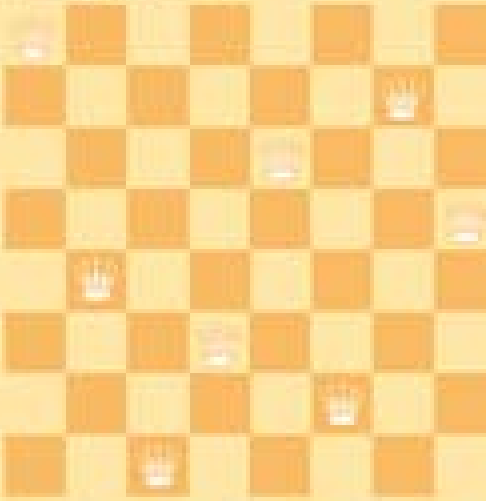
ПРОГРАММИРОВАНИЕ АЛГОРИТМОВ ОБРАБОТКИ ДАННЫХ

- *Основные принципы построения программ*
- *Модульное и объектно-ориентированное программирование*
- *Алгоритмы компьютерной обработки данных*
- *Усложненные структуры данных*
- *Примеры и упражнения*



УЧЕБНОЕ ПОСОБИЕ

bhv

ПРОГРАММИРОВАНИЕ АЛГОРИТМОВ ОБРАБОТКИ ДАННЫХ

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»

2003

УДК 681.3.06
ББК 32.973.26
У75

Ускова О. Ф. и др.

У75 Программирование алгоритмов обработки данных / Авторы: Ускова О. Ф., Огаркова Н. В., Воронина И. Е., Бакланов М. В., Мельников В. М. / — СПб.: БХВ-Петербург, 2003. — 192 с.: ил.

ISBN 5-94157-391-X

Учебное пособие для тех, кто уже приобрел начальные навыки программирования. В качестве базового используется язык Turbo Pascal. Объясняются понятия модульного и объектно-ориентированного программирования, дается представление о различных видах программ, в т. ч. рекурсивных, с возвратами. Рассматривается большое количество алгоритмов сортировки, таких как внутренние — методом подсчета, вставками, методом Шелла, быстрая, методом "пузырька", выбором и пр., и внешние — с помощью слияния, многофазная, каскадная. Приводятся также алгоритмы доступа к данным и выполняется их анализ. Введенные понятия иллюстрируются на примерах программ. Книга содержит большое количество задач и упражнений для самостоятельной работы.

Для программистов

УДК 681.3.06
ББК 32.973.26

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Дмитрий Фунт*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.08.03.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 15,5.
Тираж 3 000 экз. Заказ №
"БХВ-Петербург", 198005, 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 Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-391-X

Содержание

Введение

В последнее время на книжном рынке появилось большое количество изданий, посвященных популярным языкам и системам программирования. Однако далеко не каждое из этих изданий может быть рекомендовано в качестве учебного пособия для профессионального обучения программированию, главным образом по одной причине: эти книги описывают конкретные особенности того или иного языка, а не основополагающие принципы программирования.

Известно, что на рынке компьютерных приложений и программного обеспечения наибольшим спросом пользуются большие программы и программные системы со сложными данными. Систематический и научный подход к построению таких программ очень важен, поскольку программисты могут избежать большого количества ошибок, если со знанием дела будут применять те или иные методы программирования. Практика показывает, что невозможно реализовать все эти методы без знаний о различных вариантах структурирования данных. Н. Вирт в книге "Алгоритмы + структуры данных = программы" пишет: "Решения о структурировании данных нельзя принимать без знания алгоритмов, применяемых к этим данным, и наоборот, структура и выбор алгоритмов существенным образом зависят от структуры данных". В данном учебном пособии авторы попытались изложить классические алгоритмы и методы программирования, показывая на примерах, какая структура данных может быть использована в том или ином случае. Книга подготовлена на основе многолетнего опыта преподавания основных и специализированных дисциплин компьютерного цикла на факультете прикладной математики, информатики и механики Воронежского государственного университета.

Настоящее издание является логическим продолжением учебного пособия тех же авторов "Программирование на языке ПАСКАЛЬ: задачник", которое вышло в издательстве "Питер" в 2002 году и имеет гриф Министерства образования Российской Федерации.

Книга "Программирование алгоритмов обработки данных" предназначена для студентов вузов, углубленно изучающих информатику, преподавателей информатики, а также специалистов в области информационных технологий.

Книга состоит из двух частей. В *первой части* рассматриваются технологии реализации алгоритмов компьютерной обработки информации: модули и объекты.

Во *второй части* представлены наиболее употребительные алгоритмы компьютерной обработки данных: рекурсивные алгоритмы, задачи поиска (алгоритмы с возвратом, хеширование), алгоритмы внутренних и внешних сортировок, алгоритмы работы с сильно ветвящимися деревьями.

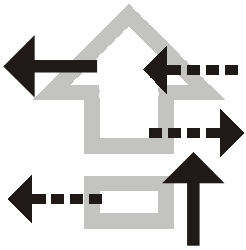
В *приложениях* приведены стандартные процедуры, необходимые для понимания многочисленных примеров программ, рассмотренных в книге.

Каждая глава начинается с краткого изложения теоретического материала, которое носит справочный характер. Последний раздел каждой главы содержит набор задач для самостоятельного решения. Практически все главы содержат примеры больших программ, которые служат хорошей иллюстрацией теоретического материала. В некоторых примерах не только приведен текст программы с комментариями, но и отражены этапы ее проектирования, развития и реализации. Авторы надеются, что их книга поможет в становлении математика-программиста и будет способствовать повышению культуры мышления. Книга предназначена для овладения компьютерными методами обработки информации путем развития профессиональных навыков разработки, выбора и преобразования алгоритмов, что является важной составляющей эффективной реализации программного продукта.

Представленный в книге материал удовлетворяет требованиям основной образовательной программы подготовки специалистов в области прикладной математики и компьютерных наук и поддерживает вузовские курсы, в которых изучаются структуры и алгоритмы обработки данных.

Авторы благодарны студентам и выпускникам факультета прикладной математики, информатики и механики Воронежского государственного университета, которые были самыми пристрастными и внимательными ценителями и судьями. Своими вопросами и замечаниями они помогли исправить шероховатости изложения материала и способствовали совершенствованию методики подачи материала.

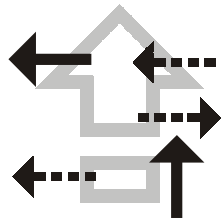
Коллектив авторов будет признателен за любые замечания, предложения, пожелания, направленные по адресу: **voronina@amm.vsu.ru** или **nataly@amm.vsu.ru**, 396006 г. Воронеж, Университетская пл., 1, факультет ПММ, тел.: (0732) 789-698.



ЧАСТЬ I

ТЕХНОЛОГИИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

Глава 1



Модульный подход в программировании

Модуль (`unit`) — это именованная совокупность констант, типов, переменных, процедур и функций. В отличие от подпрограммы (процедуры или функции) модуль хранится отдельно от основной программы и компилируется независимо от нее. Сам по себе модуль не является выполняемой программой — его объекты используются другими программными единицами.

Стандартные модули

Turbo Pascal имеет восемь стандартных модулей, в которых содержатся все системные процедуры и функции (`SYSTEM`, `DOS`, `CRT`, `PRINTER`, `OVERLAY`, `GRAPH`, `GRAPH3`, `TURBO3`).

Все перечисленные стандартные модули хранятся в системном файле Turbo.tpl. Кроме того, каждый стандартный модуль находится в одноименном TPU-файле — в системном каталоге Turbo Pascal. Имена всех используемых в программе модулей, кроме `SYSTEM`, необходимо указать в спецификации использования по обычным правилам. Модуль `SYSTEM` можно не указывать в спецификации использования, так как все его ресурсы подключаются автоматически к любой программе. Рассмотрим стандартные модули подробнее:

- в модуль `SYSTEM` входят все процедуры и функции стандартного языка и подпрограммы, ориентированные на конкретную операционную среду;
- модуль `DOS` содержит средства доступа к операционной системе и является программным представлением системного интерфейса MS DOS;
- модуль `CRT` обеспечивает возможность доступа к экрану дисплея в текстовом режиме. В этом модуле сосредоточены средства чтения информации с клавиатуры, включая расширенные коды клавиш, и простейшего управления звуком;
- модуль `PRINTER` имеет единственный интерфейсный элемент *Lst* типа *text*, системно связанный с логическим устройством PRN, то есть с печатающим устройством, если оно есть в конфигурации;

- модуль OVERLAY предоставляет средства для организации оверлейных программ больших размеров, превышающих объем доступной оперативной памяти;
- модуль GRAPH объединяет многочисленные программные средства управления графическим режимом работы дисплея, обеспечивает использование всех возможностей наиболее распространенных типов дисплеев CGA, EGA, VGA, HERCULES;
- модули TURBO3, GRAPH3 обеспечивают совместимость данной версии с более ранними версиями.

Общая структура модуля

Структурно модуль имеет заголовок и разделен на две части: интерфейс (Interface) и реализация (Implementation). Объявления, указанные в интерфейсе, являются доступными для любой программы, использующей этот модуль. Поэтому интерфейс называется открытой частью. В части реализации помещаются невидимые или скрытые объекты. Модуль в языке Turbo Pascal имеет следующий вид:

```
Unit <имя>;

  Interface {открытая часть (интерфейс)}
  Uses <список имен модулей>;
    {общие объявления}

  Implementation  {скрытая часть (реализация)}
    Uses <список имен модулей>;
    {локальные объявления}
    {процедуры и функции}

begin
  {операторы инициализации}
end.
```

Список модулей интерфейсной части содержит имена модулей, используемых в данном модуле или в вызывающей программе. Вызывающая программа может использовать эти модули, обратившись к ним с помощью **Uses** <имя>, так же, как если бы они были описаны в самой программе. Все вызванные величины являются для вызывающей программы глобальными.

Список модулей в разделе реализации делает доступными соответствующие модули только в разделе реализации данного модуля. Раздел **Implementation**

может содержать инициализирующую часть. Она, в частности, предназначена для установки начальных значений переменным модуля. В случае отсутствия операторов эта часть состоит из завершающего модуль **end**.

Таким образом, модуль состоит из заголовка и трех основных частей, любая из которых может быть пустой.

Компиляция модулей

Как и программа, модуль помещается в отдельный файл и компилируется. Имя файла, содержащего исходный текст модуля, должно совпадать с именем этого модуля. Компилятор после трансляции помещает код модуля в файл с таким же именем и расширением tpu (от английского Turbo Pascal Unit). Имя модуля, как и имя файла, должно содержать не более восьми символов. В одном файле может размещаться только один модуль. Если необходимо хранить код модуля в файле с другим именем, то можно использовать директиву `$U` для переопределения имени файла. Эта директива имеет один параметр, который трактуется как настоящее имя файла с данным модулем. Она должна находиться перед первым использованием модуля в спецификации использования. Например, конструкция

```
Uses {$U Nain} ABC;
```

приведет к тому, что компилятор будет искать код модуля ABC в дисковом файле Nain.tpu.

В среде Турбо Паскаля определены три режима компиляции:

- COMPILE;
- MAKE;
- BUILD.

При компиляции модуля или основной программы в режиме COMPILE все упоминающиеся в предложении `Uses` модули должны быть предварительно откомпилированы и результаты компиляции помещены в одноименные файлы с расширением tpu. Например, если в программе (модуле) имеется предложение

```
Uses Global;
```

то на диске в каталоге, объявленном опцией OPTIONS\DIRECTORIES уже должен находиться файл Global.tpu. Файл с расширением tpu создается автоматически в результате компиляции модуля. Так, если основная программа может компилироваться без создания исполняемого EXE-файла, то компиляция модуля всегда приводит к созданию TPU-файла.

В режиме MAKE компилятор проверяет наличие TPU-файлов для каждого объявленного модуля. Если какой-нибудь из файлов не обнаружен, система пытается отыскать и скомпилировать одноименный файл с расширением

pas, то есть файл с исходным текстом модуля. Кроме того, в этом режиме система следит за возможными изменениями исходного текста любого используемого модуля. Если в PAS-файлы внесены изменения, то система всегда осуществляет их компиляцию перед компиляцией основной программы. Более того, если изменения внесены в интерфейсную часть модуля, то будут перекомпилированы также и все другие модули, обращающиеся к нему.

В режиме BUILD существующие TPU-файлы игнорируются и система отыскивает и компилирует соответствующий PAS-файл для каждого объявленного в предложении `Uses` модуля.

Пример 1. Операции с комплексными числами

Создать модуль, реализующий средства работы с комплексными числами (листинг 1.1).

Листинг 1.1. Модуль, реализующий средства работы с комплексными числами

```
Unit Cmplx;
Interface
Type {Описание комплексного числа}
     Complex = record
                 re, im : real
               end;
  {Процедура ввода комплексного числа z}
  Procedure ReadC  (var z: Complex);
  {Процедура сложения двух комплексных чисел x и y}
  {Результат запоминается в переменной z}
  Procedure AddC   (x, y: Complex; var z: Complex);
  {Процедура вычитания двух комплексных чисел x и y}
  {Результат запоминается в переменной z}
  Procedure SubC   (x, y: Complex; var z: Complex);
  {Процедура умножения двух комплексных чисел x и y}
  {Результат запоминается в переменной z}
  Procedure MulC   (x, y: Complex; var z: Complex);
  {Процедура деления двух комплексных чисел x и y}
  {Результат запоминается в переменной z}
  Procedure DivC   (x, y: Complex; var z: Complex);
```

```
  {Функция сравнения двух комплексных чисел x и y}
  {Если числа равны, то функция возвращает}
  {значение true, иначе — значение false}
  Function EqvC (x, y: Complex): boolean;
  {Процедура вывода комплексного числа z}
  Procedure WriteC (z: Complex);

Const C: Complex = (re: 0.1; im: -2);

Implementation

  Procedure ReadC;
  begin
    with z do readln (re, im)
  end;

  Procedure AddC;
  begin
    z.re := x.re + y.re;
    z.im := y.im + x.im
  end;

  Procedure SubC;
  begin
    z.re := x.re - y.re;
    z.im := x.im - y.im
  end;

  Procedure MulC;
  begin
    z.re := x.re * y.re - x.im * y.im;
    z.im := x.re * y.im + x.im * y.re
  end;

  Procedure DivC;
  Var v: real;
  begin
    v := sqr(y.re) + sqr(y.im);
```

```
    if v=0 then
      begin
        writeln ('Ошибка! Деление на ноль'); readln; halt(0)
      end
    else
      begin
        z.re := (x.re * y.re + x.im * y.im) / v;
        z.im := (x.re * y.im - x.im * y.re) / v
      end;
  end;

  Function EqvC;
  begin
    EqvC := (x.re = y.re) and (x.im = y.im)
  end;

  Procedure WriteC;
  begin
    with z do
      if (re=0) and (im=0) then write('0')
      else
        begin
          if re<>0 then write (re:2:2);
          if im<>0 then
            begin
              if (im>0) and (re<>0) then write('+'); write(im:2:2,'i')
            end;
        end;
    writeln;
  end;
end. {конец модуля Cmplx}
```

Текст этого модуля необходимо поместить в файл Cmplx.pas. После того как он будет откомпилирован, вызывающей программе станут доступны процедуры из новой библиотеки. Например, в следующей программе (листинг 1.2) проверяется, совпадает ли результат арифметических операций сложения, вычитания, умножения над парой комплексных чисел с заданным комплексным числом.

Листинг 1.2. Пример программы, использующей модуль с комплексными числами

```
Program ComplexValue;
Uses Cmplx, Crt;
Var  a, b, c, d : Complex;
begin
  ClrScr;
  writeln('Введите действительную и мнимую части первого числа');
  ReadC(a);
  write('Первое число ');
  WriteC(a);

  writeln ('Введите действительную и мнимую части второго числа');
  ReadC(b);
  write('Второе число ');
  WriteC(b);

  writeln('Введите действительную и мнимую части третьего числа');
  ReadC(c);
  write('Третье число ');
  WriteC(c);

  AddC(a, b, d);
  write('Сумма двух первых чисел равна ');
  WriteC(d);
  If EqvC(c,d) then
     writeln('она совпадает с третьим числом')
  else
     writeln('она не совпадает с третьим числом');

  SubC(a, b, d);
  write('Разность двух первых чисел равна ');
  WriteC(d);
  If EqvC(c,d) then
     writeln('она совпадает с третьим числом')
  else
     writeln('она не совпадает с третьим числом');

  MulC(a, b, d);
```

```
  write('Произведение двух первых чисел равно ');
  WriteC(d);
  If EqvC(c,d) then
     writeln('оно совпадает с третьим числом')
  else
     writeln('оно не совпадает с третьим числом');
  readln;
End.
```

После объявления **Uses Cmplx** программе стали доступны все объекты, объявленные в интерфейсной части модуля **Cmplx**. При необходимости можно переопределить любой из этих объектов, как это произошло с объявленной в модуле типизированной константой *C*. Переопределение объекта означает, что вновь объявленный объект закрывает ранее определенный в модуле одноименный объект. Чтобы получить доступ к закрытому объекту, нужно воспользоваться составным именем: перед именем объекта поставить имя модуля и точку. Например, оператор

```
Writeln(cmplx.c.re:5:1, cmplx.c.im:5:1, 'i');
```

выведет на экран содержимое закрытой типизированной константы из приведенного примера.

Пример 2. Операции над матрицами

Задача 1

Вычислить выражение вида $f(X) = 2*X^3 - 5*X^2 + 3*X - 2*I$, где X — вводимая пользователем матрица размерности 3×3. Решение задачи дано в листинге 1.4.

Задача 2

Вычислить значение выражения $(A + B^T) * B$, где A — матрица размерности 2×3, B — матрица размерности 3×2, B^T — транспонированная матрица B. Решение задачи дано в листинге 1.5.

Решение данных задач базируется на общем модуле, в котором реализованы операции над матрицами (листинг 1.3).

Листинг 1.3. Реализация процедур для работы с матрицей

```
unit Matrix;
interface
const
```

```
  MDim  = 10;   {Максимальная размерность матрицы}
type
  TElem = Real {Integer};          {Тип элементов матрицы}
  Dimension = 1..MDim;             {Интервал индексов матрицы}
  TL = array [Dimension] of TElem; {строка матрицы}
  TM = array [Dimension] of TL;    {матрица}

  TMatrix = record
    {Действительные значения размерностей матрицы}
    Dim1, Dim2: Dimension;
    {Матрица}
    M: TM
  end;

  {Возвращает в переменной Result единичную матрицу}
  {размерности (Dim x Dim)}
  procedure MatrixI( Dim: Dimension; var Result: TMatrix );
  {Транспонирование матрицы A}
  procedure MatrixTranspose( A: TMatrix; var Result: TMatrix );
  {Ввод матрицы}
  {Размерности Dim1,Dim2 определяются при вызове процедуры}
  {( Dim1>0, Dim2>0) или вводятся пользователем}
  procedure MatrixInput(var Result: TMatrix; Dim1, Dim2: Integer);
  {Вывод матрицы}
  procedure MatrixOutput( A: TMatrix );
  {Сложение матриц. Матрицы должны быть одинаковой размерности}
  procedure MatrixSum( A,B: TMatrix; var Result: TMatrix );
  {Умножение матрицы A на число N}
  procedure MatrixNumberMul(A:TMatrix; N:TElem; var Result:TMatrix);
  {Вычитание матриц. Матрицы должны быть одинаковой размерности}
  procedure MatrixSub( A,B: TMatrix; var Result: TMatrix );
  {Умножение матриц. Result(i x k) = СУММА_ПО_j A(i x j)*B(j x k)}
  procedure MatrixMul( A, B: TMatrix; var Result: TMatrix );
  {Сравнение матриц на равенство}
  function MatrixEqual( A, B: TMatrix ): Boolean;
  {Вычисление нормы матрицы: ||A|| = МАХ_ПО_i_j |Aij|}
  function MatrixNorm( A: TMatrix ): TElem;
  {Возведение матрицы A в степень P}
```

```
  {Матрица A должна быть размерности (N x N)}
  procedure MatrixPower(A: TMatrix; P:Integer; var Result:TMatrix);

implementation

{Устанавливает размерность матрицы}
procedure SetDimension(var A: TMatrix; Dim1, Dim2: Dimension);
begin
  A.Dim1:=Dim1;
  A.Dim2:=Dim2
end;

{Возвращает в переменной Result единичную матрицу размерности
(Dim x Dim)}
procedure MatrixI( Dim: Dimension; var Result: TMatrix );
var k, j: Dimension;
begin
  for k:=1 to Dim do
    begin
      Result.M[k,k]:=1;          {на главной диагонали - единицы}
      {остальные элементы равны нулю}
      for j:=k+1 to Dim do
        begin
          Result.M[k,j]:=0;
          Result.M[j,k]:=0;
        end;
    end;
  SetDimension(Result, Dim, Dim)
end;

{Транспонирование матрицы A}
procedure MatrixTranspose( A: TMatrix; var Result: TMatrix );
var i, j: Dimension;
begin
  for i:=1 to A.Dim1 do
    for j:=1 to A.Dim2 do
      Result.M[j,i]:=A.M[i,j]; {внимательно: (i,j) -> (j,i)}
  SetDimension (Result, A.Dim2, A.Dim1)
```

```
end;

{Ввод матрицы}
{Размерности Dim1,Dim2 определяются при вызове процедуры}
{(Dim1>0, Dim2>0) или вводятся пользователем}
procedure MatrixInput( var Result: TMatrix; Dim1, Dim2: Integer );
var i, j: Dimension;
begin
  WriteLn('< Ввод матрицы >');

  if (Dim1<=0) or (Dim1>MDim) or (Dim2<=0) or (Dim2>MDim) then
  begin
    {ввод первой размерности}
    repeat
      Write('Введите первую размерность матрицы: ');
      ReadLn(Dim1)
    until (Dim1>0) and (Dim1<=MDim);

    {ввод второй размерности}
    repeat
      Write ('Введите вторую размерность матрицы: ');
      ReadLn (Dim2);
    until (Dim2>0) and (Dim2<=MDim)
  end;

  {сохраняем введенные размерности}
  SetDimension (Result, Dim1, Dim2);

  WriteLn ('Введите матрицу размерности ', Dim1, 'x', Dim2, ':');
  for i:=1 to Result.Dim1 do
  begin
    for j:=1 to Result.Dim2 do
      Read (Result.M[i,j]);
    ReadLn  {матрица вводится построчно}
  end
end;

{Вывод матрицы}
```

```
procedure MatrixOutput( A: TMatrix );
var i, j: Dimension;
begin
  for i:=1 to A.Dim1 do
  begin
    for j:=1 to A.Dim2 do
      Write(A.M[i,j]:8:2,' ');
    WriteLn
  end
end;

{Сложение матриц. Матрицы должны быть одинаковой размерности}
procedure MatrixSum( A, B: TMatrix; var Result: TMatrix );
var i, j: Dimension;
begin
  for i:=1 to A.Dim1 do
    for j:=1 to A.Dim2 do
      Result.M[i,j] := A.M[i,j] + B.M[i,j];
  SetDimension (Result, A.Dim1, A.Dim2)
end;

{Умножение матрицы A на число N}
procedure MatrixNumberMul( A: TMatrix; N: TElem; var Result:TMatrix);
var i,j: Dimension;
begin
  for i:=1 to A.Dim1 do
    for j:=1 to A.Dim2 do
      Result.M[i,j] := A.M[i,j] * N; {умножаем каждый элемент}
  SetDimension( Result, A.Dim1, A.Dim2 )
end;

{Вычитание матриц. Матрицы должны быть одинаковой размерности}
{Замечание. Данную операцию можно реализовать также через}
{умножение матрицы B на -1 и сложение с матрицей A}
procedure MatrixSub( A, B: TMatrix; var Result: TMatrix );
var i,j: Dimension;
begin
  for i:=1 to A.Dim1 do
```

```
    for j:=1 to A.Dim2 do
      Result.M[i,j] := A.M[i,j] - B.M[i,j];
  SetDimension (Result, A.Dim1, A.Dim2)
end;

{Умножение матриц. Result(i x k) = СУММА_ПО_j A(i x j)*B(j x k)}
procedure MatrixMul( A, B: TMatrix; var Result: TMatrix );
var i,j,k: Dimension;
    s: TElem;
begin
  SetDimension (Result, A.Dim1, B.Dim2);
  for i:=1 to Result.Dim1 do
    for k:=1 to Result.Dim2 do
    begin
      s := 0;
      for j:=1 to A.Dim2 do
        s := s + A.M[i,j] * B.M[j,k];
      Result.M[i,k] := s
    end
end;

{Сравнение матриц на равенство}
function MatrixEqual( A, B: TMatrix ): Boolean;
var i,j: Integer;
begin
  MatrixEqual := False;
  {размерности у равных матриц должны совпадать}
  if (A.Dim1=B.Dim1) and (A.Dim2=B.Dim2) then
    begin
      i:=0;
      repeat
        i:=i+1;
        j:=0;
        repeat
          j:=j+1
        until (j=A.Dim2) or (A.M[i,j]<>B.M[i,j])
      until (i=A.Dim1) or (A.M[i,j]<>B.M[i,j]);
      MatrixEqual := (A.M[i,j]=B.M[i,j]);
```

```
    end
end;

{Поиск максимума в строке}
function LineMax( Vector: TL; Dim: Dimension ): TElem;
var i : Dimension;
    max: TElem;
begin
  max := Vector[1]; {начальное значение}
  for i:=2 to Dim do
    if max < Vector[i] then max := Vector[i];
  LineMax := max
end;

{Вычисление нормы матрицы: ||A|| = MAX_ПО_i_j |Aij|}
function MatrixNorm( A: TMatrix ): TElem;
var i  : Dimension;
    max, Mmax: TElem;
begin
  Mmax := LineMax( A.M[1], A.Dim2 );  {начальное значение}
  for i:=2 to A.Dim1 do
  begin
    max := LineMax( A.M[i], A.Dim2 ); {находим максимум в строке}
    if max > Mmax then Mmax := max
  end;
  MatrixNorm := Mmax
end;

{Возведение матрицы A в степень P}
{Матрица A должна быть размерности (N x N)}
procedure MatrixPower( A: TMatrix; P: Integer; var Result: TMatrix );
var i: Dimension;
begin
  Result := A;
  for i:=2 to P do
    MatrixMul (A, Result, Result)
end;
end.{Конец модуля Matrix}
```

Листинг 1.4. Решение задачи 1

```
program MatrixTest1;
uses Matrix, Crt;
const Dim = 3;
var
  X: TMatrix;
  h1, h2, h3, h4: TMatrix;  {вспомогательные матрицы}
begin
  ClrScr;
  MatrixInput  (X, Dim, Dim);    {ввод матрицы}
  MatrixPower     (X, 3, h1);    {h1 = X^3}

  MatrixNumberMul (h1, 2, h1);   {h1 = 2*X^3}
  MatrixPower     (X, 2, h2);    {h2 = X^2}

  MatrixNumberMul (h2, 5, h2);   {h2 = 5*X^2}
  MatrixNumberMul (X, 3, h3);    {h3 = 3*X}

  MatrixI (Dim, h4);             {h4 = I}
  MatrixNumberMul (h4, 2, h4);   {h4 = 2*I}

  MatrixSub (h1, h2, X);   {вычисление заданного выражения}
  MatrixSum (X, h3, X);
  MatrixSub (X, h4, X);

  writeln ('Результат выражения 2*X^3-5*X^2+3*X-2*I равен ');
  MatrixOutput (X);        {вывод результата}
  WriteLn ('Нажмите клавишу <Enter> ...');
  ReadLn
end.
```

Листинг 1.5. Решение задачи 2

```
program MatrixTest2;
uses Matrix, Crt;
const
  Dim1 = 2;
```

```
  Dim2 = 3;
var
  A, B: TMatrix;
  h1: TMatrix;  {вспомогательная матрица}
begin
  ClrScr;
  MatrixInput  (A, Dim1, Dim2);   {ввод матрицы A}
  MatrixInput  (B, Dim2, Dim1);   {ввод матрицы B}

  MatrixTranspose (B, h1);         {транспонирование матрицы B}
  MatrixSum       (A, h1, h1);     {h1 = A+Bт}
  MatrixMul       (h1, B, h1);     {h2 = (A+Bт)*B}

  WriteLn ('¦(A+Bт)*B¦ = ', MatrixNorm(h1):8:4 );
  WriteLn ('Нажмите клавишу <Enter> ...' );
  ReadLn
end.
```

Пример 3. Работа с очередью

Дана строка текста, символы которой по одному вводятся с клавиатуры. Распечатать сначала все символы строки, затем все знаки препинания в том порядке, в котором они встречались в тексте. Решение задачи дано в листинге 1.7.

Для решения задачи воспользуемся такой структурой данных, как очередь, позволяющей хранить и выводить данные в том порядке, в котором они были занесены в очередь. Описание и все процедуры для работы с очередью вынесем в отдельный модуль `DinQueue` (листинг 1.6).

Листинг 1.6. Реализация процедур для работы с очередью

```
Unit DinQueue;
Interface
{Описание очереди на основе динамических структур данных}
type
  TElem = char;  {Тип элементов очереди}
  {Список, предназначенный для хранения элементов очереди}
  TList = ^TElement;
  TElement = record          {Элемент списка}
               Info: TElem;  {Информационная часть}
```

```
              Next: Tlist   {Указатель на следующий элемент списка}
            end;

  TQueue = record     {Представление очереди}
             Head: TList;  {"Голова" очереди}
             Tail: Tlist   {"Хвост" очереди}
           end;

{Описание процедур и функций для работы с очередью}
{Инициализация очереди Q}
procedure Queue_Init( var Q: TQueue );
{Проверка очереди Q на пустоту}
{Результат функции: true - очередь пуста, false - очередь не пуста}
function Queue_IsEmpty( Q: TQueue ): boolean;
{Поместить элемент E в "хвост" очереди Q}
procedure Queue_Push( var Q: TQueue; E: TElem );
{Извлечь элемент из очереди Q}
{Результат функции: извлеченный из "головы" очереди элемент}
function Queue_Pop( var Q: TQueue ): TElem;

{Реализация процедур и функций для работы с очередью}
Implementation

{Инициализация очереди Q}
procedure Queue_Init( var Q: TQueue );
begin
  {Изначально список, хранящий элементы очереди, пуст}
  Q.Head := nil
end; {Queue_Init}

{Проверка очереди Q на пустоту}
{Результат функции: true - очередь пуста, false - очередь не пуста}
function Queue_IsEmpty( Q: TQueue ): boolean;
begin
  {Очередь пуста, если пуст соответствующий список}
  Queue_IsEmpty := ( Q.Head = nil )
```

```
end; {Queue_IsEmpty}

{Поместить элемент E в "хвост" очереди Q}
procedure Queue_Push( var Q: TQueue; E: TElem );
var Z: TList;
begin
  new( Z ); {Создаем новое звено списка}
  Z^.Info := E;
  Z^.Next := nil;
  if Queue_IsEmpty( Q )
    then Q.Head := Z
    else Q.Tail^.Next := Z;
  Q.Tail := Z
end; {Queue_Push}

{Извлечь элемент из очереди Q}
{Результат функции: извлеченный из "головы" очереди элемент}
function Queue_Pop( var Q: TQueue ): TElem;
const ErrorCode = 255;
var Z: TList;
begin
  if Queue_IsEmpty( Q ) then
    begin
      writeln ('Извлечение элемента из пустой очереди !!! ');
      {Аварийное завершение программы}
      Halt (ErrorCode)
    end
  else
    begin
      Z := Q.Head;           {Сохраняем указатель на звено списка}
      Q.Head := Z^.Next;     {Перемещаемся к следующему звену}
      Queue_Pop := Z^.Info;  {Удаляем звено списка}
      dispose( Z )
    end
end; {Queue_Pop}
end. {UnQueue}
```

Листинг 1.7. Главная программа

```
program TestQueue;
Uses Crt, DinQueue;
Const {Знаки пунктуации}
      PunctMark = ['.',',',':','-','!','?'];
var   Queue : TQueue;  {Очередь}
      E : TElem;       {Очередной символ}
      flag : boolean;  {Флаг, который показывает,}
                       {надо ли выводить фразу "Результат: "}

begin
  ClrScr;
  {Инициализация очереди}
  Queue_Init (Queue);
  write ('Введите строку текста: ');
  flag := true;
  {Помещение знаков пунктуации в очередь или печать остальных символов}
  while not eoln do {Пока не конец строки}
    begin
      {Читаем следующий символ}
      read( E );
      if Flag then
        begin
          write('Результат: ');
          Flag := false;
        end;
      {Если символ - знак пунктуации}
      if E in PunctMark then
        {Помещаем его в очередь}
        Queue_Push (Queue, E)
      else
        {иначе - печатаем его}
        write(E)
    end;
  readln; {Читаем признак конца строки}
  {Печатаем знаки пунктуации}
  while not Queue_IsEmpty( Queue ) do
```

```
    {Пока очередь не пуста, извлекаем символ из очереди и выводим его}
    write (Queue_Pop(Queue));
  writeln;
  write ('Нажмите <Enter> ... ');
  readln
end.
```

Задания для самостоятельной работы

1. Составить модуль, реализующий следующие операции над векторами:
   - ввод компонент вектора с клавиатуры и из файла;
   - вывод компонент вектора на дисплей и в файл;
   - вычисление нормы вектора в различных метриках;
   - определение минимальной (максимальной) координаты вектора и ее порядкового номера;
   - вычисление скалярного произведения двух векторов;
   - проверка на возрастание или убывание последовательности координат.
2. Даны два вектора X и Y размерности $n = 30$. Используя составленный в задании 1 модуль, вычислить

   а) $\dfrac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{(n\sum x_i^2 - (\sum x_i)^2)(n\sum y_i^2 - (\sum y_i)^2}}$;

   б) $\dfrac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{(k\sum x_i^2 - (\sum x_i)^2)(p\sum y_i^2 - (\sum y_i)^2}}$,

   где k — порядковый номер максимальной координаты вектора X, p — порядковый номер минимальной компоненты вектора Y.
3. Даны три вектора X, Y, Z, каждый размерности $n = 30$. Используя составленные в задании 1 модули, вычислить

   а) minX * (A, A) + maxY * (B, C),

   где A — тот из данных трех векторов, минимальный элемент которого имеет самый большой номер (считать, что такой вектор единственный), B и C — два других вектора; $minX$ — минимальный элемент вектора X, $maxY$ — максимальный элемент вектора Y;

   б) k * (A, A) + p * (B, C),

   где A — тот из данных трех векторов, координаты которого упорядочены по возрастанию (считать, что такой вектор единственный), B и C — два

других вектора; k — номер минимального элемента вектора Z, p — номер максимального элемента вектора Y.

