А. А. Кубенский

СТРУКТУРЫ и АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

объектно-ориентированный подход и реализация на C++

- *Сложные структуры данных: стеки, деревья, графы*
- *Современные технологии работы с данными*
- *Выбор эффективных решений*
- *Нетрадиционные способы представления данных*
- *Примеры программ*

+CD

УЧЕБНОЕ ПОСОБИЕ

bhv

А. А. Кубенский

СТРУКТУРЫ и АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

объектно-ориентированный подход и реализация на C++

Допущено учебно-методическим объединением
на базе Санкт-Петербургского государственного университета
Министерства образования Российской Федерации в качестве
учебного пособия по специальности "Математическое обеспечение
и администрирование информационных систем" — 351500

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.068+800.92С++
ББК 32.973.26-018.1я73
К88

Кубенский А. А.

К88 Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на С++. — СПб.: БХВ-Петербург, 2004. — 464 с.: ил.

ISBN 5-94157-506-8

Описываются методы построения и использования сложных структур данных: стеки, деревья, графы; нетрадиционные представления данных, в частности функциональное представление. Рассматриваются различные алгоритмы обработки этих структур на простых примерах программ. Изложение осуществляется на основе объектно-ориентированного подхода с использованием языка программирования С++. Показано, как тот или иной выбор решения задач влияет на эффективность и выразительность программ. Приводится большое количество текстов программ, иллюстрирующих рассматриваемые алгоритмы.

Компакт-диск, прилагаемый к книге, содержит свободно распространяемый компилятор языка С++ (лицензия GNU) и примеры программ из книги с техническими подробностями, опущенными в тексте.

Для программистов

УДК 681.3.068+800.92С++
ББК 32.973.26-018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Наталья Бубнова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 21.09.04.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 37,41.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-506-8

Содержание

Введение

Эта книга предназначена для тех читателей, кто уже умеет писать программы на одном (или нескольких) языке программирования, но пока имеет малый опыт использования сложных структур данных, и хочет узнать, как эффективно использовать различные структуры данных, как правильно организовать структуру классов при объектно-ориентированном подходе к программированию. В общем, это книга для тех, кто хочет повысить свою программистскую квалификацию, приобретая 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 "инструментов мастера", с помощью которых истинные мастера программирования с легкостью решают довольно сложные задачи. Таким образом, часто достаточно лишь выбрать одно из известных решений. Вопрос только в том, чтобы знать эти решения. Однако, к сожалению, книг, посвященных решениям традиционных задач проектирования программ и собственно программирования, не так уж и много. Подобные книги пользуются большим спросом и составляют золотой фонд книг по программированию. Достаточно назвать лишь такие монографии, как [1], [2], [3], [5], [7], чтобы понять, насколько высоко ценятся подобные 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 структу-

рами данных и отобранные по принципу личных пристрастий самого автора. Тем не менее надеюсь, что эта книга поможет начинающим программистам ощутить вкус к программированию и приучить к поиску элегантных решений задач. Некоторые из приводимых решений можно сразу же брать в качестве готовых образцов программ и их фрагментов. Другие лишь демонстрируют некоторый подход к построению программ, и сами по себе служат только примерами конкретных решений.

Основой для построения программ в данной книге служит *объектно-ориентированный подход* (ООП) к проектированию и программированию. В настоящее время техника ООП широко используется для создания как больших проектов, так и сравнительно небольших программ. Поддержка ООП фактически является обязательной при создании систем программирования даже для тех языков, которые первоначально не предполагалось использовать для объектно-ориентированного программирования. Ярким примером может служить язык Паскаль, который стал коммерческим языком программирования после того, как в него были добавлены средства ООП.

В качестве инструментального языка программирования, на котором записываются все примеры, выбран язык C++. Выбор этого языка автором обусловлен несколькими причинами. Во-первых, язык широко используется для создания программ самого разного назначения, поэтому знание этого языка фактически является обязательным для программиста любого уровня. Во-вторых, язык обеспечивает хорошую поддержку ООП, что дает возможность ясно выражать на нем достаточно сложные конструкции при построении объектно-ориентированных программ. Средства описания шаблонов, использующиеся во всех последних версиях языка, также оказывают существенную поддержку при построении типов данных, которые впоследствии могут применяться в различных программах. Наконец, язык C++ в последнее время достаточно часто используется в программистской литературе в качестве языка для детального описания алгоритмов.

Автор предполагает, что читатели в достаточной степени владеют языком, чтобы понимать записанные на нем алгоритмы. В частности, без всяких дополнительных пояснений используется аппарат описания шаблонов, наследования классов, виртуальных функций и т. п. Не предполагается, однако, знания никаких библиотек, поддерживаемых различными системами программирования на C++. Правда, в тексте книги иногда встречаются ссылки на библиотеку стандартных шаблонов STL, однако в книге используются только самые простые и очевидные шаблоны из этой библиотеки, так что уровень знакомства с библиотекой никак не может сказаться на понимании основного текста. Не используются в книге и средства отображения графической информации и организации диалога с пользователем. Все вопросы взаимодействия со средой лежат за пределами этой книги; самое большее, что исполь-

зуется здесь — это вывод информации в стандартный выходной поток средствами пакета `iostream`.

В книге много 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. Тем не менее книга не является учебником по языку программирования, хотя и может оказаться полезной в качестве источника примеров программ на C++. В помощь тем, кто захочет самостоятельно модифицировать и запускать программы, приведенные в книге, к ней приложен компакт-диск, содержащий практически все тексты программ, иногда даже несколько расширенные по сравнению с текстами книги. Все программы, записанные на компакт-диск, готовы к немедленной компиляции и исполнению. Если у читателей не установлена никакая система программирования на C++, то они могут установить такую систему непосредственно с компакт-диска. Подробные инструкции по установке и использованию системы можно найти в корневом каталоге диска в файле `Readme.txt`. Там же описана структура и содержание папок (каталогов) этого диска.

Книга состоит из 8 глав. Условно ее можно разделить на 2 основные части. В первой части, состоящей из двух глав, приведено описание базовых структур данных и алгоритмов; во второй, состоящей из 6 глав, — описанные структуры данных и алгоритмы применяются для решения различных задач и описания других более сложных алгоритмов.

В *главе 1* книги вводятся структуры данных, используемые затем на протяжении всего дальнейшего изложения. Описаны способы представления в языках программирования 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. *Разд. 2.2* этой главы содержит также описание некоторых алгоритмов сортировки массивов.

В *главе 3* вводится еще один из основных типов данных — строка. Приводятся примеры 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.

Глава 5 посвящена системам распределения памяти. Алгоритмы распределения памяти не только могут служить хорошим примером использования раз-

нообразных структур данных, но и сами по себе имеют важное прикладное значение. Подробно объяснено, как использовать собственную систему распределения и управления памятью для хранения объектов, создаваемых в программах на C++.

В *главе 6* приведены примеры некоторых классических алгоритмов обработки графов. Опять, 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.

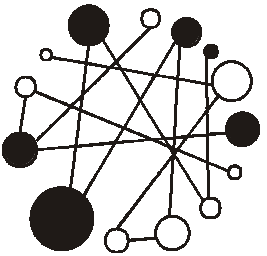
Глава 7 посвящена технологии построения программ, в которой основу программы составляют объекты, обменивающиеся между собой сообщениями. Обычно такая технология применяется только для построения очень больших программ, таких как операционные системы, системы программирования и т. п. Достаточно назвать в качестве примера операционную систему MS Windows. Тем не менее в этой главе приведены примеры достаточно простых программ, в которых применение этого подхода также может быть оправдано, хотя, конечно, существуют и более простые способы решения приводимых в этой главе задач.

Наконец, в *главе 8* приводится один из нетрадиционных способов представления информации — функциональное представление. Такое представление чаще всего используется в функциональном программировании, однако даже в традиционных императивных языках можно использовать некоторые из подходов и приемов функционального программирования. Часто это дает возможность получать необыкновенно короткие и изящные программы для решения сложных задач. Опять для приведенных в этой главе задач, наверное, можно и даже лучше использовать другие подходы, однако я хочу не столько продемонстрировать готовое решение, сколько подход, который может использоваться в более сложных случаях.

