Александр Кубенский

bhv

Создание и обработка структур данных

в примерах на Java

- Базовые структуры данных
- Объектно-ориентированное программирование
- Алгоритмы классических задач
- Примеры решений

Мастер

Александр Кубенский

Создание и обработка СТРУКТУР ДАННЫХ

в примерах на Java

Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Книга посвящена алгоритмам обработки сложных структур данных. Рассматриваются решения наиболее распространенных задач: создание и изменение деревьев, поиск кратчайшего пути между вершинами в графе, обработка списков и массивов, символьное преобразование выражений. Примеры классических алгоритмов реализованы на языке Java, обеспечивающем объектно-ориентированный подход к программированию и являющемся универсальным при работе на различных платформах. Приводятся сведения о технологии построения программ, основу которых составляют объекты, обменивающиеся сообщениями. Описывается функциональное представление информации, позволяющее получать короткие и изящные программы для решения сложных задач.

Для широкого круга программистов

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Анатолий Адаменко*
Зав. редакцией	*Наталья Таркова*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Кубенский А. А.
Создание и обработка структур данных в примерах на Java. — СПб.: БХВ-Петербург, 2001. — 336 с.: ил.

ISBN 5-94157-095-3

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 24.09.01.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 27,09.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9-я линия, 12.

Содержание

Введение

Эта книга предназначена для тех читателей, кто уже умеет писать программы на одном (или нескольких) языке программирования, но пока имеет малый опыт программирования со сложными структурами данных, хочет узнать, как эффективно использовать различные структуры данных и классы при объектно-ориентированном подходе к программированию. В общем, эта книга для тех, кто хочет повысить свою программистскую квалификацию, приобретя новые знания и умения в области технологии работы со сложными структурами программ и данных.

К сожалению, часто бывает, что, научившись писать простые программы и изучив один язык программирования, школьники или студенты считают, что на этом наука программирования закончена, и теперь они могут программировать не хуже, чем любой опытный программист. Иногда к этому добавляется еще и знание конкретных библиотек и технологий, таких как MFC фирмы Microsoft или CORBA фирмы Sun Microsystems. Однако, оказывается, что владения даже самыми современными технологиями недостаточно, чтобы писать хорошие программы.

В настоящее время выработано много инструментальных средств, с помощью которых истинные мастера программирования с легкостью решают сложные задачи. Таким образом, часто достаточно лишь выбрать одно из известных решений. Вопрос лишь в том, чтобы знать эти решения. Но, к сожалению, книг, посвященных решениям традиционных задач проектирования программ и собственно программирования, не так уж и много. Такая литература традиционно пользуется большим спросом и составляет золотой фонд книг по программированию. Достаточно назвать лишь такие монографии, как [2], [3], [5], [7], [8], чтобы понять, насколько высоко ценятся такие книги.

Конечно, автор не претендует на то, чтобы войти в этот золотой фонд. Предлагаемая вашему вниманию книга содержит лишь некоторые избранные алгоритмы и решения, посвященные, в основном, работе со сложными структурами данных, и отобранные по принципу личных пристрастий самого автора. Тем не менее, хочется надеяться, что данная книга поможет начинающим программистам ощутить вкус к программированию и приучить к поиску элегантных решений программистских задач. Одни из приводимых решений можно брать сразу же в качестве готовых образцов программ и их

фрагментов. Другие лишь демонстрируют некоторый подход к построению программ, и сами по себе служат лишь примерами конкретных решений.

В качестве инструментального языка программирования, на котором записываются все примеры, выбран язык Java. Популярность его в последнее время необычайно сильно выросла. Если раньше язык рассматривался, в основном, как средство написания апплетов для страниц Интернета, то теперь он все чаще используется как язык программирования для написания приложений, в том числе, приложений большого объема с достаточно высокими требованиями к эффективности объектного кода. Разумеется, это не случайно. Язык обладает многими привлекательными чертами: строгий, хотя и не всеобъемлющий, объектно-ориентированный подход, ясный и полный набор конструкций языка, достаточно богатая и удобная библиотека стандартных классов и методов, встроенный сборщик "мусора", позволяющий свободно и удобно работать с памятью. Немаловажным свойством языка является также его универсальность при работе на различных платформах (UNIX, Windows, SunOS и др.), хотя последнее свойство не играет никакой роли в представляемой книге. Наконец, появление и развитие технологий быстрой компиляции классов языка Java в "родной" (native) код для многих платформ ("Just-In-Time"-компиляторы) позволило использовать язык там, где требуется быстрый код.

Автор не предполагает у своих читателей детального знания языка Java и, тем более, многочисленных библиотек Java-модулей. По своим конструкциям язык близок ко многим распространенным языкам — прежде всего к C++ и объектно-ориентированному Паскалю, так что мы надеемся, что читатели, знакомые с одним из этих языков, смогут понимать программы на Java без особого труда. Автор надеется, что для читателей, не знакомых с программированием на Java, книга может послужить дополнительным стимулом к его изучению и использованию.

В книге много примеров программного кода. Наверное, можно сказать, что основная часть информации содержится именно в программах, поэтому без их глубокого изучения, а может быть, и исполнения на компьютере, книга будет почти бесполезной. Еще раз повторюсь: несмотря на то, что в книге используются только основные конструкции и стандартные библиотеки языка Java, все же она не является учебником по языку, так что от читателей все же требуется знание основ программирования.

Книга состоит из семи глав. В первых двух главах приведено описание базовых структур данных и алгоритмов; в остальных — представлены примеры решения различных задач с использованием этих базовых и других структур данных и алгоритмов.

В *главе 1* вводятся структуры данных, используемые затем на протяжении всего дальнейшего изложения. Описаны способы представления в языке программирования Java таких известных структур данных, как массивы

(с фиксированными и плавающими границами), списки, деревья, множества и графы.

В *главе 2* вводится понятие абстрактного типа данных, активно использующееся затем на протяжении всей книги, а также приведены базовые алгоритмы обработки стеков и деревьев.

В *главе 3* описывается еще один из основных типов данных — строка. Приводятся примеры различных способов представления строк, отвечающие различным потребностям в написании алгоритмов обработки строк.

Глава 4 целиком посвящена символьным преобразованиям выражений. Выбор именно этих алгоритмов обусловлен личным интересом к ним автора, однако выражения оказываются очень простым и удобным материалом, на котором можно продемонстрировать многие интересные подходы и приемы программирования.

В *главе 5* приведены примеры некоторых классических алгоритмов обработки графов. Опять же, не претендуя на полноту описания, автор выбрал некоторые из достаточно простых и, на его взгляд, интересных алгоритмов. Это алгоритмы нахождения кратчайших путей между вершинами в графе, а также алгоритмы нахождения минимальных остовных деревьев (скелетов) графа.

Глава 6 посвящена технологии построения программ, при которой основу программы составляют объекты, обменивающиеся между собой сообщениями. Обычно такая технология применяется только для построения очень больших программ, таких как операционные системы, системы программирования и т. п. Достаточно привести в качестве примера операционную систему MS Windows. Тем не менее, в этой главе представлены примеры достаточно простых программ, в которых применение данного подхода также может быть оправдано, хотя, конечно, существуют и более простые способы решения приводимых в этой главе задач.

Наконец, в *главе 7* приводится один из нетрадиционных способов представления информации — функциональное представление. Такое представление чаще всего используется в функциональном программировании, однако даже в традиционных императивных языках[1] можно применять некоторые из подходов и приемов функционального программирования. Часто это дает возможность получать необыкновенно короткие и изящные программы для решения сложных задач. Опять же, для приведенных в этой главе задач, наверное, можно и даже лучше использовать другие подходы, однако хочется не столько продемонстрировать готовое решение, сколько подход, который может использоваться в более сложных случаях.

Примеры, приводимые в книге, не предназначены для непосредственного копирования, они, скорее, могут послужить отправной точкой для самостоя-

[1] То есть языках, основанных на последовательном исполнении операторов.

тельного программирования. Несмотря на это, все программы тщательно проверялись и отлаживались с использованием системы программирования JBuilder версии 4.0 фирмы Inprise с использованием версии языка, совместимой с JDK 1.3 фирмы Sun Microsystems.

Автор благодарен Марине Валерьевне Дмитриевой, с которой вместе в издательстве СПбГУ 5 лет назад был подготовлен и выпущен первый (сильно отличающийся от этого) вариант книги, своему научному руководителю Святославу Сергеевичу Лаврову, который принимал активное участие в подготовке той, первой, книги, а также издательству "БХВ-Петербург" за предоставленную возможность издания книги.

Глава 1

Способы представления структур данных

В этой главе обсуждаются способы представления базовых структур данных, которые затем будут использоваться на протяжении всей книги — массивы, списки, деревья, множества, графы и др. Часто уже после завершения этапа проектирования возникает вопрос: "Как правильно выбрать структуры данных для своей программы?" Иногда можно просто использовать объекты, предлагаемые выбранной системой программирования, но бывают ситуации, когда приходится создавать свои собственные базовые структуры, более подходящие для целей разрабатываемого проекта, чем стандартные "универсальные" структуры. Это объясняется множеством причин. Наиболее веской из них является соображение эффективности рабочей программы. Действительно, стандартные структуры данных, спроектированные "на все случаи жизни", могут требовать излишне много ресурсов, использовать очень общие, долго работающие алгоритмы и т. д.

Не претендуя на универсальность, мы рассмотрим несколько базовых структур и основные подходы к их реализации в программных проектах.

1.1. Массивы

Массивы данных широко используются в языках программирования для представления объектов, состоящих из определенного числа компонент (элементов массива) одного и того же типа (класса). Очень часто базовой операции над массивом — индексации — бывает достаточно для решения задачи. Пусть, например, решается задача кодирования текста, в которой необходимо каждую букву текста представить некоторым целочисленным кодом. (Задача не имеет отношения к шифровке шпионских донесений или обеспечению режима секретности. Скорее, она применима к "переводу" текста из одной системы кодирования в другую.) Более точно: необходимо по заданному тексту (строке) получить массив целых чисел той же длины, в котором каждому символу исходной строки соответствует его код. Очевидно, наиболее удобным способом решения такой задачи будет составление кодирующей таблицы — массива, в котором каждому символу сопоставлен

некоторый целочисленный код. Если считать, что коды исходных символов лежат в диапазоне от 1 до 255, то кодирующая таблица может быть представлена следующим описанием.

```
final static int[] codeTable = { ... };  // элементы не показаны
```

Функция кодирования текста с помощью этой таблицы использует только операцию индексации массива:

```
static void doCode (String source, int[] dest) {
  for (int i = 0;  i < source.length();  i++) {
    dest[i] = codeTable[(byte)source.charAt(i)];
  }
}
```

Однако еще чаще встречаются ситуации, в которых только одной операции индексации недостаточно. Например, если исходные коды символов расположены в диапазоне от 32 до 255, то уже в этом случае пользоваться кодовой таблицей так, как показано выше, оказывается неудобно: индексацию приходится делать со "смещением". Не очень подходит простой массив и для таких обычных ситуаций, как неправильно заданный индекс (надо генерировать подходящее в каждом отдельном случае исключение!), динамическое добавление или удаление элементов (надо перераспределять память!) и т. д. Во всех подобных ситуациях лучше представлять массивы с помощью объектов специально спроектированного класса, подходящего для целей нашей задачи. Ниже приведено описание класса для решения вышеизложенной задачи кодирования при условии, что коды символов лежат в "смещенном" диапазоне.

В качестве исключительной ситуации при задании неправильных индексов будем использовать стандартный класс `java.lang.IndexOutOfBoundsException`, однако текст сообщения станем генерировать, исходя из каждой конкретной ситуации.

Листинг 1.1. Определение класса `Table`

```
public class Table {
  int lBound;     // нижняя граница элементов
  int hBound;     // верхняя граница элементов
  int[] array;    // собственно массив

  // В конструкторе задаются границы создаваемого массива
  public Table(int low, int high)
         throws IndexOutOfBoundsException {
```

```
    // Здесь просто вызовем другой, более общий конструктор:
    this(low, high, null);
  }

  // Следующий конструктор имеет еще один параметр — массив,
  // из которого берутся элементы для начального заполнения
  // нового массива
  public Table(int low, int high, int[] iniTable)
         throws IndexOutOfBoundsException {
    if ((hBound = high) < (lBound = low)) {
      throw new IndexOutOfBoundsException(
        "Конструктор Table: нижняя граница " + low +
        " больше, чем верхняя " + high);
      );
    }
    array = new int[high - low + 1];
    if (iniTable != null) {
      for (int ndx = 0;
           ndx <= high - lBound && ndx < iniTable.length;
           ndx++)
        array[ndx] = iniTable[ndx];
    }
  }

  // Следующий конструктор в качестве аргумента получает
  // уже имеющийся объект класса Table и создает его копию
  public Table(Table src) {
    array = new int [(hBound=src.hBound)-(lBound=src.lBound)+1];
    for (int ndx = lBound; ndx <= hBound; ndx++)
      array[ndx-lBound] = src.array[ndx-lBound];
  }

  // Реализация операции индексации — выборки значения элемента
  public int elementAt(int i)
         throws IndexOutOfBoundsException {
    if (i < lBound || i > hBound) {
      throw new IndexOutOfBoundsException(
```

```
                "Индекс " + i + " выходит за границу массива");
    }
    return array [i - lBound];
  }
};
```

Теперь решение той же задачи кодирования текста с помощью нашей новой кодовой таблицы может выглядеть так же просто, как и раньше:

```
static final Table codeTable = new Table(32, 255, tab);


static void doCode(String source, int[] dest) {
  for (int i = 0;  i < source.length();  i++) {
    dest[i] = codeTable.elementAt((byte)source.charAt(i));
  }
}
```

Конечно, определение класса `Table` не является универсальным и, скорее всего, для решения иной подобной задачи потребуется запрограммировать другой класс. Можно попробовать написать "универсальный" класс, который затем можно будет использовать во многих программах, где требуется обработка массивов. Вопреки некоторой тяжеловесности, такой класс имеет право на существование, даже несмотря на то, что стандартные библиотека поддержки Java имеют определение такого стандартного класса — `Array`. Нестандартное решение может потребоваться в том случае, когда "стандартные" средства не имеют всех необходимых функций или слишком неэффективны для проектируемой программы. В лисинге 1.2 показано, как можно самому определить массив с "плавающей" верхней границей. Такой массив можно расширять или укорачивать с помощью метода `resize`, а также добавлять в него новые элементы (в конец массива) с помощью функции `add`. Для увеличения общности в качестве класса элементов массива используется класс `Object`.

Листинг 1.2. Определение класса `DynArray`

```
public class DynArray {
  int size;         // текущий размер массива (количество элементов)
  int maxSize;      // размер отведенной памяти
  Object[] array;   // сам массив (размера maxSize)

  // Конструктор массива. Аргумент указывает, сколько памяти
  // надо отвести под его элементы
  public DynArray(int sz)
```

```
throws IndexOutOfBoundsException {
  this(sz, sz, null);  // используем вызов более общего конструктора
}

// В следующем конструкторе указывается, сколько памяти
// используется под элементы и сколько отведено всего
public DynArray(int sz, int maxSz)
throws IndexOutOfBoundsException {
  this(sz, maxSz, null);  // используем вызов более общего конструктора
}

// Еще один дополнительный аргумент содержит массив
// для начальной инициализации элементов
public DynArray(int sz, int maxSz, Object[] iniArray)
throws IndexOutOfBoundsException {
  if ((size = sz) < 0)
    throw new IndexOutOfBoundsException("Отрицательный размер: " + sz);
  maxSize = (maxSz < sz ? sz : maxSz);
  array = new Object[maxSize];    // выделение памяти
  if (iniArray != null) {         // копирование элементов
    for (int i = 0;  i < size && i < iniArray.length;  i++)
      array[i] = iniArray[i];
    // Можно было воспользоваться стандартной функцией System.arraycopy
  }
}

// Операция выборки элемента
public Object elementAt(int i)
throws IndexOutOfBoundsException {
  if (i < 0 || i >= size)
    throw new IndexOutOfBoundsException(
      "Индекс " + i + " выходит за границы диапазона [0," +
      (size - 1) + "]");
  return array[i];
}

// Изменение текущего размера массива. Аргумент delta задает
// размер изменения (положительный - увеличение размера;
// отрицательный - уменьшение)
```

```
  public void resize(int delta) {
    if (delta > 0) enlarge(delta);        // увеличение размера массива
    else if (delta < 0) shrink(-delta);   // уменьшение размера массива
  }

  // Операция расширения массива
  void enlarge(int delta) {
    if ((size += delta) > maxSize) {  // необходимо выделить
                                      // новый объем памяти
      maxSize = size;
      Object[] newArray = new Object[maxSize];
      // копируем элементы
      for (int i =0;  i < size - delta;  i++)
        newArray[i] = array[i];
      array = newArray;
    }
  }

  // Операция уменьшения размера массива
  void shrink(int delta) {
    size = (delta > size ? 0 : size — delta);
  }

  // Добавление одного нового элемента
  // (с возможным расширением массива)
  void add(Object e) {
    resize(1);
    array[size-1] = e;
  }
}
```

Как обычно, с повышением общности некоторые частные аспекты приходится реализовывать не очень простым и удобным способом, как раньше. Тем не менее, функция `doCode` для таблицы, представленной в виде динамического массива, выглядит почти так же просто, как и раньше. В приведенном ниже тексте программы предполагается, что элементами кодовой таблицы служат объекты класса-оболочки `Integer`.

```
static void doCode(String source, int[] dest) {
  for (int i = 0;  i < source.length();  i++) {
```

```
    dest[i] = ((Integer)codeTable.elementAt
                    ((byte)source.charAt(i))).intValue();
  }
}
```

В конце раздела заметим, что, вообще говоря, язык Java предлагает достаточно богатый набор стандартных интерфейсов и объектов для работы с массивами. Кроме уже упомянутого класса `Array`, имеется также большое количество классов и интерфейсов для массивов и массивоподобных структур: `Vector`, `Collection`, `Map`, `Hashtable`, `LinkedList` и др. Многие из них весьма удобны для использования, хотя на наш взгляд, несколько тяжеловесны и часто малоэффективны. Как правило, для серьезной работы с массивами приходится создавать свои собственные классы.

1.2. Списки

Если массив всегда занимает непрерывный участок памяти, то список является простейшим примером так называемой "динамической" структуры данных. В динамических структурах данных объект содержится в различных участках памяти, количество и состав которых может меняться в процессе работы. Единство такого объекта поддерживается за счет объединения его частей в описании класса.

Простейший *линейный список* представляет собой линейную последовательность элементов. Для каждого из них, кроме последнего, имеется следующий элемент и для каждого, кроме первого — предыдущий. Список традиционно изображают в виде последовательности элементов, каждый из которых содержит ссылку (указатель) на следующий и/или предыдущий элемент (рис. 1.1), однако физически в представлении элементов списка может и не быть никаких ссылок.

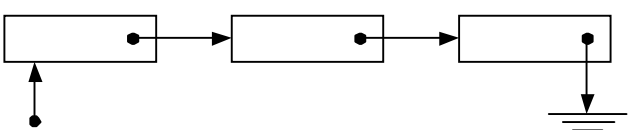


Рис. 1.1. Структура связи элементов списка

Типичный набор операций над списком будет включать добавление, удаление и поиск его элементов, вычисление длины списка, последовательную обработку всех элементов (итерацию) списка.

Как и в случае использования массивов, многие библиотеки (в том числе стандартный пакет `java.util`) для программ на языке Java включают в себя возможность описания и работы со "стандартными" списками (например, час-

то используемые классы `Vector` и `LinkedList`). Несмотря на это, часто возникает необходимость описания своих собственных структур данных в виде списков, содержащих более подходящие для решаемой задачи операции, более простые (и, следовательно, более эффективные), чем стандартные, или обладающие специфическими особенностями (например, упорядоченные списки).

Как правило, при описании списка в виде отдельного класса представляется элемент списка, содержащий ссылку на следующий и/или предыдущий элемент. В листинге 1.3 представлено описание простейшего однонаправленного списка из целых чисел с применением операций добавления нового элемента в начало и конец списка, удаления из начала списка (удаление из конца списка не является типичной операцией для однонаправленных списков ввиду своей относительной сложности и неэффективности), подсчет числа элементов в списке.

Объекты класса `IntList` содержат ссылки на первый и последний элементы списка, а также счетчик числа элементов, использующийся в функции, выдающей количество элементов списка.

Листинг 1.3. Определение класса `IntList`

```
public class IntList {
/* Класс ListItem представляет элемент списка,
   связанный со следующим с помощью поля next
*/
  static class ListItem {
    int item;          // значение элемента списка
    ListItem next;     // указатель на следующий элемент списка

    // Конструктор создания элемента списка
    public ListItem(int i, ListItem n) {
      item = i;
      next = n;
    }
  };

  int count = 0;              // счетчик числа элементов
  ListItem first = null;      // первый элемент списка
  ListItem last = null;       // последний элемент списка

  // Создание пустого списка
  public IntList() {}
```

```
// Создание копии уже имеющегося списка
public IntList(final IntList src) {
  addLast(src);   // добавляет список src в конец списка this
}

// Добавление элементов в конец списка
public void addLast(final IntList src) {
  for (ListItem cur = src.first;  cur != null;  cur = cur.next)
    addLast(cur.item);  // добавление одного элемента — см. ниже
}

// Добавление элемента в начало списка
public void addFirst(int item) {
  // Создаем новый элемент списка
  ListItem newItem = new ListItem(item, first);
  if (first == null) {
    // Новый элемент будет и первым, и последним в списке
    last = newItem;
  }
  first = newItem;
  count++;
}

// Добавление элемента в конец списка
public void addLast(int item) {
  // Создаем новый элемент списка
  ListItem newItem = new ListItem(item, null);
  if (last == null) {
    // Новый элемент будет и первым, и последним в списке
    first = newItem;
  } else {
    // Присоединяем новый элемент к последнему
    last.next = newItem;
  }
  last = newItem;
  count++;
}
```

```
  // Удаление первого элемента списка
  public int remove() {
    int res = first.item;  // содержимое первого элемента
    first = first.next;    // второй элемент становится первым
    count--;
    return res;   // удаленный элемент выдается в качестве результата
  }

  // Количество элементов списка
  public int getCount() { return count; }
}
```

В приведенном описании списка есть два существенных недостатка. Во-первых, не определены методы для итерации списка, т. е. нет способа перебрать все элементы списка, не изменяя его. Между тем, итерация списка — это одна из наиболее часто встречающихся операций над списком. Во-вторых, операции не защищены от некорректного использования. Не зная деталей реализации, невозможно узнать, что произойдет, если, например, будет предпринята попытка удалить элемент из пустого списка (в приведенной реализации возникнет исключительная ситуация `NullPointerException`).

Устранить второй недостаток не представляет особого труда. Надо лишь описать соответствующие исключительные ситуации (или воспользоваться одной из имеющихся в библиотеке) и вставить соответствующие проверки в операции, которые могут быть вызваны некорректно.

Чтобы ликвидировать первый недостаток, надо сначала понять, чего же мы, собственно, хотим, т. е. какие операции надо определить, чтобы можно было перебрать элементы списка. Обычно для итерации списка предлагают два способа: внутренний итератор и внешний итератор.

Внутренний итератор — это метод, позволяющий выполнить для каждого элемента списка одну и ту же операцию — параметр итератора (говорят, что внутренний итератор "посещает" элементы списка). Для представления этой операции опишем новый интерфейс, представляющий классы, содержащие функции обработки элементов.

```
public interface Visitor {
  void visit(int item);
}
```

В этом случае итератор будет иметь следующий вид

```
public void iterator(Visitor visitor) {
  for (ListItem cur = first;  cur != null;  cur = cur.next)
    visitor.visit(cur.item);
}
```

Несмотря на простоту и элегантность такого решения, пользоваться описанным итератором бывает неудобно. Во-первых, требуется для любой итерации определять операцию обработки каждого элемента в виде отдельного класса с методом `visit`, а это иногда приводит к не слишком понятной программе. Например, для решения задачи суммирования элементов списка можно определить следующий класс.

```
public class Summator implements Visitor {
  int sum = 0;
  public int getSum() { return sum; }
  public void visit(int item) { sum += item; }
}
```

В этом классе метод `visit` используется для добавления очередного элемента списка к текущей сумме. Если переменная `lst` имеет в качестве значения некоторый список, то вывести в выходной поток сумму его элементов можно с помощью следующих вызовов.

```
Summator summator = new Summator();
lst.iterator(summator);
System.out.println(summator.getSum());
```

Второе неудобство состоит в том, что с помощью внутреннего итератора невозможно решить такие простейшие задачи, как, например, поэлементное сравнение двух списков. В такой задаче требуется одновременная работа двух итераторов, однако итератор не может прервать работу до тех пор, пока обработка всех элементов списка не будет закончена.

В-третьих, внутренним итератором трудно управлять в случаях, когда итерацию требуется закончить досрочно, например, при поиске определенного элемента списка.

Удобным способом итерации списка был бы такой, при котором итерацию можно было бы организовать в виде обычного цикла:

```
for (<начать итерацию>; <еще есть элементы>;
     <перейти к следующему элементу>) {
  <взять очередной элемент>;
  <обработать очередной элемент>;
}
```

При таком способе организации работы управление берет на себя цикл, так что от всех недостатков внутреннего итератора удается избавиться. Действительно, теперь ничто не мешает нам организовать параллельный просмотр списков — для этого в заголовке цикла достаточно начать итерацию не одного, а сразу двух списков и, соответственно, организовать перемещение по двум спискам одновременно. Досрочное прекращение итерации также не

представляет труда, поскольку управление просмотром ведется не изнутри списка, а с помощью цикла. Такая итерация с внешним управлением называется *внешней итерацией.*

Один из способов организовать внешнюю итерацию — это перенумеровать все элементы списка и ввести функцию (скажем, `getElementAt`), которая будет выдавать элемент списка по его индексу. Цикл при этом приобретает простой и привычный вид:

```
for (int i = 0; i < lst.getCount(); i++) {
  Object next = lst.getElementAt(i);
  <обработать очередной элемент>;
}
```

Именно такая идеология списков с нумерованными элементами предлагается в стандартных классах `Vector`, `LinkedList` и других классах, реализующих стандартный интерфейс `List`. Однако операция выборки элемента по его номеру может оказаться слишком неэффективной при реализации списка в виде набора элементов, связанных ссылками. Действительно, для того чтобы найти, скажем, последний элемент списка, может потребоваться просмотреть весь список! Придется найти более эффективные методы организации внешнего итератора.

К сожалению, также оказывается, что для организации внешнего итератора недостаточно определить методы для выполнения элементарных действий, описанных словесно в заголовке цикла в виде *<начать итерацию>* и др.

Пусть, например, определены методы начала итерации (`start`), проверки конца списка (`hasMore`), перехода к следующему элементу (`next`) и выборки очередного элемента (`getCurrent`). Тогда цикл, организующий, скажем, то же суммирование элементов, мог бы выглядеть следующим образом.

```
IntList lst; ...
int sum = 0;
for (lst.start();  lst.hasMore();  lst.next()) {
  sum += lst.getCurrent();
}
```

Это проще и привычнее, чем решение данной задачи, приведенное выше для суммирования элементов с помощью внутреннего итератора.

Нетрудно определить описанные методы — `start`, `hasMore`, `next`, `getCurrent`, однако приведенное решение будет также несвободно от серьезных недостатков. Во-первых, по-прежнему невозможно организовать работу двух итераторов одновременно с одним и тем же списком, как это нужно, например, для решения задачи о поиске одинаковых элементов внутри одного списка или при сортировке списка. Причина в том, что при такой реализации состояние итерации должно храниться где-то внутри объ-

екта-списка, и это состояние не должно изменяться в промежутке между вызовами отдельных методов, таких как `hasMore()` или `next()`. Однако это условие будет нарушено при попытке вызова методов второго итератора для того же самого списка.

Во-вторых, этот внешний итератор предъявляет определенные требования к последовательности действий, и в то же время очень трудно проконтролировать эти требования. Главное из таких требований — всякая итерация должна начинаться вызовом метода `start`, в противном случае результат работы будет непредсказуем. Еще одно требование — в ситуации конца итерации (`!hasMore()`) бессмысленно обращаться к методам `next()` и `getCurrent()`.

Можно сказать, что несмотря на внешнюю независимость методов, определяемых для организации итерации, в описанном случае они все же остаются зависимыми друг от друга и от состояния объекта, к которому применяются.

Лучше всего сосредоточить все операции по организации итерации внутри объекта специального класса, отдельного от класса `IntList`. В пакете `java.util` имеются стандартные интерфейсы для классов, реализующих итераторы — `java.util.Iterator` и `java.util.Enumeration`. В первом из этих интерфейсов определены три метода — `hasNext()`, `next()` и `remove()`. Метод `hasNext()` предназначен для проверки, есть ли еще элементы для итерации. Метод `next()` служит для выдачи очередного элемента с одновременным "перемещением" итератора к следующему элементу. Метод `remove()` нужен для удаления текущего элемента из структуры в процессе итерации. В интерфейсе `Enumeration` подобную же функцию несут методы `hasMoreElements()` и `nextElement()` (метод, подобный `remove()`, в этом интерфейсе не предусмотрен). В нашем случае класс `IntList` должен обеспечить построение и выдачу объекта класса, реализующего итератор. Если операция `iterator()` класса `IntList` будет генерировать и выдавать такой итератор, то цикл, реализующий суммирование элементов списка, будет выглядеть следующим образом.

```
IntList lst; ...
int sum = 0;
for (Iterator i = lst.iterator();  lst.hasMore();  ) {
  sum += ((Integer)lst.next()).intValue();
}
```

Заметим, что в стандартном интерфейсе итератора метод `next()` выдает результат класса `Object`, поэтому в нашем примере мы воспользовались классом-оболочкой `Integer` для представления целочисленных элементов списка, последовательно выдаваемых итератором.

Реализация класса итератора и метода `iterator()` в контексте определения класса `IntList` будет выглядеть как в листинге 1.4.

Листинг 1.4. Определение внешнего итератора для списка целых

```
public static class IntListIterator implements java.util.Iterator {
  private ListItem current;     // текущий элемент

  // Конструктор итератора
  public IntListIterator(IntList lst) { current = first; }

  // Проверка существования следующего элемента
  public boolean hasNext() { return current != null; }

  // Выдача очередного элемента списка
  // и перемещение текущего указателя на следующий элемент
  public Object next() {
    if (!hasNext()) return null;
    Integer item = new Integer(current.getItem());
    current = current.getNext();
    return item;
  }

  // Пустая функция!
  public void remove() {}
};

public java.util.Iterator iterator() {
  return new IntListIterator(this);
}
```

В реализации класса итератора списка определение метода `remove()` дано только для того, чтобы обеспечить совместимость класса с интерфейсом `java.util.Iterator`. Нигде в примере эта функция не используется, поэтому в метод не вложено никакой семантики. Обычно для метода `remove()` подразумевается реализация удаления очередного (в порядке итерации, т. е. "текущего") элемента. Однако, вообще говоря, нарушение структуры списка во время итерации достаточно опасно и может привести к неожиданным эффектам в случае, когда один и тот же список обрабатывается одновременно несколькими итераторами. Возможно, более подходящим для итерации интерфейсом был бы стандартный интерфейс `java.util.Enumeration`, однако и здесь, и в дальнейшем в этой книге в основном будет использоваться интерфейс `Iterator`, хотя бы потому, что сам термин "итератор" соответст-

вует имени этого интерфейса. Чтобы показать, что для некоторого конкретного итератора операция `remove` невыполнима, вообще говоря, надо генерировать исключительную ситуацию `UnsupportedOperationException`, однако в нашей книге для простоты во всех случаях мы будем просто оставлять тело функции пустым.

Приведенная реализация итератора обеспечивает самые разнообразные потребности в обработке списков. Вот как может, например, выглядеть функция поэлементного сравнения двух списков.

```
static boolean equalLists(IntList lst1, IntList lst2) {
  boolean equal = true;
  Iterator i1 = lst1.iterator();
  Iterator i2 = lst2.iterator();
  for ( ; equal && i1.hasNext() && i2.hasNext(); ) {
    equal = ((Integer)i1.next()).equals((Integer)i2.next());
  }
  return equal && !i1.hasNext() && !i2.hasNext();
}
```

В приведенной функции одновременно запускаются два итератора над двумя различными списками. А в следующем фрагменте, в котором определена функция, проверяющая, что все элементы списка различны, два итератора работают одновременно над одним и тем же списком.

```
static boolean distinct(IntList lst) {
  for (Iterator i = lst.iterator();  i.hasNext(); ) {
    Integer item = (Integer)i.next();
    int count = 0;
    for (Iterator j = lst.iterator();  j.hasNext(); ) {
      if (((Integer)j.next()).equals(item))
        count++;
    }
    if (count > 1) return false;
  }
  return true;
}
```

Каждый из запущенных итераторов осуществляет свой перебор элементов списка, так что они могут работать совершенно независимо друг от друга.

Сравнивая между собой внешний и внутренний итераторы, можно отметить, что каждый из этих двух подходов имеет свои преимущества. О преимуществах внешнего итератора мы уже говорили, но и внутренний итератор име-

ет свои преимущества. Так, поскольку логика работы внутреннего итератора определяется только самим итератором, он может полагаться на порядок просмотра элементов, в сложных случаях он может временно изменять структуру и/или значения, хранящиеся в элементах списка, и т. д. По той же самой причине внутренний итератор лучше защищен от ошибочного использования — невозможно, например, попытаться перейти к следующему элементу, не начав итерацию.

Гибкость списковых структур особенно ярко проявляется при вставке и удалении элементов. Действительно, при этом нет необходимости перемещать элементы списка, достаточно лишь заменить значения нескольких ссылочных полей. Это может значительно ускорить работу программы в случае списков большого объема, хотя может случиться и так, что для доступа к элементу списка потребуется просмотреть значительную часть списка прежде, чем требуемый элемент будет найден.

Обычно для списков предлагаются также методы, позволяющие вставлять и удалять элементы списка в соответствии с некоторым критерием. Например, для предложенного списка из целых чисел (см. листинг 1.3) можно предложить операцию удаления элементов с заданным значением. Для реализации этой операции потребуется пройти вдоль всего списка, удаляя все элементы, содержащие конкретное значение. Здесь как раз могла бы пригодиться операция `remove` из интерфейса `Iterator`! Мы, однако, поступим по-другому и реализуем удаление непосредственно внутри списка.

Для удаления помимо указателя на текущий элемент списка придется хранить еще и указатель на предыдущий элемент. Это необходимо, поскольку при удалении элемента корректируется ссылка, содержащаяся в предыдущем элементе списка. В приведенной реализации метод возвращает значение **`true`**, если хотя бы один элемент списка был удален, и **`false`** — в противном случае.

```
public boolean remove(int n) {
  ListItem pred = null,       // указатель на предыдущий элемент
           current = first;   // указатель на текущий элемент
  boolean found = false;
  for (; current != null; current = current.next) {
    if (current.item == n) {
      found = true;
      count--;                // уменьшаем количество элементов
      if (pred != null) {     // корректируем ссылку на удаляемый элемент
        pred.next = current.next;
      }
    } else {                  // переходим к следующему элементу
```

```
      pred = current;
    }
  }
  // Корректируем ссылку на последний элемент...
  last = pred;
  // и выдаем возвращаемое значение
  return found;
}
```

В качестве примера операции вставки элементов в список приведем операцию вставки элемента в упорядоченный список. Функция будет работать правильно, только если элементы в списке расположены в порядке возрастания значений. При вставке элемента тоже придется корректировать ссылку, содержащуюся в элементе, предшествующем вставляемому, поэтому здесь, как и для операции удаления, в функции используются два текущих указателя.

```
public void insert (int n) {
  ListItem pred = null,    // элемент, предшествующий вставляемому
           succ = first;   // элемент, следующий за вставляемым
  while (succ != null && succ.item < n) { // поиск места вставки
    pred = succ;
    succ = succ.next;
  }
  // Генерируем новый элемент:
  ListItem newItem = new ListItem(n, succ);
  if (succ == null) {      // вставляемый элемент — последний
    last = newItem;
  }
  // Вставляем новый элемент в список
  if (pred == null) {
    first = newItem;
  } else {
    pred.next = newItem;
  }
  count++;
}
```

В приведенной функции производится поиск первого элемента, значение которого не меньше аргумента `n`. После того как элемент найден (или обнаружен конец списка), порождается новый элемент списка и вставка произ-

водится с помощью изменения значения поля `next` в предыдущем элементе (или поля `first` списка, если предыдущего элемента нет.

Иногда кроме линейных рассматривают еще и *кольцевые списки*. В кольцевом списке последний элемент содержит указатель на первый. Обработка кольцевых списков не очень отличается от обработки линейных списков, однако нужно аккуратно обрабатывать конец списка. Это требует дополнительных усилий при программировании и несколько замедляет обработку, но зато в кольцевых списках все элементы равноправны, и любой из них может быть "назначен" головным. Такое свойство кольцевого списка может использоваться, например, в алгоритмах обслуживания некоторого множества элемента "в порядке очереди".

Кроме того, в представлении списка можно вообще обойтись только ссылкой на последний элемент, поскольку ссылку на первый элемент можно легко извлечь из последнего элемента (рис. 1.2).

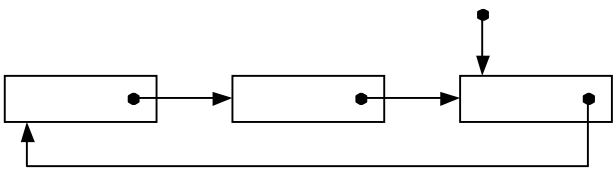


Рис. 1.2. Структура кольцевого списка

В дальнейшем списки будут часто использоваться в качестве составных частей различных структур. При этом элементами списков будут, как правило, не целые числа, как в вышеприведенном примере, а более сложные объекты. Определение класса `IntList` легко обобщить на случай списков из произвольных объектов (`Object`). Разумеется, такие операции, как вставка в упорядоченный список, неприменимы в общем случае, однако большинство других методов легко обобщаются заменой ключевого слова **`int`** на `Object`. В дальнейшем в подобных случаях будем использовать класс `ObjectList` без дополнительного определения этого класса.

Что касается вставки (`insert`) и удаления (`remove`) элементов внутри списка, то обычно эти операции обобщаются следующим образом.

При удалении элементов следует указать, какие элементы необходимо удалить. Это можно сделать двумя способами. Во-первых, можно удалить из списка объекты, содержащие определенный объект, заданный ссылкой на удаляемый. При этом реализация метода остается точно такой же, только операция сравнения целых заменяется операцией сравнения указателей. Во-вторых, можно удалить все объекты, значение которых равно заданному. Для сравнения значений предусмотрен метод `equals` класса `Object`. Этот метод в базовом случае также эквивалентен сравнению указателей, но может

быть переопределен для других классов. Таким образом, второй метод является более общим.

При вставке элементов позицию для вставки можно указывать самыми разными способами. Наиболее распространенными являются следующие:

- вставка перед указанным элементом;
- вставка после указанного элемента;
- вставка в упорядоченный список в соответствии с заданным порядком.

Порядок на множестве объектов некоторого класса считается заданным, если для этих объектов определены операции сравнения элементов. Обычно для сравнения используется метод `compareTo`, определенный в стандартном интерфейсе `java.lang.Comparable`. Таким образом, вставка в упорядоченный список определена, если объекты, содержащиеся в этом списке в качестве элементов, будут удовлетворять интерфейсу `Comparable`. Реализация метода вставки остается практически без изменения, только вместо операции сравнения чисел (<) будет использоваться метод `compareTo`.

Можно считать, что класс `ObjectList` является реализацией следующего простого интерфейса.

```
public interface IList {
  int getCount();                  // число элементов списка
  void addFirst(Object item);      // добавить элемент в начало списка
  void addLast(Object item);       // добавить элемент в конец списка
  Object removeFirst();            // удалить первый элемент
  boolean remove(Object item);     // удалить элементы, равные заданному
  void insertBefore(Object item); // вставить элемент перед заданным
  void insertAfter(Object item);  // вставить элемент после заданного
  void insert(Comparable item);    // вставить элемент в упорядоченный
                                   // список в соответствии с заданным порядком
  Iterator iterator();             // внешний итератор списка
  void iterator(Visitor visitor); // внутренний итератор списка
}
```

Конечно, использование метода вставки в соответствии с порядком элементов в списке будет корректным, только если его элементы действительно упорядочены. Это, в частности, означает, что операции над списком нельзя использовать в произвольном сочетании: скажем, если вы добавили элемент в конец списка, то не следует ожидать, что после этого операция `insert` каким-то образом преобразует список в упорядоченный. В дальнейшем в *разд. 3.3* будут приведены еще один интерфейс и реализация списка, более подходящие для случая упорядоченных списков.

Еще одно замечание. Мы не будем использовать "стандартный" интерфейс `java.util.List` (и, соответственно, его реализации), поскольку на наш взгляд этот интерфейс больше подходит для представления "массивоподобных" (индексируемых) структур. Недаром одной из реализаций указанного интерфейса в пакете `java.util` является "типичный массив" — `java.util.Vector`.

1.3. Деревья

Элементы могут образовывать и более сложную структуру, чем линейный список. Часто данные, подлежащие обработке, образуют иерархическую структуру, подобную изображенной на рис. 1.3, которую необходимо отобразить в памяти компьютера и, соответственно, описать в структурах данных. Каждый элемент такой структуры может содержать ссылки на элементы более низкого уровня иерархии, а может быть, и на объект, находящийся на более высоком уровне иерархии.

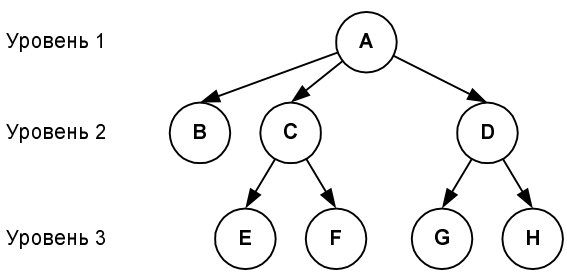


Рис. 1.3. Структура дерева

Если абстрагироваться от конкретного содержания объектов, то получится математический объект, называемый *деревом* (точнее, корневым деревом). Дадим одно из определений корневого дерева.

Корневым деревом называется множество элементов, в котором выделен один элемент, называемый *корнем* дерева, а все остальные элементы разбиты на несколько непересекающихся подмножеств, называемых поддеревьями исходного дерева, каждое из которых, в свою очередь, есть дерево.

При представлении в памяти компьютера элементы дерева (узлы) связывают между собой таким образом, чтобы каждый узел (который согласно определению обязательно является корнем некоторого дерева или поддерева) был связан с корнями своих поддеревьев. Наиболее распространенными способами представления дерева являются следующие три (или их комбинации).

При первом способе каждый узел (кроме корня) содержит указатель на породивший его узел, т. е. на элемент, находящийся на более высоком уровне

иерархии. Для корня дерева соответствующий указатель будет пустым. При таком способе представления, имея информацию о местоположении некоторого узла, можно, прослеживая указатели, подниматься на более высокие уровни иерархии. К сожалению, этот способ представления непригоден, если требуется не только "подниматься вверх" по дереву, но и "спускаться вниз", и нет возможности независимо получать информацию о местоположении узлов дерева. Тем не менее, такое представление дерева иногда используется в алгоритмах, где прохождение узлов всегда осуществляется в восходящем порядке. Преимуществом такого способа представления дерева является то, что в нем используется минимальное количество памяти, причем практически вся она эффективно используется для представления связей.

Второй способ представления применяют, если каждый узел дерева имеет не более двух (в более общем случае — не более K) поддеревьев. Тогда можно включить в представление узла указатели на корни поддеревьев. В этом случае дерево называют *двоичным* (бинарным), а два поддерева каждого узла называют соответственно левым и правым поддеревьями этого узла. Разумеется, узел может иметь только одно — левое или правое — поддерево (эти две ситуации в бинарных деревьях обычно считаются различными) или может вообще не иметь поддеревьев (и в этом случае узел называется концевым узлом или листом дерева). При этом способе представления достаточно иметь ссылку на корень дерева, чтобы получить доступ к любому узлу дерева, спускаясь по указателям, однако память при таком способе представления используется не столь эффективно.

Описание класса бинарного дерева, содержащего произвольные объекты класса `Object` в узлах, может выглядеть, как представлено в листинге 1.5 (методы не показаны).

Листинг 1.5. Определение класса `Tree`

```
public class Tree {
  // Определение класса узла дерева
  public static class Node {
    public Object item;         // содержимое узла
    public Node left = null;    // указатель на левое поддерево
    public Node right = null;   // указатель на правое поддерево

    // Конструкторы узла дерева:
    // конструктор листа
    public Node(Object item) {
      this.item = item;
    }
```

```
    // Конструктор промежуточного узла
    public Node(Object item, Node left, Node right) {
      this.item = item;
      this.left = left;
      this.right = right;
    }
  }

  Node root = null;      // корень дерева
}
```

Здесь корень дерева представлен ссылкой `root` на элемент класса `Node` (узел), а каждый узел, в свою очередь содержит две ссылки на корни левого и правого поддерева (`left` и `right` соответственно).

Если бинарное дерево имеет N узлов, то при таком способе представления всегда остаются пустыми более половины указателей (точнее, из $2N$ указателей пустыми будут $N+1$ указатель, докажите это!). Тем не менее, такое представление является настолько удобным, что потерями памяти обычно пренебрегают.

Третий способ представления состоит в том, что в каждом элементе содержатся два указателя, причем один из них служит для представления списка поддеревьев, а второй для связывания элементов в этот список. Формально описание класса для этого представления может быть тем же самым, что и для случая бинарного дерева, но смысл содержащихся в этом описании указателей меняется. При третьем способе представления деревьев часто используется генеалогическая терминология: узлы, содержащиеся в поддеревьях каждого узла, называются его *потомками*, а корни этих поддеревьев, расположенные на одном уровне иерархии, называют *братьями*. Эту терминологию можно отразить в описании класса, и тогда поля `left` и `right` будут называться `son` и `brother` соответственно.

Рекурсивную природу дерева, отраженную в его определении, можно выразить более явно и в описании класса, если в описании ссылок на поддеревья вместо класса `Node` использовать описатель `Tree`. Тогда и многие операции над деревом могут быть просто выражены в виде рекурсивных функций. Определим, например, метод для вычисления высоты бинарного дерева. *Высотой* бинарного дерева назовем максимальное число узлов, которое может встретиться на пути из корня дерева в некоторый другой узел, при условии, что этот путь проходит только по связанным между собой узлам и никогда не проходит дважды через один и тот же узел.

Будем для удобства считать, что пустой указатель представляет вырожденное "пустое" дерево, высота которого равна нулю. В этом случае высота бинарного дерева может быть выражена следующей формулой:

$$h(t) = \begin{cases} 0, & \text{если } t = null; \\ \max(h(t_{left}), h(t_{right})), & \text{если } t \neq null, \end{cases}$$

где t — исходное дерево, t_{left} и t_{right} — левое и правое поддеревья исходного дерева; *null* — пустое дерево. Тогда определение класса с методом `height`, реализующим вычисление высоты дерева, может выглядеть, как представлено в листинге 1.6.

Листинг 1.6. Рекурсивное определение дерева

```
public class Tree {
  private Object item;
  private Tree left = null;
  private Tree right = null;

  public int height() {
    int hl = (left == null ? 0 : left.height());
    int hr = (right == null ? 0 : right.height());
    return Math.max(hl, hr) + 1;
  }
}
```

Тем не менее, в общем случае удобнее пользоваться ранее приведенным представлением дерева с явным выделением структуры узла. Тогда метод для вычисления высоты дерева в контексте представленного в листинге 1.5 описания класса запишется несколько иначе:

```
public int height() { return height(root); }

private int height(Node n) {
  if (n == null) {
    return 0;
  } else {
    return Math.max(height(n.left), height(n.right)) + 1;
  }
}
```

Здесь описание метода выполнено с помощью вспомогательной рекурсивной функции, аргументом которой является указатель узла того поддерева, высоту которого требуется вычислить.

Иногда структура обрабатывающей дерево рекурсивной функции не так проста. Рассмотрим, например, следующую задачу. Пусть надо определить функцию, вычисляющую число узлов бинарного дерева, расположенных на i-м уровне при заданном значении $i > 0$. Заметим, что в пустом дереве таких вершин нет вовсе, а в непустом дереве требуемое количество узлов можно определить рекурсивно: при $i = 1$ существует лишь одна вершина — корень дерева, а при $i > 1$ число узлов, расположенных на i-м уровне дерева, равно сумме количества узлов, расположенных в его поддеревьях (левом и правом) на $(i - 1)$-м уровне.

Отсюда сразу же следует определение функции (считаем его также выполненным в контексте описания класса `Tree`, приведенного в листинге 1.5).

```
public int nodesOnLevel(int level) {
  return nodesOnLevel(root, level);
}

private int nodesOnLevel(Node n, int level) {
  if (n == null) return 0;
  if (level == 1) return 1;
  return nodesOnLevel(n.left, level-1) +
         nodesOnLevel(n.right, level-1);
}
```

Так же, как и в случае списков, важной проблемой является итерация (обход) дерева. Если в случае списков обычно обход выполняется в естественном порядке (от начала списка к концу), в крайнем случае, можно рассмотреть еще обход элементов списка в противоположном направлении — от конца к началу, то в случае дерева существует много порядков обхода его узлов, большинство из которых имеет свое самостоятельное значение и применяется в различных алгоритмах. Подробнее эта проблема обсуждается в *разд. 2.3*, а также в книгах [2, 5] и др. Здесь для примера приведем лишь один способ обхода дерева — левосторонний. Для реализации обхода определим внутренний итератор. При левостороннем обходе сначала полностью обходится левое поддерево исходного дерева (также в левостороннем порядке, разумеется, если это дерево не пусто), затем проходится корень дерева, а затем правое поддерево также обходится в левостороннем порядке.

Как и в случае списков сначала напишем интерфейс для определения посещения узла дерева.

```
public interface Visitor {
  void visit(Object item);
}
```

Заметим, что фактически это тот же самый интерфейс, который использовался нами для "посещения" элементов списка в *разд. 1.2*. Теперь собственно операция обхода может быть определена с помощью рекурсивной функции, например, следующим образом.

```
public void left2right(Visitor visitor) {
  left2right(root, visitor);
}

private void left2right(Node n, Visitor visitor) {
  if (n != null) {
    left2right(n.left, visitor);
    visitor.visit(n.item);
    left2right(n.right, visitor);
  }
}
```

Применим, например, функцию обхода дерева с помощью внутреннего итератора для подсчета числа его узлов. В этом случае аргумент `visitor` должен лишь добавлять единицу к общему количеству уже сосчитанных узлов. Для этого определим дополнительно класс `Counter`, реализующий интерфейс `Visitor`.

```
public class Counter implements Visitor {
  int counter = 0;
  public void visit(Object o) { counter++; }
  public int getCounter() { return counter; }
}
```

Вот как теперь будет выглядеть программа для заданного дерева `tree`.

```
public static void main (String[] args) {
  Tree tree = new Tree();
  /* Где-то здесь формируется дерево tree... */
  Counter counter = new Counter();
  tree.left2right(counter);  // обход дерева с посещением всех его узлов
  System.out.println("Всего узлов в дереве: " + counter.getCounter());
}
```

Иногда используются и более экзотические способы представления деревьев. Например, в некоторых случаях связи между узлами дерева можно вообще не представлять с помощью указателей, а "вычислять", если узлы дерева образуют регулярную структуру, не меняющуюся или меняющуюся очень мало в процессе работы с деревом.

Рассмотрим пример. Пусть узлы бинарного дерева пронумерованы, начиная с корня и далее вниз по иерархическим уровням, как показано на рис. 1.4.

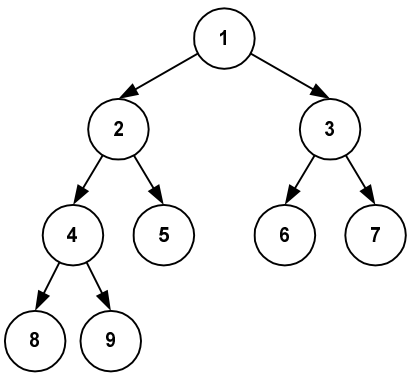


Рис. 1.4. Дерево с узлами, пронумерованными по уровням сверху вниз

При этом узлы расположены настолько плотно, что предком узла с номером `i` всегда является узел с номером `i/2`. В этом случае узлы дерева можно расположить в массиве, причем местоположение каждого узла будет задано его индексом в массиве, а максимальное число узлов указывается сразу же при создании дерева. Представление такого дерева описано в листинге 1.7. Дерево, определенное в этом листинге, обладает следующим свойством: каждый узел дерева содержит целое число, меньшее, чем значения, хранящиеся в поддеревьях этого узла. Таким образом, минимальное число всегда находится в корне дерева. Такое дерево иногда называют *"пирамидой"*.

Класс `Heap` определяет такое дерево вместе с операцией `insert` добавления нового значения в дерево. Сначала новое значение добавляется в качестве последнего узла дерева, но если оно оказывается меньше, чем значения, расположенные выше по дереву, то оно "всплывает" наверх до тех пор, пока не окажется минимальным среди всех узлов поддерева, в корне которого это значение находится.

В процессе работы операция `insert` для каждого узла определяет индекс его "родителя" и затем выясняет, надо ли продвигать новое значение дальше по направлению к вершине "пирамиды". Для сравнения двух элементов класса `Integer` используется операция `compareTo`, определенная для любых объектов, которые можно сравнивать по величине друг с другом (точнее, для объ-

ектов, класс которых реализует интерфейс `Comparable`). Выражение `obj1.compareTo(obj2)` будет:

- меньше нуля, если объект `obj1` "меньше" объекта `obj2`;
- равно нулю, если объекты равны;
- больше нуля, если объект `obj1` "больше" объекта `obj2`.

Разумеется, для объектов класса `Integer` результат такого сравнения совпадает с результатом обычного сравнения значений.

Листинг 1.7. Представление "пирамиды" в виде массива

```
public class Heap {
  int count = 0;      // число узлов дерева
  Object [] items;    // массив, представляющий узлы дерева

  // Конструктор пирамиды заданного максимального размера
  public Heap(int size) {
    items = new Object[size];
  }

  // Операция вставки нового элемента в пирамиду
  public void insert(int n) throws NoMoreSpace {
    // Проверим, есть ли еще место в массиве
    if (count >= items.length) {
      throw new NoMoreSpace();
    }
    // Создаем новый элемент
    Integer newItem = new Integer(n);
    int curIndex = count++;          // индекс текущей позиции в пирамиде
    int parentIndex;                 // индекс "родителя"
    // Цикл "всплытия" элемента
    while (curIndex > 0 && newItem.compareTo
                                (items[parent = curIndex / 2]) < 0) {
      // Сдвигаем элемент вниз...
      items[curIndex] = items[parent];
      // и переходим к элементу, лежащему выше в пирамиде
      curIndex = parent;
    }
    // Помещаем новый элемент на свое место
```

```
    items[curIndex] = newItem;
  }
}
```

Подробнее о построении и анализе деревьев будет рассказано в *разд. 2.3* и *2.4*.

1.4. Множества

Множество — это составной объект, который так же, как и массив, содержит компоненты одного и того же класса. Отличие от массива состоит в том, что над множеством определены совсем другие операции, и это определяет существенную разницу в представлении объекта. В массиве элементы упорядочены, доступ к ним осуществляется по индексу; собственно говоря, индексация — это практически единственная операция над массивом. Напротив, в множестве порядок элементов несущественен. Основная операция — это проверка принадлежности элемента множеству. Другие операции — это теоретико-множественные операции объединения и пересечения множеств, добавление элементов в множество, определение числа элементов (мощность) множества. Конечно, для массива тоже можно определить подобные операции, другими словами, можно представлять множество массивом, однако, в этом случае типичные операции над множествами будут выполняться недопустимо долго. Так, например, проверка принадлежности элемента массиву обычно реализуется с помощью просмотра, вообще говоря, всех его элементов.

Эффективная реализация множества обычно подразумевает такое его представление, при котором элементы множества не хранятся, вместо этого хранится лишь информация о том, содержится ли элемент в множестве. Это возможно в том случае, если элементы множества достаточно просты, а максимально возможное число элементов множества не слишком велико. Тогда можно каждый из возможных элементов множества кодировать одним битом, например, единицей, если элемент присутствует в множестве, и нулем, если элемент в множестве отсутствует. Наиболее частые случаи использования множеств — это множество целых чисел из некоторого диапазона и множество символов из некоторого набора символов. В обоих случаях для представления множества можно сформировать битовую шкалу, т. е. последовательность двоичных элементов, каждый из которых определяет, присутствует ли в множестве некоторый элемент. Такая битовая шкала будет содержать столько битов, сколько элементов находится в выбранном диапазоне возможных элементов.

Например, если в программе нужно работать с множествами целых чисел из диапазона от 0 до 10, то можно каждое множество представлять набором из 11 битов (пронумерованных числами от 0 до 10), причем если бит содержит единицу, то это означает, что его номер входит в множество в качестве элемента, а если ноль, то соответствующий элемент в множестве отсутствует.

Таким образом, битовая шкала 00101100010 (биты считаются пронумерованными подряд от 0 до 10) в данном случае представляет множество элементов [2, 4, 5, 9].

Еще пример. Пусть мы хотим работать с множествами строчных русских букв. Прежде всего следует закодировать все эти буквы целыми числами. Наиболее экономным способом кодирования будет такой, при котором все 33 символа получат коды из диапазона (0, 32). Например, в расширенной кодировке Win-1251 коды русских строчных букв расположены в диапазоне от 1072 до 1105 (исключая код 1104). Этот диапазон можно легко привести к диапазону (0, 32), если выполнить следующее преобразование: вычтем 1072 из кода символа; если получится число 33, то дополнительно вычтем единицу. При таком способе кодирования гласные буквы `а`, `е`, `ё`, `и`, `о`, `у`, `ы`, `э`, `ю`, `я` будут иметь коды 0, 5, 32, 8, 14, 19, 27, 29, 30, 31 соответственно. Тогда битовая шкала, представляющая множество всех гласных букв, выглядит следующим образом:

10000100100000100001000000101111

В языке Java битовая шкала может быть представлена массивом байтов, каждый элемент которого (байт) содержит восемь битовых элементов шкалы. Биты удобно нумеровать целыми числами, начиная с нуля, причем бит номер *n* будет содержаться в байте с индексом *n*/8 (имеется в виду операция целочисленного деления с отбрасыванием остатка). Например, приведенная выше шкала для представления множества гласных букв займет 5 байтов (если бы букв всего было 32, то удалось бы поместиться в 4 байта). Учитывая, что биты в байте принято нумеровать справа налево (то есть при изображении байта его младшие биты располагают справа), можно нарисовать ту же битовую шкалу в виде последовательности байтов, представленной на рис. 1.5.

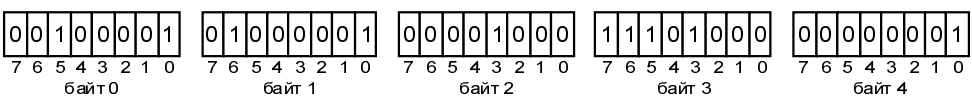


Рис. 1.5. Битовая шкала, представленная последовательностью байтов

Ниже (листинг 1.8) приводится определение класса для представления множества целых чисел из заданного диапазона. Битовая шкала в этой реализации представлена массивом байтов, а операции над битами реализуются с помощью побитовых операций. Операция побитового сложения (`|`) в данном случае представляет операцию объединения множеств, а операция побитового умножения (`&`) — операцию пересечения множеств. С помощью операций побитового сдвига получают байт с единицей, расположенной в нужном разряде. Вот, например, как в заданный байт `bt` можно добавить единичный бит, записанный в разряде номер `n`:

```
bt |= (1 << n);
```

Выражение для проверки, содержит ли заданный байт единицу в разряде номер `n`, будет выглядеть следующим образом:

```
(bt & (1 << n)) != 0
```

Аналогичным образом будут представлены и другие операции над битовыми шкалами и отдельными битами.

Листинг 1.8. Определение класса `Set`

```
public class Set {
  int minElem;    // минимальный элемент диапазона
  int maxElem;    // максимальный элемент диапазона
  byte[] elems;   // битовая шкала

  //=========================================
  // Конструктор
  //=========================================
  public Set(int min, int max) {
    // обеспечим min < max
    if (min > max) {
      minElem = max;
      maxElem = min;
    } else {
      minElem = min;
      maxElem = max;
    }
    int num = maxElem - minElem + 1;      // количество битов
    int numBytes = (num + 7) / 8;         // количество байтов
    elems = new byte[numBytes];
  }

  //=========================================
  // Проверка принадлежности элемента множеству
  //=========================================
  public boolean has(int n) throws IndexOutOfBoundsException {
    if (n > maxElem || n < minElem)
      throw new IndexOutOfBoundsException(
        "Элемент " + n + " лежит за границами диапазона [" +
        minElem + ", " + maxElem + "]");
```

```
    int bit = n - minElem;
    return (elems[bit / 8] & (1 << (bit % 8))) != 0;
  }

  //=========================================
  // Добавление элемента к множеству
  //=========================================
  public Set disjunct(int n) throws IndexOutOfBoundsException {
    if (n <= maxElem && n >= minElem) {
      int bit = n - minElem;
      elems[bit / 8] |= (1 << (bit % 8));
    } else {
      throw new IndexOutOfBoundsException(
        "Элемент " + n + " лежит за границами диапазона [" +
        minElem + ", " + maxElem + "]");
    }
    return this;
  }

  //=========================================
  // Добавление множества к множеству
  //=========================================
  public Set disjunct(Set other) throws IndexOutOfBoundsException {
    if (other.minElem != minElem || other.maxElem != maxElem) {
      throw new IndexOutOfBoundsException("Несопоставимые множества");
    }
    for (int i = 0;  i < elems.length;  i++) {
      elems[i] |= other.elems[i];
    }
    return this;
  }

  //=========================================
  // Пересечение множества с множеством
  //=========================================
  public Set conjunct(Set other) throws IndexOutOfBoundsException {
    if (other.minElem != minElem || other.maxElem != maxElem) {
      throw new IndexOutOfBoundsException("Несопоставимые множества");
    }
```

```
    for (int i = 0;  i < elems.length;  i++) {
      elems[i] &= other.elems[i];
    }
    return this;
  }

  //=========================================
  // Удаление элемента из множества
  //=========================================
  public Set remove(int n) throws IndexOutOfBoundsException {
    if (n <= maxElem && n >= minElem) {
      int bit = n - minElem;
      elems[bit / 8] &= ~(1 << (bit % 8));
    } else {
      throw new IndexOutOfBoundsException(
        "Элемент " + n + " лежит за границами диапазона [" +
        minElem + ", " + maxElem + "]");
    }
    return this;
  }

  //=========================================
  // Удаление множества из множества
  //=========================================
  public Set remove(Set other) throws IndexOutOfBoundsException {
    if (other.minElem != minElem || other.maxElem != maxElem) {
      throw new IndexOutOfBoundsException("Несопоставимые множества");
    }
    for (int i = 0;  i < elems.length;  i++) {
      elems[i] &= ~other.elems[i];
    }
    return this;
  }

  //=========================================
  // Обращение (нахождение дополнения) множества
  //=========================================
  public Set inverse() {
```

```
    for (int i = 0;  i < elems.length;  i++) {
      elems[i] = (byte)~elems[i];
    }
    return this;
  }

  //=============================================
  // Создание копии множества
  //=============================================
  public static Set copy(Set s) {
    Set result = new Set(s.minElem, s.maxElem);
    for (int i = 0;  i < result.elems.length;  i++) {
      result.elems[i] = s.elems[i];
    }
    return result;
  }

  //=============================================
  // Объединение множеств
  //=============================================
  public static Set or(Set s1, Set s2)
      throws IndexOutOfBoundsException {
    if (s1.minElem != s2.minElem || s1.maxElem != s2.maxElem) {
      throw new IndexOutOfBoundsException("Несопоставимые множества");
    }
    return copy(s1).disjunct(s2);
  }

  //=============================================
  // Пересечение множеств
  //=============================================
  public static Set and(Set s1, Set s2)
      throws IndexOutOfBoundsException {
    if (s1.minElem != s2.minElem || s1.maxElem != s2.maxElem) {
      throw new IndexOutOfBoundsException("Несопоставимые множества");
    }
    return copy(s1).conjunct(s2);
  }
```

```
  //=========================================
  // Разность множеств
  //=========================================
  public static Set diff(Set s1, Set s2)
      throws IndexOutOfBoundsException {
    if (s1.minElem != s2.minElem || s1.maxElem != s2.maxElem) {
      throw new IndexOutOfBoundsException("Несопоставимые множества");
    }
    return copy(s1).remove(s2);
  }

  //=========================================
  // Дополнение множества
  //=========================================
  public static Set not(Set s) {
    return copy(s).inverse();
  }
}
```

В приведенном классе определен достаточно богатый набор операций над множествами, однако для практических целей может понадобиться еще несколько расширить и модифицировать этот класс, например, чтобы можно было работать с множествами элементов из разных диапазонов. В приведенном выше классе при попытке, например, объединить множество чисел, лежащих в диапазоне (0, 255), с множеством чисел из диапазона (10, 25) возбуждается исключительная ситуация `IndexOutOfBoundsException`, хотя по смыслу это, конечно, совершенно законная операция.

Еще одним удобным дополнением мог бы быть набор операций для выдачи всех элементов множества, например, в виде массива или в виде итератора элементов. В дальнейшем мы будем использовать те операции над множествами, какие окажутся наиболее подходящими в конкретной ситуации, без дополнительного определения.

1.5. Графы

В процессе обработки данных на компьютере часто приходится моделировать объекты сложной структуры, содержащие в качестве составных частей (элементов) объекты более простой структуры. Иногда такой сложный объект состоит из некоторого множества элементов одного и того же типа, между которыми существуют определенные связи (отношения). В технике такие объекты часто называют сетями. Приведем несколько примеров.

Пусть моделируемым объектом является транспортная сеть для перевозки определенных грузов. Элементами (или узлами) такой сети служат начальные, конечные и транзитные пункты перевозок, а в качестве связей между пунктами выступают дороги. Узлы сети могут характеризоваться индивидуальным именем (названием), объемом принимаемой или поставляемой продукции. Дороги могут характеризоваться длиной, пропускной способностью, качеством покрытия и т. д.

Для расчета характеристик сети компьютеров создается ее модель. Узлами такой сети служат отдельные компьютеры, связанные между собой каналами связи. Каждый узел может характеризоваться интенсивностью выдачи и приема сообщений, объемом предоставляемой памяти. Канал связи может характеризоваться скоростью передачи информации, количеством одновременно передаваемых сообщений и т. д.