4. Используя модуль, разработанный для операций с комплексными числами, вычислить значение квадратного трехчлена с комплексными коэффициентами $ax^2 + bx + c$ в комплексной точке x.
5. Используя модуль, разработанный для операций с комплексными числами, вычислить с заданной точностью e значение комплексной функции

$$e^z = 1 + \frac{z}{1!} + \frac{z^2}{2!} + \ldots + \frac{z^n}{n!}.$$

6. Реализовать модуль для работы с множеством, число элементов в котором больше, чем 256. Примените представление "большого" множества в виде массива множеств. Модуль должен содержать процедуры инициализации множества, включения, исключения элементов и проверки принадлежности элемента множеству.
7. Используя модуль для работы с множеством, описанный в предыдущей задаче, составить программу нахождения целых чисел из диапазона 1..10000, удовлетворяющих представлению $n^2 + m^2$, где n и m — целые числа.
8. Реализовать модуль для работы со стеком. Напечатать в обратном порядке символы слова наибольшей длины из заданного текстового файла.
9. Реализовать модуль для работы с очередью. Напечатать наибольшее по длине предложение из заданного текстового файла.
10. Создать модуль, реализующий следующие операции над символьными строками:
    - поиск первого вхождения в строку заданного символа;
    - поиск последнего вхождения в строку заданного символа;
    - поиск последнего вхождения в строку заданной подстроки;
    - поиск в строке первого символа, отличного от пробела;
    - поиск в строке последнего символа, отличного от пробела.

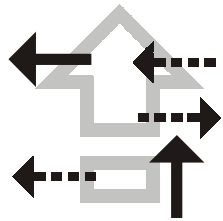
    Написать программу, иллюстрирующую работу полученного модуля.
11. Создать модуль, содержащий описание типа "дата" и операций над ним:
    - проверка корректности совокупности полей (день, месяц, год);
    - увеличение даты на заданное число дней;
    - уменьшение даты на заданное число дней;
    - преобразование типа "строка" → "дата";
    - преобразование типа "дата" → "строка".

    Написать программу, иллюстрирующую работу полученного модуля.

12. Создать модуль, реализующий представление длинного целого числа в виде строки символов, а также операции *, /, −, +. Написать программу, иллюстрирующую работу полученного модуля.
13. Создать модуль, реализующий представление полиномов в виде списка, а также операции *, /, −, +. Написать программу, иллюстрирующую работу полученного модуля.
14. Создать модуль, реализующий построение четырех видов диаграмм:
    - столбчатая;
    - круговая;
    - 3D столбчатая;
    - график.

    Написать программу, иллюстрирующую работу полученного модуля.
15. Создать модуль для работы с файлом сотрудников, где каждая запись содержит поля "фамилия", "имя", "отчество", "должность", "оклад" и признак удаленности записи. Реализовать следующие операции с файлом:
    - добавление;
    - удаление;
    - поиск по фамилии, имени, отчеству, должности и окладу;
    - расчет суммарного и среднего оклада всех сотрудников.

Глава 2



Объектно-ориентированный подход в программировании

Объектно-ориентированный язык программирования отличается высокой степенью структурированности, модульности и абстрактности. Его характеризуют три основных свойства: инкапсуляция, наследование и полиморфизм.

Инкапсуляция — это объединение записей с процедурами и функциями, что превращает их в новый тип данных — объект. Это объединение полей данных и кода в одной оболочке. Поля данных объекта содержат информацию об объекте. Процедуры и функции объекта называются методами. Метод — процедура или функция тесно связанные с данным типом. Методы описывают поведение объекта.

Принципы объектно-ориентированного программирования требуют, чтобы доступ к полям объектов осуществлялся посредством методов, а не напрямую.

Один из важнейших принципов объектно-ориентированного программирования: программист должен думать о коде и данных во время проектирования программ. Данные управляют потоком кода, а код манипулирует формой и значением данных.

Инкапсуляция данных, как и многие аспекты объектно-ориентированного программирования, — это дисциплина, которой необходимо следовать всегда. Обращаться к данным объекта следует посредством методов вместо прямого чтения данных. Можно усилить инкапсуляцию, используя зарезервированное слово `private` (личный) в объявлении объекта.

В ряде случаев может возникнуть потребность в отказе от экспорта отдельных частей объекта. Например, разработчик поставляет объекты другим программистам для использования и хочет запретить манипулировать полями объекта напрямую. С этой целью он может объявить эти поля, а при необходимости и методы, личными. Личные поля и методы доступны только внутри модуля, в котором объявлен объект. Личные поля и методы объявляются сразу после обычных методов:

```
type
  новый_объект = object (предок)
                   поля;
```

```
    методы;
  private
    поля;
    методы;
  end;
```

Наследование — это использование объекта для порождения иерархии потомков, с обеспечением возможности доступа каждого из порожденных объектов к коду и данным предка.

Экземпляры объектного типа объявляются точно так же, как объявляется любая переменная.

Объект считается *абстрактным*, если он может быть использован только в качестве базового для других объектов. Экземпляр от него не может быть образован.

Механизм наследования обеспечивает расширяемость объектов. При этом, когда задается родительский тип, методы родительского типа наследуются, но при необходимости они могут быть перекрыты. Чтобы перекрыть родительский метод, необходимо задать метод с тем же именем, что и родительский, но с другим телом и, если есть потребность, с другим набором параметров. Но (важно!) в то время как методы можно перекрывать, поле данных перекрывать нельзя. Если поле данных уже задано в иерархии объектов, порожденный тип не может задать поле данных с точно таким же идентификатором.

По умолчанию методы, объявленные в объектном типе, являются *статическими*. Они никак не помечаются.

Логика компилятора в разрешении вызовов статических методов такова: при вызове метода компилятор вначале ищет метод с этим именем, определенный внутри типа объект. Если метод с таким именем не определен внутри объектного типа, то компилятор обращается к непосредственному родительскому типу и ищет его внутри этого типа. Если метод с этим именем найден, то адрес родительского типа заменяет имя в исходном коде метода потомка. Если метод не найден, то компилятор поднимется выше к следующему предку. Когда компилятор достиг вершины иерархии, он выдаст сообщение об ошибке, указывающее, что искомый метод не определен.

Следует помнить, что используемый статический метод, найденный в родительском типе, будет точно таким, каким он определен и откомпилирован для родительского типа. Если родительский метод вызывает другие методы, то вызываемые методы будут родительскими методами, даже если потомок имеет методы, замещающие родительские.

Полиморфизм — это задание одного общего имени для действий над каждым из семейства объектов, с реализацией этого действия способом, соответст-

вующим каждому объекту в семействе. В нашем случае семейство — это наследственная иерархия.

Полиморфизм — чрезвычайно мощное средство обобщения. Реализуется он при помощи виртуальных методов. Статические методы компилятор размещает и разрешает на них ссылки во время компиляции. Такой процесс называется *ранним связыванием*. При обращении к виртуальным методам, в отличие от статических, конкретная процедура или функция определяется во время самого обращения. Этот процесс называется *поздним связыванием*.

Метод становится виртуальным, если за его объявлением следует зарезервированное слово `virtual`. Если метод в родительском типе объявлен виртуальным, то все методы с таким же именем в любом порожденном типе должны быть так же объявлены виртуальными, чтобы избежать ошибки компиляции.

Каждый тип объекта, который содержит виртуальные методы, должен иметь конструктор (`constructor`). Вызов конструктора должен быть сделан перед вызовом виртуального метода. Каждый экземпляр объекта должен инициализироваться с помощью отдельного вызова конструктора. Недостаточно инициализировать один экземпляр объекта, а затем присвоить это значение другим экземплярам. Хотя после этого другие экземпляры могут содержать правильные данные, они не будут инициализированы с помощью оператора присваивания и вызовут блокирование системы при вызове виртуальных методов. Например:

```
var  Item1, Item2: Тип_объекта; {два экземпляра Тип_объекта}
begin
  Item1.Init (параметр{ы}); {вызов конструктора для Тип_объекта}
  Item2 := Item1;           {неверно}
end;
```

Каково же функциональное назначение конструктора? Каждый тип объекта, содержащего виртуальные методы, имеет таблицу виртуальных методов (VMT). VMT содержит размер объекта и для каждого виртуального метода указатель на код, реализующий этот метод. Конструктор устанавливает связь между экземпляром, вызывающим его, и таблицей виртуальных методов. Именно поэтому вызов конструктора так важен и должен предшествовать вызову виртуального метода. Конструктор не может быть виртуальным.

Сами объекты могут быть как статическими, так и динамическими. Для размещения динамических объектов и освобождения памяти используются привычные процедуры `new` и `dispose`. Допускается расширение синтаксиса `new`, чтобы с помощью одной операции одновременно и выделить память, и проинициализировать объект:

```
var P:  ^Тип_объекта;
begin
```

```
new (P, Init (параметр{ы});
...
```

При этом `new` может иметь как функциональную, так и процедурную формы:

```
type PTR  =  ^Тип_объекта;
var P:  PTR;
begin
P:= new (PTR, Init (параметр{ы});
...
```

Освобождение памяти происходит с помощью `dispose`: `dispose (P)`. Но удаление ненужного динамического объекта может включать не только освобождение его пространства в куче. Объект сам может содержать указатель на другие объекты, которые необходимо освободить в определенном порядке, особенно когда они включают сложные динамические структуры данных. Существует специальный тип метода, предназначенный для очистки и удаления динамически распределенных объектов. Он называется деструктор (`destructor`). Деструктор комбинирует шаг освобождения памяти в куче с некоторыми другими задачами, которые необходимы для данного типа объекта.

Деструкторы можно наследовать, и они могут быть как статическими, так и виртуальными. Так как для различных типов объектов обычно требуются различные задачи освобождения памяти, деструкторы рекомендуется всегда делать виртуальными, чтобы в любом случае выполнялся бы соответствующий данному объекту деструктор. Деструкторы фактически действуют только на динамически распределенные объекты. При очистке такого объекта деструктор гарантирует, что всегда будет освобождаться правильное количество байтов в куче. Однако нет никакого вреда при использовании деструкторов со статически распределенными объектами.

Действие деструкторов наиболее полно проявляется при использовании полиморфных объектов. Полиморфный объект — это объект, который был назначен родительскому типу на основании правил расширенной совместимости типов объектов.

Расширенная совместимость типов имеет три формы:

- между экземплярами объектов;
- между указателями на экземпляры объектов;
- между формальными и фактическими параметрами.

Во всех трех формах совместимость типов распространяется от потомка к предку: родительскому экземпляру можно присваивать экземпляр любого порожденного им типа.

Для полиморфных объектов во время компиляции известно лишь то, что данный объект будет одним из объектов, порожденных заданным типом. Во время компиляции из полиморфного объекта нельзя получить никакой информации о размере объекта.

Деструктор решает проблему посредством обращения к таблице виртуальных методов данного экземпляра. В каждой таблице виртуальных методов, как было уже упомянуто, есть размер в байтах объекта, к которому относится эта таблица.

Чтобы выполнить освобождение памяти при позднем связывании, деструктор надо вызвать как часть расширенного синтаксиса процедуры `dispose`:

```
dispose (P, Done);
```

Done — рекомендуемое имя деструктора.

Следует заметить, что метод деструктора может быть пустым и все равно выполнять свою функцию:

```
destructor Тип_объекта.Done;
begin
end;
```

Полезную работу в таком деструкторе выполняет не тело метода, а код, генерируемый компилятором в ответ на зарезервированное слово `destructor`. В этом деструктор подобен модулю, который не экспортирует ничего, но выполняет некоторую неявную функцию посредством выполнения инициализации перед запуском программы.

Объектно-ориентированное программирование (ООП) позволяет рационально организовывать последовательности данных в программных структурах. Наследование и инкапсуляция являются эффективными способами для управления сложностью прикладных задач. Кроме того, ООП предоставляет возможность повторного использования кода и обеспечивает его расширяемость.

Пример 1. Обработка строки

Дана строка текста. Вывести сначала все символы, отличные от знаков препинания, в порядке появления их в тексте, а затем все знаки препинания в обратном порядке. Для решения задачи использовать стек, при этом предоставить пользователю возможность выбрать вид стека: построенный на основе динамических структур данных или построенный на основе массива. Проиллюстрировать процесс изменения данных в стеке.

Для решения этой задачи удобно воспользоваться методологией *объектно-ориентированного программирования*. В качестве исходного можно выделить абстрактный объект `TVisualStack` (родительский, вышележащий объект),

который содержит данные, отвечающие за визуализацию работы стека — координаты вершины стека. Дочерними (нижележащими) объектами будут: `TDinStack`, который отвечает за реализацию стека, построенного на базе линейного списка, и `TStatStack`, который отвечает за реализацию стека, построенного на базе массива. На рис. 2.1 приведено иерархическое дерево взаимосвязей между объектами.

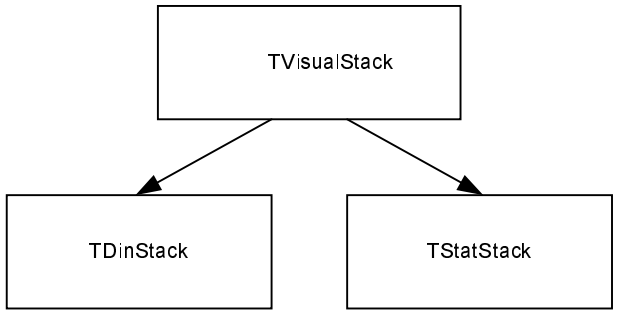


Рис. 2.1. Иерархическое дерево взаимосвязей между объектами

Обычно для любого стека определяются следующие процедуры и функции:

- процедура инициализации стека `Init`;
- функция проверки стека на пустоту `IsEmpty`;
- процедура, которая отвечает за помещение элемента в стек `Push`;
- процедура, которая отвечает за извлечение элемента из стека `Pop`.

Кроме этих процедур часто в задачах требуются такие процедуры, как:

- очистка стека (извлечение всех его элементов) `Clear`;
- печать содержимого стека и его очистка `Print`.

Поскольку реализация последних двух процедур не зависит от типа стека (эти процедуры зависят только от функции проверки стека на пустоту `IsEmpty` и процедуры извлечения элемента из стека `Pop`), то подпрограммы `Pop`, `Push` и `IsEmpty` следует сделать виртуальными, чтобы для каждого из стеков (динамического и статического) не переопределять процедуры `Clear` и `Print`. Из-за виртуальности этих процедур, потребуется конструктор и деструктор. В качестве конструктора удобно выбрать процедуру инициализации стека `Init`. Как деструктор определим процедуру `Done`, которая будет выполнять одно-единственное действие — очищать стек.

Процедуры для реализации стека на основе динамических структур данных и на основе массива не требуют дополнительного пояснения (листинг 2.1).

Листинг 2.1. Модуль, в котором реализованы различные варианты стека

```
Unit Stacks;
Interface
Uses Crt;
const {Максимальное число элементов стека, созданного на основе массива}
      MaxN = 20;
      StackColor = blue;  {цвет элементов стека}
      {горизонталь, на которой выводятся элементы стека}
      StackLine  = 20;
      CDelay = 15000;     {задержка в программе}

Type TElem = char;          {тип элементов стека}
     TVisualStack = object  {абстрактный стек}
       private
         X, Y : integer;    {координаты вершины стека}
       public
         {инициализация стека}
         constructor Init;
         {абстрактная функция проверки стека на пустоту}
         function    IsEmpty : boolean; virtual;
         {помещение элемента Elem в стек}
         procedure   Push (Elem : TElem); virtual;
         {извлечение элемента Elem из стека}
         procedure   Pop  (var Elem : TElem); virtual;
         {вывод сообщения об ошибке}
         procedure   ErrorMessage (ErrorCode:byte);
         {извлечение всех элементов из стека}
         procedure   Clear ;
         {печать всех элементов стека}
         procedure   Print;
         {уничтожение стека}
         destructor  Done;  virtual;
     end;
     {Указатель на абстрактный стек}
     PtrVisualStack = ^TVisualStack;

     {описание линейного списка}
```

```
{для стека, построенного на базе динамических структурах данных}
TList = ^TNode;
TNode = record
  inf : TElem;
  next : TList
end;

{стек, построенный на основе линейного списка}
TDinStack = object(TVisualStack)
  private
    top : TList;  {вершина стека}
  public
    {инициализация стека}
    constructor Init;
    {абстрактная функция проверки стека на пустоту}
    function    IsEmpty : boolean; virtual;
    {помещение элемента Elem в стек}
    procedure   Push (Elem : TElem); virtual;
    {извлечение элемента Elem из стека}
    procedure   Pop  (var Elem : TElem); virtual;
    {уничтожение стека}
    destructor  Done;  virtual;
end;
{указатель на динамический стек}
PtrDinStack = ^TDinStack;

{Описание массива для стека, построенного на его основе}
TElements = array [1..MaxN] of TElem;
{Представление стека на основе массива}
TStatStack = object(TVisualStack)
  private
    Elements: TElements; {Элементы стека}
    Top: integer ;       {Индекс вершины стека}
  public
    {инициализация стека}
    constructor Init;
    {абстрактная функция проверки стека на пустоту}
    function    IsEmpty : boolean; virtual;
```

```
        {помещение элемента Elem в стек}
        procedure   Push (Elem : TElem); virtual;
        {извлечение элемента Elem из стека}
        procedure   Pop  (var Elem : TElem); virtual;
        {уничтожение стека}
        destructor  Done;  virtual;
    end;
    {указатель на стек, построенный на основе массива}
    PtrStatStack = ^TStatStack;

Implementation

{Инициализация абстрактного стека}
{Инициализация координат для вершины стека}
constructor TVisualStack.Init;
begin
  X:=7;
  Y:=StackLine;
end;

{Визуальное отображение при помещении элемента в стек}
procedure TVisualStack.Push;
var TmpX,TmpY : integer;
    TmpColor : word;
begin
  {Запоминаем текущие параметры: координаты курсора и цвет}
  TmpX:=WhereX; TmpY:=WhereY;
  TmpColor:=TextAttr mod 16;
  {подсчитывает координаты новой вершины}
  X:=X+2;
  if X>79 then
    begin
      X:=1; Y:=Y+1
    end;
  {Выводим вершину}
  TextColor(StackColor);
  GotoXY(X,Y);
  write(Elem,' ');
```

```
  {Восстанавливаем параметры}
  TextColor(TmpColor);
  GotoXY(TmpX,TmpY);
end;

{Визуальное отображение при извлечении элемента в стек}
procedure TVisualStack.Pop;
var TmpX,TmpY : integer;
    TmpColor : word;
begin
  {Запоминаем текущие параметры: координаты курсора и цвет}
  TmpX:=WhereX; TmpY:=WhereY;
  TmpColor:=TextAttr mod 16;
  {стираем элемент на экране}
  GotoXY(X,Y);
  TextColor(TextAttr div 16);
  write(Elem,' ');
  {пересчитываем координаты новой вершины}
  X:=X-2;
  if X<0 then
    begin
      X:=79; Y:=Y-1
    end;
  {восстанавливаем параметры}
  TextColor(TmpColor);  GotoXY(TmpX,TmpY);
end;

{абстрактная функция проверки стека на пустоту}
function TVisualStack.IsEmpty;
begin
end;

{вывод сообщения об ошибке}
procedure TVisualStack.ErrorMessage;
const Code = 255;
var   ch : char;
begin
  TextColor(Red+Blink);
```

```
  GotoXY (20,24);
  if ErrorCode=0 then
    writeln( 'Извлечение элемента из пустого стека !!!' )
  else
    writeln( 'Переполнение стека !!!' );
  repeat until KeyPressed;
  ch:=ReadKey
  {Аварийное завершение программы}
  Halt( ErrorCode )
end;

{Извлечение всех элементов стека}
procedure TVisualStack.Clear;
var elem : TElem;
begin
  while not IsEmpty do Pop(elem);
end;

{печать и извлечение всех элементов из стека}
procedure TVisualStack.Print;
var elem : TElem;
begin
  while not IsEmpty do
    begin
      Pop(elem);
      write(elem);
      Delay(cDelay)
    end;
end;

{уничтожение стека}
destructor TVisualStack.Done;
begin
  Clear
end;

{инициализация стека, созданного на основе линейного списка}
constructor TDinStack.Init;
```

```
begin
  inherited Init; {инициируем координаты вершины стека}
  Top := nil;     {в стеке нет элементов}
end;

{проверка стека на пустоту}
function TDinStack.IsEmpty;
begin
  IsEmpty:=(top=nil)
end;

{помещение элемента Elem в стек}
procedure TDinStack.Push;
var t : TList;
begin
  new(t); {создаем новое звено}
  t^.inf:=Elem; {добавляем его в стек}
  t^.next:=top;
  top:=t;
  {отображаем это на экране}
  Inherited Push(Elem)
end;

{Извлечение элемента из стека}
procedure TDinStack.Pop;
var t: TList;
begin
  {Если стек пуст, выводим сообщение об ошибке}
  if IsEmpty then ErrorMessage(0)
  else {в противном случае}
    begin
      {запоминаем элемент из вершины}
      t:=top;
      Elem := t^.inf;
      {стираем его с экрана}
      inherited Pop(Elem);
      {Уничтожаем это звено}
      top:=t^.next;
```

```
      dispose(t)
    end;
end;

{Уничтожение стека}
destructor TDinStack.Done;
begin
  Clear
end;

{инициализация стека}
constructor TStatStack.Init;
begin
  {Инициируем координаты вершины стека}
  inherited Init;
  {Изначально стек не содержит элементов}
  Top := 0
end;

{Проверка стека S на пустоту}
function TStatStack.IsEmpty;
begin
  {Стек пуст, если он не содержит элементов}
  IsEmpty := ( Top = 0 )
end;

{Поместить элемент Elem в стек}
procedure TStatStack.Push ;
begin
  {если добавление элемента не приведет к переполнению стека}
  if Top < MaxN then
    begin
      {добавляем элемент в стек}
      Top := Top + 1;
      Elements[Top] := Elem;
      {выводим его на экран}
      inherited Push(Elem)
    end
```

```
  {иначе - печатаем сообщение об ошибке}
  else ErrorMessage(1)
end;

{Извлечение элемента из вершины стека}
procedure TStatStack.Pop;
begin
  {если стек пуст, печатаем сообщение об ошибке}
  if IsEmpty then ErrorMessage(0)
  else
    begin
      {иначе извлекаем элемент из стека}
      Elem := Elements[Top];
      {стираем его с экрана}
      inherited Pop(Elem);
      Top := Top-1
    end
end;

{уничтожение стека}
destructor TStatStack.Done;
begin
  Clear;
end;
end. {Stacks}

{основная программа}
program Main;
Uses Stacks, Crt;
Const {горизонталь, в которой выводится результат}
      TxtLine = 11;
      {Цвет строки-результата}
      TxtColor = LightGray;
      {Знаки пунктуации}
      PunctuationMark = ['.',',',':',';','-','!','?'];
var
    str : string;  {исходная строка текста}
    i: byte;       {вспомогательная переменная}
```

```
    {указатель на абстрактный стек}
    Stack : PtrVisualStack;
    {Указатель на стек, построенный на основе линейного списка}
    DinStack : PtrDinStack;
    {Указатель на стек, построенный на основе массива}
    StatStack : PtrStatStack;

{Процедура для определения типа стека,}
{который будет использоваться в программе}
procedure InputTypeStack;
var ch: char;
begin
  {Вывод меню на экран}
  writeln('Введите тип стека, с которым Вы хотите работать');
  writeln('D - стек на основе динамических структур данных');
  writeln('S - стек на основе массива');
  {запрос пользователю}
  repeat
    ch := ReadKey
  until ch in ['S','s','Ы','ы','D','d','B','в'];
  write(ch);
  {определение варианта ответа}
  case ch of
    {если пользователь выбрал статический стек}
    'S','s','Ы','ы' :
      begin
        {инициируем статический стек и запоминаем его}
        new(StatStack,Init);
        Stack:=StatStack
      end;
    {если пользователь выбрал динамический стек}
    'D','d','B','в' :
      begin
        {инициируем динамический стек и запоминаем его}
        new(DinStack, Init);
        Stack:=DinStack
      end;
  end;
```

```
end;

{процедура вывода строки str в точке с координатами X, Y цветом color}
procedure OutTextXY (X, Y: integer; color : word; Str: string);
begin
  GotoXY(x,y);
  TextColor(color);
  write(Str);
end;

begin
  {Ввод данных}
  ClrScr;
  writeln('Введите строку текста');
  readln(str);
  writeln;
  InputTypeStack;

  {Оформление экрана}
  ClrScr;
  OutTextXY(1,2,Yellow,'Исходная строка: ');
  OutTextXY(3,3,Yellow,str);
  OutTextXY(1,StackLine,StackColor,'Стек: ');
  OutTextXY(1,TxtLine-1,Cyan,'Полученная строка: ');
  GotoXY(3,TxtLine);

  {Обработка строки и вывод результата}
  for i:=1 to length(str) do
    begin
      {Если это не знак пунктуации, то печатаем его}
      if not (str[i] in PunctuationMark) then
        write(str[i])
      else
        {иначе - помещаем в стек}
        Stack^.Push(str[i]);
      {задержка}
      Delay(cDelay)
    end;
```

```
  {печать содержимого стека}
  Stack^.Print;
  {уничтожение стека}
  dispose(Stack,Done);
end.
```

Пример 2. Элементарный графический редактор

Демонстрируется минимальный набор средств для создания на экране примитивного рисунка, напоминающий детскую игрушку, когда с помощью двух вращающихся ручек управляют головкой, которая, перемещаясь, оставляет след.

По экрану с помощью стрелок и расположенных рядом на служебной части клавиатуры клавиш перемещается курсор, представляющий собой стрелку. "Горячей" точкой, то есть точкой, определяющей местоположение курсора на экране, является острие стрелки. Если нажать клавишу <+>, то курсор начинает оставлять за собой след. Повторное нажатие этой же клавиши прекращает появление следа. В качестве дополнительной возможности нажатие клавиши <R> позволяет войти в режим рисования окружности, когда клавиши <Ins> и <Del> служат для ее расширения и сжатия. Повторное нажатие <R> позволяет выйти из режима манипуляции с окружностью.

Программа (листинг 2.5) использует три модуля (листинги 2.2, 2.3 и 2.4).

Листинг 2.2. Модуль KEYBOARD для работы с клавиатурой

```
unit Keyboard;
Interface
  Type {Возможные варианты нажатых клавиш}
       tKeyType = (ktUnknown, ktEsc, ktEnter,
                   ktLeft, ktRight, ktUp, ktDown,
                   ktLeftUp, ktRightUp, ktLeftDown, ktRightDown,
                   ktIns, ktDel, ktCircle, ktTrack);
    {Функция, отвечающая за считывание клавиши и определения ее типа}
    function GetKey : tKeyType;

Implementation
  Uses Crt;
Const
```

```
  kbEsc=#27;       {клавиша <ESC>}
  kbLeft=#75;      {клавиша <СТРЕЛКА ВЛЕВО>}
  kbRight=#77;     {клавиша <СТРЕЛКА ВПРАВО>}
  kbUp=#72;        {клавиша <СТРЕЛКА ВВЕРХ>}
  kbDown=#80;      {клавиша <СТРЕЛКА ВНИЗ>}
  kbRightDown=#73; {клавиша <PgDn>}
  kbRightUp=#81;   {клавиша <PgUp>}
  kbLeftDown=#71;  {клавиша <End>}
  kbLeftUp=#79;    {клавиша <Home>}
  kbCircle='R';    {клавиша <R>}
  kbEnter=#13;     {клавиша <Enter>}
  kbTrack=#43;     {клавиша <+>}
  kbIns=#82;       {клавиша <Del>}
  kbDel=#83;       {клавиша <Ins>}

Function GetKey;
var ch : char;
begin
  GetKey:=ktUnknown;
  ch:=ReadKey;
  if ch=#0 then
    begin
      ch:=readkey;
      case ch of
        kbLeft      : GetKey:=ktLeft;
        kbRight     : GetKey:=ktRight;
        kbUp        : GetKey:=ktUp;
        kbDown      : GetKey:=ktDown;
        kbLeftUp    : GetKey:=ktLeftUp;
        kbRightUp   : GetKey:=ktRightUp;
        kbLeftDown  : GetKey:=ktLeftDown;
        kbRightDown : GetKey:=ktRightDown;
        kbIns       : GetKey:=ktIns;
        kbDel       : GetKey:=ktDel;
      end
    end
  else
    case ch of
```

```
        kbEsc    : GetKey:=ktEsc;
        kbEnter  : GetKey:=ktEnter;
        kbCircle : GetKey:=ktCircle;
        kbTrack  : GetKey:=ktTrack;
      end;
  end;
end.
```

Листинг 2.3. Модуль PRIMITIVES

```
{Модуль PRIMITIVES содержит некоторые примитивные графические объекты}
unit Primitives;
interface
uses Graph, Keyboard;
type
 {абстрактный объект первого уровня иерархии наследования,
 отвечающий за привязку потомков к координатной сетке экрана}
 Location = object
   private
    x, y: integer;
   public
    procedure Init(InitX, InitY: integer);
    function GetX: integer;
    function GetY: integer;
  end;
  {абстрактный объект второго уровня иерархии наследования,
  отвечающий за сохранение формы объекта и его корректную перерисовку}
  Figure = object (Location)
    private
      {указатель для сохранения области под объектом
       для последующего восстановления этой области
       в случае перемещения объекта на новое место}
      AreaPtr : pointer;
      {ширина и высота области}
      Width, Height : integer;
      procedure   SaveArea;
    public
      constructor Init(InitX, InitY, InitWidth, InitHeight: integer);
```

```
      destructor   Done ; virtual;
      function    GetWidth: integer;
      function    GetHeight: integer;
      function    GetSize: integer; virtual;
      procedure   Draw ; virtual;
      procedure   Show ;
      procedure   Hide ;
      procedure   MoveTo(NewX, NewY : integer);
  end;

  Point= object(Figure) {геометрический объект — точка}
     constructor Init(InitX, InitY: integer);
     destructor  Done ; virtual;
     procedure Draw ; virtual;
  end;

 Circle = object (Figure) {геометрический объект — окружность}
  private
   Radius: integer;
  public
   constructor Init (InitXcenter, InitYcenter, InitRadius: integer);
   destructor Done; virtual;
   procedure  Draw; virtual;
   procedure  SetRadius(NewRadius: integer);
   function   GetRadius: integer;
   {методы, обеспечивающие поведение окружности: ее расширение и сжатие}
   procedure  Expand(ExpandBy: integer); virtual;
   procedure  Contract(ContractBy: integer); virtual;
   procedure  ChangeSize; virtual;
 end;

implementation
{реализация методов объектов}

{Location rules}
procedure Location.Init;
begin
```

```
  x:=InitX;
  y:=InitY;
end;

function Location. GetX: integer;
begin
  GetX:=X;
end;

function Location.GetY: integer;
begin
  GetY:=Y;
end;

{Figure rules}
constructor Figure.Init;
begin
  inherited Init(InitX,InitY);
  width:=InitWidth;
  Height:=InitHeight;
  SaveArea;
end;

destructor Figure.Done;
begin
  FreeMem(AreaPtr,GetSize);
end;

function Figure.GetWidth;
begin
  GetWidth:=Width
end;

function Figure.GetHeight;
begin
  GetHeight:=Height
end;
```

```
function Figure.GetSize ;
begin
  GetSize:=ImageSize(x,y,x+width,y+Height)
end;

procedure Figure.SaveArea;
begin
  GetMem(AreaPtr, GetSize);
  GetImage(X, Y, X+Width, Y+Height, AreaPtr^);
end;

procedure Figure.Draw;
begin
end;

procedure Figure.Show;
begin
  SaveArea;
  Draw;
  PutImage(X,Y,AreaPtr^,OrPut);
end;

procedure Figure.Hide;
begin
  PutImage(X,Y,AreaPtr^,NormalPut);
end;

procedure Figure.MoveTo;
begin
  Hide;
  x:=NewX; y:=NewY;
  Show;
end;

{Point rules}
constructor Point.Init;
begin
  Inherited Init(InitX, InitY, 1, 1);
```

```
  SaveArea;
end;

procedure Point.Draw;
begin
  PutPixel(X,Y,GetColor);
end;

destructor Point.Done;
begin
  inherited Done
end;

{CircleRules}
constructor Circle.Init;
begin
  Radius:= InitRadius;
  inherited Init(InitXcenter-InitRadius,InitYcenter-InitRadius,
                 2*InitRadius+1,2*InitRadius+1);
end;

destructor Circle.Done;
begin
  inherited Done;
end;

procedure Circle.Draw;
begin
  Graph.Circle(x+Radius, y+Radius, Radius);
end;

function Circle.GetRadius: integer;
begin
  GetRadius:=Radius
end;

procedure Circle.SetRadius;
var xx, yy: integer;
```

```
begin
  xx:=x+Radius; yy:=y+Radius;
  FreeMem(AreaPtr,GetSize);
  if NewRadius<0 then NewRadius:=0;
  Radius:=NewRadius;
  x:=xx-Radius; y:=yy-Radius;
  Width:=2*Radius+1; Height:=Width;
  SaveArea;
end;

procedure Circle.Expand;
begin
  Hide;
  SetRadius(Radius+ExpandBy);
  Show;
end;

procedure Circle.Contract;
begin
  Expand(-ContractBy);
end;

procedure Circle.ChangeSize;
var key : TKeyType;
begin
  Show;
  repeat
    key:=GetKey;
    case Key of ktIns  : Expand(1);
                ktDel  : Contract(1);
    end;
  until Key=ktEnter;
end;{ChangeSize}
end.
```

Листинг 2.4. Модуль CURSORS, содержащий объект курсор

```
unit cursors;
interface
```

```
uses Graph, Primitives, Keyboard;
type
   define = procedure;
   {объект - курсор}
   cursor = object(Figure)
     private
       {точка области, которая оставляет след}
       PaintPoint : Location;
       {поле, которое показывает, оставляет ли курсор след}
       track: boolean;
       {поле, определяющее форму курсора}
       Form: define;
       {указатель для сохранения области под объектом}
       {для перерисовки курсора}
       FormPtr: Pointer;
     public
       constructor Init(InitX, InitY, PaintX, PaintY: integer;
                        InitWidth, InitHeight: integer;
                        InitTrack: boolean; InitForm: define);
       destructor Done; virtual;
       procedure Draw ; virtual;
       {метод, обеспечивающий перемещение курсора по экрану}
       procedure Drag (dragBy: integer); virtual;
   end;

implementation

constructor Cursor.Init;
begin
  inherited Init(InitX, InitY, InitWidth, InitHeight);
  PaintPoint.Init(PaintX,PaintY);
  Track:=InitTrack;
  GetMem(FormPtr, GetSize);
  Form:=InitForm;
  Form;
  GetImage(0,0,GetWidth, GetHeight, FormPTR^);
  PutImage(0,0,FormPTR^, xorPUT);
end;
```

```
destructor Cursor.Done;
begin
  Hide;
  inherited Done;
  FreeMem(FormPTR, GetSize);
end;

procedure Cursor.Draw;
begin
  PutImage(GetX, GetY, FormPTR^,  ORPUt);
end;

procedure Cursor.Drag(dragBy: integer);
var DeltaX, DeltaY: integer;
    aCircle: Circle;
  function GetInstr(var Dx, Dy: integer): boolean;
  var key: TKeyType;
  begin
    Dx:=0; Dy:=0;
    GetInstr:=true; key:=GetKey;
    case Key of
      ktLeft:      Dx:=-1;
      ktRight:     Dx:=1;
      ktDown:      Dy:=1;
      ktUp:        Dy:=-1;
      ktRightDown: begin
                     Dx:=1; Dy:=-1;
                   end;
      ktRightUp:   begin
                     Dx:=1; Dy:=1;
                   end;
      ktLeftDown:  begin
                     Dx:=-1; Dy:=-1;
                   end;
      ktLeftUp:    begin
                     Dx:=-1; Dy:=1;
                   end;
```

```
      ktCircle: begin
                  Hide;
                  aCircle.Init(GetX+PaintPoint.GetX,
                               GetY+PaintPoint.GetY, 1);
                  aCircle.ChangeSize;
                  aCircle.Done;
                  Show
                end;
      ktTrack:  track:=not track;
      ktEsc:    GetInstr:=false;
    end;
  end;{function}

begin
  Show;
  while GetInstr (DeltaX, DeltaY) do
    begin
      if track then
        begin
          Hide;
          Line(GetX+PaintPoint.GetX, GetY+PaintPoint.GetY,
               GetX+PaintPoint.GetX+DeltaX*dragBy,
               GetY+PaintPoint.GetY+DeltaY*dragBy);
          Show;
        end;
      MoveTo(GetX+DeltaX*dragBy, GetY+DeltaY*dragBy);
    end;
end;{Drag}
end.
```

Листинг 2.5. Главная программа

```
program Main;
uses Crt, Graph, Primitives, Keyboard, Cursors;
var MyCursor: cursor;
    Driver, mode: integer;
{$F+}
procedure CursForm;  {процедура "рисования" курсора}
```

```
begin
  Line(0,0,7,7); Line(0,0,4,2); Line(0,0,2,4);
end;
{$F-}

begin
  DetectGraph(Driver, Mode);  {Инициализация графического режима}
  InitGraph(Driver, Mode, ' ');
  SetGraphMode(Mode);
  {инициализация курсора}
  MyCursor.Init(159, 99,0,0, 8, 8, false, CursForm);
  MyCursor.Drag(5);                          {Управление курсором}
  MyCursor.Done;                               {Удаление курсора}
  CloseGraph;                          {Переход к текстовому режиму}
end.
```

Задания для самостоятельной работы

1. Усложнить рассмотренную в качестве примера задачу, создав небольшую библиотеку курсоров различной формы и предусмотрев поворот курсора при перемещении его в горизонтальном или диагональном направлении, а также при его движении вниз.