Примеры программ, содержащиеся на приложенном компакт-диске, также разбиты на главы в соответствии со структурой книги. Так, например, все примеры из *разд. 3.3* третьей главы содержатся на диске в папке `"\Programs\Chapter3\3.3"`. Иногда при этом несколько примеров физически находятся в составе одной исполняемой программы, однако чаще всего каждый пример содержится в своей собственной папке и может компилироваться и исполняться полностью независимо от других.

Автор благодарен Марине Валерьевне Дмитриевой, с которой вместе в издательстве Санкт-Петербургского университета 8 лет назад был подготовлен и выпущен первый (сильно отличающийся от этого) вариант книги, а также своему научному руководителю Святославу Сергеевичу Лаврову, который принимал активное участие в подготовке той первой книги, и издательству "БХВ-Петербург" за предоставленную возможность издания этой 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 и др. Конечно, если вы уже программировали на C++, то наверняка пользовались массивами; вероятнее всего, представление списков и деревьев тоже не будет для вас чем-то абсолютно незнакомым. Цель этой главы — не только напомнить известные структуры данных, но и продемонстрировать авторский подход к организации структур данных.

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. Этому может быть множество причин. Наиболее частой из них является соображение эффективности рабочей программы. Действительно, стандартные структуры данных, спроектированные на все случаи жизни, могут требовать излишне много ресурсов, использовать общие, долго работающие алгоритмы и т. д. Еще одна часто встречающаяся причина — это желание включить в создаваемый класс специфические методы, приспособленные для решения конкретной задачи.

Не претендуя на универсальность, мы рассмотрим несколько базовых структур и основные подходы к их реализации в программных проектах. Некоторые из описанных структур будут использоваться в следующих главах практически так, как они описаны в этой главе, однако в большинстве случаев придется вносить в эти структуры определенные изменения, обусловленные спецификой решаемой задачи или желанием автора показать новый подход к описанию и обработке данных.

1.1. Массивы

Массивы данных широко используются в языках программирования для представления объектов, состоящих из определенного числа компонентов (элементов массива) одного и того же типа (класса). Над массивами определена одна базовая операция — индексация массива. Очень часто одной этой базовой операции оказывается достаточно для решения задачи.

Пусть, например, решается задача кодирования текста, в которой необходимо каждую букву текста представить некоторым целочисленным кодом. (Заметим, что задача не имеет отношения к шифровке шпионских донесений или обеспечению режима секретности. Скорее, она применима к "переводу" текста, составленного в одной системе кодирования, в другую.) Если выразиться более точно, то 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 будет сопоставлен его целочисленный код. Если считать, что коды исходных символов лежат в диапазоне, определенном типом **char**, то кодирующая таблица может быть представлена следующим описанием:

```
char codeTable[] = { ... };  // элементы не показаны
```

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

```
void doCode (char * source, char * dest) {
  for (int i = 0; source[i]; i++) {
    dest[i] = codeTable[source[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 в этом случае пользоваться кодовой таблицей так, как показано выше, становится неудобно: либо в кодовой таблице оказывается много незадействованных элементов, либо индексацию приходится делать со смещением. Основное неудобство при этом состоит в том, что представление кодовой таблицы начинает влиять на те фрагменты программ, которые, вообще говоря, не должны зависеть от деталей ее представления.

Не очень подходит простой массив и для таких обычных ситуаций в работе, как неправильно заданный индекс (здесь надо генерировать подходящее в каждом отдельном случае исключение), динамическое добавление или уда-

ление элементов (при увеличении длины массива придется заново заказывать память под его элементы) и т. д. Во всех подобных ситуациях лучше представлять массивы с помощью объектов специально спроектированного класса, подходящего для целей нашей задачи. Можно сказать, что во всех случаях, когда массив используется не просто как временное хранилище объектов, а представляет собой структуру, заслуживающую отдельного названия (кодовая таблица в вышеописанной задаче), целесообразно описать отдельный класс, погрузив в него сам массив и скрыв детали реализации в таком классе.

Итак, пусть в программе необходимо использовать кодовую таблицу, содержащую коды символов в диапазоне от `low` до `high`. В листинге 1.1 приведен класс `CodeTable` для описания такой кодовой таблицы. Упомянутые выше детали реализации в данном случае — это способ индексации таблицы и генерация исключительных ситуаций в случае неправильного задания индекса. Массив `array`, который собственно и будет содержать элементы кодовой таблицы, представлен в классе `CodeTable` в виде атрибута класса. Дополнительными атрибутами класса являются нижний и верхний предельные индексы таблицы. Операции над массивом заменены в данной реализации методами класса `CodeTable`. Собственно говоря, как и для любого массива, метод только один — индексация таблицы. Кроме того, как обычно, в классе описываются конструкторы и деструктор кодовой таблицы.

В качестве исключительной ситуации при задании неправильных индексов будем использовать стандартный класс библиотеки STL `out_of_range`, однако текст сообщения будем генерировать, исходя из каждой конкретной ситуации. В тексте класса также используется идентификатор `byte` как сокращение для **`unsigned char`**. В этом листинге и везде далее в книге содержатся тексты сразу нескольких файлов. Обычно это заголовочный файл и файл реализации класса или нескольких классов. Тексты, принадлежащие разным файлам, отделены друг от друга комментариями, содержащими имена соответствующих файлов.

В этом и во всех последующих листингах в книге комментарии сделаны на русском, однако все программные строки приведены на английском языке. Это позволит получать осмысленные результаты работы программы при использовании любого компилятора и операционной среды. Английский язык используется также при составлении всех наименований (идентификаторов), применяющихся в программах. Конечно, выбор именно английского языка в качестве языка представления результатов может создать некоторые (надеемся, не очень большие) трудности для читателей, слабо знакомых с этим языком, однако это в любом случае не должно мешать пониманию сути работы программы. Независимость от среды можно было бы сохранить также при написании идентификаторов и программных строк на русском языке в латин-

ской транслитерации, но, по мнению автора, это выглядит ужасно, и часто расшифровка подобных идентификаторов оказывается еще сложнее, чем понимание не очень знакомого английского слова.

Листинг 1.1. Определение и реализация класса `CodeTable`

```
//---------- файл codetable.h ----------------------------------------
typedef unsigned char byte;

// Определение класса:
class CodeTable {
  byte lBound;    // нижняя граница элементов
  byte hBound;    // верхняя граница элементов
  byte *array;    // собственно массив с кодами символов

  public :
  // В конструкторе задаются границы создаваемого массива
  // и, возможно, массив для начального заполнения кодовой таблицы
  CodeTable(byte low, byte high, byte* iniTable = 0);

  // Следующий конструктор - это конструктор копирования
  CodeTable(const CodeTable &src);

  // Деструктор освобождает занятую память
  ~CodeTable();

  // Реализация операции индексации как для обычного массива
  byte& operator[](byte i);
};   // конец определения класса CodeTable

//---------- файл codetable.cpp --------------------------------------
// Реализация операций класса
CodeTable::CodeTable(byte low, byte high, byte* iniTable) {
  if ((hBound = high) < (lBound = low)) {
    throw out_of_range(
          "CodeTable constructor: lower bound is higher than upper one");
  }
  // Инициализация таблицы
  array = new byte[high - low + 1];
  for (byte code = lBound;; code++) {
    array[code - lBound] = code;
    if (code == hBound) break;
  }
```

```
  if (iniTable) {
    for (byte ndx = 0; ndx <= high - lBound && iniTable[ndx]; ndx++)
      array[ndx] = iniTable[ndx];
  }
}

CodeTable::CodeTable(const CodeTable &src) {
  array = new byte [(hBound=src.hBound)-(lBound=src.lBound)+1];
  for (byte ndx = lBound;; ndx++) {
    array[ndx-lBound] = src.array[ndx-lBound];
    if (ndx == hBound) break;
  }
}

// Деструктор
CodeTable::~CodeTable() { delete[] array; }

byte& CodeTable::operator[](byte i) {
  if (i < lBound || i > hBound) {
    throw out_of_range("Index is out of range");
  }
  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:

```
void doCode(byte* source, byte* dest, CodeTable & codeTable) {
  for (int i = 0; source[i]; i++) {
    dest[i] = codeTable[source[i]];
  }
}
```

Полностью текст, содержащий определение класса `CodeTable`, а также тестовая программа для проверки его работоспособности приведены на приложенном компакт-диске в папке `"\Chapter1\1.1\CodeTable"`. Вы можете попробовать внести некоторые изменения в программу, например, расширив диапазон кодируемых значений и кодов, определив новые операции над таблицей или вообще изменив ее реализацию.