Программа игры в шахматы для анализа позиции строит сеть, состоящую из позиций, связанных между собой возможными ходами соперников. Узлы сети могут характеризоваться игровой оценкой позиции, а связывающие позиции ходы — локальными целями, и т. д.

Для моделирования сетей в математике служат объекты, называемые *графами*. Графом G называется пара множеств (V, E), где V — конечное множество элементов, называемых *вершинами* графа, а E — конечное множество упорядоченных пар $e = (A, B)$, называемых *дугами*, где A, B — вершины.

Говорят, что дуга e выходит из вершины A и входит в вершину B. Вершины A и B называют инцидентными дуге e, а дугу e — инцидентной вершинам A и B.

В этом определении множество вершин V соответствует множеству узлов моделируемой сети, а множество дуг — связям между узлами. Определение, данное выше, отражает лишь структуру сети, но не характеристики ее отдельных узлов и связей. Если такие характеристики все же существенны, то рассматривают нагруженные графы, в которых с каждой вершиной или дугой (может быть, и с тем, и с другим) связана величина или несколько величин, называемых нагрузкой на граф. Формально говоря, нагрузку графа определяют функции

$$f\colon V \to W_1 \text{ и } g\colon E \to W_2,$$

где W_1 и W_2 представляют собой множества значений нагрузки вершин и дуг графа соответственно. Иногда при анализе графа неважно, какая из вершин A и B в дуге $e = (A, B)$ первая, а какая вторая, т. е. пара (A, B) не упорядочена. В этом случае дугу e называют ребром, а весь граф называют неориентированным в отличие от ориентированного графа, задаваемого исходным определением. Формально говоря, неориентированным графом называют такой граф, у которого наряду с любой дугой $e_1 = (A, B)$ имеется и противоположная дуга $e_2 = (B, A)$. Эта пара дуг и образует ребро $e = \langle A, B \rangle$ неориентированного графа.

При программировании задач обработки сетевых структур требуется решить вопрос о представлении графа структурами данных языка программирования. Выбор представления графа определяется прежде всего тем, какие алгоритмы обработки графов используются, а также соображениями экономии памяти при обработке очень больших графов или в условиях жесткого лимита памяти.

Ниже приводится несколько способов представления графов, причем для каждого способа указано, для каких алгоритмов он подходит, а также дана приблизительная оценка занимаемой памяти. Везде далее считается, что число вершин графа $N = card(V)$ и число дуг (или ребер) графа $M = card(E)$ — величины постоянные.

Структуру графа можно описать, сопоставив каждой вершине множество дуг, выходящих из нее, причем каждая дуга, выходящая из вершины A, идентифицируется своим концом — номером вершины, в которую эта дуга входит. Описанное представление графа будем называть S-графом (от англ. *set* — множество). Будем считать, что множество целых чисел в диапазоне от 0 до N представлено объектом класса `Set` из *разд. 1.4*. Тогда S-граф может быть описан следующим классом (листинг 1.9).

Листинг 1.9. Определение S-графа

```
public class SGraph {
  Set[] graph;

  public SGraph(int n) {
    graph = new Set[n];
    for (int i = 0;  i < n;  i++) {
      graph[i] = new Set(0, n-1);
    }
  }
}
```

Принадлежность дуги (A, B) графу может быть легко проверена с помощью следующего метода.

```
  public boolean has(int a, int b) throws IndexOutOfBoundsException {
    return graph[a].has(b);
  }
```

При задании неправильных номеров вершин исключительная ситуация `IndexOutOfBoundsException` возникнет либо из-за неправильного индекса в массиве, либо из-за возбуждения ее методом `has` класса `Set`.

Легко описать методы для добавления или удаления дуги при таком представлении графа, достаточно лишь добавить или удалить определенный элемент в соответствующем множестве:

```
public void add(int u, int v) {
  graph[u].disjunct(v);
}

public void remove(int u, int v) {
  graph[u].remove(v);
}
```

Несколько сложнее решается задача подсчета числа дуг, инцидентных данной вершине. Если считать, что класс `Set` содержит метод `card` для подсчета числа элементов множества (мощность множества), то число выходящих из данной вершины дуг легко получить с помощью выражения `graph[a].card()`, однако число входящих в вершину дуг может быть получено только путем перебора всех вершин графа:

```
public int cardIn(int v) {
  int s = 0;
  for (int w = 0;  w < graph.length;  w++) {
    if (has(w, v)) s++;
  }
  return s;
}
```

Еще один распространенный способ представления графа — это представление в виде матрицы смежности размером $N \times N$. В этой матрице в элементе с индексами (i, j) записывается информация о дугах, ведущих из вершины с номером i в вершину с номером j. В важном частном случае простого графа, т. е. в котором каждая упорядоченная пара вершин соединена не более, чем одной дугой, и отсутствуют дуги вида (A, A), элементы матрицы могут принимать значения 0 или 1 в зависимости от того, существует ли соответствующая дуга. Ясно, что такое представление самым тесным образом связано с представлением, описанным выше в определении класса `SGraph`.

Представление в виде матрицы смежности будем называть M-графом. Удобно связывать с матрицей смежности информацию о нагрузке на дуги. В этом случае элементом матрицы будет либо значение нагрузки на соответствующую дугу, либо некоторое стандартное значение, говорящее об отсутствии дуги. Если, например, граф моделирует транспортную сеть, в которой нагрузка на дугу представляет расстояние между пунктами, соединенными этой дугой, то признаком отсутствия дуги удобно сделать значение 0 или, напротив, максимально возможное целое (`Integer.MAX_VALUE`), смотря по тому, что окажется более удобным в используемом алгоритме. В общем ви-

де, если тип нагрузки представлен значениями класса *W*, то класс, представляющий *M*-граф, может быть описан следующим образом.

Листинг 1.10. Определение *M*-графа

```
public class MGraph {
  W graph[][];

  public MGraph(int n) {
    graph = new W[n][n];
  }

  public void add(int u, int v) {
    graph[u][v] = 1;
  }

  public void remove(int u, int v) {
    graph[u][v] = 0;
  }
}
```

Поскольку представление *M*-графа очень близко к представлению *S*-графа, то и операции над *M*-графом выполняются практически с той же степенью эффективности, что и для *S*-графа. Удобными и эффективно реализуемыми операциями над *M*-графом являются операции проверки наличия дуги и значения ее нагрузки, добавления и удаления дуг, изменение нагрузки.

Менее эффективными операциями будут операции подсчета или перебора дуг, инцидентных заданной вершине. Между тем, во многих алгоритмах именно это является самой часто используемой операцией. Структура графа может меняться крайне редко, а вот исследоваться эта структура будет достаточно часто. В этом случае представление графа в виде матрицы смежности будет неудобным.

Еще одно соображение. Если число вершин графа велико, то матрица смежности будет занимать достаточно много места. В то же время в графе количество дуг чаще всего существенно меньше $N \times N$, так что большинство элементов матрицы смежности будут пустыми. В этом случае более удобным будет представление, в котором с каждой вершиной графа связывается список исходящих из нее дуг. Каждая дуга может быть представлена номером вершины, в которую эта дуга входит, а в случае нагруженного графа может также содержать значение нагрузки на дугу.

Соответствующее представление будем называть *L*-графом (от англ. *list* — список). В листинге 1.11 приведено возможное описание класса для ненагруженного *L*-графа.

Листинг 1.11. Определение *L*-графа

```
public class LGraph {
  // Класс Arc представляет дугу, ведущую в узел end
  static class Arc {
    int end;       // номер узла, в который входит эта дуга
    public Arc(int e) { end = e; }
  };

  ObjectList[] graph;    // массив списков дуг

  public LGraph(int n) {
    graph = new ObjectList[n];
    for (int i = 0;  i < graph.length;  i++) {
      graph[i] = new ObjectList();
    }
  }
}
```

В этом определении используется класс `ObjectList` (см. замечание относительно списков в конце *разд. 1.2*). Будем считать, что элементами этого списка будут объекты класса `Arc`, определенного выше внутри класса `LGraph`. Например, чтобы добавить дугу (*u*, *v*) в граф, надо добавить в список дуг, выходящих из вершины *u*, дугу, входящую в вершину *v*.

```
public void add(int u, int v) {
  graph[u].addLast(new Arc(v));
}
```

Удаление дуги при таком представлении не удается реализовать столь же просто. Для этого надо просмотреть соответствующий список, найти в нем нужную дугу и удалить ее из списка. Впрочем, если определить операцию сравнения дуг на равенство (`equals`), то можно воспользоваться ею и для удаления нужной дуги:

```
static class Arc {
  int end;       // номер узла, в который входит эта дуга
  public Arc(int e) { end = e; }
  public boolean equals(Arc arc) {
    return arc.end == this.end;
  }
};
```

```
public void remove(int u, int v) {
  graph[u].remove(new Arc(v));
}
```

Представление графа в виде совокупности списков дуг становится неудобным в тех случаях, когда наиболее часто используются операции проверки наличия или модификации конкретной дуги, т. е. как раз те операции, которые наиболее эффективно реализуются для *S*-графов и *M*-графов.

Иногда используются и другие представления графов, например, для случая очень разреженных графов, когда при большом количестве N вершин графа число дуг не только существенно меньше $N \times N$, но и меньше N. Одним из таких представлений является представление в виде списка дуг, когда все дуги графа собраны в единый список, в каждом элементе которого записана информация об обоих концах дуги, а также, возможно, о нагрузке на дугу. Другим возможным представлением графа является матрица инцидентности, содержащая N строк и M столбцов, в которой элемент с индексами (i, j) равен единице, если i-я вершина инцидентна j-му ребру, и нулю — в противном случае. В последнем примере речь идет о неориентированном графе, поскольку в описанном представлении матрицы инцидентности не содержится информации о направленности дуги, однако это представление легко модифицировать и для случая ориентированного графа.

В заключение этой главы приведем функции преобразования, которые могли бы использоваться для изменения представления графа. Каждая из этих функций требует времени, пропорционального $N \times N$ или M. Это означает, что если время работы некоторого алгоритма на графе имеет порядок сложности не меньше $N \times N$, то скорость его работы практически не зависит от способа представления графа, поскольку всегда можно перед началом работы выполнить нужное преобразование без существенной потери эффективности основного алгоритма.

Каждая из функций считается определенной внутри того класса, который представляет исходный граф, а для построения графа в целевом представлении используются методы добавления дуг, определенные в классе результирующего графа.

Листинг 1.12. Функции преобразования представлений графов

```
public class LGraph {
  /*
    Функция строит S-граф по заданному значению L-графа
  */
  public SGraph toSGraph() {
    SGraph g = new SGraph(graph.length);
    for (int n = 0;  n < graph.length;  n++) {
```

```
      for (Iterator i = graph[n].iterator();  i.hasNext(); ) {
        Arc nextArc = (Arc)i.next();
        g.add(n, nextArc.end);
      }
    }
    return g;
  }
}

public class SGraph {
  /*
    Функция строит M-граф по заданному значению S-графа
  */
  public MGraph toMGraph() {
    MGraph g = new MGraph(graph.length);
    for (int n = 0;  n < graph.length;  n++) {
      for (Iterator i = graph[n].iterator();  i.hasNext(); ) {
        g.add(n, ((Integer)i.next()).intValue());
      }
    }
    return g;
  }
}

public class MGraph {
  /*
    Функция строит L-граф по заданному значению M-графа
  */
  public LGraph toLGraph() {
    LGraph g = new LGraph(graph.length);
    for (int i = 0;  i < graph.length;  i++) {
      for (int j = 0;  j < graph.length;  j++) {
        if (graph[i][j] == 1)
          g.add(i, j);
      }
    }
    return g;
  }
}
```

Глава 2

Базовые алгоритмы

В этой главе рассматривается несколько важных структур данных и операции над ними. Сначала вводятся понятия стека и очереди и приводятся примеры работы с ними. Затем рассматриваются способы прохождения деревьев.

2.1. Абстрактные типы данных

В предыдущей главе мы говорили о способах представления структур данных. Однако не менее важным является вопрос о способах доступа к элементам этих структур. Конечно, часто именно способ представления определяет методы доступа к элементам, но иногда можно ввести методы доступа независимо от представления структуры и рассматривать различные способы реализации этих методов для различных представлений структуры. Действительно, нам не так уж важно, как представлены, скажем, вещественные числа в памяти в конкретной реализации языка Java, однако для нас существенно, что над ними можно выполнять арифметические операции, которые подчиняются определенным точно заданным законам. В этом смысле операции над вещественными числами не зависят (ну, скажем, почти не зависят) от конкретного представления, хотя, разумеется, реализация этих операций будет использовать конкретное представление чисел.

Когда хотят отделить понятие об операциях от конкретного представления тех или иных данных, говорят, что задается *абстрактный тип данных*. Абстрактный тип данных можно задать, если перечислить все операции, разрешенные над значениями этого типа, и определить правила, которым должны удовлетворять эти операции. Часто такие правила задаются с помощью специальных уравнений или аксиом, мы, однако, будем ограничиваться словесным описанием правил, которым должны удовлетворять такие операции.

В качестве примера абстрактного типа данных рассмотрим абстрактную систему распределения памяти. Такая система обладает определенным ресурсом оперативной памяти (пулом) и способна в пределах этого пула выделять участки памяти для использования в программе, а при возврате этих участков пользователем снова возвращать их в пул. Будем считать, что система распределения памяти представлена объектом, над которым разрешены две основные операции — выделение участка памяти и освобождение участка

памяти. Смысл первой операции состоит в том, что система распределения памяти находит свободный участок памяти заданного размера и выдает его в виде блока памяти. Смысл второй операции заключается в том, что ранее выделенный блок памяти возвращается в систему.

Будем считать, что выделяемые блоки памяти — это последовательности байтов, представленные, в свою очередь, абстрактным типом данных `Block`. Каждый блок имеет определенную длину (количество байтов в блоке), а доступ к байтам блока обеспечивается с помощью операций `elementAt` и `setElementAt`, которые могут прочитать или записать байт в произвольное место этого блока.

Итак, определим следующие интерфейсы для описанных нами типов данных.

Листинг 2.1. Определение интерфейсов для системы распределения памяти

```
public interface Storage {
  Block getBlock(int size);
  void putBlock(Block block);
}

public interface Block {
  int size();
  byte elementAt(int index);
  void setElementAt(int index, byte value);
}
```

Эти интерфейсы определяют абстрактные типы данных `Storage` и `Block`. Разумеется, для того чтобы воспользоваться системой распределения памяти, необходимо реализовать эти интерфейсы, представив классы, которые содержат такую реализацию. Реализации могут быть различными, однако, если в программе используются только операции из заданного интерфейса, то такая программа не зависит от конкретной реализации.

Существует множество различных методов распределения памяти, так что можно представить себе множество различных реализаций описанных интерфейсов. Одно из самых лучших описаний различных методов распределения памяти содержится в книге [5]. Мы не будем приводить здесь ни одну из реализаций, поскольку даже наиболее простые из этих методов потребуют значительного места для воспроизведения и пояснений, не относящихся напрямую к теме данной книги.

Итак, задав интерфейсы системы распределения памяти, мы определили два абстрактных типа данных — тип данных `Block` и тип данных `Storage`. Теперь про любую реализацию этих интерфейсов в виде некоторых классов

можно сказать, что это конкретная реализация абстрактных типов данных, т. е. конкретный тип (или, если придерживаться терминологии Java, — класс) данных.

В языке Java существует еще один способ выражения понятия абстрактного типа данных — определение абстрактного класса. В основном абстрактный класс нужен, когда хотят задать не только номенклатуру методов, но и частично определить реализацию некоторых из них. Для нас сейчас различие между интерфейсом и абстрактным классом несущественно.

Чаще всего с помощью определений интерфейсов и абстрактных классов действительно выражают понятие абстрактного типа данных. Например, стандартный интерфейс `java.util.Set` безусловно выражает общую идею абстрактных множеств с обычными для множеств операциями проверки принадлежности элемента множеству, объединения, пересечения, разности и др. Таким образом, можно сказать, что интерфейс `java.util.Set` представляет собой абстрактный тип данных "множество", и его реализации (стандартные или написанные для конкретной программы) являются различными конкретными реализациями множеств. На самом деле часто классы, реализующие некоторый интерфейс, вносят дополнительные методы, так что такие классы уже не могут считаться реализацией именно этого абстрактного типа данных. В этом случае, скорее, нужно говорить о реализации некоторого типа данных, совместимого с исходным абстрактным типом.

Иногда определение интерфейса задает лишь некоторый набор операций, которым должны удовлетворять реализации этого интерфейса, но с логической точки зрения эти операции не образуют систему методов, задающую абстрактный тип данных. Например, стандартный интерфейс `java.util.Iterator`, который мы активно использовали для реализации внешних итераторов, трудно рассматривать как определение абстрактного типа данных. Действительно, та тройка методов — `hasNext`, `next` и `remove`, — которая и составляет этот интерфейс, не образует системы методов, определяющих способы работы с объектами. Скорее интерфейс задает ту общую часть всех реализующих данный интерфейс классов, которая отвечает за перечисление элементов некоторой структуры.

Таким образом, мы ни в коем случае не хотим поставить знак равенства между понятиями абстрактного типа данных и интерфейсами языка Java. Но можно сказать, что абстрактные типы данных чаще всего выражены в языке Java с помощью интерфейсов.

Если в программе можно использовать как абстрактные типы данных, так и конкретные реализации, то предпочтение следует отдавать использованию абстрактных типов. Если методы некоторого класса используют только абстрактный интерфейс с объектами других классов, то реализация такого класса становится более общей и независимой от реализаций других классов.

Пусть, например, в программе учета товаров на складе обрабатываются различные множества товаров, причем используется большинство или даже все операции с множествами, определенные в интерфейсе `Set`. Пусть класс `SetOfGoods` реализует в программе работу с множествами товаров. Тогда лучше всего, если класс `SetOfGoods` будет определен как реализация интерфейса `Set`.

```
class SetOfGoods implements java.util.Set { ... }
```

Далее, везде, где можно, следует использовать не описатель класса `SetOfGoods`, а описатель интерфейса `Set`. Например, если некоторая функция получает множество товаров в качестве аргумента и в процессе обработки использует только операции интерфейса `Set`, то ее аргумент следует описывать не как аргумент класса `SetOfGoods`, а как аргумент с интерфейсом `Set`. Конечно, с точки зрения исполнения программы, эффективности реализации и так далее совершенно безразлично, как выглядит функция — так:

```
void printGoods(SetOfGoods set) {
  for (Iterator i = set.iterator(); i.hasNext();) {
    System.out.println(i.next());
  }
}
```

или так:

```
void printGoods(Set set) {
  for (Iterator i = set.iterator(); i.hasNext();) {
    System.out.println(i.next());
  }
}
```

поскольку в любом случае используется одна и та же операция над аргументом — вызов внешнего итератора `set.iterator()`, но в дальнейшем вторая реализация дает более широкие возможности для модернизации. Во-первых, функция `printGoods` оказывается применимой для любых множеств, лишь бы только элементы этих множеств обеспечивали разумный способ внешнего представления в виде строки. Во-вторых, никакие изменения в определении класса `SetOfGoods`, даже изменение названия класса, не затрагивают реализации функции `printGoods`, пока класс `SetOfGoods` обеспечивает совместимость с абстрактным типом `Set`.

2.2. Стеки и очереди

Покажем, как принцип абстрактных типов данных применяется для описания таких важных для программирования понятий, как стеки и очереди. В соответствии с этим принципом будем вводить новые типы данных, перечисляя и описывая не столько структуру объектов, сколько операции над

ними. Разумеется, в конкретных приложениях будет необходимо предложить какую-либо конкретную структуру объектов. Мы тоже предложим несколько способов реализации стеков и очередей, впрочем, разумеется, не исчерпывающих весь спектр возможных реализаций. Тем не менее, независимо от способа представления, состав операций и способы их вызова останутся практически неизменными, что и определяет тот факт, что мы имеем дело с одним и тем же типом данных.

Стеком назовем последовательность элементов одного и того же типа, к которой можно добавлять новые элементы и убирать элементы из этой последовательности, причем как добавление новых элементов, так и удаление старых производится с одного и того же конца этой последовательности, называемого вершиной стека. Иначе говоря, при удалении элемента удаляется всегда тот элемент, который был последним добавлен в стек, таким образом, элементы удаляются в порядке, обратном порядку добавления элементов (рис. 2.1 и 2.2).

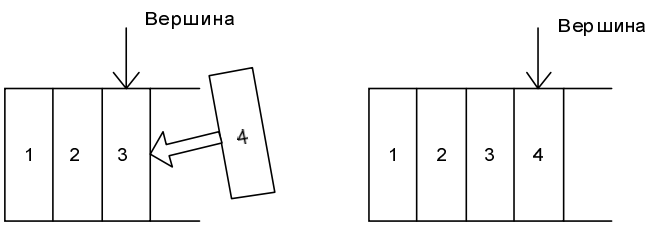


Рис. 2.1. Добавление элемента в стек

Рис. 2.2. Удаление элемента из стека

Различают *ограниченные* и *неограниченные* стеки. В первом случае количество элементов стека ограничивается некоторым числом. При попытке добавить элемент в ограниченный стек, содержащий максимальное количество элементов, возникает исключительная ситуация. Обычно в таком случае в программе можно определить программную реакцию на возникновение такой ситуации. Во втором случае размер стека ограничен только наличием доступной памяти.

Зададим набор операций, характеризующих стек как абстрактный тип данных, в виде интерфейса (листинг 2.2).

Листинг 2.2. Определение интерфейса для стека

```
public interface IStack {
  // Добавление нового элемента в стек
  // В случае ограниченного стека может возникнуть ситуация
  // "переполнения" стека - StackOverflow
  void push(Object e) throws StackOverflow;

  // Удаление верхнего элемента из стека;
  // если стек пуст, то возбуждается ситуация StackUnderflow
  Object pop() throws StackUnderflow;

  // Функция, возвращающая значение верхнего элемента из стека
  // без его удаления с вершины стека;
  // если стек пуст, то возбуждается ситуация StackUnderflow
  Object peek() throws StackUnderflow;

  // Функция удаления из стека всех элементов
  void makeEmpty();

  // Функция проверки пустоты стека
  boolean empty();
}
```

Каждая реализация этого интерфейса должна будет обеспечивать выполнение всех базовых операций над стеком, кроме того, разумеется, для каждой конкретной реализации стека будет необходимо определить конструктор.

Одна из самых простых реализаций стека представлена в листинге 2.3 — это *ограниченный стек*, реализованный в виде массива элементов, причем размер этого массива определяет максимальное число элементов в стеке. Кроме массива требуется еще указатель положения верхнего элемента в этом массиве. Роль такого указателя может играть целое число — число элементов, хранящихся в стеке.

Листинг 2.3. Определение ограниченного стека

```
public class BoundStack implements IStack {
  private Object[] stackBody;  // массив элементов
  private int nElem;           // количество элементов
```

```
public BoundStack (int n) throws StackUnderflow {
    // Количество элементов должно быть положительным
    if (n <= 0) throw new StackUnderflow();
    // Резервируем память под максимальное число элементов
    stackBody = new Object[n];
    // При создании стека элементов еще нет
    nElem = 0;
  }

  public void push(Object e) throws StackOverflow {
    // Проверка на переполнение стека
    if (nElem == stackBody.length) throw new StackOverflow();
    // Новый элемент вставляется в стек
    stackBody[nElem++] = e;
  }

  public Object pop() throws StackUnderflow {
    // Проверка на пустоту стека
    if (nElem == 0) throw new StackUnderflow();
    // Физически верхний элемент остается в массиве,
    // но количество элементов уменьшается
    return stackBody[--nElem];
  }

  public Object peek() throws StackUnderflow {
    // Проверка на пустоту стека
    if (nElem == 0) throw new StackUnderflow();
    // Верхний элемент возвращается в качестве результата
    return stackBody[nElem - 1];
  }

  public void makeEmpty() {
    // Физически элементы из массива не убираются,
    // но количество элементов сбрасывается в ноль
    nElem = 0;
  }

  public boolean empty() {
    // Проверяется только количество элементов
```

```
    return nElem == 0;
  }
}
```

В ядре языка Java имеется определение стандартного класса `java.util.Stack`, который содержит все операции, определенные нами в интерфейсе `IStack`. Однако в языке класс `Stack` наследует классу `Vector`, т. е. для него допустимы не только стековые операции, но и операции над вектором. С практической точки зрения это удобно, но понятие стека в таком случае размывается. Программист, использующий такой стек, с большой степенью вероятности не сможет удержаться в рамках чисто стековых операций, а это в конечном счете приведет к тому, что стек будет использоваться не как абстрактная структура, но как набор дополнительных операций над вектором.

Конечно, на практике описанный интерфейс стека может расширяться, в том числе и за счет операций доступа к элементам, отличным от верхнего, но общий принцип работы со стеком состоит в том, что нумерация элементов всегда производится от вершины, и при этом изменения происходят только в верхних элементах стека. Например, вполне допустимо определить такие операции над стеком:

```
// Дублирование верхнего элемента -
// операция эквивалентна push(top())
void dup();
// Обмен местами двух верхних элементов -
// операция эквивалентна последовательности
// Object a = pop(), b = pop(); push(a); push(b);
void swap();
// Копирование на вершину i-го элемента, считая от вершины
void copy(int i);
```

и др. На практике эти и подобные им операции часто включаются в определение стека как абстрактного типа данных.

Приведем простой пример использования стека в задаче анализа скобочной структуры некоторого текста. Пусть задан текст, содержащий три типа скобок: `()`, `[]` и `{}`. Программа должна проверить правильность расстановки скобок в тексте, т. е. выяснить, имеется ли для каждой открывающей скобки соответствующая закрывающая скобка и наоборот, а также не нарушены ли правила вложенности скобок. Символы, не являющиеся скобками, программа должна игнорировать. Для простоты напишем решение задачи в виде статической функции, получающей исходный текст в виде строкового аргумента и выдающей логический результат: истинное значение, если скобки расставлены правильно, и ложное, если обнаружена ошибка.

Алгоритм решения задачи может быть следующим.

Программа, приведенная в листинге 2.4, читает символ за символом, причем, встретив открывающую скобку, помещает ее в стек, а обнаружив закрывающую скобку, проверяет, находится ли уже на вершине стека соответствующая открывающая скобка. Если проверка была успешной, то после проверки скобка удаляется из стека, иначе функция немедленно заканчивает работу, выдав отрицательный результат (`false`).

Если в исходном тексте закрывающих скобок оказывается больше, чем открывающих, то программа попытается извлечь символ из пустого стека. Это приведет к ситуации `StackUnderflow`, которая перехватывается, и функция опять выдаст в качестве результата значение `false`. Наоборот, если в тексте открывающих скобок окажется больше, чем закрывающих, то после того, как весь текст будет прочитан и обработан, в стеке еще останутся "незакрытые" скобки. В этом случае функция тоже выдаст значение `false` в качестве результата. Результат проверки текста окажется успешным, если во время работы ни разу не обнаружится несоответствия скобок, а также не возникнет ситуации исчерпания стека, причем в конце обработки текста стек окажется пустым.

Размер стека определяет максимальную глубину вложенности скобок, которая может быть исследована программой. При превышении этой глубины вложенности функция возбудит исключительную ситуацию `StackOverflow`. Будем считать, что эта максимальная глубина вложенности скобок также передается в функцию как аргумент.

Листинг 2.4. Анализ скобочной структуры текста с помощью стека

```
public static boolean brackets(String txt, int maxdeep)
       throws StackOverflow {
  try {
    // Формируем стек из символов (значений класса Character)
    IStack stk = new BoundStack(maxdeep);
    for (int i = 0;  i < txt.length();  i++) {
      char c = txt.charAt(i);
      switch (c) {
        case '(': case '[': case '{':
          stk.push(new Character(c));
          break;
        case ')':
          if (((Character)stk.pop()).charValue() != '(')
            return false;
          break;
```

```
          case ']':
            if (((Character)stk.pop()).charValue() != '[')
              return false;
            break;
          case '}':
            if (((Character)stk.pop()).charValue() != '{')
              return false;
            break;
        }
      }
      return stk.empty();
    } catch (StackUnderflow e) {
      return false;
    }
  }
```

Заметим, что если в качестве исходной строки для представленной функции взять текст самой функции, то будет зафиксирована ошибка в расстановке скобок из-за символьных констант, изображающих скобки.

Рассмотрим другие возможные реализации стека.

Если в программе используется много стеков, размер которых может сильно меняться, то реализация стека в виде массива (ограниченный стек) может привести к неоправданно большим расходам памяти. При этом, если в какой-то момент времени реальное количество элементов, содержащихся в стеках, невелико, то фактически память, зарезервированная для элементов стеков, может быть загружена весьма мало. В подобных случаях имеет смысл захватывать память только под те элементы, которые фактически размещаются в стеке.

Основу приведенной в листинге 2.5 реализации стека составляет список элементов стека, представленный в *разд. 1.2* (`ObjectList`). Список имеет то преимущество перед массивом, что под его элементы память выделяется независимо от других элементов. Вершиной стека в этой реализации служит головной элемент списка. Разумеется, элемент списка — это более сложный объект, чем элемент массива: дополнительная память требуется для связывания элементов. Тем не менее, общий расход памяти может даже уменьшиться за счет того, что не расходуется память под несуществующие элементы.

Листинг 2.5. Реализация неограниченного стека в виде списка

```
public class Stack implements IStack {
  private ObjectList stackBody = new ObjectList();
```

```
  public void push(Object e) throws StackOverflow {
    stackBody.addFirst(e);
  }

  public Object pop() throws StackUnderflow {
    try {
      return stackBody.remove();
    } catch (IndexOutOfBoundsException e) {
      throw new StackUnderflow();
    }
  }

  public Object peek() throws StackUnderflow {
    try {
      return stackBody.getFirst();
    } catch (IndexOutOfBoundsException e) {
      throw new StackUnderflow();
    }
  }

  public void makeEmpty() {
    stackBody.clear();
  }

  public boolean empty() {
    return stackBody.getCount() == 0;
  }
}
```

В приведенном определении класса все операции со стеком реализуются с помощью соответствующих операций со списком. По существу, происходит просто трансляция одних понятий в другие: операция со списком `addFirst` превращается в операцию `push` со стеком; исключительная ситуация `IndexOutOfBoundsException`, возникающая при попытке обращения к несуществующему элементу списка, превращается в ситуацию `StackUnderflow` и т. д. Тем не менее, несмотря на такую близость понятий, мы бы не рекомендовали реализовывать стек как *расширение* (extension) реализации списка: программное содержание операций может совпадать, но семантический смысл в них вкладывается разный. Ситуация здесь складывается точно такая же, как и со "стандартной" реализацией стека в виде вектора — реализация

интерфейса `IStack` позволяет использовать в программе только абстрактное понятие стека, однако если стек реализуется и применяется как расширение другого класса, программа уже использует не столько абстрактный интерфейс, сколько эту конкретную реализацию, и, следовательно, начинает зависеть от этой реализации.

Иногда используется некоторая комбинация реализаций стека в виде списка и в виде массива. Например, можно применять массив для хранения элементов двух стеков, "растущих" навстречу друг другу. К сожалению, операции над "разнонаправленными" стеками требуют различной реализации, так что здесь речь идет не о реализации двух отдельных стеков, а, скорее, об объекте, представляющем сразу пару стеков. Переполнение в этом случае возникает сразу у обоих стеков, когда весь массив оказывается заполненным элементами этих двух стеков.

Еще один вариант реализации — хранение элементов многих стеков в одном массиве (пуле), причем элементы в этом случае лучше связывать между собой не с помощью стандартных указателей, а с помощью индексов в этом массиве. Такая реализация позволяет гибко перераспределять память. Переполнение в этом случае возникает сразу для всех стеков в ситуации, когда весь пул занят элементами стеков. Эта реализация хорошо подходит для случая, когда работа со стеками составляет основное существо программы, и резервирование памяти системными средствами под каждый элемент стека не годится из-за высоких накладных расходов на работу системы.

Рассмотрим еще одну задачу, для решения которой удобно воспользоваться парой стеков.

Пусть дан текст, представляющий собой выражение, содержащее операнды, соединенные между собой знаками операций `+`, `-` и `*`. Допускается также использование круглых скобок. Будем считать, что значение каждого операнда может быть вычислено, причем результатом вычисления будет некоторое целое значение. Требуется вычислить значение всего выражения.

Прежде всего уточним постановку задачи. Будем считать, что выражение не содержит ошибок (неопределенные знаки операций, несбалансированные скобки и т. п.). Кроме того, условимся, что текст уже предварительно разобран на отдельные элементы (лексемы), представляющие минимальные рассматриваемые единицы текста — операнд, знак операции, скобка. Разбор осуществляется с помощью специального анализатора — объекта `parser` класса `LexAnalyzer`. Анализатор представляет собой в некотором роде внешний итератор исходной строки. Он создается с помощью конструктора, получающего исходное выражение в качестве аргумента. После этого можно получить очередную лексему с помощью вызова функции `next()`. Проверить, есть ли еще в тексте лексемы, можно с помощью вызова логической функции `hasNext()`, так что можно считать, что класс `LexAnalyzer` является реализацией уже обсуждавшегося ранее интерфейса `java.util.Iterator`.

Каждая лексема (составной "элемент" строки), в свою очередь, является объектом класса `Lexical`, содержащего значение очередной лексемы — числовое значение операнда или знак операции. Еще одна компонента лексемы — класс — будет показывать тип лексемы.

В листинге 2.6 представлены классы `LexAnalyzer` и `Lexical`, но реализация их методов не показана.

Листинг 2.6. Определение классов `LexAnalyzer` и `Lexical`

```
public class LexAnalyzer implements Iterator {
  // Конструктор:
  public LexAnalyzer(String source) { ... }
  // Реализация операций интерфейса Iterator
  public Object next() { ... }       // выдает очередную лексему
  public boolean hasNext() { ... }   // проверяет, есть ли еще лексемы
  public void remove() {}            // пустая операция
}

public class Lexical {
  // Возможные типы лексем:
  public static final int OPERAND = 1;   // операнд
  public static final int OPERATOR = 2;  // знак операции
  public static final int LEFTPAR = 3;   // левая (открывающая) скобка
  public static final int RIGHTPAR = 4;  // правая (закрывающая) скобка
  // Проверка класса и выдача значения лексемы:
  public int lexClass() { ... }   // выдает класс лексемы
  public char operSign() { ... }  // для операции выдает знак операции
  public int intValue() { ... }   // для операнда выдает его значение
}
```

Для решения нашей задачи будем использовать два стека. Пусть в одном из них хранятся значения операндов и промежуточных результатов. Элементами этого стека будут, таким образом, целые числа, представленные объектами стандартного класса `Integer`. Во втором стеке в процессе вычислений будут храниться знаки операций вместе со своими приоритетами. Эти операции должны записываться в стек и выполняться по мере возможности, т. е. как только окажутся готовыми значения операндов для выполнения операции. В листинге 2.7 приведено определение класса `Operator`, представляющего знак операции и ее приоритет, которые хранятся в стеке операций.

Листинг 2.7. Определение класса `Operator`

```
public class Operator {
  public char sign; // знак операции
  public int prio;  // приоритет операции

  // Конструктор создает объект по знаку операции
  public Operator(char s) {
    sign = s;
    // Вычисление приоритета по знаку операции
    if (s == '*') prio = 2;
    else if (s == '+' || s == '-') prio = 1;
    else prio = 0;
  }

  // Функции доступа
  public getSign() { return sign; }
  public getPrio() { return prio; }
}
```

Теперь определим функцию (листинг 2.8), которая исполняет операцию, лежащую на вершине стека операций, над операндами, находящимися на вершине стека операндов. Операнды извлекаются с вершины стека операндов и считаются объектами класса `Integer`. Результат выполнения операции снова записывается в стек операндов. Стеки с операциями и операндами передадим функции в виде аргументов.

После этого определим несколько более сложную функцию, которая исполняет последовательность операций до тех пор, пока в стеке операций не обнаружится операция, имеющая приоритет, меньший заданного, или вовсе не окажется операций. Такая функция исполняется в тот момент, когда в процессе анализа обнаруживается новая операция, и надо выполнить все те операции такого же и более высокого приоритета, которые уже попали в стек операций, и операнды для которых уже подготовлены.

Листинг 2.8. Функции выполнения операций

```
public static void doOperator(
          IStack operands,      // стек операндов
          IStack operators      // стек операций
         ) throws StackOverflow, StackUnderflow {
```

```
  int op2 = ((Integer)operands.pop()).intValue();       // правый операнд
  int op1 = ((Integer)operands.pop()).intValue();       // левый операнд
  char opSign = ((Operator)operators.pop()).getSign();  // знак операции
  switch (opSign) {
    case '+':
      operands.push(new Integer(op1 + op2));
      break;
    case '-':
      operands.push(new Integer(op1 - op2));
      break;
    case '*':
      operands.push(new Integer(op1 * op2));
      break;
  }
}

public static void doOperators(
          IStack operands,     // стек операндов
          IStack operators,    // стек операций
          int minPrio          // граничный приоритет
         ) throws StackOverflow, StackUnderflow {
  while (!operators.empty() &&
         ((Operator)operators.peek()).getPrio() >= minPrio) {
    doOperator(operands, operators);
  }
}
```

В обеих функциях возможно возникновение ситуации `StackUnderflow` в методах `peek()` и `pop()` стека, а также ситуации `StackOverflow` при вызове метода `push()`. Для простоты в данном примере эти ситуации не обрабатываются, а просто пропускаются в вызывающую функцию.

Наконец, напишем функцию (листинг 2.9), которая определяет оба стека и выполняет все требуемые вычисления.

Листинг 2.9. Вычисление значения выражения на стеках

```
public static int exprValue(String expr) throws WrongExpression {
  IStack operands = new Stack();   // стек операндов
  IStack operators = new Stack();  // стек операций
  LexAnalizer analizer = new LexAnalizer(expr);  // анализатор
```

```
try {
  while (analizer.hasNext()) {
    Lexical lex = (Lexical)analizer.next();
    switch (lex.lexClass()) {
      case Lexical.OPERAND:
        /* операнд записывается в стек */
        operands.push(new Integer(lex.intValue()));
        break;
      case Lexical.RIGHTPAR:
        /* Выполнятся все операции до соответствующей левой скобки,
           которая затем выталкивается из стека
        */
        doOperators(operands, operators, 1);
        operators.pop();
        break;
      case Lexical.OPERATOR:
        /* Выполняются операции с приоритетом, большим или равным
           приоритету новой операции, после чего новый оператор
           записывается в стек операций
        */
        Operator newOp = new Operator(lex.operSign());
        doOperators(operands, operators, newOp.getPrio());
        operators.push(newOp);
        break;
      case Lexical.LEFTPAR:
        /* Левая скобка просто записывается в стек операций
           с приоритетом 0
        */
        operators.push(new Operator('('));
        break;
    }
  }
  // Анализ закончен, выполняем все оставшиеся в стеке операции
  doOperators(operands, operators, 0);
  // На вершине стека операндов — результат вычислений
  return ((Integer)operands.pop()).intValue();
} catch (StackOverflow ex) {
  throw new WrongExpression();
```

```
  } catch (StackUnderflow ex) {
    throw new WrongExpression();
  }
}
```

При условии правильности подаваемого на вход выражения приведенная функция произведет все необходимые вычисления и выдаст целочисленный результат. Функцию нетрудно расширить на случай других бинарных операций (скажем, деления) или других типов операндов. Не очень трудно предусмотреть и случай корректного анализа ошибок. В приведенном коде функции определена реакция только на случай переполнения или исчерпания использующегося стека. Однако очевидно, что множество допустимых лексем в каждый момент времени определено однозначно: после левой скобки или знака операции должен идти операнд или левая скобка; после операнда или правой скобки ожидается знак операции или правая скобка. Если учесть это правило и следить за правильностью расстановки скобок и допустимостью всех лексем, то в процессе вычисления любую ошибку в выражении можно легко обнаружить.

Как в этом, так и в предыдущем примере в функциях `exprValue` и `brackets` используется абстрактный стек, представленный интерфейсом `IStack`. Тем не менее, обе функции все же остаются зависимыми от конкретной реализации стека, т. к. при создании стека приходится использовать конструктор одной из реализаций. В частности, это означает, что если по каким-либо причинам в проекте будет произведена глобальная замена реализации стека, то придется вносить изменения во все места программы, где используются вызовы конструкторов. Хорошо было бы и для создания стеков тоже использовать абстрактные функции, вынеся вызовы конструкторов в отдельный класс так, чтобы при смене реализации стека можно было бы произвести замену только одной строчки в программе. Такая технология носит название "фабрики объектов".

В нашем случае можно построить абстрактный класс `StackFactory` ("фабрика стеков"), реализующий операцию создания абстрактного стека `createStack()`. Единственной задачей этого метода будет вызов конкретного конструктора для создания стека. Например:

```
public abstract class StackFactory {
  public static IStack createStack() { return new Stack(); }
}
```

Теперь во всех используемых ранее функциях можно заменить вызов конкретного конструктора стека на генерацию абстрактного стека с помощью "фабрики", например, начало функции `exprValue` теперь будет выглядеть следующим образом:

```
public static int exprValue(String expr) throws WrongExpression {
  IStack operands = StackFactory.createStack();   // стек операндов
```

```
  IStack operators = StackFactory.createStack();  // стек операций
  ...
}
```

Подробнее об абстрактных фабриках и "виртуальных конструкторах" можно прочитать в книге [3].

Похожей на стек структурой данных является *очередь*. Очередью будем называть последовательность элементов одного и того же типа, к которой добавляются и из которой удаляются элементы, однако в отличие от стека добавление элементов производится в один конец, а удаление происходит с другого конца (рис. 2.3).

Тот конец очереди, из которого выполняется удаление элементов, называется *головой очереди*, а другой ее конец, куда происходит добавление элементов — *хвостом очереди*.

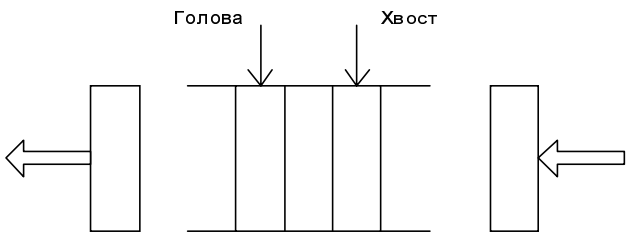


Рис. 2.3. Структура очереди

Набор операций над очередью практически тот же, что и над стеком, однако реализации обычно отличаются достаточно сильно — наличие двух концов у очереди осложняет дело.

Можно представлять очередь в виде кольца элементов с разрывом в точке, где сходятся ее голова и хвост (рис. 2.4).

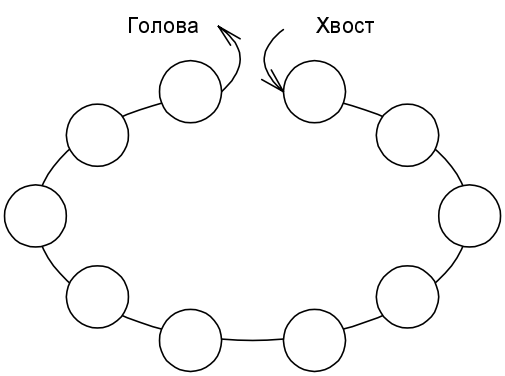


Рис. 2.4. Очередь, представленная в виде кольца

Если память под элементы очереди представлена массивом элементов, то реализация будет учитывать эту "кольцевую" структуру массива — первый элемент массива будет логически следовать за последним. Если же в качестве реализации выбирается списковая структура, то выбор кольцевой структуры для этого списка — один из популярных способов реализации очереди.

Один из вариантов интерфейса абстрактной очереди представлен в листинге 2.10.

Листинг 2.10. Интерфейс для описания очереди

```
public interface IQueue {
  // Добавить элемент в очередь
  void enqueue(Object item) throws QueueOverflow;
  // Удалить элемент из очереди
  Object dequeue() throws QueueUnderflow;
  // Выдать первый (головной) элемент
  Object head() throws QueueUnderflow;
  // Выдать последний (хвостовой) элемент
  Object tail() throws QueueUnderflow;
  // Проверить, пуста ли очередь
  boolean empty();
  // Опустошить очередь
  void makeEmpty();
}
```

Здесь предполагается, что при добавлении элемента в очередь (как и при добавлении в стек) может возникнуть ситуация переполнения очереди. Такое возможно, например, при реализации очереди в виде ограниченного массива. Ситуация исчерпания очереди может возникнуть при попытке удаления или взятия значения элемента из пустой очереди.

В нашей первой реализации (листинг 2.11) очередь будет представлена массивом элементов. Эта реализация демонстрирует "ограниченную" очередь. Максимальное количество элементов очереди задается при создании очереди в конструкторе. Реализация очень похожа на соответствующую реализацию ограниченного стека, однако есть и существенное отличие. Поскольку элементы добавляются в один конец очереди, а удаляются из другого конца, то в процессе работы с очередью элементы будут "смещаться" в конец массива. Наконец, настанет момент, когда ни одного элемента больше вставить не удастся, хотя свободное место в массиве есть — в начале массива на месте ранее существовавших, но впоследствии удаленных элементов. Можно, конечно, физически передвинуть элементы очереди, однако это может ока-

заться достаточно длительной процедурой. Лучше просто написать методы работы с очередью так, чтобы считать массив "замкнутым" в кольцо, т. е. переходить от последнего элемента массива непосредственно к первому.

В приведенной реализации переменные класса `pHead` и `pTail` содержат индексы первого и последнего элемента очереди соответственно. Количество элементов очереди равно `(pTail – pHead + 1)`, если индекс последнего элемента больше или равен индексу первого элемента (рис. 2.5), и равно `(elements.length – pHead + pTail + 1)` в противном случае (рис. 2.6).

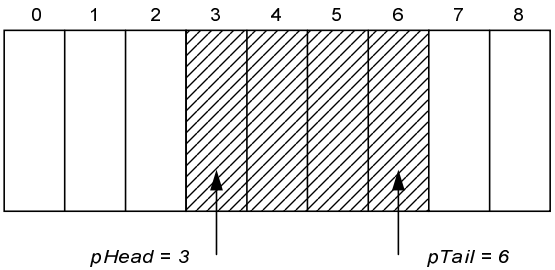


Рис. 2.5. Очередь содержит элементы (3), (4), (5) и (6)

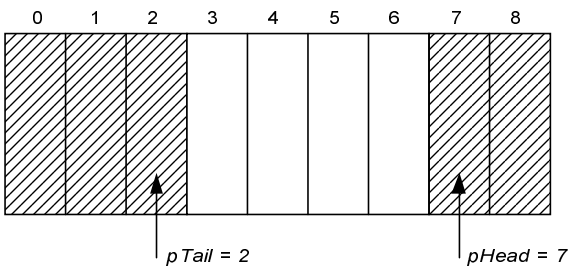


Рис. 2.6. Очередь содержит элементы (7), (8), (0), (1) и (2)

Одна из трудностей состоит в том, что при удалении последнего элемента из очереди возникает ситуация, при которой нет ни первого, ни последнего элемента. Удобно в реализации считать, что в этом случае индекс `pTail` на единицу больше индекса `pHead`. Однако точно такая же ситуация возникает и в том случае, когда все элементы массива заняты!

Существует множество способов справиться с этой ситуацией. Один из них (приведенный в этой книге) состоит в том, что дополнительно к индексам первого и последнего элементов хранится количество элементов массива (`nCount`), занятых элементами очереди. В этом случае ситуация отсутствия элементов однозначно определяется тем, что `nCount == 0`. В этом случае упрощается также реализация метода `empty()`, и дополнительно появляется возможность очень просто реализовать метод, выдающий количество эле-

ментов очереди. Платой за такую простоту является трата дополнительного времени и памяти на работу с еще одной переменной.

Все остальное в этой реализации очень просто и не нуждается в пояснениях.

Листинг 2.11. Реализация ограниченной очереди

```
public class BoundQueue implements IQueue {
  Object[] elements;     // элементы очереди
  int pHead;             // индекс первого элемента
  int pTail;             // индекс последнего элемента
  int nCount;            // количство элементов очереди

  // Конструктор
  public BoundQueue(int max) {
    elements = new Object[max];
    pHead = 0;
    pTail = max - 1;
    nCount = 0;
  }

  // Добавление элемента в очередь
  public void enqueue(Object item) throws QueueOverflow {
    if (nCount == elements.length) throw new QueueOverflow();
    if (++pTail == elements.length) pTail = 0;
    elements[pTail] = item;
    nCount++;
  }

  // Удаление элемента из очереди
  public Object dequeue() throws QueueUnderflow {
    if (nCount == 0) throw new QueueUnderflow();
    Object result = elements[pHead];
    if (++pHead == elements.length) pHead = 0;
    nCount--;
    return result;
  }

  // Взятие значения элемента из головы очереди
  public Object head() throws QueueUnderflow {
```

```
    if (nCount == 0) throw new QueueUnderflow();
    return elements[pHead];
  }

  // Взятие значения элемента из хвоста очереди
  public Object tail() throws QueueUnderflow {
    if (nCount == 0) throw new QueueUnderflow();
    return elements[pTail];
  }

  // Проверка пустоты очереди
  public boolean empty() {
    return nCount == 0;
  }

  // "Опустошение" очереди
  public void makeEmpty() {
    if ((pTail = pHead - 1) < 0) {
      pTail = elements.length - 1;
    }
    nCount = 0;
  }
}
```

Еще одна реализация этого интерфейса использует кольцевой список как основу реализации. Как и в случае со стеком, основную нагрузку берет на себя реализация операций с кольцевым списком, а в реализации очереди эти операции просто переводятся на язык ее интерфейса. В листинге 2.12 приведено описание класса `CircularList`, реализующего список кольцевой структуры, и далее определена реализация очереди на базе этого класса.

Листинг 2.12. Определение очереди на базе списка

```
/*********************************************
 * Реализация простейшего кольцевого списка
 *********************************************/
public class CircularList {
  class ListItem {
    public Object item;       // элемент списка
    public ListItem next;     // следующий элемент
```

```
  public ListItem(Object item, ListItem next) {
    this.item = item;
    this.next = next;
  }
}
ListItem last;    // Последний элемент списка содержит ссылку
                  // на первый элемент.
                  // В пустом списке last == null

public CircularList() { last = null; }

public void insertHead(Object item) {
  if (last == null) {
    // Новый элемент будет одновременно первым и последним
    last = new ListItem(item, null);
    last.next = last;
  } else {
    // Новый элемент вставляется за последним
    last.next = new ListItem(item, last.next);
  }
}

public void insertTail(Object item) {
  insertHead(item);
  // Чтобы первый элемент стал последним в кольцевом списке,
  // достаточно сдвинуться вперед на один шаг
  last = last.next;
}

public Object removeHead() throws EmptyException {
  if (last == null) throw new EmptyException();
  Object res = last.next.item;
  if (last.next == last) {
    // Удаляется единственный элемент
    last = null;
  } else {
    last.next = last.next.next;
  }
```

```
    return res;
  }

  public boolean empty() { return last == null; }

  public Object head() throws EmptyException {
    if (last == null) throw new EmptyException();
    return last.next.item;
  }

  public Object tail() throws EmptyException {
    if (last == null) throw new EmptyException();
    return last.item;
  }
}

public class ListQueue implements IQueue {
  CircularList list;    // базовый список

  public ListQueue() { list = new CircularList(); }

  public void enqueue(Object item) throws QueueOverflow {
    list.insertTail(item);
  }

  public Object dequeue() throws QueueUnderflow {
    try {
      return list.removeHead();
    } catch (EmptyException e) {
      throw new QueueUnderflow();
    }
  }

  public Object head() throws QueueUnderflow {
    try {
      return list.head();
    } catch (EmptyException e) {
```

```
      throw new QueueUnderflow();
    }
  }

  public Object tail() throws QueueUnderflow {
    try {
      return list.tail();
    } catch (EmptyException e) {
      throw new QueueUnderflow();
    }
  }

  public boolean empty() { return list.empty(); }

  public void makeEmpty() {
    try {
      while (!list.empty()) { list.removeHead(); }
    } catch (EmptyException e) {}
  }
}
```

В этой реализации при описании кольцевого списка определены все необходимые операции над списком по добавлению, удалению и доступу к концевым элементам списка, которые впоследствии используются при определении очереди. Конечно, в реальной жизни для работы с кольцевым списком этих операций будет, по всей вероятности, недостаточно, однако для реализации операций с очередью этого вполне хватит.

При попытке обратиться к несуществующим элементам списка возбуждается ситуация `EmptyException` (которую, разумеется, тоже надо было бы определить отдельно). При реализации операций над очередью эта ситуация превращается в ситуацию, характерную для очереди, — `QueueUnderflow`.

Если бы не трансляция одной исключительной ситуации в другую, почти вся реализация уложилась бы в чистый перевод одних названий (операций над списком) в другие (операции над очередью). Только для операции `makeEmpty()` не нашлось аналога среди операций над списком, поэтому ее пришлось реализовать с помощью цикла, последовательно удаляющего все элементы из списка. Конечно, в самом списке такая операция могла бы быть реализована гораздо проще.

Еще некоторые примеры применения стеков и очередей можно найти в *разд. 2.3* и *5.1*.

В заключение этого раздела приведем пример одной экзотической реализации стека (есть и аналогичная реализация очереди), предназначенной для случая неглубокого стека, содержащего только битовые значения (нули и единицы). В этом случае весь стек можно уложить в одно слово (например, в значение типа **int**). В слове помещается 32 бита, но поскольку по крайней мере один бит придется занять для служебных целей (надо отметить положение вершины стека), то остается 31 бит, в которых можно сохранить одновременно до 31 битовых элемента стека! Для многих задач этого вполне достаточно. Рассмотрим, например, задачу об анализе скобочной структуры выражения, решение которой было приведено в этой главе в листинге 2.4. Если бы в задаче речь шла не о трех, а о двух типах скобок (скажем, круглые и квадратные), то для хранения информации о типе скобки было бы достаточного одного битового значения. Стека из 31 битового элемента вполне хватило бы для практических целей — вложенность скобок редко превышает 7—8 уровней, так что ситуация переполнения стека была бы практически невероятной.

В реализации все операции определены над элементами типа **int** (на самом деле предполагается, что значениями этих элементов будут только нули и единицы). Это несколько нарушает интерфейс `IStack`, в котором предполагалось, что элементами стека служат объекты класса `Object`, так что приведенная в листинге 2.13 реализация не объявлена реализацией интерфейса `IStack`, однако фактически все операции те же самые.

Листинг 2.13. Реализация стека битовых значений в одном целом значении

```
/*******************************************************
 * Реализация ограниченного стека из битовых элементов
 * Стек может содержать до 31 бита
 *******************************************************/
public class BitStack {
  int stack = 1;   // содержимое стека; вершина - младший бит слова

  // Операция вталкивания элемента в стек
  void push(int e) throws StackOverflow {
    // Проверка переполнения: старшая единица в слове
    // служит маркером дна стека
    if ((stack & 0x80000000) != 0) throw new StackOverflow();
    stack <<= 1;      // содержимое сдвигается влево на один бит
    stack |= (e & 1); // новый элемент записывается в младший бит
  }
```

```
  // Операция выталкивания элемента из стека
  int pop() throws StackUnderflow {
    // Стек пуст, если в нем содержится только маркер дна стека
    if (stack == 1) throw new StackUnderflow();
    int res = stack & 1; // Вершина стека - младший разряд слова
    stack >>>= 1;        // Содержимое слова сдвигается вправо
    return res;
  }

  // Операция выдачи содержимого вершины стека
  int peek() throws StackUnderflow {
    if (stack == 1) throw new StackUnderflow();
    return stack & 1;
  }

  // Операция опустошения стека
  void makeEmpty() {
    stack = 1;
  }

  // Операция проверки пустоты стека
  boolean empty() {
    return stack == 1;
  }
}
```

Можно написать реализацию стека, основанную на той же идее, но с элементами данных, занимающими не один бит, а два или даже три. Разумеется, чем более крупные элементы записываются в стек, тем меньше их удастся записать. Так, для элементов шириной в три бита (например, целые числа из диапазона от 0 до 7) места в слове хватит для 10 элементов, так что максимальная глубина стека составит 10 позиций.

2.3. Прохождение деревьев

В *разд. 1.3* был представлен один простой алгоритм прохождения деревьев. Разумеется, для практических целей этого недостаточно. Итерация деревьев является важнейшим элементом многих алгоритмов обработки информации, в том числе и представленных в этой книге. Часто бывает не все равно, в

каком порядке проходятся узлы дерева, так что требуется знание того, как можно организовать различные порядки обхода узлов дерева.

Как и в случае списков, наиболее просто программируются обходы с помощью внутренних итераторов, но для пользователей гораздо чаще более удобно организовать обход узлов дерева с помощью внешнего итератора, т. е. определить специальный класс (или классы), реализующий интерфейс `java.util.Iterator` или `java.util.Enumeration` для задания алгоритма обхода дерева. Классы, задающие как внешние, так и внутренние итераторы, могут быть описаны непосредственно внутри класса, определяющего дерево. Конечно, соответствующие методы класса должны генерировать необходимый итератор.

Пусть, например, дерево определено с помощью класса `Tree`. Предположим, для этого дерева предлагаются два способа его обхода. Определим два метода `extTraverse1()` и `extTraverse2()`, которые будут генерировать и выдавать соответствующие внешние итераторы, и два метода `intTraverse1()` и `intTraverse2()`, которые будут задавать алгоритмы для соответствующих внутренних итераторов.

```
public class Tree {
  ...
  public Iterator extTraverse1() { ... }
  public Iterator extTraverse2() { ... }
  public void intTraverse1(Action action) { ... }
  public void intTraverse2(Action action) { ... }

  private static class Traverse1 implements Iterator { ... }
  private static class Traverse2 implements Iterator { ... }
  ...
}
```

Тогда в программе можно будет осуществлять итерацию дерева разными способами (с помощью внутреннего или внешнего итератора) и в разной последовательности обхода в зависимости от потребностей программы, например:

```
// Внешний итератор; первый порядок обхода
for (Iterator myIterator = myTree.extTraverse1();
     myIterator.hasNext(); ) {
  Node nextNode = (Node)myIterator.next();
  <обработка очередного узла>
}
```

или

```
// Внутренний итератор; второй порядок обхода
myTree.intTraverse2(new Visitor() {
  public void visit(Object nextNode) {
    <обработка очередного узла>
  }
});
```

Внутренние итераторы часто строятся в виде рекурсивных процедур. Действительно, сравнительно несложно написать рекурсивную процедуру, которая будет для каждого узла дерева выполнять "посещение" этого узла, заданное аргументом процедуры. Для внешнего итератора такой способ написания обхода не подходит: внешний итератор должен быть готов к тому, что его действие прерывается на время обработки узла, после чего итератор должен выдать следующий по порядку обхода узел.

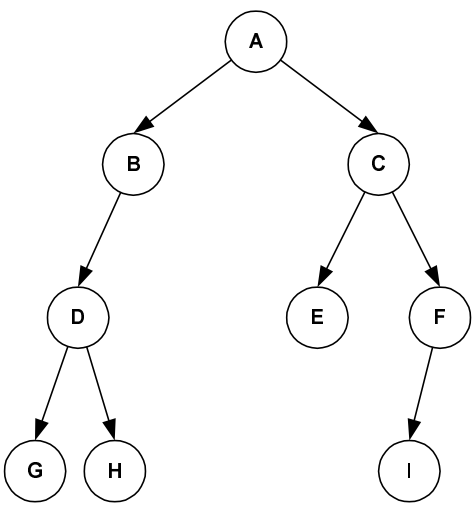


Рис. 2.7. Пример бинарного дерева

Давайте рассмотрим двоичное дерево, содержащее в узлах символы, например, такое, как на рис. 2.7. Говорят, что дерево проходится сверху вниз, если каждый узел всегда проходится раньше, чем поддеревья этого узла. Для изображенного дерева следующие порядки прохождения узлов будут порядками обхода "сверху вниз":

A B C D E F G H I

A B D H G C E F I

A C B F I E D G H

В этом случае также говорят, что дерево обходится в нисходящем порядке. Аналогично, при обходе снизу вверх (восходящий порядок обхода) узел проходится после того, как пройдены его поддеревья, например:

I H G F E D C B A

I F E C G H D B A

Очевидно, что для каждого обхода "сверху вниз" обход тех же узлов в противоположном порядке будет обходом "снизу вверх".

Порядок обхода бинарного дерева называется *инфиксным*, если каждый узел проходится после того, как пройдены все узлы одного поддерева, и до того, как пройдены узлы другого поддерева. Обычно используют либо левосторонний инфиксный обход, при котором сначала обходится левое поддерево, потом корневой узел, потом правое поддерево, либо правосторонний инфиксный обход, при котором наоборот, сначала обходится правое поддерево, затем корень, и, наконец, левое поддерево. Вот как, например, располагаются узлы нашего дерева в порядке левостороннего инфиксного обхода:

G D H B A E C I F

Для нисходящих и восходящих обходов обычно также определяются некоторые "стандартные" порядки обхода, например, при нисходящем левостороннем обходе порядок прохождения узлов определен следующим образом: непосредственно после обхода каждого узла проходятся узлы его поддеревьев в порядке слева направо (для бинарного дерева — сначала левого поддерева, затем правого). Для дерева, приведенного на рис. 2.7, узлы в порядке нисходящего левостороннего обхода располагаются следующим образом:

A B D G H C E F I

Еще два термина, часто использующихся для определения порядка обхода узлов, — "обход в глубину" (depth first) и "обход в ширину" (breadth first).

Говорят, что обход является обходом *"в ширину"*, если узлы, расположенные ближе к корню дерева, обходятся раньше, чем узлы, отстоящие дальше от корня. При этом расстояние от узла до корня измеряется количеством ребер на кратчайшем пути из этого узла до корня. Так, например, в дереве на рис. 2.7 узлы *D*, *E* и *F* находятся на одном и том же расстоянии от корня — 2, а узел *B* располагается ближе к корню, т. к. расстояние от него до корня равно 1. Обычно полагают, что корень находится на расстоянии 0 от себя, так что при обходе "в ширину" он всегда обходится первым. Левосторонний обход "в ширину" для дерева на рис. 2.7 расположит его узлы в следующем порядке:

A B C D E F G H I

Говорят, что обход является обходом *"в глубину"*, если в любом поддереве исходного дерева узлы обходятся "подряд", т. е. если обход некоторого поддерева начат, то он продолжается до тех пор, пока все поддерево не будет

обойдено. Например, нисходящий обход "в глубину" может быть представлен такой последовательностью узлов того же дерева:

A B D G H C E F I

Если в этой последовательности узлов расставить скобки для обозначения поддеревьев, то можно увидеть, что действительно, каждое поддерево занимает свой "участок" в этой последовательности, не пересекающийся с участками, занятыми другими поддеревьями:

(*A* (*B* (*D* (*G*) (*H*))) (*C* (*E* (*F* (*I*)))))

Обходы "в глубину" особенно удобно реализовывать в виде рекурсивных процедур, поскольку такая процедура, получив в качестве аргумента некоторое поддерево для обхода, будет обходить его до тех пор, пока все узлы этого поддерева не будут пройдены. Напротив, реализовать обход "в ширину" с помощью рекурсивной процедуры практически невозможно, во всяком случае, такая реализация обхода будет неестественной и неэффективной.

В данной книге рассматривается много способов и процедур обхода двоичного дерева и деревьев общего вида. В *разд. 1.3* уже был представлен внутренний итератор для левостороннего инфиксного обхода, реализованный с помощью рекурсивной процедуры. Далее в этом разделе представлены еще три способа обхода деревьев с использованием дополнительных структур данных для хранения информации в процессе обхода:

- внутренний итератор для нисходящего обхода "в глубину", использующий стек для хранения информации о еще непройденных поддеревьях;
- внешний итератор для нисходящего обхода "в ширину", использующий очередь для хранения вершин в порядке обхода;
- внешний итератор для восходящего обхода, использующий стек для хранения информации о пути из корня к вершине, рассматриваемой в текущий момент.

После этого рассматриваются еще два способа обхода для специальных деревьев с хранением всей информации внутри самой структуры дерева и его узлов. В следующих разделах рассматриваются и применяются еще несколько способов обхода дерева для использования их в специальных алгоритмах обработки информации.

Итак, начнем с описания внутреннего итератора для нисходящего обхода "в глубину".

Будем использовать следующий алгоритм. Для того чтобы обойти некоторое поддерево, сначала пройдем корень этого поддерева, затем спустимся в его левое поддерево, а правое поддерево запомним в стеке. Если левое поддерево пусто, то можно сразу спускаться в правое поддерево вместо того, чтобы запоминать его в стеке. Если оба поддерева пусты, то очередное поддерево надо извлечь из стека. В табл. 2.1 представлены последовательные стадии

обхода дерева, изображенного на рис. 2.7, при этом на каждом шаге показаны текущая вершина и вершины поддеревьев, хранящихся в стеке. Дно стека обозначено символом [, а вершина — символом).

Таблица 2.1. *Последовательность шагов при нисходящем обходе в глубину*

Номер шага	Текущая вершина	Содержимое стека	Пройденные вершины
1	*A*	[)	
2	*B*	[*C*)	*A*
3	*D*	[*C*)	*A*, *B*
4	*G*	[*C*, *H*)	*A*, *B*, *D*
5	*H*	[*C*)	*A*, *B*, *D*, *G*
6	*C*	[)	*A*, *B*, *D*, *G*, *H*
7	*E*	[*F*)	*A*, *B*, *D*, *G*, *H*, *C*
8	*F*	[)	*A*, *B*, *D*, *G*, *H*, *C*, *E*
9	*I*	[)	*A*, *B*, *D*, *G*, *H*, *C*, *E*, *F*
10	`null`	[)	*A*, *B*, *D*, *G*, *H*, *C*, *E*, *F*, *I*

В листинге 2.14 приведено описание внутреннего итератора в виде метода класса `Tree` *(см. разд. 1.3)*, при этом считается, что задано также описание интерфейса `Visitor` *(см. разд. 1.2)* и сделаны все необходимые описания классов и интерфейсов для работы со стеком *(см. разд. 2.2)*.