   Выбор нужной точки экрана обычно выполняется подводом курсора к этой точке и нажатием клавиши <Enter>. Иногда бывает полезно видеть и предыдущую выбранную точку — последнюю точку, зафиксированную клавишей <Enter>, и новую точку, на которую указывает курсор. Для этого используются метод "резиновой" нити и метод "резинового" прямоугольника:

   а) в методе "резиновой" нити один конец отрезка зафиксирован и указывает последнюю выбранную точку, второй конец перемещается в соответствии с изменением указываемой точки;

   б) в методе "резинового" прямоугольника один угол прямоугольника зафиксирован и указывает последнюю выбранную точку, а противолежащий угол перемещается в соответствии с изменением указываемой точки.

   Построить окружность по двум заданным точкам: центру и одной из точек окружности. Обе точки указываются с клавиатуры по методу "резинового" прямоугольника. Составить программу для управления размерами окружности и ее положением на экране. Управление выполняется клавишами:

   - < > > увеличивает радиус окружности;

- < < > уменьшает радиус окружности;
- клавиши управления курсором вызывают перемещение окружности в соответствующем направлении;
- <Enter> завершает работу программы.

2. Получить на экране какую-либо фигуру и "оживить" ее: пусть цыпленок летает и ходит, рыба плывет, подводная лодка поднимает и опускает перископ, сова машет крыльями, стрелки будильника и колесо вращаются, велосипед катится, телефонный диск крутится и так далее.
3. В рисованных мультфильмах иллюзия движения создается последовательной сменой кадров, каждый из которых фиксирует очередное положение движущегося объекта. Используя этот принцип, получить мультфильм, показывающий:

   а) бегущего человечка;

   б) идущего человечка;

   в) человечка, выполняющего приседания;

   г) человечка, выполняющего сигнализацию флажком.

   Для построения отдельных кадров мультфильма воспользоваться фигурками, описанными в рассказе Конан Дойля "Пляшущие человечки".
4. Аналогично предыдущей задаче получить спортивный мультфильм:

   а) о метании диска;

   б) о беге с барьерами;

   в) о подтягивании на перекладине;

   г) о прыжках в длину;

   д) о гребле;

   е) о поднятии штанги.

   Построение отдельных кадров выполнить на основе олимпийской символики.
5. Смоделировать компоненты Delphi. Каждый компонент должен являться объектом и быть размещен в отдельном модуле. Должна быть написана программа, которая демонстрирует работу этого компонента. Вместо событий использовать свойства и методы. Моделируемые компоненты:
   - Edit, MaskEdit;
   - RadioButton, RadioGroup;
   - ListBox;
   - ComboBox;
   - StringGrid;
   - OpenDialog;

- SaveDialog;
- ColorDialog (ColorGrid);
- FindDialog (FindNext);
- Calendar;
- SpinButton, SpinEdit.

6. "Жизнь". Игра моделирует жизнь поколений гипотетической колонии живых клеток, которые выживают, размножаются или погибают в соответствии со следующими правилами. Клетка выживает тогда и только тогда, когда она имеет двух или трех соседей из восьми возможных. Если у клетки только один сосед или ни одного, она погибает в изоляции. Если клетка имеет четырех или более соседей, она погибает от перенаселения. В любой пустой позиции, у которой ровно три соседа, в следующем поколении появляется новая клетка.
7. Составить программу для обучения работе с клавиатурой. Программа должна выдавать на экран буквы, цифры, слова и фразы, которые следует набрать на клавиатуре.
8. Составить программу для тренировки памяти. Программа должна высветить на экране несколько точек, играющий — указать, в каком порядке эти точки были высвечены. Координаты точек выбираются в программе с помощью датчика случайных чисел.
9. "Сбей самолет". По экрану летят вражеские самолеты. Цель — сбить их. Пусковая установка находится в нижней строке экрана. Пусковую установку можно перемещать по строке вперед и назад.
10. Написать программу, реализующую простейший вариант игры "Тетрис".
11. Библиотека графических примитивов содержит точку, круг, прямоугольник, линию, треугольник, ломаную, дугу. Реализовать возможность создания с помощью этих графических примитивов картинки. Предусмотреть выбор цвета фона и элементов картинки, а также возможность размещения картинки в кадре. Для этого создать специальный объект "рамка", который может охватывать и сохранять сцену.
12. "Морской бой". Есть пушка. Пользователь может управлять направлением ствола пушки, может стрелять. Если пушка выстрелила, то летит снаряд. В это время где-то по экрану плывет кораблик, в который играющий должен попасть. Если снаряд попадает в кораблик, то он взрывается и кораблик исчезает. И так, например, 10 раз. Если пользователь успешно использовал все выстрелы, то ему в качестве приза предлагается еще три выстрела.
13. Реализовать задание 1, дополнив его возможностью управлять курсором с помощью мыши.

14. Реализовать построение двуполостного гиперболоида $\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=-1$, c — действительная полуось, a и b — мнимые полуоси. Пусть $a = b$, тогда гиперболоид может быть получен вращением гиперболы с полуосями a и c вокруг оси $2 \cdot c$. Иерархия объектов: точка → гипербола → гиперболоид.

15. Реализовать построение эллиптического параболоида $z=\frac{x^2}{a^2}+\frac{y^2}{b^2}$. Пусть $a = b$, тогда имеем параболоид вращения, получаемый при вращении параболы $z = x^2 / a^2$, лежащей в плоскости x, z вокруг ее оси. Иерархия объектов: точка → парабола → параболоид.

16. Описать семейство объектов, описывающих файлы различных типов и действия над ними. Упрощенно первичный объект можно представить как совокупность данных:
    - длина файла;
    - текущая позиция указателя в файле;
    - дата создания файла;

    и совокупность методов работы с файлами:
    - прочитать байт из файла;
    - записать байт в файл;
    - установить дату создания файла.

    а) Написать программу, с помощью которой осуществляется форматированный ввод чисел.

    б) Написать программу, с помощью которой осуществляется форматированный ввод булевских значений.

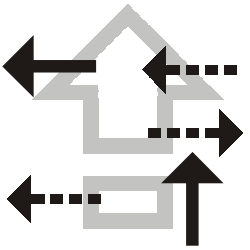
    в) Написать программу, с помощью которой осуществляется форматированный ввод строк.

17. Написать программу, реализующую некоторые функции работы с электронной таблицей, например, суммирование значений по строкам, столбцам, использование простейших формул и т. д.

18. Реализовать построение динамического списка, информационное поле которого содержит ссылки на объекты разных уровней иерархии. Описать необходимые подпрограммы для работы с таким списком.

19. Описать объект ЭЛЕКТРОННАЯ КНИГА, предполагая его объектом нижнего уровня иерархии, например, раздел-глава-часть и т. п. Придумать и описать набор методов такого объекта для реализации эффективной работы с электронной книгой.

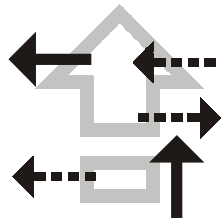
20. Описать на простейшем уровне объект СКЛАД.



ЧАСТЬ II

АЛГОРИТМЫ КОМПЬЮТЕРНОЙ ОБРАБОТКИ ДАННЫХ

Глава 3



Рекурсивные алгоритмы

Рекурсия — это такой способ организации вычислительного процесса, при котором подпрограмма (процедура или функция) в ходе выполнения составляющих ее операторов обращается сама к себе. Возможность рекурсивного обращения обеспечивается тем, что память под параметры и локальные переменные резервируется в стеке. Например, если процедура *A* имеет параметр *x* и вызывает саму себя, то параметр *x* вновь вызываемой процедуры *A* станет совершенно другим параметром; а при возвращении к первому вызову процедуры *A* прежнее значение переменной *x* будет восстановлено.

Максимальное число рекурсивных вызовов процедуры или функции без возвратов, которые происходят во время выполнения программы, называется глубиной рекурсии.

Число рекурсивных вызовов в каждый конкретный момент называется текущим уровнем рекурсии.

В практических приложениях важно не только убедиться в том, что максимальная глубина рекурсии конечна, но и в том, что она достаточно мала, чтобы не вызвать переполнение стека.

Главное требование к рекурсивным подпрограммам: в их определении должна присутствовать проверка завершения рекурсивного вызова. Структура конечного рекурсивного алгоритма может быть описана следующим образом:

```
если условие
  то      действие, выполняемое без рекурсивного вызова
  иначе   действие, выполняемое с рекурсивным вызовом
```

Во многих случаях для того, чтобы выполнение рекурсивной подпрограммы завершилось, достаточно наличия в подпрограмме некоторой неотрицательной (неположительной) величины, которая строго убывает (возрастает) при каждом рекурсивном вызове. Если значение этой величины стало равным нулю, вызовы рекурсивной функции прекращаются. В простых случаях роль такой управляющей величины играет один из параметров подпрограммы.

Структура рекурсивной процедуры Rec, включающей некоторое множество операторов и один или несколько операторов рекурсивного вызова, может иметь три разные формы.

1. С выполнением множества операторов S до рекурсивного вызова, то есть с выполнением действий на рекурсивном спуске:

```
Procedure Rec;
Begin
  S;
  if условие then Rec
End;
```

2. С выполнением действий S после рекурсивного вызова, то есть на рекурсивном возврате:

```
Procedure Rec;
Begin
  if условие then Rec;
  S;
End;
```

3. С выполнением действий S1 до рекурсивного вызова и S2 после рекурсивного вызова, то есть как на рекурсивном спуске, так и на рекурсивном возврате:

```
Procedure Rec;
Begin
  S1;
  if условие then Rec;
  S2
End;
```

или

```
Procedure Rec;
Begin
  if условие then
    begin
      S1;
      Rec;
      S2
    end;
End;
```

На практике находят применение все эти формы. Многие задачи допускают использование любой формы рекурсивной процедуры. Однако существуют

классы задач, например, задачи, использующие стековые и древовидные структуры данных, которые требуют от программиста умения направлять ход рекурсии. В таких задачах одни действия можно выполнить только на спуске, другие — только на возврате.

Рекурсивная форма организации алгоритма дает, как правило, более компактный текст программы, чем итерационная. Но рекурсия работает медленнее и может вызвать переполнение стека. Поэтому Н. Вирт [2] рекомендует применять рекурсивные подпрограммы для реализации алгоритмов, рекурсивных по своей природе, где обрабатываемые данные определяются в терминах рекурсии. Большой класс таких задач представляют комбинаторные задачи, решаемые полным или ограниченным перебором.

Рекурсивные подпрограммы имеют одну из двух форм: прямую рекурсию и косвенную рекурсию. В первом случае подпрограмма содержит оператор вызова этой же подпрограммы. Во втором — подпрограмма вызывает какую-либо другую подпрограмму, которая либо сама, либо посредством других подпрограмм вызывает исходную. В случае косвенной рекурсии вводится опережающее описание с помощью ключевого слова `FORWARD` (вперед).

Задачи, программы

1. Дано следующее описание линейного списка:

```
type
  TElem = integer;          {Тип информационной части}
  TList = ^TNode;           {Представление списка}
  TNode = record            {Звено списка}
            Info: TElem;    {Информационная часть}
            Next: Tlist     {Следующий элемент списка}
          end;
```

Написать рекурсивную процедуру удаления элемента из списка (листинг 3.1).

Листинг 3.1. Рекурсивная процедура удаления элемента из списка

```
function BList_DelElem2(var L: TList; E: TElem): boolean;
var N: TList; {указатель на удаляемое звено списка}
begin
  if L <> nil
  then {если список не пуст}
    if L^.Info = E
    then
```

```
      {если первое звено является удаляемым}
      begin
         {запоминаем указатель на удаляемое звено}
         N:=L;
         L:=L^.Next; {удаляем звено из списка}
         dispose( N ); {освобождаем память}
         List_DelElem2:=true
      end
    else
       BList_DelElem2:=BList_DelElem2( L^.Next, E )
  else BList_DelElem2:=false;
end;
```

2. Написать рекурсивную процедуру для печати линейного списка.
3. Написать рекурсивную процедуру, которая печатает элементы двунаправленного списка в прямом и обратном порядке.
4. Написать рекурсивную функцию для нахождения биномиальных коэффициентов

$$C_m^n = \begin{cases} 1, \text{ при } m = 0, n > 0 \text{ или } m = n \geq 0 \\ 0, \text{ при } m > n > 0 \\ C_{m-1}^{n-1} + C_m^{n-1}, \text{ иначе} \end{cases}$$

   Для заданного M вычислить все C_i^j, $M \geq i \geq j > 0$.

5. Задано конечное множество имен жителей некоторого города, причем для каждого из жителей перечислены имена его детей. Жители *А* и *Б* называются родственниками, если:

   а) либо *А* — ребенок *Б*,

   б) либо *Б* — ребенок *А*,

   в) либо существует некий *В* такой, что *А* является родственником *В*, а *В* является родственником *Б*.

   Перечислить все пары жителей города, которые являются родственниками (листинг 3.2).

Листинг 3.2. Родственники

```
Program Task3;
Uses Crt;
Type tName = (Nik, Pit, Kat, Mary);
```

```
      tRelative = array [tName,tName] of boolean;
      tSet =  array [tName] of boolean;
Const FirstName = Nik;
      LastName  = Mary;
      names : array [tName] of string =
              ('Коля','Петя','Катя','Маша');
Var   R : tRelative;
      name1, name2 : tName;
      ch : char;
      S : tSet;
{Рекурсивная функция, которая возвращает значение true,
если два человека с именами n1 и n2 являются родственниками,
и false — в противном случае}
Function Relative (n1, n2: tName; S : tSet): boolean;
var n : tName;
    fl : boolean;
begin
  if R[n1,n2] or R[n2,n1] then
     Relative := true
  else
    begin
      S[n1]:=true;
      s[n2]:=true;
      fl:=false;
      n:=FirstName;
      if not S[n] and Relative(n1,n,S) and Relative(n,n2,S) then
        fl:=true
      else
        repeat
          n:=succ(n);
          fl:= not S[n] and Relative(n1,n,S) and Relative(n,n2,S);
        until fl or (n=LastName);
      Relative:=fl
    end;
end;

{главная программа}
begin
```

```
  ClrScr;
  {обнуление исходных массивов}
  for name1:=FirstName to LastName do
    begin
      S[name1]:=false;
      for name2:=FirstName to LastName do
        R[name1,name2]:=false;
    end;
  {ввод исходных данных}
  for name1:=FirstName to LastName do
    begin
      writeln ('Является ли указанный человек ребенком',
               ' человека с именем ',names[name1],'(Y/N)- ?');
      for name2:=FirstName to LastName do
      if (name1<>name2) and (not Relative(name1,name2,S))then
        begin
          write(names[name2],' - ');
          readln(ch);
          R[name1,name2]:=(ch='Y')or(ch='y');
        end;
    end;
  {обработка и печать результата}
  writeln;
  writeln('Родственники:');
  writeln;
  for name1:=FirstName to pred(LastName) do
    for name2:=succ(name1) to LastName do
      if Relative(name1,name2,S) then
        writeln(names[name1],' - ',names[name2]);
  readln;
end.
```

6. Подсчитать количество различных представлений заданного натурального N в виде суммы не менее двух попарно различных положительных слагаемых. Представления, отличающиеся порядком слагаемых, различными не считаются. Решение дано в листинге 3.3.

Листинг 3.3. Решение задачи о различных представлениях натурального числа в виде суммы положительных слагаемых

```
Program CountTerm;
Uses Crt;
Const Str=' представить в виде суммы различных слагаемых ';
var n : integer;
    c : integer;
{Рекурсивная функция подсчета.
n - число, для которого ведется подсчет,
first - первое слагаемое из суммы
flag - вспомогательный параметр, который показывает,
есть ли хоть одно слагаемое в сумме.}
function Count (n, first: integer; flag: boolean): integer;
  var k, i, c: integer;
begin
  if n < first then Count:=0
  else
    begin
      c:=ord(flag);
      for i:=first to n div 2 do
        c:=c+Count(n-i,i+1,true);
      Count:=c
    end;
end;

begin {Главная программа}
  ClrScr;
  writeln ('Введите натуральное число');  readln (n);
  c:=Count(n,1,false);
  if c=0 then writeln (n,' нельзя',str)
         else writeln (n,' можно',str,c,' способами');
  readln;
end.
```

7. Вычислить определитель матрицы, пользуясь формулой разложения по первой строке

$$\det A = \sum_{k=1}^{n} (-1)^{k+1} A_{1,k} \det B_k,$$

где матрица B получается вычеркиванием из A первой строки и k-го столбца.

8. Для заданного целого n вычислить значение суммы

$$\sum_{i1=1}^{n}\sum_{i2=1}^{n}....\sum_{in=1}^{n}\frac{1}{i_1+i_2+i_3+...+i_n}$$ (листинг 3.4).

Листинг 3.4. Вычисление значения суммы

```
Program Summa;
var n:integer;

function Sum(k, n : Integer; s: real):real;
var i: integer; c: real;
begin
  c:=0;
  if k=n then
    for i:=1 to n do c:=c+1/(s+i)
  else
    for i:=1 to n do c:=c+Sum(k+1,n,s+i);
  Sum:=c
end;

begin
  writeln('Введите натуральное число'); readln(n);
  writeln('Результат равен ', Sum(1,n,0):5:3); readln
end.
```

9. Используя рекурсивную процедуру, напечатать следующую картинку

```
999.....999   (18 раз)
 88.....88    (16 раз)
  7.....7     (14 раз)
    .....
      2222     (4 раза)
       11      (2 раза)
      2222     (4 раза)
    .....
  7.....7     (14 раз)
 88.....88    (16 раз)
999.....999   (18 раз)
```

10. Построить синтаксический анализатор понятия <Идентификатор> (листинг 3.5).

<Идентификатор>::=<буква>|<Идентификатор>(<цифра>|<буква>);

Листинг 3.5. Синтаксический анализатор понятия <Идентификатор>

```
program Identificator;
const Mlen = 20; {максимальная длина идентификатора}
var   Ch : char;
       Id : array[1..Mlen] of char;
       IdLen, i : integer;

procedure Idd;
begin
  if Ch in ['A'..'Z', 'a'..'z', '0'..'9'] then
    begin
       IdLen := IdLen + 1;
       Id[IdLen] := Ch;
       read( Ch ); Idd
     end;
end;{ Idd }

procedure Ident;
begin
  if Ch in ['A'..'Z', 'a'..'z'] then
    begin
       IdLen := 1;
       Id[1] := Ch;
       read( Ch );
       Idd
     end;
end;{ Ident }

begin
  read( Ch );
  IdLen := 0;
  Ident;
  Write( 'Идентификатор: ' );
```

```
  for i := 1 to IdLen do write( Id[i] );
  readln
end.{Identificator}
```

11. Построить синтаксический анализатор для понятия <простое_выражение>.

```
<простое_выражение> ::= <простой_идентификатор> |
  (<простое_выражение> <знак_операции> <простое_выражение>).
<простой_идентификатор> ::= <буква>.
<буква> ::= a | b | c | … | z .
<знак_операции> ::= + | - | *.
```

12. Проверить, является ли вводимая последовательность символов <вещ_числом>.

```
<вещ_число> ::= <целое_число>.<целое_число> |
                <целое_число>.<целое_без_знака>E<целое_число> |
                <целое_число>E<целое_число>.
<целое_без_знака> ::= <цифра>[<цифра>].
<целое_число> ::= [+|-]<целое_без_знака>.
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
```

13. Написать процедуру, которая по заданному простому логическому выражению вычисляет его значение.

```
<простое_логическое > ::=  true | false | <идентификатор> |
   not <простое_логическое >|
   (<простое_логическое> <знак_операции> <простое_логическое>).
<идентификатор> ::= a | b | c | … | z.
<знак_операции> ::= and | or.
```

14. Написать программу, которая по заданному правильному простому логическому выражению (см. условие предыдущей задачи) вычисляет значение этого выражения. Считать, что значения переменных хранятся в массиве `value: array ['a'..'z'] of Boolean`.

15. Расставить на шахматной доске восемь ферзей таким образом, чтобы ни один не угрожал другому.

16. Получить расстановки восьми ладей на шахматной доске, при которых ни одна ладья не угрожает другой.

17. Получить все перестановки элементов 1, ..., 6.

18. Получить все размещения из 10 элементов 1, 2, ..., 10 по 3 в каждом. Размещением называется выборка из n указанных элементов m неповторяющихся элементов.

19. На шахматной доске определить поля, в которые может попасть конь за n ходов из указанной позиции.
20. Имеются n городов. Некоторые из них соединены дорогами известной длины.
    а) Найти кратчайшие маршруты из заданного города в остальные.
    б) Найти кратчайший маршрут, начинающийся в заданном городе и проходящий через все остальные.
21. Найти расстановку пяти ферзей, при которой каждое поле шахматной доски будет находиться под ударом хотя бы одного из них.
22. "Задача о рюкзаке". Имеется M различных предметов, известны вес каждого предмета и его стоимость. Определить, какие предметы надо положить в рюкзак, чтобы общий вес не превышал заданной границы, а общая стоимость была максимальной.
23. Даны целое n от 2 до 20 и вещественное $\epsilon > 0$. Найти с точностью ϵ все корни n-го многочлена Чебышева $T_n(x)$, определяемого формулами

$$T_0(x) = 1;\ \ T_1(x) = x;\ \ T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x),\ \ (k = 2, 3, ...).$$

Примечание

Многочлен $T_k(x)$ имеет k различных корней в интервале (–1 , 1); если $x_1 < x_2 < \ldots < x_k$ — корни многочлена $T_k(x)$, то многочлен $T_{k+1}(x)$ имеет по одному корню в каждом из интервалов $(-1, x_1)$, (x_1, x_2), … , $(x_k, 1)$.

24. "Ханойские башни". Имеются три колышка A, B, C и n дисков разного размера, перенумерованных от 1 до n в порядке возрастания их размеров. Сначала все диски надеты на колышек A в виде пирамиды. Требуется перенести все диски с колышка A на колышек B, соблюдая следующие правила: диски можно переносить только по одному, диск большего размера нельзя ставить на меньший. Диск C можно использовать в качестве промежуточного.

    Решение задачи удобно сформулировать в виде рекурсии (листинг 3.6): для того, чтобы переместить n дисков с башни A на башню B, нужно:

    а) переместить $(n - 1)$ диск с башни A на башню C;

    б) переместить диск с номером n с башни A на башню B;

    в) переместить $(n - 1)$ диск с башни C на башню B.

    Таким образом, решение задачи перемещения n дисков выражается через решение задачи перемещения $n - 1$ диска.

Листинг 3.6. Решение задачи "Ханойские башни"

```
program Hanoi;
var
  Tow1, Tow2, Tow3: char;
  n: integer; {количество дисков}
{процедура перемещения дисков с башни t1 на башню t2,
башня t3 - вспомогательная}
procedure Hanoi_Tower(t1, t2, t3: char; k: integer);
begin
  if k<>1 then
    begin
       Hanoi_Tower(t1, t3, t2, k-1);
       writeln(t1:10, t2:10);
       Hanoi_Tower(t3, t2, t1, k-1)
    end
  else
    writeln(t1:10, t2:10)
end; { Hanoi_Tower }

begin
  write('Введите количество дисков: ');
  readln(n);
  writeln('с диска':10, '- на диск':10);
  Tow1 := 'A';
  Tow2 := 'B';
  Tow3 := 'C';
  Hanoi_Tower(Tow1, Tow2, Tow3, n);
  write('Нажмите клавишу <Enter>...');
  readln
end. {Hanoi}
```

25. Написать программу перевода инфиксной записи арифметического выражения в постфиксную.
26. Написать программу вычисления выражения, записанного в постфиксной форме.
27. Дана строка текста, оканчивающаяся точкой. Напечатать этот текст в обратном порядке.

28. Описать рекурсивную логическую функцию, которая проверяет, является ли симметричной часть заданной строки, начинающаяся *i*-ым и кончающаяся *j*-ым символом.

29. Напишите две функции вычисления *i*-го числа Фибоначчи, рекурсивную и нерекурсивную, и напечатайте таблицу для сравнения времени вычисления *i*-ого числа.

30. Функция *f(n)* определена для целых чисел следующим образом

$$f(n) = \begin{cases} 1, & n = 1 \\ \sum_{2}^{n} f(n \text{ div } 2), & n \geq 2 \end{cases}.$$

Вычислить *f(k)*, *k = 15*, *20*, *30*, *40*.

31. Проверить, является ли вводимая последовательность символов <константным выражением>.

```
<конст_выр>::=<целое_без_знака>|
              (<целое_без_знака><знак_операции><конст_выр>).
<целое_без_знака>::=<цифра>[<цифра>].
<знак_операции>::= + | — | *.
<цифра>::=0|1|2|3|4|5|6|7|8|9.
```

32. Написать программу, которая по заданному правильному константному выражению (см. условие предыдущей задачи) либо вычисляет значение этого выражения, либо печатает сообщение об ошибке, если происходит переполнение в результате вычислений, то есть если значение выражения не принадлежит заранее известному диапазону допустимых значений.

33. Проверить, является ли вводимая последовательность символов <списком_параметров>.

```
<список_параметров>::=<параметр>[,<параметр>].
<параметр>::=<имя>=<цифра><цифра>|<имя>=(<список_параметров>).
<имя>::=<буква>|<буква>|<буква>.
<цифра>::=0|1|2|3|4|5|6|7|8|9.
<буква>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z.
```

34. Проверить, соответствует ли вводимая последовательность символов понятию <скобки>.

```
<скобки>::=<круглые> | <квадратные>.
<круглые>::=(<квадратные><квадратные>) | + .
<квадратные>::=[<круглые><круглые>] | — .
```

35. Проверить, соответствует ли вводимая последовательность символов понятию <Список_списков>.

```
<список_списков>::=<список>[;<список>].
```

```
<список>::=<элемент>[,<элемент>].
<элемент>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z.
```

36. Проверить, соответствует ли вводимая последовательность символов понятию <скобки>.

```
<скобки>::=<круглые>|<квадратные>.
<круглые>::=A|((<круглые>)[<квадратные>]).
<квадратные>::=B|[[< квадратные >](<круглые>)].
```

37. Проверить, является ли вводимая последовательность символов формулой следующего вида (листинг 3.7):

```
<формула>::=<терм>|<функция>|(<формула><знак_операции><формула>).
<функция>::=<имя_функции>(<формула>).
<имя_функции>::=sin|cos.
<терм>::=<целое>|<имя>.
<знак_операции>::=+| - | * | /.
<целое>::=[+| -]<цифра>[<цифра>].
<имя>::=<буква>[<буква>].
<цифра>::=0 |1 |2 |3 |4 |5 |6 |7 |8 |9 .
<буква>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z.
```

Листинг 3.7. Проверка соответствия последовательности символов заданной спецификации

```
Const TerminateChar=#0;
      Digit = ['0'.. '9'];
      Letter = ['a'.. 'z'];
      Sign = ['+', '-', '*', '/'];

Var   ch: char;
      Error: boolean;

{Функция, считывающая следующий символ}
procedure NextChar;
begin
  if eoln then
    ch:=TerminateChar
  else
    read(ch)
end;
```

```
{Процедура пропуска пробелов}
procedure SkipSpaces ;
begin
  while (ch=' ') do NextChar
end;

{Функция, считывающая идентификатор}
Function ReadIdent: string;
var w: string;
begin
  w:=ch;
  NextChar;
  while ch in Letter do
    begin
      w:=w+ch;
      NextChar
    end;
  SkipSpaces;
  ReadIdent:=w
end;

{Функция для пропуска символов}
procedure SkipSym (c: char);
begin
  if ch<>c then Error:=true;
  NextChar;
  SkipSpaces;
end;

{Функция проверки, является ли строка w именем функции cos или sin}
Function IsFuncName (w: string): boolean;
begin
  IsFuncName:=(w='sin')or(w='cos')
end;

{Функция для пропуска цифр}
procedure SkipDigit;
begin
```

```
  if ch in ['+', '-'] then NextChar;
  if not (ch in Digit) then
    Error:=true
  else
    begin
      while (ch in Digit) do NextChar;
      SkipSpaces;
    end;
end;

{Процедура проверки формулы}
procedure CheckFormula;
var w: string;
begin
  if not Error then
    begin
      SkipSpaces;
      if ch in Digit+['+','-'] then
        SkipDigit
      else
        if ch in Letter then
           begin
             w:=ReadIdent;
             if IsFuncName(w) then
               begin
                 SkipSym('(');
                 CheckFormula;
                 SkipSym(')')
               end;
           end
        else
          if ch='(' then
           begin
             SkipSym('(');
             CheckFormula;
             if ch in Sign then
               NextChar
```

```
            else
              Error:=true;
            CheckFormula;
            SkipSym(')');
          end
        else Error:=true;
      SkipSpaces;
    end;
end;

{Основная программа}
begin
  ch:=' ';
  writeln ('Input formula ');
  CheckFormula;
  SkipSpaces;
  Error:=Error or (ch<>TerminateChar);
  if Error then
    writeln('not ok')
  else    writeln('ok');
  readln;
  readln;
end.
```

38. Вводимая последовательность символов представляет собой формулу следующего вида

```
<формула> ::= <цифра>|
              M(<формула>,<формула>)|m(<формула>,<формула>).
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
```

M означает функцию *max*, а *m* — *min*. Вычислить значение данной формулы (листинг 3.8).

Листинг 3.8. Вычисление формулы с функциями *m*(min) и *M*(max)

```
Program Formula;
Uses Crt;
Const TerminateChar=#0; {Признак конца текста}
var ch: char; {Текущий символ}
```

```
    result: char; {Результат формулы}

{Процедура, которая считывает следующий символ}
procedure NextChar;
begin
  if eoln then
      ch:=TerminateChar
  else
    read(ch)
end;

{Процедура, которая считывает следующий значащий символ}
procedure NextSym;
begin
  NextChar;
  while (ch=' ') do NextChar;
end;

{Вывод сообщения об ошибке}
procedure Error;
begin
  writeln('Ошибка в выражении');
  repeat
  until KeyPressed;
  halt(255)
end;

{Рекурсивная функция подсчета значения выражения}
function Calc: char;
  var res: char; {результат}
      op1, op2: char; {операнды}
      op: char; {знак операции}
begin
  NextSym;
  if ch=TerminateChar then Error
  else
    if (ch>='0') and (ch<='9') then res:=ch
    else
```

```
        if (ch<>'m') and (ch<>'M') then Error
        else
          begin
            op:=ch;
            NextSym;
            if ch<>'(' then Error;
            op1:=Calc;
            NextSym;
            if ch<>',' then Error;
            op2:=Calc;
            res:=op1;
            if (op='M') and (op2>op1) or
               (op='m') and (op2<op1)
            then res:=op2;
            NextSym;
            if ch<>')' then Error;
          end;
      Calc:=res
  end;

  {Основная программа}
  begin
    ClrScr;
    writeln('Введите выражение');
    ch:=' ';
    result:=Calc;
    NextSym;
    if ch<>TerminateChar then Error;
    writeln('Результат равен ', result);
    repeat
    until KeyPressed
  end.
```

39. Проверить, удовлетворяет ли вводимая последовательность символов понятию <текст>.

```
<текст> ::= <элемент> | <элемент><текст>.
<элемент> ::= a | b| (<текст>) | [<текст>] | {<текст>}.
```

40. Реализовать рекурсивную процедуру копирования двоичного дерева (листинг 3.9).

