется в индексном файле, а не в файле данных

Операции удаления при использовании индексной схемы также становятся более эффективными. Выполнив поиск записи в индексном файле и определив удаляемую запись в файле данных, мы можем просто очистить соответствующую ячейку и не сдвигать остальные записи. Номер освободившегося места можно запомнить (см. упражнение 2) и при следующей вставке поместить в него новый элемент. Сдвиг нужно выполнить лишь в индексном файле, чтобы заполнить пробел, образовавшийся вследствие удаления индексной записи.

Хотя описанная выше схема достаточно эффективна, она далека от идеала во многих конкретных приложениях. Иногда при вставке и удалении индексной записи может потребоваться более 100 обращений к блокам, что резко снижает производительность всей программы. Хэширование и деревья поиска позволяют создать намного более эффективные реализации внешних таблиц.

Неупорядоченный файл данных в сочетании с упорядоченным индексным файлом более эффективен, чем просто упорядоченный файл данных, однако существуют более удачные схемы

Внешнее хэширование

Схема внешнего хэширования очень похожа на схему внутреннего хэширования, описанную в главе 12. При внутреннем хэшировании каждый элемент массива `table` содержал указатель на начало списка элементов, поставленных в соответствие данной ячейке с помощью функции хэширования. При внешнем хэшировании каждый элемент массива `table` продолжает содержать указатель на начало списка, теперь состоящего из *блоков индексных записей*. Иными словами, хэшируется индексный файл, а не файл данных, как показано на рис. 14.8. (Во многих приложениях массив `table` сам по себе настолько велик, что должен храниться на внешнем запоминающем устройстве, например, в первых K блоках индексного файла. Чтобы избежать излишней детализации, будем предполагать, что массив `table` хранится в оперативной памяти.)

Хэшируется индексный файл, а не файл данных

Каждому элементу массива `table[i]` поставлен в соответствие связанный список блоков индексного файла (рис. 14.8). Каждый блок, входящий в связанный список, соответствующий элементу `table[i]`, содержит индексные записи, ключи которых (и соответствующие ключи записей из файла данных) хэшированы в ячейку `i`. Чтобы создать связанный список, в каждом блоке мы должны выделить память для указателя на блок — номер следующего блока в цепочке, — как показано на рис. 14.9. Иначе говоря, указателями в этом связанном списке служат целые числа, а не указатели языка C++. Значение указателя -1 играет роль константы *NULL*.

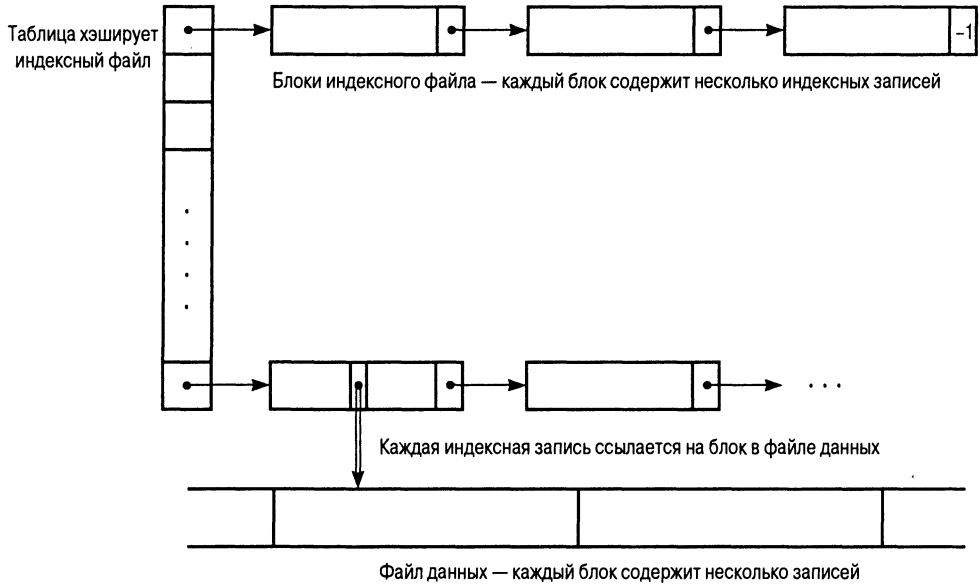


Рис. 14.8. Хэшированный индексный файл

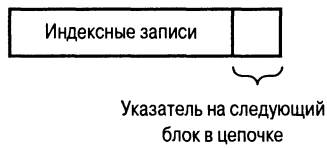


Рис. 14.9. Отдельный блок с указателем

Извлечение записи при внешнем хэшировании индексного файла. Рассмотрим псевдокод операции *tableRetrieve* при внешнем хэшировании индексного файла.

```
tableRetrieve(in tIndex:File, in tData:File,
              in searchKey:KeyItemType,
              out tableItem:TableItemType):boolean
// Извлекает из файла запись, содержащую поисковый ключ,
// равный значению аргумента searchKey, и присваивает ее
// переменной tableItem. Здесь аргумент tIndex задает имя
// индексного файла, а переменная tData — имя файла данных.
// Операция возвращает значение true, если искомая запись
// существует, в противном случае возвращается значение
// false. Применяет хэширование поискового ключа.

   i = h(searchKey)

   // Находим первый блок в цепочке индексных блоков —
   // эти блоки содержат индексные записи, хэшированные
   // в ячейку i
   p = table[i]

   // Если p == -1, в ячейку i не хэшировано ни одно значение
```

```
if (p != -1)
   buf.readBlock(tIndex, p)

// Ищем блок, содержащий искомую запись
while (p != -1 и буфер buf не содержит искомой
       индексной записи
{
   p = номер следующего блока в цепочке
   // Если p == -1, достигнут последний блок цепочки

   if (p != -1)
       buf.readBlock(tIndex, p)
} // Конец оператора while

// Извлекаем запись, если она существует
if (p != -1)
{
   // Элемент buf.getRecord(j) является индексной записью,
   // поисковый ключ которой совпадает со значением
   // аргумента searchKey
   blockNum = номер блока в файле данных, на который
              ссылается индексная запись buf.getRecord(j)
   data.readBlock(tData, blockNum)
   Найти запись данных data.getRecord(k), поисковый ключ
      которой совпадает со значением аргумента searchKey
   tableItem = data.getRecord(k)
   return true
}
else
   return false
```

Вставка при внешнем хэшировании индексного файла. Варианты операций *tableInsert* и *tableDelete* с применением внешнего хэширования очень похожи на версии с внутренним хэшированием. Основное отличие заключается в том, что при работе с внешними запоминающими устройствами нужно выполнять вставку или удаление двух записей: в индексный файл и файл данных.

Чтобы вставить в файл данных новую запись, поисковый ключ которой задается аргументом *searchKey*, нужно выполнить следующие шаги.

1. **Вставка записи в файл данных.** Поскольку файл данных не упорядочен, новую запись можно поместить куда угодно. Если в результате последнего удаления в файле образовался пробел, его можно заполнить, поместив туда новый элемент. (См. упражнение 2.)

   Если в файле нет свободных мест, новая запись помещается в конец последнего блока файла или, при необходимости, в новый блок, который добавляется в конце файла. В любом случае будем обозначать через *p* номер блока, который содержит новую запись.

2. **Вставка соответствующей индексной записи в индексный файл.** В индексный файл нужно вставить запись, содержащую ключ *searchKey* и указатель *p*. (Напомним, что указатель *p* представляет собой номер блока в файле данных, в который вставлена новая запись.) Поскольку индексный файл хэширован, сначала следует применить функцию хэширования к поисковому ключу *searchKey*.

   ```
   i = h(searchKey)
   ```

Затем индексная запись <*searchKey*, *p*> вставляется в цепочку блоков, на которую ссылается элемент *table[i]*. Эту запись можно поместить в любой блок этой цепочки, нужно лишь чтобы в этом блоке было свободное место. При необходимости в начало цепочки можно добавить новый блок.

Удаление записи при внешнем хэшировании индексного файла. Чтобы удалить из файла данных запись, содержащую поисковый ключ которой задается аргументом *searchKey*, необходимо выполнить следующие шаги.

1. **Поиск соответствующей индексной записи в индексном файле.** Применим к поисковому ключу функцию хэширования.

   ```
   i = h(searchKey)
   ```

   Затем найдем цепочку индексных блоков, ссылающихся с помощью элемента *table[i]* на индексную запись, поисковый ключ которой равен значению аргумента *searchKey*. Если такой индексной записи нет, файл данных не содержит записи, поисковый ключ которой был бы равен значению *searchKey*. Однако если в индексном файле обнаружена запись <*searchKey*, *p*>, ее следует удалить из индексного файла, а указатель *p* использовать для удаления соответствующей записи данных.

2. **Удаление записи из файла данных.** Итак, нужная нам запись находится в блоке *p* файла данных. Теперь мы получаем доступ к этому блоку, отыскиваем нужную запись, удаляем ее и записываем блок обратно в файл.

Обратите внимание, что операции *tableRetrieve*, *tableInsert* и *tableDelete* очень редко обращаются к блокам. В ходе их выполнения происходит не более одного обращения к блокам файла данных и, в худшем случае, обращения ко всем блокам, входящим в цепочку хэширования индексного файла. Используя те же приемы, что и при внутреннем хэшировании, можно минимизировать длины этих цепочек (до одного-двух блоков). Размер массива *table* должен быть достаточно большим, чтобы средняя длина цепочек не превышала одного-двух блоков, а функция хэширования должна равномерно распределять поисковые ключи. При необходимости цепочки блоков можно преобразовать во внешние деревья поиска, например, в **B-деревья**, используя методы, описанные в следующем разделе.

Внешнее хэширование следует применять при выполнении операций tableRetrieve, tableInsert и tableDelete

Внешнее хэширование позволяет повысить эффективность операций *tableRetrieve*, *tableInsert* и *tableDelete* над большими таблицами. Однако, как и в случае внутреннего хэширования, эта реализация не всегда практична. Например, при упорядоченном обходе, извлечении наименьшего или наибольшего элемента, а также при запросе по диапазону значений, в котором требуется упорядочить данные, эта реализация становится непригодной. Добавляя эти операции в список операций над абстрактной таблицей, следует использовать не хэширование, а деревья поиска.

B-деревья

Еще один способ поиска записи во внешней таблице основан на применении сбалансированного дерева поиска. Так же как и при внешнем хэшировании, во внешнее дерево следует преобразовать не файл данных, а индексный файл. Реализация этого подхода основана на обобщении 2-3 деревьев, описанных в главе 12.

Блоки внешнего файла можно представить в виде дерева, используя их номера в качестве указателей на дочерние узлы. Как показано на рис. 14.10, *а*, блоки

можно организовать в виде 2-3 дерева. Каждый блок файла становится узлом и содержит три указателя на дочерние узлы. Каждый из этих указателей представляет собой номер соответствующего блока. Значение –1 эквивалентно константе *NULL*. Таким образом, лист содержит три указателя, равных –1.

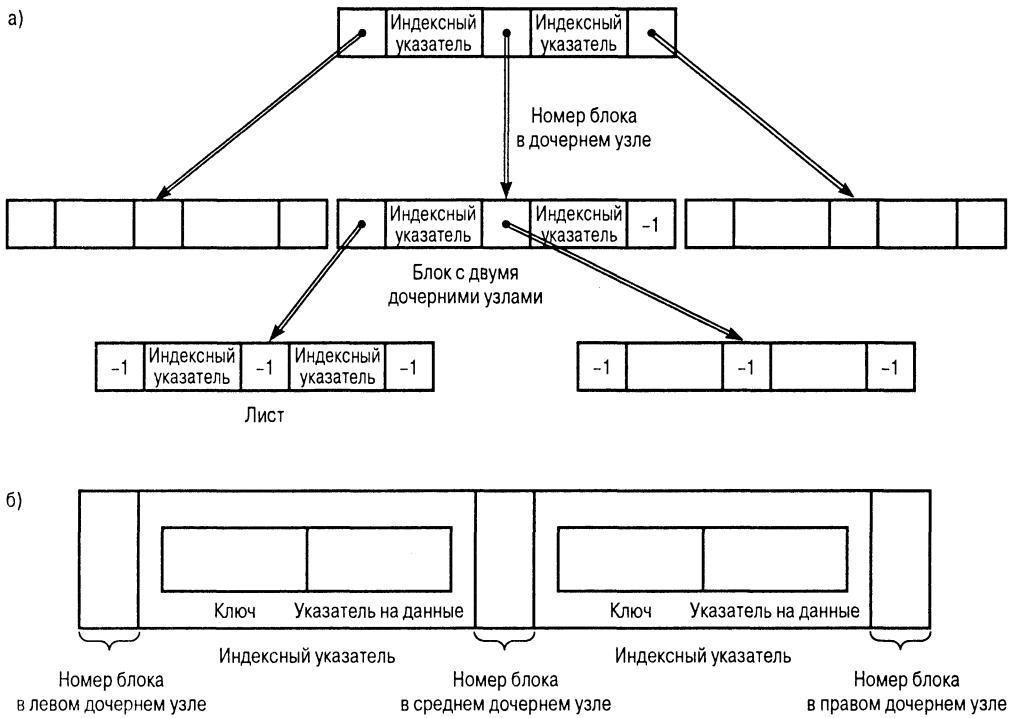


Рис. 14.10. Организация блоков в виде 2-3 дерева: а) блоки, организованные в виде 2-3 дерева; б) отдельный узел 2-3 дерева

Организация индексного файла в виде 2-3 дерева

Если индексный файл организован в виде 2-3 дерева, каждый узел (блок индексного файла) должен содержать одну или две индексных записи, имеющих вид <*key*, *pointer*>, а также три указателя на дочерние узлы. Эти указатели никак не связаны со структурой индексного файла. Указатель *pointer* ссылается на блок файла данных, содержащий поисковый ключ, равный значению поля *key*. (См. рис. 14.10, *б*.) Чтобы избежать путаницы, указатели, хранящиеся в узлах индексного файла, называются **дочерними указателями** (child index).

Индексные записи необходимо представить в виде дерева, так чтобы их ключи сохраняли порядок, присущий 2-3 деревьям. Это позволит извлекать записи данных по заданному значению поискового ключа с помощью следующего псевдокода.

```
tableRetrieve(in tIndex:File, in tData:File,
              in rootNum:integer, in searchKey:KeyItemType,
              out tableItem:TableItemType):boolean
// Извлекает запись, поисковый ключ которой равен значению
// аргумента searchKey, и присваивает ее переменной tableItem.
// Аргумент tIndex задает имя индексного файла,
```

```
// организованного в виде 2-3 дерева. Переменная rootNum
// равна номеру блока (индексного файла), содержащему корень
// дерева. Аргумент tData задает имя файла данных.
// Если запись существует, функция возвращает значение true,
// в противном случае она возвращает значение false.

   if (в файле tIndex не осталось непрочитанных блоков)
      return false
   else
   {
      // Считываем блок, содержащий корень 2-3 дерева,
      // из индексного файла во внутренний массив buf
      buf.readBlock(tIndex, rootNum)

      // Поиск индексной записи, поисковый ключ которой
      // равен значению аргумента searchKey

   if (ключ searchKey хранится в корне)
   {
      blockNum = номер блока в файле данных, на который
                 ссылается индексная запись
      data.readBlock(tData, blockNum)
      Найти запись данных data.getRecord(k), поисковый ключ
         которой равен значению аргумента searchKey
      tableItem = data.getRecord(k)
      return true
   }

   // Иначе — искать запись в соответствующем поддереве
   else if (корень является листом)
      return false

   else
   {
      child = номер блока, содержащего корень
              соответствующего поддерева
      return tableRetrieve(tIndex, tData, child,
                           searchKey, tableItem)
   } // Конец оператора if
} // Конец оператора if
```

Операции вставки и удаления можно реализовать так же, как и во внутренней версии таблицы. Отличие состоит в том, что теперь одновременно нужно вставлять и удалять записи из индексного файла и файла данных (как и в схеме внешнего хэширования). Вставляя записи в индексный файл и удаляя их оттуда, следует расщеплять и объединять узлы дерева, так же как и во внутренней версии. Вставка и удаление записей в файле данных происходят точно так же, как и при внешнем хэшировании. Напомним, что этот файл остается неупорядоченным. Таким образом, 2-3 деревья обеспечивают высокую эффективность операций над внешними таблицами.

Внешние 2-3 деревья - хороший выбор, но существует более эффективный способ

Описанные выше 2-3 деревья можно обобщить, получив структуру, которая позволяет еще эффективнее реализовать операции над внешними таблицами. Напомним, что в главе 12 мы уже упоминали о деревьях, узлы которых имеют много дочерних узлов. Уве-

личение количества дочерних узлов у каждого узла дерева позволяет уменьшить его высоту, но увеличивает количество сравнений, которые должны выполняться при обработке узлов в ходе поиска элемента по заданному значению.

Внешнее дерево поиска должно быть невысоким

Однако при работе с внешними запоминающими устройствами преимущества невысоких деревьев перевешивают недостатки, обусловленные возросшим количеством операций сравнения. При обходе внешнего дерева поиска при посещении каждого узла происходит обращение к блоку. Поскольку время, необходимое для доступа к блоку, содержащемуся во внешнем файле, как правило, намного превышает время обработки данных, основной приоритет следует отдать сокращению количества обращений к блокам. Следовательно, необходимо стремиться уменьшить высоту внешних деревьев, даже за счет дополнительного количества внутренних сравнений. Во внешнем дереве поиска каждый узел должен иметь как можно больше дочерних узлов. Единственный фактор, который препятствует этому, — размер блока.

Сколько дочерних узлов может иметь блок фиксированного размера? Если узел должен иметь m дочерних узлов, очевидно, потребуется разместить в нем m указателей. Однако, кроме указателей на дочерние узлы, блок должен содержать индексную запись.

Прежде чем ответить на вопрос, сколько дочерних узлов может иметь блок, рассмотрим другой вопрос, тесно связанный с предыдущим: если узел N в дереве поиска имеет m дочерних узлов, сколько значений ключа — и, следовательно, сколько индексных записей — он должен содержать?

Узел бинарного дерева поиска: количество записей и дочерних узлов

Если узел N в бинарном дереве поиска имеет два дочерних узла, он должен содержать одно значение ключа, как показано на рис. 14.11, *а*. Значение ключа, содержащееся в узле N, разделяет значения ключей в двух поддеревьях узла N — все ключи левого поддерева меньше, а все ключи правого поддерева больше этой величины. От этого значения зависит, в каком из поддеревьев следует продолжать поиск.

Аналогично, если узел N в 2-3 дереве имеет три дочерних узла, он должен содержать два значения ключа (рис. 14.11, *б*). Эти два значения разделяют между собой три поддерева узла N — все ключи левого поддерева меньше обоих ключей узла N, все ключи среднего поддерева заключены между этими значениями, все ключи правого поддерева больше обоих ключей узла N. Как и для бинарного дерева поиска, это позволяет выбрать направление дальнейшего поиска.

Узел дерева поиска общего вида: количество записей и дочерних узлов

Как правило, если узел N в дереве поиска должен иметь m дочерних узлов, он должен содержать m-1 ключей, чтобы правильно разделить значения ключей, содержащихся в поддеревьях (рис. 14.11, *в*). Обозначим поддеревья узла N через S_0, S_1, ... , S_{m-1}, а значения ключа в узле N — как K_0, K_1, ... , K_{m-1} (причем $K_0 < K_1 < \ldots < K_{m-1}$). Значения ключа, содержащиеся в узле N, должны разделять поддеревья следующим образом.

- Все значения в поддереве S_0 должны быть меньше величины K_1.
- Для всех чисел i из диапазона $1 \le i \le m-2$ все значения ключа в поддеревьях S_i должны быть больше K_i и меньше K_{i+1}.
- Все значения ключа в поддереве S_{m-1} должны быть больше величины K_{m-1}.

Извлечение записи из внешнего дерева поиска общего вида

Если каждый узел дерева обладает этим свойством, для поиска записи можно применить обобщенный вариант алгоритма извлечения элемента из дерева поиска. Таким образом, операцию `tableRetrieve` можно реализовать с помощью следующего псевдокода.

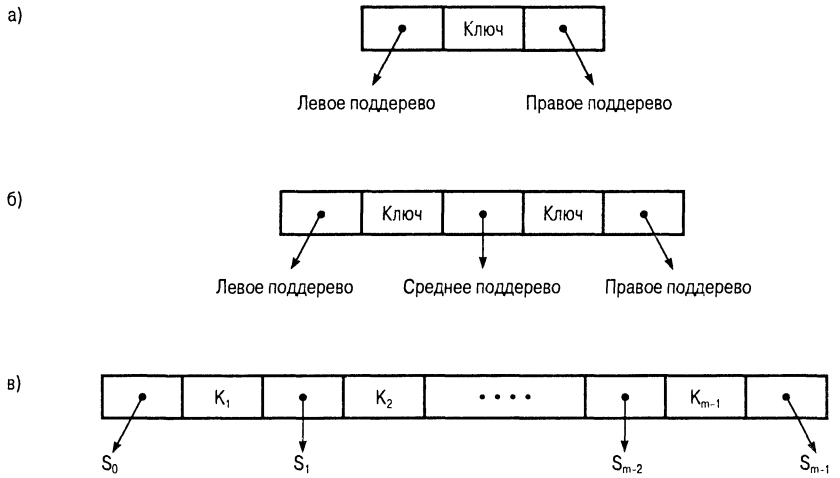


Рис. 14.11. Узлы бинарного дерева: а) узел с двумя дочерними узлами; б) узел с тремя дочерними узлами; в) узел с m дочерними узлами

```
tableRetrieve(in tIndex:File, in tData:File,
              in rootNum:integer, in searchKey:KeyItemType,
              out tableItem:TableItemType):boolean
// Извлекает запись, поисковый ключ которой равен значению
// аргумента searchKey, и присваивает ее переменной tableItem.
// Аргумент tIndex задает имя индексного файла,
// организованного в виде дерева поиска. Переменная rootNum
// равна номеру блока (индексного файла), содержащему корень
// дерева. Аргумент tData задает имя файла данных.
// Если запись существует, функция возвращает значение true,
// в противном случае она возвращает значение false.

   if (в файле tIndex не осталось непрочитанных блоков)
      return false
   else
   {
      // Считываем блок, содержащий корень 2-3 дерева,
      // из индексного файла во внутренний массив buf
      buf.readBlock(tIndex, rootNum)

      // Поиск индексной записи, поисковый ключ которой
      // равен значению аргумента searchKey

   if (значение searchKey равно одной из величин Ki,
       содержащихся в корне)
   {
      blockNum = номер блока в файле данных, на который
                 ссылается индексная запись
      data.readBlock(tData, blockNum)
      Найти запись данных data.getRecord(k), поисковый ключ
         которой равен значению аргумента searchKey
      tableItem = data.getRecord(k)
      return true
   }
```

```
    // Иначе искать запись в соответствующем поддереве
    else if (корень является листом)
       return false

    else
    {
       Определить, в каком из поддеревьев Si
          следует продолжать поиск
       child = номер блока, содержащего корень
             соответствующего поддерева
       return tableRetrieve(tIndex, tData, child,
                            searchKey, tableItem)
    } // Конец оператора if
} // Конец оператора if
```

Теперь вернемся к вопросу, сколько дочерних узлов может иметь узел дерева поиска, т.е. насколько большим может быть число m? Если нам нужно организовать индексный файл в виде дерева поиска, элементами, которые хранятся в каждом узле, будут записи <*key*, *pointer*>. Следовательно, если каждый узел дерева (который, напомним, является блоком индексного файла) должен иметь m дочерних узлов, он должен быть достаточно большим, чтобы уместить m дочерних указателей и $m-1$ записей, имеющих вид <*key*, *pointer*>. Величина m должна быть наибольшим целым числом, позволяющим m дочерним указателям (которые, напомним, являются целыми числами) и $m-1$ записи вида <*key*, *pointer*> уместиться в одном блоке файла. Если всегда выбирать только четные числа m, алгоритм немного упростится. Иными словами, величина m — это наибольшее четное целое число, такое что m дочерних указателей и $m-1$ индексных записей могут уместиться в одном блоке.

Количество дочерних узлов каждого узла

В идеале, внешнее дерево должно иметь такую структуру, чтобы каждый внутренний узел имел m дочерних узлов, где число m выбирается по правилу, описанному выше, а все листья должны находиться на одном уровне, как в полных деревьях и 2-3 деревьях. Например, на рис. 14.12 показано полное дерево, внутренние узлы которого имеют пять дочерних узлов. Хотя это дерево имеет минимально возможную высоту, при вставках и удалении его элементов с большим трудом сохраняется баланс. Следовательно, нужно искать компромисс. Можно по-прежнему настаивать, чтобы все листья дерева находились на одном и том же уровне, т.е. чтобы дерево было сбалансированным, но теперь мы должны разрешить, чтобы каждый узел имел от m до $[m/2]+1$ дочерних узлов. (Обозначение [] означает наибольшее целое число, не превышающее данное, например $[5/2]=2$.)

В-дерево степени m

Деревья такого вида называются **B-деревьями степени *m*** (B-tree of degree m) и обладают следующими свойствами.

- Все листья дерева находятся на одном и том же уровне.
- Каждый узел дерева содержит $m-1$ до $[m/2]$ записей, причем количество дочерних узлов, которые имеет каждый узел, на единицу превышает количество записей. Исключением из этого правила является корень дерева, который содержит как минимум одну запись и может иметь не меньше двух дочерних узлов. Это исключение продиктовано необходимостью эффективно выполнять операции вставки и удаления элементов, описанные ниже.

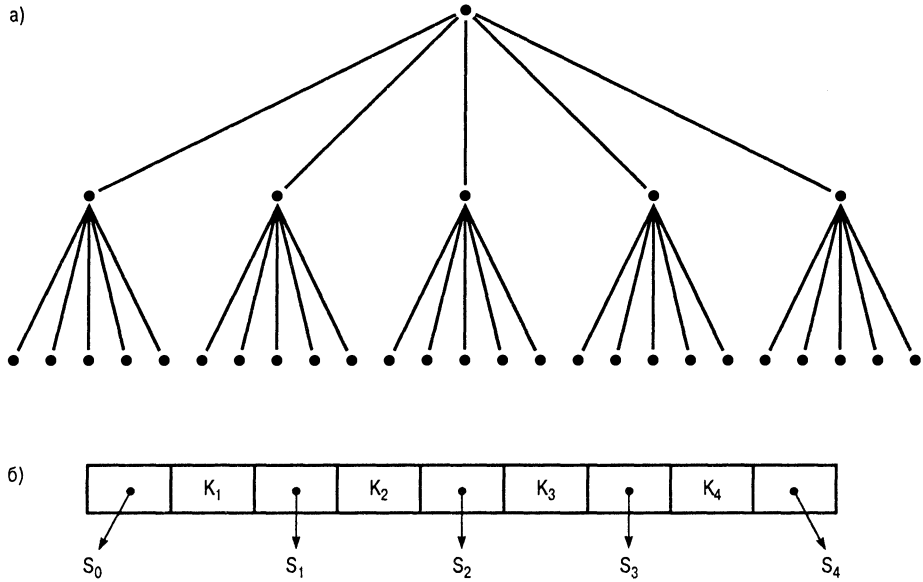


Рис. 14.12. Пример полного дерева: а) полное дерево, внутренние узлы которого имеют пять дочерних узлов; б) формат отдельного узла

2-3 дерево — это B-дерево степени 3

Итак, 2-3 дерево — это B-дерево степени 3. Более того, способ, которым алгоритмы вставки и удаления элементов B-дерева поддерживают структуру дерева является прямым обобщение стратегии расщепления и слияния узлов 2-3 дерева.

Проиллюстрируем алгоритмы вставки и удаления элементов B-дерева на конкретном примере. Будем предполагать, что индексный файл организован в виде B-дерева степени 5, т.е. число 5 — максимальное, а число 3 — минимальное количество дочерних узлов, которое может иметь внутренний узел дерева (не являющийся его корнем). (Как правило, B-дерево имеет более высокую степень, но его диаграмма становится слишком сложной!)

Вставка в B-дерево. Чтобы вставить в B-дерево, изображенное на рис. 14.13, запись с ключом 55, нужно выполнить следующие шаги.

1. **Вставка записи в файл данных.** Сначала найдем в файле данных блок p, в который следует вставить новую запись. Как и при внешнем хэшировании, блок p может быть любым блоком, содержащим вакантное место, или новым блоком.

2. **Вставка соответствующей индексной записи в индексный файл.** Теперь мы должны вставить индексную запись <55, p> в индексный файл, представляющий собой B-дерево степени 5. На первом шаге необходимо найти лист этого дерева, которому должна принадлежать данная запись. Для этого нужно определить, где остановится поиск ключа 55.

   Допустим, что поиск завершился в листе L, показанном на рис. 14.14, *а*. Вставка в лист L новой записи приведет в тому, что он будет содержать пять записей (рис. 14.14, *б*). Поскольку узел может содержать лишь четыре записи, лист L нужно расщепить на узлы L_1 и L_2. Аналогично расщеплению узлов 2-3 дерева, узел L_1 будет хранить две записи с наименьшими

ключами, а узел L_2 — три записи с наибольшими ключами, а запись со средним значением ключа (56) будет передвинута вверх — в родительский узел P (рис. 14.14, *в*).

В этом примере узел P имеет шесть дочерних узлов и пять записей, поэтому его придется разделить на узлы P_1 и P_2. Запись со средним ключом (56) будет передвинута еще дальше — в родительский узел Q. Затем дочерние узлы узла P нужно перераспределить, как это делалось в 2-3 дереве (рис. 14.14, *г*).

На этом операция вставки завершена, поскольку родитель узла P — узел Q — содержит только три записи и имеет только четыре дочерних узла. Как правило, расщепление узлов при вставке распространяется и на корень (рис. 14.14, *д*). После расщепления корня новый корень будет содержать только одну запись и иметь лишь два дочерних узла — это вполне соответствует определению B-дерева.

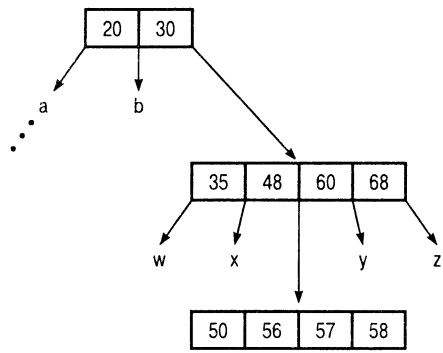


Рис. 14.13. B-дерево степени 5

Удаление из B-дерева. Чтобы удалить из B-дерева запись, содержащую заданный ключ, нужно выполнить следующие шаги.

1. **Найти индексную запись в индексном файле.** Для поиска нужной индексной записи в индексном файле применяется алгоритм поиска. Если запись уже не содержится в листе, ее нужно поменять местами с симметричными преемником (см. упражнение 5). Допустим, что лист L, показанный на рис. 14.15, *а*, содержит индексную запись с искомым ключом 73. Определив указатель p в индексной записи (он потребуется на втором шаге при удалении записи данных), следует удалить индексную запись из листа L (рис. 14.15, *б*). Поскольку узел L теперь содержит лишь одно значение (напомним, что узел должен содержать как минимум два значения), а узлы-братья не могут поделиться свободными значениями, лист L придется объединить с одним из братьев, перенеся вниз запись, содержащуюся в родительском узле P (рис. 14.15, *в*). Обратите внимание, что этот шаг алгоритма аналогичен слиянию узлов 2-3 дерева. Однако теперь узел P содержит лишь одно значение и имеет два дочерних узла. Поскольку братья узла P не могут поделиться своими записями и дочерними узлами, его придется объединить с узлом P_1, перенеся в новый узел запись из его родительского узла Q. Поскольку узел P является внутренним, его дочерние узлы присоединяются к узлу P_1 (рис. 14.15, *г*).

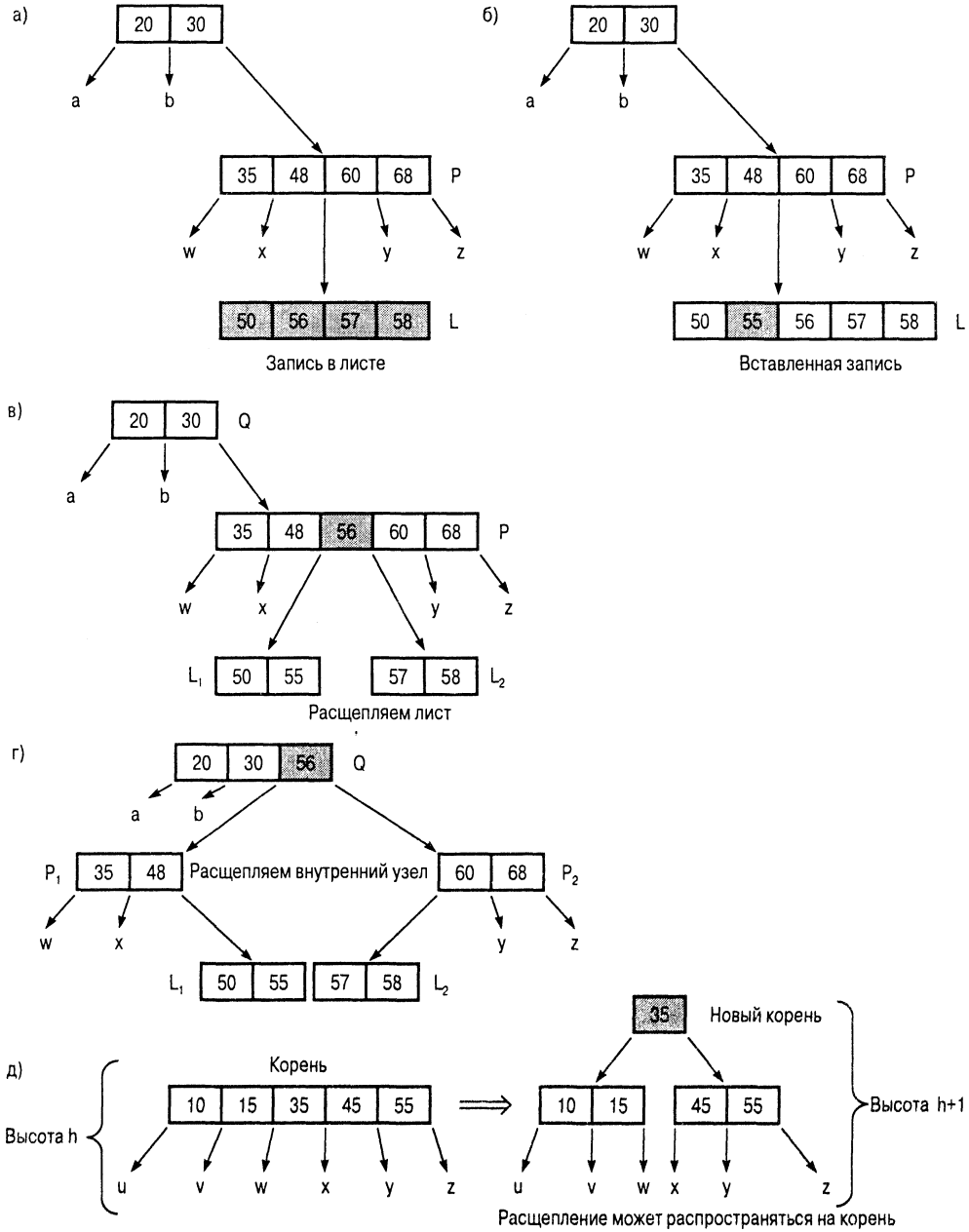
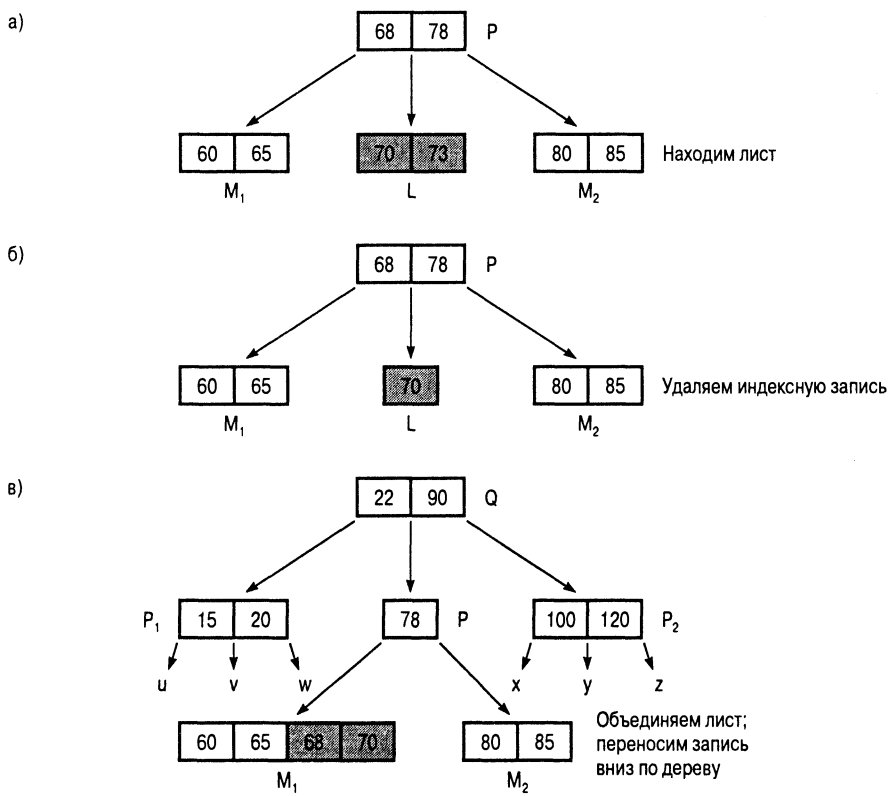


Рис. 14.14. Шаги алгоритма при вставке записи 55

После этого слияния узел Q (родитель узла P) остается лишь с двумя дочерними узлами и одной записью. Однако в этом случае узел Q_1 — брат узла Q — может поделиться одной записью и одним дочерним узлом. Это позволяет перераспределить дочерние узлы и записи среди узлов Q_1, Q и родительским узлом S. На этом операция удаления завершается (рис. 14.15, *д*). Если удаление распространяется вплоть до корня, причем у него остается лишь одна запись и только два дочерних узла, операция завершается, поскольку определение B-дерева допускает такую ситуацию. Если удаление приводит к тому, что корень имеет один дочерний узел и не содержит ни одной записи, корень удаляется из дерева, а его высота уменьшается на 1, как показано на рис. 14.15, *е*. На этом удаление индексной записи завершается, и следует переходить к удалению соответствующей записи данных.

2. **Удаление записи из файла данных.** Прежде чем удалить индексную запись, нужно запомнить указатель p. Блок p в файле данных содержит запись, подлежащую удалению. Следовательно, нужно просто обратиться к блоку p, удалить запись и записать его обратно в файл. Псевдокод алгоритма вставки и удаления аналогичны соответствующим алгоритмам для 2-3 дерева. Читатели могут сами их сформулировать.



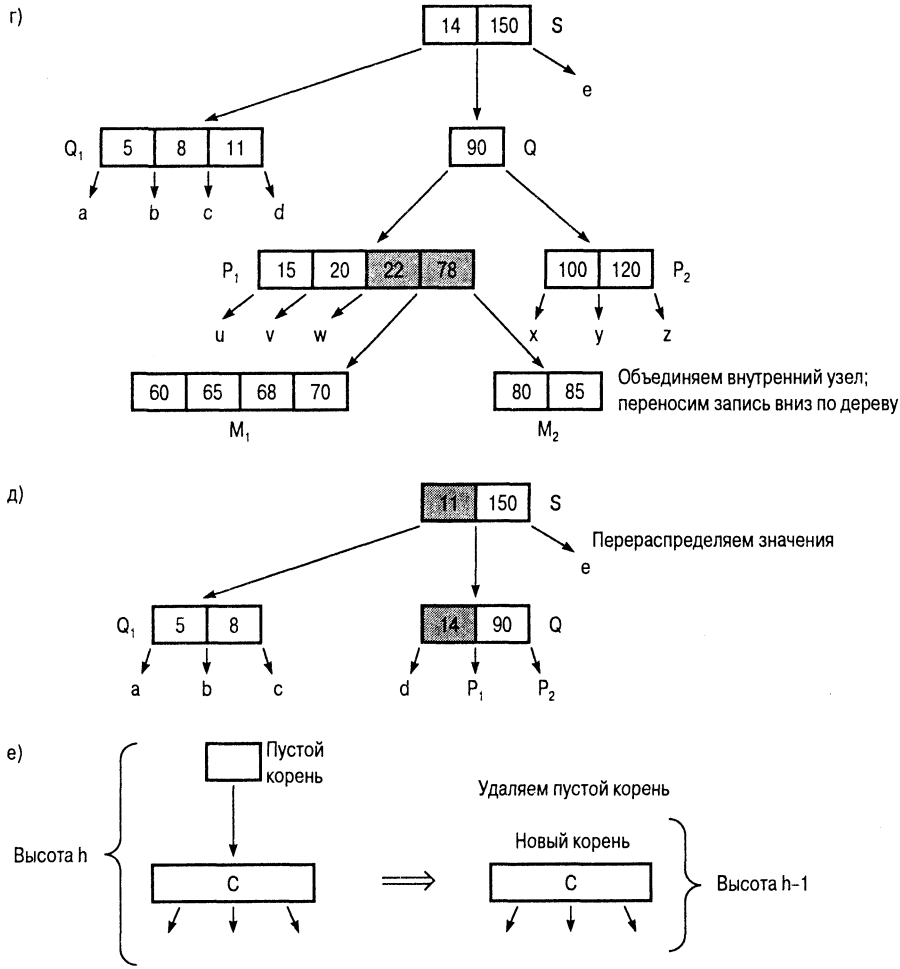


Рис. 14.15. Шаги при удалении записи 73

Алгоритмы обхода

Рассмотрим теперь операцию *traverseTable*, которая выполняется в заданном порядке. Хэширование не может повысить эффективность этой операции. Довольно часто в приложениях нужно лишь, чтобы при обходе на экран выводились значения поисковых ключей. В этих ситуациях реализация таблицы в виде B-дерева может обеспечить эффективное выполнение операции обхода, поскольку при этом не требуется обращаться к файлу данных. Мы можем посещать лишь поисковые ключи, используя алгоритм обхода B-дерева в симметричном порядке.

Алгоритм обращается только к поисковым ключам каждой индексной записи, а не к файлу данных

```
traverseTable(in blockNum:integer,
              in m:integer)
// Обходит в порядке следования ключей
// индексный файл,
```

Симметричный обход индексного файла, организованного в виде B-дерева

```
// представленный в виде B-дерева степени m. Аргумент
// blockNum задает номер блока, в котором содержится корень
// B-дерева.

   if (blockNum != -1)
   {
      // Считываем корень во внутренний массив buf
      buf.readBlock(indexFile, blockNum)

      // Обходим дочерний узел

      // Обходим поддерево S0
      Пусть p — номер блока, содержащегося в 0-м дочернем
        узле узла buf
      traverseTable(p, m)

      for (i = 1 до m - 1)
      {
         Вывести на экран ключ Ki узла buf

         // Обходим поддерево Si
         Пусть p — номер блока, содержащегося в i-м дочернем
           узле узла buf
         traverseTable(p, m)
      } // Конец оператора for
   } // Конец оператора if
```

При таком обходе происходит минимальное количество обращений к блокам, поскольку каждый блок индексного файла считывается только один раз. Однако этот алгоритм основан на предположении, что у нас имеется достаточно оперативной памяти для поддержки стека, состоящего из h блоков, где h — высота дерева. Во многих ситуациях это предположение является вполне разумным, например, высота B-дерева степени 255, индексирующего файл, состоящий из 16 миллионов записей, не превышает 3. Если оперативной памяти для поддержки стека из h блоков не достаточно, нужно использовать другой алгоритм. (См. упражнение 9.)