Листинг 2.14. Нисходящий обход дерева с хранением узлов в стеке

```
/*******************************************************
* Внутренний итератор для нисходящего обхода
* с хранением узлов в стеке
*******************************************************/
public void traverseUpDown(Visitor visitor) {
  // Стек для хранения узлов
  IStack stack = StackFactory.createStack();
  // Текущая вершина
  Node current = root;
  // Основной цикл
  for (;;) {
    //---------- Обходим текущую вершину
```

```
    visitor.visit(current.item);
    //----------- Кладем в стек правое поддерево
    if (current.right != null) {
      stack.push(current.right);
    }
    //----------- Переходим к левому поддереву, если оно есть
    if (current.left != null) {
      current = current.left;
    }
    //----------- или пытаемся извлечь очередную вершину из стека
    else try {
      current = (Node)stack.pop();
    } catch (StackUnderflow ex) {
      //--------- Стек пуст, заканчиваем работу цикла и функции
      break;
    }
  }
}
```

Если `t` — дерево, содержащее в узлах символы, как показано на рис. 2.7, то фрагмент программы

```
t.traverseUpDown(new Visitor() {
  public void visit(Object obj) {
    System.out.print(obj);
  }
});
System.out.println();
```

вызовет печать следующей строки:

```
ABDGHCEFI
```

С помощью небольшой модификации итератора можно изменить порядок обхода на левосторонний инфиксный обход. Для этого вместо прохождения узла и помещения в стек его правого поддерева следует поместить в стек сам узел, а проходить его надо в момент извлечения узла из стека. Модифицированная функция показана в листинге 2.15.

Листинг 2.15. Левосторонний инфиксный обход дерева с хранением узлов в стеке

```
/*****************************************************************
 * Внутренний итератор для левостороннего обхода
```

```
 * с хранением узлов в стеке
 ****************************************************************/
public void traverseInfix(Visitor visitor) {
  // Стек для хранения информации
  IStack stack = StackFactory.createStack();
  // Текущая вершина
  Node current = root;
  // Основной цикл
  cycle:for (;;) {
    //----------- Кладем в стек текущую вершину
    stack.push(current);
    //----------- Переходим к левому поддереву, если оно есть
    if (current.left != null) {
      current = current.left;
    }
    //----------- или извлекаем очередную вершину из стека
    else do {
      try {
        current = (Node)stack.pop();
        //----------- и обходим текущую вершину
        visitor.visit(current.item);
      } catch (StackUnderflow ex) {
        break cycle;
      }
    } while ((current = current.right) == null);
  }
}
```

В обоих случаях максимальную требующуюся глубину стека можно оценить высотой дерева, т. к. в любой момент времени в стеке присутствуют узлы (или правые поддеревья этих узлов), находящиеся на пути из корня в некоторую вершину.

Внутренние итераторы получаются достаточно простыми, но пользоваться ими часто бывает неудобно. Рассмотрим теперь примеры внешних итераторов. Собственно, методика написания внешних итераторов не сильно отличается от методики написания внутренних итераторов. В обоих случаях организуется некоторая структура данных (обычно стек или очередь) для хранения пути от корня до текущей вершины, только если во внутреннем итераторе все данные создаются в виде локальных переменных функции, то

во внешнем итераторе те же данные будут сосредоточены в описании класса, реализующего итератор.

В качестве первого примера рассмотрим внешний итератор для нисходящего обхода "в ширину", использующий очередь для хранения вершин в порядке обхода.

Очередь идеально походит для организации обхода "в ширину". Действительно, по мере продвижения от корня дерева вниз в очередь будут попадать все более удаленные от корня вершины дерева. При этом, чем раньше вершина попала в дерево, тем раньше она будет обработана — это основной принцип хранения информации в структуре очереди. Рассмотрим опять дерево, приведенное на рис. 2.7, и покажем (табл. 2.2), как последовательно перебираются его вершины, если на каждом шаге проходится первая вершина из очереди, а в очередь кладутся для последующей обработки корни поддеревьев обрабатываемой вершины.

Таблица 2.2. *Последовательность шагов при обходе в ширину*

Номер шага	Текущая вершина	Содержимое стека	Пройденные вершины
1	*A*	(*B*, *C*]	
2	*B*	(*C*, *D*]	*A*
3	*C*	(*D*, *E*, *F*]	*A*, *B*
4	*D*	(*E*, *F*, *G*, *H*]	*A*, *B*, *C*
5	*E*	(*F*, *G*, *H*]	*A*, *B*, *C*, *D*
6	*F*	(*G*, *H*, *I*]	*A*, *B*, *C*, *D*, *E*
7	*G*	(*H*, *I*]	*A*, *B*, *C*, *D*, *E*, *F*
8	*H*	(*I*]	*A*, *B*, *C*, *D*, *E*, *F*, *G*
9	*I*	(]	*A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*
10	`null`	(]	*A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*

Как и раньше, будем предполагать, что внешний итератор описывается в виде некоторого класса, реализующего стандартный интерфейс `java.util.Iterator`, описание этого класса вложим в описание класса, реализующего дерево (`Tree`), а собственно метод, позволяющий организовать итерацию дерева, будет просто порождать и выдавать в качестве результата экземпляр такого класса. Для реализации очереди используется интерфейс `IQueue`, фабрика `QueueFactory` и исключительная ситуация `QueueUnderflow` *(см. разд. 2.2)*.

Определение класса приведено в листинге 2.16.

Листинг 2.16. Внешний итератор дерева для обхода в ширину

```
/*************************************************************
 * Внешний итератор для нисходящего обхода "в ширину"
 * с хранением узлов в очереди
 *************************************************************/
static private class BreadthFirst implements Iterator {
  IQueue queue = QueueFactory.createQueue();

  //--------- Конструктор
  public BreadthFirst(Tree t) { queue.enqueue(t.root); }

  //--------- Выдача очередного элемента
  public Object next() {
    try {
      // Очередной узел берем из очереди, если он там есть
      Node v = (Node)queue.dequeue();
      // Ставим в очередь корни поддеревьев этого узла
      if (v.left != null) { queue.enqueue(v.left); }
      if (v.right != null) { queue.enqueue(v.right); }
      // Выдаем элемент, хранящийся в очередном узле
      return v.item;
    } catch (QueueUnderflow ex) {
      // Если очередь пуста
      return null;
    }
  }

  //--------- Итерация закончена?
  public boolean hasNext() {
    return !queue.empty();
  }

  //--------- Пустая функция для обеспечения интерфейса итератора
  public void remove() {}
}
```

Функция, порождающая соответствующий итератор, теперь будет выглядеть совсем просто:

```
public Iterator traverseBreadthFirst() {
  return new BreadthFirst(this);
}
```

До сих пор мы все время говорили только о бинарных деревьях. В некоторой степени это оправдано, поскольку, как уже было сказано в *разд. 1.3*, в деревьях общего вида узлы могут содержать два указателя точно так же, как и в бинарных деревьях. Соответственно, все те способы обхода, которые применимы к бинарным деревьям, могут быть применены и к деревьям общего вида. Тем не менее, смысл, вкладываемый нами в эти обходы, может меняться, поскольку логическая структура дерева может не совпадать с его физической структурой.

Поясним сказанное на примере. Пусть имеется дерево, логическая структура которого показана на рис. 2.8.

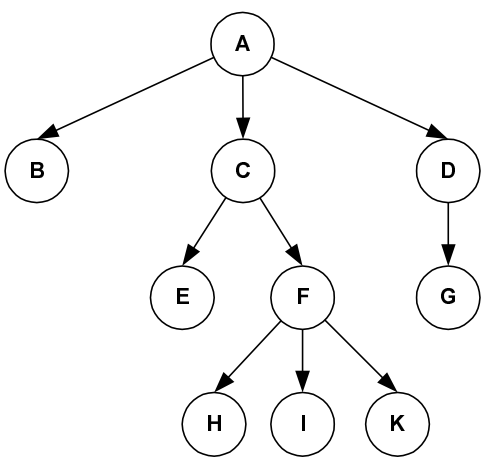


Рис. 2.8. Пример дерева общего вида для обхода

В памяти связи между узлами будут выглядеть несколько по-другому: каждый узел будет содержать указатели на первого непосредственного потомка ("старшего сына") и на соседний узел, находящийся на том же уровне иерархии ("брата"), так что физическая структура связей между теми же узлами будет выглядеть так, как на рис. 2.9 (логические связи, отсутствующие в физической структуре дерева, показаны пунктиром).

Если теперь рассмотреть какой-либо из порядков обхода узлов для исходного дерева, скажем, левосторонний обход "в ширину", то для второго дерева тот же порядок обхода уже не будет обходом "в ширину". Для исходного де-

рева узлы в порядке левостороннего обхода "в ширину" будут располагаться так:

A, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *K*,

но для второго дерева узлы в порядке обхода "в ширину" будут расположены уже по-другому:

A, *B*, *C*, *E*, *D*, *F*, *G*, *H*, *I*, *K*.

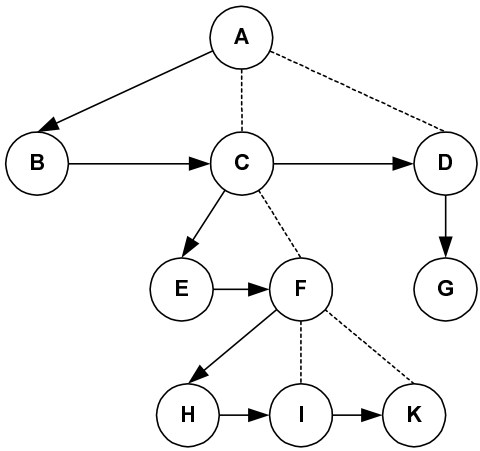


Рис. 2.9. Представление дерева общего вида в виде бинарного

Конечно, нетрудно написать функцию, которая будет представлять обход "в ширину" узлов исходного дерева, получая в качестве аргумента его физическое представление, такое, как на рис. 2.9, однако, реализация этой функции будет, разумеется, отличаться от реализации обхода "в ширину" для бинарного дерева. Тем не менее, для некоторых порядков обхода функции будут одинаковы в обоих случаях. Например, нисходящий левосторонний обход "в глубину" для обоих деревьев будет одним и тем же:

A, *B*, *C*, *E*, *F*, *H*, *I*, *K*, *D*, *G*.

Иногда очень важным оказывается вопрос о количестве дополнительной памяти, требующейся для организации обхода дерева. Так, например, в системах, реализующих язык программирования LISP, практически вся память, участвующая в работе программы, организована в виде деревьев, которые иногда требуется обходить в условиях недостатка памяти (например, для организации "уборки мусора"). Во всех приводимых ранее алгоритмах для выполнения обхода явно или неявно использовалась некоторая достаточно сложно организованная структура памяти — стек или очередь. Такая структура нужна для того, чтобы в процессе обхода помнить, какие элементы дерева еще не пройдены.

Конечно, можно каждый раз искать очередной непройденный элемент, исходя из того, какой узел дерева обходится в настоящий момент, но такой поиск очередного элемента "от корня" приводит к резкому замедлению скорости работы алгоритма. Компромиссом являются варианты, в которых в той или иной степени используются элементы памяти самого дерева. При этом обычно приходится отводить некоторую дополнительную память непосредственно в узлах дерева, так что какие-то потери памяти все равно остаются, однако их можно свести к минимуму: часто достаточно всего одного дополнительного бита памяти в каждом узле дерева для того, чтобы организовать обход дерева без использования дополнительных структур. Иногда такой дополнительный бит памяти можно получить "даром", если при хранении информации в дереве имеется некоторая избыточность. Например, если информация, хранимая в узле дерева, содержит неотрицательное целое число, то часто можно использовать знаковый разряд этого числа для хранения нужной информации. Можно также задействовать неиспользуемые пустые указатели на несуществующие поддеревья и узлы для хранения дополнительной информации.

Рассмотрим два способа обхода деревьев на основе внутренней информации узлов дерева для организации обхода. В первом способе основная идея состоит в том, чтобы задействовать пустые указатели в исходном дереве для организации перехода от "потомков" к "предкам". Во втором способе указатели трансформируются уже в процессе обхода: при движении по дереву "вниз" пройденные указатели заменяются указателями "наверх", к корню дерева; при движении по ним обратно происходит восстановление направленности указателей. В обоих случаях требуется наличие дополнительного бита, который будет представлен в программах дополнительным полем типа **boolean** в структуре узла дерева.

Сначала рассмотрим обход "сверху вниз" для дерева произвольной структуры (вообще говоря, не бинарного), представленного так, как показано на рис. 2.9.

Будем считать, что в узлах, представляющих самого младшего "брата", содержится ссылка на "родительский" узел. Для этого можно использовать свободную ссылку на "брата", а чтобы различить эти два способа применения ссылки, введем дополнительный признак в каждый узел. Получившееся представление дерева изображено на рис. 2.10. На нем "младшие братья" выделены штриховкой, а ссылки на родительские узлы представлены стрелками иного вида, чем основные ссылки. В листинге 2.17 в определении класса дерева признак "младшего брата" представлен логическим полем `youngest`, хотя в реальной ситуации для этого, скорее всего, будет использован какой-нибудь избыточный бит в представлении информации, хранящейся в узле дерева.

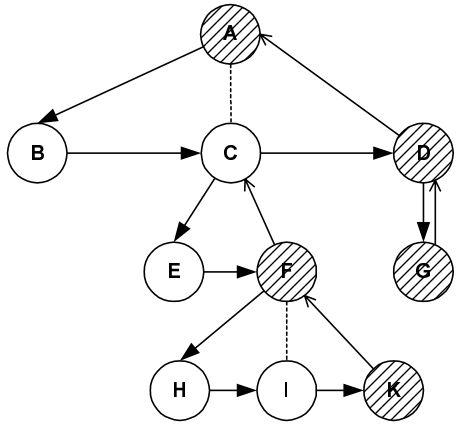


Рис. 2.10. Представление дерева с "обратными" ссылками

Для обхода построим внешний итератор дерева с описанием класса, вложенным в описание класса дерева. В этом итераторе переход к следующему элементу в порядке обхода осуществляется предельно просто: если у узла имеется "сын", то происходит переход к этому "сыну"; если имеется "брат", то выполняется переход к "брату", наконец, если нет и "брата", то происходит поиск "брата" у ближайшего предка. Обратите внимание, что поиск следующего узла осуществляется не в тот момент, когда этот узел становится действительно нужен, а "заранее", т. е. в тот момент, когда запрашивается предыдущий узел. Это дает возможность легко определить конец итерации.

Листинг 2.17. Обход дерева с использованием обратных указателей

```
public class Tree {
  // Представление узла несколько отличается от "стандартного"
  public static class Node {
    public Object item;              // содержимое узла
    public Node son = null;          // указатель на "старшего сына"
    public Node brother = null;      // указатель на "брата" или "отца"
    public boolean youngest = false; // признак того, что указатель
                                     // ссылается на "отца"

    public Node(Object item) {
      this.item = item;
    }

    public Node(Object item, Node son, Node brother, boolean youngest) {
      this.item = item;
```

```
      this.son = son;
      this.brother = brother;
      this.youngest = youngest;
    }
  }

  Node root = null;         // при создании дерево пусто

  // Определим итератор для обхода дерева сверху вниз
  static class UpDownIterator implements Iterator {
    Node current;           // ссылка на текущий узел
    // Конструктор итератора получает дерево для обхода
    // в качестве аргумента
    public UpDownIterator(Tree t) { current = t.root; }
    public boolean hasNext() { return current != null; }
    public Object next() {
      if (current == null) return null;
      Object result = current.item;  // для выдачи результата
      // теперь ищем следующий элемент
      if (current.son != null) current = current.son;
      while (current != null && current.youngest) {
        current = current.brother;   // переход вверх
      }
      if (current != null) current = current.brother;
      return result;
    }
    public void remove() {}
  }

  // Функция, выдающая итератор для обхода дерева
  // с учетом возможности перехода от потомков к предкам
  public Iterator upDownIterator() {
    return new UpDownIterator(this);
  }
}
```

Если дерево, изображенное на рис. 2.10, построено правильно, то его узлы будут пройдены в следующем порядке:

A, *B*, *C*, *E*, *F*, *H*, *I*, *K*, *D*, *G*,

что для этого дерева является естественным порядком обхода "сверху вниз".

Недостатком такого способа обхода является то, что необходимо аккуратно поддерживать правильную структуру ссылок в дереве. Если, например, некоторый узел удаляется, то необходимо проверить, не содержит ли удаляемый узел ссылку вверх по дереву на предка (то есть не является ли он "самым младшим братом"). Если это действительно так, то необходимо найти его ближайшего "старшего брата" (разумеется, с использованием той же самой ссылки вверх по дереву), и модифицировать представление этого "брата". Аналогично, необходимо аккуратно следить за корректностью представления и при добавлении узлов.

Второй способ обхода с использованием внутренней структуры дерева, представленный в листинге 2.18, свободен от этого недостатка. Нет необходимости как-то по-особому обрабатывать или представлять узлы дерева при этом способе обхода. Все, что необходимо иметь, — это свободный бит информации для использования его уже во время обхода. Недостаток этого способа обхода в другом: структура дерева динамически меняется во время его обхода. Это, в частности, означает, что для такого способа обхода внешние итераторы не подходят. Действительно, если обход будет прерван до его окончания, или во время обхода будет предпринята попытка изменить структуру дерева, или даже просто будет запущена вторая итерация — все это может привести к непредсказуемым последствиям. Поэтому второй способ обхода дерева с использованием его внутренней структуры будет представлен внутренним итератором. Кроме того, этот способ подходит для обхода бинарных деревьев.

На рис. 2.11 представлен промежуточный этап при обходе бинарного дерева, изображенного на рис. 2.7. При работе алгоритма используются следующие указатели:

- `processed` — указатель на последнюю обработанную вершину, содержащую указатель "вверх" по дереву;
- `current` — указатель на первую вершину еще не обработанной части дерева.

При движении вниз по дереву (от корня к листьям) некоторые указатели в дереве используются для того, чтобы запомнить обратный путь. При движении обратно значения указателей восстанавливаются. Дополнительный бит информации служит для того, чтобы запомнить, какой из двух указателей — левый или правый — используется для временного сохранения ссылки на родительский узел.

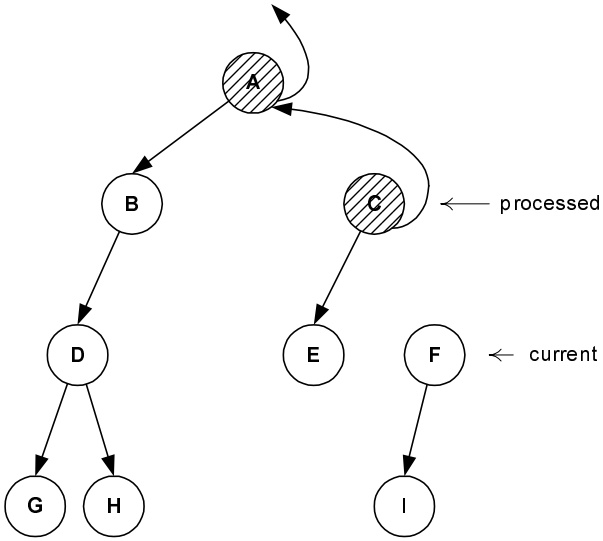


Рис. 2.11. Обход дерева с использованием внутренних указателей

На рис. 2.11 изображена ситуация, когда в двух узлах дерева — *A* и *C* — поля, в которых в нормальном состоянии записаны указатели на правое поддерево, используются для хранения ссылок на родительский узел.

В листинге 2.18 представлена операция, осуществляющая левосторонний инфиксный обход дерева с помощью внутреннего итератора. Это один из самых сложных алгоритмов, представленных в этой книге; для его успешного понимания рекомендуется попробовать проследить его работу на каком-нибудь простом примере.

Листинг 2.18. Левосторонний инфиксный обход с обращением ссылок

```
public class Tree {
  /*************************************************************
   * Определение узла включает дополнительный бит flag
   *************************************************************/
  static class Node {
    public Object item;             // содержимое узла
    public boolean flag = false;    // флажок для обхода
    public Node left = null;        // указатель на левое поддерево
    public Node right = null;       // указатель на правое поддерево

    // Конструктор узла
    public Node(Object item) {
```

```
      this.item = item;
    }
  }

  Node root = null;      // корень дерева

  public void traverseWithInversion(Visitor visitor) {
    Node processed = null;    // указатель вверх по дереву
    Node current = root;      // указатель на текущую вершину
    boolean down = true;      // направление движения
    // Цикл обхода узлов закончится, когда при движении вверх
    // окажется, что уже все узлы пройдены
    while (down || processed != null) {
      if (down) {
        if (current == null) {
          // Меняем направление движения
          down = false;
        } else {
          // Спускаемся вниз по дереву на один шаг
          Node w = current.left;
          current.left = processed;
          processed = current;
          current = w;
        }
      } else {
        if (processed.flag) {
          // Восстанавливаем указатель и продвигаемся вверх по дереву
          processed.flag = false;
          Node w = processed.right;
          processed.right = current;
          current = processed;
          processed = w;
        } else {
          // Посещаем вершину при переходе из левого поддерева в правое
          visitor.visit(processed.item);
          // Переходим к обработке правого поддерева
          Node w = processed.right;
          processed.flag = true;
```

```
            processed.right = processed.left;
            processed.left = current;
            current = w;
            // Снова двигаемся вниз
            down = true;
          }
        }
      }
    }
}
```

На этом мы закончим рассмотрение алгоритмов обхода деревьев. Некоторые из приведенных алгоритмов будут использованы позже в *главах* 3 и *4*.

2.4. Бинарные деревья поиска

Для того чтобы найти в дереве нужную информацию, можно перебрать все его узлы с помощью одного из алгоритмов обхода дерева. Однако, если объекты, хранящиеся в дереве, допускают линейное упорядочивание, то время поиска можно существенно сократить, если разместить узлы дерева в определенном порядке в соответствии с результатами сравнения объектов, хранящихся в узлах. Одним из самых распространенных способов подобного упорядочивания является такое расположение информации в узлах, при котором все значения, хранящиеся в левом поддереве некоторого узла меньше значения, хранящегося в этом узле, а значения, хранящиеся в правом поддереве, не меньше этого же значения. Если это действительно так, то дерево называют *бинарным деревом поиска*, а алгоритм поиска в таком дереве будет проходить только одну ветвь такого дерева, что может сократить время поиска с линейного до логарифмического.

В представлении дерева поиска будем считать, что в его узлах расположены объекты, интерфейс которых совместим с интерфейсом `Comparable`, т. е. помимо сравнения объектов с помощью метода `equals` допустимо их сравнение с помощью метода `compareTo`. При сравнении двух объектов `o1` и `o2` вызов `o1.compareTo(o2)` выдаст отрицательное значение, если объект `o1` в некотором смысле "меньше" объекта `o2`, выдаст нулевой результат, если объекты "равны", и положительное значение, если объект `o1` "больше" `o2`.

Алгоритм поиска объекта в бинарном дереве поиска ищет узел, содержащий объект со значением, равным некоторому заданному. Разумеется, равенство понимается в смысле, придаваемом ему методом `compareTo`. Приводя примеры обработки деревьев поиска, мы будем считать, что в его узлах расположены целые значения (объекты класса `Integer`). Разумеется, класс

`Integer` реализует интерфейс `Comparable`, так что эти объекты можно сравнивать с помощью операции `compareTo`, причем результат сравнения совпадает с ожидаемым.

На практике узлы дерева обычно содержат более сложные объекты. Например, если дерево используется для организации быстрого поиска в некотором большом массиве или файле (будем условно говорить о поиске в базе данных), то узел дерева будет содержать "ключ" (то значение, по которому производится поиск) и указатель на связанную с этим ключом запись в базе данных (индекс в массиве, номер записи в файле и т. п.). Сравнение объектов в узлах дерева в этом случае осуществляется "по ключу", т. е. операция `compareTo` выдает результат сравнения ключей.

На рис. 2.12 показан пример бинарного дерева поиска с целыми числами в узлах. При поиске в нем, скажем, значения 30, будут последовательно пройдены узлы со значениями 10, 20 и, наконец, 30. Результат поиска будет успешным. При поиске в нем же значения 18 будут последовательно пройдены узлы 10, 20, 15, после чего выяснится, что значения 18 в дереве нет, так что результат поиска окажется отрицательным.

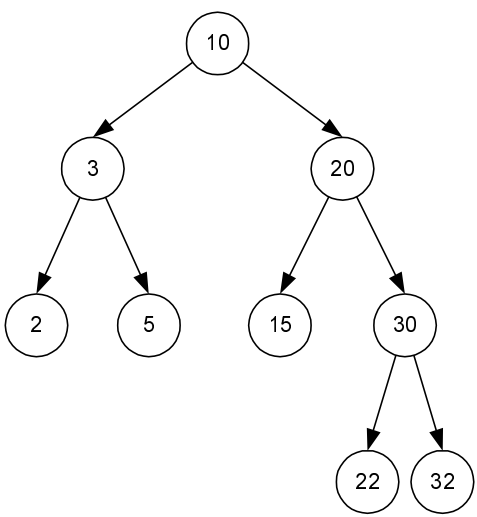


Рис. 2.12. Бинарное дерево поиска с целыми значениями в узлах

В листинге 2.19 представлено описание класса для бинарных деревьев поиска с определенной в ней операцией поиска. Операция выдает указатель на хранящийся в дереве объект в случае успешного завершения поиска, и пустой указатель **`null`** в случае, если значение в дереве не найдено.

Листинг 2.19. Описание класса для бинарного дерева поиска и операция поиска в нем

```
/************************************************
 * Бинарное дерево поиска с операцией поиска в нем
 ************************************************/
public class SearchTree {
  /*********************************************
   * Узел дерева содержит элемент Comparable
   *********************************************/
  private static class Node {
    Comparable item;
    Node left = null;
    Node right = null;

    // Конструктор узла с пустыми поддеревьями
    public Node(Comparable item) {
      this.item = item;
    }
  }

  Node root = null;     // Корень дерева; при создании дерево пусто

  /*****************************************************
   * Поиск заданного значения в дереве
   *****************************************************/
  public Comparable search(Comparable key) {
    for (Node current = root; current != null; ) {
      int comp = current.item.compareTo(key);
      if (comp == 0) {            // узел найден
        return current.item;
      } else if (comp < 0) {      // значение в корне меньше искомого
        current = current.right;  // переходим в правое поддерево
      } else {                    // значение в корне больше искомого
        current = current.left;   // переходим в левое поддерево
      }
    }
    return null;                  // узел не найден
  }
```

Следует заметить, что если попытаться изменить содержимое найденного узла, используя полученный в результате поиска указатель, то это может привести к нарушению структуры дерева поиска, если новое значение окажется не на "своем" месте. Лучше всего будет, если значение ключа в объектах, хранимых в дереве, вообще будет невозможно изменить. Это, в частности справедливо для всех стандартных простых классов в языке Java: в классах `Integer`, `Character`, `Double`, `String` и других не определены методы, позволяющие изменить значение представляемого объекта.

Левосторонний инфиксный обход бинарного дерева поиска позволяет посетить его узлы в порядке возрастания значений. В следующем примере такой обход используется для того, чтобы построить внешнее представление последовательности значений, хранящихся в дереве, в виде строки. Отдельные узлы представляются в виде строки с помощью стандартного способа преобразования их в строку методом `toString`, а представление в виде строки всего дерева составлено из представлений его узлов в порядке возрастания значений, которые разделены пробелами и заключены в квадратные скобки. В листинге 2.20 представлен метод `toString` для бинарного дерева поиска в предположении, что для него уже определен внешний итератор

```
public Iterator iteratorInfix();
```

выдающий узлы дерева в порядке левостороннего инфиксного обхода.

Листинг 2.20. Представление дерева в виде строки

```
public String toString() {
  String res = "[ ";
  for (Iterator i = iteratorInfix(); i.hasNext(); ) {
    res += i.next().toString() + " ";
  }
  return res + "]";
}
```

Для деревьев поиска наиболее важным является вопрос о способах модификации дерева при включении новых узлов и удалении имеющихся. При всех таких изменениях должна быть сохранена структура дерева поиска, т. е. новый узел не должен нарушать основное правило: в каждом узле значение должно быть больше значений узлов левого поддерева и меньше значений узлов правого поддерева. При удалении узлов также иногда требуется перестраивать дерево, так что и в этом случае необходимо следить за сохранением структуры.

Прежде всего, опишем два алгоритма добавления нового узла в дерево поиска. Первый из них образует в дереве новый лист, при этом вся остальная

структура дерева сохраняется. Второй алгоритм образует новый корень дерева, при этом структура остальной части дерева частично меняется.

При добавлении узла в лист дерева нужно лишь обеспечить, чтобы этот новый лист мог быть впоследствии эффективно найден функцией поиска. Таким образом, алгоритм добавления может работать практически так же, как и при поиске, только в последний момент вместо того, чтобы сообщать о том, что искомого узла в дереве нет, надо добавить новый узел в то самое место, где он должен находиться. Еще одна небольшая трудность состоит в правильной обработке узла, значение которого уже есть в дереве. Здесь возможны два варианта: либо новый узел с тем же самым значением ключа все же добавляется в дерево (впоследствии такой узел никогда не будет найден нашей процедурой поиска, поскольку она всегда ищет только первый из узлов с заданным значением), либо новый узел не добавляется в дерево вообще.

В листинге 2.21 представлена функция добавления нового значения в бинарное дерево поиска `SearchTree`, при этом даже если такое значение уже есть в дереве, то новый узел все равно создается и включается в дерево. В *разд. 3.3* будет представлен и другой способ добавления значения в дерево поиска с учетом особенностей этого дерева. Функция `insertLeaf`, реализующая такую вставку, считается включенной в описание класса `SearchTree` в качестве его метода.

Листинг 2.21. Включение нового узла в дерево поиска

```
public void insertLeaf(Comparable item) {
  Node newNode = new Node(item);  // новый узел дерева
  Node parent = null;             // родительский узел при поиске места
  Node current = root;            // текущий узел при поиске места
  int compare = 0;                // результат сравнения значений
  // Цикл поиска места вставки нового узла
  while (current != null) {
    parent = current;
    if ((compare = current.item.compareTo(item)) <= 0) {
      current = current.right;
    } else {
      current = current.left;
    }
  }
  // поиск закончен
  if (parent == null) {          // только если исходное дерево пусто
    root = newNode;              // новый узел становится корнем
```

```
    } else if (compare <= 0) {
      parent.right = newNode;      // корень вставляется в правое поддерево
    } else {
      parent.left = newNode;       // корень вставляется в левое поддерево
    }
}
```

Проверьте, что дерево, изображенное на рис. 2.12, может быть получено с помощью следующей последовательности операций:

```
SearchTree t = new SearchTree();
t.insertLeaf(new Integer(10));
t.insertLeaf(new Integer(3));
t.insertLeaf(new Integer(20));
t.insertLeaf(new Integer(15));
t.insertLeaf(new Integer(30));
t.insertLeaf(new Integer(2));
t.insertLeaf(new Integer(5));
t.insertLeaf(new Integer(32));
t.insertLeaf(new Integer(22));
```

После этого содержимое дерева может быть распечатано с помощью вызова

```
System.out.println(t);
```

поскольку для дерева определена операция `toString()`.

Описанная процедура добавления нового узла в дерево поиска обладает тем недостатком, что новый узел оказывается лежащим далеко от корня, хотя довольно часто бывают ситуации, когда поиск среди узлов, добавленных последними, производится гораздо чаще, чем поиск более "старых" узлов. В этих условиях было бы лучше добавлять новые узлы в корень дерева, а не в листья. К счастью, существует достаточно простой и практически такой же эффективный алгоритм для добавления нового узла в корень дерева, как и для добавления в лист. В этом алгоритме исследуется та же последовательность узлов, что и при добавлении в лист (и, разумеется, та же самая, что и при поиске узла), но структура дерева меняется более существенно. В процессе работы алгоритма отслеживаются позиции узлов, в которых производятся изменения. На рис. 2.13 показана последовательность состояний двоичного дерева поиска при добавлении в него нового узла со значением 12 в корень дерева. Кружочками со словами "меньше", "больше" отмечены ссылки в узлах дерева, в которых могут происходить изменения на следующем шаге работы алгоритма. Если текущий рассматриваемый узел оказывается меньше нового значения, то он помещается в позицию "меньше", а если он оказывается больше нового значения, — то в позицию "больше". Позиции этих ссылок отмечены в программе (листинг 2.22) парами переменных

(leftNode, leftPos) и (rightNode, rightPos). Текущий рассматриваемый узел дерева представлен на рис. 2.13 и в программе указателем current.

Листинг 2.22. Вставка нового узла в корень дерева

```
public void insertRoot(Comparable item) {
  final int LEFT = 1;              // индикатор вставки в левое поддерево
  final int RIGHT = 2;             // индикатор вставки в правое поддерево

  Node current = root;             // текущий рассматриваемый узел
  Node nodeLeft = new Node(item); // узел позиции "меньше"
  int leftPos = LEFT;
  Node nodeRight = nodeLeft;       // узел позиции "больше"
  int rightPos = RIGHT;
  // Новый узел становится корнем дерева
  root = nodeLeft;
  // Цикл просмотра узлов дерева
  while (current != null) {
    if (current.item.compareTo(item) < 0) {
      // Вставка текущего узла в позицию "меньше"
      if (leftPos == LEFT)
          nodeLeft.left = current;
      else
          nodeLeft.right = current;
      nodeLeft = current;
      leftPos = RIGHT;
      current = current.right;
    } else {
      // Вставка текущего узла в позицию "больше"
      if (rightPos == LEFT)
          nodeRight.left = current;
      else
          nodeRight.right = current;
      rightPos = LEFT;
      nodeRight = current;
      current = current.left;
    }
  }
```

```
    // "Очистка" позиций "меньше" и "больше"
    if (leftPos == LEFT)
        nodeLeft.left = null;
    else
        nodeLeft.right = null;
    if (rightPos == LEFT)
        nodeRight.left = null;
    else
        nodeRight.right = null;
}
```

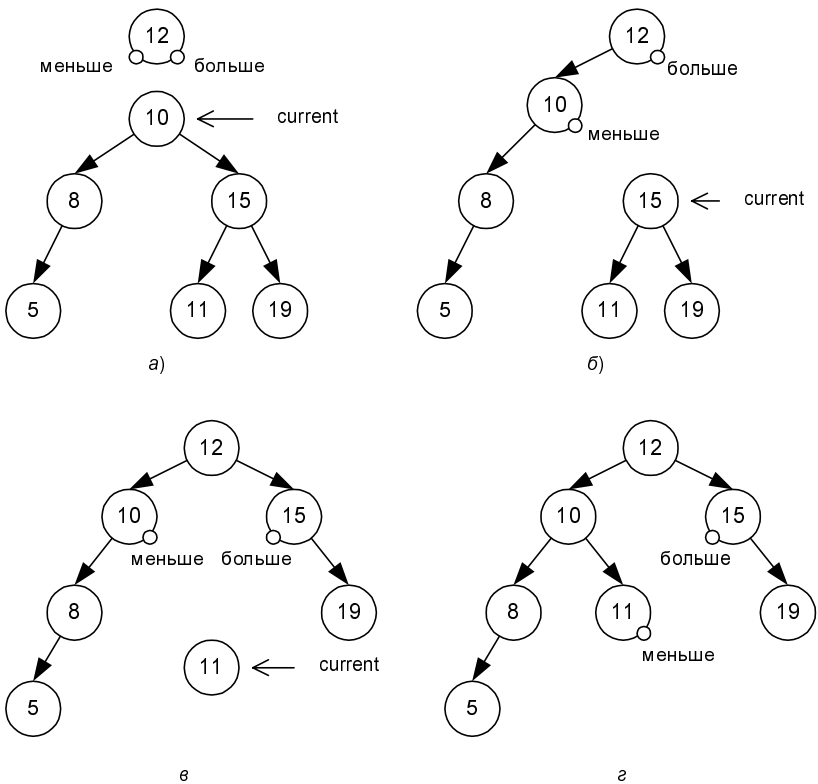


Рис. 2.13. Последовательные состояния дерева при вставке нового узла в корень

Вторая задача, которую надо решить для обеспечения нормальной практической работы с деревом поиска, — это удаление узлов из дерева. Если эта задача будет решена, то третья имеющаяся возможность — перемещение узла по дереву из-за изменения его значения — может быть решена удалением одного узла и вставкой нового вместо него.

Проще всего оказывается удалить узел, если он находится в листе дерева. В этом случае достаточно лишь убрать из дерева указатель на этот узел. Разумеется, сначала придется произвести поиск в дереве, как обычно. Не очень сильно отличается от этого и случай, при котором удаляемый узел имеет лишь один указатель на поддерево. Очевидно, что и в такой ситуации для удаления узла достаточно изменить лишь один указатель, заменив указатель на удаляемый узел указателем на его единственное поддерево. На рис. 2.14 представлена ситуация до и после удаления такого узла из дерева (удаляемый узел выделен штриховкой).

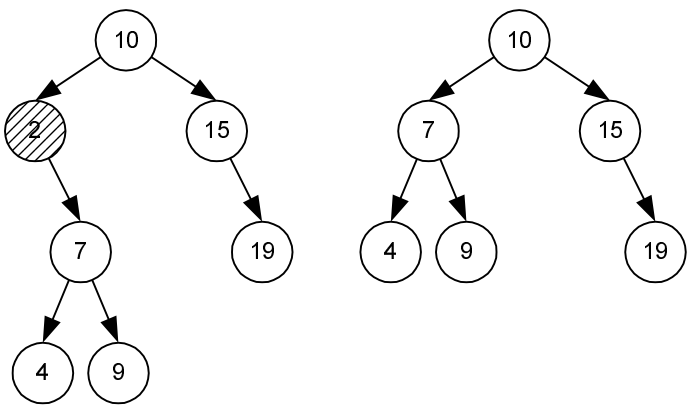


Рис. 2.14. Удаление узла из бинарного дерева поиска

Для того чтобы решить задачу удаления узла в общем виде, ее сводят к одной из описанных простых ситуаций. Для этого в одном из поддеревьев удаляемого узла находят узел, значение которого наиболее близко к значению в удаляемом узле: максимальное значение в левом поддереве или минимальное значение в правом поддереве. Очевидно, что в обоих случаях такое ближайшее значение обязательно будет расположено либо в листе дерева, либо в узле, имеющем только одно поддерево. После этого делают подмену: удаляют вместо нужного узла один из найденных ближайших узлов, а значение, которое в нем содержалось, переносят в узел, который требовалось удалить. Очевидно, что структура дерева поиска при этом не нарушится.

Процесс удаления узла по описанному алгоритму изображен на рис. 2.15.

Функция для удаления узла из бинарного дерева поиска оказывается не такой простой, как функция вставки, поскольку приходится рассматривать различные случаи при удалении узла. Эта функция представлена в качестве метода класса `SearchTree` в листинге 2.23. Для определенности в ней всегда в качестве узла-заместителя для удаляемого ищется узел в правом поддереве, т. е. узел, значение которого является ближайшим к значению удаляемого узла сверху.

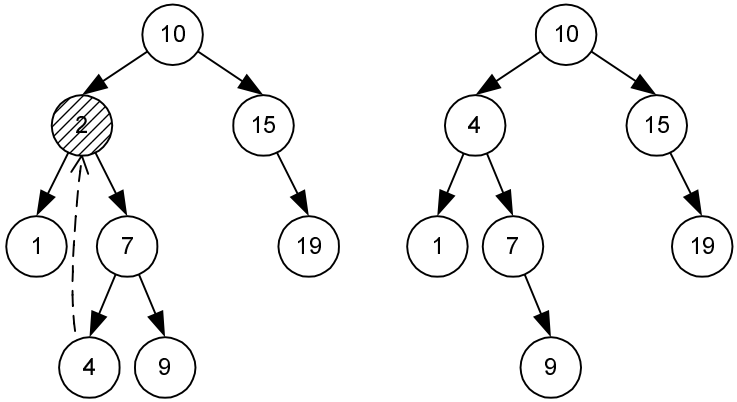


Рис. 2.15. Удаление узла из бинарного дерева поиска

Листинг 2.23. Функция удаления узла из бинарного дерева поиска

```
/*************************************************************************
 * Функция удаления узла с заданным значением из бинарного дерева поиска
 *************************************************************************/
public void remove(Comparable item) {
  // Константы, задающие направление последнего шага
  final int GO_LEFT = -1;
  final int GO_RIGHT = 1;
  final int NO_STEP = 0;

  Node parent = null;        // узел, в котором будет изменен указатель
  Node current = root;       // текущий, впоследствии удаляемый, узел
  int comp = 0;              // результат сравнения значений узлов
  int lastStep = NO_STEP;    // направление последнего шага
  // 1. Поиск удаляемого узла
  while (current != null && (comp = current.item.compareTo(item)) != 0) {
    parent = current;
    if (comp < 0) {
      lastStep = GO_RIGHT;
      current = current.right;
    } else {
      lastStep = GO_LEFT;
      current = current.left;
```

```
    }
  }
  // 2. Разбор простых случаев
  if (current == null) return;       // удаляемое значение не найдено
  if (current.left == null) {        // левое поддерево отсутствует
    if (lastStep = GO_RIGHT) {       // узел лежит справа от родительского
      parent.right = current.right;
    } else if (lastStep = GO_LEFT) {// узел лежит слева от родительского
      parent.left = current.right;
    } else {                         // родительский узел отсутствует
      root = current.right;
    }
  } else if (current.right == null) { // правое поддерево отсутствует
    if (lastStep = GO_RIGHT) {       // узел лежит справа от родительского
      parent.right = current.left;
    } else if (lastStep = GO_LEFT) {// узел лежит слева от родительского
      parent.left = current.left;
    } else {                         // родительский узел отсутствует
      root = current.left;
    }
  } else {
  // 3. Разбор общего случая: оба поддерева не пусты
    Node nodeToReplace = current.right;   // узел-заместитель удаляемого
    parent = current;
    // Цикл поиска узла-заместителя в правом поддереве
    while (nodeToReplace.left != null) {
      parent = nodeToReplace;
      nodeToReplace = nodeToReplace.left;
    }
    // Замещение значения в удаляемом узле
    current.item = nodeToReplace.item;
    // Физическое удаление узла-заместителя из дерева
    if (parent == current) {   // заместитель - корень правого поддерева
      parent.right = nodeToReplace.right;
    } else {                   // заместитель лежит в поддереве ниже
      parent.left = nodeToReplace.right;
    }
  }
}
```

Бинарные деревья поиска могут применяться для упорядочения информации. Для того чтобы расположить некоторые элементы в порядке возрастания, можно выстроить их в бинарное дерево поиска, а затем, обойдя это дерево в левостороннем порядке, получить отсортированную последовательность.

Пусть, например, требуется расположить все слова в некотором тексте в алфавитном порядке, т. е. построить таблицу всех слов, встречающихся в заданном тексте. Для того чтобы выделять слова из текста, можно воспользоваться стандартным классом `java.util.StringTokenizer`, который можно рассматривать как внешний итератор строки, позволяющий выделять в строке слова, отделенные друг от друга последовательностью "разделителей" — символов, заданных при создании этого итератора. Класс `StringTokenizer` является реализацией интерфейса `java.util.Enumeration` — одним из двух интерфейсов, используемых нами в качестве интерфейсов итерации. Разделителями слов будем считать пробелы и другие "пустые" символы (символы табуляции, перевода строки и т. п.), а также знаки пунктуации. Различные формы одного и того же слова будем считать разными словами, но если слово встречается в тексте многократно, то повторно записывать его в дерево не станем. Для этого воспользуемся модифицированной версией операции вставки элемента в дерево поиска, в которой при обнаружении в дереве элемента со значением, равным вставляемому, происходит немедленный выход из функции.

Функция, решающая эту задачу, приведена в листинге 2.24. Она получает строку текста в качестве аргумента и выводит последовательность слов, составляющую этот текст, в выходной поток, заданный вторым аргументом функции. Функция пользуется бинарным деревом поиска `SearchTree` и его операциями `insertLeaf` для вставки нового слова в дерево и `iteratorInfix` для обхода узлов в левостороннем инфиксном порядке. Все слова приводятся к нижнему регистру букв с помощью вызова метода `toLowerCase()`.

Листинг 2.24. Вывод слов заданного текста в лексикографическом порядке

```
public static void sortWords(String text, java.io.PrintStream out) {
  String delimeters = ",.?!;:-\'\" \t\n\r\f";   // все разделители
  StringTokenizer tok = new StringTokenizer(text, delimeters);
  SearchTree tree = new SearchTree();

  // Заполнение дерева словами
  while (tok.hasMoreTokens()) {
    tree.insertLeaf(tok.nextToken().toLowerCase());
  }
```

```
  // Вывод слов в выходной поток
  for (Iterator i = tree.iteratorInfix(); i.hasNext(); ) {
    out.println(i.next());
  }
}
```

Если вызвать эту функцию для некоторого текста и направить вывод результата в стандартный выходной поток, скажем, следующим образом:

```
sortWords("сэр исаак ньютон по секрету признавался друзьям, что он " +
          "знает, как гравитация ведет себя, но не знает, почему",
          System.out);
```

то в выходном потоке мы получим следующий результат:

```
ведет
гравитация
друзьям
знает
исаак
как
не
но
ньютон
он
по
почему
признавался
себя
секрету
сэр
что
```

К сожалению, не всегда есть гарантия, что при вставках новых узлов в дерево их последующий поиск будет быстрым. Скорость поиска существенно зависит от того, насколько равномерно расположены узлы в дереве. Например, при обработке вышеприведенного текста после вставки первых шести слов структура получившегося дерева будет такой, как изображено на рис. 2.16, и хотя после ввода всего текста структура дерева несколько улучшилась (рис. 2.17), все же она далека от оптимальной.

Действительно, если произвести некоторые подсчеты, то окажется, что для поиска слова в получившемся дереве в среднем потребуется сделать 4,3 шага. В то же время, если бы те же слова были расположены в дереве оптимальной структуры (например, как на рис. 2.18), то для поиска слова

требовалось бы сделать в среднем всего 3,4 шага, что вполне понятно, поскольку дерево рис. 2.17 имеет 7 уровней, а дерево рис. 2.18 — всего 5.

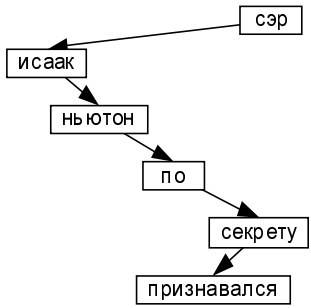


Рис. 2.16. Дерево слов после вставки первых шести слов

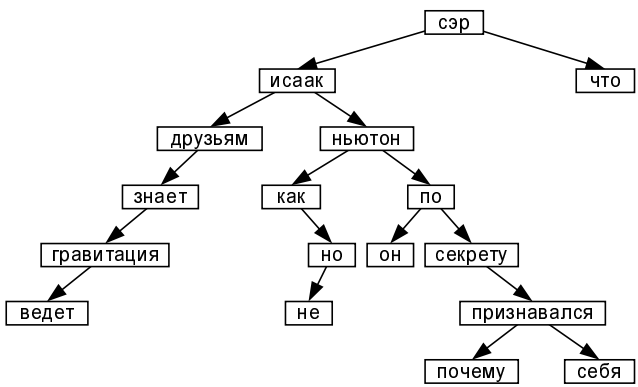


Рис. 2.17. Дерево слов после вставки всех слов

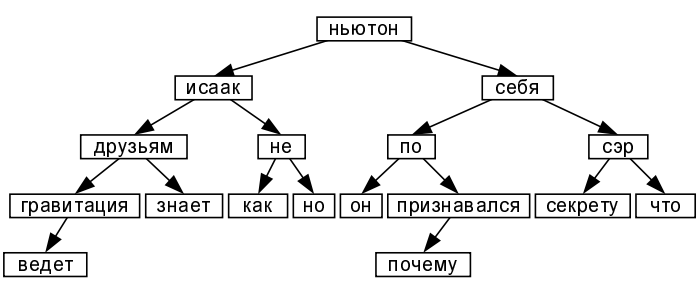


Рис. 2.18. Оптимальное дерево слов

Оптимальным для поиска будет дерево минимальной высоты, у которого к тому же узлы всех уровней, кроме, может быть, двух последних, имеют непустые указатели на оба поддерева. У оптимального дерева на уровне *n* на-

ходятся 2^{n-1} узла, и только на последнем уровне узлов может быть меньше. Соответственно, если в оптимальном дереве содержатся n узлов, то поиск в таком дереве производится приблизительно за $\log_2 n$ шагов.

Было бы хорошо, если в процессе вставки элементов удалось бы поддерживать дерево все время в оптимальном виде. К сожалению, это заняло бы столько времени, что свело бы на нет всю выгоду от быстрого поиска.

Тем не менее, имеются схемы, при которых удается поддерживать структуру дерева поиска более или менее приемлемой с точки зрения эффективности поиска, причем алгоритмы вставки и удаления лишь незначительно усложняются. Наиболее известными из таких схем являются АВЛ-деревья, прекрасно описанные во многих книгах, таких как, например [2, 6, 8], и 2-3-деревья, также многократно описанные и рассматриваемые нами далее.

Идея 2-3-дерева состоит в том, что структура бинарного дерева несколько усложняется за счет ввода специальных узлов, имеющих два значения в узле вместо одного и, соответственно, три поддерева вместо двух. За счет этого удается поддерживать структуру, при которой все листья дерева оказываются лежащими на одном и том же уровне, а все вышерасположенные узлы имеют, по крайней мере, два непустых поддерева.

Более строгое определение 2-3-дерева состоит в следующем.

- Каждый узел 2-3-дерева содержит одно или два значения.
- Узлы дерева делятся на две категории — листья и промежуточные узлы, причем если промежуточный узел содержит одно значение, то он имеет два непустых поддерева (2-узел), а если он содержит два значения, то он имеет три непустых поддерева (3-узел).
- Принцип упорядоченности значений сохраняется для 2-3-дерева в следующем виде. Для 2-узла, как и в случае бинарного дерева поиска, все значения, лежащие в левом поддереве, имеют значения, меньшие значения, хранящегося в узле, а значения, лежащие в правом поддереве, — больше или равны значениям, хранящимся в узле. Для 3-узла упорядоченность означает следующее: если ключи, хранящиеся в нем, имеют значения K_1, K_2, а поддеревья этого узла обозначены T_1, T_2 и T_3, то справедливо неравенство $T_1 < K_1 \le T_2 < K_2 \le T_3$, где неравенство вида $T_i < K_i$ указывает, что значения всех ключей, лежащих в дереве T_i, меньше значения ключа K_i, а неравенство вида $K_{i-1} \le T_i$ показывает, что значение ключа K_{i-1} не больше значений всех ключей дерева T_i.
- Все листья лежат на одном и том же уровне.

На рис. 2.19 представлен пример 2-3-дерева, содержащего в узлах целые числа. А дерево на рис. 2.20 не является 2-3-деревом: нарушено условие (2).

Физически часто 2-3-дерево представляют в виде обычного бинарного дерева, как показано на рис. 2.21. Чтобы отличить указатель на поддерево от указателя на соседнее значение, логически принадлежащее тому же узлу, вводится специальный флажок. На рис. 2.21 указатели разных типов пред-

ставлены стрелками с разными концами. Заметьте, что поиск в 2-3-дереве, представленном в виде бинарного дерева, можно производить точно так же, как и в обычном бинарном дереве поиска. Однако, если с точки зрения логической структуры дерево, изображенное на рис. 2.21, имеет высоту, равную 3, то с точки зрения его физической структуры как бинарного дерева оно имеет высоту, равную 5. Оптимальное бинарное дерево, содержащее те же значения, могло бы иметь высоту, равную 4.

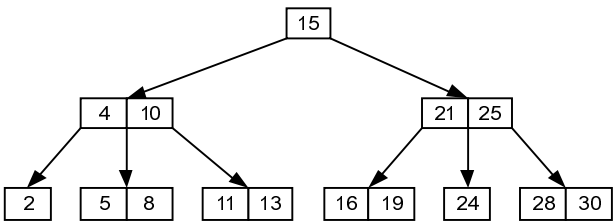


Рис. 2.19. Пример 2-3-дерева

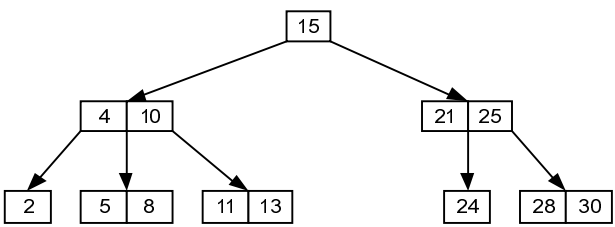


Рис. 2.20. Пример нарушения структуры 2-3-дерева

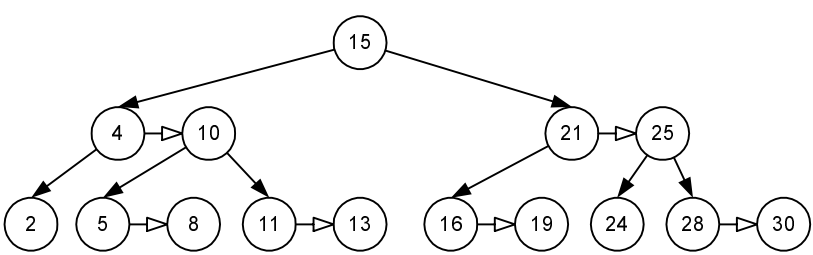


Рис. 2.21. Физическая структура 2-3-дерева

Вставка в 2-3-дерево осуществляется только в листья. При этом сначала производится поиск листа, в который можно добавить новое значение с сохранением структуры поиска, и производится вставка почти как в обычное бинарное дерево поиска. Может оказаться, что при этом структура 2-3-дерева осталась не нарушенной. Так, например, после вставки узла со значением 23 в дерево, изображенное на рис. 2.21, получившееся дерево снова

будет 2-3-деревом, так что никаких дальнейших преобразований производить не надо (рис. 2.22).

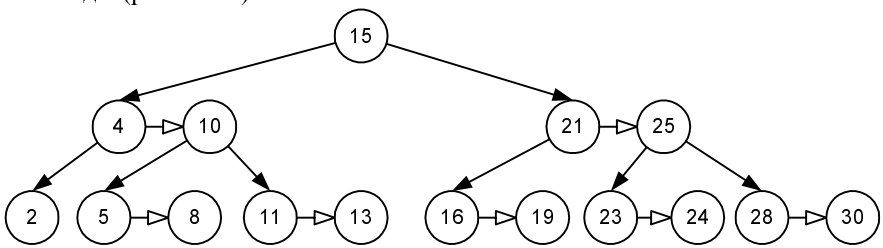


Рис. 2.22. 2-3-дерево после вставки элемента 23

Однако при вставке в то же самое дерево значения 12 после вставки нового значения в лист образуется переполнение узла. Для того чтобы исправить ситуацию, узел делится на два новых, при этом среднее из трех значений, образующих переполненный узел, поднимается на предыдущий уровень и присоединяется к узлу, лежащему на предыдущем уровне. Разумеется, на предыдущем уровне вновь может возникнуть переполнение узла, и тогда операция расщепления узла повторяется снова. На рис. 2.23 показаны последовательные стадии преобразования дерева, изображенного на рис. 2.21, при добавлении в него нового узла со значением 12. "Нестабильные" узлы выделены пунктирной границей.

Алгоритм вставки не очень сложно запрограммировать, но поскольку разных случаев при преобразованиях структуры оказывается довольно много, программа получается слишком длинной, так что мы не приводим ее здесь. Интересующихся читателей можем отослать к книге [7], где соответствующие алгоритмы приведены в более общем виде для B-деревьев, являющихся обобщением 2-3-деревьев.

Удаление элемента из 2-3-дерева выполняется в обратном порядке: там, где при добавлении элемента происходило расщепление — при удалении производится склеивание. Некоторое усложнение возникает за счет того, что, во-первых, удаление, как и в случае бинарных деревьев, надо производить из листа, а для этого может потребоваться найти сначала элемент-заместитель, а во-вторых, кроме склеивания элементов добавляется еще один способ исправления структуры дерева — переливание.

На рис. 2.24 показана последовательность состояний 2-3-дерева при удалении элемента:

- удалению подлежит элемент 10 (рис. 2.24, *а*). Он находится не в листе дерева, поэтому ищем заместителя этого элемента — ближайшее наибольшее значение. Таким значением оказывается 11;
- значение 11 переместилось в корень дерева, а соответствующее значение удалено из дерева (рис. 2.24, *б*). Структура 2-3-дерева нарушилась, поскольку узел 12 теперь не имеет левого поддерева. Чтобы восстановить

структуру, осуществляется попытка склеить два значения — 12 и 13 — в один узел, а узел 12 удалить из дерева;

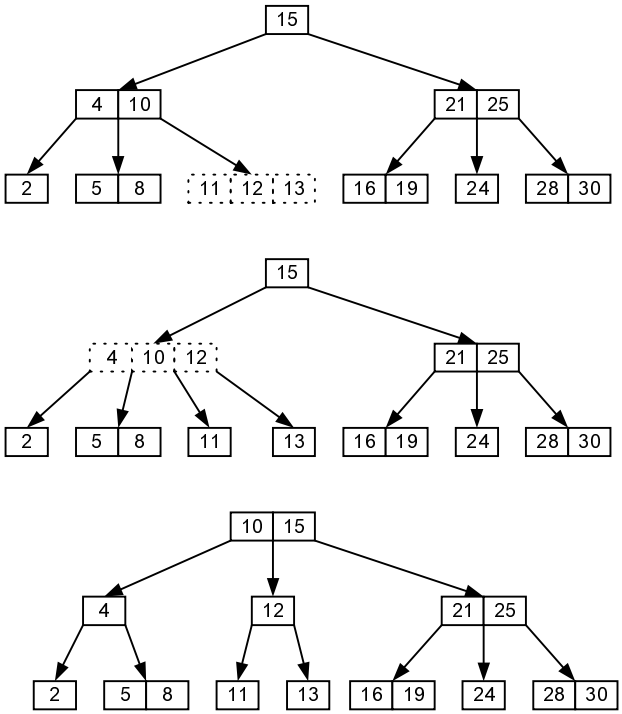


Рис. 2.23. Последовательные стадии преобразования 2-3-дерева при вставке нового узла

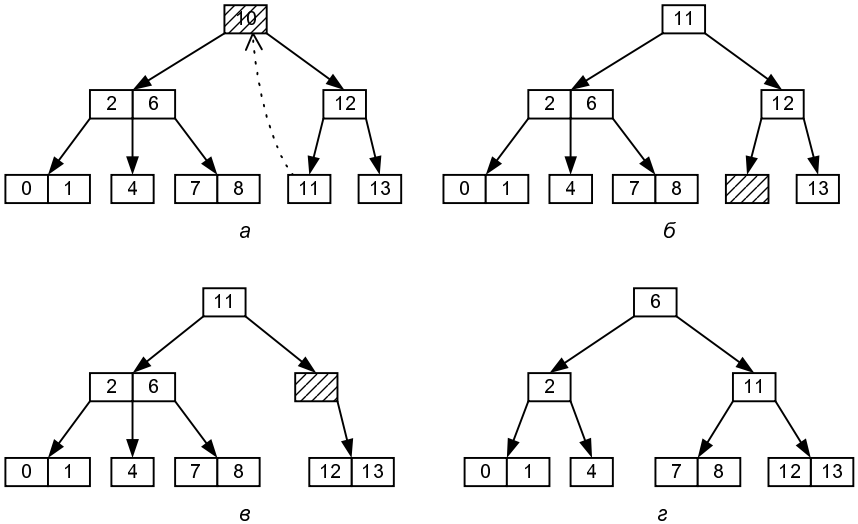


Рис. 2.24. Удаление узла из 2-3-дерева

- склейка успешно произведена (рис. 2.24, *в*), но узел из дерева удалить безболезненно опять не удается — образовался пустой узел. Склейка теперь тоже не получится — значение 11 надо было бы склеивать с узлом, уже содержащим два значения — 2 и 6. Поэтому выбран другой метод преобразования — переливание. Значение 6 поднимается в корень на место значения 11, а значение 11 опускается в пустой узел. Ссылка на узел, содержащий значения 7 и 8, перемещается вместе со значением;
- преобразования завершены (рис. 2.24, *г*); структура 2-3-дерева восстановлена.

На этом мы закончим обзор методов работы с деревом поиска. Позже мы ненадолго вернемся к этой структуре при обсуждении методов представления словаря.

Глава 3

Обработка текста

Обработка строк является важнейшей составной частью работы со структурами данных. Очень часто именно удобство работы со строками определяет выбор языка пользователем. Имеется много языков программирования, ориентированных именно на работу со строками, но часто в таких языках отсутствуют развитые средства работы с другими типами данных. Яркими представителями подобных языков являются Рефал и Снобол. В то же время в традиционных "универсальных" языках программирования, таких как Си, Паскаль, Java, средства работы со строками присутствуют в весьма ограниченном количестве. Как правило, предоставляются методы для склеивания строк, поиска подстрок, вырезания фрагментов строк. Иногда предоставляются простые способы для модификации строк — вставка новых фрагментов в заданную строку, добавление символов и строк в конец заданной строки и т. п. Часто имеются также средства для преобразования других объектов языка в строковое представление и обратно.

Тем не менее, для серьезной обработки строк этого недостаточно. Прежде всего, необходимо тщательно продумать механизмы распределения памяти под строковые объекты. Если при каждом добавлении одного символа в конец имеющейся длинной строки происходит отведение памяти под всю строку, то эффективность работы программы от этого очень страдает. Далее, часто желательно иметь средства для сравнения строк с заданным образцом, быстрые операции поиска подстроки в длинном тексте и т. д.

К счастью, "универсальные" языки программирования предоставляют необходимые возможности для организации собственных типов и данных и методов работы с ними. Ниже рассматриваются несколько способов организации строковых объектов в языке программирования. Как и везде в книге, в качестве инструментального средства используется язык Java, однако похожим образом можно организовать данные и на любом другом языке, поддерживающем определение собственных структур данных.

3.1. Способы представления строк

Строки в языке Java представляют собой константные (неизменяемые) объекты. Каждый раз, когда требуется изменить значение имеющейся строки, по существу приходится строить новую строку. В стандартном классе `String`

имеются операции для выделения отдельных символов и подстрок в строке (`charAt`, `substring`), однако нет операций для замены в строке отдельных символов и подстрок. Разумеется, это не случайно. Строки, как и другие базовые объекты классов `Integer`, `Character` и других, рассматриваются языком как неизменяемые, и это позволяет переопределять одно и то же значение в переменных и других структурах данных без опасений, что значение может неожиданно измениться. Например, во фрагменте программы

```
String s1 = "Hello, world!";
String s2 = s1;
```

переменные `s1` и `s2` содержат ссылки на один и тот же объект, и можно быть уверенным, что как бы ни работала программа с переменной `s1`, значение переменной `s2` всегда останется тем же, каким оно было в момент инициализации, до тех пор, пока это значение не будет изменено явно присваиванием переменной нового значения.

В языке Java есть еще один стандартный класс — `StringBuffer`, представляющий модифицируемые строки. Уже из его названия очевидно, что под строку в объектах этого класса отводится некоторый буфер, внутри которого и располагается собственно строка. Длина буфера может задаваться при создании объекта, и впоследствии эта длина может изменяться программой или самим объектом, если в результате модификации строки оказывается, что длины буфера недостаточно для ее представления. С помощью класса `StringBuffer` можно выполнять такие действия над строками, как изменение отдельного символа в строке (`setCharAt`), вставка подстроки в заданное место буфера (`insert`) и замена подстроки из буфера другой строкой (`replace`).

Если в приведенном выше примере заменить описатель `String` на `StringBuffer`, то изменения в содержимом строки, сделанные с помощью одной переменной, будут видны и при использовании второй переменной. Например, в следующем фрагменте программы

```
StringBuffer s1 = "Hello, world!";
StringBuffer s2 = s1;
s1.replace(7, 12, "Russia");
System.out.println(s2);
```

замена была произведена в строке `s1`, тем не менее, при последующем выводе строки `s2` в стандартный выходной поток будет напечатана измененная строка `"Hello, Russia!"`.

Во всех случаях реализации строки в виде массива обычно используют один из двух способов внутреннего представления строки: представление с хранимой длиной и представление с символом-терминатором.

В *представлении с хранимой длиной* в дополнение к элементам строки — символам, составляющим строку, — дополнительно хранится длина этого массива — количество символов в строке. Такой способ представления позво-

ляет отделить информацию о символах строки от информации о длине массива, в котором эти символы хранятся. Так, например, в языках Turbo Pascal и его потомках (Borland Pascal, Object Pascal) стандартной реализацией коротких строк является массив из 256 элементов, в первом элементе которого хранится длина строки, содержащейся в этом массиве. Таким образом, один и тот же массив можно использовать для хранения в нем строк различной длины (до 255 символов).

Вообще, в языке Turbo Pascal строка может рассматриваться не только как самостоятельный тип данных, но и как обычный массив символов. Таким образом, в этом языке для строк нарушается принцип работы с типами как с абстрактными типами данных. В программе можно не только применять те операции, которые были специально определены для строк, но также напрямую использовать представление строки. Так, например, для определения длины строки в языке имеется стандартная функция `length`, так что если значение строки `message` было задано оператором присваивания

```
message := 'hello, world';
```

то значение ее длины `lenMsg` может быть вычислено с помощью оператора присваивания

```
lenMsg := length(message);
```

Однако то же самое значение может быть получено и с помощью обращения непосредственно к представлению строки в виде массива символов, если только программист знает, что фактически длина строки хранится в нулевом элементе этого массива в виде символа с кодом, задающим эту длину:

```
lenMsg := ord(message[0]);
```

Здесь функция `ord` служит для формального преобразования элемента строки от символьного типа к типу целого значения.

Аналогично, можно использовать непосредственный доступ к символам, составляющим строку. Собственно говоря, для доступа к символам единственным способом является именно индексация массива. Так, например, для того чтобы заменить первый символ в строке `message`, можно указать следующий оператор присваивания:

```
message[1] := 'H';
```

Однако для добавления восклицательного знака в конец строки было бы ошибкой записать оператор

```
message[length(message) + 1] := '!';
```

поскольку в этом случае длина строки не увеличивается автоматически. Лучше всего здесь было бы воспользоваться какой-либо из стандартных операций над строками, например, функцией соединения строк

```
message := concat(message, '!');
```

хотя, конечно, можно использовать и внутреннее представление строки в виде массива символов. Только делать это надо чуть более аккуратно, явно увеличивая длину строки:

```
message[0] := succ(message[0]);
message[length(message)] := '!';
```

Здесь функция `succ` применяется для получения символа со "следующим" кодом, т. е. фактически для увеличения длины строки на единицу.

Другим стандартным способом представления строк является *представление с символом-терминатором*. В этом представлении длина строки не хранится, вместо этого специальный символ (обычно символ с нулевым кодом) используется в качестве конечного символа в строке. Данный способ, как и в представлении с хранимой длиной, также позволяет в массиве длины n хранить строки длиной от 0 до $(n-1)$ символа. Подобный способ хранения не позволяет быстро выдавать сведения о длине строки; операции с концом строки (например, добавление символа в конец строки) также выполняются достаточно долго (точнее, за время, пропорциональное длине строки). Однако такое представление имеет и ряд преимуществ, позволяя быстро "разрезать" строку на части вставкой внутрь строки нескольких дополнительных символов-терминаторов, рассматривать часть массива, содержащую последние символы строки, как самостоятельную строку и т. п. Представление строки с символом-терминатором характерно для языка Си и его потомков.