Листинг 3.9. Рекурсивная процедура копирования двоичного дерева

```
type
  TElem = integer; {Тип информационной части}
  TTree = ^TNode;  {Представление дерева}
  TNode = record  {Узел дерева}
    Info: TElem;   {Информационная часть}
    Left: TTree;   {Указатель на левое поддерево}
    Right: TTree   {Указатель на правое поддерево}
  end;

{Функция создает копию двоичного дерева A}
{Результатом функции является указатель на копию дерева A}
function TreeCopy( A: TTree ): TTree;
var Tree: TTree; {Указатель на копию дерева}
begin
  if A <> nil then
    begin
       new( Tree ); {создаем новое звено дерева}
       Tree^.Info := A^.Info; {копируем информационную часть}
       {сохраняем указатель на копию левого поддерева}
       Tree^.Left := TreeCopy(A^.Left);
       {сохраняем указатель на копию правого поддерева}
       Tree^.Right := TreeCopy(A^.Right)
    end
   else Tree := nil;
     TreeCopy := Tree
end; {TreeCopy}
```

41. Подсчитать число листьев (терминальных элементов) в заданном двоичном дереве.
42. Определить высоту заданного двоичного дерева (листинг 3.10).

Листинг 3.10. Определение высоты двоичного дерева

```
type
  TElem = integer; {Тип информационной части}
```

```
  TTree = ^TNode;  {Представление дерева}
  TNode = record  {Узел дерева}
    Info: TElem;   {Информационная часть}
    Left: TTree;   {Указатель на левое поддерево}
    Right: TTree   {Указатель на правое поддерево}
  end;
{Функция вычисляет высоту}
{заданного двоичного дерева A}
function TreeHeight( A: TTree ): integer;
var   lh: integer; {высота левого поддерева}
      rh: integer; {высота правого поддерева}
begin
  if A = nil then
    TreeHeight := 0
  else
    begin
      lh := TreeHeight( A^.Left );
      rh := TreeHeight( A^.Right );
      if lh > rh then
        TreeHeight := lh + 1
      else
        TreeHeight := rh + 1
    end
end; {TreeHeight}
```

43. Описать рекурсивную функцию, которая по заданным границам a и b, заданной функции $f(x)$ и заданной точности ε, методом деления отрезка пополам находит с точностью ε корень уравнения $f(x) = 0$ на отрезке $[\, a\, ,\, b\,]$. Считать, что $\varepsilon > 0$, $a < b$, $f(a) \cdot f(b) < 0$, $f(x)$ — непрерывная и монотонная на $[\, a\, ,\, b\,]$.

44. Для заданных границ интегрирования a и b вычислить (листинг 3.11) значение определенного интеграла следующего вида

$$\int \sin^n x\,dx = \begin{cases} -\dfrac{\sin^{n-1} x \cos x}{n} + \dfrac{n-1}{n}\int \sin^{n-2} x\,dx, & n > 2 \\ \dfrac{x}{2} - \dfrac{1}{4}\sin 2x, & n = 2 \\ -\cos x, & n = 1 \end{cases}.$$

Листинг 3.11. Вычисление значения определенного интеграла

```
Program IntegralSin;
var a, b: real;
    n: integer;

function F1 (x : real): real;
begin
  F1:=(-1)*cos(x)
end;

function F2 (x : real): real;
begin
  F2:=x/2-sin(2*x)/4
end;

function F3 (x: real; n: integer): real;
var i: integer;
    res,c: real;
begin
  c:=sin(x);
  res:=cos(x);
  for i:=1 to n do res:=res*c;
  F3:=(-1)*res/n
end;

function IntSinN (a, b: real; n: integer): real;
begin
  if n=1 then
    IntSinN:=F1(b)-F1(a)
  else
    if n=2 then
      IntSinN:=F2(b)-F2(a)
    else
      IntSinN:=F3(b,n)-F3(a,n)+(n-1)/n*IntSinN(a,b,n-2)
end;

begin
  writeln('Input a,b (a<b)');
```

```
    readln (a,b);
    while (b<a) do
      begin
        writeln ('Error! Please repeat');
        readln(a,b)
      end;
    writeln('Input power of sin (n>0)');
    readln (n);
    while (n<=0) do
      begin
        writeln ('Error! Please repeat');
        readln(n)
      end;
    writeln('Result : ',IntSinN(a,b,n):7:5); readln;
  end.
```

Задания для самостоятельной работы

1. Описать рекурсивную функцию, которая по заданным вещественному x и целому n вычисляет величину x^n согласно формуле

$$x^n = \begin{cases} 1, n = 0 \\ 1/x^{|n|}, n < 0 \\ x \cdot \left(x^{n-1}\right), n > 0 \end{cases}.$$

2. Написать программу вычисления функции Аккермана для всех неотрицательных целых аргументов m и n

$$A(m,n) = \begin{cases} A(0, n) = n + 1 \\ A(m, 0) = A(m-1, 1), \quad m > 0 \\ A(m, n) = A(m-1, A(m, n-1)), \quad m, n > 0 \end{cases}.$$

3. Написать рекурсивную функцию, которая вычисляет $y = \sqrt[k]{x}$ по следующей формуле

$$y_0 = 1;\ y_{n+1} = y_n + \frac{\left(\frac{x}{y_n^{k-1}} - y_n\right)}{k},\ n = 0, 1, 2, \ldots$$

За ответ принять приближение, для которого выполняется условие $|y_{n+1} - y_n| < \varepsilon$, где $\varepsilon = 0.0001$.

4. По вещественному числу $a > 0$ вычислить величину $\dfrac{\sqrt[3]{a} - \sqrt[6]{a^2 + 1}}{1 + \sqrt[7]{3 + a}}$. Корни $y = \sqrt[k]{x}$ вычислять с точностью $\varepsilon = 0.00001$ по следующей итерационной формуле

$$y_0 = 1;\ y_{n+1} = y_n + \frac{\left(\dfrac{x}{y_n^{k-1}} - y_n\right)}{k},\quad n = 0, 1, 2, \ldots,$$

приняв за ответ приближение y_{n+1}, для которого $|y_{n+1} - y_n| < \varepsilon$.

5. Подсчитать число узлов в заданном двоичном дереве.
6. Подсчитать число узлов на k-ом уровне заданного двоичного дерева (корень считать узлом 1-го уровня).
7. Даны два текстовых файла A и B. Максимальная длина слова — 20 символов. Занести в файл C те слова файла A, которых нет в файле B. Для хранения слов файла B и ускорения поиска среди них воспользуйтесь деревом двоичного поиска.
8. Реализовать рекурсивную процедуру печати всех элементов заданного двоичного дерева.
9. Описать логическую функцию, проверяющую на равенство два заданных двоичных дерева.
10. Описать логическую функцию, определяющую, есть ли в заданном двоичном дереве хотя бы два одинаковых элемента.
11. Описать процедуру, которая для заданного N строит двоичное дерево с количеством уровней N, где на каждом уровне i располагаются узлы, информационные части которых равны i.
12. Описать рекурсивную функцию, для определения максимального элемента вектора, состоящего из вещественных чисел, введя вспомогательную рекурсивную функцию, зависящую от k, находящую минимум среди k последних элементов вектора.
13. Дана строка текста, оканчивающаяся точкой. Напечатать этот текст в обратном порядке.
14. Дана последовательность ненулевых чисел, за которой следует 0. Напечатать сначала все отрицательные члены этой последовательности, а затем положительные в любом порядке.
15. Описать рекурсивную логическую функцию, которая проверяет, является ли симметричной часть заданной строки, начинающаяся i-ым и кончающаяся j-ым символом.

16. Для заданных границ интегрирования a и b вычислить значение определенного интеграла следующего вида:

а) $$\int \cos^n x\,dx = \begin{cases} \dfrac{\cos^{n-1} x \sin x}{n} + \dfrac{n-1}{n}\int \cos^{n-2} x\,dx, & n > 2 \\ \dfrac{x}{2} + \dfrac{1}{4}\sin 2x, & n = 2 \\ \sin x, & n = 1 \end{cases};$$

б) $$\int \frac{dx}{\sin^n x} = \begin{cases} -\dfrac{1}{n-1}\cdot\dfrac{\cos x}{\sin^{n-1} x} + \dfrac{n-2}{n-1}\displaystyle\int \frac{dx}{\sin^{n-2} x}, & n \ge 2 \\ \ln tg\dfrac{x}{2}, & n = 1 \\ x, & n = 0 \end{cases};$$

в) $$\int \frac{dx}{\cos^n x} = \begin{cases} \dfrac{1}{n-1}\cdot\dfrac{\sin x}{\cos^{n-1} x} + \dfrac{n-2}{n-1}\displaystyle\int \frac{dx}{\cos^{n-2} x}, & n \ge 2 \\ \ln tg\left(\dfrac{\pi}{4} + \dfrac{x}{2}\right), & n = 1 \\ x, & n = 0 \end{cases};$$

г) $$\int x^n e^{ax} dx = \begin{cases} \dfrac{x^n e^{ax}}{a} - \dfrac{n}{a}\displaystyle\int x^{n-1} e^{ax} dx, & n > 1 \\ \dfrac{e^{ax}}{a^2}(ax - 1), & n = 1 \end{cases};$$

д) $$\int x^n a^{mx} dx = \begin{cases} \dfrac{x^n a^{mx}}{n \ln a} - \dfrac{n}{m \ln a}\displaystyle\int x^{n-1} a^{mx} dx, & n > 1 \\ \dfrac{x a^{mx}}{m \ln a} - \dfrac{a^{mx}}{m(\ln a)^2}, & n = 1 \end{cases};$$

е) $$\int e^{ax}\cos^n x\,dx = \begin{cases} \dfrac{e^{ax}\cos^{n-1}x\left(a\cos x + n\sin x\right)}{a^2+n^2} + \dfrac{n(n-1)}{a^2+n^2}\displaystyle\int e^{ax}\cos^{n-2}x\,dx,\ n \ge 2 \\ \dfrac{-e^{ax}\left(\sin x + a\cos x\right)}{a^2+n^2},\ n = 1 \\ \dfrac{e^{an}}{a},\ n = 0 \end{cases};$$

ж) $$\int e^{ax}\sin^n bx\,dx =$$

$$= \begin{cases} \dfrac{e^{ax}\sin^{n-1}bx\left(a\sin x - nb\cos bx\right)}{a^2+n^2b^2} + \dfrac{n(n-1)b^2}{a^2+n^2b^2}\displaystyle\int e^{ax}\sin^{n-2}bx\,dx,\ n \ge 2 \\ \dfrac{-e^{ax}\left(a\sin bx - b\cos bx\right)}{a^2+b^2},\ n = 1 \\ \dfrac{e^{an}}{a},\ n = 0 \end{cases};$$

з) $$\int \ln^n x\,dx = \begin{cases} x\ln^n x - n\displaystyle\int \ln^{n-1} x\,dx,\ n > 1 \\ x\ln x - x,\ n = 1 \end{cases};$$

и) $$\int x^m \ln^n x\,dx = \begin{cases} \dfrac{x^{m+1}}{m+1}\ln^n x - \dfrac{n}{m+1}\displaystyle\int x^m \ln^{n-1} x\,dx, n > 1 \\ x^{m+1}\left[\dfrac{\ln x}{m+1} - \dfrac{1}{(m+1)^2}\right], n = 1 \end{cases};$$

к) $$\int \frac{dx}{\left(a^2+x^2\right)^n} = \begin{cases} \dfrac{1}{2(n-1)a^2}\left[\dfrac{x}{\left(a^2+x^2\right)^{n-1}} + (2n-3)\displaystyle\int \frac{dx}{\left(a^2+x^2\right)^{n-1}}\right],\ n > 1 \\ \dfrac{1}{a}\cdot arctg\,\dfrac{x}{a},\ n = 1 \end{cases};$$

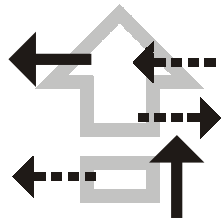
л) $$\int tg^n x\,dx = \begin{cases} \dfrac{tg^{n-1}x}{n-1} - \displaystyle\int tg^{n-2}x\,dx,\ n \ge 2 \\ -\ln\left|\cos x\right|,\ n = 1 \\ x,\ n = 0 \end{cases};$$

м) $$\int ctg^n x\,dx = \begin{cases} -\dfrac{ctg^{n-1}x}{n-1} - \int ctg^{n-2} x\,dx, & n \geq 2 \\ \ln|\sin x|, & n = 1 \\ x, & n = 0 \end{cases};$$

н) $$\int x^m \sin ax = \begin{cases} -\dfrac{x^m}{a}\cos ax + \dfrac{m}{a}\int x^{m-1}\cos ax\,dx, m \geq 1 \\ -\dfrac{\cos ax}{a}, m = 0 \end{cases};$$

о) $$\int x^m \cos ax = \begin{cases} \dfrac{x^m}{a}\sin ax + \dfrac{m}{a}\int x^{m-1}\sin ax\,dx, m \geq 1 \\ \dfrac{\sin ax}{a}, m = 0 \end{cases}.$$

Глава 4



Алгоритмы с возвратом

Особый класс составляют алгоритмы, которые ищут решения не по заданным правилам, а методом проб и ошибок. Обычно подобные алгоритмы используются в задачах искусственного интеллекта. Например, для систем искусственного интеллекта характерно так называемое недетерминированное управление. Такое управление необходимо потому, что знания в интеллектуальных системах часто накапливаются фрагментарно и это приводит к невозможности заранее определить цепочку логических выводов, в которых они используются. Необходимо методом проб и ошибок выбрать некоторую цепочку выводов, а в случае неудачи организовать перебор с возвратами для поиска другой цепочки и т. д.

Обычно процесс решения методом проб и ошибок разделяется на отдельные подзадачи. Часто эти подзадачи естественно выражаются в терминах рекурсии и, в свою очередь, требуют исследования конечного числа подзадач. Таким образом, процесс решения задачи представляется в виде поиска пути в пространстве состояний от исходного состояния до целевого путем повторения возможных преобразований. Для организации поиска в пространстве состояний удобно использовать дерево поиска или его более общую форму — граф. Во многих задачах дерево поиска растет очень быстро, рост его зависит от параметров задачи и часто бывает экспоненциальным. Стоимость поиска в этом случае достаточно высока. Иногда дерево поиска удается сократить, используя разнообразные эвристики, тем самым сводя затраты к разумным пределам.

Рассматриваемые в данном разделе задачи предполагают использование рекурсии при их решении.

Задачи, решение которых может быть представлено в виде алгоритма с возвратами (с использованием рекурсивной функции), часто представляют собой игровые ситуации, в которых из-за отсутствия аналитического решения требуемое решение получается после выполнения последовательности ходов.

Пример 1. Задача о восьми ферзях

Рассмотрим в качестве примера следующую задачу: на шахматной доске (размер 8 × 8) необходимо расставить восемь ферзей таким образом, чтобы ни один ферзь не находился под ударом другого. Это известный пример использования метода проб и ошибок. В 1850 г. задачу о восьми ферзях исследовал Г. Ф. Гаусс, но полностью он ее не решил.

Решение задачи должно быть представлено как набор координат восьми ферзей. Существует приблизительно 2^{32} способов 64! / (56! * 8!) расстановки восьми ферзей. На рис. 4.1 приведен пример одного из допустимых решений.

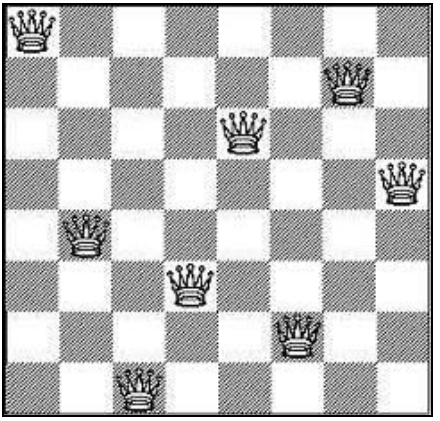


Рис. 4.1. Пример одного из допустимых решений расстановки восьми ферзей

Для того чтобы найти необходимое решение, достаточно перебрать все возможные расположения восьми ферзей, исключая те, в которых ферзи бьют друг друга. Процесс перебора решений можно представить в виде последовательного выбора позиции для каждого ферзя. Таким образом, правильное решение — это решение, которое представляет собой восемь "ходов", каждый их которых сделан с учетом того, что ни один из предыдущих ферзей не попадает под удар.

Для представления законченного решения достаточно следующих данных:

```
Const N = 8;
Var   F: array [1..N] of 1..N;
```

Каждый элемент массива *F[i]* содержит номер горизонтали для *i*-го ферзя, стоящего в *i*-й вертикали. Этого достаточно, так как два ферзя не могут находиться на одной вертикали. Для представленного решения содержимое массива будет следующим (1, 5, 8, 6, 3, 7, 2, 4).

Процесс решения задачи представляется функцией *Step*, осуществляющей *h*-й ход и возвращающей логическое значение "истина", если процесс расстановки ферзей с *h*-го хода по *N*-й был успешным (листинг 4.1).

Листинг 4.1. Задача о восьми ферзях

```
Const N = 8;
Var   F: array [1..N] of 1..N;

{пытаемся поставить h-ого ферзя в h-ую вертикаль}
Function Step (h : integer) : boolean;
Var I   : integer;
    Res : boolean;
Begin
  {предыдущие N ходов успешны, получено решение}
   If  (h = N+1) then
     Step := true
   Else
     Begin
       Res := false; {признак того, что решение найдено }
       I := 1;
       {пока не перебрали все варианты вдоль вертикали
        и не нашли решение}
       While (i<=N) and not Res do
         Begin
           {проверяем, является ли h-ый ход допустимым}
           If TestStep(h,i) then
             Begin
               {запоминаем позицию ферзя}
               F[h] := i;
               {пытаемся поставить остальных ферзей}
               Res := Step ( h+1);
             End;
          {переходим к следующему варианту
           расстановки для h-ого ферзя}
          I:=i+1;
         End;
      Step := Res;
```

```
    End;
End;

{Функция TestStep проверяет допустимость h-ого хода.
Ферзь "бьет" другого ферзя,
если располагается с ним на одной горизонтали, вертикали или диагонали.
В нашем случае нам гарантируется,
что ферзи расположены в разных вертикалях.
h - номер вертикали, pos - номер горизонтали}

Function TestStep (h, pos : integer) : Boolean;
Var I: integer;
    Res: Boolean;
Begin
  Res:=true;
  I:=1;
  While (i<h) and Res do
    {если 2 ферзя стоят на одной горизонтали}
    If (F[i]=pos) or
       {на вертикали идущей слева направо и снизу вверх}
       (F[i]-pos=i-h) or
       {на вертикали идущей слева направо и сверху вниз}
       (F[i]-pos=h-i) then
      Res := false
    Else
      i:=i+1;
  TestStep := Res;
End;

{главная программа}
Var i : integer;
Begin
  If  Step(1) then
    For i:=1 to N do
      Writeln(i,F[i]);
  Else
    Writeln ('решения не существует');
End.
```

На рис. 4.2 приведен фрагмент дерева поиска для задачи при $n = 4$.

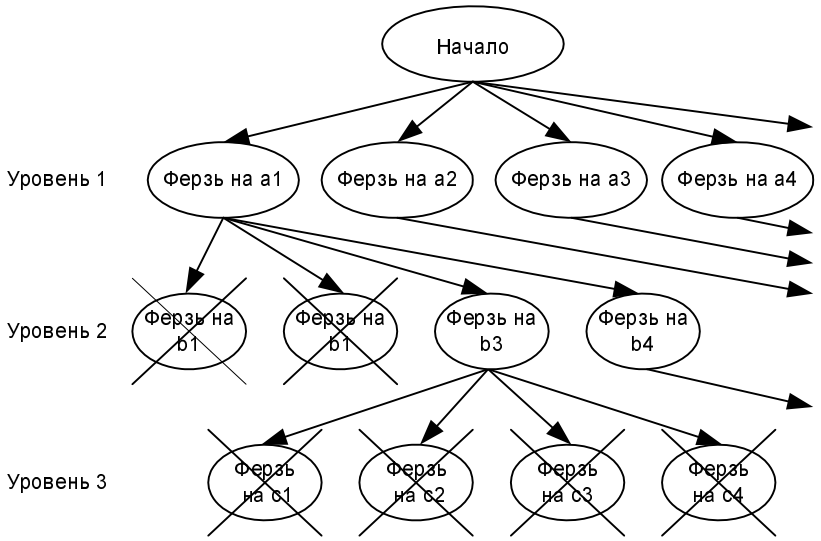


Рис. 4.2. Фрагмент дерева поиска для решения задачи о ферзях при $n = 4$

Дуги в этом дереве означают рекурсивные вызовы процедуры. Каждый узел соответствует пробной постановке ферзя на указанное место. Перечеркнутые узлы означают невозможность продолжения обхода дерева через данный узел, так как мы получили ситуацию, при которой один из ферзей находится под ударом. Решение нашей задачи представляет обход такого дерева. Такой процесс обхода дерева называют поиском в глубину. В случае если нас интересует единственное решение задачи или просто его возможность, то поиск прекращается при достижении узла дерева на n-ом уровне и выполнении условия, при котором ферзи не находятся под ударом. При подсчете всех вариантов на n-уровне необходимо увеличивать счетчик при нахождении очередного варианта.

Пример 2. Задача о костях домино

У игрока имеется набор костей домино (не обязательно полный). Найти последовательность выкладывания этих костей таким образом, чтобы получившаяся в результате цепочка была максимальной длины.

Определим структуры для представления исходных данных и результатов работы программы.

```
Const N = 28;                    {максимальное количество костей домино}
Type  domino = record            {одна кость домино}
```

```
        left, right : 0..6;        {значения}
        used: Boolean;             {использовалась ли кость/выставлена/}
      end;
Var
  F: array [1..N] of domino;       {набор костей домино у игрока}
  NF: integer;                     {количество костей у игрока}
  CurPos : array [1..N] of 1..N;   {текущий порядок выставления костей}
  BestPos : array[1..N] of 1..N;   {лучший порядок выставления костей}
  BestPosN : integer;              {количество костей в лучшем варианте}
```

Правило выставления костей состоит в следующем: на первом ходу на стол выставляется произвольная кость, затем к ней могут быть добавлены совпадающие по значению кости как слева, так и справа. Одна и та же кость может быть использована только один раз. Основная программа вместе с инициализацией значений будет выглядеть следующим образом:

```
begin
  writeln('Введите количество костей домино');
  readln(NF);
  for i:=1 to NF do
    begin
      write('Значения кости: '); readln( F[i].left,F[i].right);
      F[i].used := false;
   end;

  BestPosN := 0;
  for i:=1 to NF do
    begin
     CurPos[1] := i; {выставляем i-ую кость в качестве первой}
     F[i].used := true; {помечаем как использованную}
     {вызов процедуры выставления 2-ой, 3-ей и т. д. кости}
     Step(2,F[i].left,F[i].right);
     F[i].used := false;
   end;
   ...
```

Стоит обратить внимание на два момента в нашей программе. Первый — каждая кость, после того как она выставлена, помечается как использованная. Это необходимо для предотвращения зацикливания программы в результате повторного использования кости. Этот флаг должен быть обязательно снят после пробы, чтобы данная кость могла быть использована в цепочке на другом месте, кроме первого. Второй момент — для выставления

второй, третьей и т. д. кости мы будем использовать ту же процедуру, так как эти действия аналогичны. При этом должны быть известны значения на концах цепочки, которая образована на данный момент.

Процедура *Step* должна попробовать в качестве следующей каждую кость из всех неиспользованных. После каждого удачного выставления необходимо перейти к следующему шагу и выставить очередную кость в цепочке.

```
{пытаемся поставить очередную кость в цепочку с границами left, right}
Procedure  Step ( curn, left, right : integer );
Var I   : integer;
    Res : Boolean;
Begin
  {текущий вариант лучший по длине и мы его сохраняем}
  If  ( curn-1 > BestPosN )  then
    BestPos := CurPos;
  {просматриваем все кости как возможные варианты}
  for I := 1 to NF do
    if not F[i].used then
      begin{ пробуем поставить i-ую кость }
        F[i].used := true;
        CurPos[curn] := i;
        if F[i].left = left then {если кость подходит слева}
          Step(curn+1, F[i].right, right ); {то пробуем следующую}
        if F[i].left = right then
          Step(curn+1, left, F[i].right );
        if F[i].right = left then
          Step(curn+1, F[i].left, right );
        if F[i].right = right then
          Step(curn+1, left, F[i].left );
        F[i].used := false;
      end;
end;
```

После вызова этой процедуры в массиве BestPos будет получено лучшее из возможных решений. Это может быть даже цепочка только из одной кости. Остается только напечатать полученное решение.

```
for i:=1 to BestPosN do
  writeln( i:4, F[BestPos[i]].left:4, F[BestPos[i]].right:4 );
end.
```

Единственный недостаток нашей программы — мы не определяем, с какой стороны нужно на i-ом ходу поставить кость. Думаем, что читатели без труда решат эту проблему, изменив структуры, используемые в программе.

На рис. 4.3 приведен фрагмент дерева поиска решения для данной задачи.

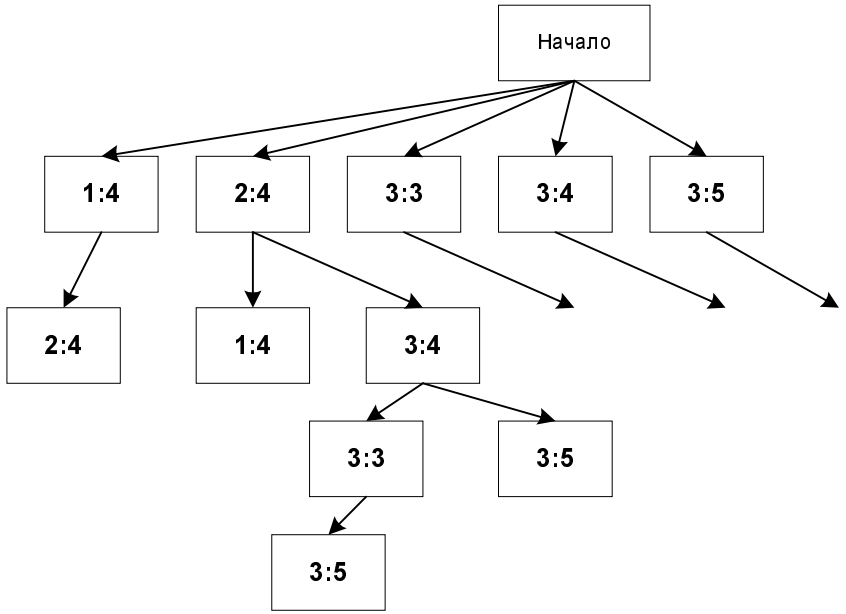


Рис. 4.3. Фрагмент дерева поиска решения для задачи о костях домино

Пример 3. Выход из лабиринта

Следующая задача представляет собой игровую ситуацию, когда некто, попавший в лабиринт (рис. 4.4), нуждается в подсказке для быстрого нахождения прохода по лабиринту в указанную точку. Представим лабиринт как поле 5×5, каждая клетка которого либо проходима (белый цвет), либо нет (черный цвет). Для удобства пронумеруем клетки поля аналогично шахматной доске. Рассмотрим теперь более точно нашу задачу: необходимо определить, можно ли из заданной клетки, скажем, *D5*, попасть в другую заданную клетку, скажем, *A1*, и если можно, то выдать путь минимальной длины, который эти клетки соединяет. Сразу нужно отметить, что решение нашей конкретной задачи не является оптимальным. Существует так называемый "волновой метод", который является более эффективным. Однако полученное нами решение демонстрирует основные принципы рекурсивного решения таких задач и будет являться основой при решении похожих задач.

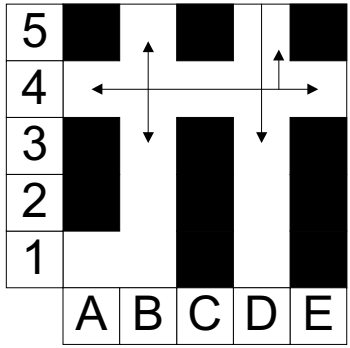


Рис. 4.4. Схема лабиринта

Рекурсивное решение задачи сводится к процедуре перебора всех возможных ходов игрока из текущей клетки в одну из соседних. Таких вариантов максимум четыре. При этом нужно помнить про проблему "зацикливания" нашей процедуры, связанной с повторным посещением клетки путем рекурсивного вызова. Во избежание зацикливания необходимо хранить информацию о тех клетках, в которых игрок уже побывал на одном из предыдущих ходов. Процесс решения задачи можно представить в виде обхода соответствующего дерева (рис. 4.5).

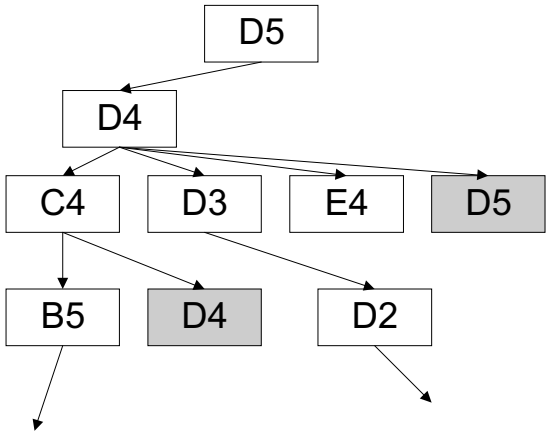


Рис. 4.5. Дерево поиска решения для задачи "Выход из лабиринта"

Серым окрашены узлы, в которых игрок был на одном из предыдущих ходов, и повторно посещать он их не должен.

Определимся с данными, нужными нам для решения задачи. В качестве ответа предлагается вывести последовательность координат клеток, которые

игрок должен пройти на пути к выходу. В процессе нахождения решения необходимо сохранять оптимальное решение и пройденный путь.

```
Const N1 = 'E'; (* размерность лабиринта *)
      N2 = 5;
Type Cell = record   (* координаты клетки *)
                  X: 'A'..N1;
                  Y: 1..N2;
            end;
Var (*лабиринт, true - проход разрешен *)
    Maze: array [1..N1, 1..N2] of Boolean;
    (* true, если клетка посещалась на предыдущем ходе *)
    MazeB: array [1..N1, 1..N2] of Boolean;
    (* текущий путь *)
    CurPath: array [1..N2*N2-1] of cell;
    (* кратчайший путь *)
    BestPath : array [1..N2*N2-1] of cell;
    (* количество ячеек в лучшем пути *)
    BestPathN : integer;
```

Теперь основная процедура. Напомним, что нам необходимо проверить все возможные варианты перехода из текущей клетки, при этом нужно избегать зацикливания и делать проверку вновь полученного решения на оптимальность.

```
{пытаемся сделать ход из текущей клетки, стараясь достичь конечной}
{CN - длина пройденного пути}
Procedure  Step ( cur: cell; endcell: cell; CN: integer );
Var Newcell : cell;
Begin
  {текущий вариант лучший по длине и мы его сохраняем}
  If  ( CN > BestPathN ) and ( cur = endcell)  then
    Begin
      BestPath  := CurPath;
      BestPathN := CN;
    End;
  If Maze[cur.X, cur.Y] and not MazeB[cur.X, cur.Y] then
    Begin  (* клетка свободна и в ней мы не были *)
      (* ставим признак, что в этой клетке мы были*)
      MazeB[cur.X, cur.Y] := true;
      CurPath[CN] := cur;
```

```
      If cur.X > 'A' then (* влево *)
        Begin
          Newcell.X := pred( cur.X);
          Newcell.Y := cur.Y;
          Step(Newcell, endcell, CN+1);
        End;
      If cur.X < N1 then (* вправо *)
        Begin
          Newcell.X := succ(cur.X);
          Newcell.Y := cur.Y;
          Step(Newcell, endcell, CN+1);
        End;
      If cur.Y > 1 then (* вниз *)
        Begin
          Newcell.X := cur.X;
          Newcell.Y := cur.Y - 1;
          Step(Newcell, endcell, CN+1);
        End;
      If cur.Y < N2 then (* вверх *)
        Begin
          Newcell.X := cur.X;
          Newcell.Y := cur.Y + 1;
          Step(Newcell, endcell, CN+1);
        End;
      (* снимаем признак, после ухода с клетки *)
      MazeB[cur.X,cur.Y] := false;
    End;
end;
```

После вызова этой процедуры в массиве BestPath будет получено лучшее возможное решение в виде координат клеток. Читателю предлагается разработать процедуру печати решения самостоятельно.

Задания для самостоятельной работы

1. Найти все вершины графа, недостижимые из заданной. Указание: использовать рекурсивную процедуру проверки доступности одной вершины из другой.
2. Раскрасить граф минимальным количеством цветов. Каждая вершина должна быть помечена цветом, отличным от цвета смежных вершин.

3. Определить, является ли связанным заданный граф. Указание: использовать рекурсивную процедуру проверки доступности одной вершины из другой.
4. Найти длину кратчайшего цикла в графе.
5. Определить вершину, удалением которой можно свести граф к дереву.
6. Найти вершину графа, которая принадлежит каждому пути между двумя заданными вершинами.
7. Задана система односторонних дорог. Найти путь, соединяющий города *А* и *Б*, не проходящий через заданное множество вершин.
8. Задана система двусторонних дорог. Найти замкнутый путь, проходящий через каждую вершину и длиной не более 100 км.
9. Найти в системе двусторонних дорог город, для которого сумма расстояний до других городов минимальна.
10. По системе двусторонних дорог определить, есть ли в ней город, из которого можно добраться в любой другой, проехав меньше 100 км. Разрешается построить дополнительно три дороги.
11. По системе двусторонних дорог определить, можно ли, закрыв какие-либо три из них, добиться того, чтобы из города *А* нельзя было попасть в город *Б*.
12. Задана система двусторонних дорог. *N*-периферией называется множество городов, расстояние от которых до выделенного города больше *N*. Определить *N*-периферию для заданного *N*.
13. Определить, изоморфны ли два графа.
14. Построить такой многоугольник (не обязательно выпуклый) с вершинами в заданном множестве, периметр которого максимален.
15. Найти минимальное множество прямых, на которых можно разместить все точки заданного множества.
16. В трехмерном пространстве задано множество точек. Найти разбиение этого множество на два непустых непересекающихся множества, чтобы их центры тяжести находились как можно ближе друг к другу.
17. В условии предыдущей задачи найти множество, содержащее ровно *N* точек, центр тяжести которого находится как можно ближе к началу координат.
18. Имеется *n* костей домино. Построить последовательность максимальной длины.
19. Дана последовательность костей домино. Определить последовательность ходов двух игроков, которые ведут к "рыбе" (ситуация, в которой у двух игроков есть кости, но ни один не может сделать ход).