Обращение ко всей записи данных

Если при обходе на экран нужно вывести всю запись данных (а не только поисковый ключ), B-дерево становится менее привлекательным. В этом случае следует применять следующий алгоритм обхода.

Упорядоченный обход файла данных, индексированного с помощью B-дерева

```
traverseTable(in blockNum:integer,
              in m:integer)
// Обходит в порядке следования ключей
// индексный файл,
// представленный в виде B-дерева степени m. Аргумент
// blockNum задает номер блока, в котором содержится корень
// B-дерева.
   if (blockNum != -1)
   {
      // Считываем корень во внутренний массив buf
      buf.readBlock(indexFile, blockNum)

      // Обходим дочерний узел
      // Обходим поддерево S0
```

```
        Пусть p — номер блока, содержащегося в 0-м дочернем
          узле узла buf
        traverseTable(p, m)

        for (i = 1 до m - 1)
        {
           Пусть p_i — указатель на i-ю индексную запись
              d массиве buf
           data.readBlock(dataFile, p_i)
           Извлечь данные из записи, поисковый ключ которой
              равен K_i
           Вывести на экран запись данных

           // Обходим поддерево S_i
           Пусть p — номер блока, содержащегося в i-м дочернем
             узле узла buf
           traverseTable(p, m)
        } // Конец оператора for
     } // Конец оператора if
```

Как правило, описанный выше алгоритм не приемлем

В этом алгоритме требуется прочитать блок из файла данных, а затем вывести на экран содержащиеся в нем данные. Иначе говоря, количество обращений к блокам файла данных равно количеству записей. Как правило, большое количество обращений к блокам файла данных недопустимо. Если такой обход выполняется достаточно часто, придется модифицировать схему B-дерева, чтобы файл данных оказался почти упорядоченным.

Множественная индексация

Завершая обзор внешних реализаций, рассмотрим множественную индексацию файла данных. В главе 12 была сформулирована задача, в которой данные в памяти компьютера следовало организовать несколькими способами. Эта проблема относится и к внешним запоминающим устройствам. Допустим, что файл данных содержит набор записей о сотрудниках, причем для извлечения записей используются два способа.

```
retrieveN(in aName:NameType):ItemType
// Извлекает элемент, поисковый ключ которого содержит имя aName

retrieveS(in ssn:SSNType):ItemType
// Извлекает элемент, поисковый ключ которого содержит
// номер карточки социального страхования ssn
```

Файлы множественной индексации позволяют по-разному организовать данные

Для решения этой задачи можно применить два независимых файла индексации одного и того же файла данных. Например, один индексный файл должен содержать индексные записи вида <*name*, *pointer*>, а второй — вида <*socSec*, *ponter*>. Оба файла или один из них можно хэшировать или представить в виде B-деревьев, как показано на рис. 14.16. Выбор способа индексации зависит от списка операций, которые должны выполняться с поисковыми ключами. (Аналогично, если в приложении необходимо очень быстро выполнять извлечение ключа *socSec*, а также обходить таблицу в порядке его возрастания, либо выполнять запросы по диапазону значений, имеет смысл создать два индексных файла: один — хэшированный, а второй — в виде B-дерева.)

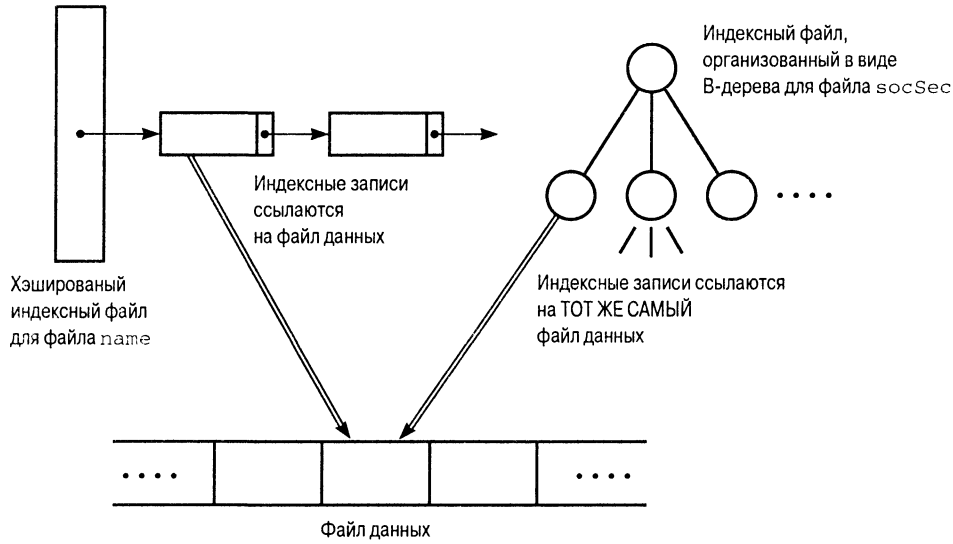


Рис. 14.16. Файлы множественной индексации

Если для извлечения записи используется только один из индексов (т.е. операция `retrieveN` использует только ключ `name`, а операция `retrieveS` — только ключ `socSec`), операции вставки и удаления должны обновлять оба индексных файла. Например, операция удаления записи по имени `deleteN(Джонс)` выполняется за три шага.

1. Найти индексную запись о сотруднике с фамилией Джонс и стереть ее.
2. Удалить соответствующие данные в файле данных, выполнив поиск по ключу `socSec`.
3. Найти индексную запись по заданному номеру карточки социального страхования и стереть ее.

Операция удаления по имени должна обновлять оба индексных файла

Как правило, множественное индексирование требует больше памяти и дополнительных затрат на обновление индексных файлов при модификации данных.

Мы лишь коснулись основных принципов организации данных на внешних запоминающих устройствах. Детали описанных алгоритмов сильно зависят от конкретных компьютерных систем. В разных ситуациях могут понадобиться разные методы, в том числе такие, которые совершенно отличаются от описанных выше. Знание этих методов приходит с опытом.

Резюме

1. Внешний файл разделен на блоки. Каждый блок обычно содержит много записей. Как правило, блок является наименьшей единицей обмена данными между внешней и оперативной памятью. Чтобы получить доступ к записи, нужно обратиться к блоку, который ее содержит.
2. Файл прямого доступа обеспечивает непосредственный доступ к i-му блоку, не требуя обхода предшествующих блоков. В этом смысле он напоминает массив.

3. Алгоритм сортировки слиянием, описанный в главе 9, можно модифицировать так, что с его помощью оказывается возможной сортировка внешнего файла без загрузки всех данных в оперативную память.
4. Индексный файл содержит индексные записи о каждой записи в файле данных. Индексная запись содержит поисковый ключ соответствующей записи данных и номер блока в файле данных, содержащего эту запись.
5. Индексный файл можно хэшировать или представить в виде B-дерева. Эти схемы позволяют выполнять основные операции над таблицами, редко обращаясь к блокам.
6. Один и тот же файл данных можно одновременно индексировать поразному. Множественная индексация позволяет эффективно выполнять разные операции, например, извлечение по имени и по номеру карточки социального страхования.

Предупреждения

1. Перед обработкой записи (проверкой или модификацией) ее нужно считать из внешнего файла в оперативную память. Модифицировав запись, ее нужно записать обратно в файл.
2. Обращение к блоку, как правило, выполняется медленнее компьютерных операций. Следовательно, нужно тщательно продумывать организацию файла, чтобы минимизировать количество обращений к блокам. В противном случае время поиска записей будет очень большим.
3. После вставки или удаления записи данных необходимо вносить соответствующие изменения в индексный файл. Если файлу данных соответствует несколько индексных файлов, необходимо обновить каждый из них. Таким образом, множественная индексация приводит к дополнительным затратам.
4. Хотя внешнее хэширование позволяет выполнять операции извлечения и удаления записей быстрее, чем B-деревья, оно не обеспечивает эффективного выполнения операций упорядоченного обхода или запроса по диапазону значений. Этот недостаток устраняется с помощью множественного индексирования.

Вопросы для самопроверки

1. Рассмотрим два файла, каждый из которых состоит из 1600 записей о сотрудниках. Записи в каждом файле разбиты на 16 блоков, содержащих по 100 записей. Один файл обеспечивает последовательный доступ, а второй — прямой. Опишите процесс вставки новой записи в конец каждого файла.
2. Выполните трассировку алгоритма `externalMergesort` на примере внешнего файла, состоящего из 16 блоков. Допустим, что в массивы `in1`, `in2` и `out` можно записать только один блок. Перечислите вызовы разных функций в порядке их появления.
3. Выполните трассировку алгоритма извлечения записи из внешнего индексированного файла. Искомая запись имеет наименьший поисковый ключ среди всех ключей в индексном файле. Допустим, что индексный файл хранит индексные записи последовательно, в порядке следования их поисковых ключей, и содержит 20 блоков по 50 записей. Кроме того, будем предполагать, что файл данных содержит 100 блоков, а каждый блок состоит из

10 записей о сотрудниках. Перечислите вызовы разных функций в порядке их появления.

4. Повторите задание 3, предполагая, что поисковый ключ равен ключу 26-й записи в 12-м блоке индексного файла. Предполагается, что запись 26 ссылается на 98-й блок файла данных.

Упражнения

1. Предполагая существование функций *readBlock* и *writeBlock*, напишите псевдокод программы, выполняющей сдвиг данных для создания свободного места в указанной позиции упорядоченного файла. Уделите особое внимание деталям сдвига последнего элемента каждого блока и первого элемента следующего блока. Предполагается, что последней записью файла является запись *lastRec* блока *lastBlock*, причем этот блок не полон. (Это предположение позволяет выполнять сдвиг данных без создания нового блока.)

2. Проблема управления блоками внешнего файла данных, индексированного с помощью B-деревьев или схемы внешнего хэширования, похожа на проблемы управления памятью при работе с внутренними структурами. Если внешней структуре, например, файлу данных, нужно больше памяти (например, для вставки новой записи), он извлекает новый блок из **списка свободной памяти** (free list), поддерживаемого оперативной системой. Иными словами, если файл содержит n блоков, система может выделить память для $(n+1)$-го блока. Если файлу блок больше не нужен, его можно удалить из памяти и вернуть освобожденный участок системе.

   Сложность при управлении внешними запоминающими устройствами заключается в том, что блок, выделенный файлу, может перемежаться другими данными. Например, после удаления записи из середины файла данных блок, содержащий эту запись, имеет достаточно свободного места, чтобы поместить туда новую запись. Следовательно, необходимо отслеживать наличие пустот в блоках, чтобы вовремя обнаружить, что блок совершенно опустел (и освободить занимаемую им память).

   Предполагая, что функции *allocateBlock* и *returnBlock*, выделяющие память для блоков и освобождающие память от пустых блоков, уже существуют, напишите псевдокод следующих функций.

```
getSlot(in dataFile:File, out blockNum:integer,
        out recNum:integer)
// Определяет номер блока (blockNum) и номер записи (recNum),
// содержащих свободное место в файле dataFile.
// При необходимости система выделяет память для нового блока.

freeSlot(in dataFile:File, out blockNum:integer,
        in recNum:integer)
// Создает запись recNum в блоке blockNum файла dataFile/
// Пустые блоки удаляются из памяти.
```

   Какая структура данных лучше всех поддерживает эти операции? Предполагается, что пустоты в блоках можно отличить от занятых позиций. Это можно сделать, помещая в запись константу *NULL* или признак полная/пустая.

3. Опишите псевдокод алгоритмов вставки записей во внешнюю таблицу и удаления их оттуда с помощью хэшированного индексного файла.

4. Выполните следующую последовательность операций над пустой абстрактной таблицей t, реализованной в виде B-дерева степени 5. Обратите внимание, что операция вставки в пустое B-дерево создает отдельный узел, содержащий вставляемую запись.

```
t.tableInsert(10)
t.tableInsert(100)
t.tableInsert(30)
t.tableInsert(80)
t.tableInsert(50)
t.tableDelete(10)
t.tableInsert(60)
t.tableInsert(70)
t.tableInsert(40)
t.tableDelete(80)
t.tableInsert(90)
t.tableInsert(20)
t.tableDelete(30)
t.tableDelete(70)
```

5. Опишите псевдокод алгоритма для нахождения симметричного преемника элемента во внешнем B-дереве.

6. Опишите псевдокод алгоритма вставки и удаления элементов абстрактной таблицы, реализованной с помощью индексного файла, представленного в виде B-дерева.

7. Напишите псевдокод функции *rangeQuery* для B-дерева. (См. упражнение 3 из главы 12.) Предполагается, что на экран выводятся только значения поискового ключа, а не вся запись.

8. Напишите псевдокод операций *tableInsert* и *tableDelete* для B-дерева и схемы внешнего хэширования, используя функции управления памятью из упражнения 2. (См. упражнения 3 и 6.)

9. Алгоритм обхода B-дерева, описанный в этой главе, был основан на предположении, что объем оперативной памяти позволяет разместить рекурсивный стек, содержащий h блоков, где h — высота B-дерева. Если это предположение не выполняется, следует модифицировать алгоритм обхода так, чтобы рекурсивный стек содержал лишь номера блоков, а не сами блоки. Сколько обращений к блокам потребуется при выполнении этого алгоритма?

10. Выполните следующие задания.

    10.1. Напишите псевдокод алгоритмов обхода B-дерева и запроса по диапазону значений, в которых обрабатываются не только поисковые ключи, но и вся запись. Сколько обращений к блокам потребуется при выполнении этих алгоритмов?

    10.2. Чтобы сократить количество обращений к блокам, необходимое при выполнении этих операций, часто используются другие варианты B-дерева. Основная идея, лежащая в основе этих структур, состоит в том, что файл данных должен быть предварительно упорядочен. Во-первых, предположим, что файл упорядочен последовательно, в соответствии со значениями поискового ключа, т.е. записи внутри каждого блока упорядочены, причем записи, содержащиеся в блоке B_{i-1}, не превышают записей, содержащихся в блоке B_i для всех i=2,3 и т.д. Перепишите реализации алгоритмов обхода и запроса по значению, пользуясь этим фактом. Сколько обращений к блокам потребуется теперь при выполнении этих алгоритмов?

10.3. Поскольку при частых вставках и удалении записей поддержка упорядоченного файла данных становится крайне неэффективной, часто предлагаются компромиссные схемы. Один из возможных компромиссов заключается в следующем. Если запись данных принадлежит блоку *B* и он полон, создается новый блок, который связывается с блоком *B*. Это позволяет вставлять в файл новую запись, сохраняя порядок следования поисковых ключей. Трудность заключается в том, что теперь нужно просматривать каждую индексную запись в B-дереве, отыскивая первый из нескольких блоков, принадлежащих цепочке, в которой может содержаться искомая запись. Перепишите функции *tableInsert*, *tableDelete*, *tableRetrieve*, *traverseTable* и *rangeQuery* в рамках такой реализации. Как изменится их эффективность?

11. Напишите итеративный вариант алгоритма *internalMergesort*, описанного в главе 9, предназначенный для сортировки внешних файлов. Иными словами, на каждом шаге алгоритма упорядоченные отрезки должны увеличиваться вдвое.

Задания по программированию

1. Выполните следующие задания.

   1.1. Реализуйте алгоритм *externalMergesort* на языке C++, используя функции *seekg* и *seekp*. Предполагается, что упорядочиваемый файл содержит 2^n целых чисел, где n — некоторое целое число, а каждый блок содержит только одно целое число.

   1.2. Теперь предположим, что каждый блок может содержать несколько целых чисел. Напишите функцию на языке C++, реализующую операции *readBlock* и *writeBlock*. Реализуйте алгоритм *externalMergesort* с помощью этих функций.

   1.3. Реализуйте алгоритм *externalMergesort*, сняв ограничение, наложенное на количество блоков.

2. Реализуйте абстрактную таблицу, используя упорядоченный индексный файл, как описано в разделе "Индексирование внешнего файла".

3. Реализуйте абстрактную таблицу, используя хэшированный индексный файл, как описано в разделе "Внешнее хэширование".

4. Реализуйте абстрактную таблицу, используя B-дерево, как описано в разделе "B-деревья".

Приложение А. Основы языка C++

В этом приложении ...

Основные конструкции языка
- Комментарии
- Идентификаторы и ключевые слова
- Основные типы данных
- Переменные
- Литеральные константы
- Именованные константы
- Перечисления
- Оператор typedef
- Присваивания и выражения

Входной и выходной потоки
- Ввод
- Вывод
- Флаги формата и манипуляторы

Функции
- Стандартные функции

Условные операторы
- Оператор if
- Оператор switch

Операторы цикла
- Оператор while
- Оператор for
- Оператор do

Массивы
- Одномерные массивы
- Многомерные массивы
- Массивы массивов

Строки
- Строки языка C++
- Строки языка C

Структуры
- Структуры внутри других структур
- Массивы структур

Исключительные ситуации
- Перехват исключительных ситуаций
- Генерирование исключительных ситуаций

Работа с файлами
- Текстовые файлы
- Бинарные файлы

Библиотеки
- Предотвращение дублирования заголовочных файлов

Сравнение с языком Java

Рузюме

Предупреждения

Вопросы для самопроверки

Упражнения

Задания по программированию

Введение. На протяжении всей книги мы предполагали, что читатели уже владеют каким-либо современным языком программирования. Если этим языком был язык C++, это приложение можно пропустить, обращаясь к нему при необходимости. Если же читатель программирует на языках Java или C, данное приложение позволит ему ознакомиться с основами языка C++.

В рамках краткого обзора невозможно описать все возможности языка C++, поэтому мы сосредоточимся лишь на тех свойствах, которые использовались в книге. Сначала рассмотрим основные типы данных, переменные, выражения, операторы и простые способы ввода/вывода данных. Затем опишем функции, условные выражения, логические конструкции, массивы, строки, структуры, исключения и файлы. Классы языка C++ были описаны в главах 3 и 8, а указатели — в главе 4.

Основные конструкции языка

Начнем с элементов языка, позволяющих выполнять простые вычисления. Например, программа, представленная на рис. А.1, вычисляет объем сферы. При выполнении этой программы на экране появятся следующие строки (входные данные выделены полужирным шрифтом).

```
Введите радиус сферы: 19.1
Объем сферы, имеющей радиус 19.1, равен 29186.927734
```

1. Описание функций ввода и вывода ---------->
2. Начало функции `main` ---------->
3. Комментарий ---------->
4. Начало тела функции ---------->
5. Определение констант ---------->
6. Объявление переменных ---------->
7. Приглашение к вводу данных ---------->
8. Ввод переменной `radius` ---------->
9. Объявление и вычисление `volume` -------->
10. Вывод результатов ---------->
11. Продолжение оператора вывода ---------->
12. Продолжение оператора вывода ---------->
13. Нормальное завершение программы ------>
14. Конец тела функции ---------->

```
#include <iostream.h>
int main()
// Функция для вычисления объема сферы по заданному радиусу.
{
  const double PI = 3.14159;
        double radius;

  cout << "Введите радиус сферы: ";
  cin >> radius;

  double volume = 4 * PI * radius * radius * radius/3;
        cout <<"Объем сферы, имеющей радиус "
    << radius << " дюймов, равен " << volume
    << " кубических дюймов.\n";
  return 0;
}  // end program
```

Рис. А.1. Простая программа на языке C++

Как правило, программа на языке C++ состоит из нескольких модулей, некоторые из них написаны самим программистом, а некоторые являются компонентами стандартных библиотек. В языке C++ предусмотрены возможности **вставки исходных текстов программ** (source-inclusion facility), позволяющие операцион-

ной системе автоматически *вставлять* (include) содержимое указанного файла в заданную точку программы перед компиляцией. Например, программа, показанная на рис. А.1, использует средства ввода и вывода данных, хранящиеся в стандартной библиотеке. Первая строка этой программы представляет собой директиву `include`, где указано имя стандартного заголовочного файла `iostream.h`, в котором содержится описание функций ввода/вывода.

Каждая программа на языке C++ должна содержать функцию main

Программа на языке C++ представляет собой набор функций, одна из которых обязательно должна иметь имя `main`. Выполнение программы всегда начинается с функции `main`. В следующих разделах мы сделаем краткий обзор основ языка C++ на примере программы, представленной на рис. А.1, проанализировав ее строка за строкой. Обратите внимание, что эта программа состоит из единственной функции `main`.

Комментарии

Каждая строка комментария начинается двумя косыми чертами

Каждый комментарий начинается двумя косыми чертами `//` и продолжается до конца строки. Кроме того, можно использовать многострочный комментарий, начинающийся символами `/*` и заканчивающийся символами `*/`. Хотя в нашей книге мы не использовали такие комментарии, при отладке они оказываются очень полезными. Для того чтобы изолировать ошибки, можно на время отключить часть программы, превратив ее в комментарий. Однако внутри такого комментария нельзя помещать другой комментарий, т.е. вложенные многострочные комментарии запрещены.

Идентификаторы и ключевые слова

В языке C++ верхний и нижний регистры букв отличаются

Идентификатор (identifier) в языке C++ представляет собой последовательность букв, цифр и символа подчеркивания. Он должен начинаться либо с буквы, либо с символа подчеркивания. В языке C++ верхний и нижний регистры букв отличаются друг от друга, поэтому, набирая на клавиатуре идентификаторы, следует быть внимательными.

Идентификаторы используются в качестве имен различных сущностей в программе. Однако некоторые идентификаторы в языке C++ зарезервированы в качестве **ключевых слов** (keywords), и их нельзя использовать для других целей. Список всех ключевых слов языка C++ приведен в Приложении Б. Ключевые слова в программах, приведенных в книге, выделяются полужирным шрифтом.

Основные типы данных

Основные типы данных в языке C++ разделяются на четыре категории: булевы, символьные, целые числа и числа с плавающей точкой. За исключением булевого типа, каждая из категорий состоит из нескольких разновидностей. Для большинства приложений достаточно использовать следующие типы:

`bool`	булевы переменные
`char`	символьные переменные
`int`	целые числа
`double`	числа с плавающей точкой

Булева переменная может принимать значения *true* и *false*. Символы, представленные их ASCII-кодами, перечислены в Приложении Б. Целые числа могут иметь знак, т.е. быть равными -5 или +98, а могут его не иметь, например, принимать значения 5 и 98, соответственно. Числа с плавающей точкой позволяют представлять действительные числа, имеющие как целую, так и дробную часть. Булевы, символьные и целочисленные типы называются **интегральными** (integral). Интегральные типы и числа с плавающей точкой образуют категорию **арифметических типов** (arithmetic types) .

Большинство типов данных имеет несколько видов и размеров, однако, как правило, для работы достаточно четырех типов, указанных выше. Для справки на рис. А.2 приведены все типы данных, существующие в языке C++.

Категория	Типы данных		
Булев	`bool`		
Символьный	`char`	`signed char`	`unsigned shar`
Целочисленный со знаком	`short`	`int`	`long`
Целочисленный без знака	`unsigned short`	`unsigned`	`unsigned long`
Число с плавающей точкой	`float`	`double`	`long double`

Рис. А.2. Основные типы данных

Размер типа влияет на диапазон его значений. Например, данные, имеющие тип *long int*, имеют больший диапазон изменения, чем данные типа *short int*. Размер (следовательно, и диапазон) данных зависит от конкретного компьютера и версии языка C++. Однако в языке C++ существует способ, позволяющий определить диапазон изменения данных, с которым мы ознакомимся в разделе "Именованные константы".

Переменные

Переменная, именем которой является идентификатор языка C++, представляет собой ячейку памяти, содержащую значение, имеющее конкретный тип. **Объявление** (declaration) переменной состоит из названия типа, за которым следует ее имя:

```
double radius; // радиус сферы
```

Обратите внимание, что в строке, содержащей объявление переменной, можно поместить и комментарий, описывающий ее предназначение.

Это объявление одновременно является **описанием** (definition) , которое выделяет память для переменной `radius`. Однако в этой памяти не содержится никакого конкретного первоначального значения, поэтому она называется **неинициализированной** (uninitialized) . В программе, представленной на рис. А.1, переменная `radius` объявлена без конкретного первоначального значения. Она получает это значение позднее — с помощью оператора ввода *cin >> radius*. Несколько переменных одного и того же типа можно объявлять в одной и той же строке.

```
int cout, index;
```

По возможности, следует избегать неинициализированных переменных. Это значит, что при объявлении переменной ей следует сразу

Следует избегать неинициализированных переменных

присваивать конкретное значение, либо объявлять ее позднее — при присваивании ее первого значения. Например, в программе, представленной на рис. А.1, переменная *volume* в первый раз появляется в строке 9.

```
double volume = 4 * PI * radius * radius * radius/3;
```

Поскольку переменная *volume* ранее объявлена не была — и, следовательно, мы избежали неинициализированного значения, — она объявляется и получает свое первое значение в одном и том же операторе программы.

Литеральные константы

Литеральные константы используются для обозначения конкретных значений, встречающихся в программе. Примерами литеральных констант являются числа 4 и 3 в строке 9 программы, показанной на рис. А.1. Эти константы используются при вычислении объема сферы. Константы можно также использовать для инициализации переменных. Например, булеву переменную при объявлении можно сразу инициализировать константой *true* или *false*.

Не начинайте десятичную целую константу с нуля

Десятичные целые константы записываются без запятых, десятичных точек и ведущих нулей.[1] По умолчанию эти константы имеют тип *int*, если они достаточно малы, либо тип *long*.

При записи постоянных чисел с плавающей точкой, по умолчанию имеющих тип *double*, используется десятичная точка. Кроме того, для их записи можно использовать так называемый "научный формат", в котором за множителем следует символ *e* или *E*, обозначающий число 10, а за ним — его степень. Например, запись *1.2e-3* означает число 1.2×10^{-3}.

Символьные константы заключаются в одинарные кавычки, например, *'A'* и *'2'*, и по умолчанию имеют тип *char*. **Литеральная символьная строка** (literal symbol string) представляет собой последовательность символов, заключенную в двойные кавычки.

Имена нескольких символов могут начинаться с обратной косой черты, как показано на рис. А.3. Эти обозначения оказываются полезными, если вставить их в литеральную символьную строку. Например, программа, представленная на рис. А.1, использует символ **перехода на новую строку** (new-line character) *\n* в строке *"cubic inches.\n"*. Это значит, что новый вывод начнется со следующей строки. Позднее мы еще столкнемся с этим символом. Кроме того, обратную косую черту следует использовать, если одинарную кавычку нужно интерпретировать как обычный символ (*'\''*), а не как начало литеральной константы или часть двойной кавычки в символьной строке.

Константа	Имя
\n	Новая строка
\t	Табуляция
\'	Одинарная кавычка
\"	Двойная кавычка
\0	Нуль

Рис. А.3. Некоторые специальные символьные константы

[1] Восьмеричные и шестнадцатеричные константы в книге не используются. Восьмеричная константа начинается префиксом *0*, а шестнадцатеричная — префиксом *0x* или *0X*.

Именованные константы

Значение именованной константы не изменяется

В отличие от переменных, которые могут изменять свои значения в ходе выполнения программы, именованные константы являются постоянными. Объявление именованных констант похоже на объявление обычной переменной, однако перед ними указывается ключевое слово *const*. Например, оператор

```
const double PI = 3.14159;
```

в программе, показанной на рис. А.1, объявляет сущность *PI* в качестве именованной константы, имеющей тип *double*. После объявления именованной константы, например, числа *PI*, ей ничего нельзя присвоить. Используя именованные константы, можно повысить читабельность программы, а также облегчить ее дальнейшую модификацию.

Именованные константы повышают читабельность программ и облегчают их модификацию

В стандартном заголовочном файле *limits.h* описаны именованные константы *INT_MIN* и *LONG_MAX*, задающие машинно-зависимый минимум и максимум целочисленных значений. Аналогично, заголовочный файл *float.h* содержит именованные константы, задающие машинно-зависимый минимум и максимум чисел с плавающей точкой. Чтобы получить доступ к этим константам, необходимо включить эти заголовочные файлы в программу с помощью директивы *include*.

Перечисления

Перечисления предоставляют еще один способ именовать константы

Перечисления предоставляют еще один способ именовать целочисленные константы. Например, оператор

```
enum {SUN, MON, TUE, WED, THU, FRI, SAT};
```

эквивалентен операторам

```
const int SUN = 0;
const int MON = 1;
...
const int SAT = 6;
```

По умолчанию значения, присвоенные константам — **счетчикам** (enumerators), — начинаются с нуля и последовательно возрастают. Однако счетчикам можно присвоить явные значения, как в следующем примере.

```
enum { PLUS = '+', MINUS = '-'};
```

Значение счетчика, которому явно ничего не присвоено, на единицу превышает значение предыдущего счетчика. Например, оператор

```
enum { WINTER = 1, SPRING, SUMMER, FALL};
```

присваивает числа 2, 3 и 4 константам *SPRING*, *SUMMER* и *FALL*, соответственно.

Именуя перечисления, можно создавать новые интегральные типы

Именуя перечисления, можно создавать новые интегральные типы. Например, с помощью оператора

```
enum Season {WINTER, SPRING, SUMMER, FALL}
```

создается тип *Season*. Переменная *whichSeason*, объявленная как

```
Season whichSeason;
```

может принимать значения *WINTER* (0), *SPRING* (1), *SUMMER* (2) и *FALL*(3). Такое использование перечислений вместо типа *int* делает программу более читабельной.

Оператор typedef

С помощью оператора *typedef* можно переименовать один из существующих типов данных. Это повышает читабельность программы и облегчает ее модификацию.

Например, оператор

```
typedef double Real;
```

объявляет слово *Real* синонимом ключевого слова *double*, следовательно, оба названия становятся взаимозаменяемыми.

Оператор typedef переименовывает существующий тип данных, повышая читабельность программы и облегчая ее модификацию

Перепишем программу, представленную на рис. А.1, используя тип *Real*.

```
int main()
{
   typedef double Real;
   const Real PI = 3.14159;
   Real radius;

   coun << Введите радиус сферы: ";
   cin >> radius;
   Real volume = 4 * PI * radius * radius * radius/3;
   ...
```

На первый взгляд, эта программа не обладает никакими преимуществами над ее первым вариантом. Однако представьте себе, что в один прекрасный момент нам понадобится повысить точность вычислений. Для этого нужно, чтобы константа *PI*, а также переменные *radius* и *volume* имели тип *long double*, а не *double*. В исходной версии программы (см. рис. А.1) нам понадобилось бы найти и изменить каждое ключевое слово *double* на *long double*. В новом варианте для этого достаточно лишь изменить оператор *typedef*.

```
typedef long double Real;
```

Учтите, что оператор *typedef* не создает новый тип данных, он просто объявляет его новое имя. Для создания нового типа данных недостаточно ввести новое имя; нужно еще определить множество операций над ним. В языке C++ для этого предназначен другой инструмент, описанный в главе 3.

Оператор typedef не создает новый тип данных

Присваивания и выражения

Выражения состоят из переменных, констант, операторов и скобок. В результате выполнения **оператора присваивания**[2] (assignment statement)

```
volume = 4 * PI * radius * radius * radius/3;
```

[2] Традиционно строка программы называется оператором. В языке C++ оператором (operator) называется символ операции (например, +), а строка программы называется statement. Для перевода этого термина в некоторых книгах предлагается слово *инструкция*. Как нам кажется, перепутать оператор + (да и любой другой) со строкой программы невозможно. Поэтому, не стремясь оспорить противоположную точку зрения, мы сохраняем верность установившейся традиции и применяем слово *оператор* в обоих случаях. Кстати, в свое время такая же ситуация сложилась вокруг терминов line (строка программы) и string (символьная строка) в языке C, однако термины *строка* (программы) и *стринг* (символьная строка) не прижились. Это проблема не терминологическая, а сугубо стилистическая. — *Прим. ред.*

ранее объявленная переменная *volume* принимает значение **арифметического выражения** (arithmetic expression), стоящего в правой части оператора. При этом значения константы *PI* и переменной *radius* считаются известными. Оператор присваивания

```
double volume = 4 * PI * radius * radius * radius/3;
```

выполняется одновременно с объявлением новой переменной *volume*.

Ниже мы обсудим разные варианты выражения, которые могут встречаться в операторе присваивания.

Арифметические выражения. Комбинируя переменные и константы с **арифметическими операторами** (arithmetic operators) и скобками, можно получить любое арифметическое выражение. Арифметическими являются следующие операторы.

*	Умножения	+	Бинарное сложение или унарный плюс
/	Деления	-	Бинарное вычитание или унарный минус
%	Остаток от деления		

Операторы *, / и % имеют одинаковый приоритет[3], причем он выше, чем приоритет операторов + и -; приоритет унарных операторов[4] выше приоритета бинарных операторов. Рассмотрим несколько примеров.

Приоритеты операторов

a-b/c	эквивалентно	*a-(b/c)*	(приоритет оператора / выше, чем оператора -)
-5/a	эквивалентно	*(-5)/a*	(приоритет унарного оператора -)
a/-5	эквивалентно	*a/(-5)*	(приоритет унарного оператора -)

Арифметические операторы и большинство других операторов является **левоассоциативными** (left-associative). Это значит, что операторы, имеющие одинаковый приоритет, в выражении выполняются слева направо.

Итак, выражение

Операторы бывают лево- и правоассоциативными

```
a / b * c
```

означает

```
(a / b) * c
```

Оператор присваивания = и все унарные операторы являются **правоассоциативными** (right-associative). Для изменения порядка выполнения операторов и преодоления правил ассоциативности можно применять скобки.

Операторы сравнения и логические выражения. Переменные и константы можно комбинировать со скобками, **операторами сравнения** (comparison operators), или операторами **отношений** (relational operators) <, <=, >= и >, а также с операторами проверки на равенство == (равно) и неравенство != (не равно). Такие выражения имеют значение *false*, если соответствующее отношение не выполняется, и значение *true*, если отношение является истинным. Например, выражение *5 != 4* имеет значение *true*, поскольку число 5 не равно числу 4. Обратите внимание на то, что оператор проверки на равенство имеет более низкий приоритет, чем операторы сравнения.

---

[3] Список всех операторов языка C++ и их приоритеты приведены в Приложении E.

[4] Унарные операторы имеют только один операнд, например, унарным является оператор "-" в записи числа -5. Бинарный оператор имеет два операнда, например, оператор + в выражении 2+3.

С помощью комбинации переменных и констант, имеющих арифметические типы, операторов сравнения и **логических операторов** (logical operators) && ("И") и || ("ИЛИ"), можно образовывать **логические выражения** (logical expressions), которые могут принимать значения *true*, если они истинны, и *false*, если они ложны. В языке C++ логические выражения вычисляются слева направо. Их вычисление останавливается, если значение всего выражения становится очевидным. Этот способ называется **сокращенным вычислением** (short-circuit evaluation).

Иногда значение логического выражения становится очевидным задолго до завершения его вычисления

Например, в языке C++ значение каждого из следующих выражений определяется без вычисления оператора *(a < b)*:

```
(5 == 4) && (a < b)
// false, поскольку выражение (5==4) ложно
(5 == 5) || (a < b)
// true, поскольку выражение (5==5) истинно
```

Условные выражения. Выражение

выражение1 *?* выражение2 *:* выражение3

принимает значение *выражения*$_2$ либо *выражения*$_3$ в зависимости от того, истинно или ложно *выражение*$_3$. Например, оператор

```
larger = ((a > b) ? a : b)
```

присваивает переменной *larger* значение большей из переменных *a* и *b*, поскольку выражение *a > b* является истинным, если значение переменной *a* больше значения переменной *b*, и ложным, если нет.

При выполнении оператора присваивания и вычислении выражений происходит неявное преобразование типов

Неявные преобразования типов. При выполнении оператора присваивания и вычислении выражений происходит автоматическое преобразование одного типа в другой. Перед выполнением оператора присваивания тип выражения, стоящего в правой части, преобразуется в тип элемента, стоящего в левой части. При преобразовании в целый тип числа с плавающей запятой будут усечены (без округления!).

При вычислении выражений любое значение типа *char* или *short* преобразуется в значение типа *int*. Аналогично, значение перечислимого типа преобразуется в значение типа *int*, если тип *int* может представить все значения конкретного перечисления *enum*, в противном случае происходит преобразование в тип *unsigned*. Эти преобразования называются **повышающими** (integral promotions). Если после выполнения этих преобразований операнды имеют разный тип, типы, находящиеся внизу иерархии, преобразуются в типы, стоящие выше.

```
int → unsigned → long → unsigned long → float → double →
long double
```

Например, если переменная *a* имеет тип *long*, а переменная *b* имеет тип *float*, то результат выражения *a + b* будет иметь тип *float*. В переменную типа *float* будет преобразована лишь копия переменной *a*, поэтому ее первоначальное значение останется неизменным.

Для явного преобразования типов следует применять функциональные обозначения

Явные преобразования типов. Для преобразования одного типа в другой существует два способа. Первый и наиболее предпочтительный способ использует запись

```
тип(выражение)
```

Это позволяет преобразовать результат *выражение* в значение, имеющее заданный *тип*. Например, выражение *int(14.9)* преобразует число с плавающей точкой 14.9 в целое число 14. Следовательно, в результате выполнения последовательности операторов

```
double volume = 14.9;
cout << int(volume);
```

на экране появится число 14, а значение переменной *volume* не изменится. Этот способ можно применять, только если желаемый тип имеет имя. Однако неименованный тип всегда можно как-то назвать с помощью оператора *typedef*.

Второй способ использует **приведение типов** (cast). Оператор приведения типов является унарным и образуется путем заключения в скобки имени желаемого типа. Таким образом, в в результате выполнения последовательности операторов

```
double volume = 14.9;
cout << int(volume);
```

на экране также появится число 14.

Рассмотрим еще раз перечисление

```
enum Season {Winter, SPRING, SUMMER, FALL}
```

Переменную типа *Season* можно присвоить переменной, имеющей тип *int*, как было показано выше. В таком случае следующие операторы являются вполне корректными.

```
Season whichSeason = SUMMER;
int    result = whichSeason; // Результат равен 2
```

Однако значение типа *int* нужно явно преобразовать в перечислимый тип, например, в значение типа *Season*. Таким образом, если переменная *index* имеет тип *int*, приведенный ниже оператор является некорректным.

```
whichSeason = index; // Неправильно
```

Он будет некорректным, даже если значение переменной *index* изменяется в диапазоне от 0 до 3. Вместо этого нужно написать выражение

```
whichSeason = Season(index);
```

Явного преобразования типа можно избежать, определив тип *Season* не как перечисление.

```
typedef int Season;
const int WINTER = 0;
const int SPRING = 1;
const int SUMMER = 2;
const int FALL = 3;
```

Множественное присваивание. Оператор присваивания образует **выражение присваивания** (assignment expression). Одно выражение присваивания можно сочетать с другим.

```
a = 5 + (b = 4)
```

Это выражение сначала присваивает число 4 переменной *b*, а затем число 9 присваивается переменной *a*. С одной стороны, такой способ записи способствует лаконичному стилю. Однако, с другой стороны, это часто запутывает программу.

Оператор присваивания является правоассоциативным. Следовательно, выражение `a = b = c` означает `a = (b = c)`.

Другие операторы присваивания. Кроме обычного оператора присваивания, язык C++ предусматривает несколько двухсимвольных операторов присваивания. Например, выражение

```
a += b
```

означает

```
a = a + b
```

Другие операторы, такие как `-=`, `*=`, `/=` и `%=`, имеют аналогичный смысл.

Еще два оператора, `++` и `--`, обеспечивают удобное увеличение и уменьшение значений переменных на единицу. Например, выражение

Операторы ++ и -- используются для увеличения и уменьшения значения переменной на единицу

```
++a
```

означает

```
a += 1
```

что, в свою очередь, эквивалентно выражению

```
a = a + 1
```

Аналогично, выражение

```
--a
```

означает

```
a -= 1
```

что, в свою очередь, эквивалентно выражению

```
a = a - 1
```

Операторы `++` и `--` могут стоять перед своим операндом или после него. Хотя выражение `a++` имеет то же значение, что и `a++`, если эти выражения являются частью оператора присваивания, результат может быть разным. Например, выражение

```
b = ++a
```

означает

```
a = a + 1;
b = a
```

Здесь оператор `++` выполняется *до* присваивания, поэтому переменная `b` будет иметь новое значение. В то же время выражение

```
b = a++
```

означает

```
b = a;
a = a + 1
```

Сначала значение переменной `a` будет присвоено переменной `b`, а затем выполнен оператор `++`. Иначе говоря, оператор `++` выполняется после оператора присваивания. Операторы `++` и `--` часто используются в циклах и для индексации массивов.

Кроме описанных выше, в языке C++ существует еще несколько операторов. Список всех операторов языка C++ и их приоритеты перечислены в Приложении Е.

Входной и выходной потоки

Как правило, программы на языке C++ считывают данные с клавиатуры и выводят их на монитор. Такой способ ввода и вывода данных основан на концепции **потоков** (streams), представляющих собой простую последовательность символов, поступающих с устройства ввода или на устройство вывода.

Входной поток имеет тип *istream*, а выходной — *ostream*. Эти типы описаны в библиотеке ***iostream***, в которой предусмотрены три стандартных потока: *cin* — **входной поток** (standard input stream), *cout* — **выходной поток** (standard output stream) и *cerr* — **поток ошибок** (standard error stream) . Программа получает доступ к потокам после включения заголовочного файла *iostream.h*. Рассмотрим кратко вопросы ввода и вывода данных.

Ввод

Оператор ввода >> считывает данные с входного потока

Для считывания целых чисел, чисел с плавающей точкой и символов в языке C++ предназначен **оператор ввода** >> (input operator). Считанные данные записываются в переменные, имеющие один из основных типов. Левым операндом оператора ввода является входной поток, а правым — переменная, получающая введенное значение. Следовательно, оператор

```
cin >> x;
```

считывает из стандартного входного потока некое значение и присваивает его переменной *x*. Оператор >> является левоассоциативным. Следовательно, выражение

```
cin >> x >> y;
```

означает

```
(cin >> x) >> y
```

Иначе говоря, оба эти выражения считывают из входного потока символы, которые присваиваются переменной *x*, а затем символы, которые предназначены для переменной *y*.