В этом случае также можно работать со строкой не только как с абстрактным типом данных, используя специально определенные операции над строкой, но и просто как с массивом символов. Например, если строка в языке Си задана с помощью следующего описания

```
char * message = "Hello, world!";
```

то убрать последний символ из строки можно с помощью записи символа-терминатора на место восклицательного знака:

```
message[strlen(message) - 1] = '\0';
```

В этом примере функция `strlen` использована для вычисления длины строки. Здесь, в отличие от представления с хранимой длиной, никакой дополнительной коррекции длины производить не надо.

Для большинства программ, работающих со строками небольшой длины, вполне достаточно одного из этих способов представления строк. Однако, если основная задача программы состоит в сложной обработке длинных строк (глобальный анализ текста программы компилятором, статистическая обработка текста на естественном языке и т. п.), стандартных средств опять окажется недостаточно.

Стандартные способы реализации строк дают определенные преимущества: легкость доступа к элементам строки (символам) по индексу; простота вы-

деления подстроки. В то же время отведение памяти одним сплошным сегментом, как это необходимо для массивов, налагает определенные ограничения на работу со строкой. Главным образом, трудности возникают, когда длина строки все время увеличивается, и это требует многократного отведения памяти под строку.

Одним из наиболее простых способов решить эту проблему является *списковое представление* строк. В списках память отводится независимо под отдельные элементы списка, так что добавление новых символов в конец или середину строки уже не вызывает заказа памяти под целую строку.

Очевидно, что наиболее гибким, но в то же время и наиболее расточительным способом представления сроки в виде списка является список отдельных символов строки. Конечно, это несколько странный способ сэкономить память — отводить под каждый символ вместо одного-двух байтов целый объект, имеющий в своем составе помимо символа, по крайней мере, один указатель. Более экономным способом отведения памяти будет сосредоточение в одном элементе списка нескольких символов. Так или иначе, списковое представление окажется эффективным способом представления в случае сильно изменяющихся длинных строк. К сожалению, в случае списковой реализации мы лишаемся одного из самых главных преимуществ массивов — простоты доступа к элементам.

На рис. 3.1 изображено представление строки в виде списка элементов, содержащих по 16 символов (не больше, но, может быть, меньше) каждый. Помимо собственно символов в элементе списка содержится указатель на следующий элемент и количество символов, хранящихся в элементе. Здесь приведено одно из возможных представлений строки "На берегу пустынных волн стоял он, дум великих полн".

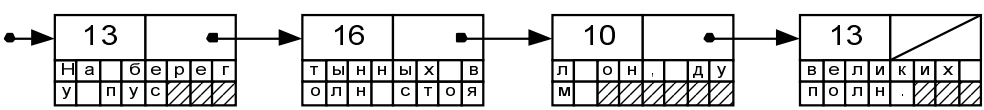


Рис. 3.1. Списковое представление строки

В листинге 3.1 приведена реализация строки в этом представлении. В состав реализации включены следующие операции:

- `charAt(index)` — извлечение символов по заданному индексу;
- `setCharAt(index, symbol)` — запись символа по заданному индексу;
- `substring(start, end)` — взятие подстроки;
- `length()` — длина строки;
- `append(symbol)` — добавление символа в конец строки;

- `append(string)` — добавление строки в конец исходной строки;
- `insert(index, string)` — вставка строки в середину исходной строки.

Конечно, на самом деле этого недостаточно. Необходимо, по крайней мере, определить операции для ввода и вывода строки в последовательный поток, расширить семантику представленных операций (например, добавить операции вставки символа, удаления подстроки и т. п.), более тщательно определить номенклатуру исключительных ситуаций и т. д. Однако, по-видимому, приведенного примера достаточно, чтобы пояснить основное содержание операций над строкой в описанном представлении, так что читатель может самостоятельно расширить или изменить определение класса, приспособив его для своих собственных потребностей.

Листинг 3.1. Реализация спискового представления строки

```
/*****************************************************************
 * Представление строки в виде списка элементов,
 * содержащих до ListString.ITEM_SIZE символов каждый
 */
public class ListString {
  public static final int ITEM_SIZE = 16; // максимальный размер элемента

  //-------------------------------------------------
  // Вспомогательные классы
  //-------------------------------------------------

  /*****************************************************************
   * Элемент списка, содержащий до ListString.ITEM_SIZE символов
   */
  private static class StringItem {
    char[] symbols = new char[ITEM_SIZE];  // массив символов
    StringItem next;          // указатель на следующий элемент
    byte size;                // количество символов

    //-------------------------------------------------
    // Конструкторы для создания элемента строки
    //-------------------------------------------------
    public StringItem(String src) { this(src, null); }

    /*****************************************************************
     * Этот конструктор позволяет построить сразу несколько
```

```
    * элементов списка в соответствии с длиной строки-аргумента
    */
   public StringItem(String src, StringItem nextItem) {
     if (src.length() > ITEM_SIZE) {
       // Рекурсивный вызов конструктора
       nextItem = new StringItem(src.substring(ITEM_SIZE), nextItem);
       src = src.substring(0, ITEM_SIZE);
     }
     size = (byte)src.length();          // Реальная длина строки
                                         // или ее начального сегмента
     src.getChars(0, size, symbols, 0);  // Сохранение символов
     next = nextItem;
   }
}

/*****************************************************************
 * Класс, представляющий позицию символа внутри строки
 * для извлечения и замещения отдельного символа
 */
private static class SymbolPosition {
  StringItem item;   // элемент строки
  byte pos;          // позиция внутри элемента строки

  // Простой конструктор
  public SymbolPosition(StringItem i, byte p) {
    item = i;
    pos = p;
  }
}

// Внутреннее представление строки

StringItem first = null;    // указатель на первый элемент

/*****************************************************************
 * Конструктор строки базируется на конструкторе первого элемента
 */
public ListString(String s) {
  if (s != null && s.length() > 0) {
```

```
      first = new StringItem(s);
    }
  }

  //-------------------------------------------------------------
  // Вспомогательные функции
  //-------------------------------------------------------------

  /**************************************************************
   * Поиск последнего элемента списка, представляющего строку
   */
  private StringItem getLast() {
    StringItem current = first;  // указатель на текущий элемент списка
    StringItem last = null;      // указатель на предыдущий элемент
    // Цикл по элементам списка
    while (current != null) {
      last = current;
      current = current.next;
    }
    return last;
  }

  /**************************************************************
   * Поиск позиции элемента с заданным индексом
   */
  private SymbolPosition findPos(int index) {
    if (index < 0) {
      throw new IndexOutOfBoundsException(
        "Индекс в строке " + index + " меньше нуля");
    }
    int curIndex = index;        // Количество символов,
                                 // которые надо еще отсчитать
    StringItem current = first;  // Указатель на текущий элемент
    // Цикл по элементам списка
    while (current != null && current.size <= curIndex) {
      curIndex -= current.size;
      current = current.next;
    }
```

```
    if (current == null) { // строка кончилась, а индекс еще не достигнут
      throw new IndexOutOfBoundsException(
        "Индекс в строке " + index + " больше максимального");
    }
    // Выдача результата
    return new SymbolPosition(current, (byte)curIndex);
  }

  //-------------------------------------------------------------
  // Доступные операции над строкой
  //-------------------------------------------------------------

  /*************************************************************
   * Преобразование к стандартной строке:
   * собирает строку из элементов
   */
  public String toString() {
    String res = "";  // переменная для сборки результирующей строки
    // Цикл по элементам строки
    for (StringItem current = first; current != null; ) {
      // Присоединение очередного элемента
      res += String.copyValueOf(current.symbols, 0, current.size);
      // Переход к следующему элементу
      current = current.next;
    }
    return res;
  }

  /*************************************************************
   * Вычисление длины строки
   */
  public int length() {
    int length = 0;  // переменная для вычисления длины
    // Цикл по элементам строки
    for (StringItem current = first; current != null; ) {
      length += current.size;
      current = current.next;
    }
```

```
    return length;
  }

  /*****************************************************************
   * Доступ к элементам: выборка символа по заданному индексу
   */
  public char charAt(int index) {
    // Вычисление позиции символа в строке с помощью
    // функции findPos, определенной выше
    SymbolPosition sp = findPos(index);
    // Выдача символа с использованием вычисленной позиции
    return sp.item.symbols[sp.pos];
  }

  /*****************************************************************
   * Доступ к элементам: замена символа с заданным индексом
   * новым значением
   */
  public void setCharAt(int index, char ch) {
    // Вычисление позиции символа в строке
    SymbolPosition sp = findPos(index);
    // Замена символа с использованием вычисленной позиции
    sp.item.symbols[sp.pos] = ch;
  }

  /*****************************************************************
   * Добавление символа в конец строки
   */
  public void append(char ch) {
    StringItem last = getLast();  // последний элемент списка
    if (last == null) {
      // Новый элемент становится первым и единственным в списке
      first = new StringItem(String.valueOf(ch));
    } else if (last.size < ITEM_SIZE) {
      // Символ просто добавляется в последний элемент списка
      last.symbols[last.size++] = ch;
    } else {
      // Новый элемент присоединяется к концу списка
```

```
      last.next = new StringItem(String.valueOf(ch));
    }
  }

  /*************************************************************
   * Добавление новой строки в конец данной
   */
  public void append(ListString string) {
    StringItem last = getLast();  // последний элемент списка
    if (last == null) {
      // Копируется указатель на первый элемент списка
      first = string.first;
    } else {
      // Новые элементы присоединяются в конец списка
      last.next = string.first;
    }
  }

  /*************************************************************
   * Вариант добавления новой строки в стандартном представлении
   * в конец данной строки
   */
  public void append(String string) {
    append(new ListString(string));
  }

  /*************************************************************
   * Вставка новой строки в середину данной
   */
  public void insert(int index, ListString string) {
    if (index < 0) {
      throw new IndexOutOfBoundsException(
        "Индекс в строке " + index + " меньше нуля");
    }
    // Определение позиции элемента с заданным индексом
    // примерно так же, как в операции findPos
    StringItem current = first;  // указатель на текущий элемент
    StringItem pred = null;      // указатель на предыдущий элемент
```

```
int curIndex = index;        // переменная для отсчета символов
while (current != null && current.size <= curIndex) {
  curIndex -= current.size;
  pred = current;
  current = current.next;
}
// Проверка правильности задания индекса
if (current == null && curIndex != 0) {
  throw new IndexOutOfBoundsException(
    "Индекс в строке " + index + " больше максимального");
}
if (string == null || string.first == null) {
  // Собственно, и вставлять-то было нечего!
  // Но правильность индекса надо было проверить все равно
  return;
}
// Последний элемент вставляемого фрагмента
StringItem lastInString = string.getLast();

if (curIndex > 0) {
  // Найденный элемент надо "разрезать" на две части.
  // Следующая переменная представляет вторую часть
  StringItem newItem = new StringItem(
    new String(current.symbols, curIndex, current.size - curIndex),
    current.next);
  // Теперь формируем длину первой части
  current.size = (byte)curIndex;
  // Вставляем новый участок списка между этими двумя частями
  current.next = newItem;
  pred = current;
  current = newItem;
}
// Окончательное формирование указателей в списке
if (pred == null) {
  first = string.first;
} else {
  pred.next = string.first;
}
```

```
    lastInString.next = current;
  }

  /*************************************************************
   * Вставка новой строки в стандартном представлении
   * в середину данной. Использует уже определенные операции
   */
  public void insert(int index, String string) {
    insert(index, new ListString(string));
  }
}
```

В приведенной реализации имеются два существенных недостатка.

Первый недостаток: в процессе обработки строк время от времени выполняется дробление элементов списка, представляющего строку. Например, такое дробление происходит во время выполнения операции `insert`, т. е. при вставке в строку другой строки. Вообще, если имеется потребность в сложной обработке длинных строк, то почти наверняка будут реализованы и другие операции, которые выделяют части исходной строки и осуществляют их перестановку, копирование, удаление и т. п. Во всех этих случаях неизбежно будет происходить дробление элементов списка. Наоборот, нигде в реализации нет слияния соседних элементов списка, даже если такое слияние можно осуществить, т. е. если суммарное количество символов, содержащихся в двух соседних элементах списка, не превосходит максимально возможного.

Такое дробление строки постепенно приводит к неоправданно большим расходам памяти. Чтобы избежать этого, нужно время от времени производить слияние соседних элементов списка там, где это возможно. Например, в реализации, приведенной в листинге 3.1, можно вставить операцию слияния соседних элементов списка в функции, осуществляющие просмотр списка. Некоторый эффект может быть достигнут, даже если такое слияние добавить в какую-нибудь одну достаточно часто работающую функцию, например в функцию `getLast()` поиска последнего элемента списка. Вот как будет выглядеть тогда эта функция:

```
/*************************************************************
 * Поиск последнего элемента списка, представляющего строку
 */
private StringItem getLast() {
  StringItem current = first;  // указатель на текущий элемент списка
  StringItem last = null;      // указатель на предыдущий элемент
  // Цикл по элементам списка
```

```
  while (current != null) {
    last = current;
    current = current.next;
    // Проверим, нельзя ли осуществить слияние элементов last и current
    while (current != null &&
           last.size + current.size <= ITEM_SIZE) {
      // Копирование символов
      for (int i = last.size; i < last.size + current.size; i++) {
        last.symbols[i] = current.symbols[i - last.size];
      }
      // Коррекция элемента last
      last.size += current.size;
      last.next = current.next;
      current = current.next;
    }
  }
  return last;
}
```

Второй недостаток реализации заключается в том, что при исполнении некоторых операций могут быть разрушены строки, передаваемые в эти операции в качестве аргументов. Так, например, при вставке в строку `str1` некоторой другой строки `str2` структура строки `str2` нарушается, так что использовать эту переменную во второй раз не удастся. Это достаточно неприятно, хотя в некоторых случаях и оправдано. Например, в реализации функции `append(ListString string)` использовать переменную аргумента повторно опасно, т. к. элементы списка, составляющего этот аргумент, после операции войдут в состав другой строки. Однако давайте рассмотрим вызов этой функции в реализации операции `append(String string)`:

```
public void append(String string) {
  append(new ListString(string));
}
```

Здесь аргумент вызова не является переменной, так что использовать его повторно никогда не удастся.

Единственный надежный способ борьбы с этим недостатком состоит в том, чтобы копировать аргументы, структура которых может быть нарушена. Это, однако, может привести к существенному снижению эффективности работы. Копирование строки может быть легко реализовано в виде отдельной операции, скажем, `clone()`. Реализация этой операции предоставляется читателям в качестве весьма полезного и несложного упражнения.

Еще один способ представления строк состоит в том, чтобы вообще не хранить символы, составляющие строку, в элементах списка. Часто встречаются ситуации, когда обработке подлежат строки, заданные с самого начала константами или введенные с внешнего устройства. Можно сказать, что в этом случае обработка строк состоит в перестановке и, может быть, копировании фрагментов, которые имелись в памяти с самого начала работы программы. Тогда можно поместить все символы всех обрабатываемых строк в некоторый буфер (пул памяти), а для представления строк использовать указатели в этот пул.

Пусть, например, в программе на языке Java имеется следующий фрагмент, формирующий строку из некоторой заданной:

```
String source = "Мы едем в далекие края";
StringBuffer dest = new StringBuffer(source);
dest.insert(7, ",").insert(8, source.substring(2, 7));
// Пока получилась строка "Мы едем, едем в далекие края"
System.out.println(dest.insert(7, dest.substring(7, 13)));
// Напечатается строка "Мы едем, едем, едем в далекие края"
```

Результирующая строка получилась из двух исходных строк (`source` и `","`) путем вырезок фрагментов и вставок. Если использовать представление строк в виде списка элементов, содержащих ссылки в буферный пул, то исходные строковые константы должны быть помещены в пул, а все операции, работающие с фрагментами строк, будут на самом деле производиться только с элементами соответствующих списков.

На рис. 3.2 изображено возможное содержимое буферного пула (на рис. 3.2, *а* числа над символами в пуле показывают их индексы в массиве) и представление строки `dest` в начале работы (рис. 3.2, *б*) и в конце работы после всех вставок (рис. 3.2, *в*). Чтобы не загромождать рисунок, указатели в буферный пул представлены в элементах списка парами чисел, первое из которых задает индекс символа в пуле, а второе — длину фрагмента строки. Собственно говоря, в программе эти указатели могут быть представлены точно так же. Это удобно, и, к тому же, позволяет полностью абстрагироваться от содержимого пула, обращаясь к нему только в том случае, когда явно требуются символы, например, при выводе строки в выходной поток. Для удобства около каждого элемента указано, какой фрагмент строки он представляет.

Разумеется, описанное представление, несмотря на некоторые преимущества, обладает также и рядом недостатков. Несомненным достоинством является тот факт, что отдельный элемент списка может описывать сколь угодно длинный участок строки. Но, как и в предыдущем представлении, при работе с фрагментами строк элементы списка будут описывать все более мелкие фрагменты, так что эффективность работы со строкой будет снижаться. Конечно, можно попробовать и здесь время от времени пытаться соединить

соседние элементы списка в один, однако в данном случае это не так просто, поскольку соединить удастся только такие фрагменты строк, которые физически расположены друг за другом в буферном пуле.

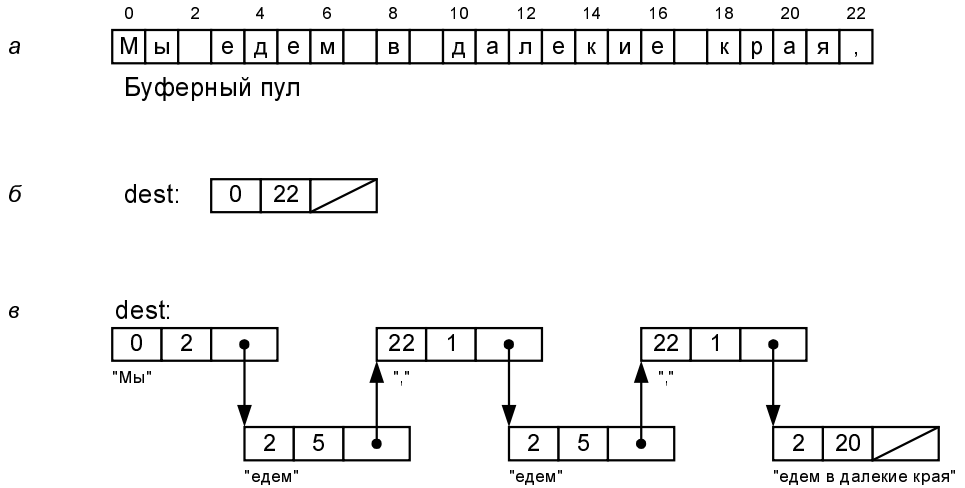


Рис. 3.2. Представление строк с помощью указателей в буферный пул

Несложно определить класс, реализующий строку в описанном представлении. Многие операции в нем будут похожи на соответствующие операции класса `ListString`, описанного в листинге 3.1. Различия в основном сводятся к тому, что вместо манипулирования с символами фрагментов строк работа производится только с их индексами в пуле. Конечно, требует отдельного представления и сам буферный пул.

В дальнейшем в книге представления строк, подобные только что описанным, нигде не используются. В тех примерах, где производится работа со строками, чаще всего используются стандартные строки языка Java, заданные классом `String`, либо массивы символов (**`char`**`[]`). Однако следует иметь в виду, что в случае длинных текстов такое представление может оказаться неэффективным, и тогда лучше выбрать одно из предложенных представлений или разработать свое собственное представление, наиболее подходящее для вашей задачи.

3.2. Хэширование и поиск в хэш-таблицах

В предыдущем разделе речь шла о представлении строк в случае, когда требуется работать с длинными строками и их фрагментами. Довольно часто встречаются также ситуации, когда обработке подлежит много маленьких

строк ("слов"), которые тоже надо сохранять в некоторой единой структуре ("словаре"). В дальнейшем для описания подобной ситуации мы будем использовать термины "слово" и "словарь" без кавычек. Слова не требуют для своего представления сложной структуры — вполне достаточно стандартных способов описания строк. Однако для словаря необходимо выбрать такое представление, которое бы обеспечило максимальную эффективность выполнения основной операции над словарем — поиск слова в словаре.

Самым простым представлениям для словаря будет, конечно, массив слов. Для обеспечения эффективности поиска можно обеспечить упорядоченность элементов этого массива, тогда поиск слов можно производить с помощью известного алгоритма двоичного поиска, который обеспечивает нахождение нужного слова среди множества из n слов за количество операций, пропорциональное $\log_2 n$. К сожалению, эффективность работы со словарем, организованным таким образом, резко снижается, когда требуется добавлять в него новые слова или удалять имеющиеся. Каждая вставка или удаление слова требуют, вообще говоря, сдвига в массиве в среднем около половины всех имеющихся в нем слов. Кроме того, при вставке слов возникает уже знакомая нам проблема выделения новой памяти, если первоначально отведенной под словарь памяти не хватит для размещения новых элементов.

Одним из способов решения этой проблемы является использование так называемой *функции расстановки* или *хэширования*. Слово "хэширование" происходит от английского "hash", означающего "дробить", "резать на кусочки", и отражает характер работы этой функции.

Основная идея применения функции расстановки состоит в следующем. Словарь будем по-прежнему представлять в виде массива слов, но помещать слова в него будем не в соответствии с алфавитным порядком, а в соответствии со значениями некоторой простой функции, вычисленной над словом. Такая функция — функция расстановки, — получая в качестве аргумента некоторое слово, выдает в результате некоторое целое число — индекс в словаре, под которым следует хранить это слово. Если каждому слову будет соответствовать свое значение функции расстановки, то поиск в словаре становится ненужным: вместо поиска осуществляется вычисление значения функции расстановки, после чего слово находится сразу же по вычисленному индексу.

К сожалению, на практике оказывается невозможным определить функцию расстановки так, чтобы каждому слову соответствовало свое уникальное значение индекса. Это приводит к тому, что два или более слов получают одно и то же значение функции расстановки. Говорят, что в этом случае происходит конфликт хэш-индексов. Один из самых простых способов разрешения этого конфликта состоит в том, чтобы помещать слово, вызвавшее конфликт, в один из соседних элементов массива. Тогда поиск и вставка

новых слов станут производиться хотя и не за один шаг, но все же достаточно быстро.

Естественным требованием для функции расстановки будет требование простоты ее вычисления. Но это далеко не единственное, что требуется от хорошей функции расстановки. Прежде всего надо обеспечить равномерное распределение вычисленных индексов слов в словаре. Например, для обработки слов русского языка функция, выдающая по заданному слову номер его первой буквы в алфавите, — это не очень удачный способ определения функции расстановки. Во-первых, слова русского языка далеко не равномерно распределяются по первым буквам. Некоторые буквы вообще не могут быть первыми в слове, другие, как, например, "С" наоборот, очень часто являются первыми буквами. Во-вторых, к этой неравномерности обычно добавляется еще и упорядоченность, присущая алгоритмам обработки текстов. Например, при записи некоторого словаря довольно естественной будет ситуация, при которой исходный поток слов уже как-то упорядочен. В такой ситуации лучше всего было бы, чтобы даже незначительно отличающиеся друг от друга слова имели бы существенно различные значения функции расстановки. Для этого по крайней мере нужно, чтобы значение функции расстановки зависело не от одного или нескольких символов слова, а от всех.

Пусть функция `code(c)` по заданной букве выдает ее номер в русском алфавите (если символ не является буквой русского языка, то будем считать, что функция `code` выдает значение 0). Вот как, например, может выглядеть эта функция на языке Java:

```
public static int code(char c) {
  return "абвгдеёжзийклмнопрстуфхцчшщъыьэюя".
                 indexOf(Character.toLowerCase(c)) + 1;
}
```

На самом деле, на практике обычно просто используют внутренний код каждого символа, так что фактически в вызове функции `code` нет необходимости.

Теперь можно применять эту функцию как базовую для определения функции расстановки. Прежде всего обеспечим зависимость значения функции расстановки от всех символов слова. Для этого сложим коды всех букв. Дополнительно, для того чтобы позиция буквы в слове также играла некоторую роль, будем добавлять к коду каждой буквы номер ее позиции. Далее решим, словарь какого размера нам потребуется. В конечном счете значение функции хэширования должно получиться достаточно равномерно распределенным на множестве всех обрабатываемых слов, и попадать в диапазон $[0, n-1]$, где n — размер словаря. Достаточно равномерное распределение получается, если вычислять индекс по формуле $(P \times X + Q)\ \%\ n$, где числа P и Q — это некоторые простые числа, по порядку близкие к n; X — вычисленная сумма кодов букв слова, а символ % задает операцию вычисления остат-

ка от целочисленного деления. Если, например, считать, что словаря в 1000 слов будет достаточно для наших целей, то можно выбрать для P и Q значения 557 и 811, тогда функция расстановки примет следующий вид:

```
public static int hash(String str) {
  int sum = 0;
  for (int i = 0; i < str.length(); i++) {
    sum += code(str.charAt(i)) + i;
  }
  return (557 * sum + 811) % 1000;
}
```

Теперь можно определить словарь (листинг 3.2), в котором будет храниться некоторое множество слов. Мы определим функцию добавления слова в словарь и функцию определения, содержит ли словарь заданное слово. В обеих функциях слово (или место для этого слова) ищется схожим образом: по слову вычисляется значение функции расстановки, затем происходит перебор позиций слов в массиве, предназначенном для хранения слов, до тех пор, пока либо искомое слово не будет найдено, либо не встретится пустое значение.

Если слова не удаляются из словаря, а только добавляются в него, то такой алгоритм поиска будет корректным, хотя при большой загруженности словаря поиск может осуществляться достаточно долго. В предельном случае, когда все позиции в словаре заняты, при поиске слова может оказаться необходимым перебрать все слова.

Листинг 3.2. Определение простого словаря

```
/*******************************************************
 * Класс определяет простой словарь, использующий функцию
 * расстановки для поиска слов
 */
public class HashDictionary {
  String[] dict = new String[1000];  // хранилище на 1000 слов

  /*****************************************************
   * Функция определяет код буквы как ее порядковый
   * номер в алфавите русского языка
   */
  private static int code(char c) {
    return "абвгдеёжзийклмнопрстуфхцчшщъыьэюя"
              .indexOf(Character.toLowerCase(c)) + 1;
  }
```

```
/*******************************************************
 * Функция расстановки, основанная на сложении кодов
 * букв со смещением, соответствующим их позиции
 */
public static int hash(String str) {
  int sum = 0;
  for (int i = 0; i < str.length(); i++) {
    sum += code(str.charAt(i));
  }
  return (557 * sum + 811) % 1000;
}

/*******************************************************
 * Внутренняя функция поиска позиции слова в словаре
 */
private int findPos(String word) {
  int counter = 0;      // счетчик перебранных слов в словаре
  int i = hash(word);   // текущий индекс слова в словаре
  for (; counter < dict.length; counter++) {
    if (dict[i] == null || dict[i].equals(word)) {
      return i;         // слово или его позиция для вставки найдены
    }
    if (++i == dict.length) i = 0;   // переход к следующему индексу
  }
  return -1;            // перебраны все позиции в словаре!
}

/*******************************************************
 * Функция добавления слова в словарь. Если слово уже есть
 * в словаре, то второй раз оно в словарь не попадает
 */
public void add(String word) {
  int i = findPos(word);    // позиция слова в словаре
  if (i == -1) return;      // только в случае переполнения словаря!
  dict[i] = word;           // добавление слова или его перезапись
}

/*******************************************************
 * Функция проверки наличия слова в словаре
 */
```

```
  public boolean hasWord(String word) {
    int i = findPos(word);
    return i != -1 && dict[i] != null;
  }
}
```

Приведем пример использования класса `HashDictionary` для решения следующей задачи. Пусть задано некоторое предложение на русском языке. Требуется проверить, есть ли в нем слова, встречающиеся по крайней мере дважды. Для решения этой задачи словарь, определенный классом `HashDictionary`, будет достаточно удобным способом представления множества слов в предложении. Действительно, мы можем исследовать заданное предложение слово за словом, добавляя каждый раз слово в словарь. Перед добавлением слова станем проверять, не встречалось ли уже это слово в предложении раньше.

Для деления предложения на слова воспользуемся стандартным классом языка Java `java.util.StringTokenizer`. Функция `twice`, представленная ниже, решает предложенную задачу и выдает слово, встретившееся в предложении второй раз. Если все слова в предложении различны, функция выдает в качестве результата пустую ссылку.

```
public static String twice(String text) {
  // В качестве разделителей слов могут использоваться пробелы,
  // знаки препинания, кавычки, скобки и другие "пустые" символы
  StringTokenizer tok =
                 new StringTokenizer(text, " ,.!?;:-()\'\"\n\t\f");
  HashDictionary dict = new HashDictionary();
  while (tok.hasMoreTokens()) {
    // Выберем очередное слово и преобразуем его к нижнему регистру букв
    String nextWord = tok.nextToken().toLowerCase();
    if (dict.hasWord(nextWord)) {
      return nextWord;        // слово уже встречалось
    } else {
      dict.add(nextWord);     // добавляем новое слово
    }
  }
  return null;                // в предложении нет одинаковых слов
}
```

Например, следующий вызов этой функции

```
System.out.println(twice(
  "Действительно, мы можем исследовать заданное предложение " +
```

```
  "слово за словом, добавляя каждый раз слово в словарь."
));
```

приведет к тому, что будет напечатано слово "слово".

Вернемся еще раз к названиям используемого нами метода: функция расстановки или функция хэширования. Первое название отражает цель использования функции — вычислить некоторый индекс, с помощью которого можно "расставить" слова в словаре. Второе название отражает метод, с помощью которого этот индекс вычисляется, — разрезание слова на части и "перемешивание" получившихся кусочков. Дословно глагол hash можно перевести с английского языка как "крошить", а в качестве существительного это слово можно перевести как "путаница", "перефразирование" или даже "мясная окрошка".

Надо сказать, что выбранный нами в вышеизложенном примере метод разрешения конфликтов при использовании функции расстановки не очень удобен. Во-первых, как уже было упомянуто, при большой загруженности словаря вся выгода, полученная от быстрого вычисления значения хэш-функции, может свестись на нет, и придется все равно перебирать значительную часть слов в словаре как при линейном поиске. Во-вторых, если требуется не только добавлять, но и удалять слова из словаря, метод оказывается вовсе неприемлемым, поскольку "дырки" в словаре могут прервать поиск несмотря на то, что нужное слово все-таки в нем содержится. Ликвидация же "дырок" — это крайне трудоемкий процесс.

Чаще используется другой метод разрешения конфликтов. Кроме основного массива, содержащего слова, в словаре организуется "область переполнения", куда и помещаются все слова при возникновении конфликтов. Чаще всего область переполнения организуется в виде списков слов с одинаковым значением функции расстановки.

Еще одно важное замечание. Чаще всего требуется хранить слова не сами по себе, а связывать с этими словами некоторую информацию, например, количество этих слов в тексте, или другую информацию, зависящую от задачи. Так, например, если в компиляторе требуется организовать словарь из всех используемых в тексте программы идентификаторов, то с каждым идентификатором можно связывать информацию о его типе, локализации и т. д. Поэтому чаще всего словари (которые еще называют хэш-таблицами) содержат не просто слова, а пары, в которых с каждым словом связан некоторый дополнительный объект. В этом случае слово, по которому происходит поиск, называют ключом поиска, а соответствующий объект — значением, связанным с этим ключом. По смыслу операции поиска в хэш-таблице каждый ключ находится в таблице в единственном экземпляре, однако, разумеется, значения функции расстановки могут быть одинаковыми для различных ключей.

В листинге 3.3 приведена еще одна реализация словаря, в которой с каждым словом (ключом) связан некоторый объект, а конфликты разрешаются путем организации списков для каждого значения ключа. Сама функция расстановки тоже несколько модифицирована и сделана более универсальной — в качестве кодов букв выбираются внутренние коды символов.

Листинг 3.3. Улучшенная реализация словаря

```
/*****************************************************************
 * Улучшенный вариант словаря, основанного на вычислении
 * значений функции расстановки
 */
public class HashDictionary {
  /*************************************************************
   * Элемент списка, содержащий ключ и связанный с ним объект
   */
  private static class ListElem {
    public String key;       // ключ
    public Object obj;       // связанный объект
    public ListElem next;    // ссылка на следующий элемент

    // Простой конструктор для создания элементов списка
    public ListElem(String key, Object obj, ListElem next) {
      this.key = key;
      this.obj = obj;
      this.next = next;
    }
  }

  private static final int LENGTH = 1000;  // размер словаря
  private static final int P = 557;        // множитель для вычисления
  private static final int Q = 811;        // слагаемое для вычисления

  // Массив списков, содержащих слова словаря
  private ListElem[] dict = new ListElem[LENGTH];

  /*************************************************
   * Функция расстановки, основанная на сложении кодов символов
   */
```

```
  private static int hash(String str) {
    int sum = 0;
    for (int i = 0; i < str.length(); i++) {
      sum += str.charAt(i) + i;
    }
    return ((P * sum + Q) & Integer.MAX_VALUE) % LENGTH;
  }

  /*****************************************************
   * Функция добавления нового объекта по ключу. Если объект,
   * связанный с этим ключом, уже содержится в словаре,
   * то новый объект не добавляется
   */
  public void add(String key, Object obj) {
    if (key == null) {
      throw new NullPointerException("Пустой ключ недопустим");
    }

    int index = hash(key);             // значение hash-функции
    ListElem current = dict[index];    // текущий элемент списка

    // Поиск ключа в словаре:
    while (current != null && !key.equals(current.key)) {
      current = current.next;
    }
    if (current != null) return;       // ключ уже есть в словаре

    // Создаем новый элемент списка и добавляем в начало списка
    ListElem newElem = new ListElem(key, obj, dict[index]);
    dict[index] = newElem;
  }

  /*****************************************************
   * Функция поиска объекта по ключу. Если ключ не найден,
   * то функция возвращает пустую ссылку
   */
  public Object find(String key) {
    if (key == null) {
```

```
      throw new NullPointerException("Пустой ключ недопустим");
    }

    int index = hash(key);              // значение hash-функции
    ListElem current = dict[index];     // текущий элемент списка

    // Поиск ключа в словаре
    while (current != null && !key.equals(current.key)) {
      current = current.next;
    }
    if (current == null) return null;  // ключ не найден
    return current.obj;
  }

  /**************************************************
   * Функция удаления объекта по заданному ключу.
   * Результат функции — указатель на удаляемый объект.
   * Если ключ не найден, то функция возвращает пустую ссылку
   */
  public Object remove(String key) {
    if (key == null) {
      throw new NullPointerException("Пустой ключ недопустим");
    }

    int index = hash(key);              // значение hash-функции
    ListElem pred = null;               // предыдущий элемент списка
    ListElem current = dict[index];     // текущий элемент списка

    // Поиск ключа в словаре
    while (current != null && !key.equals(current.key)) {
      pred = current;
      current = current.next;
    }
    if (current == null) return null;  // ключ не найден
    // Исключение элемента из списка
    if (pred == null) {
      dict[index] = current.next;
    } else {
```

```
      pred.next = current.next;
    }
    // Возврат результата
    return current.obj;
  }
}
```

С помощью такого определения словаря можно решать самые разнообразные задачи по обработке текстов. Например, уже упомянутая задача компиляции может использовать данный словарь для хранения и поиска информации об идентификаторах, содержащихся в некоторой программе. Правда, для большинства задач необходимо по крайней мере еще несколько функций, которые выдавали бы информацию обо всем словаре в целом. Нетрудно определить следующие функции:

```
public int size();          // количество слов в словаре
public Iterator keys();     // итератор всех ключей словаря
public Iterator objects();  // итератор всех связанных объектов
```

Тем не менее, у приведенной реализации имеется еще ряд недостатков. Наиболее серьезный из них — размер словаря выбран произвольно и никак не зависит от количества хранящихся в словаре слов. При небольшом количестве слов это приводит к тому, что напрасно расходуется память. Наоборот, если количество хранимых слов велико, то списки слов с одним и тем же значением функции расстановки становятся длинными, а это приводит к сильному замедлению всех основных операций со словарем. Лучше всего в таких случаях связывать алгоритм работы функции расстановки и общую организацию словаря с количеством слов, хранящихся в словаре. Например, если общее количество слов более, чем в два раза превышает размер массива списков слов `dict`, то можно провести так называемое "перехэширование". При этом массив расширяется (например, в два раза), функция расстановки изменяется в соответствии с новым размером словаря, а все слова вместе со связанными с ними объектами перемещаются в новые списки в соответствии с новыми значениями функции расстановки. Конечно, перехэширование — это серьезная и длительная операция, но зато после ее выполнения все операции со словарем начинают работать быстрее.

В языке Java имеется стандартный класс `Hashtable`, определяющий достаточно сложно организованную таблицу, содержащую ключи и связанные с ними значения. Хранение основано на применении функции расстановки `hashCode()`, при этом в качестве ключей могут выступать любые объекты, лишь бы для них можно было бы вычислить значение хэш-кода. Соответствующие функции определены для всех стандартных объектов (например, для самого объекта `Hashtable` она определена как сумма хэш-кодов всех хранящихся в ней ключей). Более того, функция `hashCode()` включена в

интерфейс "прародителя" всех классов Java — класса `Object`. Конечно, если программист хочет использовать объекты некоторого нестандартного класса в качестве ключей в хэш-таблице, то хэш-код для таких объектов следует определить самому.

В классе `Hashtable` определено частично управляемое программой перехэширование. Параметры перехэширования могут быть заданы при создании объекта класса `Hashtable`. Вообще же, структура объектов этого класса очень похожа на структуру только что определенного класса `HashDictionary`, хотя класс `Hashtable`, конечно, более общий. Определять самому классы, подобные `HashDictionary`, чаще всего не требуется. Стандартный класс может не подойти, только если требуются некоторые специфические особенности. Например, в классе `Hashtable` невозможно связать с ключом значение пустого указателя **`null`**. Еще может использоваться свой алгоритм перехэширования, может не подойти стандартная функция вычисления хэш-кодов для объектов, выбранных в качестве ключа, и т. д.

3.3. Словари, представленные списками и деревьями

Таблица, формируемая с помощью функции расстановки, — это один из основных способов представления словарей, да и вообще любых других таблиц, в которых требуется поиск по ключу. Тем не менее, это, конечно, далеко не единственный способ организации информации. В программах в зависимости от сложности, частоты поиска, набора выполняемых операций используется множество различных способов организации данных, которые чаще всего сводятся к комбинации списков, массивов и деревьев. В данном разделе мы рассмотрим несколько способов организации словарей от самых простых — упорядоченный список ключей — до достаточно сложно организованного "бора" — дерева, в котором слова разделены на буквы, и поиск осуществляется в соответствии с внутренним составом слова.

Организация словаря в виде *упорядоченного списка слов* не представляет никакой сложности. Практически все операции были уже нами рассмотрены в *разд. 1.2*, включая вставку в упорядоченный список, поэтому мы просто приведем пример задачи, в которой упорядоченные списки будут использоваться для хранения информации о словах некоторого текста. Мы будем использовать упорядоченный список как абстрактный тип данных, а для этого определим несколько интерфейсов, в которых будут заданы все основные операции.

Прежде всего, чтобы элементы списка можно было упорядочивать, над его элементами должны быть определены основные операции сравнения. Если операция `equals` определена уже в классе `Object` и, следовательно, может быть использована для сравнения любых объектов, то для использования

линейного порядка (операции типа "больше" и "меньше") обычно служит стандартный интерфейс `Comparable`, который описан следующим образом:

```
public interface Comparable {
  int compareTo(Object object);
}
```

Операция `compareTo` предназначена для сравнения заданного объекта с некоторым другим (аргументом операции), при этом функция должна выдавать отрицательное целое, если заданный объект в некотором смысле "меньше" аргумента, нуль, если объекты равны, и положительное целое, если заданный объект "больше" аргумента. Если объекты не сравнимы, то операция `compareTo` возбуждает исключительную ситуацию `ClassCastException`.

Многие стандартные классы языка Java реализуют этот интерфейс. Другими словами, во-первых, для таких объектов, как объекты класса `String`, `Integer`, `Date`, `Character` определена операция `compareTo`, позволяющая сравнивать эти объекты с объектами того же класса (иногда и с объектами других "родственных" классов), а во-вторых, объекты этих классов можно всегда использовать в качестве аргументов функций, требующих аргумента класса `Comparable`.

В нашем случае, однако, элементами списка выступают не просто слова, которые мы будем представлять строками, а пары, состоящие из слова и связанного с ним объекта. Поэтому определим сначала простой класс, представляющий такую пару и содержащий помимо операции доступа к ключу и связанному с ним объекту еще и операции сравнения `compareTo` и `equals`. Этот класс приведен в листинге 3.4.

Листинг 3.4. Описание класса для пары "ключ/значение"

```
/**
 * Этот класс представляет пару из строкового ключа
 * и связанного с ним объекта
 */

public class Pair implements Comparable {
  String key;     // ключ
  Object obj;     // связанный объект

  // ---- Конструктор пары
  public Pair(String key, Object obj) {
    this.key = key;
    this.obj = obj;
  }
```

```
  // ---- Функции доступа к ключу и объекту
  public String getKey() { return key; }
  public Object getObject() { return obj; }
  public void setKey(String newKey) { key = newKey; }
  public void setObject(Object newObject) { obj = newObject; }

  // ---- Функции сравнения
  public int compareTo(Pair anotherPair) {
    // Сравнение пар сводится к сравнению ключей
    return key.compareTo(anotherPair.getKey());
  }

  public int compareTo(Object object) {
    // Если аргумент не приводим к классу Pair,
    // то возникнет исключительная ситуация ClassCastException
    return compareTo((Pair)object);
  }

  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    } else if (obj instanceof Pair) {
      return key.equals(((Pair)obj).getKey());
    } else {
      return false;
    }
  }
}
```

В приведенном классе определены две операции `compareTo`. Одна из них вычисляет результат сравнения двух пар с помощью вызова соответствующей операции для ключа. Другая пытается выполнить приведение объекта к классу `Pair` и, если это удается, то результат сравнения будет, конечно, тем же самым. Если же переданный аргумент не является объектом класса `Pair`, то операция возбуждает ситуацию `ClassCastException` как результат неудачной попытки приведения объекта к классу `Pair`.

Операция `equals` определена в этом классе таким образом, что сравнение пар производится только по ключу. Таким образом, в случае равенства ключей обе операции — как `compareTo`, так и `equals` — будут одинаковым образом определять равенство пар.

Теперь опишем словарь как упорядоченный список пар, элементами которого являются пары класса `Pair`, причем ключом в каждой паре служит слово из словаря. Для работы с таким списком нам понадобятся операции поиска и вставки в упорядоченный список нового объекта. Поскольку работа операций с упорядоченным списком отличается от работы с неупорядоченным списком, лучше всего будет ввести в рассмотрение новый абстрактный тип данных — упорядоченный список. Этот тип данных как обычно можно будет задать с помощью интерфейса, который будет похож на интерфейс `IList`, описанный в конце *разд. 1.2*, однако в данном случае все элементы списка должны удовлетворять интерфейсу `Comparable`. Кроме того, некоторые определенные в интерфейсе `IList` операции, такие как `addFirst`, `addLast`, `insertBefore` и другие для упорядоченных списков, не имеют смысла. Дополнительно определим операцию поиска `find`, которая будет производить поиск в списке первого элемента, равного заданному.

```
public interface ISortedList {
  // Число элементов списка
  int getCount();
  // Удалить все элементы, равные заданному
  boolean remove(Comparable item);
  // Вставить элемент в упорядоченный список
  void insert(Comparable item);
  // Поиск первого элемента, равного заданному
  Object find(Comparable item);
  // Внешний итератор списка
  Iterator iterator();
  // Внутренний итератор списка
  void iterator(Visitor visitor);
}
```

Мы не станем приводить здесь никакой реализации этого интерфейса, но будем считать, что имеется "фабрика", с помощью которой можно получить экземпляр упорядоченного списка, удовлетворяющий этому интерфейсу. Описанный ниже словарь основан на упорядоченном списке, который создается в конструкторе этого словаря с помощью такой фабрики следующим образом:

```
SortedListFactory.createSortedList();
```

Описание словаря приведено в листинге 3.5 в виде описания класса `ListDictionary`.

Листинг 3.5. Словарь, базирующийся на упорядоченном списке

```
/*************************************************************
 * Класс описывает реализацию словаря, основанную на упорядоченном
 * списке пар из слова и связанного с ним значения
 */
public class ListDictionary {
  // Список list содержит упорядоченный список пар
  ISortedList list = SortedListFactory.createSortedList();

  // Функция добавляет в упорядоченный список новую пару
  public void add(String key, Object obj) {
    list.insert(new Pair(key, obj));
  }

  // Функция осуществляет поиск пары по ключу и удаляет
  // найденную пару, если она в списке была
  public void remove(String key) {
    list.remove(new Pair(key, null));
  }

  // Функция поиска связанного объекта по ключу
  public Object find(String key) {
    Pair found = (Pair)list.find(new Pair(key, null));
    return (found == null ? null : found.getObject());
  }

  // Функция выдает внешний итератор пар, используя внешний
  // итератор упорядоченного списка
  public Iterator iterator() {
    return list.iterator();
  }
}
```

Обратите внимание на то, как запрограммирована функция поиска по ключу. Для поиска формируется новая пара с помощью выражения `new Pair(key, null)` с пустым значением связанного с ключом объекта. Если пара с заданным ключом будет найдена в списке, то в результате поиска будет выдана не эта "новая" пара, а пара, содержащаяся в списке в качестве

элемента, что позволит при необходимости модифицировать найденный объект.

Теперь можно перейти к самой задаче. Пусть задан текст, содержащий слова, разделенные пробелами и знаками препинания. Требуется найти слово, встречающееся в тексте максимальное число раз.

Для решения этой задачи организуем словарь, в котором с каждым словом свяжем целое число — количество обнаруженных вхождений этого слова в текст. Такое число неудобно представлять в виде объекта стандартного класса `Integer`, поскольку этот класс предназначен для представления только констант, а нам необходимо увеличивать значение счетчика слова, как только обнаруживается новое вхождение уже встречавшегося ранее слова. Решение запишем в виде функции, получающей текст в качестве аргумента и выдающей пару из самого часто встречающегося слова и количества вхождений его в текст. Эта функция вместе с описанием класса для представления счетчика слов (`Counter`) приведена в листинге 3.6.

Листинг 3.6. Поиск самого часто встречающегося слова в тексте

```
/*************************************************************
 * Класс, представляющий изменяемое целое значение
 */
class Counter implements Comparable {
  int counter;           // поле для хранения целого значения

  // Конструкторы
  public Counter() { this(1); }
  public Counter(int n) { counter = n; }

  // Функции доступа
  public int getCounter() { return counter; }
  public void setCounter(int n) { counter = n; }

  // Увеличение значения на единицу
  public void inc() { counter++; }

  // Преобразование к строке
  public String toString() { return Integer.toString(counter); }

  // Функции сравнения
  public int compareTo(Counter anotherCounter) {
```

```
    return counter - anotherCounter.counter;
  }

  public int compareTo(Object obj) { return compareTo((Counter)obj); }

  public boolean equals(Object obj) {
    if (obj == this) {
      return true;
    } else if (obj instanceof Counter) {
      return counter == ((Counter)obj).counter;
    } else {
      return false;
    }
  }
}

/*********************************************************
 * Функция нахождения в тексте слова, встречающегося в нем
 * наибольшее количество раз. Результат выдается в виде пары,
 * состоящей из самого слова и счетчика числа его вхождений
 */
public static Pair FrequentWord(String text) {
  // tokenizer служит для выделения слов из текста.
  // Разделителями являются пробелы, знаки препинания, кавычки,
  // символы табуляции и перевода строк
  StringTokenizer tokenizer =
    new StringTokenizer(text, " ,./?;:!-\'\"\t\n\f");
  ListDictionary dict = new ListDictionary();   // словарь

  // 1. Занесение слов в словарь
  while (tokenizer.hasMoreTokens()) {
    // Выбор очередного слова
    String token = tokenizer.nextToken().toLowerCase();
    // Ищем в словаре это слово и запоминаем связанный с ним объект
    Counter wordCounter = (Counter)dict.find(token);
    if (wordCounter == null) {
      // Слова еще не было; заносим его в словарь со счетчиком = 1
```

```
      dict.add(token, new Counter());
    } else {
      // Слово уже было; увеличиваем значение счетчика на 1
      wordCounter.inc();
    }
  }

  // 2. Выбор слова с максимальной частотой
  Pair maxPair = new Pair("", new Counter(0));
  for (Iterator i = dict.iterator();  i.hasNext();) {
    Pair nextPair = (Pair)i.next();
    // Сравниваем счетчики двух пар
    if (((Counter)nextPair.getObject())
          .compareTo((Counter)maxPair.getObject()) > 0) {
      maxPair = nextPair;
    }
  }

  // 3. Выдача результата
  return maxPair;
}
```

У приведенной реализации есть два очевидных недостатка. Во-первых, само представление словаря в виде линейного списка приводит к длительному поиску в случае большого количества слов. Использование для этих целей списка может быть оправдано, только если обрабатываемый текст не слишком длинный. В этом случае простота реализации окупит наш выбор. Во-вторых, в случае, когда слово встречается в списке впервые, поиск в списке фактически производится дважды: сначала список просматривается для того, чтобы выяснить, что нового слова нет в словаре, а затем список просматривается еще раз при вставке слова на свое место. Возможно, следовало бы описать функцию, которая совместила бы в себе свойства функции поиска и вставки — функцию условной вставки слова, которая в случае, если слово не найдено, добавляет его в словарь, а если слово уже встречалось — выдает связанный с этим словом объект. Несложно внести такое изменение, гораздо труднее решить, действительно ли функция условной вставки имеет самостоятельное значение и может быть использована вне рамок решаемой задачи.

Для случая более длинного текста можно предложить представление словаря в виде бинарного дерева поиска, в котором узлами дерева служат те же самые пары, состоящие из ключа и связанного с ним объекта, что и в случае

представления словаря в виде списка. Внешне описание словаря, основанного на двоичном дереве поиска, очень похоже на описание словаря, основанного на списке. Например, если за основу взять реализацию двоичного дерева, приведенную в *разд. 2.4* (см. листинг 2.19), то реализация словаря может выглядеть так, как в листинге 3.7. В этой реализации используются операции для поиска, вставки и итерации двоичного дерева поиска. Мы не будем выделять эти операции в виде интерфейса для работы с двоичным деревом, в частности, поскольку двоичное дерево не представляет собой абстрактный тип данных, это, скорее, просто способ организации упорядоченной информации. Вместо описания интерфейса напомним определение класса `SearchTree` в виде "скелета" класса с опущенными телами функций.

```
public class SearchTree {
  public Comparable search(Comparable key) { ... }
  public Iterator iteratorInfix() { ... }
  public void insertLeaf(Comparable item) { ... }
}
```

Тогда класс `TreeDictionary`, представляющий словарь, реализованный на базе бинарного дерева поиска, может быть описан следующим образом.

```
public class TreeDictionary {
  // Дерево tree содержит упорядоченный список пар
  SearchTree tree = new SearchTree();

  // Функция добавляет в дерево новую пару
  public void add(String key, Object obj) {
    tree.insertLeaf(new Pair(key, obj));
  }

  // Функция осуществляет поиск пары по ключу и удаляет
  // найденную пару, если она в дереве была
  public void remove(String key) {
    tree.remove(new Pair(key, null));
  }

  // Функция поиска связанного объекта по ключу
  public Object find(String key) {
    Pair found = (Pair)tree.search(new Pair(key, null));
    return (found == null ? null : found.getObject());
  }
```

```
  // Функция выдает внешний итератор пар,
  // используя внешний итератор дерева поиска
  public Iterator iterator() {
    return tree.iteratorInfix();
  }
}
```

Теперь та же самая функция поиска самого часто встречающегося слова в тексте может быть реализована с использованием словаря `TreeDictionary` вместо словаря `ListDictionary`. Эксперименты показывают, что уже для текста, содержащего более 40—50 слов, реализация в виде двоичного дерева поиска оказывается более выгодной, чем реализация в виде упорядоченного списка. Впрочем, конечно, реальные цифры будут зависеть от характеристик используемого компьютера, применяемого компилятора, виртуальной Java-машины и т. п.

В данном разделе мы предложим еще один способ организации словаря — "бор". На английском языке этот способ организации данных называется *trie* — это искусственно созданное слово, полученное вырезкой средней части из слова *retrieval* — выборка. Аналогично, слово "бор" получено в русском языке путем вырезки средней части из соответствующего русского термина. Эта организация особенно хороша, если многие слова имеют одинаковые начала (префиксы) и отличаются только своими окончаниями. Например, для слов русского языка такая организация может оказаться выгодной, если в процессе работы со словарем часто будут встречаться слова, отличающиеся только падежными окончаниями. В боре слова хранятся не целиком, а побуквенно, и за счет этого поиск слов в словаре может оказаться достаточно эффективным.

Пусть, например, в словаре надо хранить следующие 10 слов: "кон", "конь", "корт", "кот", "кран", "крем", "крен", "крест", "крона", "крот". С каждым из этих слов надо связать некоторый объект. Предлагается организовать хранение этих слов так, как показано на рис. 3.3.

На этом рисунке видно, что слова хранятся в виде списков, составленных из их букв, а объекты, связанные со словами, показаны в виде треугольников. В реализации "бор" может быть представлен в виде списка деревьев, узлы которого содержат отдельные буквы слов. Связанный со словом объект может быть представлен либо в виде отдельного узла в этом же дереве, либо в виде самостоятельного поля в каждом узле. Первый способ имеет то преимущество, что связанный объект может быть представлен любым указателем, в том числе и пустым, например, если с каким-либо словом не нужно связывать никакой информации. Второй способ удобнее для реализации, однако он использует больший объем памяти, и, кроме того, пустые указатели будут иметь специальную трактовку. Если пустой указатель содержится

в некотором узле, то это будет означать, что слово не может заканчиваться в данном узле.

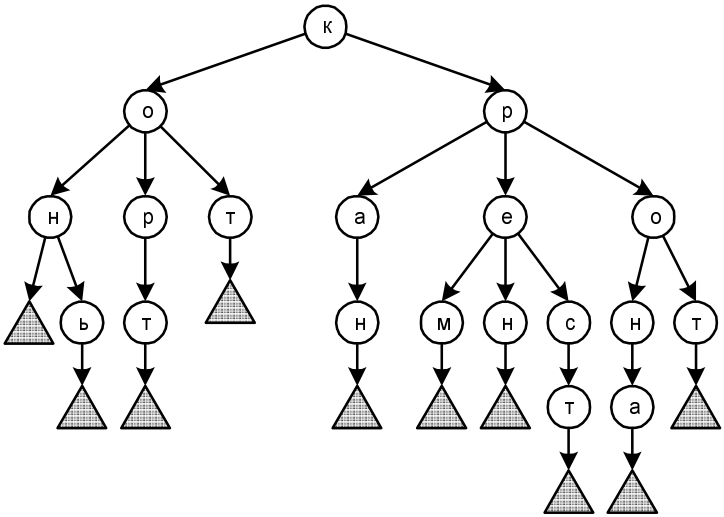


Рис. 3.3. Организация словаря в виде бора

Приведем фрагмент реализации словаря в виде бора, причем выберем первый способ представления связанных объектов. Деревья будем представлять в виде двоичных деревьев так, как описано в *разд. 1.3*, так что бор, представленный на рис. 3.3, будет иметь физическое представление, показанное на рис. 3.4, где каждый узел дерева представлен объектом, содержащим следующие поля:

- `info` — поле, содержащее очередную букву слова либо указатель на связанный со словом объект;
- `terminal` — признак связанного объекта, содержащегося в поле `info` (на рисунке не показан);
- `son` — указатель на список поддеревьев следующего уровня;
- `brother` — указатель на следующий узел в списке узлов одного уровня.

Пустые указатели на рис. 3.4 обозначены перечеркнутыми прямоугольниками, а связанные со словами объекты — заштрихованными прямоугольниками.

В приведенной в листинге 3.7 реализации класса показано только представление бора и реализация одного из методов — поиска связанного объекта по ключу. Реализация остальных методов не слишком сложна, однако требует аккуратности и хорошей техники программирования обработки сложных структур данных.

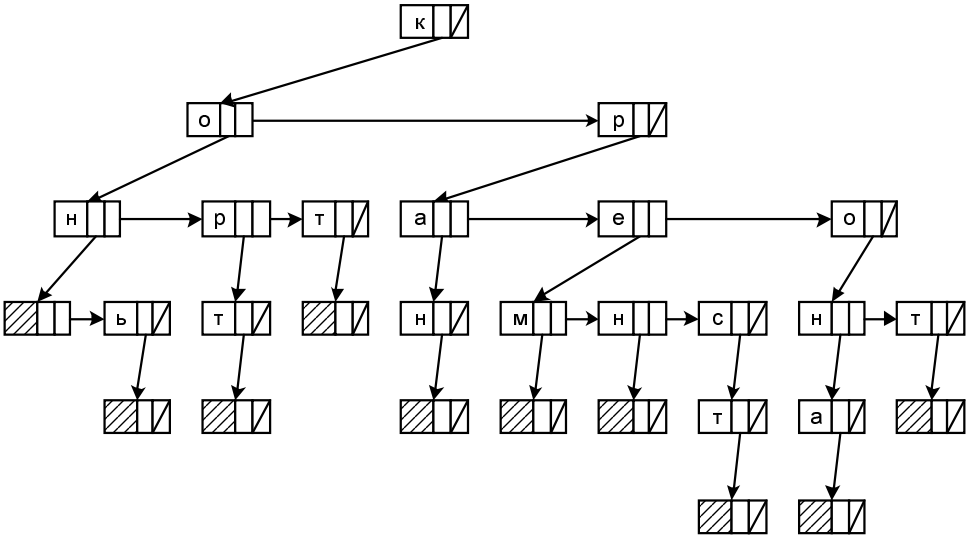


Рис. 3.4. Физическое представление бора

Листинг 3.7. Реализация поиска в боре

```
/*************************************************************
* Класс представляет собой реализацию бора
*/
public class Trie {
  /*************************************************************
  * Узел TrieNode содержит объект, представляющий букву в слове
  * или связанный со словом объект
  */
  private static class TrieNode {
    Object info;       // очередная буква или связанный объект
    boolean terminal;  // признак связанного объекта
    TrieNode son;      // указатель на следующий уровень
    TrieNode brother;  // указатель на следующий узел того же уровня

    // Конструктор узла
    public TrieNode(Object info,
                    boolean terminal,
                    TrieNode son,
                    TrieNode brother) {
```

```
      this.info = info;
      this.terminal = terminal;
      this.son = son;
      this.brother = brother;
    }
  }

  TrieNode root = null;          // корень бора

  // Функция поиска связанного объекта по ключу
  public Object find(String key) {
    TrieNode curNode = root;     // текущий исследуемый узел бора

    // Цикл поиска по очередной букве
    for (int i = 0; i < key.length(); i++) {
      // Очередная буква из ключа
      Character nextChar = new Character(key.charAt(i));
      // Ищем эту букву в текущем уровне бора
      while (curNode != null &&
             (curNode.terminal || !nextChar.equals(curNode.info))) {
        curNode = curNode.brother;
      }
      if (curNode == null) return null;  // буква не найдена
      curNode = curNode.son;             // переход на следующий уровень
    }

    // Все буквы ключа найдены. Теперь надо найти связанный
    // со словом объект, если он существует
    while (curNode != null && !curNode.terminal) {
      curNode = curNode.brother;
    }

    // Выдача результата
    return (curNode == null ? null : curNode.info);
  }
```

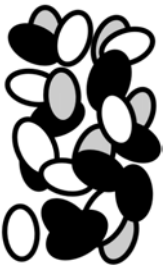
На тестах, содержащих большое количество сильно отличающихся друг от друга слов, реализация словаря в виде бора показывает не очень хорошую производительность. Этот способ представления словаря будет действительно

полезным, только если обрабатываются похожие слова, так что в общем случае следует выбирать более простые или более общие способы реализации.

На этом мы закончим описание способов представления словарей. Подведем некоторый итог, сравнивая между собой 4 способа представления словаря — упорядоченный список, бинарное дерево поиска, хэш-таблица и бор. Несмотря на некоторую условность приводимых оценок и отсутствие точных цифр, все же эти оценки могут оказаться полезными при выборе представления для словарных структур.

1. *Упорядоченный список* следует выбирать для представления словаря, если в словаре предполагается хранить небольшое число слов. С ростом словаря этот способ реализации становится все более невыгодным, и, в конце концов, оказывается наихудшим из всех, когда количество слов становится очень большим.
2. *Бинарное дерево* поиска является хорошей простой альтернативой упорядоченному списку для словаря среднего размера. К сожалению, бинарное дерево может иметь "плохую" структуру, а методы, позволяющие улучшать и поддерживать структуру дерева, уже далеко не так просты. Поэтому реализация может сильно потерять в простоте, не сильно выиграв в эффективности.
3. *Хэш-таблица* обычно является наилучшим способом реализации словаря, особенно в тех случаях, когда размер словаря можно легко оценить с самого начала. Хэш-таблицу не следует использовать только для небольших словарей, когда более простые методы оказываются и более эффективными, и, кроме того, в реализации, основанной на функциях расстановки, много времени может уйти на "перехэширование", поэтому для словарей, размер которых заранее оценить трудно, эффективность из-за этого может снизиться.
4. *Бор* стоит использовать для реализации словаря только в специфическом случае, когда слова, содержащиеся в словаре, очень похожи друг на друга. В этой ситуации общий объем данных уменьшается за счет "склеивания" одинаковых символов и, соответственно, эффективность работы может существенно улучшиться. Кроме того, организация данных в виде бора достаточно специфична — она существенным образом опирается на то, что слова действительно состоят из отдельных букв, в то время как все остальные способы организации словарей используют лишь свойство упорядоченности ключей. Таким образом, первые три способа организации словаря можно применять в случае любого упорядоченного множества ключей, а организация в виде бора предъявляет особые требования к структуре самих ключей.

Глава 4



Символьные преобразования

В этой главе представлено несколько классических алгоритмов символьных преобразований — таких как вычисление значения символьного выражения, упрощение формулы и др. Выражения представлены деревьями, узлами которых служат элементы этих выражений, так что обработка выражений обычно состоит в обходе всех узлов дерева и выполнении обработки содержащейся в узлах информации.

В *разд. 4.1* рассматриваются способы представления формул (символьных выражений) в виде деревьев. В *разд. 4.2* приводится алгоритм вычисления значения выражения по заданным значениям операндов. В *разд. 4.3* рассматривается несколько алгоритмов преобразования формул.

4.1. Представление выражений

Многие языки программирования используют понятие выражения. Хотя синтаксис и семантика выражений в различных языках программирования отличаются, тем не менее при анализе выражений можно выделить общие черты многих способов представления выражений. Обычно считается, что выражение состоит из элементарных (или первичных) операндов, соединенных в некоторую структуру с помощью пробелов, знаков операций и/или скобок. Часто требуется написать программу, которая, получив в качестве исходных данных текстовую строку, рассматривает ее как выражение, записанное по правилам некоторого языка. От программы может потребоваться выявить структуру выражения, преобразовать его в ту или иную форму записи, вычислить значение выражения, преобразовать по определенным правилам и т. п.

Рассмотрим различные способы представления формул и разберем некоторые из методов их обработки.

Будем считать, что в некотором языке формула состоит из:

- буквенно-цифровых идентификаторов (букв латинского алфавита и цифр, причем первым символом идентификатора должна быть буква);

- констант, представляющих неотрицательные целые числа в виде последовательности десятичных цифр;
- круглых скобок;
- знаков операций из допустимого набора операций.

Такую формулу можно рассматривать как строку символов (допустимыми символами являются буквы латинского алфавита, десятичные цифры, пробелы, скобки и знаки операций) и, соответственно, представлять строкой, т. е. в виде объекта типа `String`.

Чтобы работать со структурой такой формулы, надо, во-первых, выделить в этой строке элементарные единицы (то есть выполнить лексический анализ формулы), а во-вторых, разобрать формулу в соответствии с приоритетом используемых операций и расстановкой скобок (то есть выполнить синтаксический анализ). Обычно результатом лексического анализа будет последовательность лексических единиц (лексем) — атомов, из которых состоит выражение. Результатом синтаксического анализа является дерево, представляющее структуру выражения.

Пусть, например, имеется выражение, записанное в виде следующей строки:

```
C^2 == A^2 + B^2 – A*B*cos(alpha)
```

Последовательность символов, составляющих эту строку, может быть изображена так, как представлено на рис. 4.1.

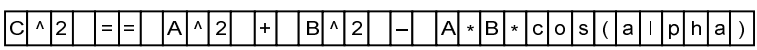


Рис. 4.1. Выражение, представленное строкой символов

После лексического анализа эта последовательность символов будет представлена следующей последовательностью лексем (рис. 4.2).

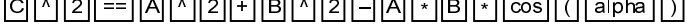


Рис. 4.2. Выражение, представленное последовательностью лексем

Дерево, выражающее структуру этого выражения, будет содержать в узлах операнды и операции, так что можно изобразить эту структуру в виде дерева (рис. 4.3).

В структуре синтаксического дерева выражения-узлы, представляющие знаки операций, имеют в качестве потомков операнды этих операций, которые, в свою очередь, могут быть как элементарными выражениями (константами и переменными), так и более сложными подвыражениями.

В следующих примерах алгоритмов и программ ограничимся не слишком сложными выражениями. Будем считать допустимыми операциями только арифметические операции над целыми числами: сложение, вычитание, умножение. Применение остальных операций, а также элементарных функций, будем считать недопустимым. Для определения порядка выполнения операций воспользуемся скобками, а при их отсутствии — обычными правилами приоритетов арифметических операций (умножение выполняется раньше сложения и вычитания; операции одного приоритета выполняются в порядке "слева направо").

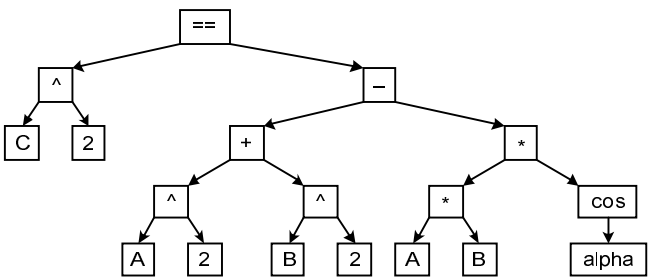


Рис. 4.3. Выражение, представленное синтаксическим деревом

Тогда узлами дерева могут быть объекты следующих типов:

- целое число, представленное объектом типа `Integer`;
- переменная, представленная строкой (объектом типа `String`);
- операция, обозначенная символом знака операции и двумя операндами, каждый из которых представлен деревом выражения.