20. Для шахматного поля размером $N \times N$, на котором расставлены черные и белые фигуры, найти наименьшее количество ферзей и их расстановку, при которой все поля доски, занятые фигурами противоположного цвета, находятся под ударом. Решить аналогичную задачу для следующих шахматных фигур:
    - конь;
    - слон;
    - ладья.
21. Найти кратчайший по количеству ходов путь ладьи, позволяющий попасть из одной клетки в другую. Известно расположение белых и черных фигур на доске. Фигуры противоположного цвета можно "бить".
22. Задана квадратная матрица размером $N \times N$, которая содержит целые числа от 0 до 6. Костью домино можно накрыть две соседние ячейки матрицы (горизонтальные или вертикальные), если числа на кости совпадают с числами в ячейках. Проверить, можно ли заданную матрицу накрыть одним комплектом костей домино.
23. Какое минимальное количество комплектов домино необходимо, чтобы построить заданную матрицу?
24. Есть набор костей домино, часть из которых выложена на столе, а остальные находятся у двух игроков. Необходимо определить, существует ли последовательность ходов, при которой второй игрок вынужден пропустить ход из-за отсутствия необходимых костей. Игроки ходят по очереди. Игру начинает первый игрок.
25. В игре на ориентированном графе два игрока поочередно накрывают белыми и черными фишками вершины. Ход является допустимым, если в заданную вершину ведет дуга из вершины, в которой располагается фишка противоположного цвета. Первым ходом белые накрывают любую вершину. Проигрывает тот, кто не может сделать очередного хода. Определить, является ли начальная комбинация выигрышной для белых.
26. Дан набор слов. Составить из них цепочку максимальной длины по количеству слов (или по количеству букв). Цепочка образуется, если первая буква следующего слова совпадает с последней буквой предыдущего слова. Повторно использовать слова нельзя.
27. В игре "Балда" для заданной конфигурации букв и заданного набора слов определить последовательность ходов указанной длины, которая принесет первому игроку наибольшее количество очков.
28. Найти максимальную длину кольца, построенного из слов, содержащих одинаковое количество гласных и согласных букв.
29. Дана схема лабиринта. Необходимо найти все возможные варианты путей выхода из лабиринта без пересечений.

30. По двум конвейерам двигаются молочные бутылки. Для каждой бутылки известно время заполнения и закупоривания. Найти расстановку бутылок, при которой время обработки минимально.

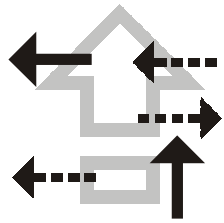
31. Некто, стоя на остановке, в течение часа отмечал интервалы прибытия автобусов. Какое минимальное количество автобусов могло работать на линии в это время и каков интервал их движения? Например, если отмеченные интервалы {3, 10, 10, 10, 10, 10}, то это означает, что работал один автобус. Первый раз он появился через три минуты после начала наблюдений и далее появлялся с интервалом в 10 минут, то есть моменты его появления на остановке следующие — {3, 13, 23, 33, 43, 53}. Если отмеченные интервалы соответствуют следующему набору чисел {2, 10, 5, 15, 3, 12, 11}, то это означает, что работали два автобуса. Первый из них появился через две минуты от начала отсчета и далее ходил с интервалом в 15 минут, то есть моменты его появления на остановке следующие — {2, 17, 32, 47}. Второй автобус впервые появился в 12 минут, а интервал его движения — 23 минуты, значит, он появлялся в следующие моменты: {12, 35, 58}. Следовательно, общее время появления двух автобусов {2, 12, 17, 32, 35, 47, 58}, что как раз соответствует интервалам {2, 10, 5, 15, 3, 12, 11}.

32. Задача о пробирках. Даны три пробирки объемом 100 мл. Две из них имеют шкалу рисок. Уровни рисок указываются пользователем. Определить, возможен ли переход из начального состояния {100 мл, 0 мл, 0 мл} (в первой пробирке 100 мл, а две другие — пусты) путем переливания жидкости из одной пробирки в другую в состояние { _ , _ , 1 мл} (в пробирках с рисками — произвольное количество миллилитров, а в пробирке без рисок — 1 мл).

33. Дан граф. Определите последовательность вершин, через которые необходимо пройти, чтобы нарисовать этот граф. Нельзя отрывать карандаш от бумаги и проходить по одной и той же дуге несколько раз.

34. Задача о рюкзаке. Дано N предметов, у каждого есть вес и цена. Предложить набор предметов максимальной стоимости, помещающихся в рюкзак 50 кг.

Глава 5



Внутренние сортировки

В этой главе рассматривается вопрос, который часто возникает в программировании: перегруппировка (сортировка) элементов в возрастающем или убывающем порядке. Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве. С отсортированными множествами объектов мы встречаемся в телефонных книгах, в словарях, в библиотеках, на складах, почти везде, где нужно искать хранимые объекты.

Алгоритмы сортировки представляют собой интересный частный пример того, как следует подходить к решению проблем программирования вообще. Методы сортировки служат великолепной иллюстрацией идей *анализа алгоритмов*, т. е. идей, позволяющих оценивать рабочие характеристики алгоритмов, а значит, разумно выбирать лучший среди, казалось бы, равноценных методов. На примерах сортировок можно показать, как усложнение алгоритма приводит к значительному выигрышу в эффективности.

Существует много различных алгоритмов сортировок. В [6] рассмотрено 25 таких алгоритмов. Но, как отмечает Д. Кнут [6], "...это пугающее количество методов на самом деле лишь малая толика всех алгоритмов, придуманных до сих пор". Он же подчеркивает, что "к сожалению, неизвестен "наилучший" способ сортировки; существует много наилучших методов в зависимости от того, что сортируется, на какой машине и для какой цели".

Выбор алгоритма зависит от структуры обрабатываемых данных, а в случае сортировки такая зависимость столь глубока, что соответствующие методы были разбиты на два класса:

- сортировки массивов;
- сортировки файлов (последовательностей).

Иногда их называют *внутренней* и *внешней* сортировкой, поскольку массивы хранятся в быстрой (оперативной или внутренней) памяти со случайным доступом, а файлы обычно размещаются в более медленной, но более емкой

внешней памяти, на устройствах, основанных на механических перемещениях (дисках или лентах).

Одним из условий выбора метода внутренней сортировки является экономное использование доступной памяти, перестановки, приводящие элементы в порядок, должны выполняться на том же месте. Руководствуясь этим критерием, можно классифицировать методы по их экономичности, то есть по времени работы. Хорошей мерой эффективности служит число необходимых сравнений и число пересылок (перестановок) элементов. Эти числа являются функциями от числа сортируемых элементов n. Хорошие алгоритмы сортировки требуют порядка ($n \times \log n$) сравнений. В простых и очевидных методах, их называют *прямыми*, требуется порядка n^2 сравнений.

Методы сортировки "на том же месте" можно разбить в соответствии с определяющими принципами на три основные категории [3]:

- сортировка с помощью включения (by insertion);
- сортировка с помощью выделения (by selection);
- сортировка с помощью обменов (by exchange).

Технология *внешней сортировки* в корне отлична от технологии внутренней сортировки, хотя в обоих случаях задача одна и та же: необходимо расположить данные из набора, для внешней сортировки — записи из файла, в определенном порядке. Объясняется это тем, что требуемое время для доступа к файлам на внешних носителях велико и зачастую эти устройства не обеспечивают произвольного (прямого) доступа к данным. Структура данных при реализации алгоритма внешней сортировки должна быть такой, чтобы учитывать эти особенности. Поэтому большинство методов внутренней сортировки (вставка, выбор, обмен) бесполезно для внешней сортировки. Алгоритмы внешней сортировки основаны на слиянии упорядоченных заранее подфайлов. Слияние оперирует только очень простыми структурами данных, пройти которые можно последовательным образом.

Для рассмотрения алгоритмов сортировки введем некоторые обозначения. Если у нас есть элементы

$$a_1, a_2, \ldots, a_n,$$

то сортировка есть перестановка этих элементов в массив

$$a_{k1}, a_{k2}, \ldots, a_{kn},$$

где при некоторой упорядочивающей функции f выполняются отношения

$$f(a_{k1}) \le f(a_{k2}) \le \ldots \le f(a_{kn}).$$

Обычно упорядочивающая функция не вычисляется по какому-либо правилу, а хранится как явная компонента (поле) каждого элемента. Ее значение

называется *ключом* (key) элемента. Поэтому для представления элементов хорошо подходит такая структура данных, как запись. Ее можно описать следующим образом

```
Type item = record
              key: integer;
              {другие компоненты записи}
            end;
```

Под "другими компонентами" подразумеваются данные, по существу относящиеся к сортируемым элементам, а ключ просто идентифицирует каждый элемент. Говоря об алгоритмах сортировки, мы будем обращать внимание лишь на компоненту-ключ.

Метод сортировки называется устойчивым, если в процессе сортировки относительное расположение элементов с равными ключами не изменяется. Устойчивость сортировки желательна, если речь идет об элементах, уже упорядоченных (отсортированных) по некоторым вторичным ключам (т. е. свойствам), не влияющим на основной ключ.

Примеры процедур, реализующих различные алгоритмы внутренних сортировок

Приведем примеры процедур, реализующих некоторые алгоритмы внутренних сортировок. В этих примерах зададимся следующим описанием данных:

```
Const N = 15;                          {Количество элементов в массиве}
      MinKey = 0;                      {Минимальное значение ключа}
      MaxKey = 100;                    {Максимальное значение ключа}
Type  tIndex = 1..N;                   {Индексы в массиве}
      tRange = MinKey..MaxKey;         {Диапазон значений ключа}
      tKey = tRange;                   {Тип ключа сортировки}
      tElem = record                   {Тип элемента записи в массиве}
                Key   : tKey;
                Other : Char
              end;
      PtrElem = ^tElem;                {Указатель на элемент}
      tArray = array [tIndex] of PtrElem;
```

Сортировка элементов массива методом подсчета

Каждый элемент сравнивается со всеми остальными элементами. Если какой-то элемент больше, чем k элементов этого же массива, то после упорядочения он должен занимать $(k + 1)$ место, следовательно, необходимо сравнить попарно все ключи и подсчитать, сколько из них меньше каждого отдельного ключа. После этого необходимо переставить элементы в соответствии с определенными местами.

Если количество элементов массива равно n, то количество сравнений в сортировке методом подсчета равно $(n^2 - n) / 2$, а количество перемещений — $(9n / 4)$.

Реализация сортировки дана в листинге 5.1.

Листинг 5.1. Сортировка методом подсчета

```
procedure Calculation (var A : tArray);
var i,j,k : Integer;
    {Вспомогательный массив для подсчета}
    Count : array [tIndex] of 0..N;
    Elem1,Elem2 : PtrElem;
begin
  {Обнуление вспомогательного массива}
  for i:=1 to N do Count[i]:=0;
  {Формирование вспомогательного массива.}
  {Сравнение каждого элемента с другими и подсчет, больше скольких он}
  for i:=N downto 2 do
    for j:=1 to i-1 do
      if A[i]^.Key>=A[j]^.Key then
        Count[i]:=Count[i]+1
      else
        Count[j]:=Count[j]+1;
  {Расстановка элементов исходного массива по своим местам
   на основании вспомогательного массива}
  i:=1;
  while i<=N do
    begin
      elem1:=A[i]; j:=Count[i]+1;
      while i<>j do
```

```
        begin
          elem2:=A[j]; A[j]:=elem1; elem1:=elem2;
          k:=Count[j]+1; Count[j]:=j-1; j:=k
        end;
      A[i]:=elem1; count[i]:=i-1;   i:=i+1
    end
end;
```

Сортировки вставками

Элементы просматриваются по одному, и каждый новый элемент вставляется в подходящее место среди ранее упорядоченных элементов.

Сортировка простыми вставками

Пусть на i-ом шаге алгоритма последовательность из $(i - 1)$ первых элементов массива упорядочена. Добавление i-го элемента в эту последовательность производится упорядоченно: элемент, который надо вставить, сравнивается с предыдущими до тех пор, пока не найдется место i-го элемента.

Если n — количество элементов массива, то среднее количество сравнений для данной сортировки будет равно $(n^2 + n - 2) / 4$, а среднее количество перемещений — $(n^2 + 9n - 10) / 4$.

Реализация сортировки дана в листинге 5.2.

Листинг 5.2. Сортировка простыми вставками

```
procedure Simple_Insert (var A : tArray);
var i,j : Integer;
    Elem : PtrElem;
begin
  {вставляем i-ый элемент}
  for i:=2 to N do
    begin
      {j - номер элемента, с которым сравниваем i-ый}
      j:=i-1;
      {запоминаем значение вставляемого элемента}
      Elem:=A[i];
      {поиск места, на котором должен находиться i-ый элемент}
      while (j>0) and (A[j]^.Key>Elem^.Key) do
        begin
```

```
          {если элемент A[j] больше, чем Elem,
           то он будет расположен правее, чем сейчас;
           элемент A[j] сдвигается вправо, и сравнение продолжается}
          A[j+1]:=A[j];
          j:=j-1
        end;
      {после выхода из цикла место элемента найдено: (j+1)}
      A[j+1]:=Elem
    end
end;
```

Бинарные вставки

Пусть на *i*-ом шаге последовательность из (i − 1) первых элементов массива упорядочена. Добавление *i*-го элемента в эту последовательность производится упорядоченно: ищется место для нового элемента методом бинарного поиска. Далее элементы, которые должны стоять за *i*-ым элементом сдвигаются вправо, и новый элемент помещается на свое место.

Количество перемещений в сортировке бинарными вставками точно такое же, как и в сортировке простыми вставками, — $(n^2 + 9n - 10) / 4$, количество сравнений — $n \times (\log n - \log e \pm 0.5)$.

Реализация сортировки дана в листинге 5.3.

Листинг 5.3. Сортировка бинарными вставками

```
procedure Bynary_Insert (var A : tArray);
var Place, i, j: integer;
    Elem: PtrElem;

{нерекурсивная процедура поиска места для элемента Elem,
left и right - левая и правая граница уже отсортированного отрезка}
Function FindPlace (left, right: Integer; Elem: PtrElem): integer;
var middle: Integer;
begin
  repeat
    {выбираем средний элемент из отрезка}
    middle:=(Left+Right) div 2;
    {сравниваем со средним элементом элемент, который надо вставить}
    if A[middle]^.Key>Elem^.Key then
      {если средний элемент больше, чем вставляемый,
```

```
        то изменяется правая граница отрезка,
        то есть поиск места для вставляемого элемента
        будет продолжаться в левой части}
        Right:=middle-1
    else
      {если средний элемент меньше, чем вставляемый,
       то изменяется левая граница отрезка,
       то есть поиск места для вставляемого элемента
       будет продолжаться в правой части}
       Left :=middle+1
      {такие действия продолжаются до тех пор,
       пока левая граница не станет больше правой}
  until Left>Right;
  {после выхода из цикла местоположение элемента найдено: left}
  FindPlace:=Left
end;

begin
  {вставляем поочередно все элементы, начиная со второго по N-ый}
  for i:=2 to N do
    begin
      {запоминается элемент, который необходимо вставить}
      elem:=A[i];
      {определяется место, где он должен находиться}
      Place:=FindPlace(1,i-1,Elem);
      {все элементы, которые должны стоять после вставляемого,
       сдвигаются на одну позицию вправо}
      for j:=i-1 downto Place do A[j+1]:=A[j];
      {вставляемый элемент ставится на свое место}
      A[Place]:=Elem
    end
end;
```

Метод Шелла или сортировка с убывающим шагом

Сначала отдельно группируются и сортируются элементы, отстоящие друг от друга на некотором расстоянии $h1$ ($h1 \geq 1$), методом простых вставок, затем элементы перегруппируются (шаг уменьшается и становится равным некоторому значению $h2$) и после этого снова сортируются. Процесс продолжается до тех пор, пока шаг сортировки не станет равным 1.

Эффективность сортировки Шелла зависит от способа определения шага сортировки. Н. Вирт в [3] говорит о том, что неизвестно, какие расстояния дают наилучший эффект. Но для некоторых видов расстояний оценки эффективности определены. Так, например, если расстояния выбираются по следующим формулам $h_{k-1} = 2 \times h_k + 1$, $h_1 = 1$ и $t = \lfloor \log_2 n \rfloor - 1$, то затраты пропорциональны $n^{1.2}$.

Реализация сортировки дана в листинге 5.4.

Листинг 5.4. Сортировка методом Шелла

```
procedure Shell (var A : tArray);
var i, j, step, p, l, T : integer; Elem : PtrElem;
{Вспомогательная функция для вычисления шага}
{В данном примере шаг вычисляется по формуле h = 2^i,}
{где i = T, T-1,  T-2, …, 0, T = [log2 N] }
Function CountStep (j:Integer):Integer;
Var i, st : integer;
begin
  st:=1;
  for i:=1 to j do st:=2*st;
  CountStep:=st-1
end;

begin
  {определяем количество шагов}
  T:=trunc(ln(N)/ln(2));
  for j:=T downto 1 do
    begin
      {вычисляем очередной шаг}
      step:=CountStep(j);
      {применяем сортировку простыми вставками для всех групп}
      {элементов, отстоящих друг от друга на шаг step}
      for p:=1 to step do
        begin
          i:=step+p;
          while (i<=N) do
            begin
              Elem:=A[i]; l:=i-step;
```

```
            while (l>=1) and (Elem^.Key<A[l]^.Key) do
              begin
                A[l+step]:=A[l]; l:=l-step
              end;
            A[l+step]:=Elem; i:=i+step
          end
      end
  end
end;
```

Сортировки обменами

Сортировка методом "пузырька"

На первом проходе соседние элементы сравниваются, и если элементы не упорядочены, они меняются местами, в результате чего самый большой элемент переставляется на последнее место. За второй проход самый большой из оставшихся (сравнение идет между $(N-1)$ элементами) перемещается на предпоследнее место и так до тех пор, пока массив не окажется отсортированным.

В данном примере рассматривается усовершенствованный метод: на каждом проходе запоминается место последнего обмена между элементами. На следующем проходе сравнение выполняется только до этого места.

Количество сравнений в строго обменной пузырьковой сортировке равно $(n^2 - n) / 2$, а среднее количество перемещений — $3 \times (n^2 - n) / 2$. Анализ улучшенного метода довольно сложен. Кнут считает, что для улучшенной сортировки пузырьком среднее число проходов пропорционально $n - k_1\sqrt{n}$, среднее число сравнений пропорционально $\left(n^2 - n(k_2 + \ln n)\right)/2$, а среднее число перемещений остается таким же, как и для строго обменного пузырька.

Реализация сортировки дана в листинге 5.5.

Листинг 5.5. Сортировка методом "пузырька"

```
procedure Bubble (var A : tArray);
var i, Bound, t : integer; Elem: PtrElem;
begin
  {выбираем границу справа для сравниваемых элементов}
  Bound:=N;
```

```
  repeat
    {предполагаем, что на данном проходе алгоритма обменов не будет,
     т. е. все элементы упорядочены}
    t:=0;
    {сравниваем все соседние элементы:
     1-ый и 2-ой, 2-ой и 3-ий,…, Bound-1 и Bound}
    for i:=1 to Bound-1 do
      {если элемент слева больше, чем элемент справа, то}
      if A[i]^.Key>A[i+1]^.Key then
        begin
          {меняем их местами}
          Elem:=A[i+1]; A[i+1]:=A[i]; A[i]:=Elem;
          {запоминаем место последнего обмена}
          t:=i
        end;
    {запоминаем место последнего обмена как правую границу}
    Bound:=t
  until t=0
end;
```

Быстрая сортировка или метод Хоара

Сортируемый массив просматривается сразу с двух сторон (слева и справа). Если слева находится элемент больший, чем элемент справа, то эти элементы меняются местами. Так продолжается до тех пор, пока указатели на левый и правый элементы не пересекутся. В результате такого прохода алгоритма получим, что весь массив разбит на две части. В левой части находятся элементы с меньшими значениями ключей, а в правой — с большими. Теперь осталось отсортировать тем же самым методом каждую часть отдельно.

Н. Вирт в [3] говорит о том, что в лучшем случае в сортировке Хоара потребуется порядка $n \times \log n$ операций, а в худшем — n^2.

Реализация сортировки дана в листинге 5.6.

Листинг 5.6. Быстрая сортировка

```
procedure Quick_Rec (var A : tArray);
  procedure QuickSort (left,right : integer);
  var i, j: integer;
      Elem: PtrElem;
```

```
  begin
  {если есть что сортировать}
  if left<right then
    begin
      {запоминаем указатели на левый и правый сравниваемые элементы}
      i:=left; j:=right;
      {до тех пор, пока указатели не пересекутся}
      while (i<j) do
        begin
          {ищем элемент справа, который был бы меньше, чем элемент слева}
          while (i<j) and (A[j]^.Key>=A[i]^.Key) do j:=j-1;
          {меняем элементы местами}
          Elem:=A[i]; A[i]:=A[j]; A[j]:=elem;
          {ищем элемент слева, который был бы больше, чем элемент справа}
          while (i<j) and (A[j]^.Key>=A[i]^.Key) do i:=i+1;
          {меняем элементы местами}
          Elem:=A[i]; A[i]:=A[j]; A[j]:=elem
        end;
      {сортируем левую часть массива}   QuickSort(left,i-1);
      {сортируем правую часть массива}  QuickSort(j+1,Right)
    end
  end;
begin
  QuickSort(1,N)
end;
```

Сортировки выбором

Простой выбор

Сначала из N элементов выбирается максимальный элемент. Далее N-ый и найденный максимальный элементы меняются местами. Затем среди оставшихся $(N-1)$ элементов от 1 до $(N-1)$ выбирается максимальный и меняется местами с элементом, стоящим на $(N-1)$ месте и т. д. Так продолжается до тех пор, пока весь массив не будет отсортирован. Последний раз максимум выбирается из двух элементов, стоящих на первом и на втором месте соответственно, и наибольший из них ставится на второе место.

Количество сравнений для данного метода всегда постоянно и равно $(n^2 - n)/2$, а среднее количество перемещений определяется следующей формулой: $n \times (\ln n + g)$, где g — постоянная Эйлера ($g = 0.577216$).

Реализация сортировки дана в листинге 5.7.

Листинг 5.7. Сортировка простым выбором

```
procedure Simple_Choice (var A : tArray);
var i, j, k : integer;
    Elem  : PtrElem;
begin
  for j:=N downto 2 do
    begin
      {ищем максимум среди элементов от j до 1}
      {местоположение максимального элемента запоминается в переменной i}
      i:=j;
      for k:=j-1 downto 1 do
        if A[i]^.Key<A[k]^.Key then i:=k;
      {j-ый и i-ый элементы меняются местами}
      Elem:=A[j];A[j]:=A[i]; A[i]:=Elem
    end
end;
```

Помимо приведенных выше примеров внутренних сортировок существуют и другие, например:

- распределяющий подсчет;
- двухпутевые вставки;
- вставки в список с вычислением адреса;
- шейкерная сортировка;
- обменная поразрядная сортировка;
- пирамидальная сортировка;
- параллельная сортировка Бэтчера;
- простое слияние (метод Неймана);
- естественное слияние.

Эффективность этих методов описана в работах Кнута [6] и Вирта [3].

Анализ алгоритмов сортировок массивов

На рис. 5.1 и 5.2 приведены графики сортировок для простых и улучшенных методов. В качестве критерия эффективности здесь используется общее количество операций — и количество сравнений, и количество пересылок.

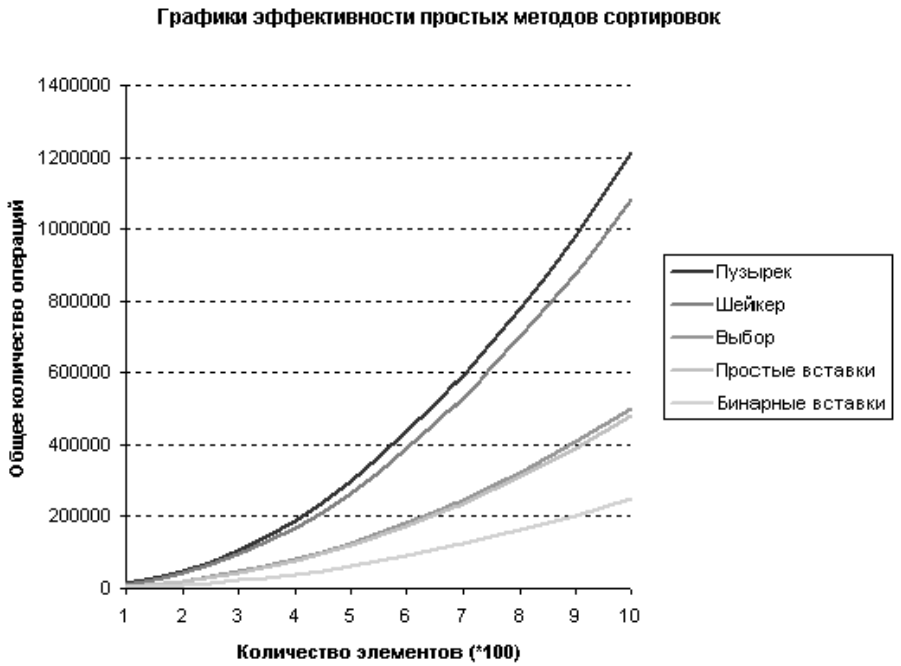


Рис. 5.1. Графики эффективности простых методов сортировок

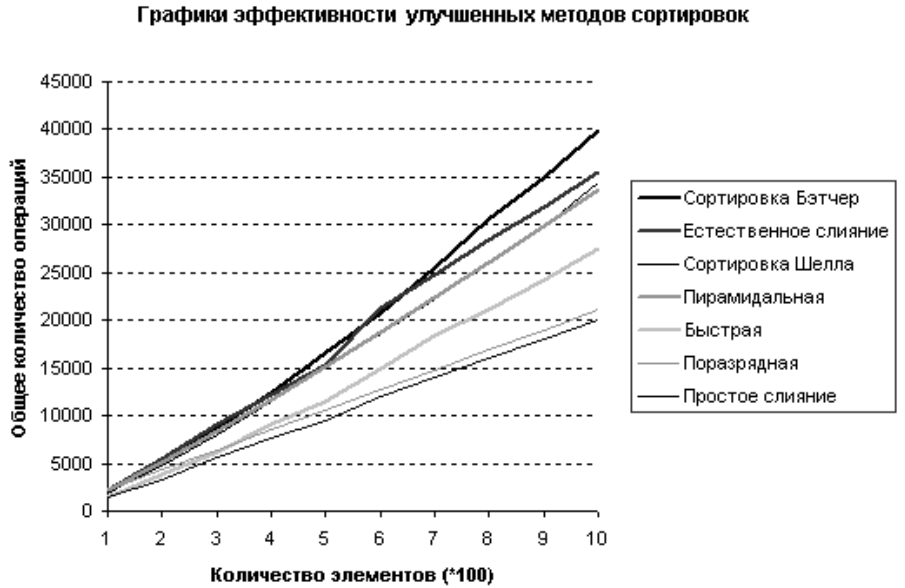


Рис. 5.2. Графики эффективности улучшенных методов сортировок

Задания для самостоятельной работы

1. Задан массив записей, поле *key* которого — целые числа. Написать программу, которая наглядно демонстрирует сортировку массива по ключу *key*:

   а) методом простого слияния;

   б) методом естественного слияния.

   Количество элементов массива таково, что все элементы отображаются на экране. В данных сортировках используется дополнительный массив.

2. Исследовать сортировки простым и естественным слиянием, построить графики зависимости количества сравнений и количества перестановок от количества элементов массива (n = 1 .. 100) для этих двух сортировок. Использовать функцию, которая по заданному массиву и количеству элементов в этом массиве подсчитывает, сколько потребуется сравнений (перестановок).

3. Задан массив записей, поле *key* которого — целые числа. Написать программу, которая наглядно демонстрирует пирамидальную сортировку по ключу *key*:

   а) массив изображен в виде последовательности элементов;

   б) при построении пирамиды на экране массив отображается не только в виде последовательности, но и в виде построенной пирамиды.

4. Написать программу, которая наглядно иллюстрирует работу обменной поразрядной сортировки для следующих типов данных:

   а) целого;

   б) символьного;

   в) строкового (короткого).

5. Написать программу, иллюстрирующую работу сортировки Хоара:

   а) реализовать рекурсивным методом;

   б) реализовать нерекурсивным методом;

   в) реализовать любым из методов, но учитывать, что для сортировки массива маленького размера лучше применять какой-нибудь другой метод сортировки (например, простыми вставками, метод "пузырька", ...).

6. Проиллюстрировать работу сортировки Бэтчера.

Указание

Обратите внимание на то, что эта сортировка параллельная, т. е. сравнение или перестановка элементов должны выполняться одновременно (параллельно).

7. Написать программу, которая наглядно иллюстрирует работу следующих методов сортировки:

   а) "пузырек";

   б) шейкерная.

   Провести сравнение сортировок методом "пузырька" и шейкерной по количеству сравнений, по количеству обменов. Для этого построить графики зависимостей данных величин от количества элементов массива.

8. Дан текстовый файл, состоящий из слов, разделенных пробелами. Отсортировать слова в этом файле методом вставки в список с вычислением адреса.

9. Написать программу, которая иллюстрирует работу метода Шелла с одной из формул вычисления шага сортировки:

   а) $h[k-1] = 3h[k] + 1$, $h[t] = 1$, $t = [\log_3 n] - 1$;

   б) $h[k-1] = 2h[k] + 1$, $h[t] = 1$, $t = [\log_2 n] - 1$;

   в) $2^k - 1$;

   г) $2^k + 1$;

   д) $(2^k - (-1)^k)/3$;

   е) $(3^k - 1) / 2$;

   ж) числа Фибоначчи.

10. Написать программу, которая исследует зависимость количества сравнений (количества перестановок) в методе Шелла от выбора разных формул для вычисления шага. Изобразить соответствующие графики функций.

11. Реализовать сортировку массива целых чисел методом двухпутевых вставок при использовании следующих дополнительных структур данных:

    а) массива;

    б) двунаправленного списка.

    Программа должна наглядно иллюстрировать работу данного алгоритма.

12. Написать программу, которая наглядно иллюстрирует работу сортировок простыми и бинарными вставками.

13. Исследовать зависимость количества сравнений в сортировках простыми и бинарными вставками от количества элементов в этих массивах. При этом отобразить необходимое количество перестановок.

14. Дан массив записей, поле *key* которого — целые числа. Написать программу, которая иллюстрирует работу сортировки по ключу *key* методом подсчета. При этом должны отображаться не только сравниваемые элементы, но и вспомогательный массив, в котором хранятся результаты сравнений. Перестановки элементов осуществляются в исходном массиве.

15. Написать программу, которая иллюстрирует сортировку массива распределяющим подсчетом. Элементом массива является запись следующего типа:

```
Record
  Ch : char;
  Key : integer
End
```

Указание

Ключом сортировки является поле целого типа.

16. Для каждого из пунктов обмена валюты известны курс покупки и курс продажи доллара. Вывести список пунктов в порядке:
    - выгоды продажи;
    - выгоды покупки;
    - убывания среднеквадратического отклонения от курса доллара на ММВБ.
17. В магазине строительных материалов в продаже имеются стеновые панели, которые характеризуются следующими величинами:
    - ширина;
    - длина;
    - количество штук;
    - цена за 1 м^2.

    Вывести в порядке возрастания цены сведения о тех стеновых панелях, общая площадь которых не меньше заданной.
18. Дан список морфем, в котором указана сама морфема и ее тип (приставка, корень, суффикс или окончание). В списке могут встречаться повторяющиеся морфемы. Написать программу, результатом работы которой должен быть список морфем, разбитый на группы для каждого типа. Внутри этих групп морфемы должны быть упорядочены в лексикографическом порядке. В группе корней одна и та же морфема может встречаться несколько раз, а в других группах морфемы должны быть уникальными.
19. Есть некий измерительный прибор, работа которого зависит от входных параметров α и x, а результат определяется следующей формулой $y = \alpha \cdot \sin(\alpha x) \cdot \cos^2(x/\alpha)$. Проводится серия опытов для значений $x_1, x_2, ..., x_n$, $\alpha = \text{const}$. Вывести результат в виде таблицы, упорядоченной по убыванию значений показаний прибора, полученных в ходе опытов.

20. В массиве произвольно хранятся сведения о жильцах (ФИО, адрес, телефон). Упорядочить эти данные по полю ФИО в алфавитном порядке. Реализовать следующие возможности:
    - добавление сведений о новом жильце;
    - удаление сведений о жильце;
    - редактирование сведений о жильце.

    Эти действия не должны нарушать упорядоченность массива.

    Указание

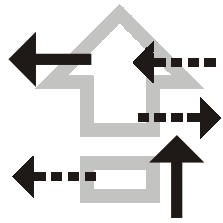
    Использовать алгоритмы, изложенные в методах сортировок простыми или бинарными вставками.

21. Информация агентства по продаже недвижимости содержит следующие сведения о квартирах: район, в котором находится квартира, этаж, количество комнат, общая площадь, цена за 1 м^2. Клиент, обращаясь в агентство, имеет возможность указать вес для каждого из критериев (важный критерий имеет большой вес, незначительный — маленький), а агентство, в свою очередь, предлагает ему список квартир, упорядоченный по невозрастанию суммы весов.
22. Дана целочисленная квадратная матрица размером n. Упорядочить значения так, чтобы $a_{11} \le a_{12}, \le \le a_{1n} \le a_{21} \le a_{22} \le\le a_{2n} \le\le a_{n1} \le a_{n2}, \le\le a_{nn}$.
23. В чемпионате России по футболу принимают участие 16 команд. Для каждой команды известен список игроков, каждый игрок из команды имеет свой рейтинг. Вывести список команд в порядке убывания вероятности победы в чемпионате. Вероятность победы равна рейтингу команды — сумме рейтингов 11 лучших игроков.
24. Дано

```
const n = 20;
type  город = (A, B, C, D, E, F, G, H);
      города = set of город;
      маршрут = record
        гор : города;
        цена : integer;
      end;
      тур_фирма = array [1..n] of маршрут;
var   ТФ : тур_фирма;  Ц : integer;
```

    Вывести список маршрутов, не превышающих по цене заданное значение *Ц*, в порядке убывания количества городов в маршруте.

Глава 6



Внешние сортировки

Внешние сортировки — это сортировки, применяемые к данным, которые хранятся во внешней памяти. В таком случае мы говорим, что данные представляют собой последовательный файл, состоящий из записей или элементов. В каждый момент непосредственно доступен для считывания только один элемент. Как и в случае внутренних сортировок существует большое количество алгоритмов, позволяющих сортировать данные, которые хранятся в файлах. Большинство их основано на слиянии частично упорядоченных вспомогательных файлов.