Оператор ввода >> пропускает пробельные символы

Оператор ввода >> игнорирует пробельные символы, такие как пробелы, знаки табуляции и символы перехода на новую строку, которые могут оказаться среди символов входной строки. Например, после выполнения фрагмента программы

```
int ia, ib;
double da, db;
cin >> ia >> da >> ib;
cin >> db;
```

из входного потока будет считана строка

```
21   -3/45   -6   4754.e-2   <cr>
```

Переменная *ia* содержит число 21, *da* — число -3.45, значение переменной *ib* равно -6, а *db* — 4.751. Последовательное считывание данных из потока *cin* прерывается командой *<cr>* (carriage return — перевод каретки). После этого происходит переход на новую строку, и ввод продолжается. Если при попытке чтения оказывается, что данных в потоке нет, либо тип считываемого значения не соответствует типу переменной, возникает ошибка. Допустим, что в приведенном выше фрагменте программы считывается строка

```
-1.23   456/1e-2   -7 8 <cr>
```

Тогда переменная *ia* содержит число -1, *da* — число 0.23, значение переменной *ib* равно 456, а *db* — 0.001, остальные символы входной строки остаются непрочитанными. Если же этот фрагмент программы попытается считать из потока строку, начинающуюся числом *.21*, то ввод прервется. поскольку переменная *ia* имеет тип *int*, а число *0.21* — нет.

Выражения, наподобие, *cin >> x*, имеют результирующее значение. Если операция ввода завершилась успешно, это значение равно *true*, в противном случае оно равно *false*. Это значение можно проверить с помощью условных или итеративных операторов, описанных ниже.

Оператор >> можно применять и для считывания отдельных символов, при этом любой пробельный символ игнорируется. Например, после того как фрагмент программы

```
char ch1, ch2, ch3;
cin >> ch1 >> ch2 >> ch3;
```

прочтет строку

```
xy z
```

переменная *ch1* будет содержать символ *'x'*, переменная *ch2* — символ *'y'*, а переменная *ch3* будет равна *'z'*.

Пробельные символы можно считать как обычно и присвоить переменным, используя функцию *get*.

Операторы

```
cin.get(ch1);
```

Для ввода пробельных символов используется функция get

или

```
ch1 = cin.get()
```

считывают из входного потока в переменную *ch1*, имеющую тип *char*, следующий символ, даже если он является пробелом, знаком табуляции или символом перехода на новую строку.

Описание ввода символьных строк содержится в разделе “Строки”.

Вывод

В языке C++ существует **оператор вывода** << (output operator), предназначенный для вывода символов и значений переменных, имеющих один из основных типов, в выходной поток. Рассмотрим следующий фрагмент программы.

Оператор вывода << записывает данные в выходной поток

```
int count = 5;
double average = 20.3;
cout << "Среднее значение " << count
     << " введенных чисел равно " << average << ".\n"
```

В результате его выполнения в выходной поток будет записана строка

```
Среднее значение 5 чисел равно 20.3.
```

Как и оператор ввода, оператор вывода является левоассоциативным. Следовательно, в ходе его выполнения во входной поток сначала будет записана строка "*Среднее значение*", затем — символы, представляющие собой значение переменной *count*, и т.д.

Использование символа *\n* намного облегчает чтение выходных данных. Обратите внимание, что оператор вывода не расставляет пробелы между записываемыми значениями автоматически, их нужно предусматривать явно. Рассмотрим еще один пример.

> Символ перехода на новую строку и пробельные символы нужно указывать явно

```
int x = 2;
int y = 3;
char ch = 'A';
cout << x << y << ch << "\n"; // Выводит на экран строку 23A
```

Хотя оператор >> позволяет выводить и отдельные символы, для этого предназначена функция *put*. Кроме того, каждый символ можно указывать как переменную типа *char* либо с помощью ASCII-кода. Следовательно, фрагмент программы

```
char ch = 'a';
cout.put(ch);   // Выводит на экран символ a
cout.put('b');  // Выводит на экран символ b
cout.put(99);   // Выводит на экран символ c, ASCII-код равен 99
cout.put(ch+3); // Выводит на экран символ d
cout.put('\n'); // Переход на новую строку
```

выведет на экран строку *abcd* и выполнит переход на следующую строку.

Более подробно вывод строк описывается в разделе "Строки".

Флаги формата и манипуляторы

В языке C++ можно контролировать формат вывода и обрабатывать пробельные символы. Допустим, например, что на экран нужно вывести среднюю оценку, причем число должно содержать плавающую точку и одну цифру в дробной части. Если значение переменной *gpa*, имеющей тип *float*, равно 4.0, то оператор

```
cout << "Моя средняя оценка равна " << gpa << "\n";
```

выведет на экран число 4 без десятичной точки. Повлиять на внешний вид строки вывода можно с помощью **флагов форматирования** (format state flags). Конкретный флаг задается с помощью вызова функции

```
cout.setf(ios::флаг);
```

где опция *flag* принимает значения, перечисленные на рис. А.4. Например, вызов функции

```
cout.setf(ios::showpoint);
```

устанавливает флаг *showpoint*, в результате все действительные числа будут выводиться в выходной поток вместе с точкой. Флаг считается установленным, пока не будет выполнен вызов

```
cout.unsetf(ios::флаг);
```

Флаг	Значение
`fixed`	При выводе чисел с плавающей точкой используется фиксированная десятичная точка
`left`	Выравнивание по левому краю
`right`	Выравнивание по правому краю
`scientific`	Использовать экспоненциальную (научную) запись числа с плавающей точкой
`showpoint`	Вывести десятичную точку
`showpos`	Выводить знак + у положительных чисел

Рис. А.4. Флаги форматирования

Внешний вид выходной строки можно регулировать с помощью манипуляторов

Даже после установки флага *showpoint* значение переменной *gpa* будет выглядеть как 4.000000 вместо 4.0. Уточнить формат вывода можно с помощью **манипуляторов** (manipulators) , которые являются встроенными значениями и функциями, предназначенными для использования в операторах ввода и вывода. Наиболее распространенные манипуляторы перечислены на рис. А.5. Например, с помощью **функции-манипулятора** (manipulator function) *setprecision* можно задать количество цифр, которые следует вывести после запятой, а с помощью **значения-манипулятора** (value manipulator) *endl* — вставить в выходной поток символ перехода на новую строку и очистить буфер вывода. В результате выполнения фрагмента программы

```
cout.setf(ios::showpoint);
cout << setprecision(1) << gpa << endl;
```

на экране появится число 4.0 и будет выполнен переход на новую строку.

Манипулятор	Значение
`endl`	Переход на новую строку и очистка потока
`setfill(`*f*`)`	Дополнить число *f* незначащими нулями
`setprecision(`*n*`)`	Установить количество знаков после точки равным *n*
`setw(`*n*`)`	Установить ширину поля вывода равной *n*
`ws`	Удалить пробельные символы

Рис. А.5. Манипуляторы потока

Действие функции *setprecision* сохраняется, пока не будет выполнен следующий вызов этой функции. Однако, за исключением этой функции, все остальные манипуляторы действуют лишь на следующий символ входной или выходной строки. Например, в результате выполнения фрагмента программы

```
cout.setf(ios::right); // Выравнивание по правому краю
cout << "abc" << setw(6) << "def" << "ghi";
```

на экране появится следующая строка

```
abc   defghi
```

Значения-манипуляторы становятся доступными после включения в программу заголовочного файла *iostream.h*, а функции-манипуляторы — после включения файла *iomanip.h*.

На оператор ввода влияют только один флаг формата и один манипулятор. Флаг *skipws* задается по умолчанию и заставляет оператор ввода игнорировать пробельные символы. Это соглашение можно изменить, вызвав функцию

```
cin.unsetf(ios::skipws);
```

Если флаг *skipws* не установлен, оператор ввода не игнорирует пробельные символы. Однако, используя манипулятор *ws*, можно выборочно пропускать пробельные символы.

Функции

Как указывалось ранее, программа на языке C++ представляет собой совокупность функций. Обычно каждая функция выполняет конкретную задачу. Например, функция *max* возвращает большее из двух целых чисел.

Программа на языке C++ представляет собой совокупность функций

```
int max(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
} // Конец функции max
```

Определение функций реализует ее задачу

Определение функции (function definition) имеет следующий вид

```
тип имя(список объявлений формальных аргументов)
{
   тело
}
```

Раздел определения, предшествующий открывающей фигурной скобке, указывает **тип возвращаемого функцией значения** (return type), ее **имя** (function name) и список **формальных аргументов** (formal agruments) . Часть определения, заключенная в фигурные собки, называется **телом** функции (function's body).

Тело функции, возвращающей какое-либо значение заданного типа, должно содержать оператор

Функция, возвращающая значение, должна содержать оператор return

```
return выражение;
```

Здесь значение *выражения* равно значению, возвращаемому функцией.

Формальные аргументы представляют собой значения, получаемые функцией на вход или возвращаемые ею в результате вычислений. В списке формальных аргументов объявляются их типы и имена, отделенные запятыми, например,

```
int x, int y
```

При **вызове** (call, or invoke) функции *max* ей передаются **фактические аргументы** (actual arguments) , количество, порядок и тип которых соответствует списку формальных аргументов. Например, приведенный ниже фрагмент программы содержит два вызова функции *max*.

При вызове функции ей передаются фактические аргументы, количество, порядок и тип которых соответствует списку формальных аргументов

```
int a, b, c;
cin >> a >> b >> c;

int largerAb = max(a, b);
cout << "Большее из чисел" << a << ", " << b << " и "
     << c << " равно " << max(largerAB, c) << "\n";
```

Как указано выше, определение функции *max* означает, что ее аргументы **передаются по значению** (passed by value). Иначе говоря, функция создает локальные копии значений фактических аргументов — например, переменных *a* и *b* — и использует их внутри тела вместо формальных аргументов *x* и *y*. В данном примере это вполне приемлемо, поскольку функция *max* должна выбрать наибольшее число, не изменяя входные аргументы *x* и *y*.

Если фактический аргумент передается по ссылке, он не копируется. Вместо этого функции предоставляется прямой доступ к нему.

Аргументы функции могут также **передаваться по ссылке** (passed by reference). В этом случае функция не копирует фактические аргументы. Вместо этого она получает прямой доступ к ячейкам памяти, где хранятся фактические аргументы, которые используются в теле функции. Это позволяет изменять значения фактических аргументов внутри тела функции, используя их в качестве результатов вычислений.

В качестве примера рассмотрим следующий вариант функции *max*.

```
void computeMax(int x, int y, int& larger)
{
   larger = ((x > y) ? x : y);
} // Конец функции computeMax
```

Выходной аргумент должен быть ссылкой

Функция *computeMax* не возвращает в вызывающий модуль никаких значений, поскольку типом возвращаемого ею значения является тип *void*.[5] Функция *computeMax* возвращает большее из двух чисел через выходной аргумент *larger*. Символ &, который приписан к типу *int*, означает, что аргумент *larger* **передается по ссылке** (reference argument). Таким образом, функция *computeMax* получит доступ к фактическому аргументу, соответствующему формальному аргументу *larger*, и изменит его. Одновременно эта функция сделает копии аргументов *x* и *y*, которые передаются по значению (value arguments).

Продемонстрируем вызов функции *computerMax*:

```
int a, b, largerAb;
cin >> a >> b;
computeMax(a, b, largerAB);
cout << "Большее из чисел" << a << ", " << b << " и "
     << c << " равно " << max(largerAB, c) << "\n";
```

Входные аргументы должны либо передаваться по значению, либо быть константными аргументами, передаваемыми по ссылке

Если входной аргумент функции представляет собой достаточно крупный объект, его не следует копировать. Следовательно, его нельзя передавать по значению. Однако входной аргумент не должен изменяться функцией. Для этой цели служит **константный аргумент, передаваемый по ссылке** (constant reference argument). Функция, получающая такой аргумент, его не копирует и не изменяет.

---

[5] В то время как функция, возвращающая значение, должна содержать оператор **return** *выражение*; *void*-функция не обязана этого делать. Однако такая функция может содержать оператор *return* ; Такой оператор позволяет вернуть управление в вызывающий модуль. В нашей книге мы не пользуемся этим приемом.

Допустим, заголовок функции *f* выглядит следующим образом.

Аргумент, который одновременно является и входным, и выходным следует передавать по ссылке

```
void f(const int& x, int y, int& z)
```

Здесь аргумент *x* является константным и передается по ссылке, аргумент *y* передается по значению, а аргумент *z* передается по ссылке. Следовательно, аргументы *x* и *y* удобно использовать в качестве входных, поскольку функция *f* не может их изменить, а аргумент *z* является выходным. Иначе говоря, этот аргумент может быть и входным значением, и значением, возвращаемым в качестве результата. Такие аргументы следует просто передавать по ссылке.

Если написать другую функцию *f*, вызывающую функцию *computeMax*, придется либо поместить определение функции *f* после определения функции *max*, либо разместить перед определением функции *f* объявление функции *computeMax* (function declaration). Например, для объявления функции *computeMax* можно использовать один из следующих операторов.

Объявление функции заканчивается точкой с запятой

```
void computeMax(int x, int y, int& max);
```

или

```
void computeMax(int, int, int max);
```

Объявление функции должно содержать типы ее формальных аргументов и возвращаемого значения. Имена аргументов в объявлении функции указывать не обязательно, хотя и рекомендуется. Однако в определении функции имена аргументов являются обязательным элементом. Обратите внимание, что объявление функции заканчивается точкой с запятой, но в определении функции их ставить не следует.

Объявления всех функций обычно помещаются в начало программы.

Как правило, программа на языке C++ содержит объявления каждой вызванной функции. Эти объявления помещаются в начале программы и сопровождаются комментариями, в которых описано предназначение функции, ее аргументы и сделанные предположения. Рассмотрим программу, содержащую объявление функции, ее определение и главный модуль.

```
#include <iostream.h>
int max(int x, int y);
// Возвращает большее из чисел x и y

int main()
{
   int a, b;
   cout << "Введите, пожалуйста, два целых числа: ";
   cin >> a >> b;

   int largerAB = max(a, b);
   cout << "Большее из чисел" << a << ", " << b << " и "
     << c << " равно " << max(largerAB, c) << "\n";
} // Конец функции main

int max(int x, int y)
{
   return (x > y) ? x : y
} // Конец функции max
```

Стандартные функции

Стандартные функции предназначены для выполнения наиболее распространенных операций. Для их использования нужно указать конкретный заголовочный файл

В языке C++ есть много стандартных функций, таких как функция извлечения квадратного корня *sqrt* и функция ввода *get*. Обзор стандартных функций и заголовочных файлов, которые необходимо включать в программу для их применения, содержится в Приложении В. Например, стандартные функции, перечисленные на рис. А.6, предназначены для обработки символов. Для их применения необходим заголовочный файл *ctype.h*. Следовательно, если в программе необходимо вызвать такие функции, как *isupper* или *toupper*, в нее необходимо вставить директиву

```
#include <ctype.h>
```

Если символьная переменная *ch* представляет собой прописную букву, то вызов функции *isupper(ch)* вернет значение *true*. Вызов *toupper(ch)* вернет прописной вариант буквы *ch*, не изменяя саму переменную *ch*.

а)

Функция	Возвращает `true` если `ch` - это...
`isalnum(ch)`	Буква или цифра
`isalpha(ch)`	Буква
`isdigit(ch)`	Цифра
`islower(ch)`	Строчная буква
`isupper(ch)`	Прописная буква

б)

Функция	Возвращает
`tolower(ch)`	Строчный вариант буквы `ch`
`toupper(ch)`	Прописной вариант буквы `ch`
`toascii(ch)`	Целочисленный ASCII-код буквы `ch`

Рис. А.6. Стандартные функции: а) стандартные функции классификации; б) стандартные функции преобразования

Условные операторы

Условные операторы позволяют выбрать разные варианты действий в зависимости от значения некоторого выражения. К этой категории операторов относятся операторы *if* и *switch*.

Оператор if

Существуют два вида оператора if

Оператор *if* можно записать двумя способами:

```
if (выражение)
    оператор1
```

или

```
if (выражение)
    оператор1
else
   оператор2
```

Здесь *оператор*$_1$ и *оператор*$_2$ представляют собой любой оператор программы на языке C++ или объявление. Такие операторы могут быть **составными** (compound). Составной оператор, или **блок** (block) представляет собой последовательность операторов, заключенную в фигурные скобки. Если *выражение* имеет значение `true`[6], выполняется *оператор1*. В противном случае в первом варианте оператора `if` ничего не происходит, а во втором — выполняется *оператор2*. Обратите внимание, что условное выражение должно быть заключено в круглые скобки.

Условное выражение в операторе if должно быть заключено в круглые скобки

Например, каждый из операторов `if`, приведенных ниже, сравнивает значения двух целочисленных переменных *a* и *b*.

```
if (a > b)
   cout << a << " больше, чем " << b << ".\n";
   cout << "Этот оператор выполняется всегда.\n";

if (a > b)
{
   largerAB = a;
   cout << a << " больше, чем " << b << ".\n";
}

else
{
   largerAB = b;
   cout << b << " больше, чем " << a << ".\n";
}
   cout << largerAB << " является наибольшим значением.\n";
```

Операторы `if` можно по-разному вкладывать друг в друга, поскольку и *оператор*$_1$, и *оператор*$_2$, в свою очередь, могут быть условными операторами `if`. Рассмотрим пример, в котором выбирается большая из трех целочисленных переменных *a*, *b* и *c*.

```
if ((a >= b) && (a >= c))
   largest = a;
else if (b >= c)
// Переменная a не больше остальных
   largest = b;
else
   largest = c;
```

Операторы if можно вкладывать друг в друга

Оператор switch

Если нужно сделать выбор из нескольких вариантов, оператор `if` становится слишком громоздким. Если выбор зависит от значения целочисленного выражения, можно применить оператор `switch`.

Оператор switch позволяет сделать выбор из нескольких вариантов в соответствии со значением целочисленного выражения

[6] Неотрицательные арифметические величины считаются эквивалентами значения `true`.

Например, приведенный ниже фрагмент программы определяет количество дней в заданном месяце. Целочисленная переменная *month* задает порядковый номер месяца от 1 до 12.

```
switch (month)
{
   // Сентябрь, апрель, июнь и ноябрь имеют по 30 дней
case 9: case 4: case 6: case 11:
   daysInMonth = 30;
   break;

// Если не указан оператор break, оператор switch выполнит
// вариант, соответствующий следующей метке варианта

// Все остальные месяцы состоят из 31 дня
case 1: case 3: case 5: case 7:
case 8: case 10: case 12:
   daysInMonth = 31;
   break;

// Исключением является февраль
case 2: // Переменная leapYear имеет значение true,
        // если текущий год является високосным,
        // в противном случае она имеет значение false
   if (leapYear)
      daysInMonth = 29;
   else
      daysInMonth = 28;
   break;

default:
   cout << "Неправильный номер месяца.\n";
} // Конец оператора switch
```

Целочисленное выражение в операторе *switch* должно быть заключено в круглые скобки. Метки вариантов имеют вид:

```
case выражение:
```

Здесь *выражение* является константным и целочисленным. После вычисления выражения, входящего в оператор *switch*, выполнение программы продолжается с метки *case*, соответствующей его значению. Все следующие операторы выполняются вплоть до оператора *break* или *return*, которые приводят к выходу из оператора *switch*.

Если вариант *case* не завершается операторами *break* или *return*, выполнение оператора *switch* продолжается. Иногда это свойство оказывается полезным, но следует иметь в виду, что ошибочный пропуск оператора *break* может изменить ход выполнения оператора *switch*.

Если значению выражения, входящего в оператор *switch*, не соответствует ни одна метка *case*, выполняются операторы, указанные после метки *default*, если эта метка предусмотрена в операторе. Если этой метки нет, происходит выход из оператора *switch*.

Операторы цикла

В языке C++ есть три оператора цикла — *while*, *do* и *for*. Они предназначены для повторения итерации, т.е. для выполнения циклических вычислений. Каждый оператор контролирует количество повторных выполнений другого оператора, называемого **телом** цикла (body). Тело цикла не может быть объявлением и чаще всего представляет собой составной оператор.

Оператор while

Общая форма оператора *while* такова.

```
while (выражение)
      оператор
```

Оператор while выполняется до тех пор, пока выражение является истинным

Оператор выполняется, пока *выражение* имеет значение *true*. Поскольку *выражение* вычисляется до выполнения *оператора*, он может вообще ни разу не выполняться. Обратите внимание, что *выражение* должно быть заключено в скобки.

Допустим, нам нужно вычислить сумму положительных целых чисел, которые вводятся с клавиатуры. Поскольку эти числа положительны, отрицательные числа и число нуль можно использовать в качестве признака конца списка. Посмотрим, как эта задача решается с помощью оператора *while*.

```
int nextValue;
int sum = 0;

cin >> nextValue;
while (nextValue > 0)
{
   sum += nextValue;
   cin >> nextValue;
} // Конец оператора while
```

Если первым считывается число 0, тело цикла никогда не выполняется.

Напомним, что выражение *cin >> nextValue* имеет значение *true*, если оператор ввода выполнен успешно, в противном случае оно равно значению *false*. Таким образом, операторы, приведенные выше, можно записать иначе.

```
int nextValue;
int sum = 0;

while ( (cin >> nextValue) && (nextValue > 0) )
   sum += nextValue;
```

Использовать оператор break в теле цикла не рекомендуется

Операторы break и continue. Оператор *break* можно использовать для прерывания выполнения оператора цикла, аналогично оператору *switch*. Выполнение оператора *break* в теле цикла приводит к немедленному выходу из цикла. В этом случае выполнение программы будет продолжено с оператора, следующего сразу за циклом. Учтите, что применение оператора *break* в теле циклов *while*, *do* и *for* является признаком дурного тона.

Оператор *continue* останавливает только текущее выполнение цикла и начинает следующую итерацию с начала. Этот оператор можно использовать только внутри тела циклов *while*, *do* и *for*.

Оператор for

Оператор *for* позволяет выполнять перечислимые циклы. Он имеет следующий вид.

Оператор for концентрирует операторы инициализации, проверки и изменения счетчика в одном месте

```
for (инициализация; проверка;
     обновление счетчика)
   оператор
```

Элементы цикла *инициализация*, *проверка* и *обновление счетчика* являются выражениями. Обычно *инициализация* представляет собой выражение присваивания, которое задает начальное значение счетчика, контролирующего выполнение цикла. Инициализация выполняется только один раз. *Проверка* обычно является логическим выражением. Если оно имеет значение *true*, выполняется *оператор*. *Обновление счетчика* выполняется в конце списка. Обычно оно сводится к увеличению или уменьшению значения счетчика. Эта последовательность вычислений повторяется до тех пор, пока результат *проверки* не станет ложным.

Рассмотрим пример, в котором оператор *for* выводит на экран целые числа от 1 до *n*.

```
for (int counter = 1; counter <= n; ++counter)
   cout << counter << " ";

   cout << endl; // Этот оператор выполняется всегда
```

Если число *n* меньше 1, оператор *for* не будет выполнен ни разу. Следовательно, приведенный выше фрагмент программы эквивалентен следующему оператору *while*.

```
int counter = 1;
while (counter <= n)
{
   cout << counter << " ";
   ++counter;
} // Конец оператора while

cout << endl; // Этот оператор выполняется всегда
```

Оператор for эквивалентен оператору while

Обычно логику оператора *for* можно представить в эквивалентном виде с помощью оператора *while*.

```
инициализация;
while (проверка)
{
   оператор;
   обновление счетчика;
}
```

Подразумевается, что *оператор* содержит в себе оператор *continue*, а *обновление счетчика* выполняется до *проверки*.

Обратите внимание, что выражение *инициализации* должно иметь либо арифметический, либо указательный тип.[7] Рассмотрим примеры, демонстрирующие гибкость оператора *for*.

[7] Указательные типы введены в главе 4.

```
for (char ch = 'z'; ch >= 'a'; --ch)
// Значение счетчика ch изменяется от 'z' до 'a'
for (double x = 1.5; x < 10; x += 0.25)
// Значение счетчика x изменяется от 1.5 до 9.75 с шагом 0.25
```

Инициализация и *обновление счетчика*, в свою очередь, могут состоять из нескольких выражений, разделенных скобками. Например, приведенный ниже цикл возводит действительное число в целую степень, используя умножение.

```
// Переменная power равна числу x, возведенному в степень n.
// Переменная expon считается целочисленной
for (power = 1.0, expon = 1; expon <= n; ++expon)
power *= x;
```

Переменные *power* и *expon* получают свои значения до первого выполнения тела цикла. Запятая в списке операторов сама является **оператором запятой** (comma operator), который вычисляет выражения, являющиеся его операндами слева направо.

Поскольку оператор *for* концентрирует операторы инициализации, проверки и изменения счетчика в одном операторе, программисты на языке C++ отдают предпочтение ему, а не оператору *while*. Например, обратите внимание, как оператор *for* вычисляет сумму положительных чисел.

Оператор for предпочтительнее оператора while

```
for (int sum = 0;
     (cin >> nextValue) && (nextValue > 0);
     sum += nextValue);
```

Фактически тело этого оператора пусто!

Выражения инициализации, проверки и обновления счетчика можно пропускать, но точки с запятой игнорировать нельзя. Например, в операторе *for*, приведенном выше, выражение обновления счетчика можно перенести в тело цикла.

Выражения инициализации, проверки и обновления счетчика можно пропускать как по отдельности, так и все сразу, но точки с запятой забывать нельзя

```
for (int sum = 0; (cin >> nextValue) &&
                  (nextValue > 0); )
sum += nextValue;
```

Кроме того, можно одновременно пропустить выражения инициализации и обновления счетчика. Тогда цикл, вычисляющий сумму положительных чисел, примет следующий вид.

```
for ( ; (cin >> nextValue) && (nextValue > 0); )
{
   Операторы, вычисляющие значение nextValue
}
```

Этот оператор *for* уже не имеет преимуществ перед оператором *while*.

```
while ( (cin >> nextValue) && (nextValue > 0) )
```

Можно даже пропустить *проверку* выхода из цикла, но делать этого не рекомендуется, поскольку тогда цикл станет бесконечным.

Оператор do

Оператор *do* следует использовать тогда, когда цикл должен быть выполнен хотя бы один раз. Общая форма этого оператора такова.

Оператор do выполняется по крайней мере один раз

```
do
   оператор
while (выражение);
```

Здесь *оператор* вычисляется до тех пор, пока *выражение* не станет ложным.

Допустим, что нам нужно выполнить последовательность операторов, а затем спросить у пользователя, стоит ли продолжать. В этой ситуации нам подойдет оператор *do*, поскольку операторы выполняются до того, как пользователь примет решение.

```
char response;
do
{
  . . . (последовательность операторов)
  cout << "Повторить?";
  cin >> response;
} while ( (response == 'Y') || (response == 'y') );
```

Массивы

Массив — это совокупность **элементов**, или **компонентов**, имеющих одинаковый тип. Элементы массива упорядочены: существует первый, второй и т.д. элементы. Таким образом, массив состоит из конечного числа элементов. Следовательно, при создании программы необходимо знать максимальное количество элементов массива еще *до* ее выполнения.

Массив представляет собой совокупность данных, имеющих одинаковый тип.

Поскольку программист имеет непосредственный доступ к элементам массива, говорят, что массив является структурой данных с **прямым** (direct access), или **произвольным доступом** (random access).

Массив предоставляет непосредственный доступ к своим элементам

Одномерные массивы

Решив применить массив, его необходимо объявить, указав тип его элементов и максимальный размер. Приведенные ниже операторы объявляют **одномерный массив** (one-dimensional array) *maxTemps*, содержащий значения максимальной температуры, измеряемой ежедневно в течение недели.

```
const int DAYS_PER_WEEK = 7;
double maxTemps[DAYS_PER_WEEK];
```

Квадратные скобки [] объявляют переменную *maxTemps* в качестве массива. Этот массив может содержать не более семи чисел с плавающей точкой.

На элементы массива *maxTemps*, имеющие тип *double*, можно ссылаться с помощью выражения, называемого **индексом** (index, или subscript), которое должно заключаться в квадратные скобки. В языке C++ ин-

Для ссылки на конкретный элемент массива используется индекс

дексы массива могут быть только целыми числами, изменяющимися от 0 до *size*-1, где *size* — количество элементов в массиве. Индексы массива *maxTemps* изменяются от 0 до *DAYS_PER_WEEK*-1. Например, элемент *maxTemps[4]* является пятым элементом массива. Если значение целочисленной переменной *k* равно 4, то элемент *maxTemps[k]* окажется пятым элементом массива, а *maxTemps[k+1]* — шестым. Кроме того, выражение *maxTemps[++k]* добавляет сначала единицу к переменной *k*, а затем использует это значение в качестве нового индекса, в то время как выражение *maxTemps[k++]* сначала обращается к элементу *maxTemps[k]*, а затем увеличивает переменную *k* на единицу.

На рис. А.7 показан массив *maxTemps*, содержащий только пять чисел. Последним значением, хранящимся в массиве *maxTemps*, является элемент *maxTemps[4]*; элементы *maxTemps[5]* и *maxTemps[6]* остаются неопределенными.

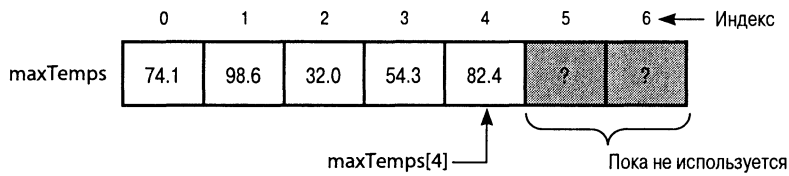


Рис. А.7. Одномерный массив, содержащий не более семи элементов

Для индексации массива можно использовать переменные перечислимого типа, поскольку они также принимают целочисленные значения. Рассмотрим следующее определение.

```
enum Day {SUN, MON, TUE, WED, THU, FRI,
          SAT};
```

В качестве индекса массива можно использовать перечисление

В таком случае выражение *maxTemps[THU]* имеет тот же смысл, что и выражение *maxTemps[4]*. Перечисления можно также использовать в качестве значения счетчика цикла, в котором обрабатывается массив. Рассмотрим в качестве примера цикл *for*, предполагая, что его счетчик *dayIndex* имеет тип *Day*.

```
for (dayIndex = SUN; dayIndex <= SAT;
                     dayIndex = Day(dayIndex+1))
cout << maxTemps[dayIndex] << endl;
```

Выражение *DAY(dayIndex+1)* преобразует целочисленную сумму *dayIndex+1* в интегральный тип *Day*.

Очевидно, что перед извлечением элемента массива необходимо присвоить ему какое-то значение. Присвоить значения элементам массива можно с помощью индексов, описанных выше. Обратите внимание, что, даже если массивы *a* и *b* имеют одинаковый тип, выражение *a=b* является некорректным.[8]

Тип данных *maxTemps* является **производным** (derived), поскольку он выводится из основных типов с помощью **оператора объявления** (declaration operator), в данном случае — с помощью оператора `[]`. Часто бывает полезно переименовать производный тип, пользуясь оператором *typedef*. Итак, можно написать следующий фрагмент программы.

[8] В языке C++ можно описать свой собственный тип, имитирующий массив, в котором такое выражение присваивания будет корректным. Для этого следует использовать классы (глава 3) и перегруженные операторы (глава 8).

```
const int DAYS_PER_WEEK = 7;
typedef double ArrayType[DAYS_PER_WEEK];
ArrayType maxTemps;
```

Это позволяет использовать переменные типа *ArrayType* в любом месте программы.

Инициализация. Элементы массива можно проинициализировать при объявлении.

Например, в объявлении

```
ArrayType maxTemps = {82.0, 71.5, 61.8,
                      75.0, 88.3};
```

Элементы массива можно проинициализировать при объявлении

инициализируется массив *maxTemps*, первые пять элементов которого имеют значения, перечисленные в фигурных скобках, а остальные равны нулю.

Передача массива в качестве аргумента функции. Допустим, что для вычисления среднего значения *n* первых элементов одномерного массива мы хотим применить функцию. Такую функцию можно объявить следующим образом.

```
double averageTemp(ArrayType temperatures, int n);
```

Поскольку компилятору не нужно знать максимальный размер массива, объявление функции можно немного изменить.

```
double averageTemp(double temperatures[], int n);
```

В любом случае эту функцию можно вызвать, написав оператор

```
double avg = averageTemp(maxTemps, 6);
```

Массив *maxTemps*, упоминающийся здесь, должен быть объявлен заранее.

Массивы никогда не передаются по значению, независимо от способа их объявления в качестве формального аргумента функции. Массивы всегда передаются по ссылке. Это позволяет не копировать элементы массива, количество которых может быть большим. Итак, функция *averageTemp* может модифицировать элементы массива *maxTemps*, если массив объявлен как входной аргумент (без указания символа &. — *Прим. ред.*) Чтобы предотвратить модификацию элементов массива, его следует сделать константным формальным аргументом, поместив перед именем его типа ключевое слово *const*.

Массивы всегда передаются по ссылке

```
double averageTemp(const double temperatures[], int n);
```

Многомерные массивы

Одномерный массив, имеющий один индекс, можно использовать в качестве простой совокупности данных. Например, можно линейно упорядочить 52 значения температуры, разместив их одна за другой. Именно такой способ организации представляет собой одномерный массив.

Кроме того, можно объявить **многомерные массивы** (multidimensional arrays). Для доступа к элементу многомерного массива одного индекса уже мало. Допустим, что нам нужно представить совокупность минимальных значений температуры, ежедневно измеряемой в течение 52 недель. С помощью приведенных ниже операторов мы можем объявить **двумерный массив** *minTemps*.

Массив может иметь несколько измерений

```
const int DAYS_PER_WEEK = 7;
const int WEEKS_PER_YEAR = 52;

typedef double ArrayType[DAYS_PER_WEEK][WEEKS_PER_YEAR];
ArrayType minTemps;
```

Эти операторы указывают диапазоны изменения двух индексов: первый индекс может изменяться от 0 до 6, а второй — от 0 до 51. Большинство людей представляют себе двумерный массив в виде **матрицы** (matrix), элементы которой образуют строки и столбцы, как показано на рис. А.8. Первая размерность, указанная в описании типа *ArrayType*, означает количество строк. Таким образом, массив *minTemps* состоит из 7 строк и 52 столбцов. Каждый столбец матрицы представляет собой семь ежедневных минимумов температуры, измеренных на протяжении конкретной недели.

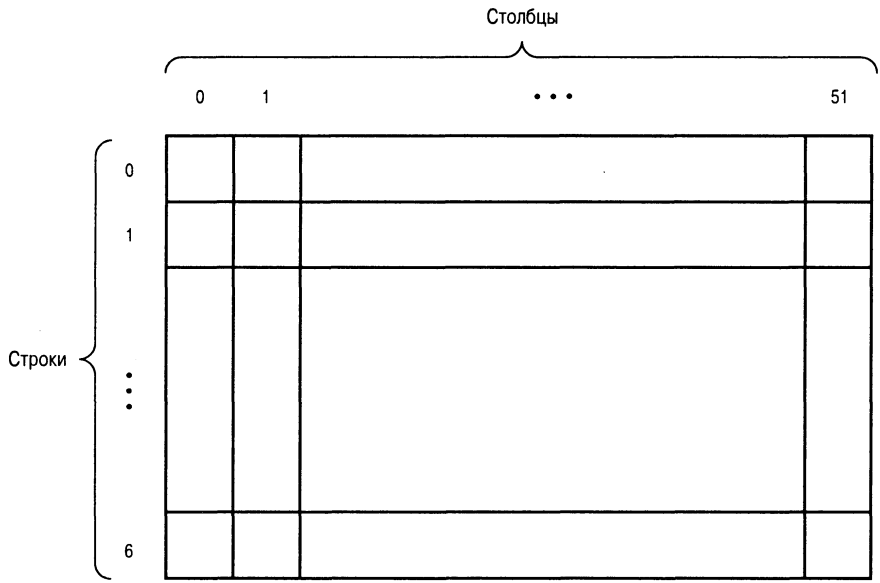


Рис. А.8. Двумерный массив

В двумерном массиве первый индекс представляет собой номер строки, а второй — номер столбца

Для ссылки на элемент двумерного массива нужно указать номер строки и столбца, в которых он расположен. Для этого можно использовать два индекса, каждый из которых заключен в квадратные скобки. Например, выражение *minTemps[1][51]* означает элемент, расположенный во 2-й строке и 52-м столбце. В нашем примере этот элемент означает минимальную температуру, измеренную во второй день (понедельник) 52-й недели. Правила индексации, применявшиеся к одномерному массиву, распространяются и на многомерные массивы.

В качестве примера использования двумерного массива, рассмотрим следующий фрагмент программы, в котором определяется наименьший элемент массива *minTemps*. Для обозначения дней недели используется перечисление

```
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
```

```
// Переменная minTemps представляет собой двумерный массив
// минимальных значений температуры, измеренных ежедневно
// на протяжении 52 недель. Каждый столбец соответствует
// отдельной неделе.

// В качестве первоначальной гипотезы наименьшим считается
// первый элемент массива.
double lowestTemp = minTemps[0][0];
Day dayOfWeek = SUN;
int weekOfYear = 1;

// search array for lowest temperature
for (int weekIndex = 0; weekIndex < WEEKS_PER_YEAR;
                        ++weekIndex)
   for (Day dayIndex = SUN; dayIndex <= SAT;
                            dayIndex = Day(dayIndex+1))
      if (lowestTemp > minTemps[dayIndex][weekIndex])
      {
         lowestTemp = minTemps[dayIndex][weekIndex];
         dayOfWeek = dayIndex;
         weekOfYear = weekIndex+1;
      } // Конец операторов if и for
// Диагностическое утверждение: переменная lowestTemp содержит //
значение наименьшей температуры, содержащееся в массиве
// minTemps, а переменные dayOfWeek и weekOfYear — день и
// неделю, когда это значение было измерено, т.е.
// lowestTemp == minTemps[dayOfWeek][weekOfYear-1].
```

Переменную *minTemps* можно было бы объявить и как одномерный массив, содержащий 364 элемента (7 * 52). В этом случае для доступа к элементу *minTemps[4][11]* нужно было бы написать выражение *minTemps[81]*. (Почему? См. упражнение 5.) Однако в этом случае программа стала бы менее понятной!

Хотя массивы могут иметь более высокие размерности, на практике размерность выше трех используется очень редко. Однако способы работы с такими массивами ничем не отличаются от методов работы с двумерными массивами.

Инициализация. Элементы двумерного массива можно инициализировать так же, как и одномерный массив. Для этого нужно заполнить список инициализации построчно, например, операторы

```
typedef int ArrayType[2][3]; // 2 строки, 3 столбца
ArrayType x = {1,2,3,4,5,6};
```

инициализируют двумерный массив *x* следующим образом.

```
1  2  3
4  5  6
```

Иначе говоря, операторы последовательно инициализируют элементы *x[0][0]*, *x[0][1]*, *x[0][2]*, *x[1][0]*, *x[1][1]* и *x[1][2]*. Как правило, при инициализации многомерного массива последний, или правый индекс увеличивается первым.

Массивы массивов

Вернемся к примеру о минимальных значениях температуры. Вместо двумерного массива объявим 52 одномерных массива, состоящих из 7 элементов.

```
typedef double WeekType[DAYS_PER_WEEK];
typedef WeekType YearType[WEEKS_PER_YEAR];

YearType temps;
```

Здесь переменная *temp* означает массив массивов, в котором элемент *temps[11]*, например, является массивом, состоящим из 7 значений температур, измеренных на протяжении 12-й недели. Если данные нужно хранить по неделям, такая организация массива предпочтительнее матрицы. Например, элемент *temps[11]* можно передать функции, которая вычисляет среднее значение по элементам одномерного массива. (См. упражнение 7.)

Массив массивов использует такое же представление и способы индексирования, что и многомерный массив

Значение температуры, измеренное на 5-й день 12-й недели, хранится в ячейке *minTemps[11][4]*. Здесь элемент *temps[11]* является массивом, а элемент *temps[11][4]* — его 5-м компонентом. В языке C++ для многомерных массивов и массивов, состоящих из массивов, используются одинаковые обозначения. Фактически язык C++ не делает различий между многомерными массивами и массивами массивов. Следовательно, способ представления данных должен зависеть от алгоритма. Если важнее представлять отдельные данные по дням, то предпочтительнее становится многомерный массив, в противном случае — массив массивов.

Инициализация. Элементы массива массивов инициализируются аналогично многомерному массиву. Если программа содержит определения

```
typedef int VectorType[3];
typedef VectorType ArrayType[2];
```

то оператор

```
ArrayType x = {{1,2,3},{4,5,6}};
```

инициализирует массив *x[0]* элементами *{1, 2, 3}*, а массив *x[1]* — элементами *{4, 5, 6}*. Оператор

```
ArrayType x = {1,2,3,4,5,6};
```

приводит к тому же результату, поскольку операторы *typedef*, приведенные выше, эквивалентны оператору

```
typedef int ArrayType[2][3];
```

Строки

Ранее мы уже встречались с литеральными символьными строками, например:

```
"Это — строка."
```

В этом разделе мы покажем, как создавать и использовать переменные, содержащие такие строки. В языке C++ есть строковый тип, позволяющий манипулировать строками так же естественно, как и целыми числами, используя хорошо знакомые операторы. В языке C, который является подмножеством языка C++, строки представляют собой одномерный массив символов. Когда нужно провести различие между двумя этими разновидностями строк, мы будем называть их **строками языка C** (C string) и **строками языка C++** (C++ string) соответственно.

Здесь мы рассмотрим далеко не все операции, которые предусмотрены для строк.

Строки языка C++

Стандартная библиотека языка C++ содержит описание типа *string*. Используя эту библиотеку, можно объявлять и применять переменные, содержащие строки. Для этого достаточно включить в программу следующие директиву и оператор.

```
#include <string>
using namespace std;[9]
```

Можно объявить строковую переменную *title* и инициализировать ее пустой строкой.

```
string title;
```

Строковую переменную можно инициализировать строковым литералом, написав оператор

```
string title = "Стены и зеркала";
```

Впоследствии переменной *title* можно присвоить другую строку, используя оператор присваивания.

```
title = "Гекльберри Финн";
```

Каждая из этих строк состоит из 16 символов. В этом случае говорят, что длина строки *title* равна 16. Для вычисления текущей длины строки можно применять либо функцию *length*, либо функцию *size*. Таким образом, значения *title.length()* и *title.size()* равны 16. Ссылаться на отдельные символы строки можно с помощью индексов, как будто строка представляет собой массив. Таким образом, в предыдущем примере ячейка *title[0]* содержит символ *'Г'*, а ячейка *title[15]* — символ *'н'*.

Строки можно сравнивать, используя обычные операторы сравнения. Причем можно проверять не только равенство, но и какая из строк предшествует другой. Строки упорядочиваются в порядке следования ASCII-кодов их символов. Таким образом, все приведенные ниже отношения являются истинными.

Примеры истинных выражений

```
"dig" < "dog"
"Star" < "star" (поскольку 'S' < 's')
"start" > "star"
"d" > "abc"
```

Строки можно конкатенировать, используя оператор +. В результате образуется новая строка, состоящая из двух частей: первой и второй строки, записанных последовательно. Например, если в программе поместить объявление

```
string str1 = "Com";
```

то операторы

```
string str2 = str1 + "puter";
str1 += "puter";
```

присвоят переменным *str1* и *str2* строку *"Computer"*. Аналогично, к строке можно приписать отдельный символ

```
str1 += 's';
```

[9] Описание пространства имен см. в главе 3.

Используя функцию

```
substr(позиция, длина)
```

Функция substr позволяет выделить часть строки

можно манипулировать частями строки. Ее первый аргумент определяет позицию начала подстроки (помните, что первый символ строки находится на *нулевой* позиции). Второй аргумент задает длину подстроки. Например, результатом вызова

```
title.substr(10,4)
```

является слово *"Финн"*.

