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.

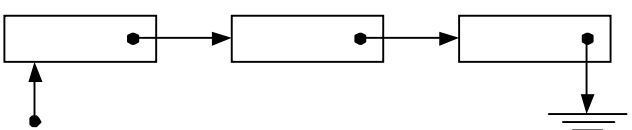

Рис. 1.1. Структура связи элементов списка

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).

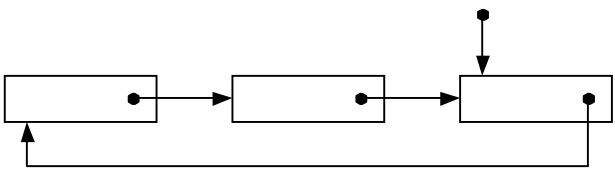

Рис. 1.2. Структура кольцевого списка

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.

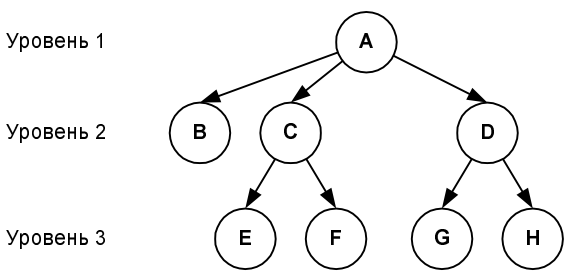

Рис. 1.3. Структура дерева

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;
  }
}
```