Конечно, определение класса `CodeTable` не является универсальным и, скорее всего, для решения другой подобной задачи потребуется запрограммировать другой класс. Можно попробовать написать универсальный класс, который затем можно будет использовать во многих программах, где требуется обработка массивов. Несмотря на некоторую тяжеловесность, такой класс имеет

право на существование, и многие стандартные библиотеки имеют в своем составе подобные классы. Оправданием для написания его может служить стремление убрать некоторые недостатки стандартных массивов, с которыми приходится сталкиваться чаще всего. Нестандартное решение все же может потребоваться в том случае, когда стандартные средства не имеют всех необходимых функций или слишком неэффективны для проектируемой программы.

В листинге 1.2 и в папке "\Chapter1\1.1\DynArray" компакт-диска приведен пример такого универсального класса DynArray. Этот пример показывает, как можно самому определить массив с плавающей верхней границей. Такой массив можно расширять или укорачивать с помощью метода resize, а также с помощью операции add добавлять в его конец новые элементы. Разумеется, определены конструкторы и деструктор. Для увеличения общности вместо определения класса определим шаблон с параметром Elem, задающим тип элементов массива. Поскольку определяется не класс, а шаблон классов, то определение всех операций этого шаблона помещено вместе с определением самого шаблона в один и тот же файл dynarray.h.

Листинг 1.2. Определение класса DynArray

```
//---------- файл dynarray.h - ----------------------------------------
#include <stdexcept>          // стандартные исключительные ситуации

using namespace std;

template <class Elem>
class DynArray {
  int size;        // текущий размер массива (количество элементов)
  int maxSize;     // размер отведенной памяти
  Elem *array;     // сам массив (размера maxSize)

public :
  // Конструктор нового массива. Аргументы указывают, сколько памяти
  // надо отвести под его элементы, и сколько всего памяти заказывается.
  // Еще один дополнительный аргумент содержит массив для
  // начальной инициализации элементов
  DynArray(int sz = 0, int maxSz = 0, Elem *iniArray = NULL);

  // Конструктор копирования
  DynArray(const DynArray<Elem> & a) {
    size = maxSize = 0;
    *this = a;
  }
```

```
  // Деструктор - освобождает занятую ранее память.
  ~DynArray() { delete[] array; }

  // Операция индексации
  Elem & operator [] (int i);

  // Операция поэлементного копирования
  DynArray<Elem> & operator = (const DynArray<Elem> & a);

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

  // Добавление нового элемента с возможным расширением массива
  void add(const Elem & e);

private:

  // Операция расширения массива
  void enlarge(int delta);

  // Операция уменьшения размера массива
  void shrink(int delta);
};     // Конец определения класса DynArray

// Далее следуют реализации функций-членов класса DynArray
// Реализация конструктора
template <class Elem>
DynArray<Elem>::DynArray(int sz, int maxSz, Elem *iniArray) {
  if ((size = sz) < 0) {
    throw out_of_range("DynArray constructor: negative array size");
  }
  maxSize = (maxSz < sz ? sz : maxSz);
  array = new Elem[maxSize];      // отведение памяти
  if (iniArray) {                 // копирование элементов
    for (int i = 0; i < size; i++)
      array[i] = iniArray[i];
  }
}

// Реализация операции индексации
template <class Elem>
```

```
Elem & DynArray<Elem>::operator [] (int i) {
  if (i < 0 || i >= size) {
    throw out_of_range("operator [] : array index is out of range");
  }
  return array[i];
}

// Реализация операции присваивания (поэлементного копирования)
template <class Elem>
DynArray<Elem> & DynArray<Elem>::operator = (const DynArray<Elem> & a) {
  resize(a.size - size);
  for (int i = 0; i < size; i++)
    array[i] = a.array[i];
  return *this;
}

// Реализация операции изменения текущего размера массива
template <class Elem>
void DynArray<Elem>::resize(int delta) {
  if (delta > 0) enlarge(delta);        // увеличение размера массива
  else if (delta < 0) shrink(-delta);   // уменьшение размера массива
}

// Реализация операции добавления элемента
template <class Elem>
void DynArray<Elem>::add(const Elem & e) {
  resize(1);
  array[size-1] = e;
}

// Реализация операции расширения массива
template <class Elem>
void DynArray<Elem>::enlarge(int delta) {
  if ((size += delta) > maxSize) {   // необходимо заказать новую память
    maxSize = size;
    Elem *newArray = new Elem[maxSize];
    // копируем элементы
    for (int i =0;  i < size - delta;  i++)
      newArray[i] = array[i];
    array = newArray;
  }
}
```

```
// Реализация операции уменьшения размера массива
template <class Elem>
void DynArray<Elem>::shrink(int delta) {
  size = (delta > size ? 0 : size - delta);
}
```

Поскольку класс DynArray определяет более сложный объект, чем обычный массив, то работа с объектами класса DynArray в программах обработки данных будет несколько более сложной, чем непосредственная работа со встроенными массивами. Однако для нашего примера с кодирующей таблицей функция doCode будет выглядеть практически одинаково как при использовании класса DynArray<char>, так и при использовании стандартных массивов. Единственное различие — это тип аргумента, задающего кодовую таблицу.

```
void doCode(char *source, char *dest, DynArray<char> & codeTable) {
  for (int i = 0; source[i]; i++) {
    dest[i] = codeTable[source[i]];
  }
}
```

В проверочной программе на компакт-диске работа проводится только с массивом символов, но вы можете попробовать поработать и с более сложными объектами. Обратите внимание на требования, которым должен удовлетворять класс-аргумент шаблона, чтобы работа с ним была возможной. Конечно, имеется широкое поле для возможных изменений в составе операций класса DynArray и реализации имеющихся операций.

В конце раздела заметим, что во всех системах программирования на C++ имеется богатый набор библиотек классов и шаблонов, которые, в частности, содержат различные шаблоны и классы для работы с массивами. Так, например, библиотека стандартных шаблонов и классов STL (Standard Template Library) имеет в своем составе шаблон vector, у которого имеются методы, похожие на описанные выше операции индексации и добавления элементов шаблона DynArray. Широко распространенная библиотека классов фирмы Microsoft — MFC (Microsoft Foundation Classes) также имеет шаблон для работы с массивами CArray. Имеются и другие классы и шаблоны для описания массивов и массивоподобных структур. Многие из них весьма удобны для использования, хотя, на наш взгляд, несколько тяжеловесны, и часто малоэффективны.

Как правило, для серьезной работы с массивами в больших программах все равно приходится создавать свои собственные классы. В нашем примере реализации динамического массива тоже имеется несколько весьма существенных недостатков. Например, сразу бросается в глаза, что при добавлении

элементов в массив по одному, увеличение размера массива будет происходить при каждом добавлении. Для массивов большого размера это совершенно неприемлемо, т. к. приводит не только к заказу памяти при добавлении каждого элемента, но и к копированию при этом всего содержимого массива. При реализации следовало бы учесть это и заказывать память с некоторым запасом (размер такого "запаса" может быть параметром реализации). Интересующемуся читателю предлагается расширить нашу реализацию и исследовать, насколько такая новая реализация может оказаться эффективнее приведенной в книге в различных ситуациях.