Чтобы выполнить ввод и вывод строк языка C++, нужно использовать современную версию библиотеки *iostream*. Для этого необходимо включить в программу следующие директиву и оператор

```
#include <iostream>
using namespace std;
```

Для вывода строки можно использовать оператор <<

Эта версия библиотеки *iostream* работает точно так же, как и старая библиотека *iostream.h*. Следовательно, оператор << можно использовать для вывода на экран как литеральной строки, так и строковой переменной. Например, фрагмент программы

```
title = "Стены и зеркала";
cout << "\"" << title << "\"\n";
```

выведет на экран строку *"Стены и зеркала"*. Обратите внимание на специальный символ *\"*, стоящий внутри литеральной строки. Он предназначен для вывода на экран двойной кавычки. Оператор << выводит на экран всю строку, включая пробелы.

Для ввода строки, не содержащей пробельных символов, можно применять оператор >>

В символьную переменную можно считать строку символов. Если оператор

```
cin >> title;
```

считает строку ввода

```
Гекльберри Финн
```

он присвоит переменной *title* строку *"Гекльберри"*. Пробельные символы во входной строке прерывают операцию чтения. Для ввода строки вместе с пробельными символами, нужно использовать функцию *getline(cin, title)*.

Строки языка C

Строкой называется массив символов, заканчивающийся нулем

Строку можно представить в виде одномерного массива, заканчивающегося нулевым символом *\0*. Итак, массив, представляющий строку *"abc"*, на самом деле содержит не три, а четыре символа. Это использование нулевого символа в качестве стража позволяет стандартным библиотечным функциям манипулировать со строками. Чтобы использовать эту библиотеку, нужно включить в программу директиву

```
#include <string.h>
```

Далее мы рассмотрим несколько функций из этой библиотеки.

Переменную, содержащую строку языка C, можно объявить следующим образом.

```
const int MAX_LENGTH = 30;
typedef char StringType[MAX_LENGTH+1];
StringType title;
```

Переменная *title* представляет собой массив символов, который можно считать строкой. Хотя при объявлении строки мы должны учитывать завершающий нулевой символ, говоря о длине строки, его удобно просто игнорировать. Итак, **максимальная длина** (maximal length) строки *title* задается константой *MAX_LENGTH*, хотя массив *title* состоит из *MAX_LENGTH*+1 компонента. Максимальная длина строки определяется во время компиляции и, следовательно, остается фиксированной. Строковые переменные имеют также **текущую длину** (current length), равную количеству символов, образующих строку в данный момент, не считая нулевого символа. Текущая длина строки является переменной величиной — она изменяется при изменении строки — и находится в диапазоне от нуля (пустая строка) до максимальной длины.

Строковую переменную, например, можно инициализировать во время объявления, присвоив ей строковый литерал.

```
StringType title = "Стены и зеркала";
```

Эту синтаксическую конструкцию можно применять только при объявлении переменной. Если позднее переменной *title* понадобится присвоить новое значение, следует вызывать стандартную функцию *strcpy*.

```
strcpy(title, "Гекльберри Финн");
```

Эта функция заменяет содержимое первого аргумента содержимым второго аргумента. При этом необходимо, чтобы текущая длина второго аргумента не превышала максимальную длину первого аргумента.

В каждом из рассмотренных ранее примеров длина строки *title* была равна 16. Для определения текущей длины строки вызывается функция *strlen*. Таким образом, значение *strlen(title)* равно 16.

Ссылаться на отдельные символы строки можно с помощью индексов, как будто строка представляет собой массив. Таким образом, как и раньше, ячейка *title[0]* содержит символ *'Г'*. Обратите внимание, что в данном примере ячейка *title[17]* содержит символ *\0*.

Сравнивать строки языка C можно с помощью стандартной функции *strcmp*. С помощью этой функции можно проверять не только равенство, но и какая из строк предшествует другой. Строки упорядочиваются в порядке следования ASCII-кодов их символов. Таким образом, все приведенные ниже отношения являются истинными.

Примеры истинных выражений

```
"dig" < "dog"
"Star" < "star" (поскольку 'S' < 's')
"start" > "star"
"d" > "abc"
```

Функция *strcmp*(*строка*$_1$, *строка*$_2$) возвращает целое значение.

Сравнение строк производится с помощью функции strcmp

- < 0, если *строка*$_1$ меньше *строки*$_2$;
- = 0, если *строка*$_1$ равна *строке*$_2$;
- > 0, если *строка*$_1$ больше *строки*$_2$.

Конкатенировать две строки можно с помощью стандартной функции *strcat*(*строка*$_1$, *строка*$_2$), которая приписывает копию второго

Конкатенация строк производится функцией strcat

аргумента первому, образуя новую строку. Например, если в программе содержится объявление

```
StringType str = "Com";
string str1 = "Com";
```

то выражение

```
strcat(str, "puter");
```

вернет строку *"Computer"* и присвоит ее переменной *str*. При этом необходимо гарантировать, что максимальная длина *строки*$_1$ позволяет поместить в ней новую строку. В нашем примере описание типа *StringType* гарантирует, что максимальная длина строки *str* равна 30. Этого вполне достаточно.

Теперь, если *str2* представляет собой другую переменную типа *StringType*, то оператор

```
strcpy(str2, strcat(str, "s"));
```

присвоит строкам *str* и *str2* строку *"Computers"*. Обратите внимание, что отдельный символ заключен в двойные кавычки *"s"*, а не в одинарные (*'s'*).

Для сравнения частей строк используется функция strncmp

Рассмотрим функции, манипулирующие с частями строк. Функция

strncmp(*строка*$_1$, *строка*$_2$, *n*)

очень похожа на функцию *strncmp*, но сравнивает не более чем *n* первых символов.

К строке можно добавить один символ

Стандартная функция

strncat(*строка*$_1$, *строка*$_2$, *n*)

добавляет к *строке*$_1$ не более чем *n* первых символов *строки*$_2$, завершая результат нулевым символом *\0*. Поскольку функция *strncat* автоматически добавляет нулевой символ, с ее помощью можно добавить к строке один символ. Если переменная *ch* имеет тип *char*, то выражение

```
strncat(str, &ch, 1)
```

добавляет символ *ch* в конец строки *str* и возвращает новую строку.

Строковую переменную можно вывести на экран с помощью оператора <<, как и литеральную константу. Например, операторы

```
title = "Стены и зеркала";
cout << "\"" << title << "\"\n";
```

выводят на экран строку *"Стены и зеркала"*. Обратите внимание на специальный символ *\"*, стоящий внутри литеральной строки. Он предназначен для вывода на экран двойной кавычки. Оператор << выводит на экран всю строку, включая пробелы, пока не обнаружит нулевой символ.

Для вывода строки на экран можно использовать функцию write

Как часть, так и всю строку можно вывести на экран с помощью функции *write*.

```
cout.write(title, 5);
// Выводит строку "Стены"
cout.write(&title[5], 11);
// Выводит строку " и зеркала"
```

Второй оператор поместит вторую строку сразу вслед за первой, не выполняя перехода на новую строку.

В символьную переменную можно считать строку символов. Если оператор

```
cin >> title;
```

Для ввода строки, не содержащей пробельных символов, можно применять оператор >>

считает строку ввода

```
Гекльберри Финн
```

он присвоит переменной *title* строку *"Гекльберри"*. Пробельные символы во входной строке прерывают операцию чтения. Для ввода строки вместе с пробельными символами, нужно использовать функции *getline*, *get* или *read*.

- Функция *cin.getline(s, count)* считывает не более *count*-1 символов в строку *s*. Если входная строка содержит не более чем *count*-1 символов, за которыми следует символ перехода на новую строку, то строка *s* будет содержать эти символы, завершающиеся нулем *\0*. Символ перехода на новую строку в переменную *s* не копируется. Однако он будет вытолкнут из буфера и при дальнейшем считывании данных учитываться не будет.

  Если входная строка состоит из более чем из *count*-1 символов, за которыми следует символ перехода на новую строку, то строка *s* будет содержать *count*-1 символов, завершающихся нулем *\0*. Следующий оператор считывания будет учитывать все символы, оставшиеся во входной строке.
- Функция *cin.get(s, count)* работает аналогично функции *getline*, но не выталкивает из буфера символ перехода на новую строку.
- Функция *cin.read(s, count)* считывает *count* символов в строку *s*, но не приписывает к ним нулевой символ *\0*.

Остальные три функции оказываются полезными при обработке символов или строк.

- Функция *cin.peek()* возвращает следующий символ, находящийся во входном потоке, не извлекая его оттуда.
- Функция *cin.ingore(n)* пропускает *n* символов, находящихся во входном потоке.
- Функция *cin.ingore(n, ch)* либо пропускает *n* символов, находящихся во входном потоке, лио пропускает символы, пока не обнаружится символ *ch*, в зависимости от того, какое событие наступит раньше.
- Функция *cin.putback(ch)* помещает символ *ch* во входной поток, чтобы он был считан при следующем выполнении оператора ввода.

Структуры

Структура — это группа связанных друг с другом элементов, которые могут иметь разный тип

В то время как массив представляет собой совокупность элементов, имеющих одинаковый тип, **структура** (structure) в языке C++ является группой связанных друг с другом элементов, тип которых не обязательно совпадает. Каждый элемент структуры называется ее **членом** (member). Хотя, в принципе, членами структуры могут быть как данные, так и функции, обычно структуры содержат только **данные-члены** (data members).[10]

[10] Структуры могут содержать, и часто действительно содержат, специальные функции-члены, называемые *конструкторами*. Эти функции рассматриваются в главе 3 в контексте обсуждения классов языка C++.

Структура, описывающая некоего студента, должна содержать его имя, возраст и среднюю оценку. Рассмотрим операторы, описывающие такую структуру.

После определения структуры обязательно ставится точка с запятой

```
struct Person
{
   string name;
   int age;
   double gpa;
}; // Конец структуры
```

Данными-членами этой структуры являются переменные *name*, *age* и *gpa*. Обратите внимание, что после определения структуры обязательно ставится точка с запятой.

Тип *Person*, называемый **агрегированным типом** (aggregate type), можно использовать для объявления структуры *student*.

```
Person student;
```

Это объявление не инициализирует данные-члены, связанные с конкретной структурой *student*.

Данные-члены структуры можно инициализировать так же, как инициализируются массивы. Например, предыдущее объявление можно заменить следующим.

```
Person student = {"Джек Пятеркин", 21, 4.0};
```

Содержание структуры *student* после инициализации показано на рис. А.9.

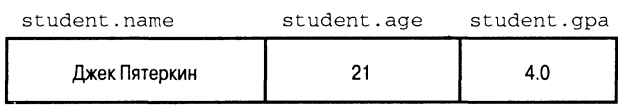


Рис. А.9. Структура student

Для того чтобы сослаться на членов конкретной структуры, необходимо **уточнить** (qualify) имя члена, указав перед ним имя структуры, которой он принадлежит, и **оператор точки** (dot operator), т.е. точку. Например, вторым членом структуры *student* является переменная

```
student.age
```

а третью букву первого члена структуры *student* можно определить с помощью оператора

```
student.name[2]
```

Одну структуру можно присвоить другой

Используя оператор присваивания, можно создавать копии целой структуры. Таким образом, приведенный ниже оператор копирует структуру *student* в заранее объявленную структуру *studentCopy*.

```
studentCopy = student;
```

Структуру можно также передавать в качестве аргумента функции. Если аргумент передается по значению, создается копия структуры. Кроме того, структура может возвращаться функцией с помощью оператора *return*.

Структуры внутри других структур

Структура может быть членом другой структуры

Иногда бывает нужно, чтобы структура была членом другой структуры. Допустим, что в предыдущем примере структура содержала адрес. Этот адрес удобно представить в виде другой структуры, членами которой являются номер дома, название улицы, города, страны, а также почтовый индекс. Эти изменения можно учесть следующим образом.

```
struct Addr
{
   int number;
   string street;
   string city;
   string state;
   string zip;
}; // Конец структуры

struct Person
{
   string name;
   int age;
   double gpa;
   Addr address;
}; // Конец структуры

Person student;
```

Обратите внимание на порядок определения структур. Определение структуры *Addr* должно предшествовать определению структуры *Person*. Теперь почтовый код, хранящийся в структуре *student*, можно определить с помощью оператора *student.address.zip*.

Массивы структур

Допустим, что преподаватель хочет иметь несколько структур, содержащих сведения о студентах, учащихся в какой-нибудь группе. Количество студентов в этой группе не может превышать число, задаваемое именованной константой *MAX_STUDENTS*. Добавим в программу операторы

```
typedef Person GroupType[MAX_STUDENTS];
GroupType csc212;
```

Теперь массив *csc212* содержит структуры. Рассмотрим, как получить доступ к ним. Например, переменная *csc212[9].name* задает имя 10-го студента в массиве, *csc212[9].name[0]* — первую букву фамилии 10-го студента, а *csc212[9].address.state* — страну, из которой прибыл 10-й студент.

Исключительные ситуации

Исключительная ситуация (exception) — это механизм, применяемый в языке C++ и других языках программирования для обработки ошибок. Если в ходе выполнения функции произошла ошибка, функция **генерирует** (throw) исключительную ситуацию. Затем она может приступить к обработке ошибки, **перехватив** (catching) исключительную ситуацию и выполнив соответствующий код. Основные сведения об исключительных ситуациях приведены в главе 3.

Используя некоторые методы из стандартной библиотеки языка C++, мы уже сталкивались с исключительными ситуациями. Например, если аргумент функции выходит за пределы допустимого диапазона, генерируется исключительная ситуация *out_of_range*.

Перехват исключительных ситуаций

Для обработки исключительных ситуаций в языке C++ предусмотрены **блоки try-catch**. В блок *try* помещаются операторы, которые могут вызвать исключительную ситуацию. Блок *try* должен сопровождаться одним или несколькими блоками *catch*. Каждый блок *catch* распознает тип исключительной ситуации, подлежащей обработке. С одним блоком *try* может быть связано несколько блоков *catch*, поскольку даже отдельный оператор может вызвать несколько исключительных ситуаций разного типа. Блок *try* также содержит несколько операторов, каждый из которых может генерировать исключительную ситуацию. Вот его общий вид.

```
try
{
   оператор(ы);
}
```

А вот как выглядит блок *catch*.

```
catch (Класс_Исключительной_Ситуации идентификатор)
{
   оператор(ы);
}
```

Если оператор в блоке *try* генерирует исключительную ситуацию, оставшаяся часть блока игнорируется, и управление передается блоку *catch*, предназначенному для обработки исключительных ситуаций этого типа. Затем выполняются операторы блока *catch*. После завершения этого блока выполнение программы возобновляется с точки, следующей за последним оператором блока *catch*. Если возникшая исключительная ситуация не может быть обработана ни одним боком *catch*, выполнение программы завершается аварийно.

Обратите внимание, что если исключительная ситуация возбуждается внутри блока *try*, вызываются деструкторы всех локальных объектов этого блока. Это позволяет гарантировать, что все ресурсы, задействованные в этом блоке, будут освобождены, даже если блок будет выполнен не полностью.

Компилятор выбирает подходящий блок *catch*, перебирая их один за другим в порядке, указанном в программе. Подходящим считается блок *catch*, аргумент которого совпадает с возникшей исключительной ситуацией. Таким образом, разделы *catch* должны быть упорядочены, так чтобы первыми оказались блоки, предназначенные для обработки более узких исключительных ситуаций, а разделы, ориентированные на более общие типы, должны размещаться за ними. Рассмотрим пример.

```
string str = "Sarah";
try
{
   str.substr(99, 1);
   // Здесь размещаются другие операторы
}  // Конец блока try
```

```
catch (exception e)
{
   cout << "Перехвачено что-то другое" << endl;
} // Конец блока catch

catch (out_of_range e)
{
   cout << "Перехвачена исключительная ситуация out_of_range"
        << endl;
} // Конец блока catch
```

При компиляции этого фрагмента программы на экране появляется следующее предупреждение.

```
TestExceptionExample.cpp(11) : warning C4286:
'class std::out_of_range' : is caught by base class
('class exception') on line 8
Linking...
```

Чтобы скомпилировать этот код без предупреждений, нужно поменять местами два раздела *catch*.

Программа, приведенная ниже, демонстрирует, что произойдет, если исключительная ситуация будет сгенерирована, но не обработана. Программа кодирует строку, выполняя простую подстановку. Каждая буква исходной строки заменяется буквой, расположенной в алфавите на три позиции ниже. Достигнув конца алфавита, мы переходим на его начало. Например, буква 'a' заменяется буквой 'd', буква 'b' — буквой 'e', а буква 'x' — буквой 'a'. Поток управления при возникновении исключительной ситуации в этой программе показан на рис. А.10.

```
#include <iostream>
#include <string>
using namespace std;

void encodeChar(int i, string& str)
{
   int base;
   if (islower(str[i]))
      base = int('a');
   else
      base = int('A');
   char newChar = (int(str[i]) - base + 3) % 26 + base;
   str.replace(i, 1, 1, newChar);
} // Конец функции encodeChar

void encodeString(int numChar, string& str)
{
   for (int i = numChar-1; i>=0; i--)
      encodeChar(i, str);
} //Конец функции encodeString

int main()
{
   string str1 = "Sarah";
   encodeString(99, str1);
   return 0;
} // Конец оператора main
```

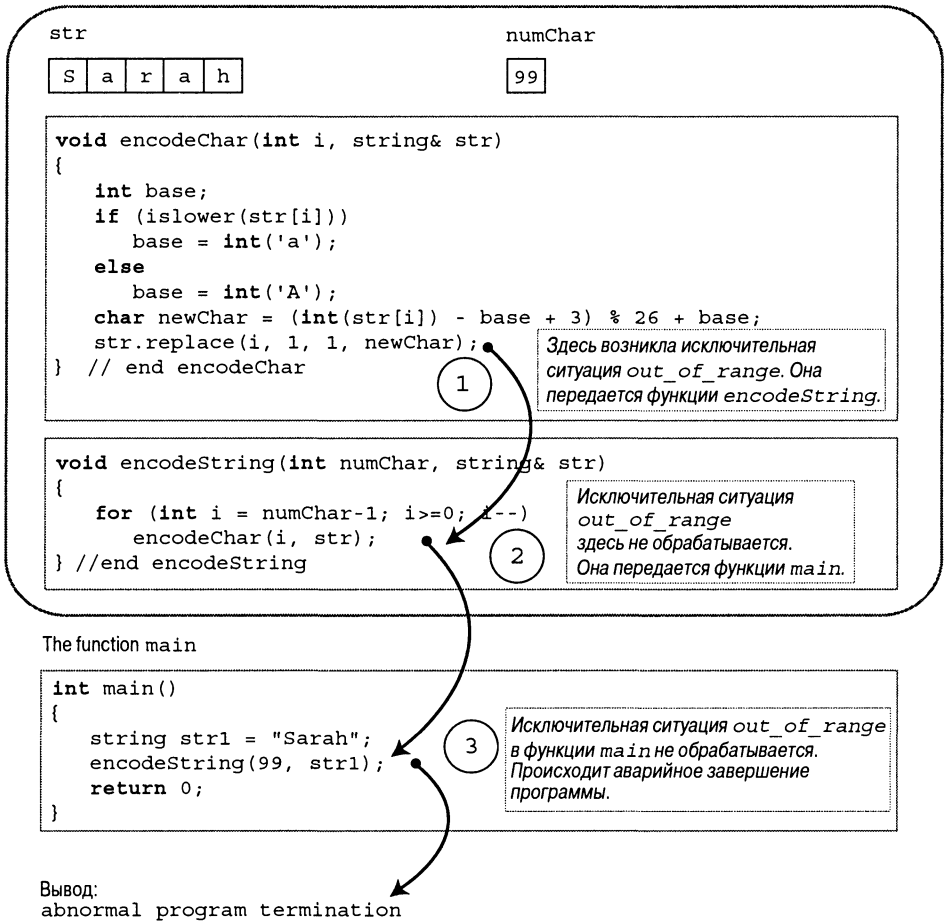


Рис. А.10. Поток управления при необработанной исключительной ситуации

На самом деле метод *encodeChar* порождает исключительную ситуацию *out_of_range*, которая генерируется, когда происходит попытка доступа к 99-му символу строки *str* при вызове *str.replace(99, 1, 1, newChar)*. Поскольку эта исключительная ситуация в функции *encodeChar* не обрабатывается, выполнение функции прерывается, а исключительная ситуация возвращается в функцию *encodeString*, а именно: в точку вызова функции *encodeChar*. Функция *encodeString* также не обрабатывает данную исключительную ситуацию, поэтому ее выполнение также прерывается, а исключительная ситуация передается в функцию *main*. Поскольку и там обработка исключительной ситуации не предусмотрена, происходит аварийное завершение программы.

В этом коде нет указаний, что функция *encodeChar* может генерировать исключительную ситуацию *out_of_range*. Однако в документации, сопровождающей функцию *encodeChar*, должны быть перечислены все исключительные ситуации, которые могут возникнуть. Следовательно, документируя функцию, как описано в главе 1, следует предусматривать все потенциальные ошибки.

Исключительная ситуация *out_of_range* может возникнуть в любой точке последовательности вызовов. Например, функцию *encodeChar* можно переписать так, что она станет перехватывать эту исключительную ситуацию.

```
void encodeChar(int i, string& str)
{
   int base;
   if (islower(str[i]))
      base = int('a');
   else
   base = int('A');

   try
   {
      char newChar = (int(str[i]) - base + 3) % 26 + base;
      str.replace(i, 1, 1, newChar);
   } // Конец блока try

   catch (out_of_range e)
   {
      cout << "На позиции " << i << "нет символов" << endl;
   } // Конец блока catch
} // Конец функции encodeChar
```

При выполнении новой версии функции *encodeChar* на экране появятся следующие строки.

```
На позиции 99 нет символа
На позиции 98 нет символа
На позиции 97 нет символа
...
```

Функция *encodeString* вызывает функцию *encodeChar* 99 раз. Следовательно, исключительная ситуация будет генерироваться *98-str.length()* раз. Если исключительная ситуация не обрабатывается, программа завершается аварийно при первой же ошибке. Обработка исключительной ситуации позволяет продолжить выполнение программы.

Хотя исключительная ситуация *out_of_range* генерируется внутри функции *encodeChar*, это не самое лучшее место для ее обработки. Например, если клиент сделает вызов *encodeString(10000, str)*, сообщение об ошибке появится на экране *9999-str.length()* раз! В этом случае имеет смысл поместить блоки *try-catch* в функцию *encodeString*, а не в функцию *encodeChar*. Таким образом, функция *encodeChar* больше не будет обрабатывать исключительную ситуацию, а передаст ее функции *encodeString*. Ниже приводится код обработки исключительной ситуации в функции *encodeString*.

```
void encodeString(int numChar, string& str)
{
   try
   {
      for (int i = numChar-1; i >= 0; i--)
         encodeChar(i, str);
   } // Конец блока try
   catch (out_of_range e)
   {
   cout << "Строка не содержит" << numChar;
   cout << " символов." << endl;
```

```
    cout << e.what();
    } // Конец блока catch
} // Конец функции encodeString
```

В результате выполнения этой функции на экран будут выведены следующие строки.

```
Строка не содержит 10 символов.
invalid string position
```

Теперь исключительная ситуация *out-of-range*, порожденная функцией *encodeChar*, будет передана назад *encodeString*. Функция *encodeString* прекращает выполнение операторов блока *try*, выполняет операторы блока *catch* и возобновляет выполнение программы с оператора, стоящего после блока *catch*. Сообщение об ошибке будет выведено лишь однажды, поскольку цикл *for* находится внутри блока *try*, который не выполняется при возникновении исключительной ситуации. Если бы блок *try* был размещен внутри цикла *for* (среди вызовов функции *encodeChar*), то исключительная ситуация обрабатывалась бы на каждой итерации цикла, что привело бы к многократному выводу на экран сообщения об ошибке.

Блок *catch* содержит также функцию *e.what()*. Напомним, что в каждом блоке *catch* указан тип исключительной ситуации, для обработки которой он предназначен, а также идентификатор. Этот идентификатор задает имя перехваченной исключительной ситуации, используемое внутри блока. В данном случае вызывается функция *what* для исключительной ситуации *e*. Функция *what* возвращает указатель на строку языка C (завершающуюся нулевым символом `\0`), которая описывает возникшую исключительную ситуацию. Эта функция доступна для любых исключительных ситуаций, предусмотренных в стандартной библиотеке.

Заголовочный файл *<stdexcept>* определяет стандартные исключительные ситуации, которые могут породить функции, определенные в стандартной библиотеке языка C++. Этот заголовочный файл содержит два вида исключительных ситуаций: **времени выполнения программ** (runtine exception) и **логические** (logical exception). Перечислим исключительные ситуации времени выполнения программ.

`overflow_error`	Арифметическое переполнение
`range_error`	Выход за пределы допустимого диапазона
`underflow_error`	Потеря значимости

К логическим исключительным ситуациям относятся следующие.

`domain_error`	Ошибка из-за выхода за пределы допустимых значений
`invalid_argument`	Недопустимый аргумент функции
`length_error`	Попытка создать слишком большой объект
`out_of_range`	Значение аргумента функции выходит за рамки допустимого диапазона

Генерирование исключительных ситуаций

Функции могут сообщать об ошибках, генерируя исключительные ситуации с помощью оператора, имеющего следующий вид.

```
throw класс_исключительной_ситуации(строка_аргумент)
```

Здесь *класс_исключительной_ситуации* означает тип генерируемой исключительной ситуации, а *строка_аргумент* является аргументом конструктора соответствующего класса. После выполнения оператора *throw* остальной код функции игнорируется.

Чтобы ограничить спектр исключительных ситуаций, генерируемых функцией, раздел *throw* следует включить в заголовок функции. Этот раздел состоит из ключевого слова *throw*, за которым следует список исключительных ситуаций, разделенных запятыми и заключенных в круглые скобки. Например, приведенные объявления функций означают, что она может генерировать исключительные ситуации *BadDataException* и *MyException*.

Функция, код которой может генерировать исключительные ситуации

```
void myFunction(int x)
   throw(BadDataException, MyException)
{
   if (x == MAX)
      throw BadDataException("BadDataException: причина");
      // Здесь находится некий код
      ...
   throw MyException("MyException: причина");
} // Конец функции myFunction
```

Включение раздела *throw* в заголовок функции гарантирует, что данная функция может генерировать только указанные исключительные ситуации. Попытка сгенерировать любую другую ситуацию приведет к возникновению ошибки при выполнении программы (runtime error). Если в заголовке не указан раздел *throw*, функция может генерировать любую исключительную ситуацию.

В стандартной библиотеке языка C++ можно найти классы исключительной ситуации, подходящие практически для любой программы. Например, при разработке абстрактного списка, описанного в главе 3, исключительная ситуация генерируется при выходе индекса за пределы допустимого диапазона. Для такого случая в стандартной библиотеке языка C++ имеется исключительная ситуация *out_of_range*. Для того чтобы использовать в программе стандартные классы исключительных ситуаций, необходимо указать имя пространства имен *std*.

Класс исключительных ситуаций *exception*, либо любой другой производный от него класс, является базовым для любых других классов. Это обеспечивает единый интерфейс при работе с любыми исключительными ситуациями. В частности, все классы исключительных ситуаций, предусмотренные в стандартной библиотеке языка C++, содержат функцию-член *what*, возвращающую сообщение об ошибке. Эта функция оказывается весьма полезной внутри блока *catch*.

Можно определять свои собственные исключительные ситуации

Кроме того, можно определять свои собственные классы исключительных ситуаций, производные от стандартного класса *exception*. Для этого также необходимо использовать пространство имен *std*. Обычно класс исключительных ситуаций состоит из конструктора, получающего строковый параметр. Например, можно определить свой собственный класс *MyException*.

```
#include <exception>
#include <string>
using namespace std;
class MyException : public exception
{
public:
   MyException(const string & Message = "")
               : exception(Message.c_str())
   { }
}; // Конец класса
```

Конструктор определяет способ, которым оператор *throw* идентифицирует условия, генерирующие исключительную ситуацию. Например, оператор

```
throw MyException("MyException: причина");
```

вызывает конструктор класса *MeException*. Сообщение, переданное конструктору, возвращается функцией *what*, унаследованной от класса *exception*. Таким образом, раздел *catch* может получить доступ к этому сообщению, как показано в следующем примере.

```
try
{
   if (size > max)
} // Конец блока try
catch (MyException e)
{
   cout << e.what();
} // Конец блока catch
```

Работа с файлами

Мы использовали файлы уже в самой первой программе. Фактически текст любой программы, написанной на языке C++, хранится в файле, созданном текстовым редактором. К таким файлам можно обращаться в любое время. Файлы содержат информацию, которая либо считывается программой, либо записывается ею. Именно этот аспект файлов нас будет интересовать больше всего.

Файл — это последовательность компонентов, имеющих одинаковый тип

Файл представляет собой *последовательность* компонентов, имеющих одинаковый тип. Они хранятся на вспомогательных запоминающих устройствах, например дисках. Файлы могут быть большими. Они могут существовать и после выполнения программы. В противоположность им, переменные основных типов, например, представляют собой ячейки памяти, доступные лишь в программе, которая их заполнила. После завершения работы программы оперативная система очищает память, уничтожая записанную там информацию.

Поскольку файлы могут существовать и после завершения работы программы, они представляют собой не только постоянное хранилище информации, но и способ связи между разными программами. Программа *A* может записать результаты своей работы в файл, который позднее прочтет программа *B*. Однако файлы, уничтожаемые после выполнения программы, также не редкость. Например, при выполнении программы могут создаваться временные файлы, в которых хранится информация, не помещающаяся в оперативной памяти.

Полезно сравнить файлы с их ближайшими родственниками в языке C++ — массивами. Файлы и массивы похожи, поскольку они представляют собой совокупности компонентов, имеющих одинаковый тип. Например, можно создать массив, имеющий тип *char*, и файл такого же типа. В обоих случаях компонентами являются символы. Однако между файлами и другими типами данных есть отличия: например, файлы могут существовать после завершения работы программы, а массивы нет. Укажем еще несколько различий между файлами и массивами.

- **Файлы могут увеличиваться в ходе выполнения программы, а массивы имеют фиксированный размер.** Объявив массив, программист указывает его максимальный размер. Следовательно, оперативная система выделит этому массиву соответствующую область памяти. Хорошо написанная программа, перед тем как вставить в массив новый элемент, всегда проверяет, достаточно ли там места. Если места не достаточно, программа должна завершить работу и вывести на экран сообщение об ошибке. Программист может увеличить размер массива, например, изменив значение именован-

ной константы, а затем скомпилировать и выполнить программу вновь.[11] Если объявить максимальную длину массива больше, чем нужно, часть памяти останется неиспользованной. С файлом этого случиться не может. Когда система создает файл в первый раз, он требует очень мало памяти. По мере того как программа добавляет в него новые записи, размеры файла увеличиваются, пока ресурсы внешнего запоминающего устройства не исчерпаются полностью. Таким образом, в любой фиксированный момент времени файл занимает именно столько памяти, сколько ему нужно.

- **Файлы обеспечивают как последовательный, так и прямой доступ; массивы обеспечивают только прямой доступ.** Если нужно обратиться к 100-му элементу массива *array*, можно просто записать выражение `array[99]`, не просматривая элементы от `array[0]` до `array[98]`. Разумеется, в массиве можно имитировать и последовательный доступ, но при этом каждый следующий элемент все равно будет доступен независимо от предыдущего.

  В то же время доступ к записи в файле может быть как последовательным, так и прямым. Если нам нужен 100-й элемент файла, его можно считать, указав позицию, не считывая первые 99 элементов. Однако при последовательном доступе прежде, чем обратиться к 100-му элементу, придется перебрать предыдущие 99.

Файлы классифицируются следующим образом. **Текстовый файл** (text file) состоит из строк символов. В частности, все файлы, которые создаются текстовыми редакторами, в том числе исходные коды программы на языке C++, являются текстовыми. Поскольку текстовые файлы состоят из символов, а обращаться к ним по номеру позиции не всегда удобно, обычно они предоставляют последовательный доступ. Файлы, которые не являются текстовыми, называются **бинарными** (binary), а иногда — **файлами общего вида** (general file), или просто **нетекстовыми файлами** (nontext file).

Текстовые файлы

Текстовые файлы предназначены для людей. Они представляют собой довольно гибкий и полезный инструмент, но не настолько эффективны с точки зрения затрат компьютерного времени и объема занимаемой памяти, как бинарные файлы.

Текстовый файл состоит из символьных строк

На первый взгляд, *кажется*, что текстовые файлы состоят из строк. Эта иллюзия часто порождает недоразумения. На самом деле, текстовые файлы, как и любые другие файлы, являются последовательностями компонентов, имеющих одинаковый тип. Иначе говоря, текстовый файл представляет собой последовательность символов. Причиной этого эффекта является **признак конца строки** (end-of-line symbol).

При создании текстового файла для перехода на следующую строку пользователи компьютеров нажимают клавишу <Enter>, вставляя в конце строки символ ее окончания. Когда устройство вывода, например, принтер или монитор, обнаруживает в текстовом файле признак конца строки, он делает переход на новую строку. В языке C++ признаком конца строки является символ *\n*.

Файлы заканчиваются специальным символом конца файла

Кроме того, существует специальный **признак конца файла** (end-of-file symbol), который находится сразу за последним компонентом

[11] В главе 4 описаны динамические массивы. Если во время выполнения программы такой массив оказывается полностью заполненным, его размер можно увеличить автоматически. Однако для этого придется копировать старый массив в новый.

файла. Этот символ может фактически отсутствовать, но в языке C++ предполагается, что он существует. Роль этого символа играет предопределенная константа *EOF*. В нашей книге мы предполагаем, что все текстовые файлы, включая пустой, содержат специальные символы конца строки и конца файла (рис. А.11).

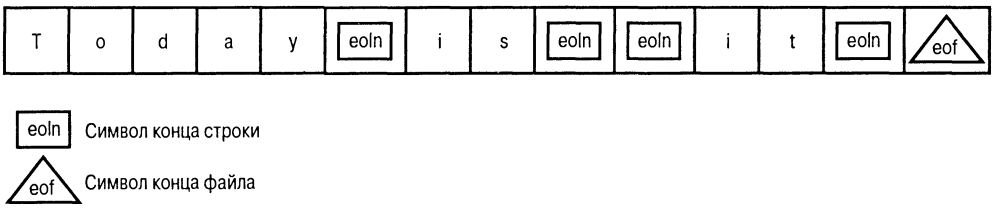


Рис. А.11. Текстовый файл с символами конца строки и конца файла

Любая программа, использующая файлы, должна иметь доступ к стандартной библиотеке потоков языка C++. Для этого необходимо включить в программу следующую директиву и оператор.

```
#include <fstream>
using namespace std;
```

Использование потоковой переменной для доступа к файлу

В этой библиотеке предусмотрены три типа потоков: *ifstream* для входных файлов, *ostream* — для выходных файлов и *fstream* — для файлов ввода/вывода.

Перед тем как использовать файл, его нужно инициализировать или открыть

Как открыть файл. Прежде чем прочитать данные из файла или записать их туда, нужно **открыть** его. Иначе говоря, файл нужно инициализировать, связав его имя с переменной потока. Открыть файл можно, указав его имя при объявлении переменной потока. Например, оператор

```
ifstream inFile("Ages.DAT"); // Входной файл
```

объявляет переменную входного потока *inFile*, связывая ее с файлом по имени *Ages.DAT*. Имя файла может быть литеральной константой либо строковой переменной.

Кроме того, переменную входного потока можно объявить следующим образом:

```
ifstream inFile;
```

а позднее применить функцию *open* для указания имени файла:

```
inFile.open("Ages.DAT");
```

Можно проверить, успешно ли открылся файл

Независимо от того, каким образом был открыт файл, переменную потока можно проверить с помощью следующего фрагмента кода.

```
if (inFile)
    ProcessError(); // Открытие было безуспешным
```

В окне файла указывается текущее положение файлового курсора

С каждым файлом программы связан отдельный **курсор** (file window), который отмечает текущую позицию в файле. При открытии файла курсор указывает на первый компонент файла, как показано на рис. А.12.

Поскольку компонентами файла являются символы, курсор перемещается с символа на символ.

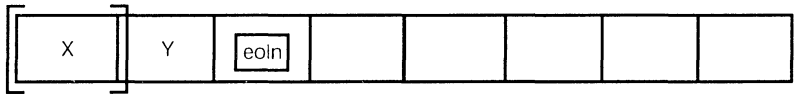


Рис. А.12. При открытии существующего текстового файла курсор указывает на первую позицию

Ввод символов. Допустим, что в программе объявлена потоковая переменная *inFile*, с которой связано имя текстового файла, заданное строковой переменной *fileName*.

```
ifstream inFile(fileName);
```

При открытии текстового файла курсор ссылается на первый компонент

Если этот файл используется для ввода, курсор указывает на компонент, который будет считан следующим. Следовательно, после открытия этого файла все готово для ввода первого компонента, как показано на рис. А.12. По мере считывания символов из текстового файла курсор перемещается от одного символа к другому. Ситуация, возникшая после ввода нескольких символов, проиллюстрирована на рис. А.13.

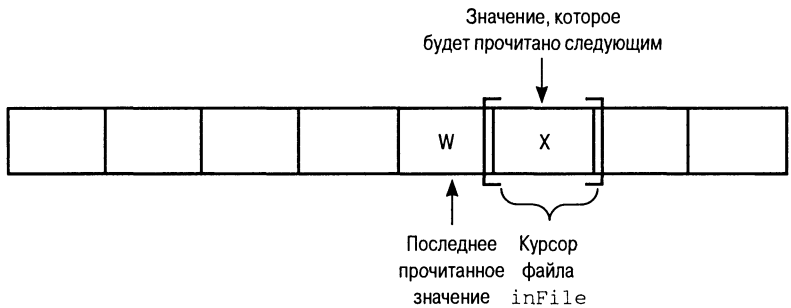


Рис. А.13. Курсор файла указывает на компонент, который будет считан следующим

Оператор ввода >> и функции ввода, использующие входной поток *cin* и описанные ранее, также могут применяться для ввода данных из текстового файла. Допустим, в программе есть символьная переменная *ch* и переменная потока *inFile*. Тогда операторы

```
inFile >> ch;
```

и

```
inFile.get();
```

означают следующее.

```
ch = значение, на которое указывает курсор файла
Переместить курсор файла на следующий компонент
```

Иллюстрация работы этих операторов приведена на рис. А.14.

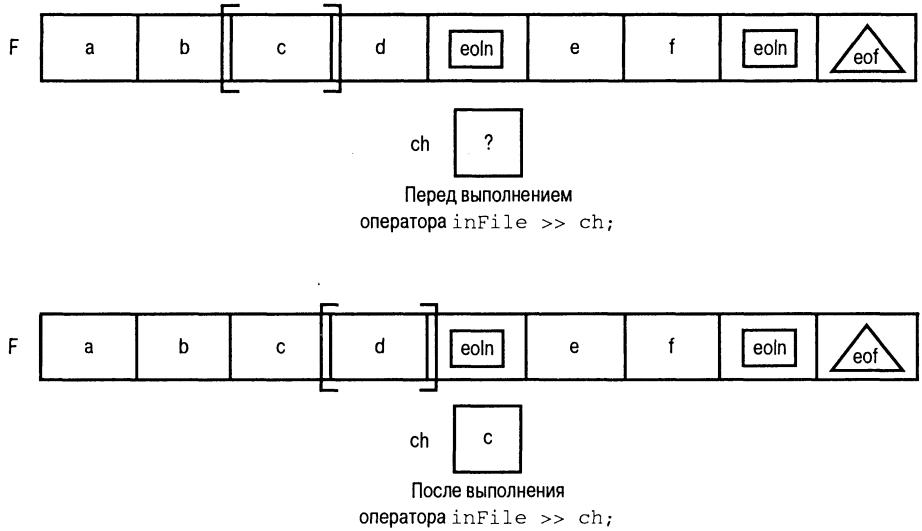


Рис. А.14. Эффект применения оператора inFile>>ch к текстовому файлу inFile

При желании каждое из этих действий можно выполнить по отдельности. Текущее значение, считанное из файла, можно присвоить переменной *ch*, не перемещая курсора. Для этого нужно написать следующий фрагмент программы.

```
ch = inFile.peek();
```

Текущее значение, на которое указывает курсор, можно проигнорировать и просто переместить курсор на *n* символов вперед с помощью оператора

```
inFile.ignore(n);
```

Курсор файла можно переместить непосредственно на позицию, следующую за очередным символом *ch*, либо просто на *n* символов, в зависимости от того, какая из ситуаций возникнет первой. Это действие осуществляется оператором

```
inFile.ignore(n, ch);
```

Функция *peek* позволяет определить, достигнут ли конец строки или файла. Например, цикл, приведенный ниже, выводит на экран строку из текстового файла.

```
while (inFile.peek() != '\n') // Цикл до конца строки
cout.put(inFile.get());
```

Кроме того, следующие операторы позволяют вывести на экран весь текстовый файл.

```
while (inFile.peek() != EOF) // Цикл до конца файла
cout.put(inFile.get());
```

В заключение рассмотрим текстовый файл *inFile*, изображенный на рис. А.14. Если переменная *ch* является символьной, то операторы, образующие приведенный ниже фрагмент программы, поочередно присвоят ей символы, считанные из текстового файла.

```
ifstream.inFile(fileName);
inFile >> ch;                  // ch = 'a'
inFile.get(ch);                // ch = 'b'
```

```
ch = inFile.get();          // ch = 'c'
inFile.get(ch);             // ch = 'e'
ch = inFile.peek();         // ch = 'f'
inFile.get(ch);             // ch = 'f'
inFile.get(ch);             // ch = '\n'
inFile.ignore(1);           // пропустить символ конца файла
inFile.get(ch);             // ошибка: попытка чтения
                            // за пределами файла
```

Кроме того, символы из текстового файла можно считывать целыми строками, используя функции *getLine*, *get* и *read*.

Вывод символов. Допустим, что в программе объявлена потоковая переменная *outFile*, с которой связано имя текстового файла, заданное строковой переменной *fileName*.

```
ifstream inFile(fileName);
```

При создании файл пуст, а курсор устанавливается в его начало (или конец). Если файл уже существует, его содержимое при открытии будет стерто[12], а курсор — установлен в его начало (или конец).

Операторы вывода << и функции, использующие поток *cin*, также могут работать с файлами. Допустим, в программе объявлены символьная переменная *ch* и переменная потока *outFile*. Тогда операторы

```
outFile << ch;
```

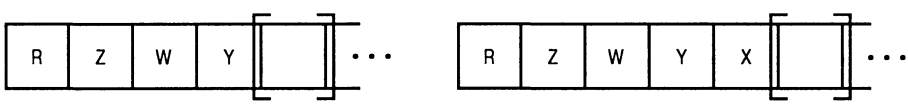
и

```
outFile.put();
```

означают следующее.

```
Записать значение переменной ch в позицию,
   на которую установлен курсор файла
Переместить курсор файла на следующую позицию
```

Иллюстрация работы этих операторов приведена на рис. А.15. Обратите внимание, что если бы переменная *X* содержала символ *\n*, то предыдущие операторы записали бы в файл признак конца строки.