Стоит заметить, что если в некоторой реальной задаче появляется необходимость отсортировать данные, хранящиеся во внешней памяти, и есть возможность загрузить их в оперативную память и упорядочить эти данные там, то рекомендуется использовать ее, поскольку время обращения к оперативной памяти гораздо меньше, чем время обращения к внешней памяти. Такая программа будет работать быстрее, чем при реализации какого-либо из алгоритмов внешних сортировок. В качестве вспомогательных структур при загрузке данных из внешней памяти могут быть выбраны массивы, множества, списки, деревья. В книге Дж. Бентли "Жемчужины программирования" [1] рассмотрены несколько замечательных примеров, когда специальное представление данных во внутренней памяти позволило быстро и легко упорядочить данные.

Наиболее известные алгоритмы внешних сортировок представлены на рис. 6.1.

Наиболее важным методом сортировки последовательностей является метод сортировки с помощью *слияния.* Прежде чем описывать принципы, заложенные в сортировке слиянием, введем определение упорядоченного отрезка.

Упорядоченным отрезком или *серией* называется последовательность элементов, для которых выполняется условие упорядоченности. Количество элементов данной последовательности называется *длиной упорядоченного отрезка (серии).* Отрезок, состоящий из одного элемента, упорядочен всегда.

Н. Вирт [3] дает следующее определение слияния: "Слияние означает объединение двух (или более) последовательностей в одну-единственную

упорядоченную последовательность с помощью повторяющегося выбора из доступных в данный момент элементов". Другими словами, сначала серии распределяются на два или несколько вспомогательных файлов. Распределение серий идет поочередно, то есть первая серия записывается в первый вспомогательный файл, вторая — во второй и так далее. После того как произошла запись в последний файл, опять начинается запись серии в первый вспомогательный файл. После распределения всех серий, они объединяются в более длинные упорядоченные отрезки: из каждого вспомогательного файла берется по одной серии, и они сливаются. Если в каком-то файле серия заканчивается, то следующая пока не рассматривается. Сформированный более длинный упорядоченный отрезок записывается либо в исходный файл, либо в какой-то из вспомогательных файлов, это зависит от вида сортировки. После того как все серии из всех вспомогательных файлов объединены в новые серии, опять начинается их распределение. Так продолжается до тех пор, пока все данные не будут упорядочены.

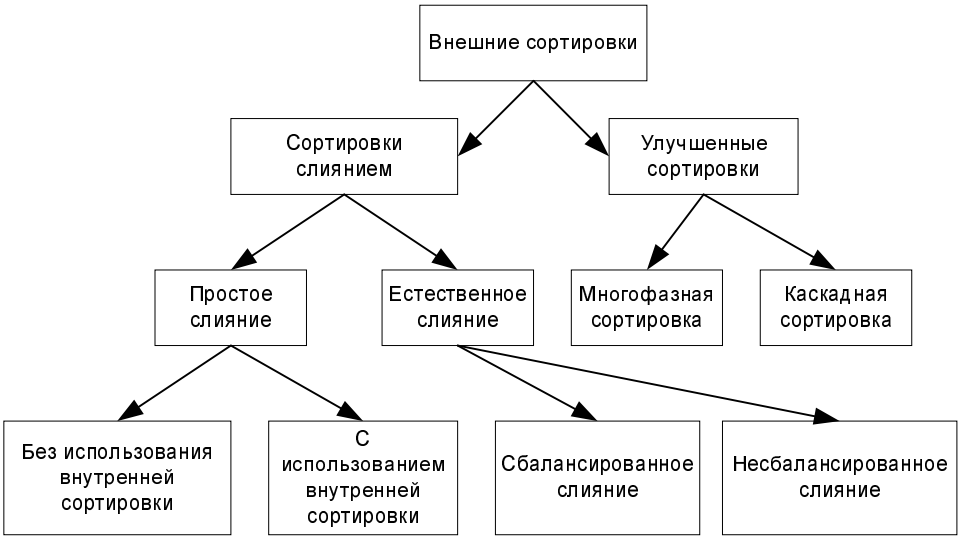


Рис. 6.1. Внешние сортировки

Можно выделить следующие две характеристики сортировки слиянием.

- Количество вспомогательных файлов, на которые идет распределение серий. Если данные распределяются на два вспомогательных файла, то сортировка называется *двухпутевым слиянием*, если на N ($N > 2$) вспомогательных файлов, то — *многопутевым слиянием*.
- Количество фаз в реализации сортировки. *Фазой* называются действия по однократной обработке всего множества. Наименьший же процесс, повторение которого составляет процесс сортировки, называется *проходом*

или *этапом*. В *двухфазной* сортировке отдельно реализуются две фазы: распределение и слияние. После того как произошло распределение данных по вспомогательным файлам, они объединяются и записываются в исходный файл. В *однофазной* сортировке обе эти фазы объединены в одну. Во время слияния серий данные не только объединяются, но и сразу же распределяются по другим вспомогательным файлам, которые становятся исходными для следующего прохода. Однофазная сортировка более эффективна, чем двухфазная, поскольку фаза распределения фактически не относится к сортировке, так как во время этой фазы элементы не переставляются и не упорядочиваются, а количество операций переписи данных такое же, как и на фазе слияния. Однако количество вспомогательных файлов, которые требуются для однофазной сортировки, больше, чем то, которое требуется для двухфазной сортировки.

Простое слияние

Отличие между простым слиянием и естественным слиянием, которое рассмотрено в следующем разделе, заключается в определении серии. В *простом слиянии* длина серии фиксирована. На первом шаге упорядоченные отрезки имеют длину, равную единице, то есть состоят из одного элемента. Потом они объединятся, и если слияние двухпутевое, то вновь сформированные отрезки будут состоять из двух элементов, если слияние многопутевое (количество вспомогательных файлов N), то — из N элементов. Предположим, что исходный файл содержит следующие данные:

```
F0 : 5 7 3 2 8 4 1.
```

В табл. 6.1 показано перемещение данных во время сортировки двухпутевым двухфазным слиянием, а в табл. 6.2 — многопутевым двухфазным слиянием.

Таблица 6.1. *Пример двухпутевого двухфазного слияния*

	Двухфазная сортировка		**Однофазная сортировка**
	Распределение	**Слияние**	
1 проход	F1 : 5 3 8 1	F0 : 5 7 2 3 4 8 1	F1 : 5 3 8 1
	F2 : 7 2 4		F2 : 7 2 4
2 проход	F1 : 5 7 4 8	F0 : 2 3 5 7 1 4 8	F3 : 5 7 4 8
	F2 : 2 3 1		F4 : 2 3 1
3 проход	F1 : 2 3 5 7	F0 : 1 2 3 4 5 7 8	F1 : 2 3 5 7
	F2 : 1 4 8		F2 : 1 4 8
4 проход			F3 : 1 2 3 4 5 7 8
			F4 :

Таблица 6.2. *Пример многопутевого двухфазного слияния (N = 3)*

	Двухфазная сортировка		Однофазная сортировка
	Распределение	**Слияние**	
1 проход	F1 : 5 2 1	F0 : 3 5 7 2 4 8 1	F1 : 5 2 1
	F2 : 7 8		F2 : 7 8
	F3 : 3 4		F3 : 3 4
2 проход	F1 : 3 5 7	F0 : 1 2 3 4 5 7 8	F4 : 1 2 3 4 5 7 8
	F2 : 2 4 8		F5 :
	F3 : 1		F6 :

Длина серии в алгоритме простого слияния зависит от количества вспомогательных файлов, на которое идет распределение серий. Если k — это номер прохода, а N — количество вспомогательных файлов, то длина серии определяется формулой N^k. Обратим внимание на то, что на первых проходах длина серии растет медленно, например, в двухпутевой сортировке, чтобы получить серию, состоящую из 100 элементов, требуется семь проходов. Это не эффективно, и поэтому для улучшения алгоритма слияния можно использовать внутреннюю сортировку в качестве вспомогательной. В этом случае на первом проходе часть данных (фиксированное количество) переписывается в оперативную память, например, в массив, и там упорядочивается. Таким образом сразу формируется серия большой длины, и для дальнейшего слияния потребуется меньшее количество проходов. Этот способ упорядочения данных называется простым слиянием с использованием внутренней сортировки.

Признаками конца сортировки простым слиянием являются следующие условия:

- длина серии не меньше количества элементов в файле (определяется после фазы слияния);
- количество серий — одна (определяется на фазе слияния);
- второй по счету вспомогательный файл при однофазной сортировке после распределения серий остался пустым.

Естественное слияние

В случае простого слияния никак не используется частичная упорядоченность элементов в исходном файле: размер сливаемых на k-ом проходе серий меньше или равен N^k и не зависит от существования более длинных

упорядоченных подпоследовательностей. Фактически любые две упорядоченные подпоследовательности длиной m и n можно сливать в одну, состоящую из $n + m$ элементов.

Сортировка, при которой всегда сливаются две самые длинные из возможных серий, называется *естественным слиянием*. В сортировке естественным слиянием объединяются серии максимальной, а не заранее фиксированной длины. Признаком конца серии в этом случае является результат сравнения двух соседних элементов, если они не упорядочены, то серия закончилась. В каждом проходе число серий уменьшается в N раз (если количество серий во вспомогательных файлах одинаково), а общее число пересылок не более, чем $n \times [\log n]$. Ожидаемое число сравнений значительно больше, поскольку кроме сравнений, необходимых для слияния, требуются дополнительные сравнения для определения конца серии. Приведем пример естественного слияния. Предположим, исходный файл содержит следующие данные:

```
F 0 : 2 3 17 7 8 9 1 4 6 9 2 3 1 18.
```

Тогда двухпутевое двухфазное естественное слияние будет проходить следующим образом (табл. 6.3). Символ ' обозначает признак конца серии.

Таблица 6.3. *Пример двухпутевого двухфазного естественного слияния*

	Двухфазная сортировка	
	Распределение	**Слияние**
1 проход	F1 : 2 3 17 ' 1 4 6 9 ' 1 18	F0 : 2 3 7 8 9 17 1 2 3 4 6 9 1 18
	F2 : 7 8 9 ' 2 3	
2 проход	F1 : 2 3 7 8 9 17 ' 1 18	F0 : 1 2 2 3 3 4 6 7 8 9 9 17 1 18
	F2 : 1 2 3 4 6 9 '	
3 проход	F1 : 1 2 2 3 3 4 6 7 8 9 9 17	F0 : 1 1 2 2 3 3 4 6 7 8 9 9 17 18
	F2 : 1 18	

Для упорядочения данных в сортировке простым слиянием при тех же самых условиях потребовалось бы четыре прохода, поскольку длина серии на четвертом проходе составляла бы 16 элементов, а в исходном файле содержится 14 элементов.

Так же как и простое слияние, сортировка естественным слиянием может быть двухпутевой или многопутевой, двухфазной или однофазной.

Обратим внимание на то, что сортировка проходит эффективно, если количество серий, распределенных на вспомогательные файлы, приблизительно одинаковое. Поскольку признаком конца серии в естественном слиянии является результат сравнения двух соседних элементов, две или несколько

серий, распределенных друг за другом во вспомогательный файл, могут объединиться в одну. Рассмотрим пример. Пусть исходный файл содержит данные:

```
F0: 1 2 9 3 39 11 4 18 13 5 16 24 15 4 25 17 317
```

После фазы распределения вспомогательные файлы будут иметь следующий вид:

```
F1: 1 2 9 11 13 15 17
F2: 3 39 ' 4 18 ' 5 16 24 ' 4 25 317
```

В первом вспомогательном файле оказалась одна упорядоченная серия, хотя записывалось туда пять серий. То есть произошло слияние нескольких серий в одну из-за того, что первый элемент $(i + 2)$-ой серии больше, чем последний элемент i-ой серии. Вследствие отмеченного явления, числа серий во вспомогательных файлах могут сильно различаться, и дальнейшее слияние будет не эффективным. Естественное слияние, у которого после фазы распределения количество серий во вспомогательных файлах отличается друг от друга не более чем на единицу, называется *сбалансированным*, в противном случае речь идет о *несбалансированном* слиянии. Чтобы в сбалансированном слиянии серии распределялись корректно, необходимо во время записи очередной серии в файл выполнять следующую проверку: если серия является продолжением предыдущей, то записать в этот файл не одну, а две серии. Для сбалансированного слияния в приведенном выше примере данные после распределения будут иметь следующий вид:

```
F1: 1 2 9 11 ' 4 18  ' 5 16 24  ' 17
F2: 3 39     ' 13 15 ' 4 25 317
```

Признаками конца сортировки естественным слиянием являются следующие условия:

- количество серий — одна (определяется на фазе слияния);
- второй по счету вспомогательный файл для однофазной сортировки после распределения серий остался пустым.

Улучшенные методы сортировки

Улучшенные методы сортировки построены на базе многопутевого естественного сбалансированного слияния. К ним можно отнести многофазную и каскадную сортировки.

Многофазная сортировка

Многофазная сортировка (Polyphase Sort) была изобретена Р. Гилстэдом. В основе улучшения лежит отказ от жесткого понятия прохода: сначала происходит начальное распределение данных, а затем используется $(N - 1)$-путевое

слияние, то есть все время серии берутся из $N-1$ файлов и сливаются в один пустой. Чтобы $(N-1)$-путевое слияние стало возможным, необходимо вначале распределить данные особым образом: распределение происходит на $N-1$ вспомогательный файл, при этом количество серий в каждом из файлов зависит от количества вспомогательных файлов и количества серий в исходном файле. Поскольку последний параметр заранее не известен, то распределение серий происходит по уровням. Если в исходном файле одна серия, то на нулевом уровне все данные должны быть записаны в один файл. На всех остальных уровнях количество серий, которое должно быть распределено на каждый из вспомогательных файлов, определяется формулами:

$$a_1^1 = 1 \quad a_i^0 = 0,\ i = 2, 3, \ldots, N$$
$$a_i^L = a_{i+1}^{L-1} + a_1^{L-1},\ i = 1, 2, \ldots, N-1,$$

где L — номер уровня, а N — количество вспомогательных файлов.

Нетрудно заметить, что количество серий в исходном файле может отличаться от такого идеального распределения. В этом случае считается, что в распоряжении имеются фиктивные (пустые) серии, причем их столько, что сумма пустых и реальных серий равна идеальной сумме. Заметим, что слияние пустых серий из всех $N-1$ источников означает, что никакого реального слияния не происходит, вместо него в выходную последовательность записывается результирующая пустая серия. Отсюда можно заключить, что пустые серии нужно как можно более равномерно распределять по $N-1$ последовательностям. Алгоритм "горизонтального распределения пустых серий", изложенный в [6], позволяет эффективно распределить серии.

1. Подсчитать число серий, которое должно быть распределено в каждый из вспомогательных файлов на данном уровне (a_i^L $i = 1, \ldots, N-1$).
2. Подсчитать количество серий, которое надо дописать в каждый из вспомогательных файлов на данном уровне, чтобы получить число серий, равное a_i^L: $d_i^L = a_i^L - a_i^{L-1}$. После того как это число определено, считаем, что в каждый из вспомогательных файлов записано d_i^L пустых серий.
3. Пока не кончился исходный файл и пока не исчерпаны все пустые серии на данном уровне, заменяем пустую серию на реальную; при этом записываем ее в тот файл, в котором количество пустых серий максимально.
4. Если исходный файл не кончился, осуществляется переход на следующий уровень: $L = L + 1$ и переход к пункту 1.

Когда все данные распределены, происходит слияние данных: сначала сливаются данные из вспомогательных файлов с номерами 1, 2, ..., $N-1$ в N-ый вспомогательный файл, при этом файл с номером $N-1$ станет

пустым (слияние на L-ом уровне). Далее номер уровня уменьшается и происходит слияние файлов с номерами 1, 2, ..., $N-1$ в $(N-1)$-ый файл, т. е. слияние на $L-1$ уровне. Так продолжается до тех пор, пока номер уровня не станет равным 0. Заметим, что во время слияния на очередном уровне, если не кончились пустые серии, то в первую очередь объединяются они, поскольку их объединение требует меньше затрат. В качестве примера приведем табл. 6.4, которая иллюстрирует распределение и слияние 129 серий в многофазной сортировке при использовании шести вспомогательных файлов.

Таблица 6.4. *Пример многофазной сортировки*

Распределение серий									
L	**a_1**	**a_2**	**a_3**	**a_4**	**a_5**	**a_6**			
1	1	1	1	1	1	0			
2	2	2	2	2	1	0			
3	4	4	4	3	2	0			
4	8	8	7	6	4	0			
5	16	15	14	12	8	0			
6	31	30	28	24	16	0			
Слияние серий									
L	**a_1**	**a_2**	**a_3**	**a_4**	**a_5**	**a_6**	**Откуда**	**Куда**	**Сколько**
6	31	30	28	24	16	0	1, 2, 3, 4, 5	6	16
5	15	14	12	8	0	16	6, 1, 2, 3, 4	5	8
4	7	6	4	0	8	8	5, 6, 1, 2, 3	4	4
3	3	2	0	4	4	4	4, 5, 6, 1, 2	3	2
2	1	0	2	2	2	2	3, 4, 5, 6, 1	1	1
1	0	1	1	1	1	1	2, 3, 4, 5, 6	1	1
0	1	0	0	0	0	0			

Обратим внимание на то, что в сортировке естественным слиянием две последовательно поступающие в один файл серии могли превратиться в одну-единственную. В многофазной сортировке это может привести к "нарушению" ожидаемого числа серий, поэтому при реализации алгоритма многофазной сортировки обязательно надо отслеживать эффект такого случайного слияния.

Каскадная сортировка

Каскадная сортировка похожа на многофазную сортировку. Отличие заключается в самом процессе слияния: сначала проводится $(N-1)$ — путевое слияние в N-ый файл до тех пор, пока файл с номером $N-1$ не опустеет; затем (N-ый файл уже не затрагивается) проводится $(N-2)$ — путевое слияние в $(N-1)$-ый файл; затем $(N-3)$ — путевое в $(N-2)$-ый файл; ...; потом двухпутевое слияние в третий файл, а в конце копирование из файла с номером 2 в первый файл. Следующий проход работает по аналогичной схеме, только в обратном направлении. Поскольку слияние в каскадной сортировке отличается от слияния в многофазной сортировке, то и начальное распределение данных осуществляется по-другому — количество серий в каждом из вспомогательных файлов должно быть другим. Оно определяется в соответствии с формулами:

$$a_1^0 = 1 \quad a_i^0 = 0,\ i = 2, 3, \ldots, N$$

$$a_i^L = a_{i+1}^L + a_{N-i}^{L-1},\quad i = N-1, N-2, \ldots, 1$$

или

$$a_1^0 = 1 \quad a_i^0 = 0, \quad i = 2, 3, \ldots, N$$

$$a_1^L = \sum_{i=1}^{N-1} a_i^{L-1}; \quad a_{i+1}^L = a_i^L - a_{N-i}^{L-1},\ i = 1, 2, \ldots, N-2.$$

Н. Вирт в [3] пишет: "Хотя такая схема выглядит хуже многофазного слияния, поскольку некоторые последовательности "простаивают", и есть просто операции копирования, тем не менее, она, что удивительно, оказывается лучше многофазной сортировки при очень больших файлах и шести или более последовательностях". Заметим, что при хорошей реализации алгоритма каскадной сортировки, от копирования можно избавиться, используя карты индексов. Более подробное объяснение алгоритмов, а также их реализацию можно увидеть в книгах Н. Вирта [3] и Д. Кнута [6].

В качестве примера приведем табл. 6.5, которая иллюстрирует распределение и слияние 190 серий в каскадной сортировке при использовании шести вспомогательных файлов.

Таблица 6.5. *Пример каскадной сортировки*

Распределение серий									
L	**a_1**	**a_2**	**a_3**	**a_4**	**a_5**	**a_6**			
1	1	1	1	1	1	0			
2	5	4	3	2	1	0			
3	15	14	12	9	5	0			
4	55	50	41	29	15	0			

Таблица 6.5 *(окончание)*

Слияние серий									
L	**a_1**	**a_2**	**a_3**	**a_4**	**a_5**	**a_6**	**Откуда**	**Куда**	**Сколько**
4	55	50	41	29	15	0	1, 2, 3, 4, 5	6	15
	40	35	26	14	0	15	1, 2, 3, 4	5	14
	26	21	12	0	14		1, 2, 3	4	12
	14	9	0	12			1, 2	3	9
	5	0	9				1	2	5
	0	5							
3	0	5	9	12	14	15	6, 5, 4, 3, 2	1	5
	5	0	4	7	9	10	6, 5, 4, 3	2	4
		4	0	3	5	6	6, 5, 4	3	3
			3	0	2	3	6, 5	4	2
				2	0	1	6	5	1
					1	0			
2	5	4	3	2	1	0	1, 2, 3, 4, 5	6	1
	4	3	2	1	0	1	1, 2, 3, 4	5	1
	3	2	1	0	1		1, 2, 3	4	1
	2	1	0	1			1, 2	3	1
	1	0	1				1	2	1
	0	1							
1	0	1	1	1	1	1	6, 5, 4, 3, 2	1	1
0	1	0	0	0	0	0			

Пример программы внешней сортировки

В качестве примера рассмотрим следующую задачу.

Сведения о товарах, хранящихся на складах, включают в себя наименование товара и его количество. В течение месяца на складе ведется учет поступивших и отпущенных товаров (если количество положительное, то речь идет о поступившем товаре, иначе — об отпущенном). Эти сведения заносятся в файл. Изначально в файле хранится информация о товарах, оставшихся на складе с предыдущего месяца.

Написать программу, которая формирует список товаров в алфавитном порядке с указанием количества, имеющегося на складе в данный момент (в конце очередного месяца).

Одним из способов решения поставленной задачи является следующий:

- сначала отсортировать все данные по наименованию товара. Таким образом, сведения об одном и том же товаре окажутся рядом;
- затем сжать полученные данные, т. е. для одного товара посчитать оставшееся количество и записать в новый файл в виде одной записи.

Для реализации этой задачи потребуется ряд вспомогательных процедур:

- процедуры ввода/вывода данных для тестирования основной программы;
- процедура сортировки данных;
- процедура сжатия отсортированных данных.

Для наглядности составим программу (листинг 6.1) со следующей структурой: процедуры ввода/вывода поместим в один модуль, а процедуры, отвечающие за решение основной задачи, — в другой. Поскольку и те и другие работают с файлами одинакового типа, то вынесем данные по описанию типизированного файла в отдельный модуль. Все вспомогательные процедуры будут вызываться из основной программы.

Структура программы:

- модуль `Common` содержит сведения о товарах и о типе файла, в котором товары хранятся;
- модуль `InpOut` содержит процедуры, отвечающие за ввод/вывод данных (мы не будем подробно описывать реализацию этих процедур, поскольку они являются вспомогательными и их написание не составляет труда);
- модуль `Main` содержит основные процедуры, отвечающие за решение поставленной задачи, а именно:
  - процедура `Sort`, осуществляющая сортировку исходного файла методом естественного двухпутевого двухфазного несбалансированного слияния (сам метод будет рассмотрен позже), имя файла передается в качестве параметра;
  - процедура `Compress`, сжимающая отсортированные данные в файле, имя которого передается в качестве параметра.

В модуле `Main` вводится дополнительный тип данных, который позволяет сделать вышеупомянутые процедуры сортировки и сжатия более понятными и читабельными.

```
type  tSequence = record
                    f : tFile;      {файловая переменная}
                    elem :  tElem;  {элемент файла}
```

```
    eof : Boolean;  {признак конца файла}
    eor : Boolean   {признак конца серии}
  end;
```

Опишем эти поля более подробно:

- `f` — имя файловой переменной, т. е. откуда (из какого файла) читаются данные или куда записываются;
- `elem` — запись (элемент) файла, которая была считана из файла или записана в файл последней;
- `eof` — поле логического типа, которое принимает значение true, если обработана последняя запись файла (обработка файла закончена). Ввод такого поля обусловлен тем, что применение функции `eof()` не всегда корректно, т. е. если считан последний элемент файла, то `eof(f) = true`, а это не означает, что этот элемент был обработан (записан);
- `eor` — поле логического типа, которое принимает значение true, если закончена обработка упорядоченного отрезка (серии). Это поле специфично только для процесса сортировки, и поэтому оно будет подробнее рассмотрено при описании естественного слияния.

В модуле `Main` введен ряд вспомогательных процедур, которые позволяют работать с переменными типа `tSequence`:

- **procedure** `OpenSeq (var s: tSequence; fn: string);`

  процедура, которая открывает файл, т. е. ассоциирует файловую переменную `s.f` с файлом на диске под именем `fn`;

- **procedure** `ReadNext (var s: tSequence);`

  если не все элементы файла `s.f` еще прочитаны, то осуществляется чтение одного элемента из этого файла в поле `s.elem`, в противном случае признак конца файла `s.eof` принимает значение `true`;

- **procedure** `StartRead (var s: tSequence);`

  процедура, которая открывает файл `s.f` для чтения, осуществляет чтение первого элемента в переменную `s.elem` и определяет признак конца серии как признак конца файла, т. е. если кончился файл, то обработана и серия, и наоборот;

- **procedure** `StartWrite (var s: tSequence);`

  процедура, которая открывает файл для записи;

- **procedure** `CloseSeq (var s: tSequence);`

  процедура, которая закрывает файл.

Теперь разберем подробнее процедуру сортировки. Опишем метод естественного двухпутевого двухфазного несбалансированного слияния, на основе которого осуществляется сортировка элементов. Идея сортировки слиянием

состоит в следующем: сначала все данные из исходного файла распределяются на несколько вспомогательных файлов (в данном случае вспомогательных файлов будет два, так как слияние двухпутевое), после этого из вспомогательных файлов данные объединяются в исходный файл. Так повторяется до тех пор, пока исходный файл не будет отсортирован. Таким образом, весь процесс сортировки состоит из двух фаз: фазы распределения данных и фазы слияния. Отсюда и название сортировки — двухфазная.

Алгоритм сортировки слиянием, записанный неформально:

```
repeat
  распределение данных;
  слияние данных
until файл отсортирован;
```

Рассмотрим подробнее обе фазы, а также критерий окончания сортировки. Серией называется последовательность файла максимальной длины, между элементами которой выполняется отношение упорядоченности. Следовательно, фаза распределения заключается в том, чтобы поочередно в каждый из вспомогательных файлов записывать по одной серии.

Одним из критериев окончания сортировки слиянием является то, что при объединении серий из вспомогательных файлов в исходном файле оказалась только одна серия. Поэтому на фазе слияния надо считать, сколько серий записалось в исходный файл, т. е. в тех местах алгоритма, где происходит запись серии, счетчик серий надо увеличивать на единицу.

Алгоритм распределения данных по двум вспомогательным файлам, записанный неформально:

```
подготовить файлы (исходный для чтения, вспомогательные — для записи)
while not f0.eof do
  begin
    записать серию во вспомогательный файл f1;
    if not f0.eof then  записать серию во вспомогательный файл f2
  end;
закрыть файлы;
```

Слияние серий осуществляется следующим образом: из каждого вспомогательного файла берется по одной серии, они объединяются в одну упорядоченную серию и записываются в исходный файл. Процесс продолжается до тех пор, пока не кончится один из вспомогательных файлов. После этого неотработанные данные копируются в исходный файл.

Алгоритм слияния из двух вспомогательных файлов в один исходный, записанный неформально:

```
подготовить файлы: (исходный — для записи, вспомогательный — для чтения)
```

```
while (not f1.eof) and (not f2.eof) do
       объединить по одной серии из файлов f1 и f2 в файл f0;
дописать данные из f1 в f0;
дописать данные из f2 в f0;
```

Обратим внимание на то, что в цикле `while` хотя бы один из файлов будет прочитан полностью, следовательно, один из пунктов "дописать" не выполнится. Расшифруем подробнее строку "объединить по одной серии": пока есть элементы в сериях (не конец серии), сравниваем их и находим минимальный, записываем его в исходный файл. Из файла, откуда был взят элемент, считываем следующий.

Алгоритм для объединения двух вспомогательных файлов:

```
while  (not f1.eor) and (not f2.eor) do
  if f1.elem.name<= f2.elem.name
    then
      begin
        записать элемент из файла f1 в f0;
        считать в файле f1 следующий элемент
      end
    else
      begin
        записать элемент из файла f2 в f0;
       считать в файле f2 следующий элемент
      end
if not f1.eor then
  дописать серию до конца из f1 в f0;
if not f2.eor then
  дописать серию до конца из f2 в f0;
для вспомогательных файлов f1 и f2 изменить признаки конца серии в зависимости от признака конца файла (если файл закончен, то и серия закончена, и наоборот).
```

"Дописать данные из вспомогательного файла f1 в исходный файл f0" можно расшифровать следующим образом:

```
while not f1.eof  do
  записать серию из f1 в f0;
```

Обратим внимание на то, что часто в алгоритмах слияния и распределения данных встречается пункт "записать серию" (из файла *x* в файл *y*). Опишем эти действия отдельной процедурой, алгоритм которой выглядит следующим образом:

```
Записать серию из файла x в файл y:

repeat
```

```
  записать элемент из файла x в файл y;
  считать элемент из файла x
until  x.eor {серия закончена}
```

Определим два действия: записать элемент и считать следующий как отдельную процедуру, но при этом предварительно оговорим, как определяется признак конца серии.

Признак конца серии в сортировке естественным слиянием определяется как результат сравнения двух соседних элементов в файле. Если упорядоченность нарушается, то серия считается законченной, т. е. признак `eor` принимает значение true, в противном случае — false.

```
Алгоритм копирования элемента из файла x в файл y:
запоминаем элемент, который надо записать в поле y.elem;
записываем его (x.elem) в файл y.f;
считываем следующий элемент из файла x.f, определяя признак конца файла;
определяем признак конца серии в файле x:
    серия закончена, если закончен файл
    или только что считанный элемент x.elem меньше предыдущего (y.elem)
    (сортировка по неубыванию)
```

Алгоритм сжатия данных:

```
подготовить файлы
(исходный — для чтения, вспомогательный — для записи)
считать первый элемент;
while not f1.eof do
  begin
    запоминаем название товара и его количество;
    считываем следующий элемент;
    while not f1.eof and имена одинаковы do
      begin
        записываем результат во вспомогательный файл
        считываем следующий элемент;
      end;
  end;
закрыть файлы;
переименовать файл.
```

Листинг 6.1. Задача о товарах

```
{Модуль описания данных}
unit Common;
Interface
Type
  {Тип элемента в файле}
  tElem  = record
             name: string;
             count: integer
           end;
  {Тип файла}
  tFile  = file of tElem;

Implementation
end.

{модуль вспомогательных процедур}
Unit InpOut;
Interface

   {процедура добавления данных в файл с именем FileName}
   procedure InputData (FileName: string);
   {процедура вывода данных на экран}
   procedure OutputData (FileName: string);

Implementation
   Uses Common, Crt;

   {процедура добавления данных в файл с именем FileName}
   procedure InputData;
     var f : tFile;
         el : tElem;
   begin
     assign (f, filename);
     reset (f);
     seek (f, FileSize(f));
     writeln ('Добавьте данные. Признак конца ввода — пустое имя');
```

```
      write ('Наименование: '); readln (el.name);
      while el.name<>'' do
        begin
          write ('Количество: '); readln (el.count);
          write (f, el);
          write ('Наименование: ');  readln (el.name);
        end;
      close(f)
    end;

    {процедура вывода данных на экран}
    procedure OutputData;
      var f : tFile;
          el : tElem;
          ch : char;
    begin
      ClrScr;
      writeln ('Содержимое файла ', filename, ':');
      assign (f, filename);
      reset(f);
      writeln ('Наименование':30, 'Количество':15);
      while not eof(f) do
        begin
          read (f, el);
          writeln (el.name:30, el.count:15);
        end;
      close(f);
      ch:=readKey
    end;
end.

{модуль основных процедур}
Unit Main;
Interface

  {процедура сортировки файла с именем FileName}
  procedure Sort (FileName: string);
  {процедура сжатия данных в файле с именем FileName}
```

```
  procedure Compress (FileName: string);

Implementation
Uses Common;
type {Информация о файле}
     tSequence = record
                   f : tFile;      {файл}
                   elem : tElem;   {первый элемент в файле}
                   eof,            {признак конца файла}
                   eor: Boolean    {признак конца отрезка}
     end;

  {Процедура, которая открывает файл с именем fn}
  procedure OpenSeq (var s : tSequence; fn: string);
  begin
    assign(s.f, fn)
  end;

  {Процедура, которая считывает из файла s.f следующий элемент}
  procedure ReadNext (var s : tSequence);
  begin
    s.eof:=eof(s.f);
    if not s.eof then read(s.f, s.elem)
  end;

  {Процедура подготовки файла s.f для чтения}
  procedure StartRead (var s : tSequence);
  begin
    Reset(s.f);
    ReadNext(s);
    s.eor:=s.eof
  end;

  {Процедура подготовки файла s.f для записи}
  procedure StartWrite (var s : tSequence);
  begin
    rewrite(s.f)
  end;
```

```
{Процедура, которая закрывает файл s.f}
procedure CloseSeq (var s: tSequence);
begin
  close(s.f)
end;

{Процедура копирования элемента из файла x в файл y}
procedure Copy (var x, y : tSequence);
begin
  y.elem:=x.elem;
  write(y.f, y.elem);
  ReadNext(x);
  x.eor:=x.eof or (x.elem.name<y.elem.name)
end;

{Процедура копирования серии из файла x в файл y}
procedure CopyRun(var x, y: tSequence);
begin
  repeat
    Copy(x, y)
  until x.eor
end;

{Процедура начального распределения серий из исходного файла f0}
{во вспомогательные файлы f1 и f2}
procedure Distribute (var f0, f1, f2: tSequence);
begin
  {подготовка файлов}
  StartRead  (f0); StartWrite (f1); StartWrite (f2);
  {Распределение данных}
  while not f0.eof do
    begin
      CopyRun (f0, f1);
      if not f0.eof then CopyRun(f0,f2)
    end;
  {Закрытие файлов}
  CloseSeq(f0); CloseSeq(f1); CloseSeq(f2)
```

```
end;

{Процедура слияния серий из вспомогательных файлов f1 и f2}
{в исходный файл f0 с подсчетом количества серий count}
procedure Marge(var f0, f1, f2: tSequence; var count: integer);
begin
  {подготовка файлов}
  StartRead (f1);  StartRead (f2);  StartWrite(f0);
  {обнуление счетчика серий}
  count:=0;

  {слияние}
  while not f1.eof and not f2.eof do
    begin
      {объединение элементов из одной серии}
      while not f1.eor and not f2.eor do
        if f1.elem.name<=f2.elem.name then
           Copy(f1,f0)
        else
           Copy(f2,f0);
      {если остались элементы серии в первом файле,
      то дописываем серию до конца}
      if not f1.eor then CopyRun(f1,f0);
      {если остались элементы серии во втором файле,
      то дописываем серию до конца}
      if not f2.eor then CopyRun(f2,f0);
      {изменяем признаки конца серий в зависимости от
      признаков конца файлов}
      f1.eor:=f1.eof;
      f2.eor:=f2.eof;
      {Увеличиваем счетчик серий}
      count:=count+1
    end;

  {дописываем элементы из первого файла, если таковые остались}
  while not f1.eof do
    begin
      CopyRun(f1,f0);
```

```
      count:=count+1
    end;
  {дописываем элементы из второго файла, если таковые остались}
  while not f2.eof do
    begin
      CopyRun(f2,f0);
      count:=count+1
    end;

  {закрываем файлы}
  CloseSeq(f1); CloseSeq(f2); CloseSeq(f0)
end;

{процедура сортировки файла с именем FileName}
procedure Sort (FileName: string);
Var {вспомогательные файлов}
    f1, f2 : tSequence;
    {исходный файл}
    f0 : tSequence;
    {счетчик количества серий}
    count: integer;

begin
  {открываем файлы}
  OpenSeq(f0,filename); OpenSeq(f1,'help1'); OpenSeq(f2,'help2');

  repeat
    Distribute(f0,f1,f2); {Распределение серий}
    Marge(f0,f1,f2,count) {Слияние серий}
  until count <= 1;

  {удаление ненужных файлов}
  Erase(f1.f); Erase(f2.f)
end;

{процедура сжатия данных}
procedure Compress;
Var {f1 — исходный файл, f2 — вспомогательный файл}
```

```
      f1, f2 : tSequence;
      {вспомогательный элемент}
      el : tElem;
  begin
    {подготовка файлов}
    OpenSeq(f1,fileName); OpenSeq(f2,'Help');
    StartRead(f1); StartWrite(f2);
    {сжатие данных}
    while not f1.eof do
      begin
        el:=f1.elem;
        ReadNext(f1);
        while not f1.eof and (f1.elem.name=el.name) do
          begin
            el.count:=el.count+f1.elem.count;
            ReadNext(f1)
          end;
        write(f2.f,el)
      end;
    {закрываем файлы}
    CloseSeq(f1); CloseSeq(f2);
    {удаляем старый файл}
    Erase(f1.f);
    {переименовываем вспомогательный файл}
    Rename(f2.f,FileName)
  end;
end.

{основная программа}
Uses Crt, Main, InpOut;
  var FileName: string;  {имя файла для обработки}
begin
  ClrScr;
  {Ввод имени файла}
  writeln ('Введите имя файла для обработки');
  write   ('Имя файла:'); readln  (FileName);
  {Дополнение файла новыми данными}
  InputData(filename);
```

```
  {Печать содержимого файла до обработки}
  writeln ('До обработки');
  OutputData (filename);
  {Обработка файла}
  Sort (FileName);
  Compress (FileName);
  {Печать содержимого файла после обработки}
  writeln ('После обработки');
  OutputData (filename)
end.
```

Пример работы программы

```
Введите имя файла для обработки
Имя файла: primer

Добавьте данные. Признак конца ввода — пустое имя
Наименование: Кукла
Количество: 9
Наименование: Кукла
Количество: 15
Наименование: Конструктор
Количество: 12
Наименование: Кукла
Количество: -5
Наименование: Конструктор
Количество: -7
Наименование: Конструктор
Количество: 12
Наименование: Кукла
Количество: -3
Наименование: Мяч
Количество: 40
Наименование: Кукла
Количество: 10
Наименование: Мяч
Количество: -20
Наименование:
```

```
До обработки
Содержимое файла primer:
 Наименование Количество
 Кукла        9
 Кукла        15
 Конструктор  12
 Кукла        -5
 Конструктор  -7
 Конструктор  12
 Кукла        -3
 Мяч          40
 Кукла        10
 Мяч          -20

После обработки
Содержимое файла primer:
 Наименование Количество
 Конструктор  17
 Кукла        26
 Мяч          20
```

Задания для самостоятельной работы

Каждая из задач (1–16) должна предоставлять следующие возможности:

- создание нового файла;
- дополнение существующего файла;
- просмотр сведений из файла;
- выполнение основной задачи, указанной в упражнении.