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 заметим, что физически в представлении элементов списка может и не быть никаких ссылок.

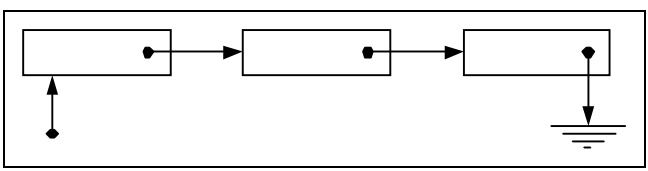


Рис. 1.1. Простейший линейный список

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

Как и в случае массивов, многие библиотеки классов включают в себя возможность описания и работы со списками (например, `CList` библиотеки классов MFC). 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 представление каждого элемента списка описывается в виде отдельного класса. В этом классе в качестве его атрибута имеется ссылка на следующий и/или предыдущий элемент. В листинге 1.3 представлено описание класса `IntList`, задающего простейший однонаправленный список из целых чисел. Элементы списка являются объектами класса `ListItem`, причем его описание вложено в описание класса `IntList`. В листинге представлены лишь некоторые наиболее характерные операции над списком: операции добавления нового элемента в начало и конец списка, операция удаления из начала списка, подсчет числа элементов в списке. Чуть более широкий набор операций содержится на компакт-диске в папке "`\Chapter1\1.2\IntList`".

Заметим, что удаление из конца списка не является типичной операцией для однонаправленных списков ввиду своей относительной сложности и неэффективности. Действительно, для удаления последнего элемента требуется иметь доступ не только к самому последнему элементу, но и к предпоследнему, а это значит, что при выполнении такой операции потребовалось бы просмотреть все элементы, возможно, длинного списка от начала до конца.

Объекты класса `IntList` содержат ссылки на первый и последний элементы списка, а также счетчик числа элементов, использующийся в функции, выдающей количество элементов списка. Этот счетчик должен модифицироваться всякий раз при добавлении или удалении элементов списка.

Мы не описываем технику работы со ссылками, т. к. считаем, что читатели владеют основами программирования на языке C++, а значит, им должны быть знакомы указатели и техника работы с ними. Но мы хотели бы обратить внимание читателей на то, что все операции по работе со списком сосредоточены внутри описания класса, так что при программировании задач обработки списков можно будет пользоваться этим классом, даже не зная о том, как именно реализованы отдельные детали в представлении списка. Это один из основных принципов объектно-ориентированного программирования — использование класса должно быть максимально независимым от его реализации.

Листинг 1.3. Определение класса `IntList`

```
//---------- файл intlist.h -----------------------------------------
class IntList {
/* Класс ListItem представляет элемент списка, связанный со
   следующим с помощью указателя, содержащегося в поле next
*/
```

```
  struct ListItem {
    int item;          // значение элемента списка
    ListItem *next;    // указатель на следующий элемент списка

    // Конструктор для создания нового элемента
    ListItem(int i, ListItem *n = NULL) { item = i; next = n; }
  };

  int count;              // счетчик числа элементов
  ListItem *first;        // первый элемент списка
  ListItem *last;         // последний элемент списка

public :

  // Конструктор по умолчанию - создание пустого списка
  IntList() { count = 0; first = last = NULL; }

  // Конструктор копирования - создание копии имеющегося списка
  IntList(const IntList & src);

  // Деструктор списка
  ~IntList();

  // Доступ к первому элементу списка
  int head() const { return first->item; }
  int & head() { return first->item; }

  // Доступ к последнему элементу списка
  int tail() const { return last->item; }
  int & tail() { return last->item; }

  // Добавить один элемент в начало списка
  void addFirst(int item);

  // Добавить один элемент в конец списка
  void addLast(int item);

  // Добавить элементы другого списка в конец этого
  void addLast(const IntList & src);

  // Удалить первый элемент
  int removeFirst();
```

```
  // количество элементов списка
  int getCount() { return count; }
};    // Конец определения класса IntList

//---------- файл intlist.cpp ----------------------------------------
#include "intlist.h"

// Реализация конструктора копирования
IntList::IntList(const IntList & src) {
  count = 0;
  first = last = NULL;
  addLast(src);   // добавляет список src в конец списка this
}

// Реализация деструктора
IntList::~IntList() {
  ListItem *current = NULL; // указатель на элемент, подлежащий удалению
  ListItem *next = first;   // указатель на следующий элемент
  while (next) {            // пока есть еще элементы в списке
    current = next;
    next = next->next;      // переход к следующему элементу
    delete current;         // освобождение памяти
  }
}

// Добавление одного элемента в начало списка
void IntList::addFirst(int item) {
  // создаем новый элемент и присоединяем его к началу списка
  ListItem *newItem = new ListItem(item, first);
  if (first == NULL) {
    // список был пуст - новый элемент будет и первым, и последним
    last = newItem;
  }
  first = newItem;
  count++;    // число элементов списка увеличилось.
}

// Добавление одного элемента в конец списка
void IntList::addLast(int item) {
  // создаем новый элемент списка
  ListItem *newItem = new ListItem(item);
  if (last == NULL) {
    // список был пуст - новый элемент будет и первым, и последним
    first = newItem;
```

```
  } else {
    // новый элемент присоединяется к последнему элементу списка
    last->next = newItem;
  }
  last = newItem;
  count++;    // число элементов списка увеличилось.
}

// Добавление элементов заданного списка в конец определяемого
void IntList::addLast(const IntList & src) {
  for (ListItem *cur = src.first; cur;  cur = cur->next)
    addLast(cur->item);  // используем добавление одного элемента
}

// Удаление первого элемента из списка
int IntList::remove() {
  int res = first->item;  // содержимое первого элемента
  first = first->next;    // второй элемент становится первым
  count--;                // число элементов списка уменьшилось
  return res;   // удаленный элемент возвращается в качестве результата
}
```

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 будет попытка обращения к объекту по пустому указателю).

Исправить 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. Это несложное упражнение может быть проделано читателями, если они захотят на практике использовать приведенный в книге пример.

Чтобы исправить первый недостаток, надо сначала понять, чего же мы, собственно, хотим, т. е. какие операции надо определить для того, чтобы можно было перебрать элементы списка. Вопрос не так прост, как это может показаться на первый взгляд, поэтому мы отложим рассмотрение методов итерации до *разд. 2.4*.

Гибкость списковых структур особенно ярко проявляется при вставке и удалении элементов. Действительно, в отличие от массива элементов, нет необ-

ходимости перемещать элементы списка, достаточно лишь заменить значения нескольких ссылочных полей. Это может значительно ускорить работу программы в случае большого количества обрабатываемых однородных объектов, 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 также предлагаются методы, позволяющие вставлять и удалять элементы списка в соответствии с некоторым критерием. Например, для описанного списка из целых чисел можно предложить операцию удаления элементов с заданным значением. Для реализации этой операции потребуется пройти вдоль всего списка, удаляя элементы, содержащие заданное значение.