В качестве выражения способен выступать любой из этих типов узлов, причем третий тип узла (операция) может быть описан с помощью следующего определения класса:

```
public class Operator {
  char operSign;              // знак операции
  ExpressionTree operand1;    // первый операнд
  ExpressionTree operand2;    // второй операнд
  // Конструктор:
  public Operator(char sign, ExpressionTree op1, ExpressionTree op2) {
    operSign = sign;
    operand1 = op1;
    operand2 = op2;
  }
```

```
  // Функции доступа
  public char getOperSign() { return operSign; }
  public ExpressionTree getOperand1() { return operand1; }
  public ExpressionTree getOperand2() { return operand2; }
}
```

Если условиться изображать дерево выражения так, что узлам разных типов будут соответствовать разные фигуры, то дерево выражения

```
(a + b) * (1 + x)
```

может быть изображено так, как показано на рис. 4.4. Здесь треугольниками изображены узлы-операторы, квадратами представлены узлы-переменные и, наконец, в кружках содержатся константы.

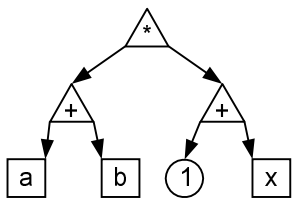


Рис. 4.4. Дерево простого выражения

Нашей первой задачей будет написание двух преобразователей: из символьной строки в синтаксическое дерево выражения и обратно из синтаксического дерева в строку. Для этого введем тип данных `ExpressionTree`, представляющий синтаксическое дерево выражения (выше этот тип уже был использован при описании класса `Operator`). Тогда функции преобразования могут быть выражены как методы класса `ExpressionTree`, например:

```
public class ExpressionTree {
  public String toString() { ... }
  public static ExpressionTree parse(String source) { ... }
}
```

Метод `toString()` может быть реализован сравнительно просто, если только мы не будем заботиться об оптимальной расстановке скобок. Основное правило преобразования выражения из дерева в символьную строку может быть выражено следующим образом:

- если в качестве дерева выступает константа или переменная, то его символьным представлением будет строковое представление этой константы или переменной;

- если в корне дерева находится знак операции, то строковые представления операндов соединяются знаком операции, и результат заключается в скобки.

Первое из этих правил поддерживается стандартными методами `toString` классов `Integer` и `String`. Второе правило можно реализовать с помощью определения функции `toString` для объектов класса `Operator`.

```
public class Operator {
  // Структура и конструктор класса показаны выше
  // в определении класса "Operator"

  // Функция преобразования в строку
  public String toString() {
    return "(" + operand1 + operSign + operand2 + ")";
  }
}
```

Здесь в определении функции `toString` неявно используется функция `toString` для класса `ExpressionTree`. Теперь, поскольку функция `toString` определена уже для всех типов выражений, в самом классе `ExpressionTree` определение метода `toString` оказывается очень простым:

```
public String toString() {
  return root.toString();
}
```

Для каждого из возможных типов корня этого дерева будет вызвана своя версия метода `toString`.

Что же касается обратного преобразования из строки в дерево выражения, то здесь ситуация намного сложнее. Тем не менее, мы можем воспользоваться уже знакомым нам алгоритмом из *разд. 2.2*, где похожая задача решалась для вычисления значения выражения, заданного строкой. Здесь мы можем применить тот же самый алгоритм, в котором задача решалась с использованием двух стеков — стека операндов и стека операций, только в роли значения выражения будет выступать теперь построенная часть синтаксического дерева выражения. В остальном алгоритм оказывается полностью пригодным для использования в этой задаче.

Так же, как и в *разд. 2.2*, необходимо сначала определить лексический анализатор, который, получив на входе исходную строку, будет на выходе генерировать лексемы — числа, переменные, скобки и знаки операций. Можно, как и раньше, использовать единственный класс `Lexical`, немного расширив его определение, чтобы включить в него новые типы лексем. Лучше, однако, по-

ступить по-другому: определим `Lexical` как абстрактный класс, а все конкретные типы лексем будем задавать в виде наследников этого абстрактного класса. В листинге 4.1 представлены описания соответствующих классов.

Листинг 4.1. Определение классов лексем

```
/**
 * Абстрактный класс Lexical представляет абстрактную лексему
 * и определяет возможные лексические классы для анализатора
 */
public abstract class Lexical {
  // Определяем возможные лексические классы
  public static final int NUMBER = 1;
  public static final int VARIABLE = 2;
  public static final int OPERATOR = 3;
  public static final int LEFTPAR = 4;
  public static final int RIGHTPAR = 5;

  public abstract int getLexClass();
}

/**
 * Класс LexNumber представляет лексему — целое число
 */
public class LexNumber extends Lexical {
  protected int number;     // представляемое число

  // Конструктор лексемы
  public LexNumber(int num) { number = num; }

  // Функция определения класса лексемы
  public int getLexClass() { return Lexical.NUMBER; }
  // Функция доступа
  public int getNumber() { return number; }
}

/**
 * Класс LexVariable представляет лексему-идентификатор
```

```
 */
public class LexVariable extends Lexical {
    protected String variable;   // представляемый идентификатор
    // Конструктор лексемы
    public LexVariable(String v) { variable = v; }

  // Функция определения класса лексемы
    public int getLexClass() { return Lexical.VARIABLE; }
  // Функция доступа
    public String getVariable() { return variable; }
}

/**
 * Класс LexOperator представляет лексему — знак операции или скобку
 */
public class LexOperator extends Lexical {
  protected char oper;   // представляемый знак операции или скобка

  // Конструктор лексемы
  public LexOperator(char oper) { this.oper = oper; }

  // Функция определения класса лексемы
  public int getLexClass() {
    return
      (oper == '(' ? Lexical.LEFTPAR :
       oper == ')' ? Lexical.RIGHTPAR :
       Lexical.OPERATOR);
  }

  // Функция вычисления приоритета операции
  public int getPrio() {
    return (oper == '-' || oper == '+' ? 1 :
            oper == '*' ? 2 : 0);
  }

  // Функция доступа
  public char getOperSign() { return oper; }
}
```

Теперь можно определить лексический анализатор подобно тому, как мы это делали в *разд. 2.2*. Анализатор берет на входе некоторую строку и преобразует ее к последовательности лексем. Удобно представлять анализатор в виде итератора, выдающего все лексемы из строки по очереди с помощью обычной для итератора операции `next()`. Дополнительно для анализа возможных ошибок определим метод, выдающий значение текущей позиции при анализе строки. Здесь мы не будем приводить никакой реализации лексического анализатора, однако ниже представлена схема такой реализации (тела функций не показаны).

```
public class LexAnalizer implements Iterator {
  // Конструктор
  public LexAnalizer(String expr) { ... }
  // Функции, реализующие интерфейс итератора
  public boolean hasNext() { ... }
  // Предполагается, что функция next() производит объекты класса Lexical
  public Object next() { ... }
  public void remove() { ... }
  // Текущая позиция при анализе строки:
  public int getCurrentPosition() { ... }
}
```

Теперь все готово для того, чтобы реализовать функцию преобразования выражения из строкового представления в представление в виде двоичного дерева. Алгоритм вычисления выражения из *разд. 2.2* модифицирован так, чтобы стек операндов теперь содержал в качестве результатов вычислений указатели на построенные части синтаксического дерева выражения, а стек операций будет содержать лексемы, представляющие знаки операций и левые скобки. Этот алгоритм представлен в листинге 4.2 в виде функции `parse` из определения класса `ExpressionTree`.

В алгоритме сделано еще одно важное дополнение: теперь он не только строит синтаксическое дерево для правильного выражения, но также готов в случае неправильного выражения возбудить исключительную ситуацию.

Анализ правильности выражения оказывается довольно простым. При анализе каждой лексемы известно, допустима ли эта лексема в выражении в текущей позиции. В программе определена переменная `waitFlag`, которая в любой момент времени содержит значение, указывающее на то, лексема какого класса ожидается на соответствующем месте. Возможны два варианта: "ожидается операнд" и "ожидается оператор". В первом случае допустимыми являются лексемы, представляющие число, переменную и левую (отрывающую) скобку. Во втором случае допустимыми являются знак операции и правая (закрывающая) скобка. Очевидно, что после очередного операнда или закрывающей

скобки должна следовать операция или закрывающая скобка, а после знака операции или открывающей скобки — операнд.

Еще один тип ошибок возможен из-за нарушения скобочной структуры выражения. В этом случае программа будет пытаться извлечь данные из пустого стека операндов или из пустого стека операций. Таким образом, ошибка этого типа распознается при возникновении ситуации исчерпания стека.

В программе имеются дополнительные пояснения, разъясняющие некоторые детали алгоритма.

Листинг 4.2. Класс `ExpressionTree` и функция анализа выражения

```
public class ExpressionTree {
  // Типы узлов дерева
  public static final int NODE_NUMBER = 1;
  public static final int NODE_VARIABLE = 2;
  public static final int NODE_OPERATOR = 3;

  // Поле root представляет узел дерева одного из трех возможных типов
  protected Object root;

  // Три конструктора для создания узлов всех возможных типов
  public ExpressionTree(int number) { root = new Integer(number); }
  public ExpressionTree(String variable) { root = variable; }
  public ExpressionTree(Operator oper) { root = oper; }

  // Функция преобразования дерева в строку
  public String toString() { return root.toString(); }

  // Функция вычисления типа узла
  public int getTreeClass() {
    return (root instanceof Integer    ? NODE_NUMBER :
            root instanceof String     ? NODE_VARIABLE :
            /*root instanceof Operator*/ NODE_OPERATOR);
  }

  // Функция "выполнения" операции на стеках
  public static void doOperator(
            IStack operands,     // стек операндов содержит деревья
            IStack operators     // стек операторов
```

```
        ) throws StackUnderflow {
  // Операнды для операции извлекаются из стека операндов
  ExpressionTree op2 = (ExpressionTree)operands.pop();
  ExpressionTree op1 = (ExpressionTree)operands.pop();
  // Знак операции берется из стека операций
  char opSign = ((LexOperator)operators.pop()).getOperSign();
  // В качестве результата "выполнения" операции строится новый узел,
  // содержащий знак операции и два поддерева
  operands.push(new ExpressionTree(new Operator(opSign, op1, op2)));
}

// Функция "выполнения" серии операций согласно их приоритетам
public static void doOperators(
          IStack operands,      // стек операндов
          IStack operators,     // стек операторов
          int minPrio           // граничный приоритет
        ) throws StackUnderflow {
  while (!operators.empty() &&
         ((LexOperator)operators.peek()).getPrio() >= minPrio) {
    doOperator(operands, operators);
  }
}

// Основная функция анализа строки и построения дерева выражения
public static ExpressionTree parse(String source)
                            throws ParseException {
  // Константы для значений "ожидаемого" класса лексемы
  final int WAIT_OPERAND = 1;
  final int WAIT_OPERATOR = 2;

  // Стеки операндов и операций создаются стандартным образом
  // с помощью "фабрики" по производству стеков
  IStack operands = StackFactory.createStack();
  IStack operators = StackFactory.createStack();
  // Лексический анализатор выражения
  LexAnalizer analizer = new LexAnalizer(source);
  // Флажок ожидаемой лексемы
  int waitFlag = WAIT_OPERAND;
```

```
try {
  // Цикл по лексемам исходной строки
  while (analizer.hasNext()) {
    Lexical lex = (Lexical)analizer.next();
    if (lex == null) {
      throw new ParseException("Неизвестная лексема",
                               analizer.getCurrentPosition());
    }
    if (waitFlag == WAIT_OPERAND) {
      switch (lex.getLexClass()) {
        case Lexical.NUMBER:
          // Число преобразуется в узел синтаксического дерева
          // и записывается в стек операндов
          operands.push(new ExpressionTree(
                                ((LexNumber)lex).getNumber()));
          waitFlag = WAIT_OPERATOR;
          break;
        case Lexical.VARIABLE:
          // Переменная преобразуется в узел синтаксического дерева
          // и записывается в стек операндов
          operands.push(new ExpressionTree(
                                ((LexVariable)lex).getVariable()));
          waitFlag = WAIT_OPERATOR;
          break;
        case Lexical.LEFTPAR:
          // Скобка просто записывается в стек операций
          operators.push(lex);
          break;
        default:
          // Лексема неожидаемого класса свидетельствует об ошибке
          throw new ParseException("Ожидался операнд",
                                   analizer.getCurrentPosition());
      }
    } else {  // waitFlag == WAIT_OPERATOR
      switch (lex.getLexClass()) {
        case Lexical.RIGHTPAR:
          // Появление закрывающей скобки вызывает "выполнение"
          // операций и удаление соответствующей открывающей скобки
          doOperators(operands, operators, 1);
```

```
                operators.pop();
                break;
              case Lexical.OPERATOR:
                // Выполняем операции с приоритетом не меньше текущего
                doOperators(operands, operators, ((LexOperator)lex).
                                                          getPrio());
                // Знак очередной операции записываем в стек операций
                operators.push(lex);
                waitFlag = WAIT_OPERAND;
                break;
              default:
                // Лексема неожидаемого класса свидетельствует об ошибке
                throw new ParseException("Ожидался знак операции",
                                         analizer.getCurrentPosition());
            }
          }
        }
        // Выполняем последние оставшиеся в стеке операции
        doOperators(operands, operators, 0);
        // Результат — последнее значение в стеке операндов
        return (ExpressionTree)operands.pop();
      } catch (StackUnderflow ex) {
        // Исчерпание стека свидетельствует об ошибке
        // в скобочной структуре выражения
        throw new ParseException("Ошибка в скобочной структуре выражения",
                                 analizer.getCurrentPosition());
      }
    }
}
```

Операции преобразования выражения из строки в синтаксическое дерево и обратно являются в некотором смысле обратными друг к другу. Тем не менее, это все-таки верно только с точностью до возможных вариантов расстановки скобок и наличия пробелов в строке. Так, например, в результате вычисления следующего оператора

```
ExpressionTree tree = ExpressionTree.parse("(a+b)*(c+d) + 1 + 2*a");
```

будет построено дерево, изображенное на рис. 4.5.

Если теперь преобразовать его обратно в строку с помощью выражения

```
tree.toString()
```

то в результате "обратного" преобразования получится строка

```
"(((a+b)*(c+d))+1)+(2*a))"
```

в которой по сравнению с исходной строкой отсутствуют пробелы между лексемами и добавлены лишние скобки.

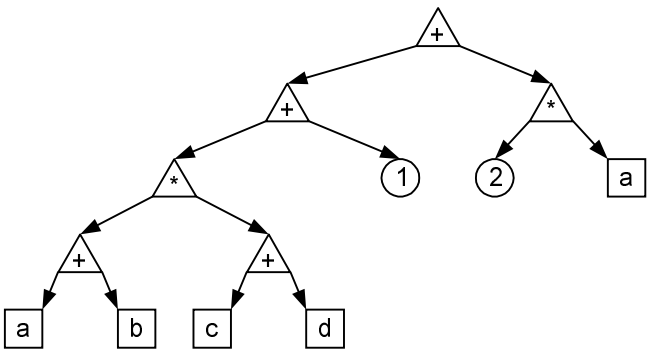


Рис. 4.5. Синтаксическое дерево выражения `(a+b)*(c+d)+1+2*a`

Что же можно делать с выражением, представленным в виде синтаксического дерева?

Спектр возможных операций достаточно широк: можно вычислить значение этого выражения, если заданы значения входящих в него переменных; можно осуществить подстановку некоторого подвыражения в исходное выражение вместо всех вхождений некоторой переменной; можно осуществить преобразование этого выражения в последовательность команд его вычисления в некоторой машине и т. д.

Некоторые из таких алгоритмов представлены дальше в этой книге. Конечно, многие из операций над выражением можно изобразить в виде обхода дерева некоторым итератором. Однако, в отличие от многих предыдущих случаев, когда итератор использовался только для получения последовательности элементов некоторой структуры в одном из возможных порядков, а сама структура при этом оставалась неизменной, при обработке выражений часто обработка узлов оказывается значительно более сложной. Здесь уже недостаточно только получать элементы в некотором порядке, итератору может потребоваться более глубокая обработка узлов. Поэтому технику работы с итераторами придется несколько обобщить и расширить.

Прежде всего, при обработке выражений практически не будут использоваться внешние итераторы. Поскольку алгоритм обработки узлов лежит за пределами внешнего итератора, то этому алгоритму становится недоступной структура выражения, которая часто самым существенным образом должна влиять на алгоритм обработки. Кроме того, недостатки, присущие внутренним итераторам, — невозможность прервать итерацию до достижения ее

конца, невозможность параллельной обработки нескольких структур и др. — для обработки выражений не очень существенны. Поэтому для обработки выражений мы будем использовать технологию, близкую к понятию внутреннего итератора — технологию "посетитель" (visitor).

В этой технологии для обработки сложной структуры предназначен специальный объект-посетитель, который отвечает за обработку определенных типов узлов в соответствии со своим внутренним алгоритмом. Сама сложная структура (например, синтаксическое дерево выражения) при этом лишь направляет посетителя к своим узлам в определенном порядке. Можно считать, что посетитель проталкивается по дереву, при этом в каждом узле выполняется одна из операций "посещения", соответствующая типу этого узла.

Данная технология вместе с некоторыми примерами хорошо описана в книге [2]. В нашей книге она будет использоваться следующим образом.

Опишем абстрактный класс `Visitor`, в котором определим методы для обработки различных типов узлов дерева выражения. Каждый метод получает в качестве аргумента объект соответствующего типа, который содержится в узле. Для обработки дерева выражения предназначен метод `accept(Visitor v)`, просто выбирающий нужный метод посетителя для посещения соответствующего узла. Посетитель, в свою очередь, может либо "протолкнуть" себя для обработки поддеревьев, либо создать для этого новый экземпляр посетителя.

Прежде, чем переходить к более сложным алгоритмам, рассмотрим два простых примера подобной обработки дерева выражения.

В первом примере определим множество различных переменных, существующих в выражении. Для представления этого множества используем объект класса `HashSet`. Обходя дерево, будем последовательно добавлять в это множество встречающиеся переменные. В конце работы посетитель узлов соберет в множество все встретившиеся переменные.

В этом примере один и тот же посетитель "проталкивается" по дереву, собирая всю нужную информацию в процессе обхода. Соответствующая программа представлена в листинге 4.3. На нем сначала описан общий интерфейс посетителей, содержащий методы обработки узлов дерева выражения для каждого из трех возможных типов узлов. Затем сделано дополнение к классу `ExpressionTree` — описан метод `accept` для "приема" посетителя. Наконец, определен посетитель `VariablesPicker` для сбора переменных. Обратите внимание, как посетитель обеспечивает проталкивание себя по дереву в реализации метода `visit(Operator operNode)`.

Листинг 4.3. Реализация посетителя для сбора информации о переменных выражения

```
/**
```

```
 * Интерфейс посетителя узлов дерева выражения
 */
public interface Visitor {
  public void visit(Integer intNode);
  public void visit(String varNode);
  public void visit(Operator operNode);
}

/** Дополнение к определению класса ExpressionTree
 *
 */
public class ExpressionTree {
  ...
  // Метод для обработки узла произвольным посетителем
  public void accept(Visitor v) {
    // Выбор метода производится согласно типу узла
    switch (getTreeClass()) {
      case NODE_NUMBER:
        v.visit((Integer)root);
        break;
      case NODE_VARIABLE:
        v.visit((String)root);
        break;
      case NODE_OPERATOR:
        v.visit((Operator)root);
        break;
    }
  }
}

// Теперь определяем класс посетителя для определения множества
// переменных выражения
public class VariablePicker implements Visitor {
  // Множество переменных реализовано в виде объекта класса HashSet
  HashSet variables = new HashSet();

  // Функция доступа
  public HashSet getVariables() { return variables; }
```

```
  // Реализация функций посетителя
  // Посещение узла — целого значения ничего не добавляет
  // к множеству переменных
  public void visit(Integer intNode) {}
  // Посещение переменной приводит к ее занесению в множество
  // Дубликаты не заносятся
  public void visit(String varNode) {
    variables.add(varNode);
  }

  // Посещение оператора сводится к посещению его операндов
  public void visit(Operator operNode) {
    // Посетитель передает себя последовательно сначала в первый
    // операнд, а потом во второй, накапливая информацию
    operNode.getOperand1().accept(this);
    operNode.getOperand2().accept(this);
  }
}
```

Если теперь взять некоторое дерево, полученное из строки, например, следующим образом:

```
ExpressionTree tree = ExpressionTree.parse("(a+b)*(c+d)+1+2*a");
```

то множество переменных этого выражения может быть напечатано с помощью следующей последовательности операторов:

```
VariablePicker picker = new VariablePicker();
tree.accept(picker);
System.out.println(picker.getVariables());
```

Результатом исполнения этих операторов будет вывод следующей строки (с точностью до порядка переменных и оформления множества):

```
[b, a, d, c]
```

Второй пример чуточку сложнее. Напишем функцию копирования дерева выражения, причем сделаем это также с помощью технологии "посетитель". Задачей посетителя при посещении им узла дерева выражения будет создать копию этого узла. Для узлов простых типов (целое число и переменная) такая копия может быть создана напрямую. Для создания копии узла-операции посетитель формирует еще два экземпляра посетителей и использует их для построения копий операндов, после чего уже легко сделать копию исходного узла.

В листинге 4.4 представлена реализация соответствующего посетителя в виде определения класса `Copier`.

Листинг 4.4. Посетитель для копирования дерева выражения

```
/**
 * Класс Copier реализует посетителя узлов дерева выражения,
 * причем результатом посещения является создание копии выражения
 */
public class Copier implements Visitor {
  // Локально хранящаяся копия выражения
  ExpressionTree copy = null;

  // Функция доступа
  public ExpressionTree getResult() { return copy; }

  // Методы посещения узлов
  // Копия узла целого числа создается непосредственно
  public void visit(Integer intNode) {
    copy = new ExpressionTree(intNode.intValue());
  }

  // Копия узла переменной создается также непосредственно
  public void visit(String varNode) {
    copy = new ExpressionTree(varNode);
  }

  // Копия узла оператора
  public void visit(Operator operNode) {
    // Создаются новые экземпляры посетителей
    Copier visitor1 = new Copier();
    Copier visitor2 = new Copier();
    // Посещение...
    operNode.getOperand1().accept(visitor1);
    operNode.getOperand2().accept(visitor2);
    // Новый узел создается из результатов посещения
    copy = new ExpressionTree(
               new Operator(operNode.getOperSign(),
                            visitor1.getResult(),
```

```
                                visitor2.getResult()));
    }
}
```

Если внимательно изучить использованное нами представление дерева и то, как это представление применяется в приведенных простых примерах, то можно увидеть, что фактически мы рассматриваем только "константные" выражения. Мы не предоставили никаких возможностей, с помощью которых можно было бы, скажем, заменить переменную на ее значение, сделать подстановку переменных и т. п. Такой подход полностью аналогичен подходу к определению строк в языке Java. Класс `String` также не содержит методов, позволяющих заменять части строк, вставлять или удалять символы и т. п. Таким образом, класс `String` в Java представляет "константные" строки.

Конечно, можно было бы определить и другие деревья выражений, например, добавив операцию присваивания нового значения имеющемуся узлу в набор операций. Тогда все операции замены частей выражения стали бы возможными. Ниже показано, как можно было бы представить такие операции присваивания в виде методов класса `ExpressionTree`.

```
public class ExpressionTree {
  protected Object root;
  ...
  public void set(int number) { root = new Integer(number); }
  public void set(String variable) { root = variable; }
  public void set(Operator oper) { root = oper; }
}
```

Интересно однако, что добавление таких операций все равно не позволяет посетителям узлов менять состав и структуру выражений. Дело в том, что посетитель получает в качестве аргумента не сам узел (класса `ExpressionTree`), а его содержимое, таким образом, у него все равно нет возможности изменить содержимое узла. Конечно, все-таки можно добиться "изменяемости" деревьев выражений с помощью еще некоторых усилий, однако мы не станем этого делать. Константные деревья имеют ряд преимуществ перед изменяемыми структурами (так же, как и константные строки класса `String` перед изменяемыми строками класса `StringBuffer`).

Свойство "константности" выражений иногда может оказаться очень удобным: если где-нибудь имеется указатель на выражение или его часть, то это выражение останется неизменным независимо от того, какие операции выполнялись над ним. Собственно, это свойство деревьев выражений и называется *"константностью"*. В дальнейшем приведенные выше операции `set` нигде использоваться не будут.

Тем не менее, мы ничего не теряем по существу. Во всех алгоритмах, приводимых далее в этом разделе, при преобразованиях выражения вместо изменения исходной структуры дерева всегда строится новое (вообще говоря, измененное) выражение.

Аппарат посетителей дает очень гибкие возможности для анализа и построения выражения. Тем не менее, эти возможности все же следует дополнить некоторыми операциями, обычными для всех объектов, независимо от того, являются они константными или нет. В интерфейсе класса `Object` — родоначальника всех классов Java — определены следующие методы, которые было бы полезно переопределить для деревьев выражений: `clone()`, `equals(Object o)`, `hashCode()`.

Что касается последнего метода (`hashCode`), то его реализация предоставляется на усмотрение читателей. По-видимому, она будет отражать вкусы авторов в области изобретения и комбинирования функций расстановки. А вот первые два метода мы рассмотрим более подробно.

По существу, метод `clone`, предназначенный для создания копии объекта, уже нами реализован с помощью посетителя `Copier`. Таким образом, можно было бы написать следующую простую реализацию этого метода.

```
public Object clone() {
  Copier copier = new Copier();
  accept(copier);
  return copier.getResult();
}
```

Тем не менее, имеет смысл написать отдельную реализацию этого метода, независимую от наличия такого достаточно специфического класса, как `Copier`. Кроме того, что такая реализация не требует определения специального класса, она еще и проще и эффективнее. Удобнее всего реализовать метод с помощью рекурсивной функции.

Функцию сравнения выражений на совпадение (`equals`) тоже можно реализовать похожим образом. Точно так же можно было бы определить посетителя, проверяющего равенство узлов, однако здесь мы ограничимся более простым подходом.

В листинге 4.5 представлены оба метода — `clone` и `equals`, — реализованных в виде простых рекурсивных функций.

Листинг 4.5. Реализация стандартных операций класса `Object` для дерева выражения

```
// Считается, что все приведенные ниже операции определены
// в контексте класса ExpressionTree
```

```
/**
 * Реализация операции копирования
 */
public Object clone() {
  // Алгоритм копирования выбирается в зависимости от типа корневого узла
  switch (getTreeClass()) {
    case NODE_NUMBER:      // узел типа "целое число"
      return new ExpressionTree(((Integer)root).intValue());
    case NODE_VARIABLE:    // узел типа "переменная"
      return new ExpressionTree((String)root);
    case NODE_OPERATOR:    // узел типа "оператор"
      ExpressionTree op1 =       // копия первого операнда
          (ExpressionTree)((Operator)root).getOperand1().clone();
      ExpressionTree op2 =       // копия второго операнда
          (ExpressionTree)((Operator)root).getOperand2().clone();
      return new ExpressionTree(
                 new Operator(((Operator)root).getOperSign(), op1, op2));
    default:               // никаких других типов узлов не предусмотрено
      return null;
  }
}

/**
 * Реализация операции сравнения на равенство
 */
public boolean equals(Object obj) {
  // Если аргумент не является объектом класса ExpressionTree,
  // то будет возбуждено прерывание ClassCastException
  return equals((ExpressionTree)obj);
}

public boolean equals(ExpressionTree tree) {
  if (tree == null) return false;
  int nodeClass = tree.getTreeClass();  // тип корневого узла аргумента
  if (nodeClass != getTreeClass()) return false;
  // Алгоритм сравнения выбирается в зависимости от типа корневого узла
  switch (nodeClass) {
    case NODE_NUMBER:      // сравниваем целые значения
```

```
      return ((Integer)root).equals(tree.root);
    case NODE_VARIABLE:    // сравниваем переменные как строки
      return ((String)root).equals((String)tree.root);
    case NODE_OPERATOR:    // сравниваем знаки операций и операнды
      return ((Operator)root).getOperSign() ==
                              ((Operator)tree.root).getOperSign() &&
             ((Operator)root).getOperand1().equals(
                              ((Operator)tree.root).getOperand1()) &&
             ((Operator)root).getOperand2().equals(
                              ((Operator)tree.root).getOperand2());
    default:               // Никаких других типов узлов не предусмотрено
      return false;
  }
}
```

В данной реализации сравнение выражений производится "формально", так что с точки зрения метода `equals` выражения `(a + b) + c` и `a + (b + c)` будут, разумеется, различными. Более "интеллектуальная" операция сравнения могла бы учитывать коммутативность операций сложения и умножения, производить константные вычисления и т. п. Конечно, мы не в состоянии написать алгоритм, который определял бы эквивалентность произвольных выражений в смысле преобразуемости их друг к другу.

Насколько общим является наш подход к определению дерева выражения и операций над ним? Что изменится, если мы попробуем перейти от целочисленных выражений, скажем, к выражениям над логическими значениями (пропозициональным формулам)? Насколько можно расширить номенклатуру операций? Что, если, наряду с бинарными операциями, понадобится включать в состав выражений унарные операции или вызовы элементарных функций?

Для ответов на эти вопросы попробуем обобщить наше дерево так, чтобы оно допускало произвольные типы операндов и различные операции.

Прежде всего, обобщим понятие узла типа "целое число" до понятия узла, содержащего некоторую произвольную константу. Поскольку ни тип этой константы, ни набор возможных операций заранее неизвестен, то следует считать, что само значение представлено объектом класса `Object`. Тем не менее, для того, чтобы отличать этот тип узла от остальных, потребуем, чтобы все наши константы принадлежали абстрактному типу `Constant`. Этого можно достичь, если ввести интерфейс `Constant`, который не будет содержать никаких операций. Единственная цель этого интерфейса состоит в том, чтобы обеспечить отличие константных значений от других типов узлов дерева.

```
public interface Constant { }
```

Для представления целых и вещественных чисел, а также логических, символьных и других стандартных значений нам не потребуется описывать реализацию новых классов, во всем подобных имеющимся стандартным классам `Integer`, `Boolean` и др. Правда, к сожалению, не удается определить новый класс как расширение существующего стандартного класса и просто указать, что этот новый класс удовлетворяет интерфейсу `Constant`. Вот как, например, могло бы выглядеть описание класса для представления целых констант, хранящихся в дереве:

```
public class TreeInteger extends Integer implements Constant { }
```

К сожалению, стандартный класс `Integer` (так же, как и другие "константные" стандартные классы) по вполне очевидным причинам объявлен как **final**, и поэтому невозможно определить его расширение. Вместо этого придется определить новый класс, имеющий объект класса `Integer` в своем составе в виде поля. Однако, разумеется, никаких новых операций над "расширенным" целым вводить не станем; все операции над объектами этого класса будут просто обращаться к соответствующим методам класса `Integer`. Такой класс представлен в листинге 4.6. Из операций над целыми в нем приведены только требующиеся нам в дальнейшем методы `toString()`, `equals()` и `intValue()`, а также, разумеется, конструкторы. Однако нетрудно расширить определение этого класса очевидным образом так, чтобы включить в определение любые другие операции над целыми.

Листинг 4.6. Класс, представляющий целые числа в виде константы дерева выражения

```
public class TreeInteger implements Constant {
  Integer value;

  // Конструкторы целых констант
  public TreeInteger(Integer i) { value = i; }
  public TreeInteger(String s) { value = new Integer(s); }
  public TreeInteger(int i) { value = new Integer(i); }

  public String toString() { return value.toString(); }
  public boolean equals(Object o) { return equals((TreeInteger)o); }
  public boolean equals(TreeInteger o) { return value.equals(o.value); }
  public int intValue() { return value.intValue(); }
}
```

Разумеется, если в дереве предполагается хранить в качестве операндов не только целые, а, скажем, вещественные или символьные значения, то потребуется определить и соответствующие типы констант.

Теперь неободимо обобщить понятие оператора. Конечно, несложно ввести новые бинарные операции, например, деление, сравнение (с получением значений нового типа — логических значений); несколько сложнее ввести операции с другим числом операндов (унарные или тернарные) или включить в число операций элементарные функции, такие как логарифм или синус вещественного числа. Можно сразу определить понятие оператора с произвольным числом операндов и знаком операции, состоящим из произвольного числа символов. Тогда частные случаи, такие как бинарный оператор, могли бы быть определены в виде расширения класса, задающего оператор в самом общем виде.

Введем абстрактный класс `Operator`, имеющий в своем составе знак операции в виде символьной строки и набор операндов, представленный списком элементов, каждый из которых является некоторым деревом выражения (при этом мы полагаем, что список может быть представлен объектом с интерфейсом `IList`, который был определен примерно так, как это было сделано нами в конце *разд. 1.2*). Этот класс представлен в листинге 4.7. В данном случае мы используем не определение интерфейса, как это было сделано нами, например, при формулировании понятия абстрактной константы, а определение абстрактного класса. Это позволит нам задать многие операции уже непосредственно при определении класса `Operator` — в частности, конструктор класса, доступ к знаку операции и его операндам, определение количества операндов (вычисление "арности" оператора). Обратите внимание, что для доступа к операндам оператора мы используем просто внешний итератор списка операндов. Некоторые другие операции (пока это только операция `clone`, позже мы добавим еще и операцию вычисления значения `evaluate`) присутствуют в определении класса как абстрактные.

Там же в листинге представлен и другой класс — `Binary`, — описывающий одну из возможных реализаций частного случая оператора — бинарную операцию. Конечно, операции над объектами этого класса можно сделать практически такими же, как это было сделано ранее для нашего дерева в классе `Operator`.

Листинг 4.7. Классы, определяющие операции в обобщенном дереве выражения

```
/**
 * Класс Operator определяет общий вид и наиболее общие действия
 * для операции с произвольным числом операндов
 */
```

```
public abstract class Operator {
  protected String operSign;                          // знак операции
  protected IList args = ListFactory.createList();    // список операндов

  // Конструктор создает операцию, не содержащую (пока) операндов
  public Operator(String sign) {
    operSign = sign;
  }

  // С помощью следующего метода можно добавлять операнды в операцию
  public void addOperand(ExpressionTree operand) {
    args.addLast(operand);
  }

  // Функции доступа позволяют определить знак операции...
  public String getOperSign() { return operSign; }

  // ... количество операндов и...
  public int arity() { return args.getCount(); }

  // ... перебрать операнды с помощью внешнего итератора
  public Iterator operands() {
    return args.iterator();
  }

  // Операция toString определяет наиболее общую префиксную запись
  // для операций с произвольным числом операндов. Конкретные расширения
  // класса Operator будут переопределять этот метод в соответствии со
  // своим представлением о том, как должно выглядеть выражение
  public String toString() {
    String result = operSign + '(';
    // Операнды заключаются в скобки и разделяются запятыми
    for (Iterator ops = args.iterator();  ops.hasNext();  ) {
      result += ops.next().toString();
      if (ops.hasNext()) result += ", ";
    }
    return result +')';
  }
```

```
  // Операция сравнения операторов проверяет равенство знаков операции,
  // количества операндов и самих операндов
  public boolean equals(Object other) {
    return equals((Operator)other);
  }
  public boolean equals(Operator other) {
    boolean eq = getOperSign().equals(other.getOperSign()) &&
                 arity() == other.arity();
    Iterator args1 = operands();
    Iterator args2 = other.operands();
    while (eq && args1.hasNext()) {
      eq = ((ExpressionTree)args1.next()).equals(args2.next());
    }
    return eq;
  }

  // Абстрактная операция clone определяет возможность создания копии
  public abstract Object clone();
}

/**
 * Класс Binary определяет бинарную операцию как реализацию
 * частного случая оператора общего вида
 */
public class Binary extends Operator {

  // Конструктор получает в качестве аргументов как знак операции,
  // так и оба операнда
  public Binary(String opSign, ExpressionTree op1, ExpressionTree op2) {
    super(opSign);
    addOperand(op1);
    addOperand(op2);
  }

  // Функции доступа к операндам выбирают их из списка операндов.
  // Во время программирования этих функций можно быть уверенным,
```

```
  // что при правильном создании объекта класса Binary оба операнда
  // обязательно будут присутствовать в списке операндов
  public ExpressionTree getOperand1() {
    Iterator args = operands();
    return (ExpressionTree)args.next();
  }
  public ExpressionTree getOperand2() {
    Iterator args = operands();
    args.next();
    return (ExpressionTree)args.next();
  }

  // Операция преобразования к строке переопределяет заданную в классе
  // Operator префиксную запись, устанавливая более привычную для случая
  // бинарной операции инфиксную запись
  public String toString() {
    ExpressionTree op1, op2;
    Iterator args = operands();
    op1 = (ExpressionTree)args.next();
    op2 = (ExpressionTree)args.next();
    return "(" + op1.toString() + getOperSign() + op2.toString() + ")";
  }

  // Реализация операции создания копии использует копирование операндов
  public Object clone() {
    return new Binary(getOperSign(),
                      (ExpressionTree)getOperand1().clone(),
                      (ExpressionTree)getOperand2().clone());
  }
}
```

Мы привели только реализацию бинарных операций, однако, если в дереве будут присутствовать, скажем, унарные операции и элементарные функции, то можно легко добавить дополнительные классы-наследники примерно так же, как это сделано для бинарных операторов.

Теперь мы полностью готовы к тому, чтобы определить дерево выражения самого общего вида. В узлах этого дерева будут находиться константы (вообще говоря, различных типов), переменные и операторы (вообще говоря, с

произвольным числом операндов). В листинге 4.8 показано новое определение класса `ExpressionTree`.

Листинг 4.8. Определение "обобщенного" дерева выражения

```
public class ExpressionTree {
  // Типы узлов:
  public static final int NODE_CONSTANT = 1;
  public static final int NODE_VARIABLE = 2;
  public static final int NODE_OPERATOR = 3;

  // Корень дерева представлен объектом одного из трех классов
  protected Object root;

  // Три конструктора задают корень одного из трех возможных типов
  public ExpressionTree(Constant number) { root = number; }
  public ExpressionTree(String variable) { root = variable; }
  public ExpressionTree(Operator oper) { root = oper; }

  // Функция выдает тип корня, анализируя класс соответствующего объекта
  public int getTreeClass() {
    return (root instanceof Constant ? NODE_CONSTANT :
            root instanceof String   ? NODE_VARIABLE :
           /*root instanceof Operator*/ NODE_OPERATOR);
  }

  // Преобразование дерева в строку
  public String toString() { return root.toString(); }

  // Функция создания копии дерева выражения переопределяет стандартную
  // функцию класса Object. Дополнительно вводится функция cloneTree,
  // выдающая объект класса ExpressionTree
  public Object clone() {
    return cloneTree();
  }

  public ExpressionTree cloneTree() {
    switch (getTreeClass()) {
```

```
    case NODE_CONSTANT:
      // Сама константа не копируется, но создается новый узел,
      // содержащий ту же константу
      return new ExpressionTree((Constant)root);
    case NODE_VARIABLE:
      // То же самое происходит и с переменной
      return new ExpressionTree((String)root);
    case NODE_OPERATOR:
      // Копия оператора создается с помощью его операции clone
      return new ExpressionTree(((Operator)root).clone());
    default:
      // Неизвестный тип узла?
      return null;
  }
}

// Операция сравнения деревьев сравнивает их поэлементно,
// проходя по всем узлам дерева
public boolean equals(Object obj) {
  return equals((ExpressionTree)obj);
}

public boolean equals(ExpressionTree tree) {
  // Первый фильтр проверяет, что аргумент не пуст
  if (tree == null) return false;
  // Вторым шагом сравниваются типы узлов, лежащих в корне
  int nodeClass = tree.getTreeClass();
  if (nodeClass != getTreeClass()) return false;
  // Далее результат сравнения зависит от типа узла, лежащего в корне.
  // Поскольку операция equals определена для всех типов узлов, ее
  // можно применять непосредственно, однако для случая сравнения
  // переменных использована другая операция
  switch (nodeClass) {
    case NODE_CONSTANT:
    case NODE_OPERATOR:
      return root.equals(tree.root);
    case NODE_VARIABLE:
      // Сравнение переменных делается с игнорированием регистра букв
      return ((String)root).equalsIgnoreCase((String)tree.root);
```

```
      default:
        // Других типов узлов не бывает
        return false;
    }
  }

  // Операция приема посетителей узлов
  public void accept(Visitor v) {
    switch (getTreeClass()) {
      case NODE_CONSTANT :
        v.visit((Constant)root);
        break;
      case NODE_VARIABLE:
        v.visit((String)root);
        break;
      case NODE_OPERATOR:
        v.visit((Operator)root);
        break;
    }
  }
}
```

В листинге приведены не все операции, которые были представлены для дерева раньше. Действительно, теперь функция `parse`, описанная ранее для преобразования выражения из строки в дерево, не может быть определена внутри этого класса, поскольку она приспособлена для анализа выражений вполне конкретного вида. Вообще, по-видимому, в новых условиях определить такую функцию в достаточно общем виде просто невозможно.

Вместо описания функции `parse` внутри класса `ExpressionTree` удобнее всего создать дополнительный класс, единственным назначением которого будет описание функции преобразования выражения из строки в дерево. Назовем такой класс `Parser` и просто перенесем в него из класса `ExpressionTree` определения функции `parse` и вспомогательных функций `doOperator` и `doOperators`. Не будем полностью воспроизводить код этого класса, приведем лишь его "скелет", в котором опущены тела всех функций. Конечно, этот класс приспособлен только для построения дерева из выражений конкретного вида, в нашем случае — выражений, содержащих бинарные арифметические операции, целые константы, переменные и скобки.

```
public class Parser {
  public static void doOperator(IStack operands, IStack operators)
                    throws StackUnderflow { ... }
```

```
  public static void doOperators(IStack operands, IStack operators,
                                 int minPrio)
                      throws StackUnderflow { ... }

  public static ExpressionTree parse(String source)
                      throws ParseException { ... }
}
```

На этом закончим данный раздел, посвященный представлению деревьев выражений, и перейдем к описанию алгоритмов их обработки. Будем использовать "обобщенное" дерево в тех ситуациях, когда алгоритмы обработки деревьев применимы в общем случае, и будем возвращаться к частному случаю простых формул с бинарными операторами тогда, когда алгоритм обработки не носит "всеобщего" характера.

4.2. Вычисления по формулам

В данном разделе рассмотрим алгоритмы вычисления значения выражения при заданных значениях переменных.

Прежде всего отметим, что, конечно, выражение, представленное в виде дерева, гораздо легче обрабатывать, чем выражение, представленное символьной строкой. Поэтому, несмотря на то, что в *разд. 2.2* уже был рассмотрен алгоритм вычисления значения константного выражения, чаще всего вместо него удобнее использовать алгоритм преобразования выражения в дерево, в котором уже и производить все вычисления.

Значение выражения можно вычислить только в том случае, если определены значения всех входящих в него переменных. Говорят, что должен быть задан "контекст" для вычисления выражения. Если мы хотим написать достаточно общий алгоритм, то не следует ограничиваться каким-нибудь одним представлением контекста, поэтому будем считать, что контекст переменных — это некоторый абстрактный тип данных, включающий в себя операции выдачи и записи значения переменной. Как обычно, этот абстрактный тип данных будем представлять интерфейсом `Context`. Этот интерфейс похож на стандартный интерфейс `java.util.Map` языка Java, однако, в отличие от него, в нашем контексте ключом является не произвольный объект, а строка — имя переменной, а значением, связанным с этой переменной, выступает константа. Конечно, контекст переменных — это разновидность словаря, который подробно обсуждался нами в *разд. 3.2*.

Итак, абстрактный контекст переменных может быть представлен интерфейсом, изображенным в листинге 4.9.

Листинг 4.9. Абстрактный контекст переменных

```
/**
 * Абстрактный контекст содержит в качестве ключей имена
 * переменных и в качестве значений — константы.
 */
public interface Context {
  // Найти значение переменной по заданному имени
  public Constant get(String key);
  // Связать переменную с заданным значением.
  // Результат true, если переменная уже была в контексте
  public boolean put(String key, Constant value);
  // Проверить, содержится ли переменная в контексте
  public boolean containsKey(String key);
  // Удалить переменную из контекста.
  // Результат true, если переменная была в контексте
  public boolean remove(String key);
  // Очистить контекст, удалив из него все переменные
  public void clear();
  // Проверить, есть ли в контексте хоть одна переменная
  public boolean isEmpty();
  // Количество переменных в контексте
  public int size();
}
```

В качестве реализации этого интерфейса может послужить любая реализация словаря, например, реализация в виде хэш-таблицы. В листинге 4.10 приведен пример такой реализации, основанной на стандартной хэш-таблице класса `java.util.Hashtable`.

Листинг 4.10. Реализация контекста переменных в виде хэш-таблицы

```
/**
 * Реализация контекста в виде хэш-таблицы
 */
public class HashContext implements Context {
  // Реализация контекста объектом класса Hashtable
  Hashtable context = new Hashtable();
```

```
  // Найти значение переменной по заданному имени
  public Constant get(String key) {
    return (Constant)context.get(key.toLowerCase());
  }

  // Связать переменную с заданным значением.
  // Результат true, если переменная уже была в контексте
  public boolean put(String key, Constant value) {
    return context.put(key.toLowerCase(), value) != null;
  }

  // Проверить, содержится ли переменная в контексте
  public boolean containsKey(String key) {
    return context.containsKey(key.toLowerCase());
  }

  // Удалить переменную из контекста.
  // Результат true, если переменная была в контексте
  public boolean remove(String key) {
    return context.remove(key.toLowerCase()) != null;
  }

  // Очистить контекст, удалив из него все переменные
  public void clear() {
    context.clear();
  }

  // Проверить, есть ли в контексте хоть одна переменная
  public boolean isEmpty() {
    return context.isEmpty();
  }

  // Количество переменных в контексте
  public int size() {
    return context.size();
  }
}
```

В реализации хорошо видно, что в основном мы просто использовали операции класса `Hashtable` для того, чтобы реализовать практически такие же операции интерфейса `Context`. Различие сводится, в основном, к уточнению типов данных — вместо `Object` для ключа и `Object` для связанного значения используются `String` и `Context` соответственно.

В приведенной реализации метод `get` выдает значение **`null`**, если переменная отсутствует в контексте. Таким образом, во всех алгоритмах, которые будут использовать эту реализацию контекста, следует учитывать, что переменным, значения которых в контексте не определены, не приписывается никакого значения "по умолчанию". Конечно, легко можно представить себе и другую реализацию, которая, например, возбуждает специальную исключительную ситуацию при попытке вычислить значение переменной, отсутствующей в контексте. Еще один способ представления контекста — выдавать некоторую заранее определенную константу в качестве неопределенного значения. Например, если речь идет только о целочисленных константах, то для неопределенных переменных можно в качестве значения выдавать ноль. Таким образом, различные способы реализации контекста определяют различные стратегии для вычисления значения выражения в заданном контексте.

Еще одно замечание. В операции проверки деревьев выражений на совпадение (метод `equals` класса `ExpressionTree`) мы не различали регистры букв при сравнении имен переменных. В соответствии с этим при реализации контекста мы тоже используем для хранения имен переменных и, соответственно, во время поиска значения в контексте, только регистр строчных букв. При желании это тоже нетрудно изменить: достаточно в операциях с переменными указывать имена переменных в строгом соответствии с тем, как они записаны. Конечно, при этом было бы разумно соответствующим образом изменить и реализацию метода `equals` в определении класса `ExpressionTree`.

Теперь, имея аппарат для определения значений, можно заняться собственно вычислением значения выражения.

Проще всего непосредственно в классе `ExpressionTree` определить метод `evaluate(Context)`, который и будет производить необходимые вычисления. По-видимому, в большинстве систем обработки выражений присутствие такой операции будет оправдано. Трудность здесь состоит только в том, что мы определили дерево в настолько общем виде, что теперь непонятно, сможем ли мы корректно определить достаточно общие правила вычисления значения этого выражения в заданном контексте.

Конечно, для этого необходимо, чтобы правила можно было запрограммировать для всех типов узлов такого дерева. Можно считать, что результатом вычисления значения константы всегда является сама эта константа, а результатом вычисления значения переменной выступает значение этой переменной в

заданном контексте. Однако с такой же легкостью определить значение выражения, представленного произвольным оператором, не удается.

Некоторые шаги по пути определения значения оператора в заданном контексте все-таки можно предпринять. Например, можно с большой долей достоверности считать, что для вычисления значения выражения, представленного некоторым оператором, сначала необходимо вычислить значения всех его операндов. На самом деле это не вполне очевидно, некоторые хорошо известные операции, такие как, например, знакомые по языкам Си и Java операции логических "И" и "ИЛИ" (`&&` и `||`) этому условию не удовлетворяют. Однако для таких особых случаев правило вычисления значений выражения всегда можно переопределить.

После того как значения операндов вычислены, надо применить операцию к вычисленным значениям. Однако семантика выполнения операции в общем виде нам неизвестна. Поэтому лучше всего определить операцию вычисления значения выражения, заданного оператором с известными значениями операндов, как абстрактную. Разумеется, в каждой конкретной реализации оператора (например, в классе `Binary`) будет необходимо задать семантику этой операции.

В листинге 4.11 приведены определения необходимых функций для вычисления значения выражения в классах `Operator`, `Binary` и `ExpressionTree`.

Листинг 4.11. Операция вычисления значения выражения в заданном контексте

```
// Реализация функций для вычисления значения выражения в классе Operator
public abstract class Operator {
  // Абстрактная функция для вычисления значения на основе значений
  // операндов получает итератор значений аргументов (класса Constant)
  // и выдает результат также класса Constant
  public abstract Constant evaluate(Iterator constArgs);

  // Функция вычисления значения в заданном контексте формирует список
  // значений операндов, используя определенную далее функцию для
  // вычисления значения выражения в классе ExpressionTree,
  // а затем применяет определенную выше абстрактную функцию
  // для получения результата
  public Constant evaluate(Context ctx) {
    IList ops = new List();   // список значений операндов
    for (Iterator it = operands(); it.hasNext(); ) {
      ops.addLast(((ExpressionTree)it.next()).evaluate(ctx));
```

```
    }
    return evaluate(ops.iterator());
  }
}

// Реализация функций для вычисления значения выражения в классе Binary
public class Binary extends Operator {
  // Функция вычисления значения бинарной операции предполагает,
  // что вычисленные значения операндов принадлежат классу TreeInteger,
  // определенному нами как одна из реализаций интерфейса Constant.
  // Если это не так, то возникнет исключительная ситуация
  // ClassCastException. Если хотя бы один из операндов имеет значение
  // null, то и все выражение будет иметь значение null.
  public Constant evaluate(Iterator args) {
    TreeInteger op1 = null, op2 = null;
    // Выборка значений операндов
    if (args.hasNext()) {
      op1 = (TreeInteger)args.next();
      if (args.hasNext()) {
        op2 = (TreeInteger)args.next();
      }
    }
    // Проверка непустоты операндов
    if (op1 == null || op2 == null) return null;
    // Выполнение операции согласно знаку операции
    switch (getOperSign().charAt(0)) {
      case '+': return new TreeInteger(op1.intValue() + op2.intValue());
      case '-': return new TreeInteger(op1.intValue() - op2.intValue());
      case '*': return new TreeInteger(op1.intValue() * op2.intValue());
      default:  return null;
    }
  }
}

// Реализация функций для вычисления значения выражения
// в классе ExpressionTree
public class ExpressionTree {
  public Constant evaluate(Context ctx) {
    switch (getTreeClass()) {
```

```
      case NODE_CONSTANT:
        // Значение константы выдается непосредственно
        return (Constant)root;
      case NODE_VARIABLE:
        // Значение переменной берется из контекста
        return ctx.get((String)root);
      case NODE_OPERATOR:
        // Значение оператора вычисляется с помощью определенного выше
        // метода evaluate класса Operator
        return ((Operator)root).evaluate(ctx);
      default:
        return null;
    }
  }
}
```

Приведенная реализация очень похожа на операции создания копии выражения и проверки совпадения выражений. Если задать дерево выражения и контекст входящих в него переменных с помощью операторов[1]

```
ExpressionTree tree = Parser.parse("(a + b)*(c + d) + 1 + 2*a");
Context vars = new HashContext();
vars.put("a", 1);
vars.put("b", 2);
vars.put("c", 3);
vars.put("d", 4);
```

то оператор

```
System.out.println("Результат выражения " + tree.toString() + " равен " +
                   tree.evaluate(vars));
```

должен напечатать следующую строку:

```
Результат выражения ((((a+b)*(c+d))+1)+(2*a)) равен 24
```

Второй вариант операции для вычисления значения выражения состоит в том, что вместо определения метода `evaluate` внутри самого класса `ExpressionTree` можно описать класс-посетитель, задачей которого будет обход узлов выражения и "сборка" значения выражения по ходу посещения узлов. Данное решение является "внешним" по отношению к классу

[1] Предполагается, что `Parser` — это класс, определенный для операции преобразования из строки в дерево выражения, содержащее переменные, целые константы класса `TreeInteger` и бинарные арифметические операторы класса `Binary`.

ExpressionTree, т. е. не затрагивает его определения. Таким образом, правила вычисления выражения (метод evaluate) выносятся из класса ExpressionTree и определяются вместо этого в специально сконструированном для вычисления выражений посетителе.

Посетитель для вычисления значения выражения может быть построен по той же схеме, что и посетитель для создания копии выражения из предыдущего раздела. При посещении простых узлов (узлов типа "константа" и "переменная") он непосредственно выдает значение, хранящееся в узле или в переданном ему контексте. При посещении узла типа "оператор" посетитель вычислит искомое значение путем применения абстрактной функции evaluate класса Operator.

Определение "вычислителя значения" приведено в листинге 4.12.

Листинг 4.12. Посетитель узлов дерева выражения для вычисления значения выражения

```
public class ExprEvaluator implements Visitor {
  Context context;          // контекст для вычисления выражения
  Constant result = null;   // результат вычисления

  // Конструктор получает контекст в качестве аргумента
  public ExprEvaluator(Context table) {
    context = table;
  }

  // Функция доступа к результату вычислений
  public Constant getResult() { return result; }

  // Функции посещения узлов разных типов.
  // Посетитель узла типа "целочисленная константа" берет значение
  // из узла
  public void visit(Constant intNode) {
    result = intNode;
  }

  // Посетитель узла типа "переменная" берет значение из контекста
  public void visit(String varNode) {
    result = context.get(varNode);
  }
```

```
  // Посетитель узла типа "знак операции" вычисляет значение
  // по значениям операндов и знаку операции
  public void visit(Operator operNode) {
    result = operNode.evaluate(context);
  }
}
```

В общем, реализация вычислителя выражений с помощью посетителя отличается от реализации с помощью внутреннего метода `evaluate` только оформлением, однако все же она носит более общий характер. Как ни просто определение функции `evaluate` класса `ExpressionTree`, все же оно содержит в себе, например, тот факт, что значением любой константы является она сама. В то же время реализация вычисления выражения с помощью посетителя может трактовать значение константы и чуточку по-другому, например, считая значением целочисленной константы объект класса `Integer` или значение типа **`int`**.

Пример применения операции вычисления значения выражения будет теперь выглядеть чуть-чуть сложнее (считаем, что дерево `tree` и контекст вычислений `vars` заданы так же, как и в предыдущем случае):

```
ExprEvaluator evaluator = new ExprEvaluator(vars);
tree.accept(evaluator);
System.out.println("Результат выражения " + tree + " равен " +
                   evaluator.getResult());
```

Все это может показаться чересчур сложным для такой, в общем-то, несложной задачи, как вычисление значения выражения по заданным значениям переменных, но вы сможете ощутить преимущества подобной организации вычислений, если попробуете расширить область применимости приведенного примера. Попытайтесь, например, заменить целочисленные константы на вещественные, расширить набор допустимых операций, ввести унарные операции и элементарные функции. Вы увидите, насколько просто и естественно осуществляются такие изменения, и именно этот факт и является основным достоинством приведенного подхода.

4.3. Преобразование формул

В этом разделе мы приведем несколько алгоритмов различных преобразований заданного дерева выражения.

Сразу же оговоримся, что под "преобразованием" всегда будет пониматься не столько изменение структуры и содержания имеющегося выражения, сколько построение нового дерева выражения, основанного на некотором исходном дереве выражения. Такой подход может показаться не очень есте-

ственным, однако, если вспомнить, что наше дерево выражения представляет собой "константное" выражение, то все встанет на свои места. Действительно, при нашем определении дерева выражения у нас просто нет средств для изменения выражения "на месте", но зато мы всегда можем использовать при всех построениях любые части исходного дерева, не опасаясь, что эти части могут впоследствии измениться под воздействием тех или иных операций.

Может показаться, что "константность" выражений приводит к излишним расходам ресурсов памяти и времени, поскольку мы всегда вместо локальных изменений внутренней структуры дерева будем строить новую структуру данных. Однако и это не совсем верно. Определенные выгоды мы сможем получить, например, в том случае, когда отдельные одинаковые части строящегося дерева встречаются в дереве неоднократно. В этом случае мы имеем возможность использовать только один экземпляр такой повторяющейся части, а это будет означать и экономию памяти, и в некоторых случаях даже экономию времени (правда, это произойдет только в том случае, когда все же некоторые изменения в структуре дерева будут допустимы; в данном разделе такие ситуации оговорены особо).

В данном разделе мы рассмотрим три случая преобразования выражений: подстановку в дерево вместо переменных других подвыражений, упрощение формул, дифференцирование формулы.

Первая из этих задач — подстановка переменных — возникает при обработке выражений достаточно часто как результат ведения аналитических преобразований. Все школьные задачи по физике обычно сводятся в конце концов к тому, что в одни известные формулы вместо некоторых переменных подставляются другие формулы, а затем производится упрощение полученного результата.

Вторая задача — дифференцирование формулы — часто используется при программировании в качестве примера преобразования с хорошо известными простыми правилами, которые, тем не менее, могут привести к значительным изменениям и усложнению исходного выражения.

Третья задача — это задача упрощения выражения. В большинстве случаев упрощение совершенно необходимо делать после сложных преобразований, например, таких как в первых двух задачах. Упрощение в обоих случаях позволит представить результаты обработки в более компактном виде.

Рассмотрим теперь эти три задачи более подробно.

Задача о подстановке переменных внешне выглядит очень похожей на задачу о вычислении значения выражения. Различие состоит лишь в том, что в случае вычисления значения контекст переменных хранит в качестве значения переменных константы, и, соответственно, результатом всего процесса будет вычисленное значение константы. В случае же подстановки контекст переменных будет содержать в качестве значений переменных подставляе-

мые подвыражения и, соответственно, результатом "вычислений" будет измененное дерево выражения.

Существенное отличие этой задачи от задачи вычисления выражения заключается в том, что в принципе подстановку значений переменных можно произвести, даже не зная, какие конкретно операторы используются в исходном выражении. Действительно, все, что нужно сделать, — это выполнить подстановку во всех операндах операции, независимо от семантики или внешнего представления этой операции. На первый взгляд кажется, что это позволит написать функцию подстановки независимо от информации о том, какие именно операторы используются в выражении. Однако здесь имеется одна трудность.

Если мы хотим сохранить свойство константности деревьев выражений, то в ходе операции подстановки нам надо будет создавать новые узлы, содержащие те же операторы, что и узлы исходного дерева. Без знания о том, что это за операции, невозможно использовать для данной цели конструктор. Правда, в классе `Operator` определена еще абстрактная функция `clone`, с помощью которой также можно создавать новые объекты, однако при этом создается копия узла, содержащая, естественно, те же самые операнды, что и исходный узел.

Очевидно, в определенном нами классе `Operator` не хватает средств для того, чтобы можно было более свободно манипулировать операндами. Такие средства можно добавить разными путями, но нужно иметь в виду, что в некоторых из таких способов реализации можно потерять свойство константности дерева выражения. В качестве простого решения проблемы можно предложить добавить в абстрактный класс `Operator` еще одну абстрактную функцию — `cloneWithNewArgs`, которая должна для заданного оператора построить новый оператор с тем же самым знаком операции, но с другими значениями операндов. Реализации абстрактного оператора (такие, как `Binary`) должны определять реализацию этой функции так, чтобы она создавала копию оператора с новыми значениями операндов. Вот как будут выглядеть теперь классы `Operator` и `Binary`. (показаны только определения новой функции `cloneWithNewArgs`).

```
public abstract class Operator {
  ...
  // Создание копии оператора с новыми значениями операндов.
  // Аргумент newArgs задает новые значения операндов
  public abstract Operator cloneWithNewArgs(Iterator newArgs);
}

public class Binary extends Operator {
  // Новый конструктор используется для создания бинарного оператора
```

```
  // без операндов
  protected Binary(String operSign) { super(operSign); }

  ...
  // Реализация функции cloneWithNewArgs ожидает ровно два операнда.
  // Если операндов будет меньше, то в качестве них будут
  // подставлены пустые указатели. Если операндов окажется больше,
  // то все операнды, кроме первых двух, игнорируются
  public Operator cloneWithNewArgs(Iterator newArgs) {
    // Создаем новый экземпляр бинарного оператора без операндов
    Binary result = new Binary(getOperSign());
    // Добавляем новые операнды, взятые из итератора newArgs,
    // если они действительно есть
    result.addOperand(
           newArgs.hasNext() ? (ExpressionTree)newArgs.next() : null);
    result.addOperand(
           newArgs.hasNext() ? (ExpressionTree)newArgs.next() : null);
    return result;
  }
}
```

Теперь, имея возможность создавать копию оператора произвольного типа с новыми значениями аргументов, мы можем полностью реализовать подстановку переменных с помощью посетителя, который будет использовать контекст переменных для выполнения этой задачи. В листинге 4.13 представлена реализация контекста переменных со значениями в виде деревьев выражений и реализация посетителя, формирующего при обходе узлов дерева выражения новое дерево, в котором все переменные заменены на поддеревья, взятые из заданного контекста.

Листинг 4.13. Решение задачи подстановки переменных в заданном выражении

```
/**
 * Реализация контекста переменных со значениями—деревьями выражений
*/
public class ExprContext {

  // Реализация также основана на стандартном классе Hashtable
  Hashtable context = new Hashtable();
```

```
  // Выборка значения переменной по ключу. Регистр букв игнорируется
  public ExpressionTree get(String key) {
    return (ExpressionTree)context.get(key.toLowerCase());
  }

  // Запись нового значения переменной.
  // Результат true, если переменная уже имеется в контексте
  public boolean put(String key, ExpressionTree value) {
    return context.put(key.toLowerCase(), value) != null;
  }

  // Проверка наличия переменной в контексте
  public boolean containsKey(String key) {
    return context.containsKey(key.toLowerCase());
  }

  // Удаление переменной из контекста.
  // Результат true, если переменная была ранее в контексте
  public boolean remove(String key) {
    return context.remove(key.toLowerCase()) != null;
  }

  // Удаление всех переменных из контекста
  public void clear() {
    context.clear();
  }

  // Проверка пустоты контеста
  public boolean isEmpty() {
    return context.isEmpty();
  }

  // Количество имеющихся переменных в контексте
  public int size() {
    return context.size();
  }
}
```

```
/**
 * Реализация посетителя узлов дерева выражения для замены
 * переменных на заданные в контексте подвыражения
 */
public class Substitutor implements Visitor {
  ExprContext context;           // контекст переменных
  ExpressionTree result = null;  // результат подстановки
  // Конструктор берет в качестве аргумента контекст переменных
  public Substitutor(ExprContext ctx) {
    context = ctx;
  }

  // Функция доступа к результату посещения узла — новому дереву
  public ExpressionTree getResult() {
    return result;
  }

  // Функции посещения
  public void visit(Constant constNode) {
    // Константа остается без изменений
    result = new ExpressionTree(constNode);
  }

  public void visit(String varNode) {
    // Значение переменной берется из контекста
    result = context.get(varNode);
    // Если переменной нет в контексте, то результатом будет
    // сама эта переменная
    if (result == null) {
      result = new ExpressionTree(varNode);
    }
  }

  public void visit(Operator operNode) {
    // Для подстановки в оператор создается итератор,
    // выдающий новые значения операндов.
    // Этот итератор берет старые операнды и заменяет их
    // на результат подстановки, полученный с помощью
```

```
    // рекурсивного вызова посетителя
    Operator newNode = operNode.cloneWithNewArgs(
                new NewArgsIterator(operNode.operands()));
    result = new ExpressionTree(newNode);
  }

  /**
   * Реализация итератора для подстановки в операнды оператора
   * вместо переменных их значений из заданного контекста.
   * Использует контекст переменных, заданный во внешнем классе.
   */
  private class NewArgsIterator implements Iterator {
    Iterator oldArgs;         // итератор имеющихся операндов

    // Конструктор получает итератор имеющихся операндов
    // в качестве аргумента
    public NewArgsIterator(Iterator args) { oldArgs = args; }

    // Метод hasNext() просто проверяет, есть ли еще операнды
    public boolean hasNext() { return oldArgs.hasNext(); }

    // Метод next() производит подстановку переменных в следующем
    // операнде, и только после этого выдает получившееся значение
    public Object next() {
      // Берем очередной операнд, если он есть
      ExpressionTree nextArg = (ExpressionTree)oldArgs.next();
      if (nextArg == null) return null;
      // Строим новый экземпляр "подстановщика" для подстановки
      // в очередной операнд и применяем его к операнду
      Substitutor argVisitor = new Substitutor(context);
      nextArg.accept(argVisitor);
      // Выдаем результат подстановки
      return argVisitor.getResult();
    }

    // Реализация метода remove(), как и метода hasNext(),
    // не вносит ничего нового
    public void remove() { oldArgs.remove(); }
  }
```

```
}
```

С помощью посетителя `Substitutor` мы можем подставлять вместо некоторых (быть может, всех) переменных выражения. Например, давайте подставим в формулу для кинетической энергии материальной точки

$$m \times v^2 / 2$$

вместо переменной v формулу мгновенной скорости при равноускоренном движении с начальной скоростью v_0 и ускорением a в момент времени t

$$v = v_0 + a \times t.$$

Если считать, что в выражении допустима операция деления, а операция возведения в квадрат выражена с помощью умножения, то такую подстановку можно выразить с помощью следующих операторов.

```
// Построим дерево формулы для кинетической энергии
ExpressionTree energy = Parser.parse("m*v*v/2");
// Создадим контекст, в котором переменной v соотвествует формула
ExprContext speed = new ExprContext();
speed.put("v", Parser.parse("v0 + a*t"));
// Посетитель substitutor использует построенный контекст для замены
Substitutor substitutor = new Substitutor(speed);
energy.accept(substitutor);
// Берем новую формулу и печатаем ее
energy = substitutor.getResult();
System.out.println(energy);
```

Исходное дерево выражения, полученное после анализа выражения `m*v*v/2`, будет выглядеть так, как показано на рис. 4.6. Дерево, представляющее выражение `v0 + a*t`, используемое в подстановке вместо переменной `v`, представлено на рис. 4.7. Наконец, результат подстановки представлен на рис. 4.8. Обратите внимание, что при подстановке одно и то же дерево, содержащееся в контексте переменных, было использовано дважды, поэтому в результирующем дереве образуются две ссылки на одно и то же поддерево. Строго говоря, результат не будет являться деревом.