Частное предприятие "Петров и К°" занимается приобретением у изготовителей шоколадных конфет в упаковках (коробках) и их продажей частным лицам и организациям. Для улучшения контроля за приходом и расходом товара предприятие ведет компьютерный учет. Очень часто покупателям данной фирмы требуются те или иные сведения о товаре, например, наиболее свежие конфеты или изготовленные каким-то определенным производителем.

Итак, сведения о конфетах включают в себя следующие данные:

- название;
- вес упаковки;

- ❒ цена упаковки;
- ❒ изготовитель;
- ❒ дата выпуска;
- ❒ срок хранения.

1. Вывести сведения о конфетах, упорядочив их по названию. Использовать двухпутевое однофазное естественное несбалансированное слияние.
2. Вывести сведения о конфетах, которые не дороже цены, заданной покупателем, и отсортированные в порядке повышения цены. Использовать многопутевое однофазное простое слияние.
3. Вывести данные о просроченном товаре, если известна текущая дата, и упорядочить их по полю "изготовитель". Использовать многопутевое двухфазное простое слияние с внутренней сортировкой.
4. Вывести сведения о конфетах с заданным весом упаковки, упорядочив их по дате выпуска (сначала самые свежие). Использовать двухпутевое двухфазное естественное сбалансированное слияние.
5. Вывести названия конфет и их изготовителей в порядке увеличения срока хранения конфет. Использовать многопутевое однофазное естественное несбалансированное слияние.
6. Вывести названия и дату истечения срока годности, упорядочив данные в порядке убывания отношения веса упаковки к ее цене. Использовать двухпутевое двухфазное простое слияние.
7. Вывести в алфавитном порядке названия конфет, выпускаемых заданным производителем, с указанием даты выпуска и срока хранения конфет. Использовать многопутевое однофазное простое слияние с внутренней сортировкой.

   Концерн "Видеосервис" предоставляет услуги по прокату и продаже видеофильмов частным лицам и организациям. Сведения о фильмах, имеющихся в наличии, хранятся в электронном виде. Для облегчения выбора заказа покупателю предоставляется возможность сократить каталог просмотра путем задания даты выхода фильма или, например, киностудии, выпустившей этот фильм.

   Сведения о фильмах включают в себя следующие данные:

   - ❒ название;
   - ❒ год выпуска;
   - ❒ киностудия;
   - ❒ режиссер;
   - ❒ длительность фильма;

- ❒ наличие приза;
- ❒ три главных героя.

8. Вывести в алфавитном порядке сведения о видеофильмах с указанием киностудий, их выпустивших в заданный период времени (для периода задан год начала и год конца). Использовать многопутевое двухфазное естественное несбалансированное слияние.
9. Распечатать названия фильмов, в которых снимался какой-то конкретный актер, в обратном порядке их выхода в свет (сначала самые свежие). Использовать многопутевое двухфазное простое слияние.
10. Вывести сведения о киностудиях, выпустивших фильмы в 2000 г., и о кинорежиссерах, работающих на этих студиях. Данные о киностудиях должны быть упорядочены в алфавитном порядке и для каждой киностудии сведения о режиссерах также должны быть упорядочены по алфавиту. Использовать двухпутевое однофазное естественное сбалансированное слияние.
11. Вывести сведения о фильмах, имеющих награды, упорядочив их по длительности просмотра. Использовать двухпутевое однофазное простое слияние с внутренней сортировкой.
12. Упорядочить названия фильмов по фамилиям актеров, принимающих в них участие. Использовать двухпутевое двухфазное естественное несбалансированное слияние.

    В ООН имеется полный перечень всех стран, который включает в себя:

    - ❒ название;
    - ❒ континент;
    - ❒ столицу;
    - ❒ площадь;
    - ❒ численность населения;
    - ❒ государственный строй.

    В разных ситуациях (например, для оказания помощи некоторой стране) требуется выбрать из всего списка те или иные государства. Так как сведения о странах хранятся в электронном виде, то часто бывает необходимо решать задачи, подобные нижеизложенным.

13. Указать сведения о государствах заданного континента в порядке возрастания численности населения. Использовать двухпутевое однофазное простое слияние.
14. Упорядочить по названиям столиц сведения о государствах с площадью меньше заданной. Использовать двухпутевое двухфазное простое слияние с внутренней сортировкой.

15. Указать названия государств с заданным государственным строем в алфавитном порядке. Использовать многопутевое однофазное естественное сбалансированное слияние.
16. Вывести названия и столицы государств в порядке убывания плотности населения. Применить многопутевое двухфазное естественное сбалансированное слияние.
17. Написать программу, которая осуществляет сортировку слов текстового файла (одно слово в одной строке) путем слияния данных из файла и некоторой заданной внутренней структуры:

    а) массива;

    б) линейного списка;

    в) дерева.

Указание

1) Исходный текстовый файл генерируется случайным образом: количество слов в нем случайно (до 10000) и сами слова сгенерированы при помощи датчика случайных чисел.
2) Сортировка осуществляется следующим способом: даны два вспомогательных файла и некоторая структура данных, например массив. Сначала первая порция данных из исходного файла упорядоченно перекачивается в заданную внутреннюю структуру. Для массива данные заносятся методом простых или бинарных вставок, для списка — методом вставок в список; для дерева — построением дерева-списка.
3) Далее из массива (списка, дерева) упорядоченные данные переписываются во вспомогательный файл. Затем следующая порция данных из исходного файла упорядоченно переносится в массив (список, дерево), после чего осуществляется слияние данных из внутренней структуры и ранее заполненного вспомогательного файла. Результат слияния записывается в другой вспомогательный файл. Так продолжается до тех пор, пока все данные не будут отсортированы.

18. Исследовать один из методов сортировки, указанный в задачах с 1 по 16 по заданному критерию, например, по количеству сравнений, по количеству проходов, по времени выполнения сортировки. Для этого необходимо:

    а) отсортировать файл целых чисел, сформированный случайным образом и определить значения критерия;

    б) отсортировать файл целых чисел, заданный в обратном порядке и определить значения критерия.

Пункты а) и б) нужно выполнить для файлов с количеством элементов, равным 100, 500, 1000. Результат вывести в виде таблицы (табл. 6.6).

Таблица 6.6. *Формат вывода результатов*

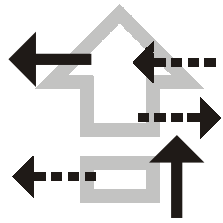
Количество элементов	**Значение критерия**	
	Случайный файл	*Обратный файл*
100		
500		
1000		

19. Реализовать алгоритм многофазной сортировки. При этом:
    а) визуализировать процесс распределения серий по вспомогательным файлам (должно быть отображено количество реальных серий, записываемых в каждый из вспомогательных файлов на данном уровне и количество фиктивных серий, остающихся во вспомогательных файлах);
    б) визуализировать процесс слияния серий, причем должны выделяться вспомогательные файлы, участвующие в слиянии, и должно отображаться изменение количества серий (и реальных, и фиктивных) в этих файлах (не сами элементы файлов).
20. Реализовать алгоритм каскадной сортировки. При этом:
    а) визуализировать процесс распределения серий по вспомогательным файлам (должно быть отображено количество реальных серий, записываемых в каждый из вспомогательных файлов на данном уровне, и количество фиктивных серий, остающихся во вспомогательных файлах);
    б) визуализировать процесс слияния серий, причем должны выделяться вспомогательные файлы, участвующие в слиянии, и должно отображаться изменение количества серий (и реальных, и фиктивных) в этих файлах (не сами элементы файлов).
21. В файле хранится последовательность русских слов. Упорядочить ее в алфавитном порядке.

Указание

1) Применять внешнюю сортировку.
2) Учесть, что порядок кодов букв русского алфавита не соответствует порядку букв в алфавите.

Глава 7



Хеширование

Большинство методов поиска основывается на упорядочении данных. Однако эффективность поиска можно повысить, если вместо упорядоченных структур использовать так называемые беспорядочные или перемешанные структуры данных. Для работы с такими структурами предназначены методы хеширования. Среднее время поиска элемента этими методами есть $O(1)$, время для наихудшего случая — $O(n)$. Здесь n — общее число элементов.

Постановка задачи

Пусть задано некоторое количество записей, среди которых осуществляется поиск. Предположим, что записи размещены в оперативной памяти и, следовательно, каждой можно поставить в соответствие число a из диапазона $0 \ldots m-1$, которое назовем *адресом* записи. Например, записи размещены в одномерном массиве; в качестве значения a можно взять смещение записи относительно начала массива, а в качестве m — размер массива.

Назовем *ключом* записи некоторый числовой идентификатор, однозначно определяющий эту запись.

Обозначим через k — ключ записи с адресом a. Тогда *задача поиска* состоит в определении по заданному аргументу k адреса записи $a(k)$, содержащей заданный ключ.

Основная идея ассоциативной адресации состоит в том, чтобы сохранить без изменения неупорядоченную структуру и, имея ключ для поиска записи, определить адрес без предварительного сравнения ключей, то есть рассматривать адрес как функцию от ключа.

Общие понятия

Организацию данных в виде таблицы назовем хеш-таблицей, если адрес каждой записи a этой таблицы определяется как значение некоторой функции $h(k)$, называемой хеш-функцией. Здесь k — значение ключа записи a.

Если число записей невелико и заранее известно, то можно построить функцию преобразования заданного множества ключей в различные адреса. Если возможно, то в последовательные адреса, что выгодно при использовании виртуальной памяти. Если же число записей велико, то найти такую функцию оказывается сложно. В случае если число различных значений ключей, вероятность появления которых отлична от нуля, превышает размер хеш-таблицы, построение функции оказывается невозможным. В этом случае приходится отказываться от идеи однозначности и рассматривать хеш-функцию как функцию, рассеивающую множество ключей во множестве адресов $0m - 1$.

Отказ от требования взаимно однозначного соответствия между ключом и адресом означает, что для двух различных ключей $k1 \neq k2$ значение хеш-функции может совпадать: $h(k1) = h(k2)$. Такая ситуация называется *коллизией*.

Ключи $k1$ и $k2$ называются *синонимами* хеш-функции h, если $h(k1) = h(k2)$.

Для метода хеширования главными задачами являются выбор хеш-функции h и нахождение способа разрешения возникающих коллизий.

Хеш-функции

При выборе хеш-функции следует учитывать сложность ее вычисления, а также равномерность распределения значений, которая позволяет не только сократить число коллизий, но и не допустить скучивания значений в отдельных частях таблицы.

Для каждого конкретного множества возможных ключей можно изобрести (подобрать, найти) свою, возможно наилучшую, хеш-функцию распределения ключей по таблице. Но существуют и универсальные хеш-функции, дающие хорошие результаты в большинстве случаев. Рассмотрим некоторые из них.

Метод деления

В методе деления в качестве значения хеш-функции h используется остаток от деления ключа на некоторое целое число M

$$h(k) = k \bmod M,$$

где M обычно равняется размеру хеш-таблицы. Эффективность рассеивания ключей во многом зависит от значения M. Не стоит выбирать M равным степени основания системы счисления, так как значением хеш-функции будут просто младшие разряды ключа. Например, для символьных ключей не следует выбирать M равным 2^8 или 2^{16}. В этом случае хеш-функция будет

равняться одной или двум последним буквам ключа. Для предотвращения скучивания ключей следует выбирать M равным простому числу.

Метод деления часто используется после применения другой хеш-функции для соответствия полученных значений размеру хеш-таблицы.

Метод свертки (слияния)

Предположим, что ключ представлен в виде последовательности разрядов a_i: $k = a_1 a_2 a_3 a_p$, где p кратно некоторому числу w. Тогда значением хеш-функции будет сумма $h(k) = a_1 a_2 a_w \oplus a_{w+1} a_{w+2} a_{2w} \oplus \ldots \oplus a_{p-w+1} a_{p-w} a_p$, где в качестве операции $\oplus$ может использоваться операция арифметического или побитового сложения, побитовая операция "исключающее или" и т. д.

Для символьных ключей в качестве w удобно выбирать значения, кратные восьми. Основной недостаток этого метода состоит в том, что он не очень чувствителен к порядку символов. Но избавиться от этого просто. Допустим, что результат каждого последовательного применения операции $\oplus$ сохраняется в переменной h', конечное значение которой было результатом вычисления хеш-функции. Тогда для воздействия порядка символов на значение $h(k)$ необходимо применять операцию циклического сдвига h перед очередным применением операции $\oplus$.

Метод умножения

Представим значение ключа k в виде двоичного числа и примем размер хеш-таблицы m равным 2^p. Умножим дробь d на k и возьмем дробную часть числа, которую обозначим как $\{k \times d\}$, а в качестве значения хеш-функции используем p старших разрядов[1], т. е. $h(k) = \lfloor m \times \{kd\} \rfloor$, где $\lfloor x \rfloor$ — наибольшее целое число, не превосходящее x. Рекомендуется в качестве значения d брать иррациональное число, например золотое сечение $(\sqrt{5} - 1)/2$. При $d = 1/m$ метод эквивалентен методу деления.

Метод "середины квадрата"

Пусть m (размер хеш-таблицы) равен 2^p. Обозначим $d = k^2$ и представим d в виде двоичного числа. Тогда значением хеш-функции $h(k)$ будет p битов средней части d. Данный метод по многим параметрам уступает методу умножения.

---

[1] Требование, чтобы при вычислении $h(k)$ размер хеш-таблицы равнялся степени 2, не является обязательным. Оно было использовано только для облегчения понимания работы метода.

Метод преобразования системы счисления

В основе метода лежит преобразование значения ключа k, выраженного в системе счисления с основанием p ($k = a_0 + a_1p + a_2p_2 + ...$), в систему счисления с основанием q ($h(k) = a_0 + a_1q + a_2q_2 + ...$) при условии, что $p < q$. Трудоемкость (число операций) этого метода оказывается большей, чем методов деления или умножения.

Метод деления многочленов

Пусть k, выраженное в двоичной системе счисления, записывается как $k = 2^n b_n + ... + 2b_1 + b_0$, и пусть размер хеш-таблицы m является степенью двойки $m = 2^p$. Представим двоичный ключ k в виде многочлена вида $k(t) = b_n t^n + ... + b_1 t + b_0$. Определим остаток от деления этого многочлена на постоянный многочлен вида $c(t) = t^m + c_{m-1}t^{m-1} + ... + c_1 t + c_0$. Этот остаток, рассматриваемый в двоичной системе счисления, используется в качестве значения хеш-функции $h(k)$. Для вычисления остатка от деления многочленов используют полиномиальную арифметику по модулю 2. Если в качестве $c(t)$ выбрать простой неприводимый многочлен, то при условии близких, но не равных $k1$ и $k2$, обязательно будет выполняться условие $h(k1) \neq h(k2)$. Многочлен $c(t)$ называется простым неприводимым многочленом, если его нельзя представить в виде произведения $c(t) = q(t) \times r(t)$, где $q(t)$ и $r(t)$ — многочлены, отличные от константы. Эта функция обладает сильным свойством рассеивания скученностей.

Методы разрешения коллизий

При работе с хеш-таблицей выделяются три основные операции: вставка, поиск и удаление элемента. Причем существует круг задач, в которых используются только первые две. При решении задачи следует учитывать набор предполагаемых операций, так как, например, операция удаления может привести к изменению структуры данных, соответствующей используемому методу разрешения коллизий.

Методы разрешения коллизий можно разделить на два класса: метод цепочек и метод открытой адресации.

Метод цепочек

В этом методе для разрешения коллизий в каждую запись хеш-таблицы добавляется указатель для поддержания связанного списка. Сами списки могут размещаться как в памяти, принадлежащей хеш-таблице (внутренние цепочки), так и в отдельной памяти (внешние цепочки).

Внешние цепочки

Хеш-таблица представляет собой массив связанных списков размера m (элементы массива обычно имеют индексы от 0 до $m - 1$). После вычисления значения хеш-функции $a = h(k)$ задача сводится к последовательному поиску в a-ом списке (см. рис. 7.1).

Возможно, что список размещается во внешней памяти. В этом случае для ускорения поиска желательно, чтобы записи с ключами-синонимами при вставке попадали в один кластер файла.

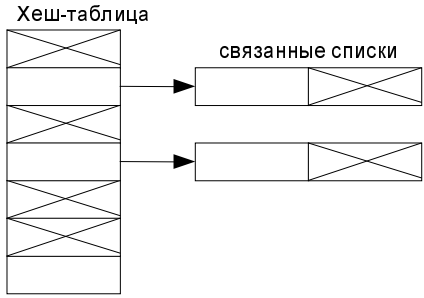


Рис. 7.1. Метод внешних цепочек

Если число синонимов становится слишком большим, можно вместо линейных списков использовать дерево поиска.

Внутренние цепочки

В методе внутренних цепочек (см. рис. 7.2) связанные списки для синонимов поддерживаются внутри таблицы. Для поиска свободного места в таблице можно использовать разные методы. Например, последовательный просмотр позиций таблицы.

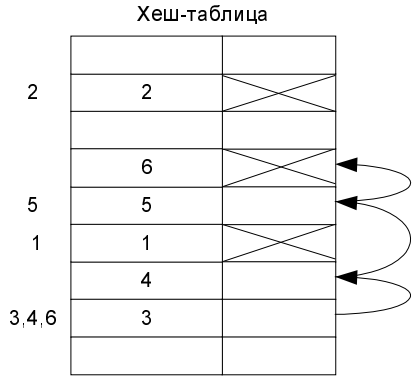


Рис. 7.2. Метод внутренних цепочек

При вставке новых ключей проявляется тенденция объединения хеш-адресов в группы, что приводит к срастанию списков. Таким образом, в один список могут попасть ключи, не являющиеся синонимами по хеш-функции, удлиняя список. Часто метод внутренних цепочек называют методом *срастающихся цепочек*.

Метод открытой адресации

В этом методе все элементы хеш-таблицы хранятся в одномерном массиве. Если при добавлении нового элемента возникает коллизия, то производится поиск свободного места в следующей ячейке таблицы. Адрес следующей ячейки вычисляется при помощи некоторой функции, аргументами которой в общем случае являются: значение ключа, первичный хеш-адрес, полученный при первом применении хеш-функции к текущему ключу, номер шага при поиске свободной ячейки.

Введем обозначения:

k — значение ключа;

$h_0(k)$ — первичный хеш-адрес;

i — номер шага при поиске свободной ячейки, $i = 1, 2, ...$;

$h_i(k)$ — значение хеш-адреса, полученного на i-ом шаге.

Тогда алгоритм поиска/добавления элемента будет выглядеть так.

1. Полагаем $i = 1$.
2. Вычисляем $a = h_i(k)$.
3. Если a свободно, то алгоритм завершается (процедура вставки сохраняет элемент в ячейке a, процедура поиска сообщает об отсутствии ключа в хеш-таблице).
4. Если ключ в ячейке a равен k, то алгоритм завершается (процедура вставки повторно не сохраняет элемент в ячейке, процедура поиска сообщает о найденном элементе).
5. Обнаружена коллизия. Полагаем $i = i + 1$ и переходим к пункту 2.

Линейное опробование

В данном методе $h_i(k) = h_0(k) + ci$, где c — константа. В простейшем случае c полагается равной 1 или -1. В этом случае возникает опасность скучивания синонимов (при использовании виртуальной памяти это свойство будет преимуществом метода). Для устранения скучивания c и m (размер хеш-таблицы) должны быть взаимно простыми, а c — не слишком малым числом.

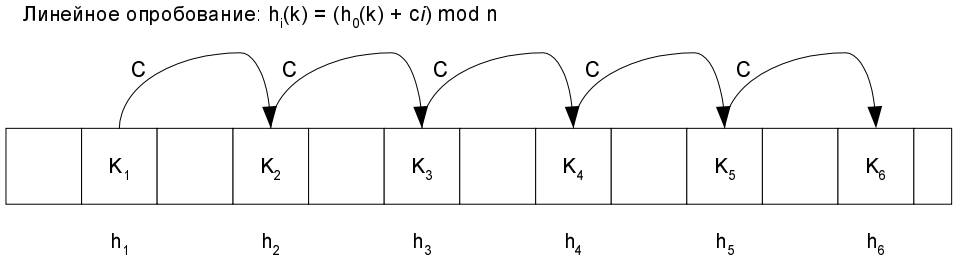


Рис. 7.3. Линейное опробование: $h_i(k) = (h_0(k) + ci) \bmod n$

Квадратичное опробование

В линейном опробовании значение $h_0(k)$, попавшее на элемент последовательности синонимов, всегда удлиняет эту последовательность. Устранить этот недостаток позволяет введение нелинейности в хеш-функцию: $h_i(k) = h_0(k) + ci + di^2$, где c и d — константы.

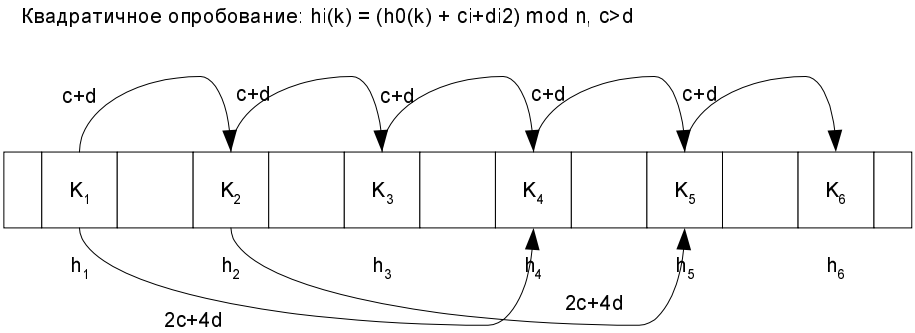


Рис. 7.4. Квадратичное опробование: $h_i(k) = (h_0(k) + ci+di^2) \bmod n$, c>d

На рис. 7.3 и 7.4 приведен пример однотипной ситуации, при которой:

- $h_0(k_1) = h_0(k_2) = h_0(k_3)$
- $h_0(k_4) = h_1(k_1)$
- $h_0(k_5) = h_0(k_6)$
- $h_1(k_5) = h_0(k_1)$

Как видно, в случае линейного опробования возникает 10 коллизий, а в случае квадратичного — 5.

Двойное хеширование

Здесь $h_i(k) = h_0(k) + ig(k)$, где $g(k)$ — хеш-функция, отличная от h_0.

Если размер хеш-таблицы m — простое число, то можно рекомендовать такой вид хеш-функции:

$h_i(k) = (k \bmod m) + i\,(1 + k \bmod (m - 2))$.

Для независимых функций h_0 и g вероятность коллизий при двойном хешировании будет составлять $1 / m^2$.

Замечание к операции удаления элементов из хеш-таблицы

Во всех рассмотренных методах разрешения коллизий, за исключением метода, основанного на хеш-таблицах с внешними цепочками, удаление элемента из хеш-таблицы не всегда является операцией обратной вставке, поскольку при удалении нарушается связь между синонимами. Чтобы этого не произошло, можно, не производя удаления записи, вместо ключа, ставшего ненужным, вписать специальный код. Этот код во время поиска пропускается, но разрешает использовать эту позицию для вставки нового элемента. Таким образом, каждая позиция хеш-таблицы может находиться в одном из трех состояний: свободном, занятом или удаленном. Если не предпринимать мер по переводу позиций из состояния "удалено" в состояние "свободно", то при постоянном изменении хеш-таблицы может наступить ситуация, когда не останется свободных ячеек. Скорость поиска заметно ухудшится и будет соответствовать линейному поиску элемента в неупорядоченной последовательности.

Интерфейс модуля HashTable (хеш-таблица)

Определим интерфейс модуля, предоставляющего набор методов для работы с хеш-таблицей.

```
unit HashTable;
interface
  {предполагается наличие следующих типов:}
  {THashTable - тип «хеш-таблица»}
  {TKey - тип ключа}
  {TItem - тип элемента данных}

  {создание хеш-таблицы заданного размера size}
```

```
  procedure Create (size: integer; var t: THashTable);

  {получение размера хеш-таблицы}
  function GetSize (t: THashTable): integer;

  {удалить все элементы в хеш-таблице}
  procedure Clear (var t: THashTable);

  {добавление элемента в хеш-таблицу.}
  {Функция возвращает значение true }
  {в случае успешности выполнения операции вставки}
  function Insert (var t: THashTable; k: TKey; i: TItem): boolean;

  {поиск элемента данных по ключу.}
  {Функция возвращает значение true}
  {в случае успешности выполнения операции поиска}
  function Find (t: THashTable; k: TKey; var i: TItem): boolean;

  {удаление элемента по заданному ключу.}
  {Функция возвращает значение true }
  {в случае успешности выполнения операции удаления}
  function Delete (var t: THashTable; k: TKey): boolean;

implementation
...
end.
```

Пример. Работа с хеш-таблицей

Постановка задачи

Реализовать модуль работы с хеш-таблицей (листинг 7.1), хранящей информацию об абоненте (номер АТС, фамилия, имя, отчество и т. п.). Необходимо обеспечить добавление новых данных в хеш-таблицу и поиск сведений об абоненте по заданному номеру телефона.

Комментарий к задаче

В данной задаче записью является информация об абоненте, а ключом — номер его телефона. Предлагается организовать хеш-таблицу в виде линейного

массива, в качестве хеш-функции выбрать метод свертки, а в качестве способа разрешения коллизий — линейное опробование.

Вариант метода свертки, используемый в данной задаче: шестизначный номер АТС $k = a_1a_2a_3a_4a_5a_6$ разбивается на две трехзначные цифры $a_1a_2a_3$ и $a_4a_5a_6$; адресом записи в хеш-таблице будет $h(k) = (a_1a_2a_3 + a_4a_5a_6) \bmod M$, где M — размер хеш-таблицы.

Линейное опробование состоит в том, что при возникновении коллизии при добавлении запись помещается в следующую свободную ячейку хеш-таблицы.

Листинг 7.1. Модуль работы с хеш-таблицей и пример его использования

```
unit Hash;
interface
const HTableLen = 200;          {Размер хеш-таблицы}
      PhoneLen  = 6;            {Число цифр в телефонном номере}
      FIOLen    = 50;           {Длина строки для хранения ФИО}

type
  TPhone = string[PhoneLen];
  TFIO   = string[FIOLen];
  TInfo  = record               {Хранимая в хеш-таблице информация}
     Phone: TPhone;             {Шестизначный телефон абонента}
     FIO  : TFIO                {ФИО абонента}
  end;
  THashItem = record            {Ячейка хеш-таблицы}
    Info: TInfo;                {Информация}
    used: Boolean               {Признак использования ячейки}
  end;
  THashTable = record           {Хеш-таблица}
    Size : Integer;
    H    : array [0..HTableLen-1] of THashItem
  end;

  {Ввод телефона}
  procedure InputPhone(var Phone: TPhone);
  {Ввод структуры Info}
  procedure InputInfo(var Info: TInfo);
```

```
  {Инициализация хеш-таблицы}
  procedure HashInit( var HTable: THashTable );
  {Поиск в хеш-таблице.
  Info - информация об абоненте искомым номером телефона Phone.}
  {Функция возвращает значение ИСТИНА, если телефон найден}
  function HashFind( var HTable: THashTable; Phone: TPhone;
                     var Info: TInfo ): Boolean;
  {Вставка в хеш-таблицу. Info - информация об абоненте.}
  {Функция возвращает значение ИСТИНА,
   если информация вставлена в хеш-таблицу}
  function HashAdd( var HTable: THashTable; Info: TInfo ):Boolean;
  {Печать заполненных ячеек хеш-таблицы}
  procedure HashPrint( var HTable: THashTable );

implementation

{Проверка правильности записи номера телефона}
function ValidPhone( Phone: TPhone ): Boolean;
var i: Integer;
begin
  if Length(Phone)=PhoneLen then
    begin
      i:=0;
      {Проверяем, чтобы номер телефона состоял только из цифр}
      repeat
        i:=i+1
      until (i>PhoneLen) or not (Phone[i] in ['0'..'9']);
      ValidPhone := i>PhoneLen
    end
  else
    ValidPhone := false
end;

{Ввод телефона}
procedure InputPhone(var Phone: TPhone);
begin
  repeat
    Write  ( 'Введите телефон: ' );
```

```
    ReadLn ( Phone )
  until ValidPhone ( Phone );
end;

{Ввод структуры Info}
procedure InputInfo(var Info: TInfo);
begin
  InputPhone ( Info.Phone );
  Write ( 'Введите ФИО: ' );
  ReadLn ( Info.FIO )
end;

{Инициализация хеш-таблицы}
procedure HashInit( var HTable: THashTable );
var i: Integer;
begin
  HTable.Size := 0;
  for i:=0 to HTableLen-1 do
    HTable.H[i].used := false
end;

function HashKey( Phone: TPhone ): Integer;
begin
  {В качестве хеш-функции используем метод свертки}
  HashKey := (Ord(Phone[1])*100+Ord(Phone[2])*10+Ord(Phone[3])+
              Ord(Phone[4])*100+Ord(Phone[5])*10+Ord(Phone[6]))
              mod HTableLen
end;

{Поиск в хеш-таблице. Info - информация об абоненте искомым номером теле-
фона Phone.}
{Функция возвращает значение ИСТИНА, если телефон найден}
function HashFind( var HTable: THashTable; Phone: TPhone;
                   var Info: TInfo ): Boolean;
var i: Integer;
begin
  i := HashKey( Phone );
  while HTable.H[i].used and (HTable.H[i].Info.Phone<>Phone) do
```

```
    i := (i+1) mod HTableLen;
  if HTable.H[i].used then
    begin
      Info := HTable.H[i].Info;
      HashFind := true
    end
  else
    HashFind := false
end;

{Вставка в хеш-таблицу. Info - информация об абоненте.}
{Функция возвращает значение ИСТИНА, если информация вставлена в хеш-
таблицу}
function HashAdd( var HTable: THashTable; Info: TInfo ): Boolean;
var inf : TInfo; i   : Integer;
begin
  if (HTable.Size=HTableLen-1) or HashFind(HTable,Info.Phone,inf)
    then HashAdd := false
    else
      begin
        i := HashKey( Info.Phone );
        while HTable.H[i].used do i := (i+1) mod HTableLen;
        HTable.H[i].used := true;
        HTable.H[i].Info := Info;
        HTable.Size := HTable.Size + 1
      end
end;

{Печать заполненных ячеек хеш-таблицы}
procedure HashPrint( var HTable: THashTable );
var i: Integer;
begin
  WriteLn('Содержимое хеш-таблицы:');
  WriteLn('<NN> <Телефон> <Фамилия Имя Отчество>');
  for i:=0 to HTableLen-1 do
    with HTable.H[i] do
      if used then
        WriteLn(i:3,':  ',Info.Phone, '    ', Info.FIO)
```

```
end;
end.

{главная программа}
program HashTest;
uses Hash, Crt;
var  H     : THashTable;  {хеш-таблица}
     Info  : TInfo;       {Информация об абоненте}
     Phone : TPhone;      {Телефонный номер абонента}

  {Ввод данных в хеш-таблицу с отображением результата операции}
  procedure Input;
  begin
    writeln('Вставляемые данные');
    WriteLn(Info.Phone, '    ', Info.FIO);
    if HashAdd( H, Info )
      then WriteLn('Запись в хеш-таблицу прошла успешно')
      else WriteLn('Операция записи в хеш-таблицу отклонена');
    writeln;
  end;

begin
  ClrScr;
  WriteLn('< Инициализация >');
  HashInit( H );
  WriteLn('< Ввод информации в хеш-таблицу >');
  Info.Phone := '711711';
  Info.FIO   := 'Мельников Вадим Митрофанович. Релекс';
  {Данная операция записи в хеш-таблицу должна быть выполнена}
  Input;
  {Данная операция записи в хеш-таблицу должна быть отклонена}
  Input;
  Info.Phone := '789698';
  Info.FIO   := 'Горбенко Олег Данилович. Кафедра МО ЭВМ';
  {Данная операция записи в хеш-таблицу должна быть выполнена}
  Input;
  {Ввод структуры Info пользователем}
  writeln('Введите новые данные');
```

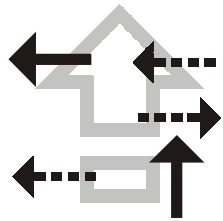
```
  InputInfo( Info );
  Input;
  {Печать заполненных ячеек хеш-таблицы}
  HashPrint( H );
  {********************** Поиск **************************}
  WriteLn('< Поиск >');
  Phone := '711711';
  WriteLn('Поиск ФИО по номеру телефона ',Phone, ':');
  if HashFind( H, Phone, Info )
    then WriteLn( 'ФИО: ', Info.FIO )
    else WriteLn( 'Указанный номер телефона не найден');
  {Ввод номера телефона пользователем}
  InputPhone( Phone );
  WriteLn('Поиск ФИО по номеру телефона ',Phone, ':');
  if HashFind( H, Phone, Info )
    then WriteLn( 'ФИО: ', Info.FIO )
    else WriteLn( 'Указанный номер телефона не найден');
  WriteLn ( 'Нажмите клавишу <Enter> ...' );
  ReadLn
end.
```

Задания для самостоятельной работы

1. Задан текст программы на языке Паскаль. Определить частоту использования:
   а) идентификаторов;
   б) ключевых слов;
   в) символьных констант.
2. Задан текстовый файл. Сформировать список слов, употребляющихся в тексте более пяти раз.
3. Задан текстовый файл. Сформировать набор словосочетаний по два и более слова, встречающихся в тексте не менее двух раз.
4. Задан набор записей следующей структуры: табельный номер, ФИО, заработная плата. По табельному номеру найти остальные сведения.
5. Задан набор записей следующей структуры: номер телефона, ФИО, адрес. По номеру телефона найти сведения о ФИО владельца и его адресе.