В такой операции для удаления помимо указателя на текущий элемент списка придется хранить еще и указатель на предыдущий элемент (похожая техника была использована нами при реализации деструктора списка). Это необходимо, поскольку при удалении элемента корректируется ссылка, содержащаяся в предыдущем элементе списка. Ниже показана одна из возможных реализаций такой операции удаления заданного элемента. Реализованная нами операция `remove` возвращает значение **`true`**, если найденный элемент списка был удален, и **`false`** в противном случае. Заметим, что если в списке было несколько элементов с заданным значением, то удаляется лишь первый из найденных элементов.

```
bool IntList::remove(int n) {
  ListItem *pred = 0,          // указатель на предыдущий элемент
           *current = first;   // указатель на текущий элемент
  while (current) {
    if (current->item == n) {
      if (pred) {              // корректируем ссылку на удаляемый элемент
        pred->next = current->next;
      }
      if (current == last) { // удаляется последний элемент
        last = pred;         // корректируем ссылку на последний элемент
      }
      delete current;        // освобождаем память
      count--;               // уменьшаем количество элементов
      return true;
    } else {                 // переходим к следующему элементу
      pred = current;
      current = current->next;
    }
  }
```

```
  // удаляемый элемент не найден
  return false;
}
```

В качестве примера еще одной операции по добавлению 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, так что, как и для операции удаления в функции, используются два указателя на соседние элементы.

```
void IntList::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 сначала 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 производится с помощью изменения значения поля `next` в предыдущем элементе (или поля `first` списка, если предыдущего элемента нет).

Заметим, что работа всех операций с упорядоченным списком должна производиться согласованно. Например, не следует в список, образованный применением методов `addFirst` и `addLast`, вставлять затем элементы с помощью нашего нового метода `insert`. Если исходный список был неупорядоченным, то предсказать, куда в этом случае попадет новый элемент, довольно трудно.

Однако если вставка элементов всегда производилась только с помощью метода `insert`, то получившийся при этом список обязательно будет упорядочен по возрастанию элементов.

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, но нужно аккуратно следить за тем, чтобы не пропустить конец списка и не уйти в бесконечный цикл по элементам кольцевого 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 списка может использоваться, например, в алгоритмах обслуживания некоторого множества элемента в порядке очереди.

В классе, реализующем линейный список, мы описали два указателя соответственно на первый и последний элементы списка. В представлении кольцевого списка можно вообще обойтись без ссылки на первый элемент, имея только ссылку на последний. В этом случае ссылку на первый элемент можно легко извлечь из последнего элемента (рис. 1.2).

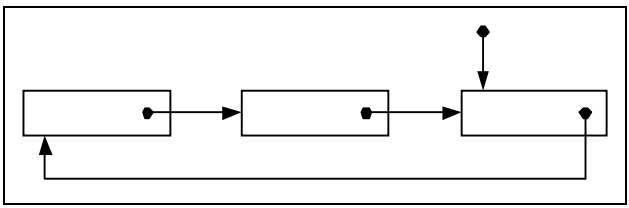


Рис. 1.2. Простейший кольцевой список

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, если вместо описания класса рассмотреть описание шаблона с параметром, задающим тип элементов списка. Разумеется, такие операции, как вставка в упорядоченный список, в самом общем случае неприменимы, однако, если класс элементов списка поддерживает операции сравнения, то даже операции с упорядоченным списком будут иметь смысл, да и реализация их будет выглядеть практически так же. В дальнейшем в подобных случаях будем использовать класс `List<Elem>` для задания списка из элементов типа `Elem`, считая, что определение соответствующего шаблона классов очевидно. Впрочем, одно такое определение приведено далее в *разд. 1.5*, где оно используется для одного из способов представления графа.

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. Целостность такого объекта также должна поддерживаться за счет описания класса, содержащего методы доступа и обработки отдельных элементов иерархической структуры.

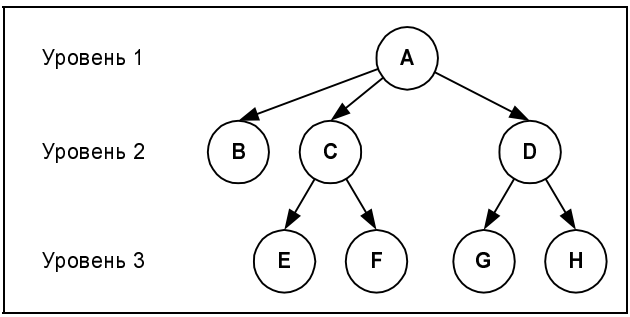


Рис. 1.3. Пример дерева

Если абстрагироваться от конкретного содержания объектов, то получится математический объект, называемый *деревом* (точнее, корневым деревом). Дадим два равносильных определения корневого дерева.

Определение 1. *Корневым деревом* называется непустое 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 поддеревьев, в свою очередь, есть дерево.

Определение 2. *Корневым деревом* называется множество элементов, называемых *узлами* этого дерева, связанных следующим отношением. Каждому из узлов, кроме одного (называемого *корнем* дерева), можно сопоставить ровно один из других имеющихся узлов, который при этом называется *непосредственным предком* этого узла. Корень дерева — это единственный узел, не имеющий непосредственного предка.

Оба определения фактически задают один и тот же объект. Существенная разница между ними состоит в том, что первое определение рекурсивно, т. е. в нем содержится ссылка на определяемое понятие (поддеревья — это де-

ревья), а второе определение — нерекурсивно. Связь между этими определениями можно установить, если понять, что каждый узел является непосредственным предком корней поддеревьев этого узла. Если узел A является непосредственным предком узла B, то узел B называют непосредственным потомком узла A.

При представлении в памяти компьютера элементы дерева (узлы) связывают между собой таким образом, чтобы каждый узел был связан со своим непосредственным предком и/или своими непосредственными потомками. Наиболее распространенными способами представления дерева являются следующие три (или их комбинации).

При первом способе каждый узел (кроме корня) содержит указатель на узел, являющийся его непосредственным предком, 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, отслеживая 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 при этом нет возможности получать независимо ссылки на узлы дерева. Тем не менее такое представление дерева иногда используется в алгоритмах, где прохождение узлов всегда осуществляется в восходящем порядке. Преимуществом этого 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 общем случае не более K) непосредственных потомка. Тогда можно включить в представление узла указатели на этих потомков. В этом случае дерево называют *двоичным* (бинарным), а два поддерева каждого узла называют соответственно *левым* и *правым* поддеревьями этого узла. Разумеется, узел может иметь и 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* или *терминальным* 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 — часть зарезервированной памяти будет содержать пустые указатели.

В листинге 1.4 описан шаблон классов, определяющий структуру бинарного дерева, содержащего в узлах объекты произвольного класса `T`. Мы пока не будем определять никаких методов для этого класса, так что в основном описание шаблона сводится к описанию структуры узлов дерева. Никакой реализации шаблона также пока не представлено.

Листинг 1.4. Определение шаблона классов `Tree`

```
template <class T>
class Tree {
  // Определение класса для узла дерева
  struct Node {
    T item;         // содержимое узла
    Node *left;     // указатель на левое поддерево
    Node *right;    // указатель на правое поддерево

    // Конструктор узлов дерева:
    Node(const T & item, Node *left = NULL, Node *right = NULL) {
      Node::item = item;
      Node::left = left;
      Node::right = right;
    }
  };

  Node *root;       // Корень дерева

public :

  // Конструктор дерева задает новое пустое дерево
  Tree() { root = NULL; }

  // Деструктор дерева должен освободить память, занятую его узлами
  ~Tree();
};
```

Здесь корень дерева представлен указателем `root` на элемент класса `Node` (узел), а каждый узел, в свою очередь, содержит два указателя 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 дерева состоит в том, что каждый узел списка содержит список своих поддеревьев. При этом можно задать такой список, непосредственно используя шаблон классов `List`, так что описание структуры узла дерева в контексте описания шаблона для самого дерева могло бы выглядеть следующим образом:

```
struct Node {
  T item;                      // содержание узла
  List<Tree<T>*> subtrees;     // список поддеревьев
};
```

Однако чаще поступают несколько по-иному: в каждый узел дерева помещают два указателя, причем один из них указывает на начало списка поддеревьев, а второй служит для связывания корней поддеревьев в этот список. На рис. 1.4, *а* изображено некоторое дерево, узлы которого помечены буквами латинского алфавита. Если его узлы представлять так, как только что описано, то физически указатели в представлениях узлов в памяти будут размещаться так, как показано на рис. 1.4, *б*.

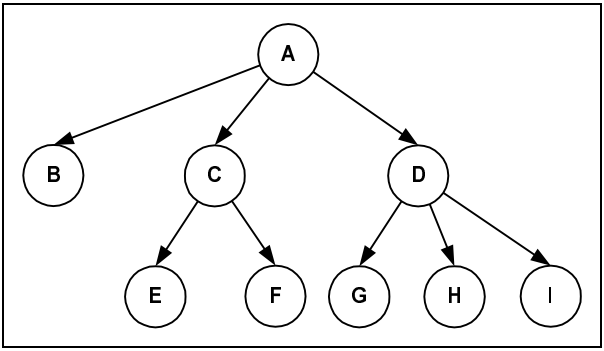


Рис. 1.4, *а*. Дерево, узлы которого помечены латинскими буквами

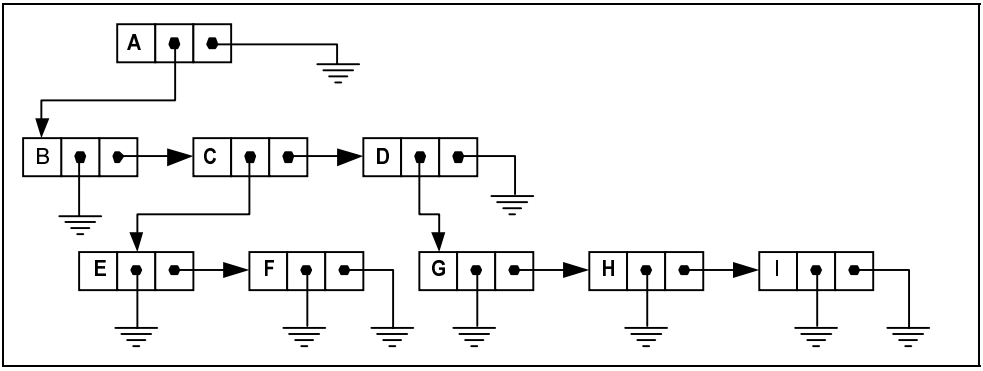


Рис. 1.4, *б*. Физическое представление такого дерева с рис. 1.4, *а* в памяти

Из рисунка хорошо видно, что представление каждого узла в точности такое же, как и у узлов бинарного дерева, которые тоже помимо смысловой нагрузки содержат два указателя на левое и правое поддеревья. Это означает, что

формально описание класса для этого представления может быть тем же самым, что и для случая бинарного дерева. Однако смысл содержащихся в этом описании указателей совершенно другой: это уже не указатели на левое и правое поддеревья, а указатели на начало списка поддеревьев и соседний узел в таком списке. При этом способе представления деревьев часто используется генеалогическая терминология. Узлы, которые мы ранее называли непосредственными потомками некоторого узла, называют также его *сыновьями*; соответственно, сыновей одного и того же узла называют *братьями* и т. д. Эту терминологию можно отразить в описании класса, и тогда поля `left` и `right` будут называться `son` и `brother` соответственно.

```
struct Node {
  T item;                // содержание узла
  Node *son, *brother;   // указатели на сына и брата
};
```

Такое определение дерева и пример работы с ним можно найти в папке "`\Chapter1\1.3\Tree`" приложенного компакт-диска.

Рекурсивную природу дерева, отраженную в одном из его определений, можно выразить более явно и в описании класса, если в описании ссылок на поддеревья вместо указателей на объекты класса `Node` использовать указатели на объекты класса `Tree`. Тогда вместо двух классов — `Node` и `Tree` — достаточно использовать всего один, и многие операции над деревом могут быть просто выражены в виде рекурсивных функций.

Определим, например, метод для вычисления высоты произвольного дерева. *Высотой* 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:

$$h(t) = \begin{cases} 0, & \text{если } t = null \\ \max\limits_{subtree}(h(t_{subtree})) + 1, & \text{если } t \neq null \end{cases},$$

где t — исходное дерево, $t_{subtree}$ — поддеревья исходного дерева, $null$ — пустое дерево.

Теперь определим шаблон класса `Tree`, в котором представление узла дерева соединено с представлением самого дерева. Объекты такого класса представляют корень непустого дерева, связанного ссылками с корнями других де-

ревьев. В листинге 1.5 представлено описание самого шаблона, а также реализации конструктора, деструктора и функции для вычисления высоты дерева.

Листинг 1.5. Рекурсивное определение дерева

```
template <class T>
class Tree {
  T item;             // содержимое корневого узла дерева
  Tree<T> *son;       // указатель на старшего сына
  Tree<T> *brother;   // указатель в список сыновей

public :

  // Конструктор корневого узла получает в качестве аргументов
  // содержимое этого узла и два указателя на дерево-сына и дерево-брата
  Tree(const T & item, Tree<T> *s = NULL, Tree<T> *b = NULL);

  // Деструктор освобождает память, занятую поддеревьями этого узла
  ~Tree();

  // Метод для вычисления высоты дерева
  int height();
};    // Конец определения класса Tree

// Далее следуют шаблонные функции, реализующие операции над деревом

// Конструктор узла дерева.
template <class T>
Tree<T>::Tree(const T & item, Tree<T> *son, Tree<T> *brother) {
  Tree<T>::item = item;
  Tree<T>::son = son;
  Tree<T>::brother = brother;
}

// Деструктор освобождает память, занятую
// поддеревьями этого узла и его братьями.
template <class T>
Tree<T>::~Tree() {
  if (son) delete son;
  if (brother) delete brother;
}
```

```
// Рекурсивная функция вычисления высоты.
template <class T>
int Tree<T>::height() {
  // В цикле вычисляются высоты всех сыновей данного узла
  // и выбирается максимальная из этих высот
  int max = 0;
  for (Tree<T> *current = son; current; current = current->brother) {
    int curHeight = current->height();
    if (curHeight > max) max = curHeight;
  }
  // К результату добавляется единица за счет самого корневого узла
  return max + 1;
}
```

Приведенное рекурсивное определение дерева и пример работы с ним также можно найти на приложенном компакт-диске в папке "\Chapter1\1.3 \RecursiveTree".

Конечно, запись операции вычисления высоты дерева в виде рекурсивной функции выглядит очень просто, однако работать с рекурсивным описанием дерева все же неудобно. В частности, не очень удобно, что дерево, не содержащее узлов (пустое дерево), представляется пустым указателем, а не объектом класса Tree, как все прочие деревья. Программы становятся более ясными, если деревья и их узлы представлены разными классами. В дальнейшем мы всегда будем пользоваться ранее приведенным представлением дерева с явным выделением структуры узла. Так, в листинге 1.6 представлено модифицированное описание бинарного дерева для представления дерева произвольной структуры (со ссылками из узлов на старшего сына и брата этого узла) и переопределен метод для вычисления высоты дерева. В примере очень хорошо видно, что изменения, которые необходимо сделать в этом методе, не очень существенны.

Листинг 1.6. Вычисление высоты дерева

```
template <class T>
class Tree {
  // Описание структуры узла со ссылками на сына и брата
  struct Node {
    T item;
    Node *son;
    Node *brother;
```

```
    Node(T i, Node *s = NULL, Node *b = NULL) {
      item = i; son = s; brother = b;
    }
  };

  Node *root;        // Корневой узел

public :

  // Конструктор, деструктор и метод для вычисления высоты дерева
  Tree() { root = NULL; }
  ~Tree() { deleteSubtree(root); }
  int height() { return height(root); }

private :

  // Вспомогательные рекурсивные функции для деструктора
  // и функции вычисления высоты дерева
  void deleteSubtree(Node *node);
  int height(Node *node);
};    // Конец определения класса Tree

// Далее следуют реализации функций-членов класса

template <class T>
void Tree<T>::deleteSubtree(Node *node) {
  if (node) {
    deleteSubtree(node->son);      // Уничтожение узлов потомков
    deleteSubtree(node->brother);  // Уничтожение узлов братьев
  }
}

template <class T>
int Tree<T>::height(Node *node) {
  if (node == NULL) return 0;
  int max = 0;
  for (Node *current = node->son; current; current = current->brother) {
    int curHeight = height(current);
    if (curHeight > max) max = curHeight;
  }
  return max + 1;
}
```

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

Иногда структура рекурсивной функции, обрабатывающей дерево, 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. Действительно, на уровне $i = 1$ существует лишь одна вершина — корень дерева, а при $i > 1$ число узлов, расположенных на i-м уровне дерева, равно сумме чисел узлов, расположенных в его поддеревьях на $(i - 1)$-м уровне.

Отсюда сразу же следует определение функции (считаем его также выполненным в контексте описания шаблона `Tree`, приведенного в листинге 1.6).

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

private :
  int nodesOnLevel(Node *node, int level) {
    // Если дерево пусто - узлов в нем нет
    if (node == NULL) return 0;
    // Если заданный уровень нулевой (или меньше), то узлов тоже нет
    if (level <= 0) return 0;
    // В остальных случаях нужная сумма складывается из следующих трех
    // значений: число узлов в поддеревьях этого узла уровня (level - 1),
    // сам узел, если требуемый уровень - первый, число узлов того же
    // уровня в поддеревьях-братьях данного узла.
    return nodesOnLevel(node->son, level-1) +
           (level == 1) +
           nodesOnLevel(node->brother, level);
  }
```

Легко убедиться, что представленная функция действительно решает поставленную задачу.