Перед выполнением оператора `outFile << ch;` После выполнения оператора `inFile << ch;`

Рис. А.15. Эффект применения оператора outFile<<ch к текстовому файлу outFile, когда переменная ch содержит символ X

Кроме того, с помощью оператора << в текстовый файл можно выводить целые строки (см. раздел "Строки").

Как закрыть файл. В каждый момент времени открытым может быть только один файл ввода и один файл вывода. Например, если файл ввода уже открыт и нужно прочитать данные из другого файла, текущий файл необходимо **закрыть**. Конкретный файл

> В каждый момент времени открытым может быть только один файл ввода и один файл вывода

[12] Фактически данные могут не стираться, однако файл будет считаться пустым.

можно закрыть, т.е. разорвать связь между потоковой переменной и его именем, с помощью функции

```
myFile.close();
```

Закрытый файл становится недоступным для ввода или вывода, пока он снова не будет открыт.

Числовые данные в текстовых файлах. Как известно, из стандартного потока ввода можно вводить целые числа и числа с плавающей точкой, присваивая их переменным, имеющим один из арифметических типов. Кроме того, из входного потока можно вводить последовательности символов. Текстовый файл тоже представляет собой последовательность символов, поэтому не удивительно, что целые числа и числа с плавающей точкой также можно считывать из него и записывать в него. В данном случае целые числа рассматриваются лишь для примера, поскольку все остальные арифметические типы обрабатываются аналогично.

При считывании из текстового файла значения переменной типа `int` система ожидает ввода последовательности символов, которые потом можно преобразовать в целое число. Например, если текстовый файл содержит символы *2*, *3* и *4*, то при считывании этих символов в целочисленную переменную *x* система преобразует их во внутреннее представление целого числа *234* и присвоит его переменной *x*. Точнее говоря, текстовый файл содержит ASCII-коды символов *2*, *3* и *4*, т.е. десятичные числа 50, 51 и 52. Однако эти коды представлены в двоичном виде на рис. А.16, *а*. При считывании этих кодов в целочисленную переменную *x*, она содержит внутреннее двоичное представление целого числа *234*, которые представлены в двоичном виде на рис. рис. А.16, *б*. Следовательно, представление цифр в текстовом файле отличается от представления соответствующих целых чисел в памяти компьютера.

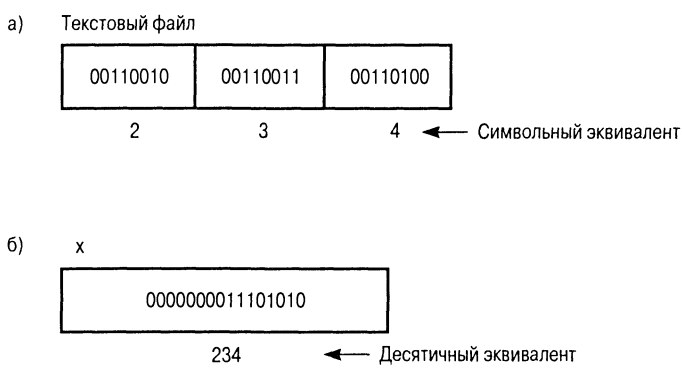


Рис. А.16. Двоичная система кодирования: а) двоичное представление ASCII-символов 2, 3 и 4, содержащихся в текстовом файле; б) внутреннее двоичное представление целого числа 234

В заключение рассмотрим случай, когда текстовый файл содержит целые числа и связан с потоковой переменной `inFile`, а переменная *x* имеет тип `int`. Тогда оператор

```
inFile >> x;
```

означает следующее.

```
Найти первый непробельный символ
Преобразовать в целое число последовательность символов,
   начинающуюся с текущей позиции курсора файла inFile
   и заканчивающуюся непосредственно перед следующим
   символом c, не являющимся цифрой
Присвоить полученное целое число переменной x
Переместить курсор на символ c
```

Последовательность этих шагов продемонстрирована на рис. А.17. Учтите, что если эта последовательность начинается с символа, отличающегося от знаков +, -, а также цифр 0, ..., 9, считывание будет прекращено. Например, система не может преобразовать последовательность символов *w123* в целое число. Однако символы *123* из последовательности *123wrt* будут успешно считаны.

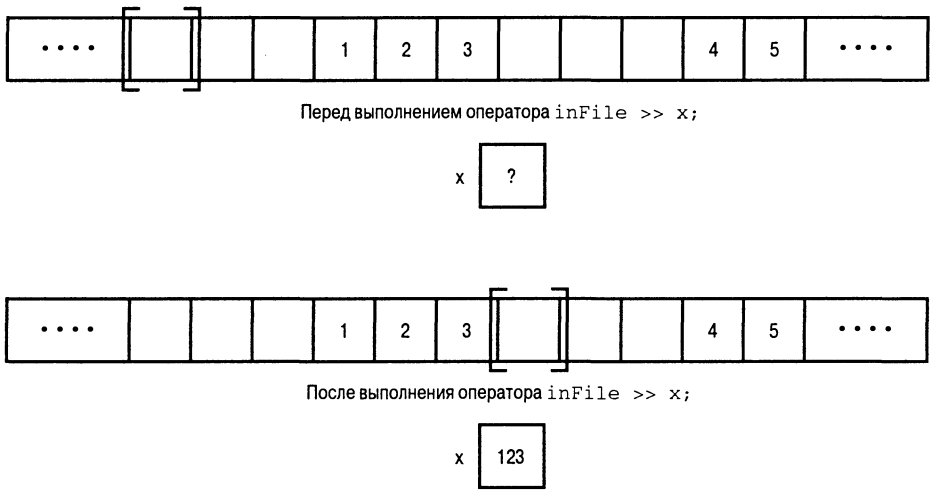


Рис. А.17. Считывание целого числа из текстового файла

Когда программа записывает в текстовый файл целое число, например, число *234*, система сначала преобразует его внутреннее двоичное представление (0000000011101010) в последовательность символов *2*, *3*, *4*, а затем записывает эти символы в файл. Если переменная *outFile* является переменной потока вывода, а переменная *x* имеет целочисленный тип, то оператор

```
outFile << x;
```

означает следующее.

```
Преобразовать значение переменной x в последовательность
   символов
Добавить эту последовательность в файл
Установить курсор файла на позицию, следующую за последним
   записанным символом
```

Приведенная ниже функция считывает и выводит на экран текстовый файл, содержащий целые числа.

```
void echoFile(string fileName)
{
   ifstream inFile(fileName);
   int x;
```

```
   while (inFile >> x) // Считывание до конца файла
   cout << x << " ";
   cout << endl;

   inFile.close();
} // Конец функции echoFile
```

Эта функция игнорирует символы конца строки и выводит все целые числа в одну строку.

Допустим, нам нужно вывести на экран все строки, содержащиеся в файле. Если каждая строка содержит одинаковое количество целых чисел, то решить эту задачу достаточно просто. Например, если каждая строка файла состоит из трех целых чисел, а x, y и z — целочисленные переменные, цикл while из предыдущего примера можно заменить следующим оператором.

```
while (inFile >> x >> y >> z)
   cout << x << " " <<  y  << " " < x << endl;
```

Другое решение, не зависящее от количества целых чисел в строке, заключается в использовании функций *peek* и *ignore*. Следовательно, предыдущую функцию можно модифицировать следующим образом. Обратите внимание, что файловая переменная всегда передается в функцию по ссылке.

```
void skipBlanks(ifstream& inFile)
// Пропускает пробелы в текстовом файле.
{
   while (inFile.peek() == ' ')
      inFile.ignore(1);
} // Конец функции skipBlanks

void echoLine(ifstream& inFile)
// Выводит одну строку текстового файла.
{
   int x;
   while (inFile.peek() != '\n')
   {
      inFile >> x;
      skipBlanks(inFile);
      cout << x << " ";
   } // Конец оператора while

   cout << "\n";
   inFile.ignore(1); // Сразу после символа \n
} // Конец функции echoLine

void echoFile(string fileName)
// Выводит содержимое текстового файла.
{
   ifstream inFile(fileName);

   skipBlanks(inFile);
   while (inFile.peek() != EOF)
      echoLine(inFile);

   inFile.close();
} // Конец функции echoFile
```

Копирование текстового файла. Допустим, что нам нужно скопировать текстовый файл, связанный с потоковой переменной *originalFile*. Для этого нужно выполнить довольно большой объем работы. Этот пример хорошо иллюстрирует работу описанных выше операторов. Копирование текстового файла осуществляется посимвольно с учетом символов конца строки и файла.

```
void copyTextFile(string originalFileName,
                  string copyFileName)
// ------------------------------------------------------
// Создает копию текстового файла.
// Предусловие: переменная originalFileName хранит имя
// существующего внешнего текстового файла, а переменная
// copyFileName — имя создаваемого текстового файла.
// Постусловие: текстовый файл, имя которого задается
// переменной copyFileName, копируется в файл с именем,
// определенным переменной originalFileName.
// ------------------------------------------------------
{
   ifstream originalFile(originalFileName); // Файл ввода
   ofstream copyFile(copyFileName); // Файл вывода
   char ch;

   // Копируем символы из одного файла в другой
   while (originalFile.get(ch))
      copyFile << ch;

   originalFile.close(); // Закрываем файлы
   copyFile.close();
} // Конец функции copyTextFile
```

Обратите внимание, что эта функция копирует признак конца строки как обычный символ. Для этого необходимо выражение

```
originalFile.get(ch)
```

поскольку оператор

```
originalFile >> ch
```

пропускает пробельные символы, включая признак конца строки.

Добавление к текстовому файлу. Открывая файл, кроме его имени, можно указать второй аргумент, имеющий вид

```
ios::режим
```

Здесь опция *режим* может принимать значения *in*, *out* или *app*. До сих пор мы этот параметр не указывали, поскольку файлы потока *ifstream* по умолчанию открываются для ввода, а файлы потока *ofstream* — для вывода. Файл вывода *outFile* можно сразу открыть в режиме добавления (опция *app*). Операторы

```
ofstream outFile("Sample.DAT", ios::app);
```

и

```
ofstream outFile;
outFile.open("Sample.DAT", ios::app);
```

подготавливают файл для вывода и устанавливают курсор вывода сразу за его последним компонентом. Следовательно, старое содержимое файла сохраняется без изменений, и в него будут добавлены новые компоненты.

В упражнении 13 предлагается написать функцию, добавляющую данные в текстовый файл.

Последовательный поиск в текстовом файле. Допустим, что в текстовом файле записана информация о сотрудниках компании. Для простоты будем предполагать, что этот файл содержит две строки информации о каждом сотруднике. В первой строке записывается фамилия сотрудника, а во второй — данные о нем, например величина его зарплаты.

Имея фамилию сотрудника, можно отыскать в файле соответствующие строки и вывести на экран информацию о нем. Алгоритм **последовательного поиска** (sequential search) перебирает фамилии сотрудников, пока не обнаружит искомую.

```
struct Person
{
   string name;
   double salary;
}; // Конец структуры struct

void searchFileSequentially(string fileName,
                            string desiredName,
                            person& desiredPerson,
                            bool& found)
// ---------------------------------------------------------
// Осуществляет поиск записи в текстовом файле.
// Предусловие: аргумент fileName задает имя текстового файла,
// содержащего фамилии и данные о сотрудниках. Запись о каждом
// сотруднике в файле состоит из двух строк: в первой строке
// содержится фамилия сотрудника, а во второй — величина его
// зарплаты. Переменная desiredName задает фамилию искомого
// сотрудника.
// Постусловие: если в файле обнаружена запись desiredName,
// то фамилия и величина зарплаты сотрудника содержатся
// в структуре desiredPerson, а пермеенаня found имеет
// значение true. В противном случае переменная found имеет
// значение false, а запись в файле отсутствует. Файл остается
// неизменным и закрывается.
// ---------------------------------------------------------
{
   ifstream inFile(fileName);
   string nextName;
   double nextSalary;

   found = false;
   while ( !found && getline(inFile, nextName) )
   {
      inFile >> nextSalary >> ws; // skip trailing eoln
      if (nextName == desiredName)
      found = true;
   } // Конец оператора while

   if (found)
   {
      desiredPerson.name = nextName;
      desiredPerson.salary = nextSalary;
   } // Конец оператора if

   inFile.close();
} // Конец функции searchFileSequentially
```

Прежде чем найти искомую запись, эта функция должна перебрать все предыдущие строки. Если фамилии перечислены в алфавитном порядке, можно определить место, где должна находиться искомая запись. Таким образом, если искомая запись отсутствует, поиск можно прекратить задолго до обнаружения конца файла (см. упражнение 10).

Прямой доступ к текстовому файлу. Несмотря на то что обычно текстовый файл обрабатывается последовательно, к символу, записанному в указанной позиции, можно обратиться напрямую, не считывая предшествующие символы.

Символы пронумерованы последовательно, в порядке их следования в файле, причем первый символ имеет номер 0. Функция *seekg* обеспечивает доступ к любому символу, хранящемуся в файле, по его позиции. Например, вызов

```
myFile.seekg(15)
```

перемещает курсор файла на символ под номером 15, который на самом деле является 16-м символом в файле.

Можно обнаружить символ, записанный в начале файла, в текущей позиции курсора или в конце файла, указав второй аргумент функции *seekg*:

```
ios::режим
```

где опция *режим* может принимать значения *beg*, *cur* или *end*. Следовательно, вызов функции

```
myFile.seekg(2, ios::cur)
```

найдет второй символ, следом за текущим положением файлового курсора.

Бинарные файлы

Файлы, не являющиеся текстовыми, называются бинарными (или файлами общего вида). Как и текстовый файл, бинарный файл представляет собой последовательность компонентов одинакового типа, причем эти компоненты не могут быть файлами. Следует подчеркнуть, что каждый **файловый компонент** (file component) является невидимым. Например, каждый компонент бинарного файла, состоящего из целых чисел, является двоичным представлением целого числа. Если записать в бинарный файл целое число 234, то система создаст именно его двоичное представление 0000000011101010, а не двоичное представление символов 2, 3 и 4, т.е. 00110010, 00110011 и 00110100, соответственно. Это относится и к бинарным файлам, содержащим числа с плавающей запятой. Как правило, бинарные файлы создаются программой и предназначены для служебных целей.

Оператор

```
ofstream outFile(myFileName, ios::binary)
```

связывает переменную файла *outFile* с внешним бинарным файлом, имя которого задано строкой *myFileName*. В конце бинарного файла, так же как и в текстовом файле, записан признак конца. Однако в бинарном файле нет разделения на строки, хотя в нем некоторые символы могут случайно совпасть с символами конца строки. За исключением этих особенностей, эти файлы обрабатываются точно так же, как и текстовые.

Библиотеки

Одно из преимуществ модульного программирования заключается в том, что функции можно реализовывать независимо друг от друга. Кроме того, одну и ту же функцию могут использовать разные программы. Следовательно, можно создать **библиотеки** (library) функций, которые можно включать в свои программы.

Современный стандарт языка C++ поддерживает библиотеки старого стиля, совместимые с языком C, и нового стиля, предназначенные только для языка C++. Любая библиотека, независимо от стиля, имеет соответствующий **заголовок** (header) , в котором записана информация о ее содержании. В библиотеках старого стиля, а также в библиотеках, определенных пользователем, заголовок обычно является файлом, имеющим расширения *.h*. В библиотеках нового стиля заголовок представляет собой абстракцию, которую компилятор может отобразить в имя файла или обработать как-то иначе. Таким образом, в библиотеках нового стиля заголовок не имеет расширения *.h*.

Мы уже использовали некоторые библиотеки, в частности библиотеку ввода-вывода. Чтобы вызвать функцию, определенную в библиотеке, необходимо в программу записать директиву *include*, указав заголовок соответствующей библиотеки. Например, чтобы использовать старую библиотеку *iostream*, нужно выполнить следующую директиву.

```
#include <iostream.h> // Библиотека iostream в старом стиле
```

Чтобы указать библиотеку нового стиля, в программу придется включить следующие строки.

```
#include <iostream>    // Библиотека iostream в новом стиле
using namespace std;   // Пространства имен описаны в главе 3
```

Хотя библиотеки нового стиля предпочтительнее, некоторые компиляторы их не поддерживают. В этом случае необходимо использовать библиотеки старого стиля. Заголовки библиотек нового и старого стиля приводятся в Приложении B.

Библиотеки, определенные пользователем, обычно разделяются на два файла. Один файл, называемый **заголовочным** (header file), содержит объявление каждой библиотечной функции, доступной в программе. Этот файл также содержит, например, определения констант, операторы *typedef*, перечисления, объявления структур и другие директивы *include*. По общепринятому соглашению имя заголовочного файла, ассоциированное с библиотекой, определенной пользователем, должно иметь расширение *.h*.[13] Другой файл — **файл реализации** (implementation file) — содержит определения функций, объявленных в заголовочном файле. Как правило, файл реализации имеет расширение *.cpp*.[14]

Предполагается, конечно, что эти файлы являются **исходными** (source files), т.е. их нужно компилировать. Очевидно, что было бы эффективнее компилировать определение функции только один раз, независимо от программы, а затем вставлять результаты компиляции в любое приложение. Фактически компилируется только файл реализации, а заголовочный файл вставляется в программу с помощью директивы *include*, например:

```
#include "MyHeader.h"
```

Чтобы подчеркнуть, что данный заголовочный файл относится к библиотеке, определенной пользователем, его имя заключается в двойные кавычки, а не в угловые скобки. Механизм вставки заголовочных файлов в программу зависит от конкретной системы.

---

[13] Существуют и другие соглашения, например, используются расширения *.hpp* и *.hxx*.

[14] Существуют и другие соглашения, например, используются расширения *.c*, *.cp* и *.cxx*.

Итак, программа может использовать заранее откомпилированные функции, написанные на языке C++, исходный код которых более не доступен. Эти функции, в свою очередь, могут быть написаны совершенно посторонним человеком, а не автором данной программы, как и все стандартные функции. Иными словами, в этом смысле библиотека, определенная пользователем, практически не отличается от стандартной библиотеки. Если в программу включен заголовочный файл библиотеки, способ реализации ее функций становится не важен, важно лишь, как их вызвать оттуда. Именно так и следует рассматривать библиотечные функции, даже если вы их сами написали.

Предотвращение дублирования заголовочных файлов

Поскольку заголовочные файлы могут содержать директивы *include*, не исключено, что в программу будет включено несколько копий одного и того же файла. Допустим, что мы написали библиотеку математических функций, заголовочный файл которой называется *PI.h*. В этом файле определена константа π. Затем мы создаем ряд более сложных библиотек, каждая из которых включает в себя заголовочный файл *PI.h*. Если программа включает в себя несколько заголовочных файлов, каждый из которых, в свою очередь, содержит заголовочный файл *PI.h*, возникнет многократное определение константы π, что, разумеется, является ошибкой.

Этой ошибки можно избежать, если определить заголовочный файл следующим образом.

```
#ifndef _PI_
const double PI=3.14159;
#define _PI_
#endif
```

Директивы *#ifndef*, *#define* и *#endif* являются командами **препроцессора** (preprocessor) языка C++, который может изменять исходный текст программы перед началом компиляции. Таким образом, препроцессор может обнаружить, что в программе содержится несколько определений константы *PI*. Для этого директива *#ifndef* проверяет, определен ли идентификатор препроцессора *_PI_*. Если заголовочный файл встречается впервые, этот идентификатор еще не мог быть определен, поэтому препроцессор переходит к выполнению следующего оператора. Он передает ключевое слово *const* компилятору, а директива *#define* определяет идентификатор *_PI_*. При последующих включениях *заголовочного файла PI.h директива #ifndef* обнаружит, что идентификатор *_PI_* уже определен и проигнорирует последующие операторы файла, избежав многократного определения константы *PI*.

Сравнение с языком Java

Ниже приводятся примеры, позволяющие сравнить конструкции языка Java с эквивалентными конструкциями языка C++.

Java	C++

`//  Однострочный комментарий языка Java`
`/*  Комментарий языка Java, который`
`    может занимать несколько строк */`
`/** Комментарий в стиле Javadoc */`
	`// Однострочный комментарий в C++`
	`/* Комментарий языка C++, который`
	`   может занимать несколько строк */`

`// Язык Java объединяет группы`
`// связанных между собой классов`
`// в пакеты`
	`// Язык C++ объединяет группы`
	`// связанных между собой функций`
	`// и классов в библиотеки`

`// Класс на языке Java`
`// Каждый член класса имеет модификатор`
`// доступа (public, private или`
`// protected). При обращении к пакету`
`// модификатор доступа может`
`// отсутствовать.`
`class Person`
`{`
`public String name;`
`  public int age;`
`  public double gpa;`
`} // Конец класса`
`// Обычно члены класса являются`
`// закрытыми. Приведенный пример,`
`// по существу, является структурой`
`// языка C++.`
`// После закрывающей фигурной.`
	`// Класс в языке C++`
	`// Если модификатор доступа не`
	`// указан явно, все члены класса`
	`// считаются закрытыми по умолчанию`
	`class Person`
	`{`
	`  public:`
	`  string name;`
	`  int age;`
	`  double gpa;`
	`};' // Конец класса`
	`// Структура в языке C++`
	`// Если модификатор доступа не указан`
	`// явно, все члены структуры считаются`
	`// открытыми по умолчанию`
	`struct Person`
	`{`
	`  string name;`
	`  int age;`
	`  double gpa;`
	`}; // Конец структуры`

`// Язык Java поддерживает только`
`// одиночное наследование классов.`
`// Наследование используется для`
`// указания дополнительных свойств`
	`// Язык C++ поддерживает`
	`// множественное наследование`

// В языке Java нет шаблонов.
// Вместо них существует иерархия
// одиночного наследования от класса
// Object
class Stack
{
public void push(Object newItem)
{
...
}
...
} // Конец класса Stack
	// Язык C++ поддерживает шаблоны
	// в качестве параметризованных типов
	template <class T>
	void stack<T>::push(T newItem)
	{
	...
	}

// Язык Java не полагается
// на препроцессор; все
// функциональные возможности
// обеспечиваются средствами
// самого языка
	// В языке C++ используются директивы
	// препроцессора, например, для
	// включения заголовочных файлов
	#include <iostream>
	#include "myClass.h"

// Все методы в языке Java должны // быть частью какого-то класса	// В языке C++ существуют функции, не // принадлежащие никакому классу

// Константа в языке Java должна быть
// объявлена в классе или методе
final int SIZE = 50;
	// Константа в языке C++ должна быть
	// глобальной либо объявлена внутри
	// класса или функции
	const int SIZE = 50;

// Метод, возвращающий значение,
// на языке Java
public bool isLeapYear(int year)
// Возвращает значение true,
// если год является високосным;
// в противном случае возвращает
// значение false.
{
	// Функция, возвращающая значение,
	// на языке C++
	bool isLeapYear(int year)
	// Возвращает значение true,
	// если год является високосным;
	// в противном случае возвращает
	// значение false.
	{

```
 bool leap = false;
 bool yearEndsIn00 = (year % 100 == 0);
 if (yearEndsIn00 && (year % 400 == 0))
    leap = true;
 else if (!yearEndsIn00 &&
         (year % 4 == 0))
    leap = true;
return leap;
} // Конец функции IsLeapYear
```

```
  bool leap = false;
  bool yearEndsIn00 =
       (year % 100 == 0);
  if ( yearEndsIn00 &&
     (year % 400 == 0))
     leap = true;
  else if (!yearEndsIn00 &&
          (year % 4 == 0))
     leap = true;
  return leap;
} // Конец функции isLeapYear
```

```
// Объявление переменной в языке Java
// Все переменные должны быть объявлены
// внутри какого-нибудь класса или метода
int day, month, year;
double power, x;
char response;
bool done;
// Объявление простой ссылки;
// объект не создается, пока
// не будет выполнен оператор new.
// Все объекты в языке Java
// размещаются в динамической памяти:
Sphere ball;
// Создание объекта с помощью
// конструктора по умолчанию:
Sphere ball = new Sphere();
// Использование конструктора
// с параметрами:
Sphere ball = new Sphere(1.0);
```

```
// Объявление переменной в языке C++
// Переменные могут быть объявлены
// как глобальными, так и локальными
// (внутри какого-нибудь класса или метода)
int day, month, year;
double power, x;
char response;
bool done;
// С помощью конструктора
// по умолчанию можно создавать
// статические объекты
Sphere ball();
// Использование конструктора
// с параметрами
Sphere ball(1.0);
// Для выделения динамической памяти // ис-
пользуются указатели и оператор
// new:
Sphere *ball = new Sphere();
```

```
// Равенство == и неравенство !=-
// в языке Java
// Для сравнения объектов нужно заместить
// соответствующие методы класса Object
```

```
// Равенство == и неравенство !=-
// в языке Java
// Для сравнения объектов операторы ==
// и != нужно перегрузить
```

// Массив в языке Java,
// элементарные типы
double[] r = new double[SIZE];
double[] s = new double[SIZE];
for (int i = 0; i < SIZE; i++)
r[i] = 0.0;
 	
// Массив в языке Java,
// применение ссылок
Sphere marbles = new Sphere [SIZE];
for (int i=0; i<SIZE; i++)
marbles[i] = new Sphere(0.3);
	// Массив в языке C++
	typedef double arrayType[SIZE];
	arrayType r;
	double s[SIZE];
	for (int i = 0; i < SIZE; ++i)
	r[i] = 0.0;

// Стандартный вывод в языке Java System.out.println("Введите месяц и " + "день " + year + "года как целые числа: ");	// Стандартный вывод в языке C++ cout << "Введите месяц и " << "день " << year << " года как целые числа: ";

// Стандартный ввод в языке Java
BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
String nextLine = stdin.readLine();
StringTokenizer input = new
StringTokenizer(nextLine);
x = integer.parseInt(input.nextToken());
y = Integer.parseInt(input.nextToken());
	// Стандартный ввод в языке C++
	cin >> x >> y;

Резюме

1. Каждая строка комментария в языке C++ начинается двумя обратными косыми чертами и продолжается до конца строки.
2. Идентификатор в языке C++ представляет собой последовательность букв, цифр и знаков подчеркивания, которая должна начинаться либо с буквы, либо со знака подчеркивания.
3. Для объявления нового имени типа можно использовать оператор *typedef*. Это имя является просто синонимом существующего типа. Оператор *typedef* не создает новый тип данных.
4. Именованная константа объявляется с помощью оператора, имеющего следующий вид.

   ```
   const тип идентификатор = значение;
   ```

5. Перечисление позволяет иначе назвать целочисленные константы, например:

   ```
   enum {SUN, MON, TUE, WED, THU, FRI, SAT};
   ```

   или определить новый интегральный тип, например:

   ```
   enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
   ```

6. В языке C++ применяется сокращенное вычисление выражений, содержащих логические операторы `&&` ("И") и `||` ("ИЛИ"). Иначе говоря, вычисление производится слева направо и прекращается, как только значение выражения становится очевидным.
7. Оператор вывода `<<` помещает значение в поток вывода, а оператор ввода `>>` извлекает значение из потока ввода. Можно считать, что эти операторы указывают направление потока данных. Таким образом, в выражении *cout << myVar* оператор указывает в направлении от переменной *myVar* к потоку, а в выражении *cin >> myVar* — в обратном направлении.
8. Определение функции имеет следующий вид.

   ```
   тип имя(список объявлений формальных аргументов)
       {
          тело
       }
   ```

   Функция, вычисляющая значение, возвращает его с помощью оператора *return*. Хотя функция, имеющая тип *void*, также может применять оператор *return* для выхода из своего тела, вычисленные ею значения можно возвращать только через аргументы.
9. При вызове функции количество, порядок следования и тип фактических аргументов должны соответствовать формальным аргументам.
10. Функция создает локальные копии всех фактических аргументов, переданных по значению. Следовательно, фактические аргументы, передаваемые по значению, функцией не изменяются. Такие аргументы называются входными.

    Копии аргументов, передаваемых по ссылке, не создаются. Вместо этого функция получает доступ к ячейкам памяти, где они расположены. Ссылки позволяют функции изменять значение таких аргументов, поэтому они называются выходными.

    Константные аргументы, передаваемые по ссылке, не копируются и не изменяются. Поскольку копирование входных аргументов может быть довольно трудоемким, их следует передавать по ссылке как константные.

11. Оператор *if* имеет следующий вид.

```
if (выражение)
       оператор1
   else
       оператор2
```

Если *выражение* имеет значение *true*, то выполняется *оператор*$_1$, в противном случае выполняется *оператор*$_2$.

12. Оператор *switch* имеет следующий вид.

```
switch (выражение)
{
   case константа1:
        оператор1
        break;
         ...
    case константаn:
        операторn
        break;
   default:
        оператор
}
```

Соответствующий *оператор* вычисляется в зависимости от значения *выражения*. Обычно в конце каждого раздела *case* вслед за *оператором* указывается оператор *break* (а иногда — оператор *return*). Если этот оператор пропустить, то поток управления пройдет по всем следующим разделам *case*, выполняя остальные операторы.

13. Оператор *while* имеет следующий вид.

```
while (выражение)
       оператор
```

Оператор выполняется, если *выражение* истинно. Следовательно, возможна ситуация, когда *оператор* никогда не будет выполнен.

14. Оператор *for* имеет следующий вид.

```
for (инициализация; проверка; обновление счетчика)
   оператор
```

Обычно выражение *инициализации* является выражением присваивания и выполняется только один раз. *Оператор* выполняется, если логическое выражение, которым является *проверка*, имеет значение *true*. Затем выполняется оператор *обновления счетчика*, увеличивающий или уменьшающий его значение. Эта последовательность действий повторяется, пока в результат проверки не окажется ложным.

15. Оператор *do* имеет следующий вид.

```
do
       оператор
while (выражение);
```

Оператор выполняется до тех пор, пока значение *выражения* не станет ложным. Обратите внимание, что *оператор* выполняется по крайней мере один раз.

16. Массив представляет собой совокупность элементов, имеющих одинаковый тип. К элементам массива можно обращаться с помощью индексов, отсчет которых начинается с нуля. Массивы всегда передаются функциям по ссылке.

17. Строка — это последовательность символов. Допускаются манипуляции с целой строкой, подстрокой, а также с индивидуальными символами.
18. Структура является группой связанных между собой элементов, называемых ее членами. Эти элементы могут иметь разный тип, а также могут быть другими структурами или массивами.
19. Если в ходе выполнения программы обнаружилась ошибка, можно генерировать исключительную ситуацию с помощью оператора *throw*. Исключительная ситуация перехватывается и обрабатывается предназначенным для этого кодом, помещенным в разделе *catch*.
20. Файл — это последовательность компонентов, имеющих одинаковый тип. Программа может записывать данные в файл, который будет существовать и после ее завершения. Такие файлы позволяют постоянно хранить результаты работы программы, а также передавать их другим программам в качестве входной информации. В ходе выполнения программы можно создавать временные файлы, которые могут уничтожаться после ее завершения.
21. Текстовый файл представляет собой последовательность символов, содержащую признаки конца строк. Эти символы можно считывать наравне с другими.
22. Несмотря на то что текстовый файл состоит из символов, в него можно записывать целые числа и числа с плавающей запятой. Например, если переменная *x* содержит целое число 234, то в текстовый файл будут записаны символы *2*, *3* и *4*. При этом система выполнит преобразование внутреннего представления целого числа в представление трех соответствующих символов. Аналогично, из текстового файла можно считывать символы, представляющие числовые величины, а затем преобразовывать их в целые числа или числа с плавающей точкой.
23. Бинарный файл сохраняет компоненты, используя их внутреннее представление в компьютере. Все компоненты бинарного файла также должны иметь одинаковый тип.
24. Как правило, программы на языке C++ используют заголовочные файлы, которые вставляются в них с помощью директивы *include*. Заголовочные файлы содержат объявления функций, определения констант, операторы *typedef*, перечисления, объявления структур и другие директивы *include*. Для использования функций в программе необходимо откомпилировать их файлы реализации, поместив их в библиотеку. Операционная система находит требуемые файлы реализации и объединяет их с программой.

Предупреждения

1. Помните, что оператор = является оператором присваивания, а оператор == — оператором проверки на равенство.
2. Не начинайте десятичную целочисленную константу с нуля. В этом случае она будет рассматриваться как восьмеричная или шестнадцатеричная.
3. Выражение, имеющее ненулевое значение, считается истинным. Если выражение имеет значение ноль, оно считается ложным.
4. Если в операторе *switch* пропущен оператор *break*, поток управления переходит к следующему разделу *case*.
5. Работая с индексами массива, нужно следить, чтобы они не выходили за пределы допустимого диапазона. В языке C++ нет автоматической проверки диапазона. Это относится и к строкам.

6. При ссылках на элементы многомерного массива не используются индексы, разделенные запятыми, как, например, в языке Java. Например, выражение *myArray[3,6]* является синтаксически правильным, но неверным. Для ссылки на этот элемент массива *myArray* следует использовать обозначение *myArray[3][6]*. Значением выражения *3,6*, которое называется оператором запятой, является число 6, т.е. последний элемент, перечисленный с списке. Следовательно, значением выражения *myArray[3,6]* является элемент *myArray[6]*, т.е. элемент *myArray[0][6]*.
7. Будьте осторожны, ссылаясь на элементы структуры. При обращении к ним нужно записывать как имя структуры, так и их идентификатор. Это особенно важно, когда несколько разных структур имеют члены с одинаковыми именами.
8. Исключительная ситуация, не обработанная в блоке *try-catch*, может привести к аварийному завершению работы программы.
9. Открытие существующего файла стирает содержащиеся в нем данные, если не объявлен режим добавления данных.
10. Хотя стандартные потоки ввода *cin* и вывода *cout* можно считать текстовыми файлами, они являются исключением из правил.
    - Потоки *cin* и *cout* не надо объявлять.
    - К этим потокам не применяются функции *open* и *close*. Поток *cin* всегда открыт для чтения, а поток *cout* — для вывода.
11. Потоковую переменную следует передавать функции по ссылке.

Вопросы для самопроверки

1. Какие значения будут присвоены переменным в результате выполнения следующих операторов?

```
int a = 5;
    a += 2;
    int b = a++;
    int c = (2*a + 3) % b;
    int d = (b != c) && (a + b == 3*c);
    int e = a <= b ? a : c;
```

2. Допустим, что переменные *a* и *b*, имеющие тип *int*, имеют значения 5 и 6, соответственно. Что будет выведено на экран в результате выполнения следующего оператора?

```
cout << a << b << endl;
```

3. Если переменная *title* является символьной строкой, что будет выведено на экран в результате выполнения следующих операторов?

```
string title = "Ñòåíû è çåðêàëà";
    cout << """ << title << ""\n";
```

4. Что будет выведено на экран (и будет ли выведено) в результате выполнения следующего фрагмента программы?

```
int j = 13;
    int k = 10;
    while (j >= k)
    {
```

```
        if ( (k + 1 < 10) || (k > 12) )
          cout << "Значение k выходит за допустимые пределы.\n";
        else
           switch (k + 1)
           {
              case 11:
                 cout << "j = " << j << endl;
                 break;
              case 12: case 13:
                 j += 2;
              if (j < 15)
                 cout << "j = " << j << endl;
              else
                 cout << "k = " << k << endl;
           } // end switch
        --j;
        ++k;
     } // Конец оператора while
```

5. Напишите операторы *if*, присваивающие переменной *grade* баллы A, B, C, D и F, в зависимости от значения переменной *score*: от 90 до 100 — A, от 80 до 89 — B, от 70 до 79 — C, от 60 до 69 — D, ниже 60 — F.
6. Выполните задание 5 с помощью оператора *switch*. Какое из этих решений лучше?
7. Напишите функцию, возвращающую сумму первых *n* элементов массива, состоящего из целых чисел.
8. Допустим, что массив *ArrayType* определен следующим образом.

```
const int DAYS_PER_WEEK = 7;
    const int WEEKS_PER_YEAR = 52;

    typedef double ArrayType[DAYS_PER_WEEK][WEEKS_PER_YEAR];
    ArrayType minTemps;
```

Предположим, что каждый столбец этого массива содержит минимальные значения температуры, измеренные в течение семи дней конкретной недели. Напишите на языке C++ операторы, выполняющие следующие действия.

8.1. Вывод минимальных значений температуры, измеренной в каждый день первой недели.

8.2. Вывод минимальных значений температуры, измеренной в первый день первых пяти недель.

8.3. Вывод всех минимальных значений температуры в строку, отдельно для каждой недели.

9. Рассмотрим структуру *Student*, определенную в разделе "Структуры внутри других структур". Напишите ссылки на следующие элементы структуры *student*.

9.1. Страна.

9.2. Первая цифра почтового индекса.

9.3. Средний балл.

9.4. Первая буква фамилии.

10. Рассмотрим функцию *copyTextFile*, создающую копию текстового файла, описанную в разделе "Текстовые файлы". Выполните трассировку этой функции, если файл *originalFile* содержит следующие символы.

    a b <eoln> *c d* <eoln> <eof>,

    где символ <*eoln*> является признаком конца строки, а <*eof*> — конца файла. Определите содержимое переменной *ch* и файла *copeFile*, а также положения файлового курсора в ходе выполнения функции.

Упражнения

1. Напишите функцию, получающую на вход длину сторон треугольника. Функция должна выводить на экран буквы *E*, *I* и *S*, если треугольник является равносторонним, равнобедренным или неравносторонним, соответственно. Напомним, что треугольник называется равносторонним, если у него равны все стороны, равнобедренным, если у него равны две стороны, и неравносторонним, если все стороны имеют разную длину.
2. Напишите функцию, возвращающую балл успеваемости, соответствующий заданной букве, по следующей схеме: A — это 4, B — это 3, C — это 2, D — это 1, F — это 0 и I — также 0. Используйте оператор *switch*.
3. Напишите цикл, считывающий буквы, означающие баллы успеваемости, определяющий их значение и вычисляющий средний балл. Предполагается, что каждый курс состоит из трех прослушанных серий лекций. Кроме того, для окончания работы программы пользователь должен нажимать клавишу *Q*. Примените функцию, описанную в упражнении 2.
4. Приведенные ниже операторы неверно реагируют на ввод строки *YES*. Опишите их поведение и объясните его причину.

```
char response;
do
{
    . . . (последовательность операторов)
    cout << "Повторить?";
    cin >> response;
} while ( (response == 'Y') || (response == 'y') );
```

5. Рассмотрим массив *minTemps*, описанный в задании 8 из раздела "Вопросы для самопроверки". Допустим, что одномерный массив *mt* содержит 364 (7 * 54) действительных числа.

    5.1. Допустим, что первые семь элементов массива *mt* представляют семь дней первой недели, вторые семь элементов — вторую неделю и т.д. Какой элемент массива *mt* соответствует 5-му дню 12-й недели? (Иначе говоря, какой элемент массива *mt* соответствует элементу *minTemps[4][11]*?)

    5.2. Допустим, что первые 52 элемента массива *mt* соответствуют первой строке массива *minTemps*, вторые 52 элемента — второй строке и т.д. Какой элемент массива *mt* соответствует *minTemps[4][11]*?

6. Рассмотрим массив *temps*, определенный следующими операторами.

```
typedef double WeekType[DAYS_PER_WEEK];
typedef WeekType YearType[WEEKS_PER_YEAR];
YearType temps;
```

Элемент *temp[12]* представляет собой одномерный массив, содержащий семь чисел с плавающей точкой. Напишите функцию *average*, возвращающую среднее значение его элементов.

7. Напишите функцию, создающую строку из двух заданных строк, задающих имя и фамилию студента. В результате должно получиться его полное имя.

8. Рассмотрим переменную *csc212*, определенную в разделе "Массивы структур".

   8.1. Повторите задание 9 из раздела "Вопросы для самопроверки", используя вместо структуры *student* переменную *csc212[3]*.

   8.2. Напишите операторы, считывающие имена, в поля структуры *csc212* под названием *name*.

9. Измените определение поля *name* в структуре *Person*, описанной в разделе "Структуры внутри других структур" так, чтобы переменная *Name* была структурой, состоящей из трех полей: *first*, *middle* и *last*.

10. Рассмотрим функцию *searchFileSequentially*, описанную в разделе "Текстовые файлы". Модифицируйте эту функцию так, чтобы получить преимущества, предоставляемые файлом, упорядоченным по алфавиту. Иными словами, функция не должна просматривать файл полностью, если заданного имени в нем нет.

11. Рассмотрим два текстовых файла, состоящих из целых чисел, записанных в порядке возрастания. Эти файлы можно **слить воедино** (merge), образовав третий файл, числа в котором упорядочены по возрастанию. Например, если первый файл содержит числа 1, 4 и 8, а второй — 2 и 4, то третий файл содержит числа 1, 2, 4, 4 и 8.

    Напишите функцию, объединяющую два упорядоченных файла, состоящих из целых чисел, в третий упорядоченный файл. Предполагается, что каждая строка файлов содержит только одно целое число.

12. Напишите функцию

    ```
    void advance(ofstream& myFile, char target)
    ```

    устанавливающую курсор файла *myFile* на первое вхождение символа, указанного переменной *target*. Если в переменной *target* записан пробел, функция должна установить курсор на первый пробел в файле, но не на символ конца строки. Если символа, заданного переменной *target*, в файле нет, следует вывести на экран сообщение.

13. Выполните следующие задания.

    13.1. Рассмотрим текстовый файл, состоящий из целых чисел. Напишите функцию, добавляющую в этот файл 20 целых чисел, считанных из стандартного потока ввода. Используйте режим добавления.

    13.2. Рассмотрите текстовый файл, содержащий символы. Напишите функцию, добавляющую в этот файл 20 символов, считанных из стандартного потока. Вместо режима добавления используйте временный файл.

Задания по программированию

1. Напишите программу, конвертирующую ярды, футы и дюймы в метры и сантимерты. Предполагается, что вводятся только целые числа, не содержащие ошибок. На экран должны быть выведены целые числа, приведен-

ные к нормальному виду. Иначе говоря, если в результате работы программы получается величина 3 метра и 105 сантиметров, на экран следует вывести ответ: 4 метра и 5 сантиметров.

2. Напишите программу, определяющую, образует ли матрица $N \times N$ магический квадрат. Матрица образует магический квадрат, если сумма целых чисел, стоящих в каждой строке, столбце и диагонали, одинакова.

   Например, приведенная ниже матрица является магическим квадратом, поскольку сумма целых чисел, стоящих в каждой строке, каждом столбце и каждой диагонали, равна 15.

   ```
   6  1  8
   7  5  3
   2  9  4
   ```

3. Напишите программу, считывающую текст на английском языке и перечисляющую все обнаруженные слова в алфавитном порядке, одновременно подсчитывая их частоту.

   Ядром этой программы является функция, считывающая отдельное слово. Если принято соглашение, что все символы, отличные от букв, считаются разделителями, то словом можно назвать любую строку, содержащую до восьми символов и окруженную разделителями. Если строка между разделителями состоит из более чем восьми символов, следует считать всю строку, обрезав ее при записи.

   Для простоты можно предположить, что текст содержит не более 100 слов.

4. Напишите функцию, имитирующую сложение двоичных величин, состоящих из n бит. Количество бит изменяется от 0 до $n-1$. Для представления n-битовых величин используйте массив `bitArray`: элемент `bitArray[i]` равен 1 тогда и только тогда, когда i-й бит n-битовой величины равен 1.