В результате исполнения этих операторов в выходном потоке будет получена следующая строка:

```
(((m*(v0+(a*t)))*(v0+(a*t)))/2)
```

Теперь можно продолжить пример. Добавим в контекст переменных подстановку вместо начальной скорости v_0 нулевого значения и произведем подстановку.

```
speed.put("v0", Parser.parse("0"));
```

```
energy.accept(substitutor);
energy = substitutor.getResult();
System.out.println(energy);
```

В процессе подстановки будет создана новая копия дерева выражения, поэтому часть дерева, представляющая подвыражение `v0 + a*t`, будет дублирована, и дерево приобретет вид, показанный на рис. 4.9.

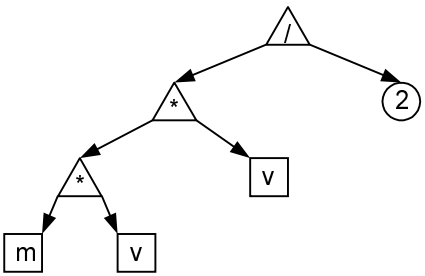


Рис. 4.6. Исходное дерево выражения

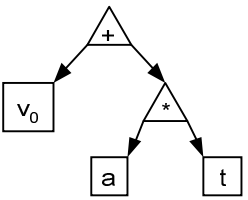


Рис. 4.7. Дерево выражения, подставляемого вместо переменной *v*

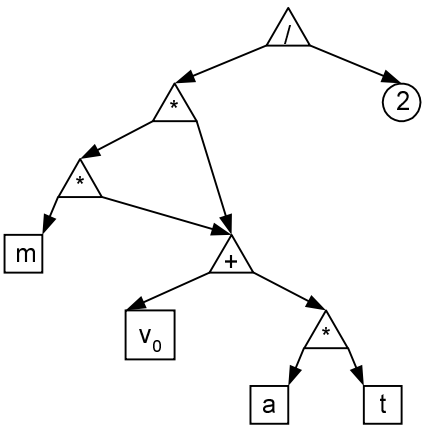


Рис. 4.8. Результат подстановки

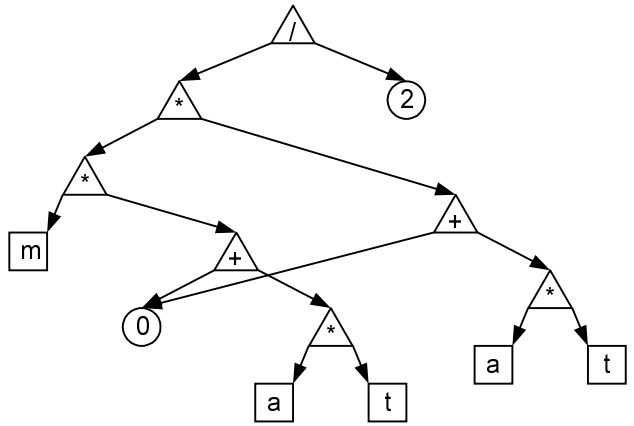


Рис. 4.9. Результат подстановки в выражение значения $v_0 = 0$

Результат исполнения всей последовательности операторов представлен в следующей строке:

```
(((m*(0+(a*t)))*(0+(a*t)))/2)
```

Конечно, надо бы упростить полученное выражение, убрав нулевое слагаемое, однако пока мы не имеем средств сделать это. Нашей следующей задачей как раз и будет упрощение формул.

При упрощении формулы производится замена узлов дерева выражения на другие узлы, вообще говоря, имеющие более простую структуру. Опять возникает вопрос о "константности", и опять будем разрешать его теми же методами: вместо изменения структуры имеющегося дерева построим на его основе новое дерево с более простой структурой.

Насколько серьезными могут быть упрощения? Это, конечно, зависит от глубины анализа выражения, который вы хотите и можете проделать. Мы ограничимся следующими простыми преобразованиями:

- добавление и вычитание нуля можно убрать; результатом упрощения будет второй операнд;
- умножение и деление на единицу можно убрать; результатом упрощения также будет второй операнд;
- умножение на ноль и ноль, деленный на ненулевое значение, дают в результате ноль;
- операции над константными значениями можно выполнить (для этого пригодится метод `evaluate` с пустым контекстом переменных) и оператор заменить на полученный результат.

Чтобы проделать все эти преобразования, запрограммируем посетителя, который при обнаружении бинарной операции будет исследовать операнды в

поисках одного из четырех перечисленных случаев и, в случае удачного поиска, станет выполнять соответствующие преобразования (точнее, строить новый узел в качестве результата упрощения).

Несмотря на то, что наш посетитель будет написан, вообще говоря, для абстрактного дерева, он, конечно, сможет выполнять преобразования только для тех операторов, смысл которых ему известен. Поэтому преобразования будут выполнены только для операторов, представляющих бинарные операции. Определение соответствующего класса представлено в листинге 4.14.

Листинг 4.14. Посетитель для упрощения дерева выражения

```
/**
 * Посетитель Simplifier посещает узлы дерева выражения и строит новое,
 * более простое дерево, представляющее результат упрощения выражения
 */
public class Simplifier implements Visitor {
  // Результат упрощения — новое построенное дерево
  ExpressionTree result = null;

  // Функция доступа к построенному результату
  public ExpressionTree getResult() { return result; }

  // Константа не упрощается, а просто копируется в результат
  public void visit(Constant constNode) {
    result = new ExpressionTree(constNode);
  }

  // Переменная также не подлежит упрощению
  public void visit(String varNode) {
    result = new ExpressionTree(varNode);
  }

  // Упрощение оператора возможно только в том случае,
  // когда оператор представлен бинарной операцией
  public void visit(Operator operNode) {
    if (operNode instanceof Binary) {
      Binary binOperator = (Binary)operNode;

      // Выделяем отдельные элементы бинарной операции.
```

```
// Прежде всего, выбираем операнды, пробуем упростить их с помощью
// специально созданных для этого посетителей класса Simplifier
Simplifier visitor1 = new Simplifier();
binOperator.getOperand1().accept(visitor1);
ExpressionTree op1 = visitor1.getResult();  // первый операнд

Simplifier visitor2 = new Simplifier();
binOperator.getOperand2().accept(visitor2);
ExpressionTree op2 = visitor2.getResult();  // второй операнд

// Теперь специально для сравнений создадим две константы -
// ноль и единицу — в виде деревьев выражений
ExpressionTree zero = new ExpressionTree(new TreeInteger(0));
ExpressionTree unit = new ExpressionTree(new TreeInteger(1));
// Случай 1: 0+e, 1*e заменяем на e
if ((binOperator.getOperSign().equals("+") &&
     op1.equals(zero)) ||
    (binOperator.getOperSign().equals("*") &&
     op1.equals(unit))) {
          result = op2;
// Случай 2: e+0, e-0, e*1, e/1 заменяем на e
} else if (((binOperator.getOperSign().equals("+") ||
             binOperator.getOperSign().equals("-")) &&
            op2.equals(zero)) ||
           ((binOperator.getOperSign().equals("*") ||
             binOperator.getOperSign().equals("/")) &&
            op2.equals(unit))) {
          result = op1;
// Случай 3: 0*e, e*0, 0/e заменяем на ноль
} else if (binOperator.getOperSign().equals("*") &&
             (op1.equals(zero) || op2.equals(zero)) ||
           binOperator.getOperSign().equals("/") &&
              op1.equals(zero) && !op2.equals(zero)) {
          result = zero;
// Случай 4: c+c, c-c, c*c, где c — константы,
// заменяем на результат выполнения операции
} else if (op1.getTreeClass() == ExpressionTree.NODE_CONSTANT &&
           op2.getTreeClass() == ExpressionTree.NODE_CONSTANT) {
      // Сначала строим дерево из знака операции
```

```
            // и двух константных операндов...
            result = new ExpressionTree(
                        new Binary(binOperator.getOperSign(), op1, op2));
            // ... а затем выполняем операцию с помощью вызова метода
            // evaluate с пустым контекстом в качестве аргумента
            result = new ExpressionTree(result.evaluate((Context)null));
        } else {
          // Собираем узел из знака операции и упрощенных операндов
          result = new ExpressionTree(
                      new Binary(binOperator.getOperSign(), op1, op2));
        }
      } else {
        // Оператор, не представленный бинарной операцией,
        // просто копируется в результат
        result = new ExpressionTree(operNode);
      }
    }
}
```

В приведенном алгоритме сначала рекурсивно упрощаются операнды бинарной операции, причем для этого используются дополнительно созданные экземпляры посетителя `Simplifier`. На самом деле можно было взять тот же самый экземпляр посетителя, "проталкивая" его в операнды и получая результат на выходе, но приведенный вариант кажется более ясным и естественным. После упрощения операндов разбираются частные случаи возможного упрощения анализируемой операции, причем для этого используются некоторые из ранее разобранных операций: для сравнения с константами служит операция `equals`, определенная для деревьев, а для константных вычислений — операция `evaluate` с пустым контекстом переменных.

Вспомним теперь, что при подстановке в формулу для вычисления кинетической энергии вместо переменной v_0 нулевого значения мы получили выражение

```
(((m*(0+(a*t)))*(0+(a*t)))/2)
```

Давайте упростим теперь это выражение с помощью посетителя класса `Simplifier`.

```
Simplifier simplifier = new Simplifier();
energy.accept(simplifier);
energy = simplifier.getResult();
System.out.println(energy);
```

Выражение упростится до

```
(((m*(a*t))*(a*t))/2)
```

Если теперь подставить нулевое значение также и вместо переменной *a*, а затем вновь упростить полученный результат,

```
speed.put("a", Parser.parse("0"));
energy.accept(substitutor);
energy = substitutor.getResult();
energy.accept(simplifier);
energy = simplifier.getResult();
System.out.println(energy);
```

то выражение упростится до нуля, и в выходном потоке окажется результат:

```
0
```

Подстановка и упрощение — это, наверное, наиболее часто употребляющиеся и естественные способы преобразования выражений. Однако, так же просто можно реализовать и более сложные преобразования. В качестве последнего примера в этой главе рассмотрим дифференцирование (вычисление первой производной) выражения по заданной переменной. Опять ограничимся только дифференцированием бинарных операций, констант и переменных, однако, если выражение будет составлено из других элементов, включающих, например, элементарные функции, то алгоритм легко можно распространить и на эти дополнительные случаи.

Напомним правила построения первой производной для всех использующихся нами типов выражений:

- производная любой константы есть ноль;
- производная переменной есть единица, если это та переменная, по которой производится дифференцирование, в остальных случаях производная переменной равна нулю;
- производные для четырех арифметических операций вычисляются по следующим формулам:
  - $(u \pm v)' = u' \pm v'$;
  - $(u \times v)' = u' \times v = u \times v'$;
  - $(u / v)' = (u' \times v - u \times v') / v^2$.

Используя эти формулы, несложно написать соответствующий преобразователь выражения по той же самой схеме, что уже использовалась нами во всех примерах данного раздела. Посетитель `Diff`, представленный в листинге 4.15, выполняет эту работу.

Листинг 4.15. Посетитель, вычисляющий первую производную выражения

```
public class Diff implements Visitor {
  // Опишем две константы, представляющие деревья для нуля и единицы
  final static ExpressionTree zero =
               new ExpressionTree(new TreeInteger(0));
  final static ExpressionTree unit =
               new ExpressionTree(new TreeInteger(1));

  String variable;         // переменная, по которой берется производная
  ExpressionTree result = null;    // результат вычислений

  // Конструктор получает переменную, по которой берется производная,
  // в качестве аргумента и запоминает ее
  public Diff(String variable) { this.variable = variable; }
  // Функция доступа к результату вычислений
  public ExpressionTree getResult() { return result; }

  // Производная константы есть ноль
  public void visit(Constant constNode) {
    result = zero;
  }

  // Производная переменной есть ноль либо единица.
  // Сравнение переменных производится без учета регистра символов
  public void visit(String varNode) {
    result = (varNode.equalsIgnoreCase(variable) ? unit : zero);
  }

  // Наиболее сложная работа выполняется при посещении бинарной операции
  public void visit(Operator operNode) {
    // Умеем дифференцировать только бинарные операторы
    if (operNode instanceof Binary) {
      Binary binOperator = (Binary)operNode;

      // Сделаем выборку операндов и вычислим их производные.
      // Первый операнд:
      ExpressionTree op1 = binOperator.getOperand1();
      Diff visitor1 = new Diff(variable);
      op1.accept(visitor1);
```

```
// Производная первого операнда:
ExpressionTree op1dash = visitor1.getResult();

// Второй операнд:
ExpressionTree op2 = binOperator.getOperand2();
Diff visitor2 = new Diff(variable);
op2.accept(visitor2);
// Производная второго операнда:
ExpressionTree op2dash = visitor2.getResult();

// 1. Производная суммы и разности: u + v,  u — v
if (binOperator.getOperSign().equals("+") ||
    binOperator.getOperSign().equals("-")) {
      result = new ExpressionTree(
                  new Binary(binOperator.getOperSign(),
                             op1dash, op2dash));
// 2. Производная произведения: u * v
} else if (binOperator.getOperSign().equals("*")) {
      result = new ExpressionTree(
                  new Binary("+",
                      new ExpressionTree(
                          new Binary("*", op1dash, op2)),
                      new ExpressionTree(
                          new Binary("*", op1, op2dash))));
// 3. Производная частного: u / v
} else if (binOperator.getOperSign().equals("/")) {
      result =
          new ExpressionTree(
              new Binary("/",
                  new ExpressionTree(
                      new Binary("-",
                          new ExpressionTree(
                              new Binary("*", op1dash, op2)),
                          new ExpressionTree(
                              new Binary("*", op1, op2dash)))),
                  new ExpressionTree(
                      new Binary("*", op2, op2))));
```

```
      } else {
        // Для других знаков операций результат не определен
        result = null;
      }
    } else {
      // Для операторов других типов результат не определен
      result = null;
    }
  }
}
```

Так же, как и в случае подстановки, после операции дифференцирования дерево, строго говоря, может оказаться уже не деревом из-за того, что фактически из разных узлов этого "дерева" могут образоваться ссылки на одни и те же подвыражения. Давайте проследим, как меняется дерево при выполнении операции дифференцирования на простом примере.

На рис. 4.10 изображено дерево выражения $(x + 1)/(x - 1)$. Узлы этого дерева, как и раньше, изображены геометрическими фигурами разной формы в зависимости от типа узла: квадраты содержат переменные, круги — константы, а треугольники — знаки бинарных операций. Такое дерево могло бы быть порождено с помощью вызова функции

```
Parser.parse("(x+1)/(x-1)");
```

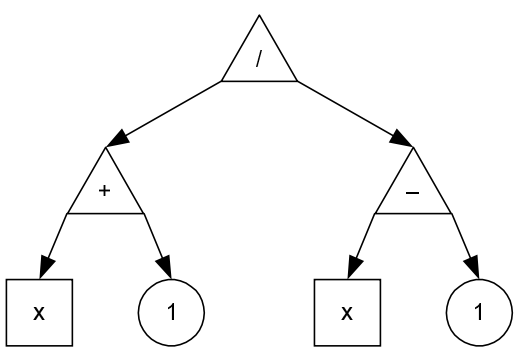


Рис. 4.10. Дерево выражения $(x + 1)/(x - 1)$

После дифференцирования будет построено дерево, содержащее довольно большое количество нулей и единиц, причем эти нули и единицы не строятся каждый раз заново, а постоянно используются одни и те же константы, определенные как статические поля в классе `Diff`. Кроме того, во время

дифференцирования частного одни и те же выражения используются при построении бинарных операций многократно. Поэтому после дифференцирования будет построена довольно сложная конструкция, которую можно изобразить так, как показано на рис. 4.11.

Конечно, получившееся дерево можно и нужно упростить. В результате упрощения многие узлы исчезнут, но некоторые узлы — наоборот, появятся! Это произойдет потому, что посетитель `Simplifier` некоторые узлы дерева игнорирует, а другие — создает заново (копирует), так что после упрощения дерево приобретет вид, как на рис. 4.12. Структура дерева после обработки становится действительно более простой, однако количество узлов в обеих структурах оказалось одинаковым. Правда, количество связей после упрощения несколько сократилось: в дереве, построенном "упростителем" вместо 18 ссылок осталось всего 14.

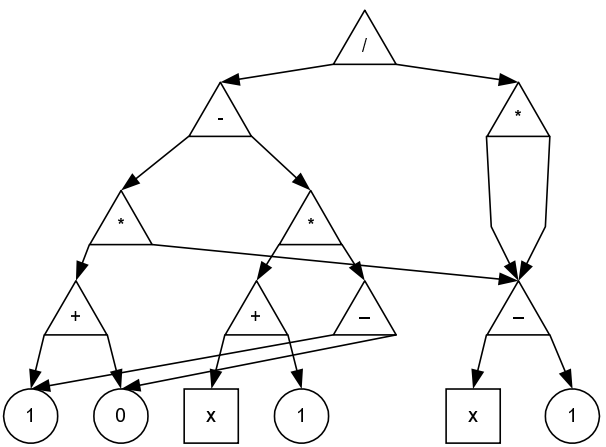


Рис. 4.11. Дерево, получившееся после дифференцирования выражения $(x + 1)/(x - 1)$

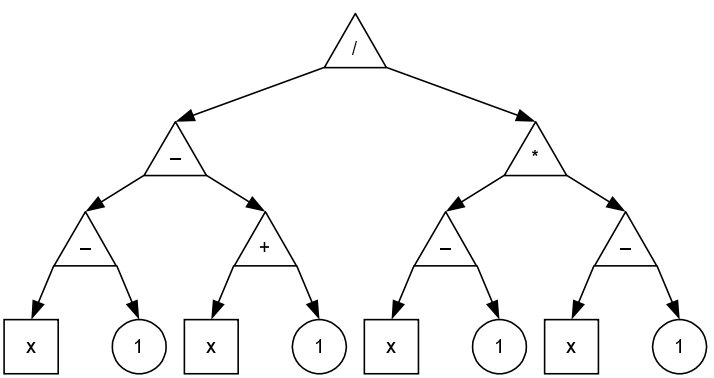


Рис. 4.12. После упрощения результата дифференцирования

В заключение данного раздела — одно замечание.

Попробуйте ввести в набор используемых операторов операцию возведения в целую неотрицательную степень. Соответственно, вам придется добавить новые правила для константных вычислений, новые правила упрощения (выражение в нулевой степени равно единице и др.), новые правила дифференцирования формул. Если вы проделаете все это, то увидите, что изменение состава и структуры выражений обходится довольно дорого — надо внести много мелких изменений в разные места реализации. Это говорит о некоторой ограниченности нашего подхода к организации дерева выражения. Он хорошо приспособлен только для случая, когда в процессе реализации структура выражений меняется редко. Зато очень легко вводить новые способы обработки выражений — это мы видели на примере определения новых операций. Тем не менее, в рамках этой книги мы не будем рассматривать других подходов к организации сложных структур данных вообще и, в частности, деревьев выражений.

Глава 5

Алгоритмы обработки сетевой информации

В этой главе представлены некоторые классические алгоритмы обработки графов и приведены примеры их использования. Алгоритмы на графах традиционно относятся к наиболее сложным, в этой области было получено много интересных результатов. Большое количество литературы посвящено описаниям алгоритмов на графах, достаточно упомянуть такие фундаментальные работы, как [1], [7] и [8]. В нашу задачу не входит глубокий анализ этих алгоритмов, мы даже не будем пытаться находить во всех случаях оптимальные решения. Задача этой главы состоит в том, чтобы определить, как выбранное представление данных влияет на выбор алгоритмов, и, кроме того, интересно посмотреть, как реализуются классические алгоритмы на языке Java с применением современных технологий создания программ.

5.1. Обходы и поиск по сети

Основу многих алгоритмов обработки сетевой информации составляют алгоритмы обхода (итерации) графов, в процессе которого производится поиск необходимой информации или определение каких-либо характеристик сети. Если граф имеет N вершин и M дуг, то говорят, что алгоритм обхода индуцирует нумерацию вершин и дуг графа, приписывая им номера от 1 до N и от 1 до M соответственно в порядке обхода. Итераторы графа могут быть как внешними, так и внутренними, при этом внешний итератор, как обычно, выдает вершины или дуги графа в порядке обхода, а внутренний итератор, обходя граф, посещает его вершины и дуги и выполняет в каждой из них "процедуру посещения".

В процессе обхода обычно используется информация о связях между вершинами, т. е. алгоритмы обходы, просматривая граф, переходят всегда от вершины к одной из связанных с нею вершин. Разумеется, это не обязательно означает, что порядок обхода вершин также таков, что за одной вершиной непременно в качестве следующей вершины идет одна из связанных с ней. Более того, в большинстве случаев такой обход построить просто невозможно (обходы, описанные выше, называются *Гамильтоновыми путями*. Для существования Гамильтонова пути граф должен удовлетворять опреде-

ленным условиям). Но алгоритмы обхода просто используют связи между вершинами для перехода от одних вершин к другим.

Если дуги, связывающие вершины графа, — *направленные* (с технической точки зрения это означает, что если от вершины *A* можно перейти к вершине *B*, то это не означает, что от вершины *B* можно обязательно непосредственно перейти к вершине *A*), то даже для связного графа не всегда удается, выбрав некоторую вершину в качестве исходной, обойти его весь, проходя только по направлениям дуг. В этом случае часто поступают следующим образом. Выбирают некоторую вершину графа и обходят все его вершины, достижимые из выбранной. Если в графе остались еще непройденные вершины, то выбирают одну из таких вершин и снова обходят все вершины, достижимые из выбранной (разумеется, если в процессе обхода попадется одна из уже обойденных вершин, то не только ее, но и все, достижимые из нее вершины повторно рассматривать не надо). Процесс продолжается до тех пор, пока в графе не останется ни одной непройденной вершины. Если дуги в графе не направленные, то описанный алгоритм приведет к тому, что каждый раз после выбора начальной вершины будет пройдена одна компонента связности графа. Разумеется, если граф — *связный*, то какую бы вершину ни выбрать в качестве начальной, все остальные будут из нее достижимы, так что никогда не потребуется после обхода всей компоненты связности производить еще один выбор вершины.

Наряду с обходами, при которых посещаются все вершины графа, можно также рассматривать обходы, предназначенные для прохождения всех дуг (ребер) графа. Такие обходы индуцируют некоторую нумерацию на множестве ребер подобно тому, как обходы вершин индуцируют нумерацию на множестве вершин. На практике обычно можно использовать для обходов дуг и вершин графа одни и те же алгоритмы. Действительно, большинство алгоритмов обхода графа хотя бы по одному разу обязательно исследуют как вершины, так и дуги графа. Для того чтобы понять, не находится ли на конце некоторой дуги еще непройденная вершина, требуется исследовать эту дугу. И наоборот: если проходятся все дуги, то значит, и все вершины будут пройдены, поскольку каждая вершина находится на конце некоторой дуги.

Мы будем рассматривать алгоритмы обхода, использующие представление графа, описанное в *разд. 1.5*, под названием *L*-графа, в котором для каждой вершины имеется список дуг, выходящих из этой вершины. Для алгоритмов обхода такое представление наиболее естественно, поскольку именно в этом представлении яснее видна структура графа. Имея указатель на некоторую вершину, мы легко сможем найти все смежные с ней вершины, исследуя выходящие из этой вершины дуги. В описании алгоритмов будем считать, что эти алгоритмы последовательно проходят вершины графа, но на самом деле их легко модифицировать так, чтобы нумерации подвергались не вершины, а дуги графа.

Представление в виде *L*-графа описывает ориентированный граф, в котором все дуги имеют направление. Таким образом, если некоторая дуга связывает вершину *A* с вершиной *B*, то из этого еще не следует, что из вершины *B* можно пройти в вершину *A*. Может оказаться, что даже для связного графа не обязательно удастся пройти его весь, начав обход с произвольно выбранной вершины.

Как и в случае обхода деревьев, для графов существуют два основных класса обходов: обходы "в глубину" и обходы "в ширину".

Обходы "в глубину" пытаются каждый раз после прохождения некоторой вершины *A* выбрать в качестве следующей такую вершину *B*, в которую из вершины *A* ведет дуга. Если таких непройденных вершин нет, то алгоритм возвращается к предыдущей, ранее пройденной дуге, и пытается найти очередную дугу, ведущую из нее в какую-либо другую еще не пройденную вершину. Если и таких вершин нет, то снова происходит "откат", и так до тех пор, пока либо все вершины не будут пройдены, либо алгоритм вернется назад в вершину, выбранную в качестве исходной.

Обходы "в ширину" пытаются, начав с некоторой вершины, в качестве очередных рассматривать лишь вершины, непосредственно связанные с исходной. Только после того, как все такие вершины окажутся рассмотренными, в алгоритме будет осуществлена попытка проанализировать следующий "слой" вершин, непосредственно связанных с только что пройденными. Таким образом, алгоритм обхода "в ширину" последовательно рассматривает все более удаленные от исходной вершины до тех пор, пока либо все вершины не будут пройдены, либо не останется больше вершин, достижимых из исходной.

Алгоритмы обхода графов (так же, как и деревьев, и списков) могут быть реализованы как внешними, так и внутренними итераторами. Внешний итератор берет представление графа в качестве аргумента в момент создания итератора, а затем осуществляет последовательную выдачу вершин графа (или их номеров), используя для обхода локальные объекты, такие, как стек или очередь. Внутренний итератор реализуется в виде рекурсивной функции, которая, проходя по очереди вершины графа, выполняет для каждой из них некоторое действие, заданное в качестве аргумента итератора.

Рассмотрим в качестве примера граф, изображенный на рис. 5.1.

Этот граф содержит 9 вершин и 12 дуг. Он состоит из двух компонент связности, так что заведомо не удастся обойти его весь, выбрав некоторую вершину в качестве исходной и проходя только по направлениям дуг. Тем не менее, если сначала выбрать в качестве исходной вершины вершину 1, то все остальные вершины одной компоненты связности из нее будут достижимы, поэтому, по крайней мере, эта компонента связности будет пройдена до того, как потребуется вновь выбирать начальную вершину. Тем же свойством обладают и все остальные вершины графа, кроме вершины 7. Если

выбрать ее в качестве начальной, то будет пройдена только она сама, а потом потребуется вновь выбирать некоторую вершину в качестве начальной для обхода остальных вершин.

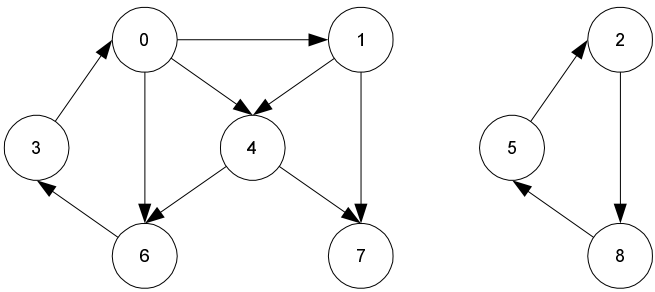


Рис. 5.1. Пример ориентированного графа

Если в качестве алгоритма обхода взять алгоритм обхода "в глубину", который всегда из всех возможных альтернатив выбирает проход к вершине с наименьшим номером, то получится следующий порядок обхода вершин графа:

0; 1; 4; 6; 3; 7; 2; 8; 5.

Порядок обхода "в ширину" в данном случае отличается от порядка обхода "в глубину" незначительно:

0; 1; 4; 6; 7; 3; 2; 8; 5.

Но это, конечно, просто потому, что граф, выбранный нами для примера, очень небольшой. Обычно в случае больших графов порядки обхода отличаются весьма существенно.

На рис. 5.1 изображена логическая структура графа. Для того чтобы яснее понимать работу алгоритмов обхода, необходимо хорошо представлять также и физическую структуру памяти объекта этого графа. В представлении в виде *L*-графа имеется 9 списков дуг (для каждой вершины существует список исходящих из нее дуг). На рис. 5.2 приведено физическое представление графа, показанного на рис. 5.1.

На рис. 5.2 изображен массив списков дуг (около каждого элемента массива показан номер соответствующей вершины графа). Дуги представлены прямоугольниками, содержащими номер той вершины, в которую дуга входит.

В качестве первого примера рассмотрим внешний итератор вершин графа при обходе "в глубину". Для этого определим класс, объекты которого получают исходный граф в виде аргумента конструктора, и затем осуществляют его обход, реализуя стандартный интерфейс итератора. Для того чтобы перейти от текущей вершины графа к следующей, итератор должен, во-первых, запоминать, какие из вершин уже пройдены, а во-вторых, для эф-

фективного возврата к предыдущей вершине помнить последовательность дуг на пути из выбранной начальной вершины в текущую. Эту последовательность дуг удобнее всего представлять в виде стека, элементами которого и будут дуги в выбранном представлении графа.

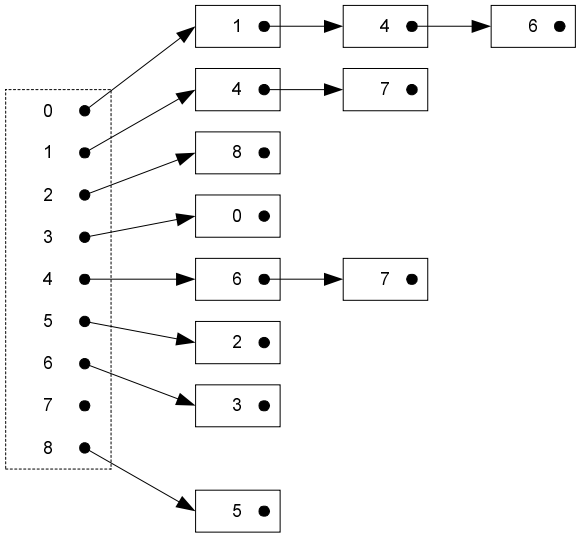


Рис. 5.2. Физическое представление графа

Пусть алгоритм выбрал в качестве начальной вершину с номером 0. Тогда обход графа начнется в ситуации, приведенной в первой строке табл. 5.1.

Таблица 5.1. *Последовательность состояний при обходе графа "в глубину"*

Шаг	Список пройденных вершин	Текущая вершина	Стек дуг на пути к текущей вершине
1	—	0	—
2	0	1	0→1
3	0, 1	4	0→1, 1→4
4	0, 1, 4	6	0→1, 1→4, 4→6
5	0, 1, 4, 6	3	0→1, 1→4, 4→6, 6→3
6	0, 1, 4, 6, 3	7	0→1, 1→4, 4→7
7	0, 1, 4, 6, 3, 7	—	—
8	0, 1, 4, 6, 3, 7	2	—
9	0, 1, 4, 6, 3, 7, 2	8	2→8

Таблица 5.1 *(окончание)*

Шаг	Список пройденных вершин	Текущая вершина	Стек дуг на пути к текущей вершине
10	0, 1, 4, 6, 3, 7, 2, 8	5	2→8, 8→5
11	0, 1, 4, 6, 3, 7, 2, 8, 5	—	—

Алгоритм обходит текущую вершину, выдавая ее в качестве очередной (`next()`), и выбирает следующую вершину из числа тех, в которые ведут дуги из текущей вершины. После первого шага ситуация будет такой, как показано в строке 2 табл. 5.1. Следующие три шага выполняются аналогично, а последовательное изменение ситуации показано в табл. 5.1 в строках 3—5.

В тот момент, когда текущей стала вершина 3, оказывается, что единственная дуга, ведущая из нее, ведет в уже пройденную вершину 0, поэтому следует, используя стек, вернуться на шаг назад и попытаться найти очередную вершину по одной из следующих дуг, ведущих из вершины 6. Поскольку новых дуг, ведущих из вершины 6, тоже больше нет, то делаем еще шаг назад к вершине 4. Теперь находится дуга, ведущая из этой вершины в еще не пройденную вершину 7, так что образуется новая ситуация, показанная в табл. 5.1 в строке 6.

После прохождения вершины 7 оказывается, что больше нет дуг, ведущих в еще не пройденные вершины ни из вершины 7, ни из всех предыдущих вершин на пути в эту вершину. В строке 7 таблицы показана ситуация, возникшая после проверки всех этих дуг. Теперь можно снова выбрать произвольно одну из непройденных вершин, скажем, вершину 2, и далее алгоритм последовательно переберет вершины 2, 8 и 5 так, как показано в строках 8—11. Теперь все вершины графа оказываются пройденными.

В листинге 5.1 приведено описание класса, реализующего граф так, как это было сделано в *разд. 1.5*, и описание класса, реализующего внешний итератор вершин графа в соответствии с указанным алгоритмом. Основное содержание алгоритма включено в реализацию функции `next()`. По сравнению с описанием, приведенным выше, в определении функции сделано небольшое изменение: вместо множества пройденных вершин во время работы алгоритма хранится множество еще не пройденных вершин. Это изменение, разумеется, ничего не меняет по существу алгоритма, но облегчает реализацию метода `hasNext()`.

Стек дуг представлен объектом `arcs` класса `Stack`, элементами этого стека будут объекты, задающие дуги в L-графе. Множество вершин представлено объектом `setNotPassed` класса `Set`, как он был определен в *разд. 1.4*. Дополнительно считаем определенным метод `first()` этого класса, выдающий первый элемент множества (если, разумеется, множество не пусто) и метод

card(), выдающий количество элементов в множестве. Элементами этого множества являются номера вершин графа. Номер текущей вершины содержится в переменной current, при этом переменная имеет значение −1, если текущая вершина не задана.

Листинг 5.1. Внешний итератор вершин графа

```
public class LGraph {
  // Класс Arc представляет дугу, ведущую в узел end
  static class Arc {
    int end;       // номер узла, в который входит эта дуга
    Arc next;      // следующая дуга в списке
    public Arc(int e, Arc n) { end = e; next = n; }
  };

  Arc[] graph;     // массив списков дуг графа

  ////////////////////////////////
  // Конструктор:
  //    n - количество вершин
  ////////////////////////////////
  public LGraph(int n) {
    // Инициализация массива списков дуг
    graph = new Arc[n];
    for (int i = 0;  i < graph.length;  i++) {
      graph[i] = null;
    }
  }

  ////////////////////////////////
  // Функция выдает количество вершин
  ////////////////////////////////
  public int vertexCount() {
    return graph.length;
  }

  ////////////////////////////////
  // Добавление дуги в граф
  ////////////////////////////////
  public void addArc(int from, int to) {
```

```
    graph[from] = new Arc(to, graph[from]);
  }
}

public class ExtGraphIterator implements Iterator {
  Set setNotPassed;       // множество непройденных вершин
  int current = -1;       // текущая вершина
  IStack arcs = StackFactory.createStack();  // стек пройденных дуг
  LGraph graph;           // граф

  ////////////////////////////////////////////////
  //  Внешний итератор:
  //    graph - граф для обхода
  ////////////////////////////////////////////////
  public ExtGraphIterator(LGraph graph) {
    this.graph = graph;
    setNotPassed = new Set(0, graph.vertexCount() - 1);
    setNotPassed.inverse();  // множество содержит все вершины
  }

  // Функция remove обеспечивает совместимость интерфейсов
  public void remove() {}    // не реализовано

  // Функция проверяет, остались ли еще непройденные вершины
  public boolean hasNext() {
    return setNotPassed.card() > 0;
  }

  // Функция, выдающая номер очередной вершины на пути обхода
  public Object next() {
    if (!hasNext()) return null;
    // 1. Если текущая вершина не определена, то она выбирается
    //    из setNotPassed
    if (current == -1) {
        current = setNotPassed.first();
    }
```

```
    // 2. Проходим текущую вершину
    Integer result = new Integer(current);
    setNotPassed.remove(current);

    // 3. Находим очередную вершину
    //    Сначала рассматриваем дуги, выходящие из текущей вершины
    LGraph.Arc nextArc = graph.graph[current];
    for (;;) {
      if (nextArc == null) {
        // Очередной дуги нет
        if (arcs.empty()) {
          // Стек дуг также пуст: компонента связности пройдена
          current = -1;
          return result;
        } else {
          // Выбираем очередную дугу из стека
          try {
            nextArc = (LGraph.Arc)arcs.pop();
          } catch (StackUnderflow x) {
          }
        }
      }
      // Дуга, ведущая в некоторую вершину, найдена
      int vertex = nextArc.end;     // это будет следующая вершина
      if (setNotPassed.has(vertex)) {
        // Это непройденная вершина; она объявляется текущей
        arcs.push(nextArc);
        current = vertex;
        return result;
      } else {
        // Переходим к следующей дуге
        nextArc = nextArc.next;
      }
    }
  }
}
```

Обход "в глубину" оказывается полезным в случае неориентированного графа (напомним, что в представлении неориентированного графа вместе с

любой дугой (A, B) обязательно содержится и дуга (B, A)). В этом случае, какую бы вершину ни взять в качестве исходной, следующими будут выбраны все достижимые из нее вершины. Таким образом, все вершины, принадлежащие одной и той же компоненте связности, получают последовательные порядковые номера.

Можно заметить, что при переходе от одной компоненты связности к другой переменная `current` итератора класса `ExtGraphIterator` получает значение −1. Поэтому легко можно добавить функцию, которая будет выдавать количество пройденных компонент связности. Для этого в классе `ExtGraphIterator` надо лишь создать счетчик компонент связности, который будет увеличиваться на единицу каждый раз, как только переменная `current` будет принимать значение −1. В листинге 5.2 показано такое расширение класса `ExtGraphIterator`, а также приведен пример применения описанного алгоритма для перечисления вершин графа по компонентам связности.

Листинг 5.2. Перечисление вершин графа по компонентам связности

```
public class ExtGraphIterator implements Iterator {
  int components = 0;    // количество пройденных компонент связности

  ////////////////////////////////////////////////
  //  Функция для подсчета пройденных компонент связности
  ////////////////////////////////////////////////
  public int getCompPassed() {
    return components;
  }

  // ... здесь не показана оставшаяся неизменной часть реализации.
  //     Реализация изменяется только в методе next()

  // Функция, выдающая номер очередной вершины на пути обхода
  public Object next() {
    if (!hasNext()) return null;
    // 1. Если текущая вершина не определена, то она выбирается
    //    из setNotPassed
    if (current == -1) {
        current = setNotPassed.first();
    }
```

```
    // 2. Проходим текущую вершину
    Integer result = new Integer(current);
    setNotPassed.remove(current);

    // 3. Находим очередную вершину
    //    Сначала рассматриваем дуги, выходящие из текущей вершины
    LGraph.Arc nextArc = graph.graph[current];
    for (;;) {
      if (nextArc == null) {
        // Очередной дуги нет
        if (arcs.empty()) {
          // Стек дуг также пуст: компонента связности пройдена
          current = -1;
          components++;
          return result;
        } else {
          // Выбираем очередную дугу из стека
          try {
            nextArc = (LGraph.Arc)arcs.pop();
          } catch (StackUnderflow x) {
          }
        }
      }
      // Дуга, ведущая в некоторую вершину, найдена
      int vertex = nextArc.end;     // это будет следующая вершина
      if (setNotPassed.has(vertex)) {
        // Это непройденная вершина; она объявляется текущей
        arcs.push(nextArc);
        current = vertex;
        return result;
      } else {
        // Переходим к следующей дуге
        nextArc = nextArc.next;
      }
    }
  }
}
```

```
public class Test {

  public static void main(String[] args) {
    LGraph graph;
    /* Здесь переменная graph получает значение */
    int componentNo = -1;
    for (ExtGraphIterator i = new ExtGraphIterator(graph);
         i.hasNext(); ) {
      int curComp = i.getCompPassed();
      if (curComp > componentNo) {
        componentNo = curComp;
        System.out.println();
        System.out.print("Вершины компоненты связности #" +
                         componentNo + ": ");
      }
      System.out.print(i.next());
      System.out.print("; ");
    }
  }
}
```

При обходе вершин (и дуг) графа часто приходится выполнять более сложную работу, чем элементарная нумерация вершин. В этой ситуации более гибким оказывается аппарат применения внутренних итераторов. При этом структура графа, как правило, вне определения класса, реализующего граф, неизвестна. Поэтому техника, при которой посетитель узла сам определяет порядок и содержание дальнейших посещений, используется редко. Более удобным и часто используемым решением оказывается определение внутреннего итератора, который обходит узлы и дуги графа по определенному им самим алгоритму, вызывая при этом различные функции при заходе в узлы, при выходе из узлов, при прохождении по дуге, и т. п.

Определим, например, обход "в глубину" в виде внутреннего итератора, который получает объект-посетитель в качестве аргумента и вызывает различные методы этого посетителя во время прохождения вершин и дуг графа. Во время обхода "в глубину" для каждой вершины графа есть два существенных момента времени. Первый — это момент, когда вершина посещается в первый раз. Именно тогда наш внешний итератор `ExtGraphIterator` выдавал номер этой вершины в качестве очередной (`next()`). Второй существенный момент — это момент окончательного покидания вершины, когда уже рассмотрены все выходящие из нее дуги. Это тот момент, когда происходит возврат по дуге, ведущей в эту вершину, а для вершины, выбранной в каче-

стве начальной, — это момент, когда заканчивается рассмотрение одной компоненты связности.

Для каждой дуги также можно выделить два основных момента: момент, когда происходит переход по этой дуге в новую или в уже посещенную ранее вершину, и момент, когда происходит возврат по этой дуге. Если дуга ведет в уже посещенную ранее вершину, то возврат происходит сразу же. Если же дуга ведет в "новую" вершину, то возврат по этой дуге произойдет только после того, как эта вершина будет покинута окончательно.

Существенным моментом при обходе в глубину является также момент выбора произвольной вершины, когда уже исследованы все пути, ведущие из ранее выбранной вершины. Для неориентированного графа это соответствует моменту перехода к новой компоненте связности графа. Соответствующий этому моменту метод также следует включить в интерфейс работы посетителя.

Итак, определим интерфейс посетителя узлов и дуг графа таким образом, чтобы в нем содержались методы для обработки вершин на "входе" и "выходе", для обработки дуг на пути "туда" и "обратно", а также для обработки момента выбора очередной вершины. Методы обработки вершин получат в качестве аргумента номер обрабатываемой вершины, а методы обработки дуг получат в качестве аргументов номера вершин, которые эта дуга соединяет, и информацию о том, посещалась ли ранее та вершина, в которую ведет эта дуга. Вот как может выглядеть определение такого интерфейса:

```
public interface IGraphDepthVisitor {
  void vertexIn(int vertex);
  void vertexOut(int vertex);
  void arcForward(int begin, int end, boolean newVertex);
  void arcBackward(int begin, int end);
  void newSelection(int vertex);
}
```

Сам обход вершин и дуг графа может быть реализован по тому же самому алгоритму, который был использован нами для реализации внешнего итератора. В листинге 5.3 такая реализация приведена в виде метода `traverseDepth()` класса `LGraph`. Несколько изменена структура объектов, помещаемых в стек дуг. В представлении L-графа дуга содержит информацию только о номере вершины, в которую она входит. Однако при прохождении дуги требуется знать также и вершину, из которой дуга исходит. Поэтому введен новый класс объектов — `ExtendedArc`, являющийся расширением класса `Arc` и содержащий в качестве расширения информацию о начальной вершине дуги.

Листинг 5.3. Обход "в глубину" с помощью внутреннего итератора

```
public class LGraph {
/* Основное определение графа остается неизменным.
 * Показано только дополнение: класс ExtendedArc и функция traverseDepth
 */

  static class ExtendedArc extends Arc {
    int begin;       // номер вершины, находящейся в начале дуги

    // Конструктор получает обычную дугу графа
    // в качестве одного из аргументов
    public ExtendedArc(int b, Arc arc) {
      super(arc.end, arc.next);
      begin = b;
    }
  };

  // Собственно функция обхода графа "в глубину"
  public void traverseDepth(IGraphDepthVisitor visitor) {
    // Множество непройденных вершин
    Set    setNotPassed = (new Set(0, graph.length-1)).inverse();
    // Стек дуг
    IStack arcs = StackFactory.createStack();
    // Текущая рассматриваемая вершина
    int    current = -1;
    // Последняя выбранная вершина
    int    selected = -1;

    while (setNotPassed.card() != 0) {
      if (current == -1) {
        selected = current = setNotPassed.first();
        visitor.newSelection(selected);
      }
      visitor.vertexIn(current);
      setNotPassed.remove(current);
      ExtendedArc nextArc =
                  (graph[current] == null ? null :
                   new ExtendedArc(current, graph[current]));
```

```
      for (;;) {
        if (nextArc == null) {
          if (arcs.empty()) {
            current = -1;
            visitor.vertexOut(selected);
            break;
          } else {
            try {
              nextArc = (ExtendedArc)arcs.pop();
            } catch (StackUnderflow x) {
            }
            visitor.vertexOut(nextArc.end);
            visitor.arcBackward(nextArc.begin, nextArc.end);
            nextArc = (nextArc.next == null ? null :
                       new ExtendedArc(nextArc.begin, nextArc.next));
          }
        } else {
          visitor.arcForward(nextArc.begin, nextArc.end,
                             setNotPassed.has(nextArc.end));
          if (setNotPassed.has(nextArc.end)) {
            current = nextArc.end;
            arcs.push(nextArc);
            break;
          } else {
            visitor.arcBackward(nextArc.begin, nextArc.end);
            nextArc = (nextArc.next == null ? null :
                       new ExtendedArc(nextArc.begin, nextArc.next));
          }
        }
      }
    }
  }
}
```

Полный "протокол" работы итератора теперь можно посмотреть, если определить посетителя, который будет протоколировать все заявленные действия. Так, например, если для приведенного на рис. 5.1 примера выполнить обход с помощью вызова:

```
graph.traverseDepth(new IGraphDepthVisitor() {
  public void vertexIn(int vertex) {
```

```
    System.out.println("vertexIn: " + vertex);
  }
  public void vertexOut(int vertex) {
    System.out.println("vertexOut: " + vertex);
  }
  public void arcForward(int begin, int end, boolean newVertex) {
    System.out.println("arcForward: beg=" + begin +
                       "; end=" + end +
                       "; new=" + newVertex);
  }
  public void arcBackward(int begin, int end) {
    System.out.println("arcBackward: beg=" + begin +
                       "; end=" + end);
  }
  public void newSelection(int vertex) {
    System.out.println("newSelection: " + vertex);
  }
});
```

то в результате работы итератора будет выведен следующий протокол:

```
newSelection: 0
vertexIn: 0
arcForward: beg=0; end=1; new=true
vertexIn: 1
arcForward: beg=1; end=4; new=true
vertexIn: 4
arcForward: beg=4; end=6; new=true
vertexIn: 6
arcForward: beg=6; end=3; new=true
vertexIn: 3
arcForward: beg=3; end=0; new=false
arcBackward: beg=3; end=0
vertexOut: 3
arcBackward: beg=6; end=3
vertexOut: 6
arcBackward: beg=4; end=6
arcForward: beg=4; end=7; new=true
vertexIn: 7
vertexOut: 7
```

```
arcBackward: beg=4; end=7
vertexOut: 4
arcBackward: beg=1; end=4
arcForward: beg=1; end=7; new=false
arcBackward: beg=1; end=7
vertexOut: 1
arcBackward: beg=0; end=1
arcForward: beg=0; end=4; new=false
arcBackward: beg=0; end=4
arcForward: beg=0; end=6; new=false
arcBackward: beg=0; end=6
vertexOut: 0
newSelection: 2
vertexIn: 2
arcForward: beg=2; end=8; new=true
vertexIn: 8
arcForward: beg=8; end=5; new=true
vertexIn: 5
arcForward: beg=5; end=2; new=false
arcBackward: beg=5; end=2
vertexOut: 5
arcBackward: beg=8; end=5
vertexOut: 8
arcBackward: beg=2; end=8
vertexOut: 2
```

Та же самая задача о переборе всех компонент связности неориентированного графа будет теперь решаться с помощью подходящего определения посетителя дуг и вершин графа. Этот посетитель должен будет при выборе очередной вершины (`newSelection`) выдавать информацию о начале новой компоненты связности, а при заходе в каждую новую вершину печатать ее номер. Ниже показан соответствующий фрагмент программы.

```
graph.traverseDepth(new IGraphDepthVisitor() {
  int componentNo = 0;
  public void vertexIn(int vertex) {
    System.out.print(vertex);
    System.out.print("; ");
  }
  public void vertexOut(int vertex) {}
```

```
  public void newSelection(int vertex) {
    System.out.println();
    System.out.println("Вершины компоненты связности # " +
                       (++componentNo) + ":");
  }
  public void arcForward(int begin, int end, boolean newVertex) {}
  public void arcBackward(int begin, int end) {}
});
```

С помощью внутреннего итератора, определенного таким образом, можно решать и более сложные задачи. Например, следующая задача известна под названием топологической сортировки вершин графа.

Пусть ориентированный граф не имеет циклов, т. е., если, следуя по направлению дуг, можно попасть из вершины *A* в вершину *B*, то из вершины *B* попасть в вершину *A* заведомо невозможно. В частности, это означает, что в графе нет петель — дуг вида (*A*, *A*). Примером такого графа может служить схема выполнения некоторого множества работ, причем элементарные работы обозначаются вершинами графа, а дуга проводится из вершины *A* в вершину *B*, если работа *b* может быть выполнена только после того, как закончится выполнение работы *a*.

Задача состоит в том, чтобы перенумеровать все вершины графа таким образом, чтобы любая дуга вела из вершины с меньшим номером в вершину с большим номером. Если граф представляет собой схему выполнения работ, то можно сказать, что такая нумерация задает возможную линейную последовательность работ, например, для последовательного выполнения всех работ одним и тем же работником.

Несложно доказать, что для ориентированного графа без циклов задача имеет по крайней мере одно решение. Одно из возможных решений можно найти с помощью следующего алгоритма.

Будем нумеровать вершины, начиная с максимального номера. Для этого запустим алгоритм обхода "в глубину", и станем приписывать очередной номер вершине, когда эта вершина окончательно покидается алгоритмом обхода (`vertexOut`). В этот момент все вершины, достижимые из покидаемой, уже будут пронумерованы и, таким образом, будут иметь номер, заведомо больший номера, приписываемого текущей покидаемой вершине. Поскольку граф не имеет циклов, то эта вершина не достижима из нее самой, поэтому все условия задачи оказываются выполненными.

Будем считать, что нумерацию вершины можно осуществить с помощью метода `setMark(int vertex, int number)`. Предположим также, что определены метод `getMark(int vertex)`, выдающий приписанный вершине номер (−1, если вершина еще не была пронумерована), и метод `isMarked(int`

vertex), проверяющий, приписан ли вершине некоторый номер. Разумеется, реализация этих методов не представляет никакого труда. Тогда решение задачи топологической сортировки вершин графа может быть записано следующим образом:

```
graph.traverseDepth(new IGraphDepthVisitor() {
  int number = graph.vertexCount();
  public void vertexIn(int vertex) {}
  public void vertexOut(int vertex) {
    graph.setMark(vertex, --number);
  }
  public void newSelection(int vertex) {}
  public void arcForward(int begin, int end, boolean newVertex) {}
  public void arcBackward(int begin, int end) {}
});
```

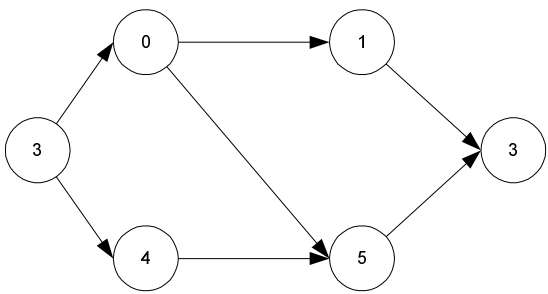


Рис. 5.3. Пример ориентированного ациклического графа

Если применить эту процедуру к графу, изображенному на рис. 5.3, то получится маркировка вершин, приведенная ниже.

```
Вершина # 0; mark=2
Вершина # 1; mark=4
Вершина # 2; mark=0
Вершина # 3; mark=5
Вершина # 4; mark=1
Вершина # 5; mark=3
```

С помощью того же самого алгоритма можно проверить, действительно ли граф не имеет циклов. Для этого достаточно при прохождении дуги (arcForward) проверять, не ведет ли эта дуга в уже пройденную, но еще не пронумерованную вершину. Реализовать такое расширение предоставляется читателям.

Внутренний итератор для обхода вершин и дуг графа "в глубину" можно реализовать и проще, чем это представлено в листинге 5.3. Для этого достаточно определить рекурсивную функцию, обходящую все вершины и дуги, достижимые из заданной вершины. Эта рекурсивная функция (`traverseRec`) и соответствующий метод обхода "в глубину" (`traverseDepthRec`) представлены в листинге 5.4.

Листинг 5.4. Рекурсивный алгоритм обхода графа "в глубину"

```
public class LGraph {
  /* Показаны только новые функции traverseRec и traverseDepthRec */
  private void traverseRec(
                  int v, IGraphDepthVisitor visitor, Set setNotPassed) {
    visitor.vertexIn(v);
    setNotPassed.remove(v);
    for (Arc arc = graph[v];  arc != null;  arc = arc.next) {
      boolean newVertex = setNotPassed.has(arc.end);
      visitor.arcForward(v, arc.end, newVertex);
      if (newVertex) {
        traverseRec(arc.end, visitor, setNotPassed);
      }
      visitor.arcBackward(v, arc.end);
    }
    visitor.vertexOut(v);
  }

  public void traverseDepthRec(IGraphDepthVisitor visitor) {
    Set setNotPassed = (new Set(0, graph.length-1)).inverse();
    while (setNotPassed.card() != 0) {
      int selected = setNotPassed.first();
      visitor.newSelection(selected);
      traverseRec(selected, visitor, setNotPassed);
    }
  }
}
```

Эта реализация выглядит более естественной еще и потому, что вызовы основных операций посетителя — `vertexIn`, `vertexOut`, `arcForward` и `arcBackward` — расположены внутри определения функции `traverseRec` симметрично на "своих" естественных местах.

Теперь рассмотрим алгоритмы обхода графа "в ширину". Для обходов "в ширину" не существует такого простого и изящного рекурсивного алгоритма, как для только что рассмотренного способа обхода "в глубину". Поэтому разберем два способа обхода графа "в ширину" — с помощью внешнего и внутреннего итераторов, которые, по существу, представляют собой модификации одного и того же алгоритма, схематично описанного ниже.

Выберем произвольную вершину графа и начнем обход с этой вершины. В процессе работы алгоритма будем делить все вершины графа на три класса:

- пройденные вершины;
- вершины, которые будут пройдены на следующем шаге алгоритма;
- все остальные непройденные вершины.

В начальный момент работы выбранная исходная вершина помещается во второе множество, а все остальные вершины — в третье. Один шаг работы алгоритма состоит в том, что берутся все вершины второго множества, проходятся (и помещаются в первое множество), находятся все вершины третьего множества, в которые ведут дуги из только что пройденных вершин, и все найденные вершины переводятся из третьего множества во второе.

Если второе множество оказывается пустым, то это означает, что пройдены уже все вершины, достижимые из исходной. Если при этом в графе еще остались непройденные вершины, то можно снова выбрать одну из них в качестве исходной и повторить работу алгоритма.

На практике второе множество можно организовать в виде очереди, тогда вершины можно брать по одной из головы очереди, а новые вершины, в которые ведут из нее дуги, сразу же помещать в конец очереди.

В листинге 5.5 представлен внешний итератор, реализующий описанный алгоритм. В нем первое множество — множество пройденных вершин — физически никак не представлено. Второе множество представлено очередью `qNext`, и третье множество — переменной `setNotPassed` класса `Set`.

Листинг 5.5. Внешний итератор для обхода графа "в ширину"

```
public class BreadthGraphIterator implements Iterator {
  LGraph graph;                                  // граф для обхода
  IQueue qNext = QueueFactory.createQueue();   // очередь вершин
  Set    setNotPassed;          // множество непройденных вершин

  public BreadthGraphIterator(LGraph g) {
    graph = g;
    setNotPassed = (new Set(0, graph.vertexCount()-1)).inverse();
  }
```

```
  public boolean hasNext() {
    // Есть еще непройденные вершины?
    return !(qNext.empty() && setNotPassed.card() == 0);
  }

  public Object next() {
    if (!hasNext()) return null;
    if (qNext.empty()) {
      // Выбираем новую исходную вершину из еще непройденных
      int selected = setNotPassed.first();
      qNext.enqueue(new Integer(selected));
      setNotPassed.remove(selected);
    }
    // Выбираем вершину из очереди
    Integer vertex = null;
    try {
      vertex = (Integer)qNext.dequeue();
    } catch (QueueUnderflow x) {
    }
    // Просматриваем дуги, ведущие из этой вершины
    for (LGraph.Arc arc = graph.graph[vertex.intValue()];
         arc != null;
         arc = arc.next) {
      if (setNotPassed.has(arc.end)) {
        setNotPassed.remove(arc.end);
        qNext.enqueue(new Integer(arc.end));
      }
    }
    return vertex;
  }

  // Функция remove() задана только для совместимости
  public void remove() {}
}
```

В листинге 5.6 тот же алгоритм приведен в виде внутреннего итератора, представленного методом `traverseBreadth` с аргументом класса `IGraphBreadthVisitor`. Поскольку при обходах "в ширину" бессмысленно говорить о возвратах по дуге, то методы `arcBackward` и `vertexOut` отсутствуют в интерфейсе `IGraphBreadthVisitor`.

Листинг 5.6. Внутренний итератор для обхода графа "в ширину"

```
public class LGraph {
  /* В листинге показана только новая функция traverseBreadth */
  public void traverseBreadth(IGraphBreadthVisitor visitor) {
    // Очередь вершин
    IQueue qNext = QueueFactory.createQueue();
    // Множество еще не пройденных вершин
    Set    setNotPassed = (new Set(0, vertexCount()-1)).inverse();

    while (!(qNext.empty() && setNotPassed.card() == 0)) {
      if (qNext.empty()) {
        // Выбор очередной исходной вершины
        int selected = setNotPassed.first();
        visitor.newSelection(selected);
        qNext.enqueue(new Integer(selected));
        setNotPassed.remove(selected);
      }
      // Выбор вершины из очереди
      Integer vertex = null;
      try {
        vertex = (Integer)qNext.dequeue();
      } catch (QueueUnderflow x) {
      }
      visitor.vertexIn(vertex.intValue());
      // Просмотр всех дуг, ведущих из этой вершины
      for (LGraph.Arc arc = graph[vertex.intValue()];
           arc != null;
           arc = arc.next) {
        boolean newVertex = setNotPassed.has(arc.end);
        visitor.arcForward(vertex.intValue(), arc.end, newVertex);
        if (newVertex) {
          setNotPassed.remove(arc.end);
          qNext.enqueue(new Integer(arc.end));
        }
      }
    }
  }
}
```

Обходы графа "в ширину" применяются в следующем разделе для поиска кратчайших путей в графе.

5.2. Поиск кратчайших путей

Самая простая задача поиска кратчайшего пути в графе состоит в следующем. Пусть заданы две вершины — начальная и конечная. Известно, что имеется по крайней мере один путь, по которому можно пройти, чтобы попасть из начальной вершины в конечную. Требуется указать один из таких путей, содержащий наименьшее количество промежуточных вершин.

Для решения данной задачи можно использовать несколько модифицированный алгоритм обхода графа "в ширину". Для этого выберем в качестве исходной вершины для обхода начальную вершину. Далее будем обходить вершины и дуги графа так, как это делалось в обычном алгоритме обхода, только для каждой проходимой вершины станем отмечать ту вершину, из которой она была достигнута по дуге (сохраняя таким образом пути, ведущие из начальной вершины во все остальные, достижимые из нее). В тот момент, когда будет достигнута конечная вершина, работу алгоритма можно заканчивать. Теперь кратчайший путь можно получить, если проследить все вершины, от конечной и до начальной по оставленным отметкам.

Приведенная в листинге 5.7 функция решает поставленную задачу, используя основу алгоритма обхода "в ширину". В качестве результата функция возвращает массив, содержащий номера всех вершин на пути из начальной вершины в конечную. Если такого пути не существует, функция вернет в качестве результата пустой массив.

Листинг 5.7. Нахождение кратчайшего пути из заданной начальной вершины в конечную

```
public class LGraph {
  /* В листинге показана только новая функция getShortestPath */
  public int[] getShortestPath(int beg, int end) {
    // Тривиальный случай совпадения начальной и конечной вершин
    // рассматривается отдельно
    if (beg == end) {
      return new int[] { beg };
    }

    // Выбираем начальную вершину в качестве исходной
    int    selected = beg;
    IQueue qNext = QueueFactory.createQueue();
```

```
Set    setNotPassed = (new Set(0, vertexCount()-1)).inverse();
int[]  marks = new int[vertexCount()];

// Инициализация массива обратных меток
for (int i=0;  i < marks.length;  i++) {
  marks[i] = -1;
}

setNotPassed.remove(selected);
qNext.enqueue(new Integer(selected));

// Основной цикл обхода "в ширину"
loop:
while (!qNext.empty()) {
  Integer vertex = null;
  try {
    vertex = (Integer)qNext.dequeue();
  } catch (QueueUnderflow x) {
  }
  for (Arc arc = graph[vertex.intValue()];
       arc != null;
       arc = arc.next) {
    boolean newVertex = setNotPassed.has(arc.end);
    if (newVertex) {
      setNotPassed.remove(arc.end);
      marks[arc.end] = vertex.intValue();
      // Здесь происходит проверка на достижение конечной вершины
      if (arc.end == end) {
        break loop;
      }
      qNext.enqueue(new Integer(arc.end));
    }
  }
}

if (marks[end] == -1) {
  // Путь в конечную вершину из начальной не найден
  return new int[] {};
```

```
    } else {
      // Подсчет количества вершин на пути в конечную вершину
      int count = 1;
      for (int i = end;  marks[i] != -1;  i = marks[i]) {
        count++;
      }
      // Формирование результирующего массива
      int[] result = new int[count];
      for (int i = end;  marks[i] != -1;  i = marks[i]) {
        result[--count] = i;
      }
      result[0] = beg;
      return result;
    }
  }
}
```

Если запустить функцию нахождения кратчайшего пути на графе, изображенном на рис. 5.1, с аргументами 0 и 3, то в результате работы функции получится массив с тремя компонентами: 0, 6, 3, а если в качестве аргументов задать 0 и 8, то в результате работы будет получен пустой массив.

Несколько более интересная задача состоит в том, чтобы найти кратчайший путь между двумя вершинами при условии, что каждая дуга имеет определенную длину, так что мерой такого расстояния будет не количество дуг на кратчайшем пути, а суммарная длина пути. Граф, в котором с каждой дугой связано некоторое значение, называется *нагруженным графом*.

Первым решение этой задачи предложил и опубликовал Э. Дейкстра (E. W. Dijkstra), поэтому описываемый ниже алгоритм называется алгоритмом Дейкстры.

Алгоритм поиска кратчайшего пути в этом случае также несколько напоминает алгоритм обхода графа "в ширину" с той разницей, что на каждом шаге во множество рассматриваемых на следующем шаге вершин (множество № 2) добавляется лишь одна вершина — та, которая находится на конце самой короткой дуги из дуг, ведущих из вершин множества № 2 в еще не пройденные вершины. Для подсчета суммарной длины пути в массив обратных ссылок записывается также не только информация о предыдущей вершине на пути в данную, но также длина дуги, по которой был сделан последний шаг. Работа с очередью также несколько отличается от классического множества операций по работе с очередью, так что в этом алгоритме уместнее говорить о массиве вершин, рассматриваемых на следующем шаге.

Реализация этого алгоритма приведена в листинге 5.8 в виде метода `getDijkstraPath`, в качестве параметров которому задаются номера начальной и конечной вершин на кратчайшем пути, а также переменная, которая будет содержать массив номеров промежуточных вершин на пути из начальной вершины в конечную. Для этого параметра специально определен новый класс `Path`, определение которого также содержится в листинге 5.8. Результатом работы метода будет суммарная длина кратчайшего пути.

Для реализации алгоритма требуется также внести некоторые коррективы в представление самого графа. Теперь вместе с каждой дугой графа в нем должна храниться его длина, поэтому определение внутреннего класса `Arc` в классе `LGraph` изменено.

Листинг 5.8. Алгоритм определения кратчайшего пути с учетом длин дуг

```
public class Path {
  public int[] path;
}

public class LGraph {
  // Класс Arc представляет дугу, ведущую в узел end
  static class Arc {
    int end;       // номер узла, в который входит эта дуга
    Arc next;      // следующая дуга в списке
    int length;    // длина дуги
    public Arc(int e, Arc n, int len) {
      end = e; next = n; length = len;
    }
  };

  // Далее определение класса остается неизменным до определения
  // нового метода getDijkstraPath

  public int getDijkstraPath(int beg, int end, Path path) {
    // Тривиальный случай совпадения начальной и конечной вершин
    // рассматривается отдельно
    if (beg == end) {
      path.path = new int[] { beg };
      return 0;
    }
```

```
// Выбираем начальную вершину в качестве исходной
int    selected = beg;
// Вектор, представляющий множество вершин,
// рассматриваемых на следующем шаге
Vector arrNext = new Vector();
Set    setNotPassed = (new Set(0, vertexCount()-1)).inverse();
int[]  marks = new int[vertexCount()];
int[]  length = new int[vertexCount()];

// Инициализация массива обратных меток
for (int i=0;  i < marks.length;  i++) {
  marks[i] = -1;
  length[i] = 0;
}

setNotPassed.remove(selected);
arrNext.add(new Integer(selected));

// Основной цикл обхода "в ширину"
while (setNotPassed.card() > 0) {
  // Поиск минимальной дуги
  Arc minArc = new Arc(0, null, Integer.MAX_VALUE);
  int minVertex = -1;
  for (int i = 0; i < arrNext.size(); i++) {
    int vertex = ((Integer)arrNext.get(i)).intValue();
    for (Arc a = graph[vertex]; a!= null; a = a.next) {
      if (setNotPassed.has(a.end) && a.length < minArc.length) {
        minArc = a;
        minVertex = vertex;
      }
    }
  }
  if (minArc.length == Integer.MAX_VALUE) {
    // Больше нет ни одной новой вершины
    break;
  }

  arrNext.add(new Integer(minArc.end));
  marks[minArc.end] = minVertex;
```

```
        length[minArc.end] = minArc.length;
        setNotPassed.remove(minArc.end);
        if (minArc.end == end) {
          // Конечная вершина достигнута!
          break;
        }
      }

      if (marks[end] == -1) {
        // Путь в конечную вершину из начальной не найден
        path.path = new int[] {};
        return 0;
      } else {
        // Подсчет количества вершин на пути в конечную вершину
        int count = 1;
        for (int i = end;  marks[i] != -1;  i = marks[i]) {
          count++;
        }
        // Формирование результирующего массива
        int wholeLength = 0;
        path.path = new int[count];
        for (int i = end;  marks[i] != -1;  i = marks[i]) {
          path.path[--count] = i;
          wholeLength += length[i];
        }
        path.path[0] = beg;
        return wholeLength;
      }
    }
}
```

Для примера нагрузим дуги графа, изображенного на рис. 5.1, некоторыми целыми числами, обозначающими длину дуги. В результате получится граф, представленный на рис. 5.4.