Так же, как и в случае списков, важной задачей обработки деревьев является итерация (обход) дерева. Сейчас эту задачу тоже обсуждать не будем. Заметим только, что обходы элементов дерева могут быть значительно более разнообразны, чем обходы элементов списка. В случае списков обход обычно выполняется в естественном порядке (от начала списка к концу), также можно рассмотреть еще обход элементов списка в противоположном направлении — от конца к началу. В случае дерева существует много порядков обхода

его узлов, большинство из которых имеет свое самостоятельное значение и применяется в различных алгоритмах. Подробнее эта задача обсуждается в *разд. 2.5* и некоторых последующих главах настоящей книги, а также в книгах [1, 3] и многих других.

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 следующий пример. Пусть узлы бинарного дерева пронумерованы, начиная с корня и далее вниз по иерархическим уровням, как показано на рис. 1.5.

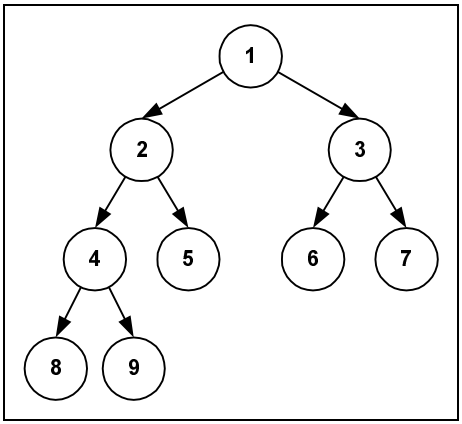


Рис. 1.5. Пронумерованное дерево

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 задается сразу же при создании дерева. Представление такого дерева описано в листинге 1.7. В нем определено дерево, обладающее следующим свойством: каждый узел этого дерева содержит целое число, меньшее, чем значения, хранящееся в поддеревьях этого узла. Таким образом, минимальное число всегда находится в корне дерева. Такое дерево иногда называют *пирамидой* или *кучей*. В *разд. 2.2* пирамида используется для одного из способов сортировки элементов массива — так называемой *пирамидальной сортировки*.

Класс `Heap` определяет такое дерево вместе с операцией `insert` добавления нового элемента в дерево. Сначала это 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` определяет для каждого узла индекс его родителя и затем определяет, надо ли продвигать новое значение дальше по направлению к вершине пирамиды. Для простоты в этом листинге опущено определение исключительной ситуации `NoMoreSpace`.

Листинг 1.7. Представление пирамиды в виде массива

```
template <class T>
class Heap {
  int count;      // Количество узлов в дереве
  T *items;       // Массив, представляющий узлы дерева
  int size;       // Размер этого массива

public :
  // В конструкторе задается размер массива; элементов пока нет
  Heap(int size) {
    count = 0;
    items = new T[Heap::size = size];
  }

  // Операция вставки нового элемента в пирамиду
  void insert(const T & element);
};

// Реализация операции вставки элемента
template <class T>
void Heap<T>::insert(const T & elem) {
    // Проверяем, есть ли еще место в массиве
    if (count >= size) throw NoMoreSpace();
    // Записываем новый элемент
    int curIndex = count++;          // индекс текущей позиции в пирамиде
    int parentIndex;                 // индекс родителя
    // Цикл "всплытия" элемента
    while (curIndex > 0 && elem < items[parentIndex = (curIndex-1)/2]) {
      // Сдвигаем элемент вниз
      items[curIndex] = items[parentIndex];
      // и переходим к элементу, лежащему выше в пирамиде
      curIndex = parentIndex;
    }
```

```
    // Помещаем новый элемент на свое место
    items[curIndex] = elem;
}
```

Подробнее о построении и анализе деревьев будет рассказано в следующих главах этой книги.

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. В множестве, напротив, порядок элементов несуществен. 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 многие типичные операции над множествами будут выполняться недопустимо долго. Так, например, проверка принадлежности элемента массиву может потребовать просмотра 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 максимальное число элементов множества не слишком велико. Тогда можно присутствие каждого из возможных элементов множества кодировать одним битом, например, единицей, если элемент присутствует в множестве, и нулем, если элемент в множестве отсутствует. Наиболее часто используются 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, содержится ли в множестве некоторый элемент. Такая битовая шкала будет содержать столько битов, сколько элементов содержится в выбранном диапазоне возможных элементов (универсальном множестве для данного проекта или задачи).

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 (занумерованных числами от 0 до 10), причем если бит содержит единицу, то это означает, что его номер содержится в множестве в качестве элемента, а если бит содержит ноль, то соответствующий элемент в множестве отсутствует. Таким образом, битовая шкала 00101100010 (биты считаются занумерованными подряд от 0 до 10) в данном случае представляет множество элементов [2, 4, 5, 9].

Приведем 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 еще 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. Битовая шкала, представляющая множество всех гласных букв, будет выглядеть тогда следующим образом:

10000100100000100001000000001011111

В языке C++ битовая шкала может быть представлена массивом 16-разрядных слов (значений типа WORD), каждый элемент которого (слово) содержит 16 битовых элементов шкалы. Биты удобно нумеровать целыми числами, начиная с нуля, причем бит номер n будет содержаться в слове с индексом n / 16 (имеется в виду операция целочисленного деления с отбрасыванием остатка). Например, приведенная выше шкала для представления множества гласных букв займет 3 слова (если бы букв всего было 32, то удалось бы поместиться в 2 слова). Учитывая, что биты в словах принято нумеровать справа налево (т. е. при изображении слова его младшие биты располагают справа), можно нарисовать ту же битовую шкалу в виде последовательности слов, представленной на рис. 1.6.

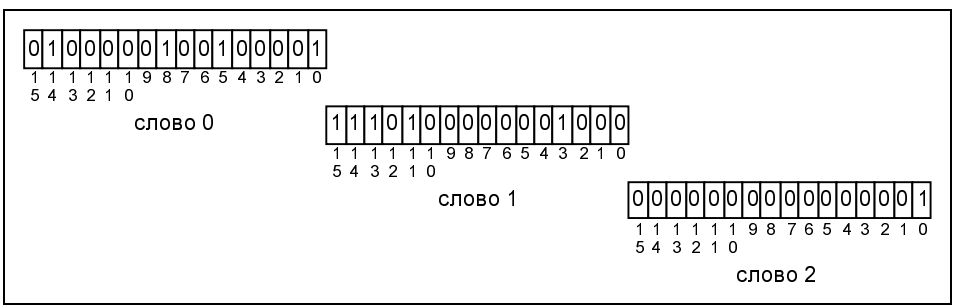


Рис. 1.6. Представление множества в виде битовой шкалы

Ниже (см. листинг 1.8, а также папку "\Chapter1\1.4\Set" компакт-диска) приводится определение класса для представления множества целых чисел из заданного диапазона. Битовая шкала в этой реализации представлена массивом слов, 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 слово с единицей, расположенной в нужном разряде. Вот, например, как в заданное слово wd можно добавить единичный бит, записанный в разряде номер n:

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

Выражение для проверки того, содержит ли заданное слово единицу в разряде номер n, будет выглядеть следующим образом:

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

Аналогичным образом будут представлены и другие операции над битовыми шкалами и отдельными битами. Операции деления нацело на 16 и получения остатка от деления на 16, необходимые для правильного вычисления адреса нужного бита, тоже можно выразить с помощью побитовых операций C++. Так, результат деления числа n на 16 можно получить с помощью выражения n >> 4, а остаток от деления на 16 — с помощью выражения n & 15.