Приложение Б. ASCII-коды символов

Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL	32	(пробел)	64	@	96	` (обратная кавычка)
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'(апостроф)	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	, (запятая)	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x

Код	Символ	Код	Символ	Код	Символ	Код	Символ
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_(подчеркивание)	127	DEL
Примечание: коды 0-31 и 127 зарезервированы за управляющими символами и не выводятся на печать.							

Ключевые слова языка C++

Ключевые слова языка C++ нельзя использовать для других целей.

```
asm             else            operator        throw
auto            enum            private         true
bool            explicit        protected       try
break           extern          public          typedef
case            false           register        typeid
catch           float           reinterpret_cast typename
char            for             return          union
class           friend          short           unsigned
const           goto            signed          using
const_cast      if              sizeof          virtual
continue        inline          static          void
default         int             static_cast     wchar_t
delete          long            struct          while
do              mutable         switch
double          namespace       template
dynamic_cast    new             this
```

Приложение В. Заголовочные файлы и стандартные функции в языке C++

В этом приложении приведен список заголовочных файлов, используемых в языке C++. Имена старых версий этих файлов указаны в скобках.

cassert (assert.h)

Эта библиотека содержит только функцию *assert*. Эта функция применяется для проверки диагностических утверждений.

```
assert(диагностическое утверждение);
```

Если диагностическое утверждение ложно, функция *assert* выводит на экран сообщение об ошибке и прекращает выполнение программы. Все вызовы функции *assert* в программе можно заблокировать с помощью макроса *#define NDEBUG*, который следует поместить перед директивами *include*.

cctype (ctype.h)

Большинство функций этой библиотеки позволяет распознавать буквы, цифры и т.д. Две остальные функции преобразуют строчные буквы в прописные, и наоборот.

Функции, предназначенные для распознавания, возвращают значение *true*, если символ *ch* принадлежит указанной группе; в противном случае они возвращают значение *false*.

`isalnum(ch)`	Возвращает значение *true*, если символ *ch* является буквой или десятичной цифрой
`isalpha(ch)`	Возвращает значение *true*, если символ *ch* является буквой
`iscntrl(ch)`	Возвращает значение *true*, если символ *ch* является управляющим (т.е. его ASCII-код равен 127 или изменяется от 0 до 31)
`isdigit(ch)`	Возвращает значение *true*, если символ *ch* является десятичной цифрой
`isgraph(ch)`	Возвращает значение *true*, если символ *ch* может быть выведен на печать и не является пробелом
`islower(ch)`	Возвращает значение *true*, если символ *ch* является строчной буквой
`isprint(ch)`	Возвращает значение *true*, если символ *ch* может быть выведен на печать (включая пробел)
`ispunct(ch)`	Возвращает значение *true*, если символ *ch* является знаком пунктуации

`isspace(ch)`	Возвращает значение *true*, если символ *ch* является пробельным: пробелом, знаком табуляции, командой перехода на новую строку или прогона бумаги
`isupper(ch)`	Возвращает значение *true*, если символ *ch* является прописной буквой
`isxdigit(ch)`	Возвращает значение *true*, если символ *ch* является шестнадцатеричной цифрой
`toascii(ch)`	Вернуть ASCII-код символа *ch*
`tolower(ch)`	Если символ *ch* является строчной буквой, преобразовать ее в прописную; в противном случае вернуть символ *ch*
`toupper(ch)`	Если символ *ch* является прописной буквой, преобразовать ее в строчную; в противном случае вернуть символ *ch*

cfloat (float.h)

В этой библиотеке определены именованные константы, указывающие диапазон изменения значений с плавающей точкой.

climits (limits.h)

В этой библиотеке определены именованные константы, указывающие диапазон изменения целочисленных значений.

cmath (math.h)

Функции, содержащиеся в этой библиотеке, предназначены для стандартных математических вычислений. Эти функции являются перегруженными и выполняют вычисления с числами, имеющими тип *float*, *double* и *long double*. Если не указано иное, каждая функция имеет один аргумент, а возвращаемое значение и аргумент имеют одинаковый тип (*float*, *double* и *long double*).

`acos`	Вычисляет арккосинус
`asin`	Вычисляет синус
`atan`	Вычисляет арктангенс
`atan2`	Вычисляет арктангенс *x/y* для аргументов *x* и *y*
`ceil`	Выполняет округление с избытком
`cos`	Вычисляет косинус
`cosh`	Вычисляет арккосинус
`exp`	Вычисляет экспоненту
`fabs`	Вычисляет абсолютную величину
`floor`	Выполняет округление с недостатком
`fmod`	Возвращает остаток от деления аргумента *x* на аргумент *y*
`fexp`	Для аргументов *x* и *eptr* , где $x=m*2^e$, возвращает число *m* и устанавливает *eptr* равным *e*
`ldexp`	Возвращает значение $x*2^e$ для аргументов *x* и *e*
`log`	Возвращает значение натурального логарифма
`log10`	Возвращает значение десятичного логарифма
`modf`	Для аргументов *x* и *iptr* возвращает дробную часть числа *x* и устанавливает *iptr* равным целой части аргумента *x*

pow	Для аргументов *x* и *y* возвращает значение x^y
sin	Вычисляет синус
sinh	Вычисляет гиперболический синус
sqrt	Вычисляет квадратный корень
tan	Вычисляет тангенс
tanh	Вычисляет гиперболический тангенс

cstdlib (stdlib.h)

abort	Аварийно завершает выполнение программы
abs	Вычисляет абсолютную величину числа
atof	Преобразовывает строку в число с плавающей точкой
atoi	Преобразовывает строку в целое число
exit	Прерывает выполнение программы
rand	Вычисляет псевдослучайное целое число
srand	Инициализирует генератор псевдослучайных чисел аргументом, а в отсутствие аргументов — единицей.

cstring (string.h)

Библиотека содержит функции, позволяющие манипулировать строками языка C, завершающимися нулевым символом *\0*. Если не указано иное, функции возвращают указатель на результирующую строку, модифицируя один из аргументов. Аргумент *ch* является символом, *n* — целое число, остальные строки являются строками.

strcat(toS, fromS)	Копирует строку *fromS* в конец строки *toS*
stncat(toS, fromS, n)	Копирует не более *n* символов строки *fromS* в конец строки *toS* и дописывает символ *\0*
strcmp(str1, str2)	Возвращает отрицательное целое число, если *str1<str2*, нуль, если *str1=str2*, положительное целое число, если *str1>str2*.
stricmp(str1, str2)	Аналогична функции *strcmp*, но игнорирует различия между прописными и строчными буквами
strncmp(str1, str2, n)	Аналогична функции *strcmp*, но сравнивает только первые *n* символов каждой строки
strcpy(toS, fromS)	Копирует строку *fromS* в *toS*
strncpy(toS, fromS, n)	Копирует *n* символов строки *fromS* в *toS*, при необходимости обрывая ее или дополняя нулевыми символами *\0*
strspn(str1, str2)	Возвращает количество первых последовательных символов строки *str1*, которые отсутствуют в строке *str2*
strcspn(str1, str2)	Возвращает количество первых последовательных символов строки *str1*, которые принадлежат строке *str2*

`strlen(str)`	Возвращает длину строки *str*, не учитывая символ *\0*
`strlwr(str)`	Преобразовывает прописные буквы строки *str* в строчные, не изменяя других символов
`strupr(str)`	Преобразовывает строчные буквы строки *str* в прописные, не изменяя других символов
`strchr(str, ch)`	Возвращает указатель на первое вхождение символа *ch* в строку *str*; если в строке символа нет, возвращает *NULL*
`strrchr(str, ch)`	Возвращает указатель на последнее вхождение символа *ch* в строку *str*; если символа в строке нет, возвращает *NULL*
`strpbrk(str1, str2)`	Возвращает указатель на последнее вхождение символа строки *str1* в строку *str2*; если таких символов нет, возвращает *NULL*
`strstr(str1, str2)`	Возвращает указатель на последнее вхождение символа строки *str2* в строку *str1*; если таких символов нет, возвращает *NULL*
`strtok(str1, str2)`	Находит в строке *str1* следующую лексему, за которой следует строка *str2*, возвращает указатель на эту лексему и записывает непосредственно после нее символ *NULL*

exception

В этой библиотеке определены классы, типы и функции, связанные с обработкой исключительных ситуаций. Ниже приведен фрагмент класса *exception*.

```
class exception
{
public:
   exception() throw();
   virtual ~exception() throw();
   exception &operator=(const exception %exc) throw();
   virtual const char *what() const throw();
}
```

fstream (fstream.h)

В этой библиотеке объявлены классы, поддерживающие ввод и вывод.

iomanip (iomanip.h)

Манипуляторы, содержащиеся в этой библиотеке, влияют на формат ввода и вывода. Обратите внимание, что в библиотеке *iostream* есть дополнительные манипуляторы.

`setbase(b)`	Задает основание счисления *b*=8, 10 или 16
`setfill(f)`	Задает символ заполнения *f*
`setprecision(n)`	Параметр *n* задает точность представления чисел с плавающей точкой
`setw(n)`	Параметр *n* задает ширину поля вывода

iostream (*iostream.h*)

Манипуляторы, содержащиеся в этой библиотеке, влияют на формат ввода и вывода. Обратите внимание, что в библиотеке *iomanip* есть дополнительные манипуляторы.

`dec`	Вынуждает операторы, выполняемые в последующем, использовать десятичное представление чисел
`endl`	Вставляет символ перехода на новую строку `\n` и очищает выходной поток
`ends`	Вставляет символ окончания строки `\0` в выходной поток
`flush`	Очищает выходной поток
`hex`	Вынуждает операторы ввода-вывода, выполняемые в дальнейшем, использовать шестнадцатеричное представление чисел
`oct`	Вынуждает операторы ввода-вывода, выполняемые в дальнейшем, использовать восьмеричное представление чисел
`ws`	Извлекает из входного потока пробельные символы

string

Эта библиотека позволяет манипулировать со строками языка C++. Ниже представлены некоторые из функций, предусмотренных в этой библиотеке. Кроме того, к строкам можно применять операторы `=`, `+`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `<<` и `>>`. Обратите внимание, что нумерация позиций строки начинается с нуля.

`erase()`	Стирает содержимое строки, делая ее пустой
`erase(pos, len)`	Удаляет из строки подстроку, начинающуюся с позиции *pos* и содержащую *len* символов
`find(subString)`	Возвращает позиции подстроки в строке
`length()`	Возвращает количество символов в строке (то же, что и функция *size*)
`replace(pos, len, str)`	Заменяет подстроку, начинающуюся с позиции *pos* и содержащую *len* символов, строкой *str*
`size()`	Возвращает количество символов в строке (то же, что и функция *length*)
`substr(pos, len)`	Возвращает подстроку, начинающуюся с позиции *pos* и содержащую *len* символов

Приложение Г. Метод математической индукции

Многие доказательства теорем в компьютерных науках основаны на **методе математической индукции** (mathematical induction). Индукция — это математический принцип, напоминающий ряд костяшек домино, поставленных вертикально. Если толкнуть первую костяшку, все остальные упадут одна за другой. Что характерно для такой ситуации? Если мы докажем, что вследствие падения первой костяшки, вторая костяшка упадет, то придем к выводу, что падение первой костяшки приведет к падению всех остальных, одна за другой. Говоря более формально, все костяшки упадут, если выполняются следующие два условия.

- Первая костяшка падает.
- Для любого номера $k \geq 1$, если k-я костяшка падает, то упадет также (k+1)-я костяшка.

Принцип математической индукции — это аксиома, формулирующаяся следующим образом.

Аксиома D-1. Принцип математической индукции. Утверждение $P(n)$, зависящее от номера n, является истинным для всех чисел $n \geq 0$, если истинными являются следующие утверждения.

1. Утверждение $P(0)$ истинно.
2. Если утверждение $P(k)$ истинно для любого $k \geq 0$, то утверждение $P(k+1)$ также истинно.

Доказательство методом индукции по n представляет собой одно из применений принципа математической индукции. Такое доказательство состоит из двух шагов, описанных в аксиоме D-1. Первый шаг называется **базисом индукции** (basis, или basis case) . Второй шаг называется **шагом индукции**. Как правило, шаг индукции разделяется на две части: **индуктивную гипотезу** ("если утверждение $P(k)$ истинно для любого $k \geq 0$") и **индуктивное заключение** ("то утверждение $P(k+1)$ также истинно").

Пример 1

Рекурсивная функция, заданная приведенным ниже псевдокодом, вычисляет значение x^n.

```
pow2(in x:integer, in n:integer)

   if (n == 0)
      return 1
   else
      return x * pow2(x, n-1)
```

Используя индукцию по n, легко доказать, что функция *pow2* вычисляет значение x^n для всех $n \geq 0$.

Базис. Покажем, что утверждение истинно при n=0. Иначе говоря, нужно доказать, что значение *pow2(x, 0)* равно x^0, т.е. 1. Это утверждение непосредственно следует из определения функции *pow2*.

Теперь необходимо выполнить шаг индукции. Предполагая, что утверждение истинно для n=k (индуктивная гипотеза), следует доказать, что оно будет истинным при n=k+1 (индуктивное заключение).

Индуктивная гипотеза. ***Допустим, что утверждение истинно для n=k.*** Иными словами, предположим, что

```
pow2(x, k) = x^k
```

Индуктивное заключение. ***Покажем, что утверждение истинно при n=k+1.*** Иначе говоря, необходимо доказать, что функция *pow2(x, k+1)* возвращает значение x^{k+1}. По определению функции *pow2*

```
pow2(x,k+1) = x*pow2(x,k)
```

По индуктивной гипотезе, величина *pow2(x,k)* равна x^k, следовательно,

```
pow2(x, k+1) = x*x^k = x^(k+1)
```

Именно это и требовалось доказать.

Таким образом, индуктивное доказательство завершено. Мы продемонстрировали, что два утверждения, входящих в аксиому D-1, истинны, поэтому принцип математической индукции гарантирует, что функция *pow2* возвращает значение x^n для всех чисел $n \geq 0$. **Конец доказательства.**

Пример 2

Доказать, что

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2} \text{ для всех } n \geq 1.$$

Обозначим сумму $1 + 2 + \ldots + n$ через S_n.

Базис. Иногда утверждение при n=0 является тривиальным, как в данном случае. Тогда в качестве базиса следует взять значение n=1. (На самом деле в качестве базиса можно выбирать любое значение $n \geq 0$, однако, как правило, используются значения 0 или 1.)

Нужно доказать, что сумма S_1 равна значению выражения 1(1+1)/2. Этот факт очевиден.

Индуктивная гипотеза. Допустим, что при n=k формула истинна, т.е. S_k=k(k+1)/2.

Индуктивное заключение. Покажем, что формула истинна при n=k+1. Для этого можно поступить следующим образом.

S_{k+1} = (1 + 2 + ... + k) + (k+1)	(определение суммы S_{k+1})
= S_k + (k+1)	(определение суммы S_k)
= k(k+1)/2 + (k+1)	(индуктивная гипотеза)
= (k(k+1) + 2(k+1))/2	(приведение к общему знаменателю)
= (k+1)(k+2)/2	(факторизация)

Последнее выражение при n=k+1 равно $n(n+1)/2$. Следовательно, если формула для суммы S_k верна, то формула для суммы S_{k+1} также верна. Следова-

тельно, по принципу математической индукции формула справедлива для всех $n \geq 1$. **Конец доказательства.**

Пример 3

Доказать, что $2^n > n^2$ при $n \geq 5$

Базис. Эта задача иллюстрирует случай, когда базисом индукции является не условие n=0 или n=1, а условие n=5. Очевидно, что при n=5 отношение истинно, поскольку

$2^5 = 32 > 5^2 = 25$.

Индуктивная гипотеза. Допустим, что неравенство выполняется при $n = k \geq 5$, т.е. $2^k > k^2$ при $k \geq 5$.

Индуктивное заключение. Покажем, что неравенство выполняется при $n = k$+1, т.е. $2^{k+1} > (k+1)^2$ при $k \geq 5$. Для этого запишем следующую цепочку утверждений.

$(k+1)^2 = k^2 + (2k + 1)$ (квадрат k+1)
$< k^2 + k^2$ при $k \geq 5$ ($2k + 1 < k^2$, см. упражнение 2)
$< 2^k + 2^k$ при $k \geq 5$ (индуктивная гипотеза)
$= 2^{k+1}$.

Следовательно, по принципу математической индукции $2^n > n^2$ при $n \geq 5$. **Конец доказательства.**

Иногда индуктивная гипотеза в аксиоме D-1 не подтверждается, т.е. кроме утверждения $P(k)$ необходимо предположить что-то еще. Эта ситуация описывается более строгой формой принципа математической индукции.

Аксиома D-2. Принцип математической индукции (строгая форма). Утверждение $P(n)$ зависящее от числа n истинно для всех $n \geq 0$, если выполняются следующие условия.

1. Утверждение $P(0)$ истинно.
2. Если утверждения $P(0)$, $P(1)$, ... , $P(k)$ истинны для любого $k \geq 0$, то утверждение P(k+1) также истинно.

Обратите внимание, что индуктивная гипотеза аксиомы D-2 ("если утверждения $P(0)$, $P(1)$, ... , $P(k)$ истинны для любого $k \geq 0$") включает в себя индуктивную гипотезу аксиомы D-1 ("если утверждение $P(k)$ истинно для любого $k \geq 0$").

Пример 4

Доказать, что каждое целое число, которое больше 1, можно записать в виде произведения простых целых чисел.

Напомним, что простым называется число, которое делится только на 1 и само на себя. Рассмотрим индуктивное доказательство этого утверждения.

Индуктивная гипотеза. Предположим, что утверждение истинно для каждого целого числа 2, 3, ..., k, где $k \geq 2$.

Индуктивное заключение. Покажем, что утверждение выполняется для $n = k + 1$, т.е. число $k + 1$ можно записать как произведение простых чисел.

Если число $k + 1$ является простым, то утверждение очевидно. Однако, если число $k + 1$ не является простым, оно должно делиться на целое число x, удовлетворяющее условию $0 < x < k + 1$. Таким образом,

$k + 1 = x * y$,

где $1 < y < k + 1$. Обратите внимание, что числа x и y не превосходят числа k, поэтому можно применить индуктивную гипотезу. Иначе говоря, числа x и y можно

записать в виде произведения простых чисел. Очевидно, что произведение $x * y$, равное числу $k+1$, представляет собой произведение простых чисел. Поскольку формула верна для $n = k + 1$, она верна для всех $n \geq 2$. **Конец доказательства.**

Пример 5

В главе 2 обсуждалось следующее рекурсивное решение.

$rabbit(1) = 1$,

$rabbit(2) = 2$,

$rabbit(n) = rabbit(n-1) + rabbit(n-2)$ при $n > 2$.

Доказать, что

$rabbit(n) = (a^n + b^n)/\sqrt{5}$,

где $a = (1 + \sqrt{5})/2$, а $b = (1 - \sqrt{5})/2 = 1 - a$.

Базис. Число $rabbit(0)$ не определено, поэтому начнем с условия $n = 1$. Вычисления показывают, что $rabbit(1) = (a^1 + b^1)/\sqrt{5} = 1$. Заметим, однако, что значение $rabbit(2)$ также является особым случаем. Иначе говоря, значение $rabbit(2)$ невозможно вычислить по значению $rabbit(1)$, пользуясь указанной рекуррентной формулой. Следовательно, в базис следует включить условие $n = 2$.

Вычисления при $n = 2$ показывают, что $rabbit(2) = (a^2 + b^2)/\sqrt{5} = 1$. Следовательно, формула верна при $n = 1$ и $n = 2$.

Индуктивная гипотеза. Допустим, что формула верна для всех чисел n из диапазона $1 \leq n \leq k$, где число k не меньше 2.

Индуктивное заключение. Покажем, что формула истинна при $n = k + 1$. Для этого запишем следующие соотношения.

$$\begin{aligned} rabbit(k+1) &= rabbit(k) + rabbit(k-1) && \text{(рекуррентное соотношение)} \\ &= [(a^k - b^k) + (a^{k-1} - b^{k-1})] / \sqrt{5} && \text{(индуктивная гипотеза)} \\ &= [a^{k-1}(a^2) - b^{k-1}(b^2)] / \sqrt{5} && (a + 1 = a^2;\ b + 1 = b^2) \\ &= (a^{k+1} + b^{k+1})/\sqrt{5}\,. \end{aligned}$$

Поскольку эта формула верна при $n = k + 1$, по принципу математической индукции она выполняется при всех $n > 2$. **Конец доказательства.**

Обратите внимание, что в предыдущем доказательстве использовался факт, что $a + 1 = a^2$ и $b + 1 = b^2$. Эти соотношения можно проверить с помощью простых вычислений. Вообще говоря, в индуктивных доказательствах часто возникает необходимость доказать какое-либо вспомогательное утверждение. Следовательно, наше доказательство можно считать законченным, только если мы показали, что $a + 1 = a^2$ и $b + 1 = b^2$. Как видим, индуктивное доказательство часто связано с громоздкими алгебраическими вычислениями!

Вопросы для самопроверки

1. Докажите, что $1 + 2^1 + 2^2 + \ldots + 2^m = 2^{m-1} - 1$ при всех $m \geq 0$.
2. Докажите, что сумма первых n нечетных положительных чисел равна n^2.
3. Докажите неравенство $rabbit(n) \geq a^{n-2}$ при $n \geq 2$ и $a = (1 + \sqrt{5})$.

Упражнения

1. Докажите, что сумма первых n четных положительных целых чисел равна $n(n+1)$.
2. Докажите, что $1^2 + 2^2 + \ldots + n^2 = n(n+1)(2n+1)/6$ при всех $n \geq 1$.
3. Докажите, что $2n + 1 < n^2$ для всех $n \geq 3$.
4. Докажите, что $n^3 - n$ делится на 6 при всех $n \geq 0$.
5. Докажите, что $2^n > n^3$ при $n \geq 10$.
6. Докажите, что $n! > n^3$ при достаточно большом n.
7. Напомним рекурсивное определение из главы 2.

   $c(n, 0) = 1$,

   $c(n, n) = 1$,

   $c(n, k) = c(n-1, k-1) + c(n-1, k)$ при $0 < k < n$,

   $c(n, k) = 0$ при $k > n$.

   7.1. Докажите, что $c(n, 0) + c(n, 0) + \ldots + c(n, n) = 2^n$.

   Подсказка: воспользуйтесь тем, что $c(n + 1, 0) = c(n, 0)$ и $c(n + 1, n) = c(n, n)$.

   7.2. Докажите, что $(x + y)^n = \sum_{k=0}^{n} c(n,k)x^k y^{n-k}$.

8. Докажите неравенство: $rabbit(n) \leq a^{n-1}$ при $n \geq 1$ и $a = (a + \sqrt{5})/2$.
9. Предположим, что популяция кроликов каждый год удваивается. В начальный момент времени она состоит из двух кроликов. Докажите формулу, предсказывающую количество кроликов через n лет.

Приложение Д. Стандартные шаблонные классы

Класс list

Контейнерный класс *list* из стандартной библиотеки шаблонов STL имеет два параметра. Первый шаблонный параметр является типом данных, хранящихся в контейнере, а второй — распределителем памяти для контейнера. По умолчанию распределитель памяти является объектом класса *allocator*. Как правило, большинству приложений этого достаточно. Ниже приводится фрагмент листинга методов класса *list*.

```
template < class T, class A = allocator<T> >
class list
{
public:
   list();
   // Конструктор по умолчанию; инициализирует пустой список.

   list(size_type num, const T& val = T());
   // Конструктор; инициализирует список из num элементов,
   // имеющих значение val.

   list(const list<T> & anotherList);
   // Конструктор; инициализирует список, являющийся копией
   // списка anotherList.

   bool empty() const;
   // Определяет, пуст ли список.

   size_type size() const;
   // Возвращает количество элементов, находящихся в списке.
   // Тип size_type является интегральным.

   size_type max_size();
   // Определяет максимально возможное количество
   // элементов списка.

   iterator insert(iterator i, const T& val = T());
   // Вставляет элемент val в список, непосредственно перед
   // элементом, указанным итератором i. Возвращается
   // итератор, установленный на вставленный элемент.
```

```
    void remove(const T& val);
    // Удаляет из списка все элементы, имеющие значение val.

    iterator erase(iterator i);
    // Удаляет из списка элемент, на который указывает
    // итератор i. Возвращает итератор на элемент, следующий
    // за удаленным. Если удаленный элемент был последним,
    // значение итератора совпадает со значением итератора,
    // возвращаемого функцией end().

    iterator begin();
    // Возвращает итератор на первый элемент списка.
    // Если список пуст, значение итератора совпадает со
    // значением итератора, возвращаемого функцией end().

    iterator end();
    // Возвращает значение итератора, которое можно
    // использовать для проверки, достигнут ли конец списка.
} // Конец класса list
```

Класс stack

Стандартный класс *stack* из библиотеки STL является адаптерным и имеет два шаблонных параметра. Первый шаблонный параметр является типом данных, хранящихся в контейнере, а второй — контейнером, который используется в реализации стека. По умолчанию в качестве контейнера применяется объект класса *deque*. Ниже приводится фрагмент листинга методов класса *stack*.

```
template <class T, class Container = deque <T> >
class stack
{
public:
    explicit stack(const & cnt = Container());
    // Конструктор по умолчанию; инициализирует пустой стек.

    bool empty() const;
    // Определяет, пуст ли стек.

    size_type size() const;
    // Возвращает количество элементов, находящихся
    // в стеке в данный момент. Тип size_type является
    // интегральным.

    T &top();
    // Возвращает ссылку на вершину стека.

    void pop();
    // Удаляет вершину стека.

    void push(const T& x);
    // Добавляет элемент на вершину стека.
} // Конец класса stack
```

Класс queue

Стандартный класс *queue* из библиотеки STL является адаптерным и имеет два шаблонных параметра. Первый шаблонный параметр является типом данных, хранящихся в контейнере, а второй — контейнером, который используется в реализации очереди. По умолчанию в качестве контейнера применяется объект класса *deque*. Ниже приводится фрагмент листинга методов класса *queue*.

```
template <class T, class Container = deque <T> >
class queue
{
public:
explicit queue(const& cnt = Container());
   // Конструктор по умолчанию; инициализирует пустую очередь.

   bool empty() const;
   // Определяет, пуста ли очередь.

   size_type size() const;
   // Возвращает количество элементов, находящихся в стеке
   // в данный момент. Тип size_type является интегральным.

   T &front();
  // Возвращает ссылку на первый элемент очереди.

   T &back();
   // Возвращает ссылку на последний элемент очереди.

   void pop();
   // Удаляет первый элемент из очереди.

   void push(const T& x);
   // Вставляет элемент в конец очереди.
} // Конец класса queue
```

Приложение E. Операторы языка C++

Операторы языка C++, помещенные внутри одной и той же рамки, имеют одинаковый приоритет. Операторы, имеющие более высокий приоритет, указываются в таблице выше остальных операторов.

Оператор	Смысл	Ассоциативность	Использование
`::`	глобальная переменная	правая	`::` *имя*
`::`	разрешение области видимости	левая	*имя_класса*`::`*имя_члена*
`->`	выбор члена	левая	*указатель*`->`*член*
`.`	выбор члена	левая	*объект*`.`*член*
`[]`	индекс массива	левая	*имя_массива*`[`*выражение*`]`
`()`	вызов функции	левая	*имя_функции*`(`*список_выражений*
`()`	создание типа	левая	`type` *(список выражений)*
`++`	постфиксная инкрементация	правая	*левое_значение*`++`
`--`	постфиксная декрементация	правая	*левое_значение*`--`
`typeid`	идентификация типов	правая	`typeid` (*тип*) `typeid` (*выражение*)
`dynamic_cast`	контролируемое преобразование типа	правая	`dynamic_cast<`*тип*`>` *(выражение)*
`static_cast`	контролируемое преобразование типа	правая	`static_cast<`*тип*`>`*(выражение)*
`reinterpret`	неконтролируемое преобразование типа	правая	`reinterpret_cast<`*тип*`>` *(выражение)*
`const_cast`	преобразование константы	правая	`const_cast<`*тип*`>`*(выражение*
`sizeof`	размер типа	правая	`sizeof`(*тип*)
`sizeof`	размер типа	правая	`sizeof`(*тип*)
`++`	префиксная инкрементация	правая	`++`*левое_значение*

Оператор	Смысл	Ассоциативность	Использование
`--`	префиксная декрементация	правая	`--`*левое_значение*
`~`	побитовое дополнение	правая	`~`*выражение*
`!`	логическое НЕТ	правая	`!`*выражение*
`+`	унарный плюс	правая	`+`*выражение*
`-`	унарный минус	правая	`-`*выражение*
`*`	разыменование	правая	`*`*выражение*
`&`	взятие адреса	правая	`&`*левое_значение*
`()`	приведение	правая	`(`*тип*`)` *выражение*

`new`	выделение памяти	правая	`new` *тип*
			`new` *тип* `(`*список_выражений*`)`
			`new` `(`*список_выражений*`)` *тип*
			`new` `(`*список_выражений*`)` *тип* `(`*список_выражений*`)`

`delete`	очистка памяти	правая	`delete` *указатель* `delete []` *указатель*
`->*`	выбор члена	левая	*указатель*`->`*указатель_на_член*
`.*`	выбор члена	левая	*объект*`.*`*указатель_на_член*
`*`	умножение	левая	*выражение* `*` *выражение*
`/`	деление	левая	*выражение* `/` *выражение*
`%`	остаток	левая	*выражение* `%` *выражение*
`+`	сложение	левая	*выражение* `+` *выражение*
`-`	вычитание	левая	*выражение* `-` *выражение*
`<<`	побитовый сдвиг влево[1]	левая	*выражение* `<<` *выражение*
`>>`	побитовый сдвиг вправо	левая	*выражение* `>>` *выражение*
`<`	меньше	левая	*выражение* `<` *выражение*
`<=`	меньше или равно	левая	*выражение* `<=` *выражение*
`>`	больше	левая	*выражение* `>` *выражение*
`>=`	больше или равно	левая	*выражение* `>=` *выражение*
`==`	равно	левая	*выражение* `==` *выражение*
`!=`	не равно	левая	*выражение* `!=` *выражение*

[1] Обычно перегружается для ввода-вывода.

Оператор	Смысл	Ассоциативность	Использование
`&`	побитовое И	левая	*выражение* `&` *выражение*
`^`	побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ	левая	*выражение* `^` *выражение*
`|`	побитовое ИЛИ	левая	*выражение* `|` *выражение*
`&&`	логическое И	левая	*выражение* `&&` *выражение*
`||`	логическое ИЛИ	левая	*выражение* `&&` *выражение*
`?:`	условный	левая	*выражение* `?` *выражение* `:` *выражение*
`=`	присваивание	левая	*левое_значение* `=` *выражение*
`*=`	умножение и присваивание	левая	*левое_значение* `*=` *выражение*
`/=`	деление и присваивание	левая	*левое_значение* `/=` *выражение*
`%=`	вычисление остатка и присваивание	левая	*левое_значение* `%=` *выражение*
`+=`	сложение и присваивание	левая	*левое_значение* `+=` *выражение*
`-=`	вычитание и присваивание	левая	*левое_значение* `-=` *выражение*
`<<=`	сдвиг влево и присваивание	левая	*левое_значение* `<<=` *выражение*
`>>=`	сдвиг вправо и присваивание	левая	*левое_значение* `>>=` *выражение*
`&=`	побитовый оператор “И” и присваивание	левая	*левое_значение* `&=` *выражение*
`|=`	побитовый оператор “ИЛИ” и присваивание	левая	*левое_значение* `|=` *выражение*
`^=`	побитовый оператор “ИСКЛЮЧАЮЩЕГО ИЛИ” и присваивание	левая	*левое_значение* `^=` *выражение*
`,`	запятая	левая	*выражение*`,` *выражение*

Словарь терминов

2-3 дерево — дерево, в котором каждый внутренний узел (не лист) имеет два или три дочерних узла, причем все листья расположены на одном и том же уровне. Каждый узел может иметь левое, среднее и правое поддерево. Если узел имеет два дочерних узла и содержит один элемент данных, значение поискового ключа этого узла должно быть больше, чем значение поискового ключа его левого дочернего узла, и меньше, чем значение поискового ключа его правого дочернего узла. Если узел имеет три дочерних узла и содержит два элемента данных, значение наименьшего поискового ключа этого узла должно быть больше, чем значение поискового ключа его левого дочернего узла, и меньше, чем значение поискового ключа его среднего дочернего узла, а значение наибольшего поискового ключа этого узла должно быть больше, чем значение поискового ключа его среднего дочернего узла, и меньше, чем значение поискового ключа его правого дочернего узла.

2-3-4 дерево — дерево, в котором каждый внутренний узел (не лист) имеет два, три или четыре дочерних узла, причем все листья расположены на одном и том же уровне. Каждый узел может иметь левое, левое среднее, правое среднее и правое поддерево. Если узел имеет четыре дочерних узла и содержит три элемента данных, значение его наименьшего поискового ключа этого узла должно быть больше, чем значение поискового ключа его левого дочернего узла, и меньше, чем значение поискового ключа его левого среднего дочернего узла; значение среднего поискового ключа должно быть больше, чем значение поискового ключа его левого среднего дочернего узла, и меньше, чем значение поискового ключа его правого среднего дочернего узла; значение наибольшего поискового ключа должно быть больше, чем значение поискового ключа его правого среднего дочернего узла, и меньше, чем значение поискового ключа его правого дочернего узла.

$a_1 = 1$, $a_2 = 1$, $a_n = a_{n-1} + a_{n-2}$ для $n>2$.

AVL-дерево — сбалансированное бинарное дерево, в котором баланс после каждой вставки или удаления узла восстанавливается с помощью вращения.

B-дерево степени m — сбалансированное дерево поиска, листья которого находятся на одинаковом уровне, а узлы содержат от $m-1$ до $[m/2]$ записей. Каждый узел, не являющийся листом, может содержать по меньшей мере одну запись и иметь не меньше двух дочерних узлов. Обычно хранится во внешнем файле.

Interface — механизм взаимодействия между модулями системы.

O($f(n)$) — порядок функции $f(n)$. См. *обозначение O-большое* и *порядок алгоритма.*

`void`-функция — функция, не возвращающая никаких значений. См. также *функция, имеющая значение.*

Абстрактный базовый класс — класс, не имеющий экземпляров и являющийся основой для создания производных классов. Абстрактный базовый класс должен содержать по крайней мере одну чисто виртуальную функцию.

Абстрактный тип данных (АТД) — совокупность данных и точно определенных операций над ними.

Абстрактный тип данных, ориентированный на значение, — абстрактный тип данных, операции над которым зависят от значений его элементов, а не от их позиции. См. также *позиционно-ориентированный абстрактный тип данных*.

Абстракция данных — принципы разработки программ, которые позволяют отделить набор операций, применяемых к совокупности данных, от способа их реализации. См. также *функциональная абстракция*.

Абстракция — см. *абстракция данных* и *функциональная абстракция*.

Агрегированный тип данных — тип данных, состоящий из нескольких элементов. Примерами агрегированных типов данных являются массивы, структуры и файлы.

Адаптер — контейнерный класс, обеспечивающий ограниченный интерфейс для работы с другим контейнером, используемым в его реализации.

Адрес — номер ячейки оперативной памяти.

Аксиома — математическое правило или отношение. Аксиомы можно применять для описания операций над абстрактным типом данных.

Алгоритм BFS — см. *поиск в ширину*.

Алгоритм DFS — см. *поиск в ширину*.

Алгоритм поиска в ширину (BFS) — стратегия обхода графа, при которой сначала посещаются все вершины, смежные с вершиной v, а затем — все остальные. Таким образом, алгоритм не переходит к другим вершинам, пока не обойдет все вершины, смежные с вершиной v. См. также *поиск в глубину*.

Алгоритм распознавания — алгоритм, основанный на грамматике языка и определяющий, принадлежит ли заданная строка данному языку.

Алгоритм — пошаговое описание метода решения задачи за конечный отрезок времени.

Анализ алгоритмов — отрасль компьютерных наук, изучающая эффективность алгоритмов.

Анализ наилучшего варианта — определение минимального объема памяти, затрачиваемого данным алгоритмом на решение задачи, имеющей размер n. См. также *анализ среднего варианта* и *анализ наихудшего варианта*.

Анализ наихудшего варианта — определение максимального объема времени, необходимого данному алгоритму для решения задачи, имеющей размер n. См. также *анализ среднего варианта* и *анализ наилучшего варианта*.

Анализ среднего варианта — определение среднего объема времени, затрачиваемого данным алгоритмом на решение задачи, имеющей размер n. См. также *анализ наилучшего варианта* и *анализ наихудшего варианта*.

Аналитическая формула — не рекурсивное алгебраическое выражение.

Аргумент — см. *фактический аргумент* и *формальный аргумент*.

Аргумент, передаваемый по значению, — формальный аргумент, инициализируемый значением фактического аргумента. Любые изменения формального аргумента внутри функции не отражаются на соответствующем фактическом аргументе в вызывающем модуле. Если к типу аргумента не приписан знак &, аргумент по умолчанию считается передаваемым по значению.

Аргумент, передаваемый по ссылке, — формальный аргумент, представляющий собой фактический аргумент. Любое изменение, которое функция производит с аргументом, передаваемым по ссылке, изменяет значение фактического аргумента в вызывающем модуле. Аргумент, передаваемый по ссылке, указывается в объявлении функции с помощью символа &, приписанного к его типу.

АТД — см. *абстрактный тип данных*.

Атрибут — см. *данные-члены*.

Базис — см. *базовый случай*.

Базовый класс — класс, из которого выводится новый класс. Производный класс наследует члены базового класса. Синонимы: *родительский класс*, *суперкласс*.

Базовый случай — ситуация, в которой результат рекурсивного определения или индуктивного доказательства очевиден. Синонимы: *базис*, *вырожденный случай*.

Бинарное дерево поиска — бинарное дерево, в котором поисковый ключ в любом узле N больше, чем в любом узле левого поддерева узла N, и меньше, чем в любом узле правого поддерева узла N.

Бинарное дерево — множество узлов, разделенных на корень и два возможно пустых множества, представляющих собой бинарные деревья. Таким образом, каждый узел бинарного дерева имеет по крайней мере два дочерних узла, левый и правый.

Бинарный оператор — оператор, имеющий два операнда, например, оператор + в выражении 2 + 3. См. также *унарный оператор*.

Бинарный поиск — алгоритм поиска конкретного элемента в упорядоченной коллекции, основанный на повторяющемся делении коллекции пополам и определении, какая из половин содержит искомый элемент.

Бинарный файл — файл, элементы которого представлены в двоичном виде. Бинарный файл не содержит строк. Синонимы: *файл общего вида*, *нетекстовый файл*.

Блок `try-catch` — код на языке C++, предназначенный для реагирования на исключительную ситуацию. Разновидность обработчика исключительной ситуации.

Блок хэширования — структура, связанная с хэш-адресом, в которой может храниться несколько элементов. Массив таких блоков можно использовать в качестве таблицы хэширования для разрешения конфликтов.

Блок — группа записей в файле.

Братья — узлы дерева, имеющие общего родителя.

Буфер — ячейка или группа ячеек, предназначенных для временного хранения данных при их обработке или передаче. Буфер позволяет синхронизировать обмен данными между процессами, выполняющимися с разной скоростью.

Быстрая сортировка — алгоритм сортировки, который разбивает элементы массива с помощью опорного элемента p, порождая две аналогичные задачи меньшего размера: сортировать левую часть, элементы которой меньше величины p, и сортировать правую часть, элементы которой больше величины p.

Вершина стека — конец стека, на котором выполняются операции вставки, удаления и извлечения элементов.

Вершина — узел графа.

Вес пути — см. *стоимость пути*.

Вес ребра — числовая метка ребра во взвешенном графе.

Взвешенный граф — граф, ребра которого помечены числовыми значениями.

Виртуальная функция — функция-член базового класса, которую можно заместить в производном классе, т.е. переопределить. Тело виртуальной функции определяется в ходе выполнения программы. См. также *раннее связывание*, *позднее связывание*, *статический метод* и *таблица виртуальных методов*.

Виртуальный метод — см. *виртуальная функция*.

Внешнее событие — событие, определенное по входным данным в рамках событийно-ориентированного моделирования. См. также *внутреннее событие*.

Внешние методы — алгоритмы, предназначенные для работы с внешними файлами, поскольку данные не могут целиком поместиться в оперативной памяти.

Внешняя сортировка — алгоритм сортировки, использующийся для упорядочения набора данных, хранящегося на внешнем запоминающем устройстве. См. также *внутренняя сортировка*.

Внутреннее событие — событие, определенное в результате вычисления в ходе событийно-ориентированного моделирования. См. также *внешнее событие*.

Внутренний узел дерева — узел, не являющийся листом.

Внутренняя сортировка — алгоритм сортировки, для которого необходимо, чтобы упорядочиваемые данные были целиком записаны в оперативной памяти компьютера. См. также *внешняя сортировка*.

Возможность включения файла — свойство языка, позволяющее вставлять в указанное место программы содержимое файла до его компиляции. Обеспечивается в языке C++ директивой `#include`.

Вращение — операция, используемая для восстановления баланса красно-черного или AVL-дерева.

Время доступа к блоку — время, требуемое для чтения или записи блока данных, хранящихся в файле.

Время доступа — время, необходимое для получения доступа к конкретному элементу структуры данных, например, массива, связанного списка или файла.

Выделение динамической памяти — связывание ячейки памяти с переменной во время выполнения программы, а не на этапе компиляции. См. также *выделение статической памяти*.

Выделение памяти — см. *выделение динамической памяти* и *выделение статической памяти*.

Выделение статической памяти — размещение переменной в памяти на этапе компиляции, а не в ходе выполнения программы. См. также *выделение динамической памяти*.

Вырожденный случай — см. *базовый случай*.

Высота дерева — количество узлов на самом длинном пути от корня до листа.

Выталкивание — удаление элемента из стека.

Генерирование исключительной ситуации — сообщение о возникновении исключительных условий.

Глобальная переменная — переменная, область видимости которой распространяется на всю программу. См. также *локальная переменная*.

Глобальное пространство имен — набор идентификаторов, объявленных вне какого-либо пространства имен. Идентификаторы из глобального пространства имен доступны в любой точке программы.

Глубокая копия объекта — копия, включающая в себя структуры данных, на которые ссылаются члены объекта. См. также *поверхностная копия*.

Голова очереди — конец очереди, в котором выполняются операции удаления и извлечения.

Голова — см. *указатель на голову*.

Грамматика — правила, определяющие язык.

Граф — множество V, состоящее из вершин, или узлов, и множество E, состоящее из ребер, соединяющих эти вершины.

Данные-члены — часть структуры или класса, в которой хранятся данные конкретного типа.

Дважды связанный список — связанный список, узлы которого содержат два указателя: на предыдущий и следующий узлы.

Двойное хэширование — схема разрешения конфликтов, использующая две функции хэширования. Поиск свободной ячейки в таблице выполняется путем перебора всех n-х ячеек, начиная с ячейки, заданной первой функцией хэширования, а число n вычисляется второй функцией хэширования.

Двунаправленный итератор — итератор, способный перемещаться в обе стороны от текущего элемента контейнера.

Двухместный узел — узел дерева, содержащий один элемент данных и имеющий два дочерних узла. См. также *четырехместный узел* и *трехместный узел*.