6. Задан набор записей следующей структуры: название кинофильма, режиссер, список актеров, краткое содержание. По заданному названию фильма найти остальные сведения.
7. Задан набор записей следующей структуры: номер автомобиля, его марка и ФИО владельца. По номеру автомобиля найти остальные сведения.
8. Задан список имен людей. Определить частоту использования каждого имени в некотором тексте.
9. Создать модуль, реализующий методы для работы с хеш-таблицей: инициализация, добавление элемента, удаление элемента, поиск. Кроме того, при заполнении хеш-таблицы выше заданного уровня размер хеш-таблицы должен автоматически увеличиваться.
10. Исследовать зависимость числа коллизий от коэффициента заполненности хеш-таблицы.
11. Из файла, содержащего текст на русском языке, слова помещаются в хеш-таблицу. Определить среднее число коллизий для нескольких заданных хеш-функций.

Глава 8



Сильно ветвящиеся деревья

Можно дать несколько различных определений дерева.

- *Деревом* называется связный ориентированный граф без циклов, каждая вершина которого имеет только одно входящее ребро.
- Н. Вирт в [3] приводит следующее определение: "Древовидная структура с базовым типом T — это либо пустая структура, либо узел типа T, с которым связано конечное число древовидных структур с базовым типом T, называемых поддеревьями".

Узел и поддеревья связаны между собой ветвями. Число ветвей, выходящих из узла, определяют его *арность*. Если все узлы имеют одинаковую арность, равную n, то такая структура называется *n-арным деревом*. Если $n = 2$, то дерево называется *бинарным*.

Упорядоченное дерево — это дерево, у которого ветви каждого узла упорядочены. Еще такие деревья называются *деревьями поиска*. Например, бинарное дерево упорядочено, если для каждого из узлов слева находится узел с меньшим значением ключа, а справа — с большим. Ключ узла, расположенного выше, может совпадать с ключом или частью ключа, нижерасположенного узла.

Упорядоченные n-арные деревья, где $n > 2$, называются *сильно ветвящимися* (multiway trees). Структура сильно ветвящихся деревьев существенно отличается от структуры бинарных деревьев.

Важная область применения сильно ветвящихся деревьев — формирование и использование крупномасштабных деревьев поиска, в которых необходимы и включения, и удаления, но для которых оперативная память недостаточно велика. Сильно ветвящиеся деревья могут оказаться удобными для поиска во внешней памяти, а также в методах индексно-последовательной организации файла.

Рассмотрим несколько видов сильно ветвящихся деревьев.

Для поиска данных, хранящихся во внешней памяти, важно сократить число перемещений данных из внешней памяти в оперативную. Более быстрым

оказывается последовательный доступ к смежным участкам памяти, нежели прямой доступ к данным, находящимся в разных местах памяти.

Для организации поиска во внешней памяти системное программное обеспечение использует так называемые B-деревья.

B-деревом порядка n называется сильно ветвящееся дерево арности $2n + 1$, обладающее следующими свойствами:

- каждый узел, за исключением корня, содержит не менее n и не более $2n$ ключей;
- корень содержит не менее одного и не более $2n$ ключей;
- все листья расположены на одном уровне;
- каждый нетерминальный узел содержит два списка: упорядоченный по возрастанию значений список ключей и соответствующий ему список указателей. Для терминальных узлов список указателей отсутствует;
- каждый нетерминальный узел содержит число указателей на единицу большее числа ключей.

  Алгоритмы включения и удаления элементов в B-дерево подробно описаны в [3].

Одной из разновидностей B-деревьев являются *B+ -деревья*. Истинные значения ключей B+ -дерева содержатся только в листьях, а в нетерминальных узлах находятся ключи-разделители, задающие диапазон изменения ключей для поддеревьев.

Другая разновидность B-деревьев — B-деревья первого порядка ($n = 1$). Бинарное B-дерево (BB-дерево) может содержать один или два ключа и, соответственно, иметь две или три связи. Все его листья расположены на одном уровне. Еще такие деревья называются *(2-3)-деревьями* или *(3-2)-деревьями* (по количеству связей). Поведение (2-3)-деревьев при случайных вставках ключей характеризуется следующим. Если дерево содержит n ключей, то существует ровно $n + 1$ путь от вершины до внешнего узла. При случайных вставках вероятности выбора любого из этих путей равны. При этом среднее число внутренних узлов при достаточно больших n лежит между $0.7n$ и $0.79n$.

Название *Trie-дерево* происходит от искусственно образованного слова *trie*, от полного слова *retrieval* (поиск). Такое дерево используется тогда, когда ключами поиска являются достаточно короткие слова. Чем больше коротких слов содержится в *Trie-дереве*, тем эффективнее соотношение количества слов к количеству памяти, расходуемой под хранение элементов в *Trie-дереве*. Каждый ключ в *Trie-дереве* можно рассматривать как список символов, а все списки вместе — как дерево поиска. В такой структуре узлу уровня $i + 1$ ставится в соответствие i-ый символ слова, поэтому каждый узел содержит лишь один символ. Методы поиска по такому дереву экономны по памяти и по времени.

Пример. Реализация Trie-дерева

Реализовать Trie-дерево как объект, позволяющий осуществлять следующие операции (листинг 8.1):

- создание Trie-дерева;
- проверка на пустоту;
- добавление слова в Trie-дерево;
- поиск слова в Trie-дереве;
- удаление слова из дерева;
- печать Trie-дерева;
- очистка и уничтожение дерева.

Листинг 8.1. Модуль для работы с Trie-деревом

```
unit TrieTree;
interface
Type
  {индексы, на основе которых построено Trie-дерево}
  TIndex = 'a'..'z';
  {Описание Trie-дерева}
  PtrTrieTree = ^ TNode;
  TNode  = record
             Ptrs : array [tIndex] of PtrTrieTree;
             eow  : boolean;
           end;

  {Объект Trie-дерево}
  TTrieTree = object
               private
                 {Указатель на корень дерева}
                 TTree : PtrTrieTree;
               public
                 {Инициализация дерева}
                 Constructor Init;
                 {Проверка дерева на пустоту}
                 function IsEmpty : boolean;
                 {Процедура помещения слова в дерево}
```

```
                Procedure   Push (s : string);
                {Функция проверки, есть ли слово s в дереве}
                Function    Find (s : string): boolean;
                {Процедура удаления слова из дерева}
                procedure   Pop  (s : string);
                {Процедура печати дерева}
                procedure   Print;
                {Процедура удаления всех слов из дерева}
                procedure   Clear;
                {Удаление дерева}
                Destructor  Done; virtual;
              end;
Implementation

Constructor TTrieTree.Init;
begin
  TTree:=nil;
end;

function TTrieTree.IsEmpty;
begin
  IsEmpty:=TTree=nil
end;

procedure TTrieTree.Push;

  procedure PushString(var T : PtrTrieTree; i:byte);
  var ch: TIndex;
  begin
    if T=nil then
      begin
        new(T);
        T^.eow:=false;
        for ch:=low(TIndex) to High(TIndex) do
          T^.Ptrs[ch]:=nil;
      end;
    if Length(s)<i then
       T^.eow :=true
```

```
    else
      PushString(T^.Ptrs[S[i]],i+1);
  end;

begin
  if Length(s)>0 then PushString(TTree,1)
end;

function TTrieTree.Find;

  function FindString(T : PtrTrieTree; i:byte):boolean;
  var ch: TIndex;
  begin
    if T=nil then
      FindString:=false
    else
      if Length(s)<i then
        FindString:=T^.eow
      else
        FindString:=FindString(T^.Ptrs[s[i]],i+1);
  end;

begin
  if Length(s)=0 then
    Find:=false
  else
    Find:=FindString(TTree,1)
end;

procedure TTrieTree.Pop ;

  function AllEmpty (T:PtrTrieTree) : boolean;
  var ch : TIndex;
      fl : boolean;
  begin
    fl:=not T^.eow;
    ch:=low(TIndex);
    while (ch<=High(TIndex)) and fl do
```

```
      if T^.Ptrs[ch]=nil then
        ch:=succ(ch)
      else
        fl:=false;
    AllEmpty:=fl
  end;

  procedure PopString (var T: PtrTrieTree; i:byte);
  var ch: TIndex;
  begin
    if (T<>nil) then
      if i<=length(s) then
        PopString(T^.Ptrs[s[i]],i+1)
      else
        begin
          T^.eow :=false;
          if AllEmpty(T) then
            begin
              dispose(T);
              T:=nil
            end;
        end;
  end;

begin
  if length(s)>0 then
    PopString(TTree,1);
end;

procedure TTrieTree.Print;

  procedure PrintString (T:PtrTrieTree; s:string);
    var ch : TIndex;
  begin
    if T^.eow then
       writeln(s);
    for ch:=Low(TIndex) to High(TIndex) do
      if T^.Ptrs[ch]<>nil then
```

```
        PrintString(T^.Ptrs[ch],s+ch)
  end;
begin
  if not IsEmpty then
    PrintString(TTree,'')
  else
    writeln('Дерево пусто')
end;

procedure TTrieTree.Clear;

  procedure DelNodes (var T:PtrTrieTree);
  var ch : TIndex;
  begin
    for ch:=low(TIndex) to High(TIndex) do
      if T^.Ptrs[ch] <> nil then
        DelNodes(T^.Ptrs[ch]);
    Dispose(T);
    T:=nil;
  end;

begin
  if not IsEmpty then DelNodes(TTree);
end;

destructor TTrieTree.Done;
begin
  if TTree<>nil then Clear;
end;
end.

{Основная программа для тестирования}
Uses TrieTree;
var TT : TTrieTree; s : string;
begin
  TT.Init; {Инициализация Trie-дерева}
  {Ввод данных в дерево}
  writeln('Введите слова, которые надо поместить в дерево');
```

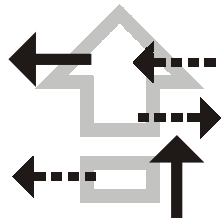
```
  readln(s);
  while s<>'' do
    begin
      TT.Push(s);
      readln(s)
    end;
  {Печать дерева}
  TT.Print;
  readln;
  {Поиск слова в дереве и его удаление}
  writeln('Введите слово, которое надо удалить');
  readln(s);
  if TT.Find(s) then
    begin
      TT.Pop(s);
      writeln('Слово удалено');
    end
  else
    writeln('Такого слова нет в дереве');
  writeln('Контрольная печать содержимого дерева');
  TT.Print;
  readln;
  {Очистка дерева}
  TT.Clear;
  if TT.IsEmpty then
    writeln('Дерево пусто')
  else
    writeln('Дерево не пусто');
  readln;
  TT.Done;
end.
```

Задания для самостоятельной работы

1. Дано Trie-дерево. Подсчитать количество слов, начинающихся с заданной последовательности символов.
2. Дан текстовый файл, состоящий из слов, разделенных пробелами. Построить Trie-дерево, которое содержит перевернутые слова из файла. Найти все слова, имеющие заданное окончание.

3. В Trie-дереве определить количество слов, содержащих букву A.
4. В Trie-дереве подсчитать количество слов, которые содержат определенное количество согласных.
5. Дан текстовый файл, который содержит слова, возможно, повторяющиеся. Распечатать их в алфавитном порядке, указав число вхождений каждого слова в текст.
6. Дано Trie-дерево и некоторое слово. Вывести все формы данного слова, содержащиеся в дереве. Формой слова считается некоторое другое слово, отличающееся от заданного не более чем на *n* последних символов, где *n* вводит пользователь.
7. Из Trie-дерева удалить все слова, которые содержат указанную подстроку.
8. В Trie-дереве найти все слова, содержащие только символы заданного множества.
9. Для заданного Trie-дерева построить Trie-дерево, в котором все слова расположены в обратном порядке.
10. Из Trie-дерева удалить все слова с четной длиной.
11. Посчитать в Trie-дереве количество слов, оканчивающихся на согласную букву.
12. Написать процедуру, реализующую вставку элемента в B-дерево.
13. Написать процедуру, реализующую удаление элемента из B-дерева.
14. Написать процедуру, реализующую вставку элемента в (2-3)-дерево.
15. Написать процедуру, реализующую удаление элемента из (2-3)-дерева.
16. Написать процедуру, реализующую вставку элемента в B+-дерево.
17. Написать процедуру, реализующую удаление элемента из B+-дерева.

Приложение 1



Модуль *CRT*. Работа с текстом

Модуль `CRT` представляет собой библиотеку функций и процедур, предназначенных для увеличения возможностей текстового ввода-вывода данных. Все описание возможностей данного модуля представлено в табл. П1.1. Для задания одного из текстовых режимов предназначена процедура

```
PROCEDURE TEXTMODE (MODE: WORD);
```

где `Mode` — код текстового режима, который может принимать следующие значения:

```
BW40      = 0  - черно-белый режим 40 x 25;
C040      = 1  - цветной режим 40 x 25;
BW80      = 2  - черно-белый режим 80 x 25;
C080      = 3  - цветной режим 80 x 25;
MONO      = 7  - монохромный для черно-белого адаптера;
FONT8X8   = 256 - используются для загружаемого шрифта
            в режиме 80 x 45 или 80 x 50 с адаптерами VGA или EGA.
```

Какой бы режим ни был установлен, координаты верхнего левого угла экрана всегда определяются как X1 = 1 и Y1 = 1. Приращение значений по оси X происходит слева направо, а по оси Y — сверху вниз. Значение координат нижнего правого угла зависит от режима: либо X2 = 40 и Y2 = 25, либо X2 = 80 и Y2 = 25, либо X2 = 80 и Y2 = 45, либо X2 = 80 и Y2 = 50.

Таблица П1.1. *Функции и процедуры модуля CRT*

Функция/процедура	Назначение и прототип функции/процедуры
`KeyPressed`	Определяет, была ли нажата клавиша на клавиатуре `Function KeyPressed: boolean;`
`ReadKey`	Читает значение нажатой клавиши `Function ReadKey: char;`
`TextBackground`	Устанавливает цвет фона `Procedure TextBackground(Color: byte);`

Таблица П1.1 *(окончание)*

Функция/процедура	Назначение и прототип функции/процедуры
`TextColor`	Устанавливает цвет выводимых символов `Procedure TextColor (Color: byte);`
`TextMode`	Устанавливает текстовый режим `Procedure TextMode (Mode: Word);`
`CirScr`	Очищает экран и устанавливает курсор в левый верхний угол экрана `Procedure ClrScr;`
`Window`	Определяет текстовое окно на экране `Procedure Window (XI, Yl. X2, Y2 : Byte);`
`WhereX`	Возвращает значение абсциссы `Function WhereX: Byte;`
`WhereY`	Возвращает значение ординаты `Function WhereY: Byte;`
`GotoXY`	Переводит курсор в указанное место `Procedure GotoXY(X. Y : Byte);`
`DelLine`	Уничтожает всю строку с курсором `Procedure DelLine;`
`InsLine`	Вставляет строку `Procedure InsLine;`
`ClrEol`	Стирает часть строки от курсора до правой границы `Procedure ClrEol;`
`HighVideo`	Устанавливает повышенную яркость символов `Procedure HighVideo;`
`NormVideo`	Устанавливает нормальную яркость символов `Procedure NormVideo;`
`LowVideo`	Устанавливает пониженную яркость символов `Procedure LowVideo;`
`AssignCrt`	Связывает текстовый файл с окном CRT `Procedure AssignCrt(F: Text);`
`Delay`	Вводит задержку в миллисекундах `Procedure Delay(T: Word);`
`Sound`	Заставляет динамик звучать с нужной частотой `Procedure Sound(F: Word);`
`NoSound`	Отключает динамик `Procedure NoSound;`

Для того чтобы стали доступны указанные в табл. П1.1 функции и процедуры, необходимо явно указать модуль `CRT` командой

```
uses Crt;
```

С помощью процедуры

```
TextBackGround(Color: byte);
```

можно устанавливать различные цвета экрана или текстового окна. Допустимые значения переменной `Color` колеблются от 0 (черный цвет) до 7 (белый цвет).

Для установления цветов символов служит процедура

```
TextColor (Color: byte);
```

где значение цвета изменяется от 0 до 15, а мерцание символов устанавливается значением 128.

В модуле `CRT` можно устанавливать цвета фона и цвета символов с помощью мнемонических констант (табл. П1.2).

Таблица П1.2. *Константы цвета модуля CRT*

Наименование константы	Значение константы	Цвет/функция
BLACK	0	Черный
BLUE	1	Темно-синий
GREEN	2	Темно-зеленый
CYAN	3	Бирюзовый
RED	4	Красный
MAGENTA	5	Фиолетовый
BROWN	6	Коричневый
LIGHTGRAY	7	Светло-серый
DARKGRAY	8	Темно-серый
LIGHTBLUE	9	Голубой
LIGHTGREEN	10	Салатный
LIGHTCYAN	11	Светло-бирюзовый
LIGHTRED	12	Розовый
LIGHTMAGENTA	13	Малиновый
YELLOW	14	Желтый
WHITE	15	Белый
BLINK	128	Мерцание символа

Процедура TextBackGround устанавливает цвет, а ClrScr очищает экран или текстовое окно и закрашивает его требуемым цветом. Курсор при этом перемещается в левый верхний угол. По умолчанию цвет экрана черный.

Например, для того, чтобы последовательно закрашивать экран во все цвета палитры, нужно организовать следующий цикл:

```
FOR i:=O TO 15 DO
BEGIN
  TEXTBACKGROUND(I);  CLRSCR
END;
```

Если вы хотите установить текстовое окно и закрашивать его в различные цвета, то это можно сделать, обратившись к процедуре WINDOW:

```
WINDOW(X1, Y1, X2, Y2);
FOR I:=1 TO 15 DO
BEGIN
  TEXTBACKGROUND(I);  CLRSCR
END;
```

где X1, Y1 — координаты верхнего левого угла окна, X2, Y2 — нижнего правого. Сразу после вызова процедуры WINDOW курсор помещается в левый верхний угол окна, а само окно заполняется цветом фона. Обратите внимание, что обращение к процедуре WINDOW игнорируется, если какая-либо из координат выходит за границу экрана или нарушается условие (X2 > X1) AND (Y2 > Y1).

В модуле CRT дополнительные возможности по управлению клавиатурой реализуются функциями READKEY и KEYPRESSED.

Функция READKEY возвращает значение типа CHAR, которое извлекается из буфера клавиатуры в виде кода символа. Если буфер пуст, то функция будет ждать нажатия на любую клавишу. Обратите внимание, что эта функция не отображает вводимые символы на экране.

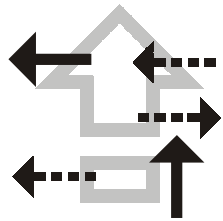
При использовании READKEY в буфер помещается расширенный код, т. е. для любой алфавитно-цифровой клавиши он соответствует коду вводимого символа, а при использовании функциональных клавиш <F1>, <F2>, <F10> и <Ins> генерируется двухбайтовая последовательность из #0 и кода клавиши. Кроме того, функция игнорирует нажатие <Shift>, <Ctrl>, <Alt>, <CapsLock>, <NumLock>, <ScrollLock>, <F11> и <F12>, воспринимая их только в комбинации с чем-нибудь еще.

Функция KEYPRESSED возвращает значение TRUE, если буфер клавиатуры не пуст, и FALSE — в противном случае. Например, для вывода всего буфера (до 16 символов) на экран можно использовать следующую программу:

```
PROGRAM ONE_10;
```

```
  USES CRT;
  VAR RR: STRING;
  BEGIN
    RR:='';
    WHILE KEYPRESSED DO RR:=RR+READKEY;
    WRITE('СОДЕРЖИМОЕ БУФЕРА КЛАВИАТУРЫ: ');
    WRITELN(RR)
END.
```

Приложение 2



Модуль *Graph*. Графика

Библиотека графических подпрограмм GRAPH (табл. П2.1), содержащая более 50 разнообразных процедур и функций, расширяет возможности PASCAL 7.0 по созданию изображений.

Таблица П2.1. *Функции и процедуры модуля GRAPH*

Функция/процедура	Назначение и прототип функции/процедуры
Arc	Построение дуги окружности procedure Arc (X, Y: Integer; StAngle, EndAngle, Radius: Word);
Bar	Построение закрашенного прямоугольника procedure Bar(xl, yl, x2, y2: Integer);
Bar3D	Построение закрашенного параллелепипеда procedure Bar3D(xl, yl, x2, y2: Integer; Depth: Word; Top: Boolean);
Circle	Построение окружности procedure Circle(X, Y: Integer; Radius: Word);
ClearDevice	Очистка экрана и заливка его цветом фона procedure ClearDevice;
ClearViewPort	Очистка окна и заливка его черным цветом procedure ClearViewPort;
CloseGraph	Завершение работы графического режима procedure CloseGraph;
DetectGraph	Возвращает тип драйвера и режим его работы procedure DetectGraph(var GraphDriver, GraphMode: Integer);
DrawPoly	Построение многоугольника procedure DrawPoly(NumPoints: Word; var PolyPoints);
Ellipse	Построение эллипса procedure Ellipse(X, Y: Integer; StAngle, EndAngle: Word; XRadius, YRadius: Word);

Таблица П2.1 *(продолжение)*

Функция/процедура	Назначение и прототип функции/процедуры
`FillEllipse`	Построение закрашенного эллипса procedure FillEllipse(X, Y : Integer; XRadius, YRadius: Word);
`FillPoly`	Построение закрашенного многоугольника procedure FillPoly(NumPoints: Word; var PolyPoints);
`FloodFill`	Заполнение замкнутой фигуры с использованием текущего цвета и узора procedure FloodFill(X, Y: Integer; Border: Word);
`GetArcCoords`	Возвращает координаты центра, начала и конца дуги procedure GetArcCoords(var ArcCoords: ArcCoordsType);
`GetAspectRatio`	Возвращает размер сторон экрана procedure GetAspectRatio(var Xasp, Yasp: Word);
`GetBkColor`	Возвращает цвет фона function GetBkColor: Word;
`GetColor`	Возвращает цвет линий и контуров function GetColor: Word;
`GetDefaultPalette`	Возвращает значение текущей палитры procedure GetDefaultPalette(var Palette: PaletteType);
`GetDriverName`	Возвращает имя текущего драйвера function GetDriverName: string;
`GetFillPattern`	Возвращает тип узора заполнения procedure GetFillPattern(var FillPattern: FillPatternType);
`GetFillSettings`	Возвращает тип заполнения procedure GetFillSettings(var FillInfo: FillSettingsType);
`GetGraphMode`	Возвращает номер графического режима function GetGraphMode: Integer;
`GetImage`	Сохраняет изображение в буфере procedure GetImage(x1, y1, x2, y2: Integer; var BitMap);
`GetLineSettings`	Возвращает параметры линии procedure GetLineSettings(var LineInfo: LineSettingsType);
`GetMaxColor`	Возвращает максимальный номер в палитре цветов function GetMaxColor: Word;
`GetMaxMode`	Возвращает количество возможных графических режимов function GetMaxMode: Integer;
`GetMaxX`	Возвращает максимальную координату X function GetMaxX: Integer;

Таблица П2.1 *(продолжение)*

Функция/процедура	Назначение и прототип функции/процедуры
`GetMaxY`	Возвращает максимальную координату Y function GetMaxY: Integer;
`GetModeName`	Возвращает имя заданного графического режима function GetModeName(ModeNumber: Integer): string;
`GetModeRange`	Возвращает минимальный и максимальный номера графических режимов procedure GetModeRange (GraphDriver : Integer; var LoMode, HiMode: Integer);
`GetPalette`	Возвращает цвета палитры procedure GetPalette(var Palette: PaletteType);
`GetPaletteSize`	Возвращает количество цветов в палитре function GetPaletteSize: Integer;
`GetPixel`	Возвращает цвет пиксела function GetPixel(X, Y: Integer): Word;
`GetTextSettings`	Возвращает параметры текста procedure GetTextSettings(var TextInfo: TextSettingsType);
`GelViewSettings`	Возвращает параметры текущего окна procedure GetViewSettings(var ViewPort: ViewPortType);
`GetX`	Возвращает координату X курсора function GetX: Integer;
`GetY`	Возвращает координату Y курсора function GetY: Integer;
`GraphDefaults`	Сброс параметров графического режима procedure GraphDefaults;
`ImageSize`	Задание требуемой для изображения памяти function ImageSize(x1, y1, x2, y2: Integer): Word;
`InstallUserDriver`	Установка графического драйвера пользователя function InstallUserDriver(Name: string; AutoDetectPtr: pointer): Integer;
`InstallUserFont`	Установка нового шрифта пользователя function InstallUserFont(FontFileName: string): Integer;
`Line`	Рисует линию procedure Line(x1, y1, x2, y2: Integer);
`LineRel`	Рисует линию в относительных координатах procedure LineRel(Dx, Dy: Integer);
`LineTo`	Рисует линию к указанной точке procedure LineTo(X, Y: Integer);

Таблица П2.1 *(продолжение)*

Функция/процедура	Назначение и прототип функции/процедуры
`MoveRel`	Перемещает курсор в точку с относительными координатами procedure MoveRel(Dx, Dy: Integer);
`MoveTo`	Перемещение курсора в указанную точку procedure MoveTo(X, Y: Integer);
`OutText`	Выводит текст procedure OutText(TextString: string);
`OutTextXY`	Выводит текст, начиная с указанной точки procedure OutTextXY(X, Y: Integer; TextString: string);
`PieSlice`	Строит и закрашивает часть круга procedure PieSlice(X, Y: Integer; StAngle, EndAngle, Radius: Word);
`PutImage`	Помещает изображение из буфера на экран procedure PutImage(X. Y: Integer; var BitMap; BitBlt: Word);
`PutPixel`	Рисует точку procedure PutPixel(X, Y: Integer; Pixel: Word);
`Rectangle`	Рисует прямоугольник procedure Rectangle(x1, y1, x2, y2: Integer);
`RegisterBGIdriver`	Регистрация драйвера function RegisterBGIdriver(driver: pointer): Integer;
`RegisterBGIfont`	Регистрация шрифта function RegisterBGIfont(Font: pointer): Integer;
`RestoreCrtMode`	Возвращение в текстовый режим procedure RestoreCrtMode;
`Sector`	Рисует и закрашивает сектор procedure Sector(x, y: Integer; StAngle, EndAngle, XRadius, YRadius: Word);
`SetActivePage`	Задает активную страницу procedure SetActivePage(Page: Word);
`SetAllPalette`	Задает палитру procedure SetAllPalette(var Palette);
`SetAspectRatio`	Задает соотношение между шириной и высотой экрана function SetAspectRatio(Xasp, Yasp: Word): Word;
`SetBkColor`	Задает цвет фона procedure SetBkColor(ColorNum: Word);
`SetColor`	Задает цвет линий, точек и т. д. procedure SetColor(Color: Word);

Таблица П2.1 *(продолжение)*

Функция/процедура	Назначение и прототип функции/процедуры
`SetFillPattern`	Закрашивает произвольную замкнутую фигуру procedure SetFillPattern(Pattern: FillPatternType; Color: Word);
`SetFillStyle`	Устанавливает стиль заполнения procedure SetFillStyle(Pattern: Word; Color: Word);
`SetGraphBufSize`	Устанавливает размер буфера procedure SetGraphBufSize(BufSize: Word);
`SetGraphMode`	Устанавливает тип графического режима procedure SetGraphMode(Mode: Integer);
`SetLineStyle`	Устанавливает стиль линии procedure SetLineStyle(LineStyle: Word; Pattern: Word; Thick-ness: Word);
`SetPalette`	Устанавливает один цвет палитры procedure SetPalette(ColorNum: Word; Color: ShortInt);
`SetRGBPalette`	Устанавливает палитру для VGA procedure SetRGBPalette(ColorNum, RedValue, GreenValue, BlueValue: Integer);
`SetTextJustify`	Устанавливает стиль выравнивания текста procedure SetTextJustify(Horiz, Vert: Word);
`SetTextStyle`	Устанавливает стиль вывода текста procedure SelTextStyle(Font, Direction: Word; CharSize: Word);
`SetUserCharSize`	Устанавливает высоту и ширину символов procedure SetUserCharSize(MultX, DivX, MultY, DivY: Word);
`SetViewPort`	Устанавливает размеры окна procedure SetViewPort(x1, y1, x2, y2: Integer; Clip: Boolean);
`SetVisualPage`	Устанавливает параметры видимой страницы procedure SetVisualPage(Page: Word);
`SetWriteMode`	Задание способа рисования линии procedure SetWriteMode(WriteMode: Integer);
`TextHeight`	Возвращает высоту строки в пикселах function TextHeight(TextString: string): Word;
`TextWidth`	Возвращает ширину строки в пикселах function TextWidth(TextString: siring): Word;

Модуль `GRAPH` требует установления графического режима. Для того чтобы это стало возможным, необходимо проделать следующие действия.

Во-первых, в программе должна быть ссылка на модуль

```
USES GRAPH;
```

Во-вторых, работу модуля нужно инициировать командой

```
INITGRAPH(DRIVER, MODE, 'C:\BP\BGI');
```

где DRIVER — параметр установки типа графического драйвера видеоадаптера; MODE — задание режима его работы; 'C:\BP\BGI' — строка, указывающая на путь к директории, где расположены графические драйверы *.bgi. Если в той же директории, где находится ваша программа, располагается и требуемый BGI-драйвер, то эта строка должна быть пуста:

```
INITGRAPH(DRIVER, MODE, '');
```

В-третьих, необходимо настроить среду PASCAL. Для этого в меню OPTIONS\DIRECTORIES среды в поле UNIT необходимо указать каталог, где размещен файл Graph.tpu.

Если вы затрудняетесь указать режим работы вашего видеоадаптера, то позвольте системе определить это самой с помощью функции DETECT.

```
PROGRAM PROBA;
USES GRAPH;
VAR DRIVER, MODE: INTEGER;
BEGIN
DRIVER := DETECT;
INITGRAPHDRIVER, MODE, 'C:\BP\BGI');
...
```

Перед выходом из программы графический режим нужно закрыть командой CLOSEGRAPH.

Список литературы

1. Бентли Дж. Жемчужины программирования, 2-е изд. — СПб.: Питер, 2002. — 272 с.: ил.
2. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. / Пер. с англ. — М.: Бином, СПб.: Невский диалект, 1998.
3. Вирт Н. Алгоритмы и структуры данных. — СПб.: Невский диалект, 2001. — 352 с.
4. Дмитриева М. В., Кубенский А. А. Турбо Паскаль и Турбо Си: построение и обработка структур данных: Учебное пособие. — СПб.: Изд-во С.-Петербургского университета, 1995. — 245 с.
5. Касьянов В. Н., Сабельфельд В. К. Сборник заданий по практикуму на ЭВМ. — М.: Наука, 1986. — 272 с.
6. Кнут Д. Искусство программирования. Т. 3. Сортировка и поиск: Пер. с англ. — М.: Вильямс, 2000. — 822 с.
7. Мейер Д., Бодуэн К. Методы программирования. Т. 1, 2. — М.: Мир, 1985.
8. Программирование на языке паскаль: задачник. / Ускова О. Ф., Бакланов М. В., Горбенко О. Д., Воронина И. Е., Вощинская Г. Э., Огаркова Н. В., Мельников В. М. — СПб.: Питер, 2002. — 336 с.: ил.
9. Пешио К. Н. Вирт о культуре разработки ПО. // Открытые системы, 1998. № 1(27), с. 40–44.
10. Сибуя М., Ямамото Т. Алгоритмы обработки данных: Пер. с япон. — М.: Мир, 1986.

Предметный указатель

М

Н

О

П

Р

С

Т

У

Ф

Х

Э