Если теперь попытаться найти кратчайший путь между вершинами 3 и 4, то будет найден путь, проходящий через вершины 3, 0, 1, 4. Суммарная длина этого пути составляет 6 единиц и, хотя и существует путь, проходящий через меньшее число вершин (3, 0, 4), но его суммарная длина оказывается больше (8 единиц), так что в качестве кратчайшего будет выбран именно первый путь.

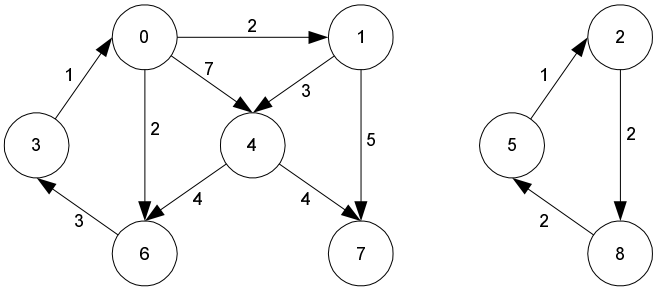


Рис. 5.4. Нагруженный граф

Вызовем для этого графа метод `getDijkstraPath` с параметрами для нахождения кратчайшего пути из вершины 3 в вершину 4:

```
Path path;
System.out.println("Кратчайший путь из вершины 3 в вершину 4 составляет "
                   + graph.getDijkstraPath(3, 4, path = new Path())
                   + " единиц и проходит через вершины:");
for (int i = 0; i < path.path.length; i++) {
  System.out.print("" + path.path[i] + "; ");
}
System.out.println();
```

Тогда в результате получится следующий текст:

```
Кратчайший путь из вершины 3 в вершину 4 составляет 6 единиц и проходит
через вершины:
3; 0; 1; 4;
```

Алгоритм Дейкстры последовательно просматривает все дуги, исходящие из расширяющегося множества вершин. Если граф имеет N вершин и M дуг, то на каждом шаге алгоритма рассматривается вплоть до M дуг, причем из них выбирается только одна с минимальной длиной. Это означает, что в худшем случае скорость работы алгоритма приближается к $M \times N$. Это, впрочем, довольно естественно, т. к. в худшем случае действительно приходится пересмотреть все дуги, выходящие из всех вершин графа.

Рассмотрим теперь задачу, в которой требуется найти всевозможные кратчайшие пути из всех вершин графа во все другие вершины. Результатом работы такого алгоритма должна быть матрица, в которой для каждой пары вершин (i, j) содержится длина минимального пути из вершины i в вершину j. Для нахождения самих путей можно также построить матрицу, в которой в клетке (i, j) содержится номер первой вершины на кратчайшем пути из вершины i в вершину j. Разумеется, можно воспользоваться алгоритмом Дейкстры, применив его последовательно для всех пар вершин, однако эффективность его работы будет невелика.

Прежде, чем перейти к рассмотрению алгоритма для решения приведенной задачи, разберем еще одну задачу, вообще говоря, более простую. Ее обобщение даст нам искомый алгоритм. Задача состоит в том, чтобы для заданного графа G найти его транзитивное замыкание, т. е. граф G^T такой, что в нем дуга ведет из вершины i в вершину j только в том случае, если в исходном графе существовал путь из вершины i в вершину j. Разумеется, исходный граф G будет содержаться в результирующем графе G^T в качестве подграфа.

Для решения этой задачи удобно воспользоваться представлением графа в виде матрицы смежности (M-граф), поскольку операции над матрицами удачно представляют собой отдельные шаги алгоритма.

Напомним, что *матрица смежности* — это матрица размером $N \times N$ элементов (где N — число вершин графа), каждый элемент которой с индексами (i, j) содержит логическое значение `true`, если в графе имеется дуга, ведущая из вершины i в вершину j, и значение `false` — в противном случае. Если ввести операции сложения и умножения для логических значений, взяв в качестве операции сложения операцию "ИЛИ" над логическими значениями, а в качестве операции умножения — операцию "И", то можно определить и операцию умножения матрицы смежности на другую матрицу смежности (используя обычные правила для умножения матриц), в результате которой получается новая матрица смежности.

Если матрица A является матрицей смежности графа G_A, а матрица B — матрицей смежности графа G_B с теми же вершинами, то матрица $A \times B$ будет представлять собой матрицу графа, в котором дуга ведет из вершины i в вершину j в том и только в том случае, когда существует вершина k такая, что в графе A существует дуга, ведущая из вершины i в вершину k, а в графе B — дуга, ведущая из вершины k в вершину j. Это позволяет построить алгоритм нахождения транзитивного замыкания графа, используя только операции умножения над матрицами смежности различных промежуточных графов.

Для исходного графа G обозначим через G^k матрицу путей длины не большей k, т. е. в этой матрице элемент с индексами (i, j) будет содержать значение `true` в том и только в том случае, если в исходном графе существовал путь длиной не больше k, ведущий из вершины i в вершину j. Заметим, что требуемый результат — матрица G^T есть не что иное, как матрица G^N в наших обозначениях, где N — число вершин графа.

Теперь заметим, что матрицу G^k нетрудно получить, имея исходную матрицу G и предыдущую матрицу G^{k-1}. Действительно, легко убедиться, что справедливо равенство

$$G^k = G^{k-1} \times G + G.$$

То есть матрицу G^k можно получить, имея предыдущую матрицу G^{k-1} и исходную матрицу G, за одно матричное умножение и одно матричное сложение. Если к тому же заметить, что исходная матрица G совпадает с матрицей

G^0, то алгоритм построения транзитивного замыкания G^T можно описать следующим образом. Сформируем последовательность матриц, начиная с матрицы $G = G^0$ и далее G^1, G^2, ... $G^N = G^T$. Поскольку каждое умножение матриц требует порядка N^3 логических операций, а логическое сложение требует порядка N^2 логических операций, то весь алгоритм займет приблизительно $N^4 + N^3$ логических операций, что при больших N примерно эквивалентно N^4 операций. Если определить операции умножения и сложения непосредственно над графами (на самом деле операции будут производиться над их матрицами смежности), то алгоритм получения транзитивного замыкания графа может выглядеть так, как представлено в листинге 5.9. В этом листинге напомним определение M-графа и введем операции для копирования, печати графа, а также сложения и умножения матриц смежности и операцию нахождения транзитивного замыкания графа.

Листинг 5.9. Алгоритм нахождения транзитивного замыкания графа

```
/*************************************************************
 * Определение M-графа и операций над ним
 *************************************************************/
public class MGraph {
  boolean graph[][]; // Внутреннее представление в виде матрицы смежности

  // Конструктор создает пустую матрицу смежности
  public MGraph(int n) {
    graph = new boolean[n][n];
  }

  // Добавление новой дуги, ведущей из u в v
  public void add(int u, int v) {
    graph[u][v] = true;
  }

  // Удаление дуги, ведущей из u в v
  public void remove(int u, int v) {
    graph[u][v] = false;
  }

  // Количество вершин в графе
  public int vertexCount() { return graph.length; }
```

```
// Создание копии графа
public MGraph copy() {
  MGraph res = new MGraph(vertexCount());
  for (int i = 0; i < vertexCount(); i++) {
    for (int j = 0; j < vertexCount(); j++) {
      res.graph[i][j] = graph[i][j];
    }
  }
  return res;
}

// Умножение матриц смежности согласно введенным операциям
// сложения и умножения логических значений
public static MGraph mult(MGraph G1, MGraph G2) throws Incompatible {
  // Прерывание Incompatible возбуждается, если в качестве операндов
  // операции умножения поданы графы разных размеров
  if (G1.vertexCount() != G2.vertexCount()) throw new Incompatible();
  int n = G1.vertexCount();
  MGraph res = new MGraph(n);
  // Классический алгоритм умножения матриц
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        if (G1.graph[i][k] && G2.graph[k][j]) {
          res.add(i, j);
          break;
        }
      }
    }
  }
  return res;
}

// Сложение матриц смежности согласно введенным операциям
// сложения логических значений
public static MGraph add(MGraph G1, MGraph G2) throws Incompatible {
  if (G1.vertexCount() != G2.vertexCount()) throw new Incompatible();
  int n = G1.vertexCount();
```

```
    MGraph res = new MGraph(n);
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        res.graph[i][j] = G1.graph[i][j] || G2.graph[i][j];
      }
    }
    return res;
  }

  // Операция нахождения транзитивного замыкания графа,
  // использующая введенные операции сложения и умножения матриц графов
  public static MGraph closure(MGraph G) {
    MGraph Gi = G.copy();
    for (int i = 0; i < G.vertexCount(); i++) {
      try {
        Gi = add(mult(Gi, G), G);
      } catch (Incompatible x) {
      }
    }
    return Gi;
  }

  // Вывод матрицы смежности графа в выходной поток
  public void print(java.io.PrintStream os) {
    for (int i = 0; i < vertexCount(); i++) {
      os.println();
      for (int j = 0; j < vertexCount(); j++) {
        os.print((graph[i][j] ? " 1" : " 0"));
      }
    }
  }
}
```

Если теперь взять в качестве исходного граф, изображенный на рис. 5.1, и вывести как исходный граф, так и его транзитивное замыкание в стандартный выходной поток, то в выходном потоке получится следующий текст.

```
Исходный граф:
 0 1 0 0 1 0 1 0 0
 0 0 0 0 1 0 0 1 0
```

```
0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0
```

```
Его транзитивное замыкание:
1 1 0 1 1 0 1 1 0
1 1 0 1 1 0 1 1 0
0 0 1 0 0 1 0 0 1
1 1 0 1 1 0 1 1 0
1 1 0 1 1 0 1 1 0
0 0 1 0 0 1 0 0 1
1 1 0 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 1
```

Описанный алгоритм нахождения транзитивного замыкания кажется очень естественным, однако его эффективность на самом деле очень невысока. Существует алгоритм, который выполняет ту же работу за гораздо меньшее число операций, к тому же не расходуя столько памяти под большое количество промежуточных матриц.

Идея этого алгоритма, называемого алгоритмом Флойда—Уоршолла, состоит в следующем. Будем также рассматривать последовательность матриц смежности, только теперь матрица G^k будет обозначать матрицу, в которой элемент с индексами (i, j) станет содержать значение **true** в том и только в том случае, когда в исходном графе существует путь из вершины i в вершину j, проходящий через вершины с номерами из множества $\{0, 1, ..., k-1\}$. Очевидно, что сам исходный граф G в этом случае совпадает с матрицей G^0, а искомая матрица G^T совпадает с матрицей G^N.

Рассмотрим теперь, каким образом можно вычислить значение элементов матрицы G^k с индексами (i, j). Ясно, что если в предыдущей матрице G^{k-1} значение этого элемента было **true**, то в матрице G^k значение элемента также будет **true**. Если же значение этого элемента было равно **false**, то это означает, что в исходном графе не было пути из вершины i в вершину j, проходящего через промежуточные вершины из множества $\{0, 1, ..., k-2\}$. В этом случае все же может существовать путь, проходящий через промежуточные вершины из множества $\{0, 1, ..., k-1\}$ при условии, что существовал путь из вершины i в вершину $k-1$, проходящий через промежуточные вер-

шины из множества {0, 1, ..., $k-2$}, и путь из вершины $k-1$ в вершину j, проходящий через промежуточные вершины из множества {0, 1, ..., $k-2$}. Отсюда сразу же получается следующая формула:

$$G^k(i,j) = \begin{cases} true, & \text{если } G^{k-1}(i,j) = true; \\ G^{k-1}(i,k-1) \,\&\, G^{k-1}(k-1,j), & \text{если } G^{k-1}(i,j) = false. \end{cases}$$

Из этой формулы видно, что во-первых, значения элементов i-го ряда зависят только от элементов этого же и $(k-1)$-го рядов, а во-вторых, что значения в $(k-1)$-м ряду остаются неизменными. Также не меняется и значение элемента G^k с индексами $(i, k-1)$, поскольку элементы $G^{k-1}(k-1, k-1)$ всегда равны **true**. Это означает, что элементы новой матрицы можно вычислять прямо на месте, не отводя дополнительного места под новую матрицу. Отсюда сразу же следует алгоритм для вычисления транзитивного замыкания графа, показанный в листинге 5.10 в виде метода `closure1` класса `MGraph`.

Листинг 5.10. Более эффективный алгоритм транзитивного замыкания графа

```
public class MGraph {
  boolean graph[][]; // внутреннее представление в виде матрицы смежности
  ...
  public static MGraph closure1(MGraph G) {
    MGraph G1 = G.copy();
    for (int k = 1; k < G.vertexCount(); k++) {
      for (int i = 0;  i < G.vertexCount();  i++) {
        for (int j = 0;  j < G.vertexCount();  j++) {
          if (i != k-1 && !G1.graph[i][j]) {
            G1.graph[i][j] = G1.graph[i][k-1] && G1.graph[k-1][j];
          }
        }
      }
    }
    return G1;
  }
}
```

Этот алгоритм, разумеется, выдает тот же самый результат, что и предложенный ранее алгоритм, реализованный с помощью метода `closure`. Однако работает он при этом на порядок эффективнее, причем (редкий случай!) и по ресурсам памяти, и по затраченному времени. Этот же алгоритм можно положить и в основу алгоритма вычисления всех кратчайших путей в графе между всеми парами вершин. Действительно, кратчайший путь между вер-

шинами i и j может либо проходить через вершину с номером $k-1$, либо нет. В первом случае длина кратчайшего пути равна сумме длин кратчайших путей от вершины i до вершины $k-1$ и от вершины $k-1$ до вершины j. Во втором случае кратчайший путь совпадает с длиной кратчайшего пути, не содержащего вершину $k-1$ в качестве промежуточной. Таким образом, мы можем получать матрицу длин кратчайших путей по тому же самому алгоритму, только в качестве начальной матрицы берется модифицированная матрица расстояний между вершинами в графе. В этой модифицированной матрице элемент с индексами (i, j) равен $+\infty$, если в графе нет ребра, ведущего из вершины i в вершину j. В противном случае элемент содержит длину ребра (расстояние) от вершины i к вершине j. Имея эту начальную матрицу в качестве матрицы G^0, будем получать далее последовательность матриц G^1, G^2, ..., G^N, используя формулу:

$$G^k(i, j) = min(G^{k-1}(i, j), G^{k-1}(i, k-1) + G^{k-1}(k-1, j)).$$

Кроме матрицы длин кратчайших путей надо формировать еще и матрицу направлений. Ее можно получать сходным способом. В этой матрице элемент $D^k(i, j)$ равен номеру начальной промежуточной вершины на кратчайшем пути из вершины i в вершину j, проходящему через промежуточные вершины из множества $\{0, 1, ..., k-1\}$. Ясно, что начальная матрица направлений D^0 совпадает с исходной матрицей смежности графа, в которой элемент с индексами (i, j) имеет значение j, если имеется ребро, ведущее из вершины i в вершину j, и содержит -1, если такого ребра нет. В дальнейшем, при вычислении матрицы кратчайших путей, как только обнаруживается, что элемент $G^k(i, j)$ следует заменить на сумму $G^{k-1}(i, k-1) + G^{k-1}(k-1, j)$, элемент $D^k(i, j)$ меняется на $D^{k-1}(i, k-1)$. Таким образом, в конце концов в матрице D^N будут содержаться направления по всем кратчайшим путям между всеми парами вершин в исходном графе. Итак, имеем алгоритм, реализованный в виде метода `getMaxPaths` в листинге 5.11. Этот метод получает в качестве аргументов исходный граф G и две целочисленные квадратные матрицы, в которые в процессе работы алгоритма и будут записаны значения элементов матрицы кратчайших путей и матрицы направлений. Разумеется, поскольку граф нагруженный, то его определение несколько модифицировано. Теперь помимо собственно матрицы смежности `graph` в определении графа существует еще и матрица нагрузок `W`, содержащая в своих элементах `W[i][j]` нагрузки на дугу (i, j). Соответственно изменились и некоторые методы, которые также показаны в листинге.

Листинг 5.11. Алгоритм нахождения матрицы кратчайших путей и матрицы направлений

```
public class MGraph {
  boolean graph[][];
  int W[][];
```

```
public MGraph(int n) {
  graph = new boolean[n][n];
  W = new int[n][n];
}

public void add(int u, int v, int w) {
  graph[u][v] = true;
  W[u][v] = w;
}

public void remove(int u, int v) {
  graph[u][v] = false;
  W[u][v] = 0;
}

public MGraph copy() {
  MGraph res = new MGraph(vertexCount());
  for (int i = 0; i < vertexCount(); i++) {
    for (int j = 0; j < vertexCount(); j++) {
      res.graph[i][j] = graph[i][j];
      res.W[i][j] = W[i][j];
    }
  }
  return res;
}

public static void getMinPaths(MGraph G, int paths[][], int dirs[][])
       throws IndexOutOfBoundsException {
  if (paths.length != G.vertexCount()
      || paths[0].length != G.vertexCount()
      || dirs.length != G.vertexCount()
      || dirs[0].length != G.vertexCount()) {
        throw new IndexOutOfBoundsException("Wrong matrix sizes");
  }
  // Инициализация матриц
  for (int i=0; i<G.vertexCount(); i++) {
    for (int j=0; j< G.vertexCount(); j++) {
      paths[i][j] = (G.graph[i][j] ? G.W[i][j] : Integer.MAX_VALUE);
```

```
        dirs[i][j] = (G.graph[i][j] ? j : -1);
      }
    }

    // Вычисление кратчайших путей и направлений
    for (int k = 1; k < G.vertexCount(); k++) {
      for (int i = 0;  i < G.vertexCount();  i++) {
        for (int j = 0;  j < G.vertexCount();  j++) {
          if (i != k-1 &&
              paths[i][k-1] < Integer.MAX_VALUE &&
              paths[k-1][j] < Integer.MAX_VALUE &&
              paths[i][j] > paths[i][k-1] + paths[k-1][j]) {
            paths[i][j] = paths[i][k-1] + paths[k-1][j];
            dirs[i][j] = dirs[i][k-1];
          }
        }
      }
    }
  }
}
```

Если применить этот алгоритм к графу, изображенному на рис. 5.4, то получится следующий результат:

```
Кратчайшие пути:
  6  2  -  5  5  -  2  7  -
 11 13  - 10  3  -  7  5  -
  -  -  -  -  -  -  -  -  2
  1  3  -  6  6  -  3  8  -
  8 10  -  7 13  -  4  4  -
  -  -  1  -  -  -  -  -  3
  4  6  -  3  9  -  6 11  -
  -  -  -  -  -  -  -  -  -
  -  -  3  -  -  2  -  -  5


Направления:
 6 1 - 6 1 - 6 1 -
 4 4 - 4 4 - 4 7 -
 - - - - - - - - 8
 0 0 - 0 0 - 0 0 -
```

```
6 6 - 6 6 - 6 7 -
- - 2 - - - - - 2
3 3 - 3 3 - 3 3 -
- - - - - - - - -
- - 5 - - 5 - - 5
```

Имея эти результаты, можно установить, что, например, кратчайший путь из вершины 2 в вершину 6 не существует, а кратчайший путь из вершины 3 в вершину 7 содержит 8 единиц и проходит последовательно через вершины 3, 0, 1, 7.

Имеется еще много интересных алгоритмов, связанных с поиском минимальных характеристик путей в графе, однако будем двигаться дальше и рассмотрим несколько алгоритмов, связанных с поиском минимальных подграфов заданного графа.

5.3. Определение остовных деревьев

Остовным деревом (скелетом) неориентированного графа называется его подграф, не имеющий циклов и содержащий все вершины исходного графа. Так, например, для нагруженного графа, изображенного на рис. 5.5, *а*, скелетами являются все его подграфы, изображенные на рис. 5.5, *б*.

Эти различные поддеревья исходного графа имеют различный суммарный вес ребер графа. В первом варианте — сумма весов скелета равна 15, во втором варианте — 27, в третьем варианте — 19. В разных ситуациях бывает необходимо найти остовное дерево с максимальным или минимальным общим весом ребер. Так первое поддерево имеет минимальный суммарный вес из всех возможных поддеревьев. Второе поддерево — максимальный. Поставим задачу найти минимальное остовное дерево (то есть остовное дерево минимального суммарного веса ребер) для заданного неориентированного графа. Разумеется, поиск максимального остовного дерева будет производиться точно по такому же алгоритму.

Первый из предлагаемых алгоритмов принадлежит классу так называемых "жадных" алгоритмов. "Жадные" алгоритмы пытаются найти решение поставленной задачи "в лоб", стараясь сразу же найти минимальное остовное дерево, выбирая для решения ребра с минимальным весом. Разумеется, для этого необходимо сначала отсортировать все ребра графа по возрастанию их весов. Лучше всего, если граф с самого начала представлен списком ребер, в этом случае сортировка будет наиболее естественной. Но даже и в других случаях сортировка ребер графа по весу занимает не слишком много времени — в худшем случае $M \times \log_2 M$, где M — количество ребер графа. После этого "жадный" алгоритм выбирает ребра по очереди, начиная с ребер с наименьшим весом, и пытается включить каждое очередное ребро в строящееся дерево. Очередное ребро не удается включить в дерево, если оно за-

мыкает какой-либо цикл в графе, а это можно проверить, если удастся выяснить, принадлежат ли оба конца очередного ребра уже построенной части остовного дерева. Отсюда возникает алгоритм, называемый также алгоритмом Крускала (J. B. Kruskal) [7].

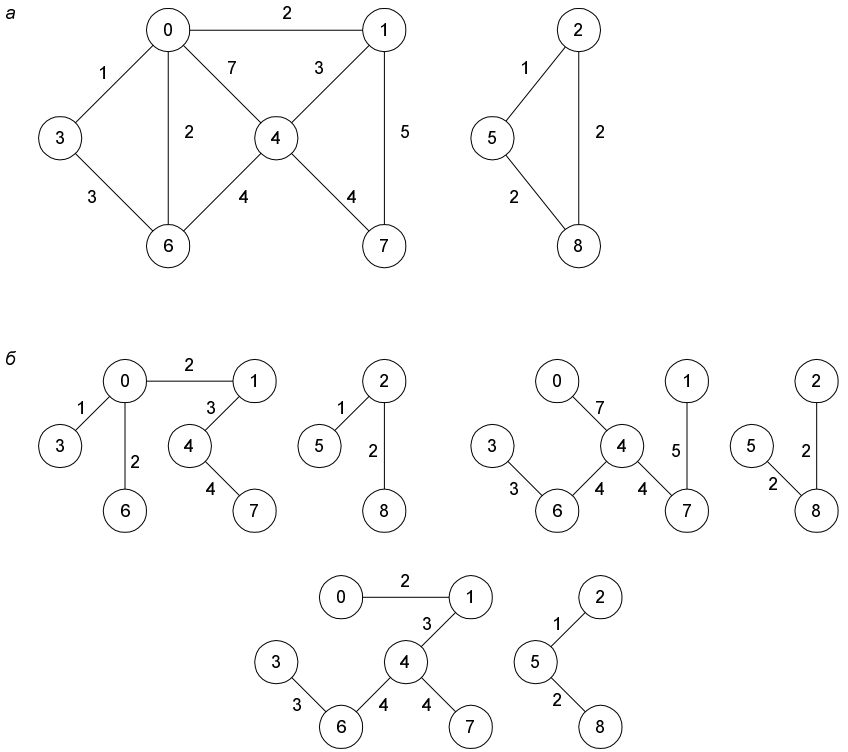


Рис. 5.5. Нагруженный неориентированный граф и его остовные поддеревья: *а* — нагруженный неориентированный граф; *б* — несколько вариантов остовных деревьев графа

После сортировки ребер по весу организуется строящееся дерево в виде несвязных фрагментов этого дерева. Каждое вновь включаемое ребро добавляет одну вершину в уже существующую компоненту (если другая вершина уже принадлежит этой компоненте), либо соединяет две компоненты в одну (если обе вершины принадлежат разным компонентам). Если обе вершины ребра принадлежат одной и той же компоненте, то такое ребро не включается в строящееся остовное дерево и игнорируется алгоритмом. Читателям в качестве упражнения предлагается доказать, что описанный алгоритм действительно строит минимальное остовное дерево заданного графа.

Для хранения информации о компонентах строящегося дерева предлагается структура данных в виде массива, каждый *i*-й элемент которого содержит

ссылку на одну из вершин компоненты, содержащей *i*-ю вершину. Фактически такой массив является одним из способов представления дерева компонент (точнее, леса из нескольких деревьев), в котором ссылки из узлов дерева проводятся от потомков к предкам.

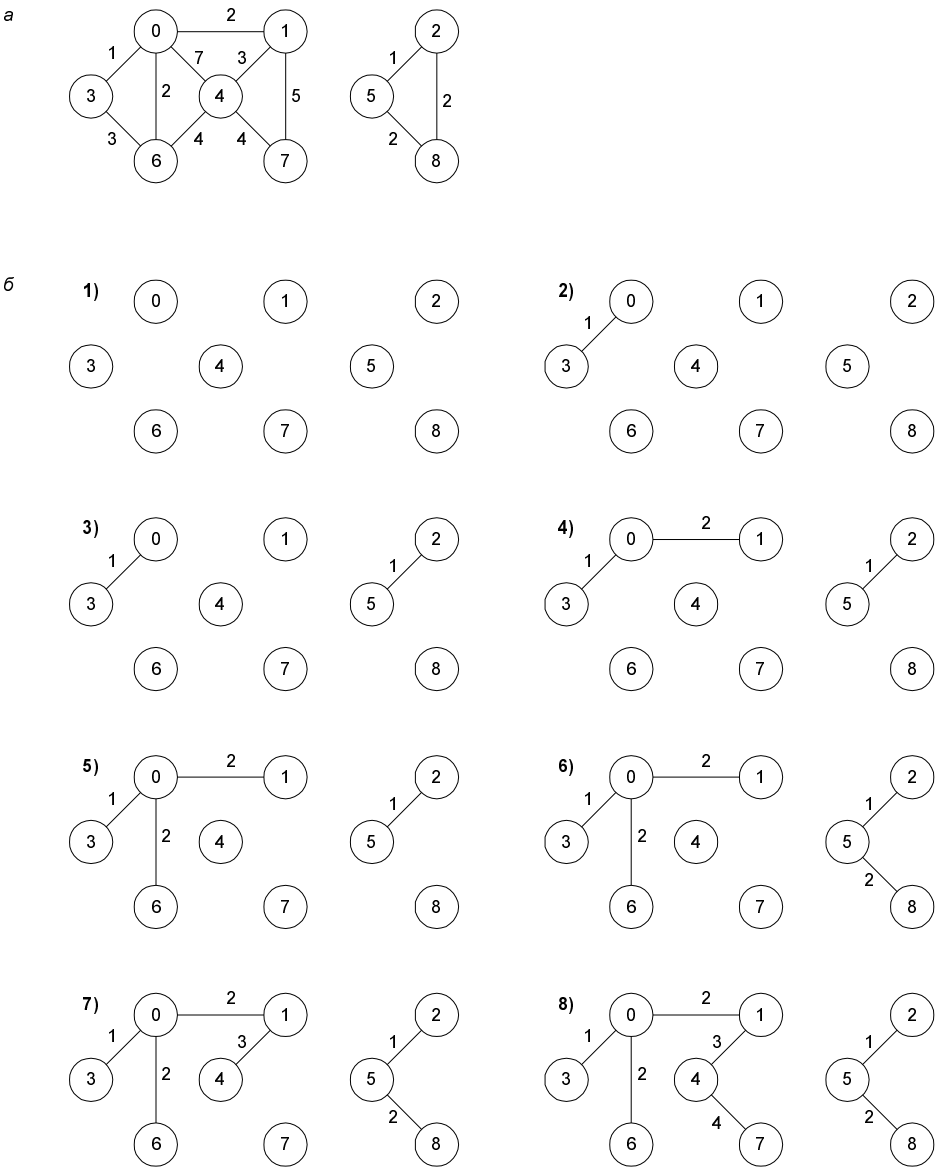


Рис. 5.6. "Жадный" алгоритм построения минимального остовного дерева графа: *а* — исходный граф; *б* — 8 этапов построения минимального остовного дерева с помощью "жадного" алгоритма

На рис. 5.6 представлена последовательность этапов при построении компонент остовного дерева согласно "жадному" алгоритму. В качестве исходного выбран граф рис. 5.5, *а*; первоначально остовное дерево состоит из отдельных вершин исходного графа. "Жадный" алгоритм постепенно добавляет к нему ребра, начиная с ребер с наименьшим весом.

На рис. 5.6, *а* показан исходный граф, а на рис. 5.6, *б* — 8 последовательных этапов построения остовного дерева согласно "жадному" алгоритму. На каждом этапе в остовное дерево включается новое ребро. Ребра, замыкающие цикл в уже построенной части, игнорируются алгоритмом.

В листинге 5.12 показана реализация "жадного" алгоритма построения минимального остовного дерева. Метод `minSkeleton` графа получает в качестве аргумента выходной поток для печати информации о ребрах графа, включаемых в остовное дерево. Информация выводится в выходной поток как последовательность пар целых чисел из номеров вершин. Эти пары задают ребра графа.

На первом этапе работы алгоритма последовательность ребер сортируется и заносится в массив ребер, содержащий пары вершин (ребра) и их вес. Информация о ребрах графа берется из матрицы смежности графа и матрицы весов ребер `W`.

На втором этапе начинается построение остовного дерева, информация о котором хранится в массиве. Каждый раз, как только очередное ребро попадает в остовное дерево, информация о нем выводится в выходной поток.

Листинг 5.12. Реализация "жадного" алгоритма построения минимального остовного дерева

```
public class MGraph {
  boolean graph[][];       // матрица смежности
  int W[][];               // матрица нагрузки

  public void add(int u, int v, int w) {
    // Добавление нагруженного ребра
    graph[u][v] = true;
    graph[v][u] = true;
    W[u][v] = w;
    W[v][u] = w;
  }

  /*********************************************************************
   * Реализация ребра для хранения его в упорядоченном множестве ребер.
   * Ребра считаются равными, если у них совпадают начала и концы.
   * Ребра упорядочиваются по нагрузке.
   *********************************************************************/
```

```
public static class Edge implements Comparable {
  int beg;          // начальная вершина
  int end;          // конечная вершина
  int weight;       // вес (нагрузка)
  // Конструктор:
  public Edge(int b, int e, int w) {
    beg = b;
    end = e;
    weight = w;
  }
  // Сравнение на равенство
  public boolean equals(Object o) {
    return equals((Edge)o);
  }
  public boolean equals(Edge e) {
    return beg == e.beg && end == e.end;
  }
  // Сравнение по порядку. Равенство включено как частный случай
  public int compareTo(Object o) {
    return compareTo((Edge)o);
  }
  public int compareTo(Edge e) {
    return equals(e) ? 0 : weight - e.weight;
  }
};

// Проверка ребра по дереву компонент
public boolean sameComponent(int a, int b, int[] tree) {
  int root1 = a;            // корень первой компоненты
  while (tree[root1] != -1) { root1 = tree[root1]; }
  int root2 = b;            // корень второй компоненты
  while (tree[root2] != -1) { root2 = tree[root2]; }
  return root1 == root2;    // корни совпадают?
}

// Основная функция вычисления минимального остовного дерева графа
public int minSkeleton(java.io.PrintStream ps) {
  SortedSet edges = new TreeSet();        // упорядоченный массив ребер
```

```
    int []    tree = new int [vertexCount()];  // дерево компонент
    int       sumWeight = 0;                   // суммарный вес

    // 1. Построение упорядоченного массива ребер
    for (int i = 1;  i < vertexCount();  i++) {
      for (int j = 0;  j < i;  j++) {
        if (graph[i][j]) {     // имеется ребро (i, j)
          edges.add(new Edge(i, j, W[i][j]));
        }
      }
    }

    // 2. Инициализация дерева компонент
    for (int i = 0;  i < vertexCount();  i++) {
      tree[i] = -1;
    }

    // 3. Собственно "жадный" алгоритм
    while (!edges.isEmpty()) {
      Edge minEdge = (Edge)edges.first();
      edges.remove(minEdge);
      if (!sameComponent(minEdge.beg, minEdge.end, tree)) {
        // а) включить в дерево компонент
        int root = minEdge.beg;
        while (tree[root] != -1) { root = tree[root]; }
        tree[root] = minEdge.end;
        // б) вывести в выходной поток
        ps.println(" " + minEdge.beg + " " + minEdge.end);
        sumWeight += minEdge.weight;
      }
    }

    return sumWeight;
  }
}
```

При попытке вычислить минимальное остовное дерево для графа, приведенного на рис. 5.5, с помощью вызова метода

```
System.out.println(graph.minSkeleton(System.out));
```

в стандартный системный поток будет выведена следующая информация:

```
 5 2
 3 0
 8 5
 6 0
 1 0
 4 1
 7 4
15
```

свидетельствующая о том, что минимальное остовное дерево построено правильно, поскольку содержит ребра (5, 2), (3, 0), (8, 5), (6, 0), (1, 0), (4, 1), (7, 4) и суммарный вес всех ребер составляет минимально возможную величину 15.

Скорость работы "жадного" алгоритма определяется скоростью построения упорядоченного списка ребер, т. к. вся остальная работа занимает меньше времени. Тем не менее, для больших графов построение и проверка дерева компонент может занимать значительное время, если дерево вершин превращается в линейный список вершин, что возможно в некоторых вырожденных ситуациях. Это снижает эффективность работы алгоритма. К счастью, ситуацию нетрудно исправить. Для этого надо лишь в процессе просмотра дерева компонент устраивать его локальное перестроение, продвигая проходимые в процессе работы вершины ближе к корню дерева. Модификацию алгоритма в соответствии с этим замечанием предоставляется выполнить читателям.

Еще один алгоритм построения минимального остовного дерева напоминает алгоритм Дейкстры для поиска наименьшего пути между двумя вершинами в графе и носит название алгоритма Прима (R. C. Prim).

В этом алгоритме построение остовного дерева начинается с одной вершины, к которой затем добавляются ребра таким образом, чтобы каждое новое ребро одним своим концом опиралось в уже построенную часть дерева, а другой конец лежал бы в множестве еще неприсоединенных к дереву вершин. Из всех таких ребер на каждом шаге выбирается ребро с наименьшим весом. Для того чтобы выбор ребра был наиболее эффективен, в алгоритме используется промежуточная структура данных — массив, элементы которого содержат информацию о "расстоянии" от всех вершин до уже построенной части остовного дерева. Таким образом, с помощью однократного просмотра такого массива всегда можно выбрать ребро минимальной длины. Вершины, не соединенные ребрами с уже построенной частью остовного дерева, находятся от этой части на расстоянии $+\infty$. Если очередная выбранная вершина находится на расстоянии $+\infty$ от уже построенной части дерева, то это означает, что завершено построение дерева для одной компоненты связности графа, так что новая выбранная вершина будет служить ядром дерева для нового компоненты связности графа.

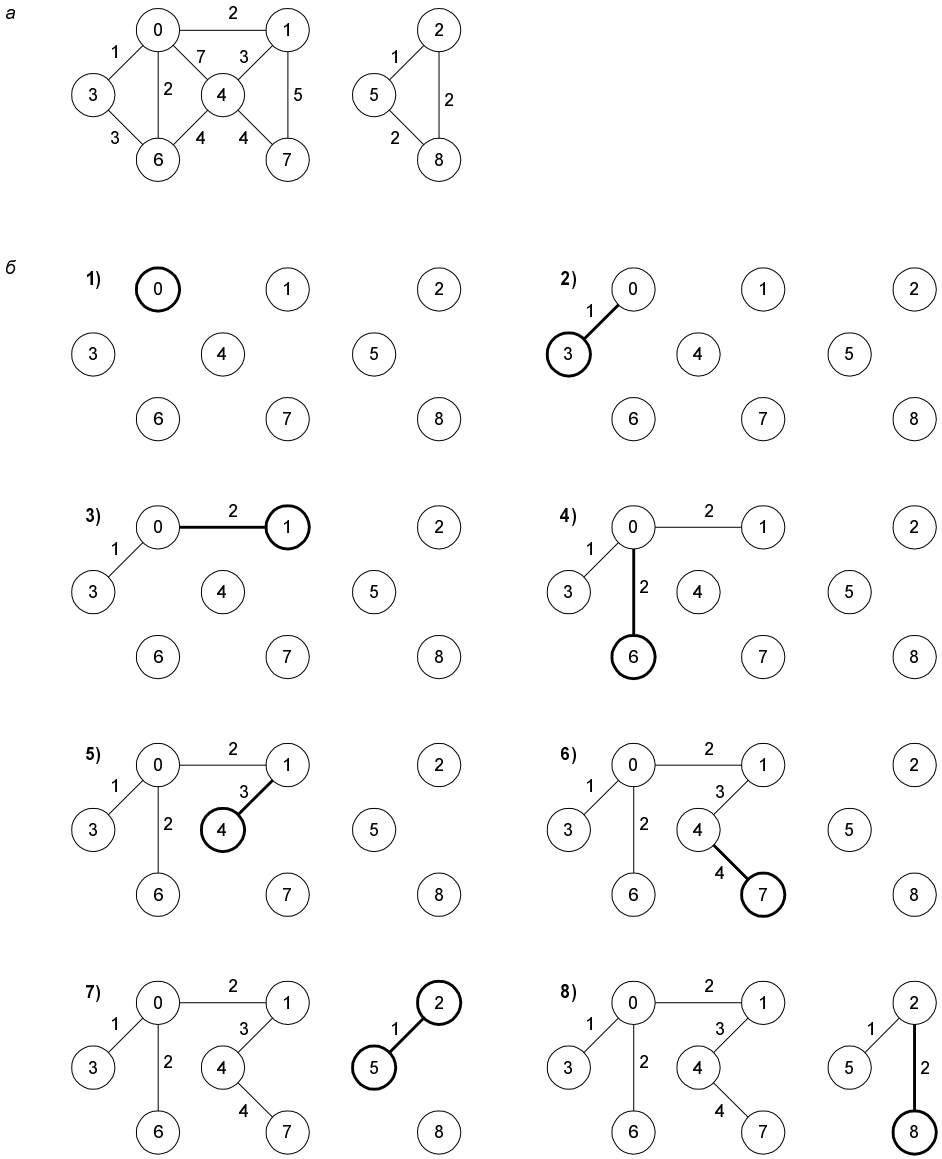


Рис. 5.7. Этапы построения остовного дерева согласно алгоритму Прима: *а* — исходный граф; *б* — 8 этапов построения минимального остовного дерева с помощью алгоритма Прима

Если для графа, изображенного на рис. 5.5, начать поиск минимального остовного дерева с вершины 0, то к дереву будут последовательно присоеди-

няться ребра (0, 3) длиной 1, (0, 1) длиной 2, (0, 6) длиной 2, (1, 4) длиной 3, (4, 7) длиной 4, вершина 2, (2, 5) длиной 1, (2, 8) длиной 2.
На рис. 5.7 показана последовательность построения минимального остовного дерева для графа, изображенного на рис. 5.5

Реализация алгоритма Прима приведена в листинге 5.13 в виде определения метода `minSkeletonPrim`, который так же, как и метод `minSkeleton`, в качестве аргумента получает выходной поток для печати информации о найденных ребрах минимального остовного дерева, а в качестве результата выдает суммарный вес полученного остовного дерева.

Листинг 5.13. Алгоритм Прима нахождения минимального остовного дерева

```
public class MGraph {
  /* ... Показан только метод minSkeletonPrim ... */
  public int minSkeletonPrim(java.io.PrintStream ps) {
    // Множество непройденных вершин (сначала - все вершины)
    Set notPassed = (new Set(0, vertexCount() - 1)).inverse();
    // Массив расстояний от вершин до уже построенной части
    int [] distances = new int [vertexCount()];
    // Массив направлений от новых вершин к уже построенной части
    int [] ends = new int [vertexCount()];
    // Суммарный вес построенной части дерева
    int sumWeight = 0;

    // Инициализация массива расстояний
    for (int i = 0; i < distances.length; i++) {
      distances[i] = Integer.MAX_VALUE;
      ends[i] = -1;
    }

    // Основной цикл поиска новых вершин
    while (notPassed.card() != 0) {
      // Поиск ближайшей вершины
      int minVertex = -1;
      int minDistance = Integer.MAX_VALUE;
      for (int nextVertex = 0; nextVertex<vertexCount(); nextVertex++) {
        if (notPassed.has(nextVertex) &&
            distances[nextVertex] < minDistance) {
          minDistance = distances[nextVertex];
          minVertex = nextVertex;
        }
```

```
      }
      if (minVertex == -1) {
        // Начинаем новую компоненту связности
        minVertex = notPassed.first();
      } else {
        // Присоединяем очередное ребро
        ps.println(" " + ends[minVertex] + " " + minVertex);
        sumWeight += minDistance;
      }
      notPassed.removeElem(minVertex);
      // Новая вершина присоединена;
      // корректируем информацию о расстояниях
      for (int u = 0;  u < vertexCount();  u++) {
        if (notPassed.has(u) && graph[minVertex][u] &&
            W[minVertex][u] < distances[u]) {
              distances[u] = W[minVertex][u];
              ends[u] = minVertex;
        }
      }
    }
    return sumWeight;
  }
}
```

Если применить метод `minSkeletonPrim` к графу, изображенному на рис. 5.5

```
System.out.println(graph.minSkeletonPrim(System.out));
```

то, разумеется, результат будет примерно тем же самым, что и после применения метода `minSkeleton`, реализующего "жадный" алгоритм:

```
 0 3
 0 1
 0 6
 1 4
 4 7
 2 5
 2 8
15
```

Следует, однако, заметить, что в нашей реализации даже на таком небольшом графе алгоритм Прима показывает заметно лучшую производительность (приблизительно в два раза), чем алгоритм Крускала.

Глава 6

Технология обмена сообщениями

При объектно-ориентированном подходе к проектированию и программированию часто бывает удобно рассматривать программу как набор независимых объектов, обменивающихся сообщениями. Особенно удобно это для организации среды взаимодействия с пользователем, где источниками сообщений могут служить внешние устройства, такие как мышь или клавиатура, однако подобная организация программы может оказаться удобной и в других ситуациях. Для того чтобы объекты могли обмениваться сообщениями, требуется организовать специальный объект — *диспетчер сообщений*, который будет контролировать работу всей системы и передавать сообщения от одних объектов к другим. Именно таким образом построены многие современные операционные системы, например, MS Windows, в которой одни объекты (окна) посылают сообщения другим окнам, используя для этого системные вызовы, т. е. в роли диспетчера сообщений выступает сама операционная система. В этой главе мы рассмотрим организацию собственного диспетчера сообщений и выполним с его помощью вычисление различных конечных сумм, правда, несколько необычным способом.

6.1. Схема обмена сообщениями

Обычно сообщение, передаваемое от одного объекта другим, имеет следующие характеристики:

- *код класса сообщения*, отличающий сообщения одного класса от сообщений другого класса;
- *адрес объекта*, для которого это сообщение предназначено; адрес может быть не задан, если сообщение предназначено всем, кому оно "интересно";
- *объект-параметр сообщения*, содержащий информационную нагрузку сообщения.

Для того чтобы сообщение могло быть передано по назначению, оно отправляется диспетчеру сообщений, который затем пытается передать это

сообщение одному из объектов-обработчиков сообщения. Схема работы диспетчера сообщений может быть следующей.

1. Каждый объект, желающий генерировать и передавать сообщения другим объектам, регистрируется у диспетчера в качестве производителя сообщений.
2. Каждый объект, желающий получать и обрабатывать сообщения от других объектов, регистрируется у диспетчера в качестве обработчика сообщений. Разумеется, один и тот же объект может выступать и в роли производителя и в роли обработчика сообщений.
3. Любой объект, даже тот, который не зарегистрирован у диспетчера, может создать сообщение и передать его диспетчеру. Для таких сообщений диспетчер ведет "очередь сообщений".
4. Диспетчер в цикле пытается обнаружить одно из сообщений, предназначенных для передачи объектам-обработчикам. Для этого он сначала пытается выбрать очередное сообщение из очереди сообщений, а если эта очередь пуста, то опрашивает зарегистрированные у него генераторы сообщений.
5. Если сообщение найдено, то диспетчер просматривает список зарегистрированных обработчиков сообщений и пытается передать это сообщение адресату, а в случае, если сообщение предназначено "всем, кому интересно", пробует передать его всем по очереди обработчикам, каждый из которых может после обработки сообщения известить диспетчера, следует ли ему пытаться передавать это сообщение другим объектам.

При такой схеме вся работа программы контролируется диспетчером, и реальная деятельность происходит только в результате исполнения реакций на сообщения. Например, при работе в операционной системе MS Windows стиль написания программ состоит в том, что определяется набор объектов-окон, реагирующих на различные сообщения. Роль диспетчера сообщений исполняет сама система, при этом в качестве генераторов сообщений выступают драйверы клавиатуры, мыши и других внешних устройств, а все приложения определяют набор объектов-окон, регистрирующихся в системе в роли обработчиков сообщений. Сообщения посылаются с помощью стандартной системной функции `sendMessage`, при этом каждое сообщение имеет код класса сообщения, указатель адресата сообщения и два информационных параметра длиной в простое и двойное слово, содержимое которых определяется классом сообщения и тем, как сообщение интерпретируется обработчиком.

Обычно один из классов сообщений выделяется как сообщение диспетчеру об окончании работы. Так, например, в системе MS Windows нажатие комбинации клавиш <Ctrl>+<Alt>+<Del> приводит к генерации специального сообщения, которое обрабатывает сама система.

В этом разделе мы определим и запрограммируем простой диспетчер сообщений, который затем используем в следующем разделе для задачи вычисления конечных сумм зависящих друг от друга элементов.

Прежде всего выясним структуру и методы работы с сообщениями. Для этого определим класс `Message` и методы для формирования и анализа сообщений. Затем создадим интерфейсы, идентифицирующие объекты как генератор или обработчик сообщения. Наконец, определим класс `Dispatcher`, содержащий методы для регистрации объектов в роли генераторов и обработчиков сообщений, для постановки сообщения в очередь сообщений диспетчера, а также опишем основной цикл работы диспетчера в виде статической функции `run` диспетчера.

Основная функция работы генератора сообщений состоит в том, что объект-генератор способен создать объект-сообщение в ответ на запрос диспетчера. Поэтому интерфейс, определяющий работу генератора сообщений, будет выглядеть следующим образом:

```
public interface IGenerator {
  Message generate();
}
```

Аналогично, обработчик сообщений должен "уметь" обработать сообщение, переданное ему диспетчером, и ответить, следует ли передавать это сообщение другим обработчикам. Поэтому определим интерфейс обработчика сообщения так:

```
public interface IHandler {
  boolean handle(Message msg);
}
```

Договоримся, что метод `handle` будет выдавать значение **`false`**, если диспетчер должен передавать это же сообщение другим обработчикам, и **`true`**, если дальнейший просмотр обработчиков следует прекратить.

В нашей простой системе не будет "адресных" сообщений. Действительно, если некий объект знает, кому он хочет передать некоторое сообщение, то он может вызвать метод `handle` непосредственно, минуя диспетчер сообщений. Поэтому в структуре сообщений останутся лишь два поля — класс сообщения и параметр сообщения.

```
public class Message {
  private int    msgClass;     // класс сообщения
  private Object param;        // параметр сообщения

  /**   Конструкторы   */
  public Message(int cl, Object par) {
```

```
    msgClass = cl;
    param = par;
  }

  public Message(int cl) { this(cl, null); }

  /**   Функции доступа   */
  public int getMsgClass() { return msgClass; }

  public Object getParam() { return param; }
}
```

Определить диспетчер несколько сложнее. Прежде всего, в системе может быть только один диспетчер. Поэтому постараемся описать класс `Dispatcher` таким образом, чтобы было невозможно создавать объекты этого класса с помощью оператора **new**. Вместо этого определим в данном классе статическую функцию `getInstance`, которая при первом обращении создаст единственный объект этого класса с помощью конструктора, определенного с атрибутом **private**, а при всех последующих обращениях будет просто выдавать объект, созданный при первом обращении.

В диспетчере также определим "системные" классы сообщений. У нас будет только один такой класс — команда диспетчеру об окончании работы. Договоримся, что все "системные" сообщения будут иметь отрицательные коды для того, чтобы каждая программа, использующая диспетчер для организации работы, могла оперировать с положительными числами для кодирования своих собственных классов сообщений.

Списки сообщений, объектов-генераторов сообщений и объектов-обработчиков сообщений будут представлены объектами стандартного системного класса `Vector`.

В листинге 6.1 представлено определение класса `Dispatcher`.

Листинг 6.1. Определение диспетчера сообщений

```
public class Dispatcher {
  public static int msgQuit = -1;  // класс стандартного сообщения

  private Vector generators = new Vector();  // список генераторов
  private Vector handlers = new Vector();    // список обработчиков
  private Vector messages = new Vector();    // очередь сообщений
```

```
//================================================
//  Определение конструктора и функции getInstance
//================================================
private static Dispatcher dispatcher = null;

private Dispatcher() {}

public static Dispatcher getInstance() {
  if (dispatcher == null) {
    dispatcher = new Dispatcher();
  }
  return dispatcher;
}

//================================================
//  Функции для работы со списком обработчиков
//================================================
public void addHandlerHead(IHandler hnd) {
  handlers.add(0, hnd);
}

public void addHandlerTail(IHandler hnd) {
  handlers.add(hnd);
}

public IHandler removeHandler(IHandler hnd) {
  if (handlers.removeElement(hnd)) {
    return hnd;
  } else {
    return null;
  }
}

//================================================
//  Функции для работы со списком генераторов
//================================================
public void addGenerator(IGenerator gen) {
  generators.add(gen);
}
```

```
public IGenerator removeGenerator(IGenerator gen) {
  if (generators.removeElement(gen)) {
    return gen;
  } else {
    return null;
  }
}

//================================================
//  Передача сообщения в очередь сообщений
//================================================
public void sendMessage(Message msg) {
  messages.add(msg);
}

//================================================
//  Основной цикл работы диспетчера сообщений
//================================================
public void run() {
  // Цикл может закончиться только при получении системной команды
  for (;;) {
    //----- 1. Попытаемся найти какое-либо сообщение
    Message msg = null;
    if (messages.size() > 0) {
      // Очередь сообщений не пуста — берем сообщение из нее
      msg = (Message)messages.elementAt(0);
      messages.removeElementAt(0);
    }
    if (msg == null) {
      // Очередь сообщений пуста, запрашиваем генераторы
      for (int i = 0;  i < generators.size();  i++) {
        IGenerator generator = (IGenerator)generators.elementAt(i);
        if ((msg = generator.generate()) != null) {
          break;
        }
      }
    }
```

```
      //----- 2. Проверка сообщения
      if (msg != null && msg.getMsgClass() == msgQuit) {
        // Заканчивается работа диспетчера,
        // а вместе с ним и всей программы
        break;
      }

      //----- 3. Передача сообщения на обработку
      if (msg != null) {
        for (int i = 0;  i < handlers.size();  i++) {
          IHandler handler = (IHandler)handlers.elementAt(i);
          if (handler.handle(msg)) {
            break;
          }
        }
      }
    }
  }
}
```

Как же будет выглядеть программа, использующая определенный нами диспетчер для организации работы?

В качестве примера рассмотрим достаточно сложную задачу моделирования работы пассажирского лифта в многоэтажном доме. В этой задаче параллельно действуют две нити процессов. Одна из них организует работу моделей лифта и пассажиров. Именно эта нить будет пользоваться диспетчером сообщений для выполнения обмена сообщениями во время работы. Другая нить служит для создания новых моделей пассажиров и ввода их в систему через некоторые промежутки времени.

Лифт может находиться в следующих состояниях:

- стоит, движется вниз, движется вверх;
- находится на уровне этажа с некоторым заданным номером;
- двери открыты (лифт стоит), двери закрыты (лифт стоит или движется);
- в лифте находится некоторое заданное число человек.

Пассажир лифта также может находиться в некоторых состояниях, характеризующихся следующими значениями:

- пассажир находится на уровне этажа с некоторым заданным номером;

- пассажир находится внутри лифта (и в этом случае номер его этажа должен совпадать с номером этажа лифта) или вне лифта;
- пассажир имеет желание попасть на этаж с некоторым заданным номером.

При каждом изменении состояния лифт извещает об этом пассажиров отсылкой соответствующего сообщения. Получив такое сообщение, пассажир может изменить также и свое состояние (например, войти в лифт). В свою очередь, пассажир, изменив свое состояние (например, захотев поехать на некоторый этаж), должен сообщить об этом лифту, который, получив запрос, будет менять свое состояние. Поскольку лифт в системе всего один, то сообщение об изменении состояния пассажира можно послать лифту и напрямую, минуя диспетчера, однако для общности удобнее воспользоваться для этого также общим механизмом.

Итак, имеем следующую структуру объектов в системе:

- один лифт (класса `Elevator`), который является обработчиком событий, генерируемых пассажирами лифта;
- несколько пассажиров (класса `Passanger`), каждый из которых является как обработчиком событий (генерируемых лифтом), так и генератором событий (например, если пассажир устал ждать лифт, он может отменить свою заявку и отправиться по лестнице пешком).

Таким образом, имеем такое описание классов (только схематически):

```
public class Elevator implements IHandler {
  final static int stateStop = 1;
  final static int stateUp   = 2;
  final static int stateDown = 3;

  final static int msgStateChanged = 1; // класс сообщения
                                        // об изменении состояния

  //---------------------------------------------------------------------
  // Класс, описывающий состояние лифта. Объекты этого класса будут
  // являться параметрами сообщения об изменении состояния лифта
  //---------------------------------------------------------------------
  public static class ElevatorState {
    private int state = stateStop;      // состояние лифта
    private int level = 1;              // этаж, на котором он находится
    private boolean doorsOpen = true;   // состояние дверей
    private int people = 0;             // количество людей в лифте

    public ElevatorState() {}
    public ElevatorState(int stat, int lev, boolean open, int pple) {
```

```
      state = stat;
      level = lev;
      doorsOpen = open;
      people = pple;
    }
    public int getState() { return state; }
    public void setState(int stat) { state = stat; }
    public int getLevel() { return level; }
    public void setLevel(int lev) { level = lev; }
    public boolean isDoorsOpen() { return doorsOpen; }
    public void setDoorsOpen() { doorsOpen = true; }
    public void setDoorsClose() { doorsOpen = false; }
    public int getPeople() { return people; }
    public void setPeople(int pple) { people = pple; }
  }

  // Состояние лифта
  ElevatorState state = new ElevatorState();
  // Массив людей, от которых поступили запросы на обслуживание
  Passanger[] passangers = new Passanger[0];

  public void addPassanger(Passanger p) {   // ... добавить пассажира
  }

  public void removePassanger(Passanger p) { // ... удалить пассажира
  }

  public boolean handle(Message msg) {
    // Обработка сообщения об изменении состояния одного из пассажиров.
    // Лифт регистрирует пассажира, если тот намеревается воспользоваться
    // лифтом, но еще не обслужен. Если заявка может быть выполнена,
    // лифт начинает двигаться в направлении обслуживания заявки
  }
}

public class Passanger implements IHandler, IGenerator {
  //-----------------------------------------------------------------------
  // Класс, описывающий состояние пассажира. Объекты этого класса будут
```

```
// являться параметрами сообщения об изменении состояния пассажира
//-----------------------------------------------------------------------
public static class PassangerState {
  private boolean inElevator = false;   // состояние пассажира
  private int     level = 1;            // номер этажа
  private int     wish  = 10;           // куда пассажир хочет попасть

  public PassangerState() {}
  public PassangerState(boolean inout, int lev, int w) {
    inElevator = inout;
    level = lev;
    wish = w;
  }
  public isInElevator() { return inElevator; }
  public setInElevator() { inElevator = true; }
  public setOutElevator() { inElevator = false; }
  public int getLevel() { return level; }
  public void setLevel(int lev) { level = lev; }
  public int getWish() { return wish; }
  public void setWish(int w) { wish = w; }
}

PassangerState state = new PassangerState();
long wait = 0;    // время ожидания обслуживания

public boolean handle(Message msg) {
  // Обработка сообщения об изменении состояния лифта.
  // В результате обработки пассажир может изменить свое состояние.
  // Изменение состояния, в свою очередь, может вызвать
  // появление нового сообщения об изменении состояния
}

public Message generate() {
  // Пассажир может сгенерировать сообщение, если он уже слишком долго
  // ждет. В этом случае он может отменить свой запрос (сообщив,
  // разумеется, как порядочный человек, об изменении своего состояния)
  // и покинуть систему. Впрочем, лифт может и сам,
```

```
    // не обнаружив на нужном этаже человека, сформировавшего запрос,
    // удалить его из системы
  }
}
```

Вторая нить служит для "поставки" пассажиров в систему. Она сформирует диспетчера, а затем через определенные промежутки времени будет создавать объекты-пассажиры и добавлять их в систему, используя вызовы `dispatcher.addGenerator(psg)` и `dispatcher.addHandlerTail(psg)`. Она же инициирует работу диспетчера в первой нити, вызвав его работу функцией `dispatcher.run()`. Эта же нить может также и прекратить процесс моделирования, передав диспетчеру системное сообщение `msgQuit`.

В следующем разделе рассмотрим более простой пример взаимодействия объектов путем передачи сообщений. Однако следует отметить, что этот пример является несколько надуманным. Поставленную в нем задачу гораздо легче можно решить традиционными методами, не прибегая к аппарату генерации и обмена сообщениями.

6.2. Об одном способе вычисления конечных сумм

Рассмотрим задачу вычисления конечной суммы некоторых слагаемых. Конечно, если эти слагаемые заданы с помощью некоторой формулы (функции), то наиболее простым будет следующий обычный способ вычисления суммы:

```
static public double sum(SumMember member, int n) {
  double s = 0;
  for (int i = 0;  i < n;  i++) {
    s += member.member(i);
  }
  return s;
}
```

В этом решении предполагается, что функции суммирования передается в качестве аргумента способ вычисления членов суммы в виде объекта класса `SumMember`, в котором и определен метод `member` для вычисления *i*-го члена суммы. Однако, может быть значения некоторых членов суммы удобно вычислять, используя уже вычисленные ранее значения других членов суммы. Например, в известной формуле для приближенного вычисления числа *e*

$$e = \sum_{i=0}^{n-1} \frac{1}{i!}$$

каждый член суммы кроме первого легко может быть определен с использованием значения предыдущего члена, таким образом, нецелесообразно рассчитывать каждый из членов суммы независимо от других. Для вычисления этой суммы было бы удобнее воспользоваться другим алгоритмом:

```
static public double sum(int n) {
  double s = 0;
  double u = 1;
  for (int i = 0;  i < n;  i++) {
    s += u;
    u /= i;
  }
  return s;
}
```

Если требуется вычислить сумму первых n членов последовательности Фибоначчи, каждый из которых (кроме первых двух, равных единице) может быть вычислен по известной формуле

$$F_n = F_{n-1} + F_{n-2},$$

то опять лучше воспользоваться специальным алгоритмом:

```
static public double sum(int n) {
  double s = 0;
  int F1 = 1, F2 = 1;
  for (int i = 2;  i < n;  i++) {
    F = F1 + F2;
    s += F;
    F1 = F2;  F2 = F;
  }
  return s;
}
```

Хотелось бы применить общую схему для вычисления сумм, несмотря на то, что в каждом конкретном случае зависимость значений одних членов суммы от других различна. Попробуем воспользоваться для этого схемой обмена сообщениями между объектами разных классов.

Определим класс-сумматор, объект которого будет ожидать сообщений от объектов — членов суммы. В каждом сообщении объект — член суммы сообщает свое значение и свой номер. Когда сумматор получит сообщения от всех членов суммы, его работа будет закончена.

В свою очередь, каждый объект — член суммы ждет сообщений от других членов суммы, от которых он зависит. Разумеется, в этой цепочке должны быть

члены, которые готовы выдать свое значение, не ожидая значений от других членов, в противном случае было бы невозможно завершить вычисления.

Таким образом, в нашей системе объектов имеется один объект — сумматор, и *n* объектов — членов суммы, каждый из которых порождает и принимает сообщения. Сумматор является обработчиком сообщений. Сам он генерирует единственное сообщение об окончании работы, возникающее тогда, когда сумматор получит сообщения от всех членов суммы. Члены суммы являются обработчиками сообщений от других членов суммы, а по крайней мере один из членов является также генератором сообщения, не зависящего от других значений. Итак, имеем следующие описания классов.

```
public class Summator implements IHandler {
  public static int msgMemberReady = 1; // класс сообщения от члена суммы

  int n;              // ожидаемое количество членов суммы
  boolean[] ready;    // массив отметок о пришедших членах суммы
  double sum = 0;     // накапливаемая сумма
  int members = 0;    // количество просуммированных членов

  // Класс, определяющий структуру параметра сообщения
  public static class MemberValue {
    int number;       // номер члена суммы
    double value;     // значение члена суммы
    public MemberValue(int num, double val) {
      number = num;
      value = val;
    }
    public int getNumber() { return number; }
    public double getValue() { return value; }
  }

  // Конструктор сумматора задает количество ожидаемых им членов суммы
  public Summator(int n) {
    this.n = n;
    ready = new boolean[n];
  }

  // Метод обработки сообщения
  public boolean handle(Message msg) {
```

```
    // Проверка класса сообщения: реагируем только на сообщения
    // класса msgMemberReady
    if (msg.getMsgClass() != msgMemberReady) return false;
    // Проверка типа параметра сообщения
    if (!(msg.getParam() instanceof MemberValue)) return false;
    // Взятие параметра сообщения
    MemberValue memVal = (MemberValue)msg.getParam();
    // Отметка члена суммы и суммирование
    if (!ready[memVal.getNumber()]) {
      members++;
      ready[memVal.getNumber()] = true;
      sum += memVal.getValue();
    }
    // Кончаем работу, если уже все члены суммы просуммированы
    if (members == n) {
      Dispatcher.getInstance().sendMessage(
                                        new Message(Dispatcher.msgQuit));
    }
    // Во всех случаях значение члена суммы передается другим объектам
    return false;
  }

  // Результат суммирования
  public double getResult() { return sum; }
}
```

Структура члена суммы, вообще говоря, не определена, поэтому можно пока описать лишь абстрактный класс `Member`, реализующий интерфейсы `IHandler` и `IGenerator`.

```
public abstract class Member implements IHandler, IGenerator {
  public boolean handle(Message msg) {
    return false;
  }

  public Message generate() {
    return null;
  }
}
```

Теперь мы можем определить несколько различных конкретных классов для суммирования различных последовательностей. Каждое новое определение будет задавать свой способ вычисления суммы. Таким образом мы можем определить функцию суммирования как функцию, параметризованную абстрактной фабрикой, порождающей те или иные члены конечной суммы.

```
public interface SumMemberFactory {
  Member createMember(int i);
}


public static double sum(SumMemberFactory factory, int n) {
  // Организуем диспетчер сообщений
  Dispatcher dispatcher = Dispatcher.getInstance();
  // Формируем "сцену" — набор обработчиков и генераторов сообщений
  Summator summator = new Summator(n);
  dispatcher.addHandlerTail(summator);
  for (int i = 0; i < n; i++) {
    Member nextMember = factory.createMember(i);
    dispatcher.addHandlerTail(nextMember);
    dispatcher.addGenerator(nextMember);
  }
  // Запускаем процесс обмена сообщениями
  dispatcher.run();
  // Получаем готовый результат
  return summator.getResult();
}
```

Здесь `SumMemberFactory` — абстрактная фабрика, порождающая абстрактные члены конечной суммы. Для того чтобы функция могла работать и выдавать конкретный результат, надо определить конкретный класс члена суммы и, соответственно, конкретную фабрику, порождающую объекты этого класса. Рассмотрим, например, простую задачу вычисления приближенного значения числа e по формуле

$$e = \sum_{i=0}^{n-1} \frac{1}{i!}.$$

Для вычисления необходимо описать классы `ESumMember` и `ESumMemberFactory`, которые будут определять алгоритмы вычисления членов суммы и порождать объекты — члены суммы соответственно. Ниже приведен код для класса `ESumMember`, а класс `ESumMemberFactory` явно не определяется, вместо этого экземпляр данного класса порождается сразу же при вызове функции суммирования.

```
public class ESumMember extends Member {
  int number;          // номер члена суммы (от нуля до некоторого n)
  boolean generated = false;   // Значение уже сгенерировано?

  // Конструктор запоминает номер
  public ESumMember(int i) {
    number = i;
  }

  // Генератор порождает сообщение о готовом значении,
  // только если номер члена ряда равен нулю.
  // В остальных случаях значение порождается по готовности
  // предыдущего члена суммы. Обратите внимание, что значение порождается
  // только один раз
  public Message generate() {
    if (number == 0 && !generated) {
      generated = true;
      return new Message(Summator.msgMemberReady,
                 new Summator.MemberValue(number, 1));
    } else {
      return null;
    }
  }

  // Обработчик принимает значение некоторого члена суммы и выдает
  // свое значение, если номер этого члена на единицу меньше собственного
  // номера. Обратите внимание, что значение генерируется только один раз
  public boolean handle(Message msg) {
    if (generated) return false;
    // Проверка класса сообщения
    if (msg.getMsgClass() != Summator.msgMemberReady) return false;
    // Проверка типа параметра сообщения
    if (!(msg.getParam() instanceof Summator.MemberValue)) return false;
    // Анализ параметра сообщения
    Summator.MemberValue value = (Summator.MemberValue)msg.getParam();
    if (value.getNumber() != number-1) return false;
    // Генерация нового сообщения
    Dispatcher.getInstance().sendMessage(
```

```
                new Message(Summator.msgMemberReady,
                            new Summator.MemberValue(
                                number, value.getValue() / number)));
    generated = true;
    // Во всех случаях сообщение передается дальше для анализа
    // его другими обработчиками (на самом деле это значение
    // может понадобиться разве что сумматору)
    return false;
  }
}
```

Теперь для вызова функции суммирования нужно лишь определить фабрику для порождения новых членов суммы ряда и общее количество суммируемых членов.

```
System.out.println(sum(new SumMemberFactory() {
  public Member createMember(int i) { return new ESumMember(i); }
}, 18));
```

Приведенный выше вызов выведет в стандартный выходной поток значение

```
2.7182818284590455
```

что практически находится на пределе машинной точности вычислений.