Дерево общего вида — множество, состоящее из одного или нескольких узлов, разделенных на корень и подмножества, каждое из которых является поддеревом общего вида.

Дерево поиска — дерево, организация которого облегчает извлечение его элементов. См. также *AVL-дерево*, *бинарное дерево поиска*, *B-дерево степени m*, *красно-черное дерево*, *2-3 дерево* и *2-3-4 дерево*.

Дерево — связный неориентированный граф без циклов. См. также *бинарное дерево* и *дерево общего вида*.

Дерево, сбалансированное по высоте, — см. *сбалансированное бинарное дерево*.

Деструктор — метод, выполняющий все необходимое для удаления объекта.

Диагностическое утверждение — утверждение, описывающее состояние алгоритма или программы в определенный момент.

Диаграмма класса — диаграмма, созданная с помощью универсального языка моделирования для описания класса. Указывает имя класса, его члены и операции над ними.

Динамический объект — объект, размещенный в динамической памяти. Создается в ходе выполнения программы и существует вплоть до своего явного или неявного удаления. См. также *статический объект*.

Динамическое связывание — см. *позднее связывание*.

Длина пути — см. *стоимость пути*.

Дочерний клас — см. *производный класс*.

Дочерний узел узла *N* — узел, расположенный непосредственно под узлом N в дереве.

Друзья класса — класс или функция, не являющаяся членом класса, которые имеют доступ к закрытым и защищенным членам данного класса.

Жизненный цикл программного обеспечения — фазы разработки программного обеспечения: спецификация, проектирование, анализ рисков, верификация, кодирование, тестирование, уточнение, производство и эксплуатация.

Заголовок — механизм предоставления информации о содержании библиотеки, включая объявления функций, типов данных и констант. В текущей версии языка C++ заголовок представляет собой простую абстракцию, которую компилятор может преобразовать в имя файла или обработать как-то иначе. В старых версиях языка C++ и пользовательских библиотек заголовок представлял собой файл. Синонимы: *заголовочный файл*, *файл спецификаций*.

Заголовочный файл — см. *заголовок*.

Закрытое наследование — вид наследования, посредством которого открытые и защищенные члены базового класса становятся закрытыми членами производного класса.

Закрытый раздел — раздел класса, доступный только функциям-членам и друзьям класса.

Замещение — переопределение виртуальной функции-члена в производном классе. См. также *переопределение*.

Запись активации — запись, содержащая локальное окружение функции, образующееся в результате ее вызова.

Запись данных — элемент файла. Запись данных может быть как обычным числом, например, целым, так и структурой, например, записью о сотруднике. См. также *блок* и *запись*.

Запись — группа связанных друг с другом элементов, называемых полями, которые не обязательно имеют одинаковый тип. См. также *запись данных*.

Запрос по диапазону — операция, извлекающая из таблицы все элементы, ключ которых лежит в заданном диапазоне значений.

Заталкивание — добавление элемента в стек.

Защищенное наследование — вид наследования, посредством которого открытые и защищенные члены базового класса становятся защищенными членами производного класса.

Защищенный раздел — раздел класса, доступный для функций-членов производных классов.

Зондируемая последовательность — последовательность ячеек таблицы хэширования, которые проверяются схемой разрешения конфликтов.

Иерархия — отношение "родительский-дочерний" между узлами дерева.

Инвариант цикла — диагностическое утверждение, являющееся истинным до и после каждого выполнения цикла внутри алгоритма или программы.

Инвариант — диагностическое утверждение, которое должно всегда выполняться в конкретной точке алгоритма или программы.

Индекс — 1) целое число, позволяющее ссылаться на элемент массива; 2) другое название индексного файла.

Индексная запись — элемент индексного файла, ссылающийся на запись в соответствующем внешнем файле данных. Этот элемент содержит поисковый ключ и указатель.

Индексный файл — структура данных, элементы которой, называемые индексными записями, используются для обнаружения элементов, хранящихся во внешнем файле. Синоним: *индекс*.

Индуктивная гипотеза — см. *шаг индукции*.

Индуктивное доказательство — доказательство, использующее принцип математической индукции.

Индуктивное заключение — см. *шаг индукции*.

Индукция — см. *математическая индукция*.

Инкапсуляция — метод сокрытия информации путем объединения данных и операций в объекте.

Инфиксное выражение — алгебраическое выражение, в котором каждый бинарный оператор находится между двумя своими операндами. См. также *постфиксное выражение* и *префиксное выражение*.

Исключительная ситуация — необычное или исключительное событие, возникающее во время выполнения программы.

Исходная программа — программа, написанная на языке программирования и подлежащая компиляции. Например, программа, написанная на языке C++. Синоним: *исходный код*.

Итеративное решение — решение, использующее циклы.

Итератор — класс, взаимодействующий с другим классом, представляющим собой коллекцию объектов и обеспечивающий доступ к следующему или предыдущему элементу коллекции. Итератор позволяет перемещаться по объектам коллекции.

Итерация — 1) повторяющийся процесс; 2) один проход цикла.

Квадратичное зондирование — схема разрешения конфликтов, выполняющая поиск занятых ячеек, начиная с исходной ячейки, которую задает функция хэширования, и продолжая с шагом 1^2, 2^2, 3^2 и т.д.

Класс — конструкция языка C++, позволяющая определить новый тип данных.

Кластеризация — тенденция, проявляющаяся в стремлении элементов ассоциативного массива конденсироваться в группы, а не равномерно распределяться по ячейкам. Характерна для *линейного зондирования*, являющегося схемой разрешения конфликтов при хэшировании. Может приводить к увеличению времени поиска элементов.

Клиент — программа, модуль или абстрактный тип данных, использующие класс.

Ключ сортировки — часть записи, определяющая порядок всех записей в коллекции. Алгоритм сортировки использует ключ сортировки для установления заданного порядка в коллекции.

Ключ — 1) часть индексной записи, в которой хранится поисковый ключ записи, содержащейся во внешнем файле; 2) синоним поискового ключа.

Код — строки программы.

Кодирование — реализация алгоритма на языке программирования.

Кольцевой дважды связанный список — дважды связанный список, в котором указатель на узел, *предшествующий* первому узлу, ссылается на последний узел, а указатель на узел, *следующий* за последним, ссылается на первый узел.

Кольцевой связанный список — связанный список, в котором последний узел ссылается на первый.

Компилятор — программа, переводящая программу, написанную на языке высокого уровня, например, на языке C++, на машинный язык.

Константная функция-член — функция-член класса, возвращающая значение данного-члена. См. также *модифицирующая функция-член*.

Конструктор по умолчанию — конструктор без аргументов.

Конструктор — метод, инициализирующий новый экземпляр класса. См. также *конструктор по умолчанию*.

Контейнерный класс — класс, содержащий совокупность объектов.

Конфликт — ситуация, которая возникает, когда функция хэширования отображает два разных поисковых ключа в одну и ту же ячейку памяти.

Координация — степень взаимозависимости между функциями в программе.

Корень — единственный узел дерева, не имеющий родителя.

Коэффициент нагрузки — степень относительного заполнения таблицы хэширования, выраженная в долях от ее максимального размера.

Красно-черное дерево — представление 2-3-4 дерева в виде бинарного дерева, узлы которого содержат красные и черные указатели.

Кратчайший путь — путь между двумя заданными вершинами графа, имеющий наименьшую сумму весов ребер.

Куча — совершенное бинарное дерево, каждый узел которого содержит значение приоритета, большее или равное значениям приоритета своих дочерних узлов. Синоним: *максимальная куча*. См. также *минимальная куча*.

Левое поддерево узла *N* — левый дочерний узел узла *N* и его преемники.

Левый дочерний узел узла *N* — узел, расположенный непосредственно ниже и левее узла *N*.

Линейная реализация — реализация абстрактного типа данных с помощью массива или указателей.

Линейное зондирование — схема разрешения конфликтов, выполняющая последовательный поиск свободных ячеек в таблице хэширования, начиная с исходной ячейки, указанной с помощью функции хэширования.

Линейный связанный список — связанный список, не являющийся кольцевым.

Лист — узел дерева, не имеющий дочерних узлов.

Логиковременное моделирование — моделирование, в процессе которого время наступления события, например, прибытия или отбытия, вычисляется случайным образом и сравнивается с моделируемыми часами. См. также *событийно-ориентированное моделирование*.

Локальная переменная — переменная, объявленная внутри функции и доступная только внутри нее. См. также **глобальная переменная**.

Локальное окружение функции — локальные переменные, определенные в функции, копии значений фактических аргументов, адрес точки возврата в вызывающем модуле и возвращаемое значение.

Локальный идентификатор — идентификатор, область видимости которого ограничена границами блока, содержащего его объявление.

Максимальная куча — синоним *кучи*. См. также *минимальная куча*.

Массив — структура данных, содержащая фиксированное максимальное количество индексированных элементов одинакового типа.

Математическая индукция — метод доказательства утверждений, использующих натуральные числа. Начиная с базового случая доказательство сводится к проверке утверждения: "если свойство выполняется для произвольного натурального k, то оно имеет место и для числа $k+1$".

Машинный язык — язык, состоящий из основных инструкций, выполняемых компьютером непосредственно.

Метод блок-схем — систематический способ трассировки рекурсивных функций.

Метод — см. *функция-член*.

Минимальная куча — совершенное бинарное дерево, каждый узел которого содержит значение приоритета, меньшее или равное значениям приоритета своих дочерних узлов. См. также *максимальная куча*.

Минимальное остовное дерево — остовное дерево графа, сумма весов ребер которого является минимальной среди всех остовных деревьев данного графа.

Многократное индексирование — процесс, использующий несколько индексных файлов для индексации одного внешнего файла.

Множественное наследование — отношение между классами, посредством которого класс наследует свойства от одного или нескольких ранее определенных классов. См. также *производный класс*.

Моделирование — способ имитации поведения природных и искусственных систем. Как правило, целью моделирования является сбор статистических данных о производительности существующей системы или предсказание эффективности предложенной системы. Моделирование отражает долговременное усредненное поведение системы, а не предсказывает конкретные события.

Модифицирующая функция-член — функция-член класса, изменяющая значение поля. См. также **константная функция-член**.

Модуль — индивидуальный компонент программы, например, функция, группа функций или блок кода.

Модульная программа — программа, разделенная на изолированные компоненты, или модули, имеющие четкое предназначение и порядок взаимодействия.

Мультиграф — структура, похожая на граф, но допускающая дублирование ребер.

Надежное программирование — метод программирования, заключающийся в проверке и предотвращении ошибок, которые могут возникнуть в ходе выполнения программы.

Наследование — отношение между классами, посредством которого один класс наследует свойства ранее определенного класса. См. также *производный класс* и *множественное наследование*.

Неориентированный граф — граф, любые две вершины которого соединены ребрами, не имеющими ориентации. См. также *ориентированный граф*.

Несвязный граф — граф, не являющийся связным, т.е. граф, в котором существует хотя бы одна пара вершин, не соединенных ни одним путем.

Нетекстовый файл — см. *бинарный файл*.

Область видимости идентификатора — часть программы, в которой данный идентификатор имеет смысл.

Обозначение О-большое — обозначение, использующее прописную букву О для указания порядка алгоритма. Например, запись "O($f(n)$)" означает "порядок функции $f(n)$". См. также *порядок алгоритма*.

Обработчик исключительной ситуации — код, реагирующий на исключительную ситуацию при ее возникновении.

Обратный обход — обход бинарного дерева, при котором узел посещается после обхода обоих его поддеревьев. См. также *симметричный обход* и *прямой обход*.

Обход графа — процесс, начинающийся в вершине v и посещающий все вершины w, до которых существует путь из вершины v. В ходе обхода графа все вершины, независимо от начальной точки, посещаются тогда и только тогда, когда граф является связным.

Обход — операция, посещающая каждый элемент абстрактного типа данных или структуры.

Объект — экземпляр класса.

Объектная совместимость типов — свойство объектов, позволяющее использовать экземпляр производного класса вместо экземпляра базового класса, но не наоборот. Фактическим аргументом функции может быть наследник соответствующего формального аргумента.

Объектно-ориентированное программирование (ООП) — метод разработки программного обеспечения, рассматривающий программы как совокупность объектов, взаимодействующих друг с другом. ООП основано на трех фундаментальных принципах: инкапсуляция, наследование и полиморфизм.

Округление с избытком — результатом этой операции является ближайшее целое число, превышающее число x, например `⌈6.1⌉`= 7.

ООП — см. *объектно-ориентированное программирование*.

Оператор разрешения области видимости — оператор : : языка C++. При реализации любой функции-члена ее имя уточняется с помощью имени класса, за которым следует оператор разрешения области видимости. Это позволяет отличить данную функцию от других функций, которые могут иметь такое же имя.

Опорный элемент — основной элемент алгоритма. Например, в алгоритме быстрой сортировки массив разбивается на части относительно конкретного элемента, называемого *опорным*.

Орграф — см. ориентированный граф.

Ориентированное ребро — ребро ориентированного графа, т.е. ребро, имеющее направление.

Ориентированный граф — граф, ребра которого имеют направление. Называется также *орграфом*. См. также *неориентированный граф*.

Ориентированный путь — последовательность ориентированных ребер, начинающаяся в одной вершине и заканчивающаяся в другой вершине ориентированного графа. См. также *путь* и *простой путь*.

Остовное дерево BFS — остовное дерево, возникающее в ходе поиска в ширину при обходе вершин графа.

Остовное дерево алгоритма DFS — остовное дерево, возникающее в ходе поиска в глубину при обходе вершин графа.

Остовное дерево — подграф связного неориентированного графа *G*, содержащий все вершины графа G и достаточное количество ребер, чтобы образовать дерево. См. также *остовное дерево алгоритма BFS* и *остовное дерево алгоритма DFS*.

Отдельное связывание — схема разрешения конфликтов, использующая в качестве таблицы хэширования массив связанных списков, в котором *i*-й связанный список, или цепочка содержит все элементы, отображенные в ячейку *i*.

Откат — стратегия решения задач, в которой выход из тупика осуществляется с помощью выполнения шагов алгоритма в обратном порядке с последующим выполнением новой последовательности шагов.

Открытая адресация — категория схем разрешения конфликтов при хэшировании, в которых производится зондирование пустых, или открытых ячеек в таблице хэширования. См. также *двойное хэширование*, *линейное зондирование* и *квадратичное зондирование*.

Открытие — процесс подготовки файла для ввода или вывода и установки файлового курсора. Открытый файл находится в состоянии готовности к вводу/выводу.

Открытое наследование — вид наследования, посредством которого открытые и защищенные члены базового класса остаются открытыми и защищенными членами производного класса, соответственно.

Открытый раздел — раздел класса, доступный любому пользователю класса, включая функции-члены самого класса, а также функции-члены производных от него классов.

Отношение "подобен" — отношение между классами, посредством которого один класс реализуется на основе другого с помощью закрытого наследования. См. также *отношение "содержит"* и *отношение "является"*.

Отношение "содержит" — отношение между классами, посредством которого один класс содержит экземпляр другого класса.

Отношение "является" — отношение между классами, в котором один класс представляет собой разновидность другого. Отношение "является" реализуется с

помощью открытого наследования. См. также *отношение "подобен"* и *отношение "содержит"*.

Отношение включения — см. *отношение "содержит"*.

Оценка порядка — анализ объема времени, необходимого алгоритму, выраженный в виде функции, зависящей от размера задачи. См. также *порядок алгоритма*.

Очередь с двусторонним доступом — очередь, имеющая два конца. Этот абстрактный тип данных допускает вставку и удаление элементов с обоих концов.

Очередь с приоритетами — абстрактный тип данных, в котором порядок элементов зависит от их приоритета. Первым удаляется элемент, имеющий наивысший приоритет.

Очередь — абстрактный тип данных, из которого первым извлекается или удаляется элемент, вставленный туда раньше остальных. Это свойство называется *"первым вошел, первым вышел"*, или просто *принцип FIFO*. Элементы добавляются в конец очереди, а удаляются — из начала.

Палиндром — символьная строка, которая слева направо и справа налево читается одинаково, например "ротор".

Параметр — см. *фактический аргумент* и *формальный аргумент*.

Перегруженный оператор — оператор, имеющий несколько значений, каждое из которых зависит от контекста, в котором используется оператор.

Переменная указательного типа — переменная в языке C++, ссылающаяся на ячейку памяти. Синоним: *указатель*.

Переопределение — функция-член производного класса, не изменяющая виртуальную функцию-член базового класса и имеющая такое же объявление. См. также *замещение*.

Перехват — распознавание исключительной ситуации с целью ее дальнейшей обработки.

Период выполнения программы — время, во время которого выполняется программа. См. также *период компиляции*.

Период компилирования — время, на протяжении которого компилятор транслирует исходный текст программы в машинный код. См. также *период выполнения программы*.

Пирамидальная сортировка — алгоритм сортировки, который сначала преобразует массив в кучу, а затем удаляет из нее корень (наибольший элемент), меняя его местами с последним элементом кучи. В конце полученная полукуча преобразуется обратно в кучу.

Планарный граф — граф, который можно изобразить на плоскости так, чтобы ни одна пара ребер не пересекалась.

Побочный эффект — 1) изменение переменной, существующей вне функции и не передаваемой ей в качестве аргумента; 2) событие, не предусмотренное в модуле.

Поддерево узла N — дерево, содержащее дочерний узел узла N и его преемников.

Поддерево — любой узел дерева вместе со своими преемниками.

Подкласс — см. *производный класс*.

Позднее связывание — ассоциация переменной с ее типом в ходе выполнения программы. Синоним: *динамическое связывание*. См. также *раннее связывание*, *статический метод* и *виртуальная функция*.

Позиционно ориентированные абстрактные типы данных — абстрактный тип данных, операции которого зависят от позиций его элементов. См. *абстрактные типы данных, ориентированные на значение*.

Поиск в глубину (DFS) — стратегия обхода графа, в которой сначала выполняется как можно более глубокий обход вершин, а затем — откат. Иными словами, после посещения очередной вершины, алгоритм посещает, если это возможно, еще не посещенную смежную вершину. Когда алгоритм достигает вершины, у которой нет еще не посещенных смежных вершин, выполняется откат, а затем — посещение, если это возможно, еще не посещенной смежной вершины. См. также *поиск в ширину*.

Поиск — процесс выделения конкретного элемента из совокупности данных.

Поисковый ключ — часть записи, идентифицирующая ее среди совокупности записей. Алгоритм поиска использует поисковый ключ для обнаружения записи внутри совокупности. Синоним: *ключ*.

Поле данных — см. *данные-члены*.

Поле — компонент записи.

Полиморфизм — способность связывать имя переменной с разными экземплярами родственных классов, производных от одного базового класса, во время выполнения программы.

Полное бинарное дерево — бинарное дерево высоты h, не имеющее недостающих узлов. Все листья расположены на уровне h, причем все остальные узлы имеют по два дочерних узла.

Полностью сбалансированное бинарное дерево — бинарное дерево, в котором левое и правое поддеревья каждого узла имеют одинаковую высоту.

Полный перебор — стратегия поиска, в ходе которого отсутствие элемента обнаруживается лишь после перебора всех элементов набора.

Полукуча — совершенное бинарное дерево, в котором левое и правое поддеревья корня являются кучами.

Пользователь — человек, использующий программу.

Пользовательский интерфейс — часть программы, обеспечивающая ввод данных и управление работой программы.

Поразрядная сортировка — алгоритм сортировки, обрабатывающий каждый элемент как символьную строку и быстро организующий данные в группы в соответствии с i-м символом каждого элемента.

Порядок алгоритма — объем времени, необходимый алгоритму для решения задачи, выраженный в виде функции, зависящей от размера задачи. Алгоритм A имеет порядок $f(n)$, если существуют константы K и n_0, такие что алгоритму A требуется не более чем $k * f(n)$ единиц времени для решения задачи размера $n \geq n_0$. См. также *обозначение О-большое*.

Посещение — обработка элемента при обходе абстрактного типа данных или структуры.

Последовательность Фибоначчи — последовательность целых чисел 1, 1, 2, 3, 5, ..., определяемых по рекуррентному соотношению.

Последовательный доступ — процесс обращения к элементу структуры данных, для которого необходимо перебрать все предшествующие элементы. См. также *прямой доступ*.

Последовательный поиск — алгоритм, обнаруживающий заданный элемент внутри совокупности путем последовательного перебора, начиная с первого элемента.

Постусловие — формулировка условий, выполняющихся в конце работы модуля.

Постфиксное выражение — алгебраическое выражение, в котором каждый бинарный оператор указывается после своих операндов. См. также *инфиксное выражение* и *префиксное выражение*.

Поток данных — поток данных между модулями.

Правое поддерево узла *N* — правый дочерний узел узла *N* и его преемники.

Правый дочерний узел узла *N* — узел дерева, лежащий непосредственно ниже и правее узла *N*.

Правый дрейф — 1) смещение начала очереди к концу массива; 2) смещение к правому полю начала вложенных блоков в программе на языке C++.

Предусловие — формулировка условий, которые должны выполняться в начале модуля, для того чтобы он работал правильно.

Предшественник — 1) в связанном списке предшественником узла *x* является узел, ссылающийся на него; 2) в ориентированном графе вершина *x* называется предшественником вершины *y*, если существует ориентированное ребро из вершины *x* в вершину *y*, т.е. если вершина *y* является смежной с вершиной *x*. См. также *преемник*.

Преемник узла *N* — узел, лежащий на пути от узла *N* к листу дерева.

Преемник узла *N* — узел, находящийся на пути от корня к узлу *N*.

Преемник — 1) в связанном списке преемником узла *x* называется узел, на который он ссылается; 2) в ориентированном графе вершина *y* называется преемником вершины *x*, если существует ориентированное ребро из вершины *x* в вершину *y*, т.е вершина *y* является смежной с вершиной *x*. См. также *предшественник*.

Префиксное выражение — алгебраическое выражение, в котором каждый бинарный оператор указывается перед своими операндами. См. также *инфиксное выражение* и *постфиксное выражение*.

Принцип "первым вошел — первым вышел" (FIFO) — свойство очереди, заключающееся в том, что операции удаления и извлечения применяются к элементу, который был вставлен раньше остальных (первым). См. также *принцип "последним вошел — первым вышел"*.

Принцип "последним вошел — первым вышел" — свойство стека, заключающееся в том, что операции удаления и извлечения применяются к элементу, вставленному позднее остальных (последнему). См. также принцип *"первым вошел — первым вышел"*.

Принцип FIFO — см. *принцип "первым вошел — первым вышел"*.

Принцип LIFO — см. *принцип "последним вошел — первым вышел"*.

Приоритет — величина, приписанная элементам очереди с приоритетами, для указания очередности их удаления.

Проектирование сверху вниз — процесс последовательного уточнения деталей решения, приводящий к созданию независимых модулей.

Производный класс — класс, наследующий члены другого класса, называемого базовым. Синонимы: *дочерний класс*, *подкласс*. См. также *базовый класс*, *наследование* и *множественное наследование*.

Простой путь — путь в графе, не проходящий ни через одну вершину более одного раза. См. также *ориентированный путь*.

Простой тип данных — тип данных, не являющийся агрегатным, например `int` и `double`.

Простой цикл — цикл в графе, не проходящий ни через одну вершину более одного раза.

Пространство имен — механизм, предусмотренный в языке C++ для логической группировки объявлений и определений в общей декларативной области. Каждый идентификатор в пространстве имен имеет единственное значение.

Процедурная абстракция — см. *функциональная абстракция*.

Прямой доступ — процесс, обеспечивающий доступ к любому элементу структуры данных по его позиции, для которого не нужно предварительно получать доступ к остальным элементам структуры.

Прямой обход — обход бинарного дерева, в котором узел посещается раньше своих поддеревьев. См. также *симметричный обход* и *обратный обход*.

Пустая строка — строка нулевой длины.

Пустое дерево — дерево, не имеющее узлов.

Путь — последовательность ребер в графе, начинающаяся с одной вершины и заканчивающаяся в другой. Поскольку дерево является разновидностью графа, можно говорить о пути в дереве. См. также *ориентированный путь* и *простой путь*.

Разделение — разделение структуры данных на части, например массива на сегменты.

Разделяй и властвуй — стратегия, которая разделяет исходную задачу на множество более мелких задач, каждая из которых решается отдельно.

Разработка программного обеспечения — отрасль компьютерных наук, изучающая методы облегчения процесса разработки компьютерных программ.

Разрешение конфликта — процесс, с помощью которого элементы с разными поисковыми ключами, вступившие в конфликт, распределяются в разные ячейки таблицы хэширования. См. также *блоки хэширования*, *цепочка*, *кластеризация*, *двойное хэширование*, *свертывание*, *линейное зондирование*, *открытая адресация*, *последовательность зондирования*, *квадратичное зондирование* и *отдельное связывание*.

Раннее связывание — ассоциация переменной со своим типом во время компиляции. Синоним: *статическое связывание*. См. также *позднее связывание*, *статический метод* и *виртуальная функция*.

Распределитель памяти — объект, управляющий распределением памяти для контейнера.

Расширяемый класс — класс, позволяющий добавлять новые возможности в свои производные классы без вмешательства в реализацию базового класса. Расширяемые классы должны содержать виртуальные функции.

Реализация — 1) процесс кодирования алгоритма; 2) использование структуры данных для воплощения абстрактного типа данных.

Реализация с помощью массива — реализация абстрактного типа данных, в которой для хранения значений используется массив.

Реализация с помощью указателей — реализация абстрактного типа данных, использующая указатели для организации его элементов.

Ребро — связь между двумя вершинами графа.

Рекуррентное отношение — математическая формула, выражающая значения элементов последовательности через значения предыдущих элементов.

Рекурсивный вызов — вызов, при котором функция вызывает сама себя.

Рекурсия — способ решения исходной задачи с помощью решения последовательности таких же задач, но имеющих меньший размер.

Решение — алгоритмы и способ хранения данных, предназначенные для решения конкретной задачи.

Родитель узла *N* — узел дерева, расположенный непосредственно над узлом *N*.

Родительский класс — см. *базовый класс*.

Сбалансированное бинарное дерево — бинарное дерево, в котором высота левого и правого поддерева любого узла отличается не больше чем на 1. Оно называется также деревом, сбалансированным по высоте.

Свертка — метод хэширования, разбивающий поисковый ключ на две части и комбинирующий их между собой для создания нового адреса.

Свободный список — список доступных узлов в реализации абстрактного типа данных или структуры в виде массива.

Связанный список — список элементов или узлов, связанных между собой так, что каждый элемент ссылается на следующий.

Связный граф — граф, в котором существует путь, соединяющий любую пару вершин.

Связный компонент — подмножество вершин несвязного графа, с которых начинается его обход.

Связывание — ассоциация переменной с адресом ячейки и типом данных, который она может содержать.

Связывание — см. *отдельное связывание*.

Симметричная матрица — матрица A размером n x n, элементы которой удовлетворяют отношению $A_{ij}=A_{ji}$.

Симметричный обход — обход бинарного дерева поиска, обрабатывающий (посещающий) узел после обхода его левого поддерева, но перед обходом его правого поддерева. См. также *обратный обход* и *прямой обход*.

Симметричный преемник ключа *x* — поисковый ключ узла в дереве поиска, который алгоритм симметричного обхода посещает сразу после узла, содержащего ключ x.

Симметричный преемник узла *N* — симметричный преемник поискового ключа узла N, являющийся крайним левым узлом в правом поддереве узла N.

Слабо связанные модули — несколько модулей, не зависящих друг от друга. См. также *связь*.

Словарь — см. *таблица*.

Случайный доступ — см. *прямой доступ*.

Смежные вершины — две вершины графа, соединенные ребром. В ориентированном графе вершина y является смежной с вершиной x, если существует ориентированное ребро из вершины x в вершину y.

Событие — например, прибытие или отбытие, в событийно-ориентированном моделировании. См. также *внешнее событие* и *внутреннее событие*.

Событийно-ориентированное моделирование — моделирование, использующее события, генерируемые с помощью математической модели, основанной на статистике и вероятности. Время наступления события либо считывается из входного потока, либо вычисляется на основе времени наступления других событий. Поскольку между двумя моментами никакие действия не предпринимаются, событийно-ориентированное моделирование выполняет переход от одного момента непосредственно к следующему. См. также *логиковременное моделирование*.

Совершенная функция хэширования — идеальная функция хэширования, отображающая каждый поисковый ключ в отдельную ячейку таблицы хэширова-

ния. Совершенная функция хэширования существует, если все возможные поисковые ключи заранее известны.

Совершенное бинарное дерево — бинарное дерево высоты h, являющееся полным вплоть до высоты h-1, последний уровень которого заполнен слева направо.

Совершенный граф — граф, в котором каждая пара вершин соединена ребрами.

Совместимость типов — см. *объектная совместимость типов*.

Сокрытие информации — процесс, позволяющий скрыть детали реализации внутри модуля, сделав их недоступными для сущностей, находящихся вне модуля.

Сообщение — запрос, поступивший в виде вызова функции, на выполнение заданной операции указанным объектом.

Сортировка методом вставок — алгоритм сортировки, проверяющий элементы по одному и вставляющий их в соответствующую позицию.

Сортировка методом выбора — алгоритм сортировки, в ходе которого из упорядочиваемой совокупности последовательно выбираются и вставляются на правильные места наибольший и наименьший элементы.

Сортировка методом пузырька — алгоритм сортировки, сравнивающий соседние элементы и меняющий их местами, если они нарушают заданный порядок. Сравнивая первый и второй элементы, второй и третий и т.д., алгоритм перемещает наибольший элемент в конец массива. Повторение этого процесса приводит к упорядочению всего массива в возрастающем порядке.

Сортировка слиянием — алгоритм сортировки, разделяющий массив на две части, сортирующий каждую по отдельности, а затем объединяющий упорядоченные части в один упорядоченный массив. Сортировку слиянием можно адаптировать для упорядочения внешнего файла.

Сортировка — процесс организации совокупности данных в порядке возрастания или убывания. См. также *внешняя сортировка* и *внутренняя сортировка*.

Список смежности — состоит из n связанных списков, реализующих граф, состоящий из n вершин с номерами 0, 1, ..., n-1, так что элемент *graph[i][j]* равен 1 тогда и только тогда, когда существует ребро, соединяющее вершину i с вершиной j.

Список событий — абстрактный тип данных, применяемый в событийно-ориентированном моделировании для отслеживания событий прибытия и отбытия, которые еще не наступили.

Список — абстрактный тип данных, элементы которого перечисляются по номерам их позиций. См. также *упорядоченный список*.

Стандартная библиотека шаблонов (STL) — библиотека, содержащая шаблонные классы для большинства широко используемых абстрактных типов данных, например, списков, стеков и очередей. Кроме того, библиотека содержит шаблонные функции для большинства алгоритмов, например для сортировки.

Статический метод — метод, тело которого определяется (связывается с объектом) на этапе компиляции. См. также *раннее связывание*, *позднее связывание* и *виртуальная функция*.

Статический объект — объект, размещенный в статической памяти. Эта память выделяется на этапе компиляции и остается в его распоряжении до завершения работы программы. См. также *динамический объект*.

Статическое связывание — см. *раннее связывание*.

Стек — абстрактный тип данных, в котором первым удаляется или извлекается элемент, вставленный позже всех. Это свойство называется *"последним вошел,*

первым вышел", или просто *принцип LIFO*. Вставка элементов стека производится на его вершине.

Стоимость остовного дерева — сумма весов ребер в остовном дереве взвешенного графа.

Стоимость программы — факторы, например, компьютерные ресурсы (длительность вычислений и объем памяти), потребляемые программой, сложности, возникающие при ее эксплуатации, а также последствия, возникающие в результате неправильной работы программы.

Стоимость пути — сумма весов ребер, лежащих на пути внутри взвешенного графа. Называется также *весом* или *длиной пути*.

Строка — последовательность символов. В языке C++ строка является объектом, имеющим тип `string`. В языке C строка является массивом, завершающимся нулевым символом `/0`.

Структура данных — конструкция, определенная в языке программирования и предназначенная для хранения набора данных.

Структурная диаграмма — иллюстрация иерархии модулей, предназначенных для решения конкретной задачи.

Суперкласс — см. *базовый класс*.

Сцепление — степень взаимозависимости между разными частями модуля.

Таблица виртуальных методов (ТВМ) — таблица, существующая для любого класса, в котором определена виртуальная функция. Для каждой виртуальной функции, существующей в объекте, соответствующая таблица виртуальных методов содержит указатель на фактические инструкции, реализованные в функции. Этот указатель задается конструктором во время выполнения программы.

Таблица хэширования — массив, содержащий элементы таблицы, размещенные с помощью функции хэширования.

Таблица — абстрактный тип данных, элементы которого хранятся и извлекаются в соответствии со значением поискового ключа. Синоним: *словарь*.

ТВМ — см. *таблица виртуальных методов*.

Текстовый файл — файл, состоящий из символов, организованных в строки.

Топологическая сортировка — процесс упорядочения вершин ориентированного графа без циклов в топологическом порядке.

Топологический порядок — список вершин ориентированного графа без циклов, в котором вершина x предшествует вершине y, если существует ориентированное ребро из вершины x в вершину y. Как правило, топологический порядок не единственен.

Трехместный узел — узел дерева, содержащий два элемента данных и имеющий три дочерних узла. См. также *четырехместный узел* и *двухместный узел*.

Узел — элемент связанного списка, графа или дерева, который обычно содержит данные и указатель на следующий элемент структуры данных.

Узкоспециализированный модуль — модуль, выполняющий одну точно поставленную задачу. См. также *координация*.

Указатель на голову — указатель на первый узел связанного списка. Синоним: *голова*.

Указатель на хвост — указатель на последний узел связанного списка. Синоним: *хвост*.

Указатель — 1) переменная указательного типа в языке C++; 2) элемент, ссылающийся на ячейку памяти (как правило); 3) индикатор элемента структуры,

например, целое число (редко). Например, индексная запись, ссылающаяся на запись данных во внешнем файле, содержит такой индикатор, а именно: номер блока, содержащего данную запись данных.

Унарный оператор — оператор, требующий наличия только одного операнда, например, оператор - в выражении -5. См. также *бинарный оператор*.

Универсальный язык моделирования (UML) — язык моделирования, используемый для описания процесса объектно-ориентированного проектирования. Язык UML позволяет создавать диаграммы и текстовые описания. См. также *диаграмма класса*.

Упорядоченный отрезок — упорядоченные данные, представляющие собой часть внешней сортировки.

Упорядоченный список — абстрактный тип данных, элементы которого записаны в определенном порядке, а их извлечение производится с помощью указания конкретной позиции. См. также *список*.

Уровень узла — корень находится на первом уровне. Если узел не является корнем, то его уровень на единицу превышает уровень его родительского узла.

Утечка памяти — потеря динамической памяти, на которую не ссылается ни один указатель.

Уточнение — в языке C++ используется для ссылки на элемент `a` объекта `b` с помощью записи `b.a`. В этом случае говорят, что "объект `b` уточняет элемент `a`" или "элемент `a` уточняется объектом `b`".

Файл общего вида — см. *бинарный файл*.

Файл последовательного доступа — файл, элементы которого просматриваются последовательно, т.е. для обращения к данным, записанным на указанной позиции, необходимо сначала переместить файловый курсор через все данные, предшествующие заданной позиции.

Файл прямого доступа — файл, элементы которого доступны по позиции и не требуют получения доступа к остальным записям. См. также *последовательный доступ*.

Файл реализации — файл, содержащий определение каждой функции, объявленной в соответствующем заголовке.

Файл спецификации — см. *заголовок*.

Файл — структура данных, содержащая последовательность компонентов одинакового типа. См. также *бинарный файл*, *индексный файл* и *текстовый файл*.

Файловая переменная — идентификатор, обозначающий имя файла.

Файловый компонент — невидимая часть данных, хранящихся в файле.

Файловый курсор — маркер, обозначающий текущую позицию в файле.

Фактический аргумент — переменная или выражение, передаваемое функции. Фактический аргумент задается при вызове функции и должен соответствовать формальному аргументу, указанному в списке объявлений функции. См. также *формальный аргумент*, *аргумент, передаваемый по ссылке*, и *аргумент, передаваемый по значению*.

Фиксированный размер — характеристика структуры данных, память для которой выделяется на этапе компиляции и не может изменять свои размеры в ходе выполнения программы. См. также *выделение статической памяти*.

Фиктивный головной узел — первый узел связанного списка, который не содержит никаких данных, но всегда существует. Элемент, находящийся на первой позиции списка, фактически хранится во втором узле.

Формальный аргумент — идентификатор, указанный в списке объявлений функции и соответствующий фактическому аргументу, который вызывающий модуль передает в функцию. См. также *фактический аргумент, аргумент, передаваемый по ссылке, и аргумент, передаваемый по значению.*

Функциональная абстракция — принцип разработки программ, позволяющий отделить цель и применение модуля от его реализации. Синоним: *процедурная абстракция.* См. также *абстракция данных.*

Функция роста — функция, зависящая от размера задачи, используется для указания порядка алгоритма.

Функция хэширования — функция, отображающая поисковый ключ элемента таблицы в ячейку, предназначенную для этого элемента.

Функция, имеющая значение, — функция, возвращающая значение. См. также *void-функция.*

Функция-член — функция, являющаяся членом класса. Синоним: *метод.*

Хвост очереди — конец очереди, в который вставляются элементы. Синоним: *конец очереди.*

Хвостовая рекурсия — разновидность рекурсии, при которой рекурсивный вызов применяется к последнему полученному результату.

Хэширование — метод, позволяющий получить доступ к элементу таблицы за почти постоянное время, независимо от того, где он находится, с помощью функции хэширования и схемы разрешения конфликтов.

Цепочка — связанный список, используемый при *отдельном связывании* для разрешения конфликтов, возникающих при хэшировании.

Цепь -- особый цикл, проходящий через каждую вершину (или ребро) графа только один раз.

Цикл — путь, начинающийся и заканчивающийся в одной и той же вершине графа. См. также *цепь* и *простой цикл.*

Четырехместный узел — узел дерева, содержащий три элемента данных и четыре дочерних узла. См. также *трех-* и *двухместный узел.*

Чисто виртуальная функция — виртуальная функция с неопределенным телом. В определении класса записывается как `virtual` *прототип* = *0.*

Член — компонент структуры или класса, являющийся либо данными, либо функцией. См. также *данные-члены* и *функция-член.*

Шаблон — см. *шаблонный класс.*

Шаблонный класс — спецификация класса с помощью шаблонных параметров, задающих тип данных.

Шаг индукции — шаг индуктивного доказательства, начинающийся с индуктивной гипотезы (“если утверждение $P(k)$ истинно для любого $k \geq 0$”) и демонстрирующий индуктивное заключение (“то утверждение $P(k+1)$ истинно”).

Экземпляр — объект, являющийся результатом объявления переменной конкретного класса или вызова оператора `new` с указателем на класс.

Язык — множество строк, подчиняющихся правилам грамматики.

Ответы на вопросы для самопроверки

Глава 1

1. $0 \leq index \leq n$ и $sum = item[0] + ... + item[index]$.
2. Спецификации включают в себя определения типов, аргументы, а также пред- и постусловия.

```
sum(in anArray: arrayType, in n:integer):elementType
// Вычисляет сумму первых пяти положительных элементов
// массива anArray.
// Предусловие: массив anArray состоит из n элементов, n >= 5,
// по крайней мере 5 элементов массива являются положительными.
// Постусловие: возвращает сумму первых пяти положительных
// элементов массива anArray; массив остается неизменным.
```

Другое решение:

```
computeSum(in anArray: arrayType, in n:integer,
           out sum:elementType, out success:boolean)
// Вычисляет сумму первых пяти положительных элементов
// массива anArray.
// Предусловие: массив anArray состоит из n элементов.
// Постусловие: если по крайней мере 5 элементов массива
// являются положительными, то параметр sum равен сумме первых
// пяти положительных элементов, а параметр success имеет значение
// true. В противном случае параметр sum равен 0, а параметр
// success имеет значение false; массив остается неизменным.
```

Глава 2

1. Произведение n чисел определено через произведение n-1 числа, которое представляет собой задачу меньшего размера. Если число n равно 1, произведение хранится в элементе *anArray[0]*. Это — базис рекурсии. Поскольку $n \geq 1$ и при каждом рекурсивном вызове увеличивается на 1, базис достигается.
2. Листинг функции *computeProduct* имеет следующий вид.

```
void computeProduct(const double anArray[],
                    int n, double &product)
{
   if (n == 1)
     product = anArray[0];
   else
   {
     computeProduct(anArray, n-1, product);
     product = anArray[n-1] * product;
   } // Конец оператора if
} // Конец функции computeProduct
```

3. Листинг функции *countDown* имеет следующий вид.

```
void countDown(int n)
// Предусловие: n > 0.
// Постусловие: выводит на экран числа n, n - 1, ... , 1.
{
   if (n > 0)
   {
      cout << n << endl;
      countDown(n-1);
   } // Конец оператора if
} // Конец функции countDown
```

4. Листинг функции *product* имеет следующий вид.

```
double product(const double anArray[],int first, int last)
// Предусловие: аргумент anArray[first..last] является
// массивом действительных чисел, причем first <= last.
// Постусловие: возвращает произведение чисел, хранящихся
// в массиве anArray[first..last].
{
   if(first == last)
      return anArray[first];
   else
      return anArray[last] * product(anArray, first, last-1);
} // Конец функции product
```

5. *writeBackward*, *binarySearch*, *kSmall* и функция *countdown* из задания 3.

6. *c(4, 2) = 6*

7. Три рекурсивных вызова приводят к следующим перемещениям: со стержня *A* — на стержень *C*, со стержня *A* — на стержень *B* и со стержня *C* — на стержень *B*.

Глава 3

1. Стена является аллегорией абстракции и модульности. Модули должны быть максимально независимыми: стены предотвращают взаимное пересечение разных частей программы и модулей. Контракт — это спецификация модуля, который скрыт за стенами. Контракт образует щели в стене. Он определяет способ доступа к модулю и возвращаемые им результаты. Контракт не описывает внутреннее устройство модуля.

   Эта концепция побуждает разделять программу на небольшие части и сосредотачивать внимание на том, что они делают, а не на том, как они выполняют свою задачу.

2. Псевдокод функции *swap* имеет следующий вид.

```
swap(inout aList:List, in i:integer, in j:integer)
// Меняем местами i-й и j-й элементы списка aList.

   // Копируем i-й и j-й элементы
   aList.retrieve(i, ithItem, success)
   aList.retrieve(j, jthItem, success)

   // Заменяем i-й элемент j-м
   aList.remove(i, success)
```

```
aList.insert(i, jthItem, success)

// Заменяем j-й элемент i-м
aList.remove(j, success)
aList.insert(j, ithItem, success)
```

Порядок операций очень важен, поскольку при удалении элемента операция *remove* перенумеровывает оставшиеся элементы. Если существование *i*-го и *j*-го элементов не гарантировано, после каждой операции следует проверять значение переменной *success*.

3. Молоко, яйца, масло.