Листинг 1.8. Определение класса Set

```
//---------- файл set.h -------------------------------------------------
class Set {

  int minElem;    // минимальный элемент диапазона
  int maxElem;    // максимальный элемент диапазона
  WORD *elems;    // битовая шкала
  int numWords;   // длина шкалы (в словах)

  friend const Set & operator | (const Set & s1, const Set & s2);
  friend const Set & operator & (const Set & s1, const Set & s2);
  friend const Set & operator - (const Set & s1, const Set & s2);
  friend const Set & operator - (const Set & s);

public:

  // Конструктор
  Set(int min = 0, int max = 255);
```

```
  // Конструктор копирования
  Set(const Set & s);

  // Деструктор
  ~Set() { delete[] elems; }

  // Проверка принадлежности элемента множеству
  bool has(int n) const;

  // Добавление элемента в множество
  Set & operator |= (int n);

  // Добавление множества в множество (объединение)
  Set & operator |= (const Set & other);

  // Пересечение множества с множеством
  Set & operator &= (const Set & other);

  // Удаление элемента из множества
  Set & operator -= (int n);

  // Удаление множества из множества (вычитание множеств)
  Set & operator -= (const Set & other);

  // Обращение (нахождение дополнения) множества
  Set & inverse();
};

//---------- файл set.cpp ------------------------------------------------
#include <stdexcept>
#include "set.h"

// Конструктор
Set::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
  numWords = (num + 15) >> 4;           // количество слов
  elems = new WORD[numWords];
```

```
  // Инициализация пустого множества
  for (int i = 0; i < numWords; i++) elems[i] = 0;
}

// Конструктор копирования
Set::Set(const Set & s) {
  minElem = s.minElem;
  maxElem = s.maxElem;
  elems = new WORD[numWords = s.numWords];
  for (int i = 0;  i < numWords;  i++) {
    elems[i] = s.elems[i];
  }
}

  // Проверка принадлежности элемента множеству
bool Set::has(int n) const {
  if (n > maxElem || n < minElem)
    return false;    // элемент находится за пределами границ множества
  int bit = n - minElem;
  return (elems[bit >> 4] & (1 << (bit & 15))) != 0;
}

  // Добавление элемента в множество
Set & Set::operator |= (int n) {
  if (n <= maxElem && n >= minElem) {
    int bit = n - minElem;
    elems[bit >> 4] |= (1 << (bit & 15));
  } else {
    throw out_of_range("Cannot add an element: it is out of range");
  }
  return *this;
}

  // Добавление элементов другого множества в данное (объединение)
Set & Set::operator |= (const Set & other) {
  if (other.minElem != minElem || other.maxElem != maxElem) {
    throw out_of_range("Sets incomparable");
  }
  for (int i = 0;  i < numWords;  i++) {
    elems[i] |= other.elems[i];
  }
  return *this;
}
```

```
// Пересечение множества с другим множеством
Set & Set::operator &= (const Set & other) {
  if (other.minElem != minElem || other.maxElem != maxElem) {
    throw out_of_range("Sets incomparable");
  }
  for (int i = 0;  i < numWords;  i++) {
    elems[i] &= other.elems[i];
  }
  return *this;
}

// Удаление элемента из множества
Set & Set::operator -= (int n) {
  if (n <= maxElem && n >= minElem) {
    int bit = n - minElem;
    elems[bit >> 4] &= ~(1 << (bit & 15));
  }
  return *this;
}

// Удаление элементов другого множества из данного (вычитание)
Set & Set::operator -= (const Set & other) {
  if (other.minElem != minElem || other.maxElem != maxElem) {
    throw out_of_range("Sets incomparable");
  }
  for (int i = 0;  i < numWords;  i++) {
    elems[i] &= ~other.elems[i];
  }
  return *this;
}

// Обращение (нахождение дополнения) множества
Set & Set::inverse() {
  for (int i = 0;  i < numWords;  i++) {
    elems[i] = ~elems[i];
  }
  return *this;
}

// Объединение множеств
const Set & operator | (const Set & s1, const Set & s2) {
  return Set(s1) |= s2;
}
```

```
// Пересечение множеств
const Set & operator & (const Set & s1, const Set & s2) {
  return Set(s1) &= s2;
}

// Вычитание множеств
const Set & operator - (const Set & s1, const Set & s2) {
  return Set(s1) -= s2;
}

// Дополнение множества до универсального
const Set & operator - (const Set & s) {
  return Set(s).inverse();
}
```

В приведенном классе определен достаточно богатый набор операций над множествами, однако для практических целей может понадобиться еще 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 `out_of_range`, хотя по смыслу это, конечно, совершенно законная операция. Полезным упражнением было бы модифицировать определение класса `Set` таким образом, чтобы все логически корректные операции над множествами были допустимы.

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, которые 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. Примерами таких структур являются все рассмотренные ранее структуры — массивы, множества, списки и деревья. Иногда 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.

Еще пример. 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.

Последний пример. 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 могут характеризоваться 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 $e = (u, v)$, называемых *дугами*, где u, v — вершины.

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

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:

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

где W_1 и W_2 представляют собой множества значений нагрузки вершин и дуг графа соответственно. Иногда при анализе графа неважно, какая из вершин u и v в дуге $e = (u, v)$ первая, а какая вторая, т. е. пара (u, v) не упорядочена. В этом случае дугу e называют *ребром*, а весь граф называют *неориентированным* в отличие от *ориентированного* графа, задаваемого исходным определением. Разумеется, в этом случае бессмысленно говорить о том, из какой

вершины ребро выходит и в какую вершину входит. Формально говоря, неориентированным графом называют такой граф, у которого наряду с любой дугой $e_1 = (u, v)$ имеется и противоположная дуга $e_2 = (v, u)$. Эта пара дуг и образует ребро $e = <u, v>$ 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 жестких ограничений на расход памяти.

Говоря о представлении графа, имеют в виду прежде всего вопрос о представлении в памяти его структуры, т. е. считается, что по представлению графа нужно уметь определять, какие вершины с какими другими вершинами в графе связаны. На практике кроме структуры графа нужно каким-то образом представлять и нагрузку на его вершины и дуги. От того, какие элементы графа нагружены и какого типа эта нагрузка, тоже зависит представление графа.

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. Можно считать, что вершины графа имеют номера от 0 до N–1, а каждая дуга характеризуется парой номеров вершин, инцидентных этой дуге.

В листингах будем представлять только структуру графов, причем для каждого способа представления опишем только конструктор соответствующего класса и операции добавления новой дуги и проверки наличия дуги с заданными концами. Для простоты опущены деструкторы графов и некоторые другие, несущественные для данного раздела, подробности. Поскольку большинство операций, представленных ниже в листингах, одинаковы для всех представлений графов, имеет смысл описать абстрактный класс `Graph`, в котором и определить все эти операции в виде пустых виртуальных функций. Тогда конкретные представления графов будут описаны в виде классов, производных от базового класса `Graph`.

Первое из описываемых представлений графа основано на следующих соображениях. Структуру графа можно описать, сопоставив каждой вершине множество дуг, выходящих из нее, причем каждая дуга, выходящая из вершины u, идентифицируется своим концом — номером вершины, в которую эта дуга входит. Таким образом, граф представляется массивом, в котором каждому номеру вершины u в диапазоне от 0 до N сопоставлено множество