Приведем еще несколько примеров вычисления различных сумм практически без комментариев.

Сумма n первых членов последовательности Фибоначчи получается, если определить класс `FibSumMember` следующим образом.

```
public class FibSumMember extends Member {
  int number;                 // номер члена суммы
  int fPred = 0;              // значения двух предыдущих членов
  int fPredPred = 0;
  boolean generated = false;  // значение уже сгенерировано?

  // Конструктор запоминает номер члена суммы
  public FibSumMember(int i) {
    number = i;
  }

  // Генератор порождает сообщение о готовом значении, только если
  // номер члена ряда меньше двух. В остальных случаях значение
  // порождается по готовности предыдущих двух членов суммы.
```

```
  // Обратите внимание, что значение порождается только один раз
  public Message generate() {
    if ((number == 0 || number == 1) && !generated) {
      generated = true;
      return new Message(Summator.msgMemberReady,
                  new Summator.MemberValue(number, 1));
    } else {
      return null;
    }
  }

  // Обработчик принимает значение некоторого члена суммы и выдает
  // свое значение, если уже готовы оба предыдущих члена суммы.
  // Обратите внимание, что значение генерируется только один раз
  public boolean handle(Message msg) {
    if (generated) return false;
    if (msg.getMsgClass() != Summator.msgMemberReady) return false;
    if (!(msg.getParam() instanceof Summator.MemberValue)) return false;
    Summator.MemberValue value = (Summator.MemberValue)msg.getParam();
    if (value.getNumber() == number - 2) {
      // Готов член, находящийся перед предыдущим
      fPredPred = (int)value.getValue();
    } else if (value.getNumber() == number - 1) {
      // Готов предыдущий член
      fPred = (int)value.getValue();
    } else {
      return false;
    }
    if (fPred != 0 && fPredPred != 0) {
      Dispatcher.getInstance().sendMessage(
                  new Message(Summator.msgMemberReady,
                              new Summator.MemberValue(
                                  number, fPred + fPredPred)));
    }
    return false;
  }
}
```

Вызов сумматора

```
System.out.println((int)sum(new SumMemberFactory() {
  public Member createMember(int i) { return new FibSumMember(i); }
}, 10));
```

породит результат

```
143
```

А вот пример, в котором члены суммы не зависят друг от друга. Конечно, в этом случае данный подход к вычислению суммы совсем не оправдан, однако схема все же работает правильно. Итак, вычислим сумму членов по следующей формуле:

$$S = \sum_{i=0}^{n-1} (-1)^n \frac{1}{i+1}.$$

Эта формула при больших значениях n дает приближенное значение натурального логарифма 2. Класс для вычисления значений членов этой суммы вообще не обрабатывает приходящие к нему сообщения, вместо этого он сразу же генерирует нужное значение по запросу диспетчера на основании знаний о собственном номере.

```
public class LnSumMember extends Member {
  int number;                    // номер члена суммы
  boolean generated = false;     // значение уже сгенерировано?

  // Конструктор просто запоминает номер члена суммы
  public LnSumMember(int i) {
    number = i;
  }

  // Обработчик сообщений выключен
  public boolean handle(Message msg) {
    return false;
  }

  // Генератор сразу же выдает значение по номеру члена суммы
  public Message generate() {
    if (generated) return null;
    generated = true;
    return new Message(Summator.msgMemberReady,
           new Summator.MemberValue(
```

```
                number,
                ((number & 1) == 1 ? -1 : 1) / (double)(number + 1)));
  }
}
```

В результате соответствующего вызова функции суммирования

```
System.out.println(sum(new SumMemberFactory() {
  public Member createMember(int i) { return new LnSumMember(i); }
}, 100));
```

будет получено приближенное значение числа ln2.

```
0.688172179310195
```

Это не очень точное значение, даже несмотря на то, что просуммировано 100 членов. Оно отличается от истинного значения более, чем на 0,005. Но это просто потому, что соответствующий ряд очень медленно сходится.

Заметим, что во всех описанных случаях объекты — члены суммы становились пассивными сразу же после того, как выдадут свое значение в виде сообщения. Это, конечно, правильно, в противном случае диспетчер может оказаться "засыпанным" повторными сообщениями от этих членов. Тем не менее, такое поведение объектов снижает общую эффективность работы системы, поскольку все обработчики и генераторы сообщений остаются в системе "балластом", и диспетчер при появлении каждого нового сообщения все равно вынужден опрашивать все эти объекты.

Можно организовать работу объектов чуточку иначе. Породив свое значение, член суммы может просто исключить себя из списка генераторов или, соответственно, обработчиков сообщений. В этом случае не потребуется вводить и переменную — члена класса `generated`, поскольку член класса, породивший значение, убирает себя как из списка генераторов, так и из списка обработчиков сообщений. Вот как будет выглядеть модифицированное определение класса `FibSumMemebr`:

```
public class FibSumMember extends Member {
  int number;                  // номер члена суммы
  int fPred = 0;               // значения двух предыдущих членов
  int fPredPred = 0;

  // Конструктор запоминает номер члена суммы
  public FibSumMember(int i) {
    number = i;
  }
```

```
// Генератор порождает сообщение о готовом значении, только если
// номер члена ряда меньше двух. В остальных случаях значение
// порождается по готовности предыдущих двух членов суммы.
// Обратите внимание, что значение порождается только один раз
public Message generate() {
  if ((number == 0 || number == 1)) {
    Dispatcher.getInstance().removeGenerator(this);
    Dispatcher.getInstance().removeHandler(this);
    return new Message(Summator.msgMemberReady,
           new Summator.MemberValue(number, 1));
  } else {
    return null;
  }
}

// Обработчик принимает значение некоторого члена суммы и выдает
// свое значение, если уже готовы оба предыдущих члена суммы.
// Обратите внимание, что значение генерируется только один раз
public boolean handle(Message msg) {
  if (msg.getMsgClass() != Summator.msgMemberReady) return false;
  if (!(msg.getParam() instanceof Summator.MemberValue)) return false;
  Summator.MemberValue value = (Summator.MemberValue)msg.getParam();
  if (value.getNumber() == number - 2) {
    // Готов член, находящийся перед предыдущим членом
    fPredPred = (int)value.getValue();
  } else if (value.getNumber() == number - 1) {
    // Готов предыдущий член
    fPred = (int)value.getValue();
  } else {
    return false;
  }
  if (fPred != 0 && fPredPred != 0) {
    Dispatcher.getInstance().removeGenerator(this);
    Dispatcher.getInstance().removeHandler(this);
    Dispatcher.getInstance().sendMessage(
              new Message(Summator.msgMemberReady,
                        new Summator.MemberValue(
```

```
                                    number, fPred + fPredPred)));
    }
    return false;
  }
}
```

К сожалению, теперь уже диспетчер оказывается не готовым к динамическому удалению генераторов и обработчиков в процессе обработки сообщения. В результате он может просто пропустить одного из обработчиков сообщения, что приводит к катастрофическим последствиям! Чтобы этого не произошло, надо либо изменить представление очередей генераторов и обработчиков в диспетчере, либо удалять их более аккуратно, просто заменяя ссылки на объекты пустыми ссылками. Конечно, теперь следует предусмотреть возможность появления пустой ссылки в списке обработчиков или генераторов. Такую пустую ссылку можно просто проигнорировать или удалить из соответствующего списка в процессе обработки.

В листинге 6.2 приведено модифицированное описание диспетчера (только измененные определения методов `removeGenerator`, `removeHandler` и `run`).

Листинг 6.2. Модифицированное определение диспетчера сообщений

```
public class Dispatcher {
  /* Неизмененная часть определения пропущена... */

  //================================================
  //  Функции для работы со списком обработчиков
  //================================================
  public IHandler removeHandler(IHandler hnd) {
    int ndx = handlers.indexOf(hnd);
    if (ndx >= 0) {
      handlers.setElementAt(null, ndx);
      return hnd;
    }
    return null;
  }

  //================================================
  //  Функции для работы со списком генераторов
  //================================================
  public IGenerator removeGenerator(IGenerator gen) {
```

```
    int ndx = generators.indexOf(gen);
    if (ndx >= 0) {
      generators.setElementAt(null, ndx);
      return gen;
    }
    return null;
  }

  //================================================
  //  Основной цикл работы диспетчера сообщений
  //================================================
  public void run() {
    // Цикл может закончиться только после обработки системной команды
    for (;;) {
      //----- 1. Попытаемся найти какое-либо сообщение
      Message msg = null;
      if (messages.size() > 0) {
        // Очередь сообщений не пуста — берем сообщение из нее
        msg = (Message)messages.elementAt(0);
        messages.removeElementAt(0);
      }
      if (msg == null) {
        // Очередь сообщений пуста, запрашиваем генераторы
        for (int i = 0;  i < generators.size();  i++) {
          IGenerator gen = (IGenerator)generators.elementAt(i);
          if (gen == null) {
            generators.removeElementAt(i);
          } else if ((msg = gen.generate()) != null) {
            break;
          } else {
            i++;
          }
        }
      }

      //----- 2. Проверка сообщения
      if (msg != null && msg.getMsgClass() == msgQuit) {
        // Заканчивается работа диспетчера,
```

```
        // а вместе с ним и всей программы
        break;
      }

      //----- 3. Передача сообщения на обработку
      if (msg != null) {
        for (int i = 0;  i < handlers.size();  i++) {
          IHandler hnd = (IHandler)handlers.elementAt(i);
          if (hnd == null) {
            handlers.removeElementAt(i);
          } else if (hnd.handle(msg)) {
            break;
          } else {
            i++;
          }
        }
      }
    }
  }
}
```

Разумеется, задача о суммировании конечного числа взаимозависимых величин может быть решена и более простыми средствами, а главное — легко написать более эффективную программу и в каждом конкретном случае, да, наверное, и в общем случае тоже. Эта задача приведена здесь только как иллюстрация метода обмена сообщениями. В реальной жизни обмен сообщениями чаще всего используется для организации взаимодействия пользователя с программными объектами, такими как диалоговые элементы, документы, рисунки и т. п.

Глава 7

Функция как носитель информации

Довольно часто самым удобным способом представления данных является функция. Вообще, иногда говорят о двух формах представления знаний — объектной и функциональной — причем часто функциональная форма представления оказывается предпочтительней. Действительно, часто ли вы предпочитаете на вопрос о том, где расположен нужный вам дом, получить действительную информацию о его адресе? Пожалуй, чаще вам захочется услышать, как туда добраться, а это, конечно, одна из форм функционального представления знания. Нередко люди вообще, ориентируясь в знакомом месте, хорошо представляют себе, как дойти до нужного им места, но плохо представляют себе, как это место расположено в пространстве. То есть их знание оказывается преимущественно функциональным.

Если перейти к более точным математическим объектам, то и в этом случае функциональное представление знания оказывается удобным. Например, если описывается некоторое множество объектов, то иногда легче описать их характеристическое свойство (задать функцию), чем перечислить эти объекты явно. В случае бесконечных множеств функциональная форма задания множества часто оказывается единственно возможной.

В *разд. 7.1* мы рассмотрим функциональное представление множеств целых чисел и работу с таким представлением. В *разд. 7.2* попробуем применить функциональную форму представления знаний к более традиционной области — для решения классической задачи о расстановке восьми ферзей на стандартной шахматной доске (решение задачи восходит к решению Г. С. Цейтина[1]).

В *разд. 7.3—7.5* представлено еще несколько задач, в которых используется функциональное представление знаний.

---

[1] Доктор физико-математических наук, профессор Г. С. Цейтин известен как автор многих работ по математической логике и ассоциативным сетям. Долгое время заведовал лабораторией в Санкт-Петербургском Государственном Университете. В настоящее время является сотрудником фирмы Rational, США.

7.1. Еще о представлении множеств

Представление множеств было описано в *главе 1*. Тогда же мы договорились, что будем рассматривать только множества целых чисел из заданного достаточно ограниченного диапазона. Говорить о представлении множества всех целых чисел или хотя бы о множестве только четных чисел смысла не было. Тем не менее, часто встречаются задачи, в которых требуется работать с гораздо более обширными множествами. Например, весьма популярна задача о нахождении множества всех простых чисел (обычно, впрочем, нет нужды в представлении всех простых чисел сразу, но только о потенциальной возможности найти любое по порядку простое число, разумеется, в пределах диапазона представления целых чисел в компьютере).

Во всех этих случаях допустимо представление множества в виде функции (или ряда функций). Например, вполне адекватным представлением множества служит итератор, выдающий по очереди все элементы множества. Конечно, вполне может оказаться, что элементов бесконечно много, но это не помешает исправно получать элементы один за другим. Правда одного итератора явно недостаточно, если надо проверить принадлежность заданного элемента множеству, получить пересечение или разность множеств и т. д. Удобным способом представления множества служит так называемая *"характеристическая функция"* — функция, которая по заданному элементу выдает логическое значение — `true`, если элемент принадлежит множеству, и `false`, — если не принадлежит. Такая функция позволяет легко проверить, входит ли элемент в множество, построить объединение или пересечение множеств, но она практически бесполезна при попытке перечислить все элементы множества.

В этом разделе мы будем рассматривать множества неотрицательных целых чисел. В качестве основного способа представления множества выберем характеристическую функцию множества. Имея такую функцию, мы легко сможем получить итератор элементов множества, перебирая всевозможные неотрицательные целые числа и проверяя, не входят ли они в множество, вызывая для такой проверки характеристическую функцию. Конечно, во многих случаях это окажется не самым эффективным способом реализации итератора, кроме того, у нас практически нет возможности проверить, есть ли еще элементы в множестве, т. к. для этого нужно перебрать все целые неотрицательные числа. Но ведь и применяться такое представление будет, скорее всего, для бесконечных множеств! Так что не следует, используя реализованный нами итератор, пытаться получить список *всех* элементов множества. Это приведет лишь к зацикливанию вашей программы, даже если множество на самом деле конечно.

Итак, определим абстрактный тип "множество неотрицательных целых чисел" заданием следующего интерфейса:

```
public interface IntSet {
  boolean contains(int n);
  Iterator elements();
}
```

Для удобства дальнейших определений различных множеств мы теперь можем задать абстрактный класс `AbstractSet`, в котором определим итератор `elements` через интерфейсную функцию `contains`. Внутри того же определения класса мы можем задать и другие операции над множествами, например, добавление и удаление отдельных элементов некоторого множества, объединение, пересечение, дополнение и разность множеств, добавление и удаление отрезка целых чисел. Для каждой такой операции задается новое определение класса, содержащее свое собственное определение функции `contains`, и, следовательно, реализующее некоторое новое множество. Определение класса `AbstractSet` приведено в листинге 7.1.

Листинг 7.1. Определение абстрактного множества целых и операций над ним

```
//-------------------------------------------------------------
//  Абстрактный класс AbstractSet реализует функцию итератора
//  элементов множества через стандартную интерфейсную функцию
//  contains
//-------------------------------------------------------------
public abstract class AbstractSet implements IntSet {
  //-----------------------------------------------------------
  //  Внутренний класс IntSetIterator реализует итератор
  //  элементов множества
  //-----------------------------------------------------------
  private class IntSetIterator implements Iterator {
    int current = 0;      // текущий проверяемый элемент

    // Функция hasNext всегда возвращает true, т. к. множество,
    // вообще говоря, предполагается бесконечным
    public boolean hasNext() { return true; }

    // Функция next возвращает следующий элемент множества, если он
    // есть. Если множество конечно, то эта функция может зациклиться
```

```
  public Object next() {
    // Цикл поиска очередного элемента
    while (!contains(current++)) ;
    // Выдача результата
    return new Integer(current - 1);
  }

  // Функция remove приведена только для совместимости с интерфейсом
  // итератора. Для удаления элементов предусмотрена другая функция
  public void remove() {}
}

// Функция elements возвращает итератор элементов
// этого абстрактного множества
public Iterator elements() {
  return new IntSetIterator();
}

//-----------------------------------------------------------------
//  Класс SetPlusElement реализует понятие множества
//  с добавленным элементом
//-----------------------------------------------------------------
private static class SetPlusElement extends AbstractSet {
  IntSet oldSet;                    // множество
  int newElement;                   // и добавленный к нему элемент

  // Конструктор множества запоминает, какой элемент добавлен
  public SetPlusElement(IntSet set, int n) {
    oldSet = set;
    newElement = n;
  }

  // Новая функция contains отличается от старой только тем,
  // что выдает информацию о добавленном элементе
  public boolean contains(int n) {
    return n == newElement || oldSet.contains(n);
  }
}
```

```
// Функция add возвращает множество с добавленным элементом
public static IntSet add(IntSet set, int n) {
  return new SetPlusElement(set, n);
}

//-----------------------------------------------------------------
//  Класс SetPlusRange реализует понятие множества с добавленными
//  элементами из заданного диапазона
//-----------------------------------------------------------------
private static class SetPlusRange extends AbstractSet {
  IntSet oldSet;      // множество
  int minElement;     // минимальный из добавленных к нему элементов
  int maxElement;     // максимальный из добавленных к нему элементов

  // Конструктор множества запоминает, какие элементы добавлены
  public SetPlusRange(IntSet set, int min, int max) {
    oldSet = set;
    minElement = min;
    maxElement = max;
  }

  // Новая функция contains отличается от старой только тем,
  // что выдает информацию о добавленных элементах
  public boolean contains(int n) {
    return n >= minElement && n <= maxElement || oldSet.contains(n);
  }
}

// Функция addRange возвращает множество с добавленными элементами
public static IntSet addRange(IntSet set, int min, int max) {
  return new SetPlusRange(set, min, max);
}

//-----------------------------------------------------------------
//  Класс SetMinusElement реализует понятие множества
//  с удаленным элементом
//-----------------------------------------------------------------
private static class SetMinusElement extends AbstractSet {
```

```
  IntSet oldSet;                    // множество
  int remElement;                   // и удаленный из него элемент

  // Конструктор множества запоминает, какой элемент удаляется
  public SetMinusElement(IntSet set, int n) {
    oldSet = set;
    remElement = n;
  }

  // Новая функция contains отличается от старой только тем,
  // что выдает информацию об удаленном элементе
  public boolean contains(int n) {
    return n != remElement && oldSet.contains(n);
  }
}

// Функция remove возвращает множество с удаленным элементом
public static IntSet remove(IntSet set, int n) {
  return new SetMinusElement(set, n);
}

//-----------------------------------------------------------------
//  Класс SetMinusRange реализует понятие множества
//  с удаленными элементами из заданного диапазона
//-----------------------------------------------------------------
private static class SetMinusRange extends AbstractSet {
  IntSet oldSet;      // множество
  int minElement;     // минимальный удаленный из него элемент
  int maxElement;     // максимальный удаленный из него элемент

  // Конструктор множества запоминает, какие элементы удаляются
  public SetMinusRange(IntSet set, int min, int max) {
    oldSet = set;
    minElement = min;
    maxElement = max;
  }

  // Новая функция contains отличается от старой только тем,
  // что выдает информацию об удаленных элементах
```

```
  public boolean contains(int n) {
    return (n < minElement || n > maxElement) && oldSet.contains(n);
  }
}

// Функция removeRange возвращает множество с удаленными элементами
public static IntSet removeRange(IntSet set, int min, int max) {
  return new SetMinusRange(set, min, max);
}

//---------------------------------------------------------------------
//  Класс Conjunction реализует понятие множества,
//  образованного пересечением двух других множеств
//---------------------------------------------------------------------
private static class Conjunction extends AbstractSet {
  IntSet s1;          // первое из пересекаемых множеств
  IntSet s2;          // второе из пересекаемых множеств

  // Конструктор запоминает пересекаемые множества
  public Conjunction(IntSet s1, IntSet s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  // Элемент входит в пересечение, если он содержится в обоих
  // пересекаемых множествах
  public boolean contains(int n) {
    return s1.contains(n) && s2.contains(n);
  }
}

// Функция conjunct возвращает пересечение двух множеств
public static IntSet conjunct(IntSet s1, IntSet s2) {
  return new Conjunction(s1, s2);
}

//---------------------------------------------------------------------
//  Класс Disjunction реализует понятие множества,
```

```
//  образованного объединением двух других множеств
//----------------------------------------------------------------------
private static class Disjunction extends AbstractSet {
  IntSet s1;          // первое из объединяемых множеств
  IntSet s2;          // второе из объединяемых множеств

  // Конструктор запоминает объединяемые множества
  public Disjunction(IntSet s1, IntSet s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  // Элемент входит в объединение, если он содержится хотя бы
  // в одном из объединяемых множеств
  public boolean contains(int n) {
    return s1.contains(n) || s2.contains(n);
  }
}

// Функция disjunct возвращает объединение двух множеств
public static IntSet disjunct(IntSet s1, IntSet s2) {
  return new Disjunction(s1, s2);
}

//----------------------------------------------------------------------
//  Класс Inversion реализует понятие множества, образованного
//  дополнением другого множества
//----------------------------------------------------------------------
private static class Inversion extends AbstractSet {
  IntSet oldSet;          // инвертируемое множество

  // Конструктор запоминает инвертируемое множество
  public Inversion(IntSet s) {
    oldSet = s;
  }

  // Элемент содержится в инверсии, если он не содержится
  // в исходном множестве
```

```
  public boolean contains(int n) {
    return n >= 0 && !oldSet.contains(n);
  }
}

// Функция inverse возвращает дополнение множества
public static IntSet inverse(IntSet s) {
  return new Inversion(s);
}

//---------------------------------------------------------------------
//  Класс Difference реализует понятие множества,
//  образованного разностью двух других множеств
//---------------------------------------------------------------------
private static class Difference extends AbstractSet {
  IntSet s1;          // уменьшаемое
  IntSet s2;          // вычитаемое

  // Конструктор запоминает вычитаемые множества
  public Difference(IntSet s1, IntSet s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  // Элемент содержится в разности, если он содержится
  // в уменьшаемом, но не содержится в вычитаемом
  public boolean contains(int n) {
    return s1.contains(n) && !s2.contains(n);
  }
}

// Функция diff выдает разность двух заданных множеств
public static IntSet diff(IntSet s1, IntSet s2) {
  return new Difference(s1, s2);
}

// Еще две полезных константы типа IntSet
public static final IntSet emptySet = new AbstractSet() {
```

```
    public boolean contains(int n) {
      return false;
    }
  };

  public static final IntSet fullSet = new AbstractSet() {
    public boolean contains(int n) {
      return n >= 0;
    }
  };
}
```

Теперь мы определили достаточно богатый набор операций, чтобы попробовать запрограммировать какую-нибудь задачу, в которой происходит обработка бесконечных множеств. В качестве примера рассмотрим задачу о нахождении множества всех простых чисел.

Для нахождения множества простых чисел будем использовать алгоритм, известный под названием "решето Эратосфена". Сначала возьмем множество всех целых чисел, бóльших единицы. Затем будем последовательно выполнять шаги по "просеиванию" этого множества. Сначала выберем наименьший элемент множества — число 2 — и вычеркнем из множества все числа, кратные двойке, кроме самого числа 2. После этого перейдем к следующему оставшемуся в множестве элементу — числу 3 — и вычеркнем все элементы, кратные 3 (опять кроме самого числа 3). Потом возьмем очередной элемент — число 5, и т. д. Множество простых чисел получается после выполнения бесконечного числа таких шагов.

Конечно, на самом деле мы не сможем выполнить бесконечное множество шагов, так что в результате получим не множество всех простых чисел, а лишь некоторое его подмножество, например, все простые числа, меньшие 100.

Сначала определим исходное множество целых чисел, бóльших единицы. Это просто:

```
IntSet from2 = new AbstractSet() {
  public boolean contains(int n) { return n >= 2; }
};
```

Далее определим функцию, которая вычеркивает из множества все числа, кратные заданному целому числу. Это оказывается тоже не очень сложно:

```
private static IntSet filter(final int f, final IntSet set) {
  return new AbstractSet() {
    public boolean contains(int n) {
```

```
      return n % f != 0 && set.contains(n);
    }
  };
}
```

Теперь мы, наконец, готовы написать основную рекурсивную функцию, которая и выдает множество всех простых чисел, меньших заданного.

```
private static IntSet sieve(int max, IntSet set) {
  int first = ((Integer)set.elements().next()).intValue();
  return (first * first > max ? set :
          AbstractSet.add(sieve(max, filter(first,
                                AbstractSet.remove(set, first))), first));
}
```

Определение этой функции построено на том факте, что если из множества уже вычеркнуты все числа, кратные корню квадратному из числа `max`, то все простые числа, меньшие `max`, уже содержатся в качестве первых элементов множества. Конечно, если из определения функции выбросить проверку на максимальное значение, то функция при вызове зациклится в попытке определить все бесконечное множество простых чисел. Теперь можно сформировать множество всех простых чисел, меньших 100 с помощью вызова

```
IntSet primes = sieve(100, from2);
```

Если теперь распечатать все значения из этого множества с помощью оператора

```
int n;
for (Iterator it = primes.elements();
     (n = ((Integer)it.next()).intValue()) < 100; ) {
  System.out.print(" " + n);
}
System.out.println();
```

то в результате получится следующая строка:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

то есть и в самом деле будут напечатаны все простые числа, меньшие 100.

Надо сказать, что эффективность работы представленной программы невысока. Функции, которые строятся по ходу работы этой программы, оказываются очень сложными. Поэтому, в конце концов, выяснение, принадлежит ли некоторый элемент множеству простых чисел, выливается в довольно длинную последовательность проверок, не делится ли данное число на 2, не делится ли оно на 3, 5, и т. д. последовательно на все простые числа. Тем не менее, программа довольно изящная и наглядно демонстрирует, как более сложные множества строятся из более простых.

7.2. Задача о расстановке ферзей на шахматной доске

Рассмотрим хорошо известную задачу о расстановке 8 ферзей на стандартной шахматной доске, такой, чтобы никакие два ферзя не атаковали друг друга. Под стандартной шахматной доской понимается квадратная доска размером 8×8 клеток, в каждой клетке которой может располагаться одна фигура (ферзь). Считается, что два ферзя атакуют друг друга, если они расположены на одной вертикали, одной горизонтали или одной диагонали.

Существует несколько модификаций этой задачи. Согласно одной из них, надо найти всевозможные правильные расстановки ферзей, причем расстановки считаются совпадающими, если они получаются друг из друга поворотом доски или ее зеркальным отражением. Известно, что таких позиций всего 92. Другая модификация требует лишь определить, существует ли такая расстановка, и если существует, то найти одну из них.

Традиционно эта задача решается с помощью определения рекурсивной функции, основная идея которой состоит в том, чтобы свести задачу о расстановке ферзей на доске размером $n \times n$ клеток к задаче о расстановке ферзей на доске меньшего размера. Рассмотрим одно из традиционных решений задачи, причем будем искать только одну какую-нибудь правильную расстановку ферзей.

Прежде всего, необходимо решить, как представлять в программе позиции на шахматной доске с расставленными на ней ферзями. В традиционном решении это представление в виде массива, содержащего информацию о позициях ферзей, причем поскольку никакие два ферзя в правильной расстановке не могут находиться на одной горизонтали, то достаточно хранить для каждой горизонтали только номер вертикали, на которой стоит ферзь. Понятно, что в правильной расстановке все восемь чисел, представляющие номера вертикалей в массиве, будут различны, таким образом, любая правильная расстановка 8 ферзей на шахматной доске может быть представлена перестановкой набора натуральных чисел от 1 до 8.

Самое простое решение задачи, очевидно, состоит в том, чтобы исследовать все перестановки набора чисел от 1 до 8, причем для каждой из них необходимо проверить, представляет ли она правильную расстановку ферзей. В любой перестановке, естественно, выполнены условия о том, что никакие два ферзя не находятся на одной вертикали (поскольку все числа в перестановке различны) и не находятся на одной горизонтали (просто по способу представления позиции). Таким образом, необходимо лишь проверить, не стоят ли два ферзя на одной диагонали. Если массив P типа `int[8]` представляет некоторую перестановку целых чисел 1, 2, ..., 8, то в соответствующей расстановке ферзей два ферзя с номерами i и j будут находиться на одной диагонали, если

```
abs(P[i] – P[j]) == abs(i - j)
```

Отсюда можно сразу же получить простую программу, приведенную в листинге 7.2. Сразу же заметим, что на самом деле мы все время решаем несколько более общую задачу — нахождение правильной расстановки n ферзей на доске размером $n \times n$.

Листинг 7.2. Простая программа поиска расстановки ферзей на шахматной доске

```
// Функция correct проверяет, представляет ли заданная перестановка чисел
// от 1 до n корректную расстановку ферзей на шахматной доске
static boolean correct(int[] pos) {
  int n = pos.length;         // размер стороны доски
  // Перебираем всевозможные пары двух различных ферзей
  for (int i = 0;  i < n;  i++) {
    for (int j = i+1;  j < n;  j++) {
      if (Math.abs(i-j) == Math.abs(pos[i]-pos[j]))
        // Ферзи стоят на одной диагонали
        return false;
    }
  }
  // Проверка завершена успешно, ни одной пары ферзей,
  // стоящих на одной диагонали, не найдено
  return true;
}

// Функция recQueen находит правильную расстановку ферзей на доске
// размером nxn при условии, что первые k ферзей уже расставлены,
// поэтому переставлять можно только числа, находящиеся в элементах
// с индексами, большими k. Если такая расстановка существует, то функция
// выдает ее в качестве результата. Если расстановки с такими начальными
// данными не существует, функция выдает null
static int[] recQueen(int[] p, int k) {
  int n = p.length;           // количество ферзей
    if (k == n) {             // все ферзи уже расставлены?
      // Проверяем расстановку на корректность
      return (correct(p) ? p : null);
    }
    // Находим всевозможные перестановки элементов с индексами
    // от k до n-1
```

```
    for (int i = k;  i < n;  i++) {
      // Переставляем местами i-й и k-й элементы
      int c = p[k];  p[k] = p[i];  p[i] = c;
      // Пытаемся найти расстановку с фиксированными k+1 ферзями
      int[] per = recQueen(p, k+1);
      if (per != null) return per;
    }
    // Ничего не найдено
    return null;
  }

// Главная функция поиска расстановки n ферзей на доске размером nxn
static int[] queen(int n) {
  int[] pos = new int[n];
  // Формирование начальной расстановки
  for (int i = 0;  i < n;  i++) {
    pos[i] = i+1;
  }
  // Вызов основной рекурсивной процедуры
  return recQueen(pos, 0);
}
```

Теперь, имея эту программу, можно находить правильные расстановки ферзей на досках разного размера и печатать их в удобном для человека виде. Например, следующим образом:

```
// Ищем расстановку ферзей с помощью функции queen
int [] pos = queen(8);
if (pos == null) {
  System.out.println("Не существует!");
} else {
  // Распечатываем найденную позицию в удобном виде
  // Это цикл по всем горизонталям, начиная с первой
  for (int i = 0;  i < pos.length;  i++) {
    // Печать одной горизонтали
    int queenPos = pos[i];
    for (int k = 1;  k < queenPos;  k++) {
      System.out.print(". ");
    }
```

```
    System.out.print("Q ");
    for (int k = queenPos+1;  k <= pos.length;  k++) {
      System.out.print(". ");
    }
    System.out.println();
  }
  System.out.println();
}
```

На самом деле только что приведенное решение далеко не самое эффективное. Прежде всего нет смысла перебирать всевозможные перестановки чисел от 1 до *n*, если из них заведомо большинство не представляет правильных расстановок ферзей. Более эффективным является метод, при котором расстановки ферзей строятся и одновременно сразу в процессе построения выясняется их правильность. Другими словами, при постановке очередного ферзя на доску сразу же проверяется, не атакует ли он какого-нибудь из уже существующих ферзей. Эта идея сразу же приводит нас к похожей программе, показанной в листинге 7.3, которая, однако, работает значительно быстрее первого варианта.

```
// Функция recQueen находит правильную расстановку ферзей на доске
// размером nxn при условии, что первые k ферзей уже расставлены
// и друг друга не атакуют, так что расставлять нужно только ферзей
// с номерами, большими k. Если такая расстановка существует, то функция
// выдает ее в качестве результата. Если расстановки
// с такими начальными данными не существует, функция выдает null
static int[] recQueen(int[] p, int k) {
  int n = p.length;              // количество ферзей
  if (k == n) return p;          // все ферзи уже расставлены корректно
  for (int j = 1;  j <= n;  j++) {
    // Пробуем поставить (k+1)-го ферзя последовательно на все вертикали
    boolean correct = true;      // признак корректной расстановки
    for (int i = 0;  i < k;  i++) {
      // Проверяем, не атакует ли новый ферзь уже поставленных
      if (p[i] == j || k - i == Math.abs(j - p[i])) {
        // Если новый ферзь находится на одной вертикали или диагонали
        // с некоторым i-м ферзем, то расстановка некорректна
        correct = false;
        break;
      }
    }
```

```
    if (correct) {
      p[k] = j;
      // Установили ферзя в свою позицию и пробуем расставить
      // остальных ферзей с помощью рекурсивного вызова функции
      int[] pos = recQueen(p, k+1);
      if (pos != null)      // расстановка найдена!
        return pos;
    }
  }
  // Перебрали все варианты постановки (k+1)-го ферзя,
  // но ничего не нашли!
  return null;
}

// Главная функция поиска расстановки n ферзей на доске размером nxn
static int[] queen(int n) {
  return recQueen(new int[n], 0);
}
```

Этот вариант программы работает гораздо быстрее. Так, на домашнем компьютере автора вариант расстановки ферзей на доске 13×13 был получен за 0,01 секунды, в то время как программа, написанная по первому варианту, работала 12,5 секунд, чтобы получить тот же результат.

Теперь рассмотрим вариант решения, из-за которого, собственно, мы и начали весь этот разговор. В этом варианте речь идет о совершенно ином способе представления позиции на шахматной доске. В действительности нам совсем не обязательно иметь физическую информацию о том, где конкретно расположен тот или иной ферзь. Эта информация нужна нам только из-за необходимости:

- проверять, можно ли поставить очередного ферзя в некоторую позицию (*i*, *j*);
- выводить информацию о расстановке ферзей в конце работы.

Таким образом, на самом деле можно представлять позицию с помощью пары функций: функции проверки возможности установки ферзя в позицию (*i*, *j*) и функции "распечатки" позиции. Соответствующий (абстрактный) тип данных может быть представлен следующим интерфейсом:

```
public interface IPosition {
  boolean permits(int i, int j);
  void print(PrintStream ps);
}
```

В этом интерфейсе функция `permits` обеспечивает проверку возможности установки нового ферзя в позицию, а функция `print` — вывод позиции в выходной поток. Теперь каждая конкретная позиция может быть представлена объектом, удовлетворяющим этому интерфейсу. Например, пустая позиция (позиция, на которой нет ни одного ферзя) может быть представлена следующим объектом класса `IPosition`:

```
final private IPosition empty = new IPosition() {
  public boolean permits(int i, int j) { return true; }
  public void print(PrintStream ps) { ps.println(); }
};
```

Разумеется, это пример очень простой позиции — в ней нового ферзя можно поставить в любую клетку, а при ее выводе просто ничего не печатается.

Тем не менее, такая элементарная позиция может служить основой для построения другой, чуть более сложной позиции, в которой уже некоторый ферзь стоит в верхней горизонтали и занимает там указанную позицию *k*. Если значение *k* задано константой, то такая позиция может быть представлена следующим образом:

```
final int k;
final private IPosition oneRow = new IPosition() {
  public boolean permits(int i, int j) {
    return empty.permits(i, j) && (i != 1) &&
           (Math.abs(i-1) != Math.abs(j-k));
  }
  public void print(PrintStream ps) {
    empty.print(ps);
    for (int j = 1; j < k; j++) { ps.print(". "); }
    ps.print("Q ");
    for (int j = k+1; j <= maxQueens; j++) { ps.print(". "); }
  }
}
```

В этом представлении для задания размера доски использована еще одна константа — `maxQueens`.

Теперь понятно, как, имея одну позицию, можно получить другую, "нарастив" представляющие эту позицию функции. Решение, представленное в листинге 7.3, использует эту идею для решения задачи.

Листинг 7.3. Расстановка ферзей с помощью функционального представления позиции

```
// Размер доски
private static int maxQueens = 8;
```

```
// Рекурсивная функция, получающая новую правильную позицию из старой
// с помощью добавления нового ферзя на следующую горизонталь.
// Аргумент row задает количество уже расставленных ферзей
// (заполненных горизонталей)
private static IPosition recQueen(
        final IPosition position,
        final int row) {
  if (row == maxQueens) {
    // Уже расставлены все ферзи
    return position;
  } else {
    // Пытаемся поставить нового ферзя в ряд с номером row
    for (int col = 1;  col <= maxQueens;  col++) {
      final int fcol = col;              // номер колонки в ряду
      if (position.permits(row, col)) {
        // Новый ферзь не атакует ни одного из уже расставленных.
        // Формируем новую позицию и делаем рекурсивный вызов
        IPosition newPos = recQueen(new IPosition() {
          // Функция проверки правильности расстановки
          public boolean permits(int i, int j) {
            // Проверяем:
            //  а) совместимость с прошлой позицией;
            //  б) несовпадение номеров колонок с новой;
            //  в) отсутствие ферзя на одной диагонали с новым.
            return position.permits(i, j) && j != fcol &&
                   Math.abs(i - row) != Math.abs(j - fcol);
          }
          // Функция распечатки новой позиции
          public void print(PrintStream ps) {
            position.print(ps);      // печать старой позиции
            // Печатаем новую горизонталь
            for (int i = 1; i < fcol; i++) { ps.print(". "); }
            ps.print("Q ");
            for (int i = fcol+1; i <= maxQueens; i++) { ps.print(". "); }
            ps.println();
          }
        }, row + 1);
        // Если удалось обнаружить правильную позицию с таким ферзем
```

```
        if (newPos != null) return newPos;
      }
    }
    // Не удалось поставить нового ферзя ни на одну вертикаль
    return null;
  }
}

public static IPosition queen() {
  // В начале работы позиция пуста, и ни один ферзь еще не расставлен
  return recQueen(empty, 0);
}
```

Новая функция нахождения расстановки работает несколько медленнее, чем старая. Так, например, на компьютере автора расстановку ферзей на доске 20×20 функциональный вариант программы нашел за 4,2 секунды, в то время как предыдущая версия справилась с этой работой за 1,5 секунды. Однако потребности в памяти у функциональной программы заметно меньше. Все, что ей нужно — это память под стек вызовов процедур, причем стек этот не очень глубок. В предыдущем варианте помимо стека память требовалась под собственно представление позиции — вектор целочисленных элементов.

7.3. Задача о назначениях

Конечно, принцип функционального представления информации применим не только к задаче о расстановке ферзей и работе с бесконечными множествами. Почти в любой задаче наряду с объектным представлением существует и параллельный вариант решения с функциональным представлением информации. Вопрос лишь в том, чтобы правильно определить набор функций, адекватно отражающий нужную информацию.

Рассмотрим, например, классическую задачу о назначениях, несколько напоминающую задачу о расстановке ферзей. Пусть имеется n рабочих мест и n работников, каждый из которых может выполнять любую работу, но производительность труда у каждого рабочего на разных рабочих местах различна. Неотрицательная величина производительности труда каждого рабочего на каждом рабочем месте задана матрицей W размером $n \times n$, хранящей вещественные значения. Требуется расставить рабочих по рабочим местам так, чтобы их суммарная производительность для всех рабочих мест была максимальной.

Так же, как и в случае расстановки ферзей, одно из возможных представлений расстановки рабочих — это вектор длины n, в котором i-й элемент со-

ответствует порядковому номеру рабочего, поставленного на *i*-е рабочее место. Ясно, что если этот вектор обозначить через *P*, то все значения *P*[*i*] должны быть различны, поскольку никакого рабочего нельзя поставить сразу на два рабочих места. При таком представлении данных можно построить решение задачи в виде рекурсивной функции, несколько напоминающей второй вариант решения задачи о расстановке ферзей.

Другое возможное представление данных — функциональное. Все, что нам нужно знать о каждой позиции — это ее общая производительность, информация о том, свободен ли некоторый *i*-й рабочий, и способ распечатки результата. Таким образом, как и в случае задачи о расстановке ферзей, тип данных, представляющий некоторую расстановку рабочих по рабочим местам, может быть записан в виде следующего интерфейса:

```
// Расстановка рабочих в задаче о назначениях
public interface IAssignment {
  // Вычисление суммарной производительности
  double prod();
  // Проверка, свободен ли рабочий с заданным номером i
  boolean free(int i);
  // Вывод информации о расстановке рабочих в выходной поток
  void print(java.io.PrintStream);
}
```

Конечно, для того чтобы задать некоторую конкретную расстановку, надо иметь информацию, какова матрица производительности *W* и сколько всего имеется рабочих.

Решение может быть получено путем написания рекурсивной функции, которая, получая в качестве аргумента некоторую расстановку с уже сделанными *k* назначениями, пытается расставить остальных рабочих так, чтобы в результате иметь максимальную производительность. В листинге 7.4 приведена полная программа решения задачи для заданного количества рабочих (6) и заданной матрицей производительности *W*. Впрочем, в этом примере матрица такова, что решение сразу же находится невооруженным взглядом.

Листинг 7.4. Решение задачи о назначениях

```
private static int maxWorkers = 6;  // количество рабочих

// Матрица производительности
private static double[][] W = new double[][] {
  {1, 2, 3, 4, 5, 6},  // производительность рабочего номер 0
  {6, 5, 4, 3, 2, 1},  // производительность рабочего номер 1
```

```
  {2, 6, 5, 3, 4, 1},  // производительность рабочего номер 2
  {1, 2, 5, 6, 4, 3},  // производительность рабочего номер 3
  {3, 5, 6, 4, 2, 1},  // производительность рабочего номер 4
  {1, 2, 4, 5, 6, 3}   // производительность рабочего номер 5
};

// Пустая расстановка
private static IAssignment empty = new IAssignment() {
  public double prod() { return 0; }
  public boolean free(int i) { return true; }
  public void print(java.io.PrintStream ps) {}
};

// Функция recAssignment получает в качестве аргументов некоторую
// расстановку k рабочих и пытается дополнить ее расстановкой
// остальных рабочих так, чтобы получить максимальную
// производительность согласно матрице W
private static IAssignment recAssignment(
                          final IAssignment assignment,
                          final int k) {
  if (k == maxWorkers) {
    // Все рабочие уже расставлены
    return assignment;
  } else {
    // Максимальная достигнутая производительность
    double maxAchieved = 0;
    // Лучшая расстановка, при которой была достигнута
    // максимальная производительность
    IAssignment bestAchieved = null;
    // Цикл перебора оставшихся рабочих
    for (int num = 0;  num < maxWorkers;  num++) {
      if (assignment.free(num)) {
        final int fnum = num;   // фиксируем очередного работника
        // Ставим этого работника на очередное место и пытаемся
        // получить лучшую производительность при этой расстановке
        IAssignment best = recAssignment(new IAssignment() {
          public double prod() {
            return assignment.prod() + W[fnum][k];
```

```
        }
        public boolean free(int i) {
          return i != fnum && assignment.free(i);
        }
        public void print(java.io.PrintStream ps) {
          assignment.print(ps);
          ps.println("На рабочем месте номер " + k +
                     " находится рабочий номер " + fnum);
        }
      }, k+1);
      // Достигнутая производительность
      double curAchieved = best.prod();
      if (curAchieved > maxAchieved) {
        // Получили что-то лучшее, чем было
        maxAchieved = curAchieved;
        bestAchieved = best;
      }
    }
  }
  // Результат - лучшая из достигнутых расстановок
  return bestAchieved;
 }
}

public static void main(String[] args) {
 // Распечатываем лучшую расстановку
 recAssignment(empty, 0).print(System.out);
}
```

В приведенном решении программа перебирает всевозможные перестановки, и поэтому она не слишком-то эффективна. Уже при 20 рабочих дождаться, пока она вычислит лучшую расстановку, становится довольно затруднительно. Существенно лучшее решение можно получить, если заранее вычислить максимальную производительность для каждого рабочего. Тогда можно при каждом очередном назначении оценивать максимальную производительность, которой в принципе можно достигнуть при расстановке оставшихся рабочих. Если окажется, что эта максимальная производительность не лучше, чем уже достигнутая, то не стоит и стараться.

При таком подходе становится существенным порядок, в котором находятся расстановки. Если попытаться "жадным" алгоритмом сразу же найти прием-

лемую расстановку, то многие варианты будут отброшены почти сразу же. Так например, для заданной в примере матрицы производительности лучшую расстановку можно найти с первой же попытки. Эффективность работы программы можно таким образом повысить очень сильно.

Не станем приводить здесь оптимизированного решения задачи в надежде, что каждый из вас при желании сможет его восстановить по приведенным выше замечаниям.

7.4. Задача о принадлежности слова языку

Рассмотрим еще одну задачу, в которой используется функциональное представление множества слов. Задача заимствована из книги [9].

Пусть требуется проверить, принадлежит ли слово языку, заданному простой грамматикой. Грамматика будет определять язык (множество слов) с помощью двух базовых операций — операции альтернации и операции катенации. Альтернация — это просто объединение двух или более множеств слов, т. е. некоторое слово *w* принадлежит альтернации двух множеств слов *A* и *B*, если оно принадлежит хотя бы одному из множеств *A* или *B*. Катенация — это более сложная операция. Слово *w* принадлежит катенации двух множеств *A* и *B*, если его можно разбить на два подслова *alfa* и *beta* (каждое из этих слов может быть и пустым) так, что слово *alfa* принадлежит *A*, а слово *beta* принадлежит *B*. В последнем случае условимся записывать, что

$$w = alfa \cdot beta,$$

при этом новое множество слов, полученное катенацией множеств *A* и *B*, будет записываться в виде

$$A \cdot B.$$

В листинге 7.5 приведено полное решение задачи. В нем сначала представлен интерфейс `IWords`, определяющий понятие множества слов, а потом — класс `Language`, содержащий грамматические операции над этими множествами. Наиболее сложным является определение операции катенации двух множеств (`cat`). В этом определении сначала непустое исследуемое слово разбивается на два — первую букву слова и оставшуюся часть, а потом с помощью рекурсивного вызова проверяется, нельзя ли остаток слова разбить на две части таким образом, чтобы первая буква с первой частью остатка принадлежала бы первому множеству, а вторая часть остатка принадлежала бы второму множеству.

Листинг 7.5. Представление множеств слов, определяемых простой грамматикой языка

```
//=======================================================//
// Интерфейс представляет множество слов в виде
```

```
// характеристической функции множества
//=====================================================//
public interface IWords {
  // Возвращает true, тогда и только тогда, когда слово
  // принадлежит множеству
  boolean contains(String word);
}

//=====================================================//
// Класс Language определяет грамматические конструкции,
// позволяющие строить новые множества слов из уже
// имеющихся по правилам простой грамматики
//=====================================================//
public abstract class Language {
  // Функция строит объединение двух множеств слов в традиционном
  // стиле: слово принадлежит объединению, если оно принадлежит
  // хотя бы одному из членов этого объединения
  public static IWords or(final IWords w1, final IWords w2) {
    return new IWords() {
      public boolean contains(String word) {
        return w1.contains(word) || w2.contains(word);
      }
    };
  }

  // Функция строит катенацию двух множеств
  public static IWords cat(final IWords w1, final IWords w2) {
    return new IWords() {
      public boolean contains(String word) {
        // Если слово пусто, то оно может быть катенацией только
        // двух пустых слов
        if (word == null || word.equals("")) {
          return w1.contains(word) && w2.contains(word);
        }
        // Непустое слово можно представить как катенацию пустого
        // слова и себя самого
        if (w1.contains("") && w2.contains(word)) {
          return true;
        }
```

```
        // Слово можно представить как катенацию первой буквы...
        final String first = word.substring(0, 1);
        // ... и остатка слова
        final String last = word.substring(1);
        // В этом случае остаток попытаемся представить как катенацию
        // двух слов, причем первая его часть должна при присоединении
        // к ней первой буквы исходного слова принадлежать множеству w1,
        // а вторая часть — множеству w2
        return cat(
          new IWords() {
            public boolean contains(String word) {
              return w1.contains(first + word);
            }
          }, w2
        ).contains(last);
      }
    };
  }
}
```

Теперь можно строить различные языки (множества слов) и проверять, принадлежит ли некоторое слово множеству. Зададим, например, сначала в качестве базовых два множества — множество гласных букв и множество согласных букв (точнее, в обоих случаях определяются множества однобуквенных слов). Такие множества можно указать напрямую с помощью их характеристических функций.

```
// Множество слов, представленных одной гласной буквой
// В этом примере гласными считаются буквы 'a', 'o' и 'e'
private static IWords vow = new IWords() {
  public boolean contains(String word) {
    return word != null &&
           (word.equals("a") ||
            word.equals("e") ||
            word.equals("o"));
  }
};

// Множество слов, представленных одной согласной буквой
// В этом примере согласными считаются буквы 'b', 'c' и 'd'
```

```
private static IWords cont = new IWords() {
  public boolean contains(String word) {
    return word != null &&
           (word.equals("b") ||
            word.equals("c") ||
            word.equals("d"));
  }
};
```

Теперь на их базе можно определить более сложные множества, например, множество открытых слогов и множество слов, составленных из этих открытых слогов

```
// Множество слов, представляющих открытый слог английского языка
// В этом примере считаем, что открытый слог состоит из одной согласной
// и одной или двух гласных букв
private static IWords part = Language.or(
  Language.cat(cont, vow),
  Language.cat(cont, Language.cat(vow, vow))
);

// Множество правильных слов. В этом примере считаем, что правильное
// слово состоит из одного или двух открытых слогов
private static IWords word = Language.or(
  part, Language.cat(part, part)
);
```

Проверка принадлежности слова множеству осуществляется путем вызова метода `contains` множества.

```
// Проверим на "правильность" несколько слов
System.out.println(word.contains("do"));     // правильное слово
System.out.println(word.contains("bee"));    // правильное слово
System.out.println(word.contains("bead"));   // неправильное слово
System.out.println(word.contains("coca"));   // правильное слово
System.out.println(word.contains("cola"));   // неправильное слово
System.out.println(word.contains("boodoo")); // правильное слово
```

В этом примере слово `"bead"` оказывается неправильным, поскольку не состоит из открытых слогов, а слово `"cola"` неправильное, поскольку с точки зрения программы оно содержит "неправильную" букву.

Надо сказать, что наша программа ни в коей мере не является универсальной и, несмотря на кажущуюся "всеядность", применима далеко не к каж-

дой грамматике. Так, например, оказывается невозможным определить весьма естественную с точки зрения грамматик операцию над множествами слов — итерацию множества. Попытка определить ее в классе `Language` в следующем виде

```
public abstract class Language {
  ...
  public static IWords iteration(IWords w) {
    return or(w, cat(w, iteration(w)));
  }
}
```

приводит к немедленному "зацикливанию" при любом обращении к функции `iteration`.

Это, впрочем, совершенно очевидно, поскольку в определении функции содержится прямое обращение к самой определяемой функции, не связанное никакими условиями (это, конечно, пример "дурной" рекурсии). Но эту операцию не удается запрограммировать и с помощью более "тонких" способов. В итоге, программа будет постоянно пытаться разложить заданное слово на катенацию бесконечного количества пустых слов в тщетной попытке проверить, не состоит ли слово из одних только пустых слов. А без операции итерации построить хоть сколько-нибудь полезную грамматику невозможно.

Несколько улучшить ситуацию можно, если отказаться вообще от возможности включать пустые слова в множества слов. Тогда все разбиения слов оказываются конечными, и за счет этого удается определить более богатые грамматики.

7.5. Задача о построении фигур

Последняя задача этой главы, также заимствованная из книги [9], состоит в следующем. Пусть имеются объекты, главным назначением которых служит возможность изображения их на некотором графическом устройстве (например, экране монитора или графопостроителе). Мы не будем определять в подробностях такие базовые функции, однако определим соответствующий интерфейс для них:

```
public interface Figure {
  void draw(int x, int y);
}
```

Метод `draw` задает основную функцию рисования фигуры. Параметры `x` и `y` определяют координаты "опорной точки". Для разных фигур такой опорной точкой могут служить разные точки фигуры, например, для окружности это

может быть ее центр, для прямоугольника — координаты одного из его углов, и т. д.

Теперь, как и раньше, основная задача будет состоять в определении набора операций, которые позволят строить сложные фигуры из более простых. Для этого определим класс `Drawing`, в котором и зададим все необходимые операции. Собственно, таких операций будет немного — сдвиг фигуры по горизонтали или вертикали на некоторое количество точек и объединение фигур. Используя эти операции над фигурами, можно получать достаточно сложные узоры даже из весьма простых фигур. Очень интересные узоры можно получить, если добавить к операциям над фигурами несколько более сложных операций, таких как, например, размещение нескольких экземпляров фигуры вдоль окружности с заданными координатами центра и радиусом.

Мы, впрочем, ограничимся программированием только самых простых операций. Определение класса `Drawing` приведено в листинге 7.6. Там же приведено описание класса, задающее конкретную простую фигуру — прямоугольник. Мы не приводим описание его функции `draw`, поскольку оно, конечно же, зависит от того, где и как рисуются наши фигуры.

Листинг 7.6. Определение простых операций над фигурами

```
//=============================================================
// Класс Drawing определяет несколько простых функций
// для построения новых фигур на базе уже имеющихся
//=============================================================
public abstract class Drawing {
  // Метод shiftX строит новую фигуру путем сдвига старой
  // на shift единиц по горизонтали
  public static Figure shiftX(final Figure f, final int shift) {
    return new Figure() {
      public void draw(int x, int y) { f.draw(x + shift, y); }
    };
  }

  // Метод shiftY строит новую фигуру путем сдвига старой
  // на shift единиц по вертикали
  public static Figure shiftY(final Figure f, final int shift) {
    return new Figure() {
      public void draw(int x, int y) { f.draw(x, y + shift); }
    };
  }
```

```
  // Метод join строит фигуру как объединение двух старых
  public static Figure join(final Figure f1, final Figure f2) {
    return new Figure() {
      public void draw(int x, int y) { f1.draw(x, y); f2.draw(x, y); }
    };
  }
}

//=============================================================
// Класс Rectangle определяет конкретную фигуру - прямоугольник
//=============================================================
public class Rectangle implements Figure {
  private int baseX;   // координаты левого верхнего угла
  private int baseY;
  private int sizeX;   // размеры прямоугольника
  private int sizeY;

  // Конструктор задает базовые величины для определения прямоугольника
  public Rectangle(int baseX, int baseY, int sizeX, int sizeY) {
    this.baseX = baseX;
    this.baseY = baseY;
    this.sizeX = sizeX;
    this.sizeY = sizeY;
  }

  // Функции доступа нужны для определения позиции
  // и размеров прямоугольника
  public int getBaseX() { return baseX; }

  public int getBaseY() { return baseY; }

  public int getSizeX() { return sizeX; }

  public int getSizeY() { return sizeY; }

  // Основная функция рисования прямоугольника
  public void draw(int x, int y) {
    /* Здесь должно быть определение функции рисования */
  }
}
```

Теперь, имея определение прямоугольника, можно взять конкретный прямоугольник и на его основе построить другие фигуры. Вот как, например, выглядит функция построения фигуры, состоящей из некоторого количества других фигур, выстроенных в один ряд по горизонтали и расположенных в этом ряду с заданным смещением друг относительно друга.

```
// Горизонтальный ряд из фигур:
//    fig - базовая фигура;
//    number - количество фигур в ряду;
//    shiftX - расстояние между опорными точками фигур в ряду
private static Figure row(Figure fig, int number, int shiftX) {
  return (number == 1 ?
              fig :
              Drawing.join(fig,
                  Drawing.shiftX(row(fig, number-1, shiftX),
                      shiftX)));
}
```

Если имеется некоторая базовая фигура, например такая, как на рис. 7.1, *а*, то ряд, построенный из подобных фигур, будет выглядеть, как показано на рис. 7.1, *б*.

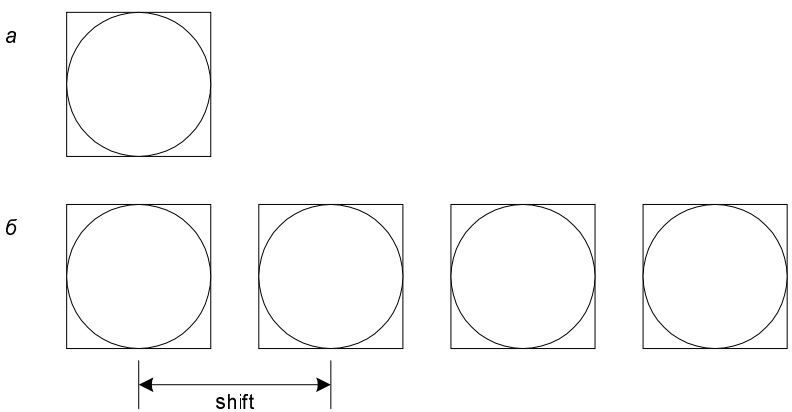


Рис. 7.1. Базовая фигура и ряд, построенный из базовых фигур

Аналогично строится и "колонка" из базовых фигур:

```
// Вертикальный ряд из фигур:
//    fig - базовая фигура;
//    number - количество фигур в колонке;
//    shiftY - расстояние между опорными точками фигур в колонке
```

```
private static Figure column(Figure fig, int number, int shiftY) {
  return (number == 1 ?
              fig :
              Drawing.join(fig,
                  Drawing.shiftY(column(fig, number-1, shiftY),
                      shiftY)));
}
```

Теперь можно построить фигуру, лежащую в основе построения "кирпичной стенки" и состоящую из двух рядов базовых фигур, находящихся друг под другом с некоторым смещением. Эта фигура будет выглядеть так, как показано на рис. 7.2.

```
// Фигура, построенная из двух рядов базовых фигур,
// лежащих друг над другом, причем нижний ряд сдвинут относительно
// верхнего на половину величины shiftX:
//    fig - базовая фигура;
//    number - количество фигур в ряду;
//    shiftX - расстояние между опорными точками фигур в ряду
//    shiftY - расстояние между опорными линиями рядов
private static Figure twoRows(Figure basic,
                              int number, int shiftX, int shiftY) {
  Figure row1 = row(basic, number, shiftX);
  return Drawing.join(row1,
             Drawing.shiftY(
                 Drawing.shiftX(row1,
                     shiftX / 2), shiftY));
}
```

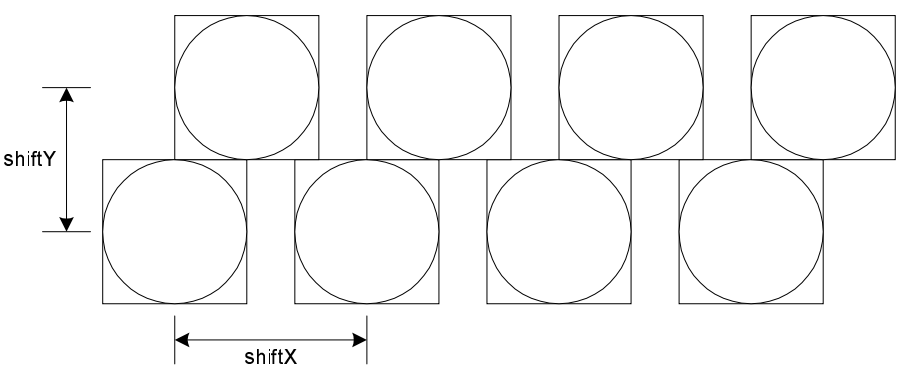


Рис. 7.2. Два ряда из базовых фигур со сдвигом друг относительно друга

Наконец, "кирпичная стенка" строится из прямоугольных "кирпичей" и состоит из четного количества рядов, в каждом из которых находится некоторое количество базовых фигур.

```
// Построение "кирпичной стенки" из прямоугольников
//     brick - базовый "кирпич";
//     inRow — количество "кирпичей" в ряду;
//     inColumn — количество рядов в "стенке"
private static Figure wall(Rectangle brick, int inRow, int inColumn) {
  return column(twoRows(brick, inRow,
                        brick.getSizeX(), brick.getSizeY()),
                inColumn / 2,
                2 * brick.getSizeY());
}
```

Если задать базовый "кирпич" с помощью описания

```
private static Rectangle brick = new Rectangle(0, 0, 100, 50);
```

то стенку из таких кирпичей можно создать и нарисовать с помощью вызова

```
wall(brick, 16, 10).draw(0, 0);
```

Возможно, получится что-то, похожее на рис. 7.3, но, впрочем, реальная "картинка" зависит только от того, как определены базовые функции рисования.

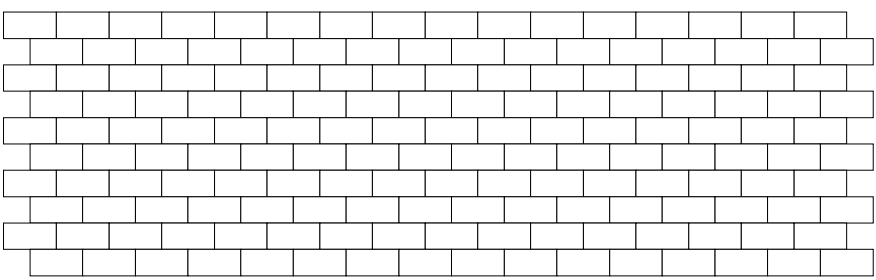


Рис. 7.3. "Кирпичная стенка", полученная с помощью функции `wall`

В приведенном выше примере интересно, что базовые фигуры (прямоугольники, возможно, другие фигуры) имеют вполне традиционное представление, при котором объекты строятся из некоторых более простых объектов. Так, например, прямоугольник содержит в качестве элементов координаты верхнего левого угла и размеры. Однако, поскольку такой прямоугольник обладает функциональной возможностью "быть нарисованным", то он может использоваться в функциях, построенных на основе только этой функцио-

нальной информации. В итоге мы строим объекты, которые сами по себе не содержат фактической информации (у построенной нами "кирпичной стенки" нет ни опорной точки, ни размеров), однако имеют некоторое функциональное представление, с помощью которого мы можем манипулировать ими (строить сложные объекты на базе более простых и, конечно же, рисовать эти объекты).

На этом мы заканчиваем главу о функциональном представлении объектов, а вместе с нею заканчивается и вся книга.

Заключение

Мы рассмотрели много разных структур данных и привели массу алгоритмов их обработки. Большинство из этих алгоритмов и способов представления данных являются абсолютно необходимыми инструментами в работе каждого программиста. Однако достаточно ли знать их, чтобы считать себя опытным программистом, готовым разрабатывать любые новые алгоритмы и программы? К сожалению, нет. Практически каждая программистская задача требует для своего решения создания новых структур данных и разработки новых алгоритмов.

Тогда, может быть, изучать предлагаемые алгоритмы и структуры данных по книге вовсе не стоит? Да нет, конечно, же, стоит! Базовые структуры данных служат основой для разработки новых специализированных структур данных. Разумеется, невозможно написать сложную программу, не имея представления о массивах. Точно так же невозможно написать сложную программу, не зная алгоритмов обработки списков, не умея обходить деревья или не представляя способы хэширования информации.

Конечно, с развитием языков программирования все больше и больше средств и методов программирования включаются либо в сам язык, либо в библиотеки классов для этого языка. Java является хорошей иллюстрацией этому, поскольку имеет в своем составе не только такие стандартные структуры данных, как массивы, не только такие известные классы, как списки или потоки ввода/вывода, но и совсем новые средства — итераторы, адаптеры, хэш-таблицы и многое другое. Конечно, можно пользоваться перечисленными средствами, даже не очень четко представляя, что стоит за каждым из этих понятий. Однако эффективность работы программиста будет существенно выше, если он сможет правильно выбрать средства программирования, а это возможно только в том случае, если он делает выбор осознанно, не только на основании сведений о внешнем поведении объектов, но и зная их работу "изнутри".

Есть еще один вопрос: а так ли уж важно добиваться максимальной эффективности функционирования программы? Производительность компьютеров постоянно повышается, объемы доступной памяти растут. Сейчас возможно написание таких программ, которые никогда бы не смогли выполняться на машинах еще 10-летней давности. Поэтому, может быть, не стоит так уж сильно стремиться к эффективности работы программ, а следует больше заботиться об эффективности работы программистов?

Отчасти, это, конечно, справедливо. Труд программиста дорог, и повышение эффективности его работы является насущной задачей. Однако знание

основных принципов построения структур данных, алгоритмов их обработки являются одним из основных залогов успешной и эффективной работы программиста! Кроме того, не следует думать, что оптимизация программ — дело далекого прошлого. Все время появляются не только более мощные персональные и стационарные компьютеры, но и микрокомпьютеры, встраиваемые во всевозможные приборы, начиная от сотового телефона и кончая холодильником или кроватью. Такие микрокомпьютеры имеют весьма ограниченные ресурсы памяти и невысокое быстродействие процессора, так что вопрос эффективности работающих на них программ стоит чрезвычайно остро.

Надеемся, что эта книга помогла вам глубже понять, что представляет собой труд программиста, научиться правильно выбирать структуры данных для своих программ, обогатила знаниями современных технологий их обработки. И все же, к сожалению, ни одна книга не может превратить заурядного программиста в хорошего и грамотного специалиста, если только он сам не будет постоянно совершенствовать свои навыки практическим путем и пополнять знания.

Автор не представляет лучшего способа научить программированию, чем показывать и объяснять хорошие программы, и не знает другого способа стать хорошим программистом, чем постоянно писать все новые и новые программы, пытаясь применить в них свои знания. Вот это, наверное, то, что хотелось сказать в заключение этой книги.

Будьте хорошими программистами, это так интересно!

Литература

1. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979.
2. Вирт Н. Алгоритмы + структуры данных = программы. — М.: Мир, 1985.
3. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2001.
4. Дмитриева М. В., Кубенский А. А. Турбо Паскаль и Турбо Си: Построение и обработка структур данных: Учебн. пособие. — СПб.: СПбУ, 1996.
5. Кнут Д. Искусство программирования. Т. 1. Основные алгоритмы / 3-е издание. Серия: Искусство программирования. — Киев: Вильямс, 2000.
6. Кнут Д. Искусство программирования. Т. 3. Сортировка и поиск / 2-е издание. Серия: Искусство программирования. — Киев: Вильямс, 2000.
7. Кормен Е., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 2000.
8. Рейнгольд Э., Нивергельд Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. — М.: Мир, 1980.
9. Хендерсон П. Функциональное программирование. Применение и реализация. — М.: Мир, 1983.

Предметный указатель

З

И

К

М

О

С

Т

У

Ф

Х