4. Опишите операции *createList*, *destroyList*, *isEmpty* и *getLength*, как если бы они были операциями над абстрактным списком.

```
+insert(in newItem:ListItemType, out success:boolean)
// Вставляет элемент newItem в конец списка. Флаг success
// позволяет определить, успешно или нет выполнена вставка.

+remove(out success:boolean)
// Удаляет элемент из конца списка. Флаг success позволяет
// определить, успешно или нет выполнено удаление.

+retrieve(out dataItem:ListItemType, out success:boolean)
// Присваивает параметру dataItem элемент, стоящий
// в конце списка. После операции список остается неизменным.
// Флаг success позволяет определить, успешно или нет
// выполнено удаление.
```

5. Пред- и постусловия операций над абстрактным упорядоченным списком таковы.

```
+createSortedList()
// Предусловие: нет.
// Постусловие: создан пустой упорядоченный список.

+destroySortedList()
// Предусловие: нет.
// Постусловие: список уничтожен.

+sortedIsEmpty():boolean {query}
// Предусловие: нет.
// Постусловие: если список пуст, возвращает значение true:
// в противном случае возвращает значение false.

+sortedGetLength():integer {query}
// Предусловие: нет.
// Постусловие: возвращает количество элементов,
// содержащихся в упорядоченном списке.

+sortedInsert(in newItem:ListItemType, out success:boolean)
// Предусловие: в список вставляется элемент newItem.
// Постусловие: элемент newItem вставляется в соответствующее
// место упорядоченного списка, а значение параметра success
// равно true. Если вставка завершилась неудачно, значение
// параметра success равно false, а список остается неизменным.
```

```
+sortedRemove(in anItem:ListItemType, out success:boolean)
// Предусловие: элемент anItem подлежит удалению.
// Постусловие: если элемент anItem содержался в списке,
// он удален, а значение параметра success равно true.
// в противном случае значение параметра success равно false.

+sortedRetrieve(in index:integer, out dataItem:ListItemType,
                out success:boolean) {query}
// Предусловие: параметр index задает номер искомого элемента
// в списке
// Постусловие: если 1 <= index <= sortedGetLength(),
// параметру dataItem присвоено значение элемента с номером
// index, а параметр success равен true. В противном случае
// параметр success равен false.

+locatePosition(in anItem:ListItemType, out position:integer,
                out isPresent:boolean)
// Предусловие: элемент anItem — искомый.
// Постусловие: если элемент anItem содержится в списке,
// параметру position присвоен номер элемента, а значение
// параметра isPresent равно true. В противном случае,
// если элемента anItem в списке нет, параметр position
// задает его возможное положение, а параметр isPresent
// равен false. Элемент anItem и список остаются неизменными.
```

6. Псевдокод функции *sortList* имеет следующий вид.

```
sortList(in aList:List, out aSortedList:SortedList)
// Создает упорядоченный список aSortedList из элементов
// списка aList.

   aSortedList.createSortedList()
   for (i = 1 to aList.getLength())
   {
      aList.retrieve(i, item, success)
      aSortedList.sortedInsert(item, success)
   }
```

7. Дублирование значений, содержащихся в абстрактном списке, допускается и не влияет на его спецификацию, поскольку все операции являются позиционно-ориентированными. Однако для упорядоченного списка спецификацию придется пересмотреть. Можно либо запретить вставку дубликатов, либо позволить ее. Если вставка дубликатов допускается, нужно решить, где их следует размещать, удалять ли все вхождения элемента или только первое, а также какой дубликат извлекать из списка.

Глава 4

1. Значения указателей при трассировке программы обозначены символами *xxxx*.

   xxxx xxxx

   7 11 11

   7 18 18

   7 4 18

2. Ответы таковы.

   2.1. Нет. Если указатель *cur* ссылается на последний узел, а указатель *prev* — на предпоследний, то оператор `prev->next = cur->next` присвоит указателю *next* в предпоследнем узле значение *NULL*.

   2.2. Этот случай аналогичен удалению первого узла.

   2.3. Для обнаружения последнего узла придется пройти по всему списку, поэтому поиск последнего узла более трудоемок, чем поиск первого узла. Однако после обнаружения последний элемент удалить легче, чем первый, поскольку для этого не нужно изменять указатель *head*.

3. Соответствующие фрагменты имеют следующий вид.

   3.1. Первый фрагмент.

```
head = new node;
head->item = 'J';
head->next = NULL;
p = new node;
p->item = 'E';
p->next = head;
head = p;
p = new node;
p->item = 'B';
p->next = head;
head = p;
```

   3.2. Второй фрагмент.

```
head = new node;
head->item = 'B';
p = new node;
p->item = 'E';
head->next = p;
q = new node;
q->item = 'J';
q->next = NULL;
p->next = q;
```

4. Ответы таковы.

   4.1. Первый фрагмент.

```
prev->next = cur->next;
cur->next = NULL;
delete cur;
cur = NULL;
```

   4.2. Второй фрагмент.

```
head = cur->next;
cur->next = NULL;
delete cur;
cur = NULL;
```

   4.3. Третий фрагмент.

```
p = new node;
p->item = 'A';
p->next = head;
head = P;
```

4.4. Указатель *head* ссылается на узел, содержащий символ *'A'*. Этот узел, в свою очередь, ссылается на узел, содержащий символ *'J'*. Указатель *next* в последнем узле равен константе *NULL*.

5. Ответ:

```
void displayNodeI(Node *head, int i)
{
   Node *cur = head;
   for (int count = 1; count < i; ++count)
      cur = cur->next;
      cout << cur->item << endl;
} // Êîíåö ôóíêöèè displayNodeI
```

6. *i*, игнорируя присваивание переменной *count* в операторе *for*.

7. Код функции *ithItem*:

```
int ithItem(Node *head, int i)
{
   if (i == 1)
      return head->item;
   else
      return ithItem(head->next, i-1);
} // Êîíåö ôóíêöèè ithItem
```

8. Результаты трассировки.

```
writeBackward2(указатель на 'B') // Исходный вызов
writeBackward2(указатель на 'E')
writeBackward2(указатель на 'J')
writeBackward2(NULL)
write J
write E
write B
```

9. Ответы.

9.1. После каждого удаления элементы перенумеровываются так, что исходный второй элемент оказывается первым. Таким образом, первый элемент можно удалить снова.

9.2. Да.

9.3. Можно, поскольку при удалении последнего элемента список не перенумеровывается.

9.4. Цикл

```
for (int position = 1- position <= getLength();
     ++position)
remove(1);
```

некорректен, поскольку значение *getLength()* изменяется при удалении элементов из списка. Если вызов *remove(1)* заменить вызовом *remove(position)*, цикл по-прежнему останется неверным, поскольку, кроме упомянутого изменения значения *getLength()*, после первого удаления элементы окажутся перенумерованными, так что новый первый элемент удален не будет.

10. Модифицированный деструктор имеет следующий вид.

```
List::~List()
{
```

```
    int len = getLength();
    // Повторное удаление первого элемента связанного списка
    for (int position = 1; position <= len; ++position)
    {
       Node *cur = head;
       head = head->next;
       cur->next = NULL;
       delete cur;
    } // Конец оператора for
} // Конец деструктора
```

Глава 5

1. Положения ферзя задаются парой координат (*строка*, *столбец*).

   Решение 1: (2, 1), (4, 2), (1, 3), (3, 4).

   Решение 2: (3, 1), (1, 2), (4, 3), (2, 4).
2. *-*/abc*+def*
3. *ab*c-d/ef-+*
4. *(a-b/c(c=d*e))-f*
5. Нет.
6. *<T>=$|cc<T>d*

Глава 6

1. D, C, B, A
2. *stack1*: 1 4 5; *stack2*: 3 6 9 (элементы перечислены снизу вверх)
3. Если максимальная длина строки известна заранее и средняя длина строки не превышает максимального значения, следует использовать массив. Очевидно, если максимальная длина строки заранее неизвестна, нужно применять связанный список. Кроме того, если максимальная длина строки, например, равна 300, а средняя длина строки равна 30, то связанный список займет меньше памяти, чем массив.
4. Если спецификации операций над стеком в обеих реализациях идентичны, в самой программе изменений не будет. В обеих версиях операции над стеком за пределами "стены" будут выполняться одинаково. Разумеется определение реализации стека в виде массива следует заменить определением реализации стека в виде связанного списка.
5. Результаты трассировки таковы.

   5.1. После закрытия последней фигурной скобки стек оказывается пустым. После завершения цикла переменная *balancedSoFar* имеет значение *false*.

   5.2. После завершения цикла стек содержит одну открывающую фигурную скобку, а значение переменной *balancedSoFar* равно *true*.

   5.3. После завершения цикла стек оказывается пустым, а значение переменной *balancedSoFar* равно *true*.
6. 2
7. *ab/c**
8. Правила приоритетов определяют порядок ассоциативности. Оператор ≥ обладает левой ассоциативностью, если операторы имеют одинаковый приоритет.

9. Результаты трассировки таковы.

   9.1. Стек содержит символ A, а затем — A B.

   9.2. Стек содержит символ A, затем — A B, а затем — A B D.

   9.3. Стек содержит символ C, а затем — C D, затем C D H, а затем — C D H G.

Глава 7

1. A, B, C, D.
2. *queue1*: 2 3 5; *queue2*: 4 6 (элементы перечислены от головы к хвосту).
3. Результаты трассировки таковы.

   3.1. Когда цикл *for* завершается, стек и очередь выглядят следующим образом.

   Стек: `a b c d` ← вершина

   Очередь: `a b c d` ← хвост

   Символ *a*, находящийся на вершине стека, соответствует символу *a*, расположенному в начале очереди. После удаления символа *a* из обеих структур, символ *d*, находящийся на вершине стека, не соответствует символу *b*, расположенному в начале очереди, поэтому строка палиндромом не является.

   3.2. Из стека и очереди удаляются одинаковые буквы, поэтому строка является палиндромом.

4. Ответы таковы.

   4.1. 1;

   4.2. 3;

   4.3. 2;

   4.4. 3;

   4.5. 1;

   4.6. 2;

   4.7. 2;

   4.8. 1;

   4.9. 1;

   4.10. 2.

5. Сгенерировать отбытие для заданного прибытия независимо от других событий нельзя. Поэтому, чтобы считать файл прибытий и сгенерировать отбытия, необходимо выполнить некоторые вычисления, связанные с моделированием.

6. Ответ.

Время	Действие	*bankQueue* (от головы до хвоста)	*anEventList* (от головы до хвоста)
29	Обновить список *anEventList* и очередь *bankQueue*. В банк прибывает посетитель № 2.	23 2	A 30 3
	Посетитель № 3 начинает банковскую операцию, сгенерировать отбытие	23 2	A 30 3 D 31

30	Обновить список *anEventList* и очередь *bankQueue*. В банк прибывает посетитель № 4	23 2	30 3	D 31
31	Обновить список *anEventList* и очередь *bankQueue*. Из банка уходит посетитель № 3	30 3		*пусто*
	Посетитель № 3 начинает банковскую операцию, сгенерировать отбытие	30 3		D 34
34	Обновить список *anEventList* и очередь *bankQueue*. Из банка уходит посетитель № 4	*пусто*		*пусто*

Глава 8

1. Ответы.

   1.1. `Sphere mySphere(2.9);`

   1.2. `Ball myBall(6.0, "Пляжный волейбол");`

   1.3.
   ```
   cout << mySphere.getDiameter() << " "
      << myBall.getDiameter();
   ```

2. Класс *Planet* определяется следующим образом.
   ```
   class Planet: public Ball
   {
   public:
      double getDistanceFromSun();
      void setDistanceFromSun(double newDistance);
   private:
      double distance;
   };
   ```

3. Ответы.

   3.1. Функция *resetBall* не имеет непосредственного доступа в члену *theRadius*. Переменная *theRadius* является закрытым членом класса *Sphere*, поэтому производный класс не имеет к ней доступа.

   3.2. Функция *resetBall* имеет непосредственный доступ в члену *theRadius*. Вместо вызова *setRadius(r)* в реализации функции *resetBall* можно написать оператор *theRadius = r*. Однако делать это не обязательно.

4. Функции *play* и *record*, поскольку класс VCR может их замещать.

5. Объект *aList* должен быть экземпляром класса *SortedList*. Алфавитный список идентичен упорядоченному списку имен, в котором параметр *ListItemType* определен как строка. Если в объект *aList* нужно включить новые методы, необходимо вывести из класса *SortedList* новый класс.

6. Да. Поскольку объект *aList* является экземпляром производного класса, этот класс не может быть абстрактным. Если в производном классе нет реализации метода *displayList*, он может быть абстрактным классом, у которого нет объектов.

7. Производный класс не имеет доступа к закрытому методу своего базового класса и, следовательно, не может замещать его.

8. Операторы таковы.

```
NewClass<char> MyClass;
MyClass.setData('c');
cout << MyCLass.getData() << endl;
```

Глава 9

1. $(n - 1) + (n - 2) + \ldots + 1 = n * (n - 1)/2$

2. $n + (n - 1) + \ldots + 2 = n * (n + 1)/2 - 1$

3. Оценки таковы.

   3.1. $O(n^3)$;

   3.2. $O(\log n)$;

   3.3. $O(n)$.

4. Ответы.

   4.1. Поиск можно прекратить, как только значение *searchValue* станет меньше элемента, поскольку точка, в которой мог находиться искомый элемент уже пройдена.

   4.2. Упорядоченные данные с использованием схемы, описанной в задаче 4.1: наилучший вариант: O(1); средний вариант: $O(n)$, худший вариант: $O(n)$.

   Неупорядоченные данные: $O(n)$ во всех случаях.

   4.3. Независимо от того, упорядочены данные или нет, оценка сложности в лучшем случае равна O(1) (элемент обнаруживается после выполнения одного сравнения), а в среднем и худшем вариантах — $O(n)$ (элемент обнаруживается после $n/2$ или n сравнений, соответственно).

5. На каждом проходе выбранный элемент подчеркнут.

```
20 80 40 25 60 30
20 30 40 25 60 80
20 30 40 25 60 80
20 30 25 40 60 80
20 25 30 40 60 80
20 25 30 40 60 80
```

6. При каждом проходе отыскивайте не наибольший, а наименьший элемент.

```
20 80 40 25 60 30
30 80 40 25 60 20
30 80 40 60 25 20
60 80 40 30 25 20
60 80 40 30 25 20
80 60 40 30 25 20
```

7. Первый проход.

```
25 30 20 80 40 60
25 30 20 80 40 60
25 20 30 80 40 60
25 20 30 80 40 60
25 20 30 40 80 60
25 20 30 40 60 80
```

Второй проход.

```
25 20 30 40 60 80
20 25 30 40 60 80
20 25 30 40 60 80
20 25 30 40 60 80
20 25 30 40 60 80
```

На третьем проходе обменов больше нет, поэтому алгоритм завершается.

8. Результаты трассировки.

25 30 20 80 40 60

25 30 20 80 40 60

20 25 30 80 40 60

20 25 30 80 40 60

20 25 30 40 80 60

20 25 30 40 60 80

9. Ответ.

- Алгоритм *mergesort* упорядочивает массив, применяя сортировку методом слияния к каждой половине массива.
- Сортировка половины массивы представляет собой задачу меньшего размера, чем сортировка всего массива.
- Базисом является сортировка массива, состоящего из одного элемента.
- Разделяя массив пополам и повторяя этот процесс вновь, мы получим сегменты, состоящие из одного элемента, т.е базис.

10. Вертикальные линии разделяют массив на области. Опорным элементом является число 38.

```
38 | 16 40 39 12 27       ОЭ | НЧ   16 остается в S1 (на месте).
38 | 16 | 40 39 12 27     ОЭ | S1 | НЧ
38 | 16 | 40 | 39 12 27   ОЭ | S1 | S2 | НЧ
38 | 16 | 40 39 | 12 27   ОЭ | S1 | S2 | НЧ  Меняем местами 12 и 40
38 | 16 12 | 39 40 | 27   ОЭ | S1 | S2 | НЧ  Меняем местами 27 и 39
38 | 16 12 27 | 40 39     ОЭ | S1 | S2 | НЧ  Меняем местами  38 и 27
27 16 12 | 38 | 40 39     S1 | ОЭ | S2
```

Здесь ОЭ — опорный элемент, НЧ — неупорядоченная часть.

11. Ответы.

11.1. Сложность бинарного поиска оценивается величиной O(log n), поэтому он быстрее, чем алгоритм сортировки слиянием, сложность которого имеет порядок O(n log n).

11.2. Сложность бинарного поиска оценивается величиной O(log n), поэтому он быстрее, чем алгоритм вывода массива на экран, сложность которого имеет порядок O(n).

Глава 10

1. Ответы.

1.1. 60;

1.2. 60, 20, 40;

1.3. 20, 70; 10, 40; 30, 50;

1.4. 20 и 70, 10 и 40; 30 и 50.

1.5. 40, 20, 60;

1.6. 10, 40, 30, 50;

1.7. 70, 10, 30, 50.

2. Ответы.

   2.1. 1: A; 2: B, C 3: D, E; 4: F; 5: G

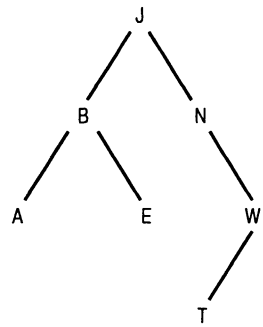
   2.2. 1: A; 2: B; 3: C; 4: D; 5: E; 6: F; 7: G

3. 4

4. Совершенное: *b*, *c*, *d*, *e*; полное: *e*; сбалансированное: *b*, *c*, *d*, *e*.

5. Обход в прямом порядке: A, B, D, E, C, F, G; в симметричном порядке: D, B, E, A, F, C, G; в обратном порядке: D, E, B, F, G, C, A.

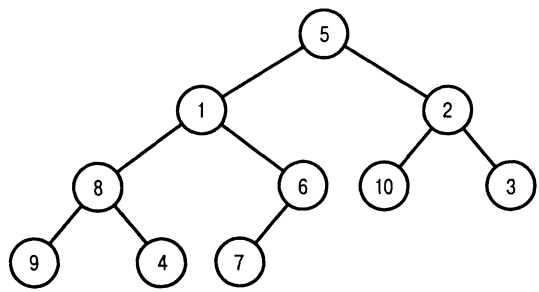
6.



7. Одно из возможных направлений обхода: 60, 20, 10, 40, 30, 50 и 70 (прямой порядок обхода).

8. Массив состоит из чисел 30 20 50 10 25 40 60

9.



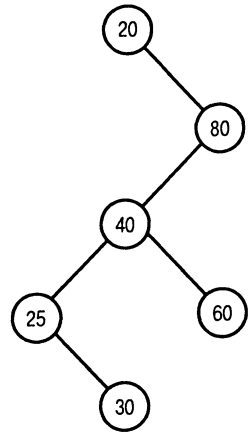
10. Нет. Дерево H должно быть правым поддеревом в дереве G. Деревья U и V должны быть правыми поддеревьями в дереве T.

11. Алгоритм сравнивает каждый заданный поисковый ключ с ключами следующих узлов.

    11.1. 60, 20, 40, 30;

    11.2. 60, 20, 10.

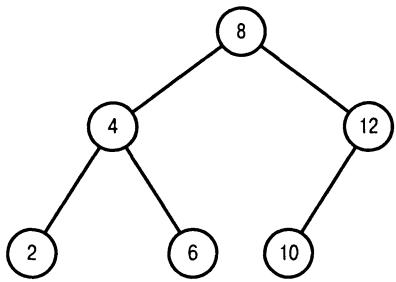
12. Вставка элементов массива в бинарное дерево поиска приводит к следующему результату.



В результате симметричного обхода этого дерева возникает упорядоченный массив.

13. Ответы.

13.1.



13.2. Это дерево имеет минимальную высоту и является совершенным, но не полным.

Глава 11

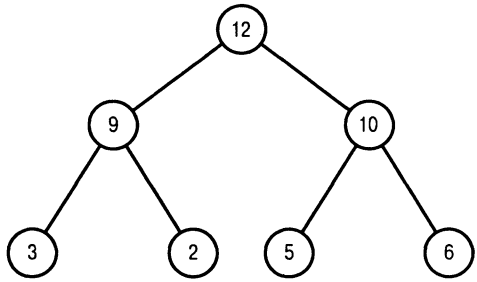
1. Псевдокод имеет следующий вид.

```
tableReplace(in t:Table, in x:KeyType,
             in replacementItem:TableItemType)
             throw TableException
t.tableDelete(x)
t.tableInsert(replacementItem)
```

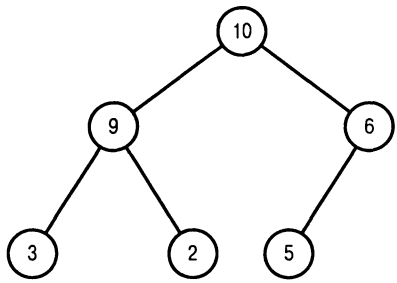
2. Нет.

3. Это не полукуча и не куча.

4. После вставки элемента 12 получается следующее дерево.



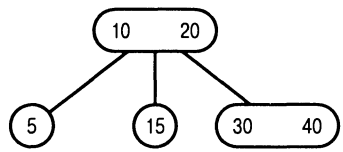
После удаления элемента 12 получается следующее дерево.



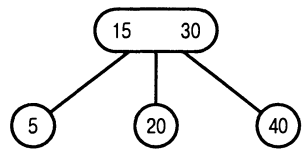
5. Массив, представляющий кучу, состоит из элементов 6 4 5 1 2 3.
6. Массив, представляющий кучу, состоит из элементов 7 5 6 4 3 2.
7. Массив состоит из элементов 10 9 5 8 7 2 3 1 4 6.

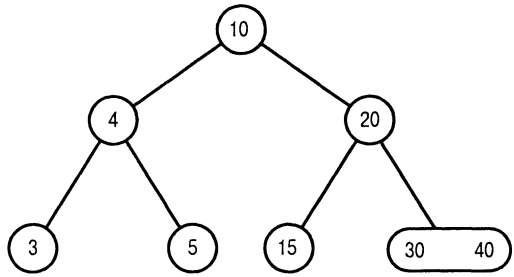
Глава 12

1.



2. Получаются следующие деревья.

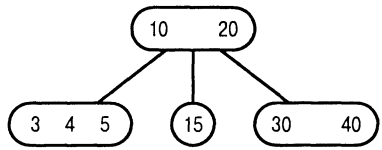




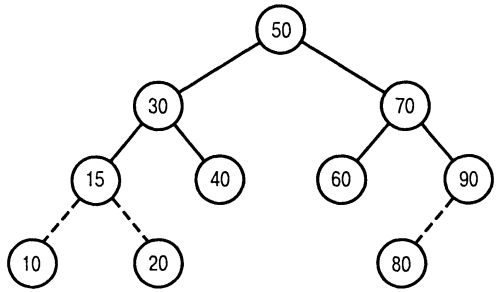
3. Ответы.

3.1. См. ответ к заданию 1.

3.2.



4.



5. Сбалансированное бинарное дерево поиска.

6. Каждый узел красно-черного дерева состоит из двух указателей и двух индикаторов цвета. Эти указатели и индикаторы занимают не больше памяти, чем четыре указателя в узле 2-3-4 дерева. Кроме того, узел красно-черного дерева запрашивает память только для одного элемента данных, в то время как узел 2-3-4 дерева хранит три элемента данных.

7. Псевдокод имеет следующий вид.

```
+tableDelete(in searchKey: KeyType) throw TableException

i = h(searchKey)
while ( (table[i] занята и table[i].getKey() != searchKey)
        или (table[i] удалена) )
   ++i

   if (table[i] не пуста)
   {
      // table[i].getKey() == searchKey
      Пометить ячейку table[i] как удаленную
   }
```

```
else
    Сгенерировать исключительную ситуацию TableException
```

8. 8, 10, 1, 3, 5, 7, 9, 0, 2, 4, 6.

9. Таблица хэширования выглядит следующим образом.

```
table[1] → 15 → 8
table[2] ðàâíà NULL
table[3] → 17 → 24 → 10
table[4] → 32
```

Глава 13

1. Ответы.

   1.1. Ориентированный, связанный.

   1.2. Неориентированный, связанный.

2. DFS: 0, 1, 2, 4, 3; BFS: 0, 1, 2, 3, 4.

3. Матрица смежности имеет следующий вид.

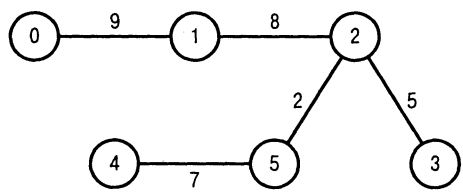
```
  |0 1 2 3 4
0 |0 1 0 0 0
1 |0 0 1 1 0
2 |0 0 0 0 1
3 |0 1 0 0 0
4 |1 0 0 0 0
```

4. Возможные топологические порядки обхода.

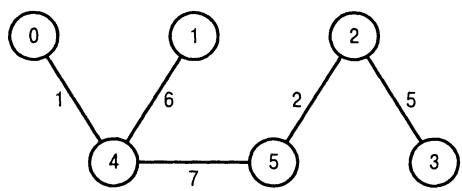
```
a g d b e c f
g a d b e c f
a g d b e f c
g a d b e f c
```

5. Нет.

6. Дерево имеет следующий вид.



7. Дерево имеет следующий вид.



8. Путь 0, 4, 2, 1 имеет вес 7.

Путь 0, 4, 2 имеет вес 5.

Путь 0, 4, 2, 3 имеет весь 8.

Путь 0, 4 имеет вес 4.

Глава 14

1. Последовательный доступ: копируем исходный файл *file1* в файл *file2*. Записываем новый блок, содержащий нужную запись и 99 пустых записей. Копируем файл *file2* в исходный файл *file1*.

2. Прямой доступ: создаем новый блок, содержащий требуемую запись и 99 пустых записей. Записываем новый блок в файл под 17-м номером.

   Псевдокод алгоритма *externalMergesort* имеет следующий вид.

```
externalMergesort(in unsortedFileName:string,
                  in sortedFileName:string)

   Связываем параметр unsortedFileName с файловой переменной
      inFile, а параметр sortedFileName — с файловой
      переменной outFile

   blocksort(inFile, tempFile1, numBlocks)
   // Записи в каждом блоке теперь упорядочены; numBlocks==16
   mergeFile(tempFile1, tempFile2, 1, 16)
      mergeRuns(tempFile1, tempFile2, 1, 1)
      mergeRuns(tempFile1, tempFile2, 3, 1)
      mergeRuns(tempFile1, tempFile2, 5, 1)
      mergeRuns(tempFile1, tempFile2, 7, 1)
      mergeRuns(tempFile1, tempFile2, 9, 1)
      mergeRuns(tempFile1, tempFile2, 11, 1)
      mergeRuns(tempFile1, tempFile2, 13, 1)
      mergeRuns(tempFile1, tempFile2, 15, 1)
   mergeFile(tempFile2, tempFile1, 2, 16)
      mergeRuns(tempFile2, tempFile1, 1, 2)
      mergeRuns(tempFile2, tempFile1, 5, 2)
      mergeRuns(tempFile2, tempFile1, 9, 2)
      mergeRuns(tempFile2, tempFile1, 13, 2)
   mergeFile(tempFile1, tempFile2, 4, 16)
      mergeRuns(tempFile1, tempFile2, 1, 4)
      mergeRuns(tempFile1, tempFile2, 9, 4)
   mergeFile(tempFile2, tempFile1, 8, 16)
      mergeRuns(tempFile2, tempFile1, 1, 8)
   copyFile(tempFile1, outFile)
```

3. Результаты трассировки таковы.

```
tableRetrieve(tIndex[1..20], tData, searchKey, tableItem)
  buf.readBlock(tIndex[1..20], 10)
  tableRetrieve(tIndex[1..9], tData, searchKey, tableItem)
    buf.readBlock(tIndex[1..9], 5)
    tableRetrieve(tIndex[1..4], tData, searchKey, tableItem)
      buf.readBlock(tIndex[1..4], 2)
      tableRetrieve(tIndex[1..1], tData, searchKey, tableItem)
        buf.readBlock(tIndex[1..1], 1)
        return false
```

4. Результаты трассировки таковы.

```
tableRetrieve(tIndex[1..20], tData, searchKey, tableItem)
  buf.readBlock(tIndex[1..20], 10)
  tableRetrieve(tIndex[11..20], tData, searchKey, tableItem)
    buf.readBlock(tIndex[11..20], 15)
    tableRetrieve(tIndex[11..14], tData, searchKey, tableItem)
      buf.readBlock(tIndex[11..14], 12)
      j = 26
      blockNum = 98
      data.readBlock(tData, 98)
      Найти запись data.getRecord(k), поисковый ключ которой
         равен значению параметра searchKey
      tableItem = data.getRecord(k)
      return true
```

Приложение А

1. Переменная *a* равна 5, затем 7, затем 8; переменная *b* равна 7; переменная *c* равна 5; переменная *d* равна 1; переменная *E* равна 5.
2. 56
3. `<< Title <<`
4. `j = 13`
   `j = 14`
   `k = 12`

   Переменная k вышла за пределы допустимого диапазона
5. Фрагмент кода имеет следующий вид.

```
if ((score >= 90) && (score <= 100))
   grade = 'A';
else if ((score >= 80) && (score < 90))
   grade = 'B';
else if ((score >= 70) && (score < 80))
   grade = 'C';
else if ((score >= 60) && (score < 70))
   grade = 'D';
else
   grade = 'F';
```

6. Применение оператора *switch* допускается, однако необходимо задать вариант *case* для каждого значения переменной *score* в диапазоне от 0 до 100.
7. Фрагмент кода имеет следующий вид.

```
int sum(int anArray[], int n)
{
   int s, i;
   for (s = 0, i = 0; i < n; s += anArray[i], ++i);
   return s;
} // Конец функции sum
```

8. Фрагменты кода имеют следующий вид.

   8.1. Первый фрагмент.

```
for (int day = 1; day <= DAYS_PER_WEEK; ++day)
    cout << minTemps[day-1][0] << " ";
```

8.2. Второй фрагмент.

```
for (int week = 1; week <= 5; ++week)
    cout << minTemps[0][week-1] << " ";
```

8.3. Третий фрагмент.

```
for (week = 1; week <= WEEKS_PER_YEAR; ++week)
   {
      for (day = 1; day <= DAYS_PER_WEEK; ++day)
      cout << minTemps[day-1][week-1] << " ";
      cout << endl;
   } // end for
```

9. Ответы.

9.1. `student.address.state;`

9.2. `student.address.zip[0];`

9.3. `student.gpa;`

9.4. `student.name[0].`

10. Символ подчеркивания отмечает положение файлового курсора.

```
originalFile                  ch       copyFile
a b <eoln> c d <eoln> <eof> ?          _
a b <eoln> c d <eoln> <eof> a          a _
a b <eoln> c d <eoln> <eof> b          a b_
a b <eoln> c d <eoln> <eof> <eoln>     a b <eoln> _
a b <eoln> c d <eoln> <eof> c          a b <eoln> c _
a b <eoln> c d <eoln> <eof> d          a b <eoln> c d _
a b <eoln> c d <eoln> <eof> <eoln>      a b <eoln> c d <eoln> _
a b <eoln> c d <eoln> <eof>
```

Приложение Г

1. Доказательство проводится методом математической индукции по m, когда $m = 0$, $2^0 = 2^1 - 1$. Предположим теперь, что утверждение истинно для $m = k$; иначе говоря, допустим, что $1 + 2^1 + 2^2 + \ldots + 2^k = 2^{k+1} - 1$. Покажем, что утверждение выполняется и для $m = k + 1$.

$(1 + 2^1 + 2^2 + \ldots + 2^k) + 2^{k+1} = (2^{k+1} - 1) + 2^{k+1} = 2^{k+2} - 1$.

2. Доказательство проводится методом математической индукции по n. Когда $n = 1$, первое нечетное число равно 1. Предположим теперь, что утверждение истинно для $n = k$; иначе говоря, допустим, что $1 + 3 + \ldots + (2k - 1) = k^2$. Покажем, что утверждение выполняется и для $n = k + 1$.

$(1 + 3 + \ldots + (2k - 1)) + (2k - 1) = k^2 + (2k + 1) = (k + 1)^2$.

3. Доказательство проводится методом математической индукции по n, когда $n = 2$, $rabbit(2) = 1 = a^0$. Предположим теперь, что утверждение истинно для $n = k$; иначе говоря, допустим, что $rabbit(n) > a^{n-2}$ для всех значений $n \leq k$. Покажем, что утверждение выполняется и для $n = k + 1$.

Предметный указатель

А

Абстрактный тип данных, 37
  бинарное дерево, 463
    реализация
      в виде массива, 470
      в виде связанного списка, 473
  бинарное дерево поиска, 488
    запись, 488
    ключ, 489
    поле записи, 488
  вершина, 271
  куча, 558
    максимальная, 559
    минимальная, 559
    реализация в виде массива, 564
  ориентированный на значения, 456
  очередь, 320
    двусторонняя, 339
    конец, или хвост, 320
    реализация
      в виде массива, 330
      в виде связанного списка, 325
      с помощью абстрактного списка, 335
  очередь с приоритетами, 555
    реализация в виде кучи, 567
  позиционно-ориентированный, 341, 456
  полукуча, 561
  список, 130
    неупорядоченный, 135
    реализация
      в виде массива, 154
      в виде связанного списка, 190
      с помощью итераторов, 397
    упорядоченный, 135
  стек, 269, 271
  таблица, 537
    реализация
      линейная, 541
        в виде упорядоченного массива, 548
      нелинейная, 542
        в виде бинарного дерева поиска, 552
Абстракция, 36
  данных, 37, 126
  процедурная, 37
  функциональная, 37, 124
Адрес ячейки памяти, 171
Аксиома, 140
Алгоритм, 25
  квадратический, 415
  кубический, 415
  линейный, 414
  логарифмический, 414
  порядок, 413
  сложность, 413
  экспоненциальный, 415
Алгоритм Дейкстры, 669
Анализ алгоритмов, 409
  наихудший вариант, 416
  средний вариант, 417

Б

Библиотека функций, 774
  заголовок, 774
  заголовочный файл, 774
  файл реализации, 774
Блок try-catch, 159
Блок-схема, 76

В

Верификация, 30
Выражение, 56
  алгебраическое, 246
    инфиксное, 246
    постфиксное, 247
    префиксное, 247
  арифметическое, 726
  логическое, 727
  условное, 727

Г

Граф, 646
  вершина, 646
    смежная, 646
  взвешенный, 648
  матрица смежности, 650
  мультиграф, 648
  неориентированный, 649
  несвязный, 647
  обход
    поиск в глубину, 654
    поиск в ширину, 656
  ориентированный, или орграф, 649
  остовное дерево, 662
  остовное дерево минимальное, 666
  остовное дерево поиска в глубину, 663
  остовное дерево поиска в ширину, 663
  планарный, 675
  подграф, 646
  простая цепь, 672
    Эйлера, 672
  путь, 646
    длина, 668
  ребро, 646
    ориентированное, 649
    петля, 648
  связный, 647
  связный компонент, 654
  совершенный, 647
  список смежности, 651

Д

Дерево, 457
  2-3, 581
    узел
      двухместный, 581
      трехместный, 581
  2-3-4, 599
    узел
      двухместный, 599
      трехместный, 599
      четырехместный, 599
  AVL, 611
    вращение, 612
  B-дерево степени m, 706
  n-арное, 522
  бинарное, 458
    поиска, 459
    полное, 461
    порядок обхода
      обратный, 468

прямой, 468
симметричный, 468
сбалансированное, 462
совершенное, 462
высота, 460
красно-черное, 607
общего вида, 458
поддерево, 457
поддерево узла, 457
ребро, 457
узел, 457
брат, 457
дочерний, 457
левый, 459
правый, 459
корень, 457
лист, 457
потомок, 457
предок, 457
родительский, 457
уровень, 460
Диагностическое утверждение, 30
Дрейф вправо, 58, 330

З

Заголовочный файл
cassert, 790
cctype, 790
cfloat, 791
climits, 791
cmath, 791
cstdlib, 792
cstring, 792
exception, 793
fstream, 793
iomanip, 793
iostream, 794
string, 794
Задача
о странствующем коммивояжере, 675
о трех услугах, 675
о ханойских башнях, 105
о четырех красках, 675
Замещение функций, 369
Запись активации, 76
Зондируемая последовательность, 622

И

Идентификатор, 57, 721
Инвариант, 30
Инвариант цикла, 30
Инициализатор, 148
Инкапсуляция, 38, 40
Исключительная ситуация, 54, 755
времени выполнения программы, 760
генерирование, 54, 159, 760
логическая, 760
перехват, 54, 159, 756
Итератор, 220, 395
двунаправленный, 221
Итерации, 70

К

Класс, 39
атрибуты, 39
базовый, 151
абстрактный, 375
данные-члены, 39
деструктор, 145
дружественный, 377
конструктор, 145
автоматический, 148
копирования, 191
по умолчанию, 147
методы, 39
потомок, 44
предок, 44
производный, 151
раздел
private, 364
protected, 364
public, 364
расширяемый, 373
функции-члены, 39, 145
члены
закрытые, 145
открытые, 145
шаблонный, 220, 386
шаблонный параметр, 220
экземпляр, 39
Кластер, 623
Кластеризация
вторичная, 624
первичная, 623
Клиент, 132
Ключевое слово, 721
Кодирование, 33
Комментарий, 721
Константа
именованная, 48, 724
литеральная, 723
Контейнер, 220
адаптерный, 339
Контракт, 28
Копирование
глубокое, 193
поверхностное, 192

М

Массив, 743
динамический, 176
логический размер, 154
физический размер, 154
Методы проектирования "сверху вниз", 40
модульный подход, 36
объектно-ориентированный подход, 38
Моделирование
временное, 345
событийное, 344
Модуль, 27
интерфейс, 27
слабо связанный, 27
узкоспециализированный, 27
Модульность, 27

Н

Наследование, 40, 359
закрытое, 365
защищенное, 365
открытое, 365

О

Объявление
using, 153
Оператор
break, 740
continue, 740
delete, 178
do, 743
for, 741
if, 737
switch, 738
throw, 160
typedef, 48, 725
while, 740
арифметический, 726
ввода, 730
взятия адреса, 172
вывода, 731
декрементации, 729
запятая, 742
инкрементации, 729
логический, 727
перегруженный, 40, 392
правоассоциативный, 726
присваивания, 725
разрешения области видимости, 148
составной, 738
сравнения, 726
точка, 754

условный, 738
цикла, 740
Отладка
программы, 61
средство наблюдения, 61
точка прерывания, 61
условных операторов, 62
функций, 62
циклов, 62
Отношение
" подобен", 368
"содержит", 367
"является", 366
Оценка риска, 29

П

Палиндром, 243
Передача параметров
по значению, 56, 735
по ссылке, 56, 735
Переменная
инициализация, 723
неинициализированная, 722
объявление, 722
описание, 722
статическая, 172
Перечисление, 724
Побочный эффект, 56
Поиск, 96
k-го наименьшего элемента массива, 102
бинарный, 71, 98, 419
наибольшего элемента в массиве, 97
полный перебор, 297
последовательный, 419
с возвратом, 237
Полиморфизм, 40
Порядок
лексикографический, 213
топологический, 658
Постусловие, 28
Поток
входной, 730
выходной, 730
данных, 27
ошибок, 730
Предусловие, 28
Преобразование типа
неявное, 727
повышающее, 727
явное, 727
Препроцессор, 775
Приведение типов, 728
Принцип
FIFO, 271
LIFO, 271
Принцип математической индукции, 32, 795
базис индукции, 32, 795
индуктивная гипотеза, 795
индуктивное заключение, 795
шаг индукции, 32, 795
Программа
макетная, 27
стоимость, 34
Программная документация, 60
Программное обеспечение
жизненный цикл, 26
технология разработки, 25
Пространство имен, 152
глобальное, 153

Р

Распределение памяти
динамическое, 174
статическое, 172
Рекуррентное отношение, 73
Рекурсия, 70
базис, 71
вырожденная задача, 71
Решение задачи, 25

С

Связывание
динамическое, или позднее, 369
статическое, или раннее, 364
Симметричный преемник, 501
Событие
внешнее, 344
внутреннее, 345
Совместимость типов, 366
Сокрытие информации, 38, 124
Сообщение, 150
Сортировка, 420
быстрая, 433
внешняя, 420, 686
внутренняя, 420
древовидная, 518
ключ, 420
методом
вставок, 202, 426
выбора, 421
пузырька, 424
пирамидальная, 569
поразрядная, 444
слиянием, 428
топологическая, 658
Список
голова, 180
дважды связанный, 211
кольцевой, 212
свободный, 234, 471
связанный, 170
кольцевой, 209
линейный, 209
фиктивный головной узел, 211
хвост, 201
узел, 179
Список смежности, 316
Стандартная библиотека шаблонов, 219
класс
deque, 339
list, 800
queue, 802
stack, 801
vector, 339
класс list, 222
класс queue, 337
класс stack, 290
Строка, 748
Структура, 753
Структура данных, 37, 126
Структурная схема, 40

Т

Таблица
виртуальных методов, 373
символов, 575
Тестирование, 33
Тип данных
агрегированный, 754
арифметический, 722
булев, 722
интегральный, 722
символьный, 722
целочисленный, 722
число с плавающей точкой, 722

У

Указатель, 171
Универсальный язык моделирования UML, 42
Утечка памяти, 174
Уточнение решения, 33

Ф

Файл
бинарный, 773
блок, 683

блочный доступ, 683
заголовочный, 146
запись, 683
индексная запись, 695
ключ, 695
указатель, 695
индексный, 694
исходный, 774
множественная индексация, 696
последовательного доступа, 682
прямого доступа, 682
реализации, 149
спецификации, 146
текстовый, 763
упорядоченный отрезок, 686
Форматирование
манипулятор, 733
флаг, 732
Функция
виртуальная, 369
вызов, 734
вычисляющая значение, 56
дружественная, 377
запись активации, 308
имя, 734
контракт, 125
локальное окружение, 76
полиморфная, 369
пустая, 56
стандартная, 737
тело, 734
тип возвращаемого значения, 734
фактические параметры, 734
формальные параметры, 734
чисто виртуальная, 375

X

Хэширование, 615
внешнее, 698
конфликт, 618
коэффициент загрузки, 629
механизм вычисления адреса, 616
схема предотвращения конфликтов, 619
двойное хэширование, 624
квадратичное зондирование, 623
линейное зондирование, 622
открытая адресация, 622
повторное хэширование, 625
таблица хэширования, 617
блок, 626
отдельное связывание, 626
цепочка, 626
функция хэширования, 617
вторичная, 625
выбор цифр, 619
идеальная, 618
модульная арифметика, 620
первичная, 625
свертка, 620

Ш

Шаблонный параметр, 386

Я

Язык, 241
алгоритм распознавания, 242
грамматика, 242

Научно-популярное издание

Фрэнк М. Каррано, Джанет Дж. Причард

Абстракция данных и решение задач на C++
Стены и зеркала, 3-е издание

Литературный редактор	*Е.П. Перестюк*
Верстка	*В.В.Терещенко*
Художественный редактор	*М.А. Смолина*
Обложка	*С.А. Чернокозинский*
Корректоры	*З.В. Александрова, Л.А. Гордиенко*
	Л.В. Коровкина, О.В. Мишутина

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 27.01.2003. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 68,37. Уч.-изд. л. 50,72.
Тираж 3500 экз. Заказ № 2365.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.