Е.В. СИМОНОВА

СТРУКТУРЫ ДАННЫХ

Часть II

НЕЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

2007

СГАУ

САМАРА

УДК 004.9(075)
ББК 32.97
С 375

Инновационная образовательная программа "Развитие центра компетенции и подготовка специалистов мирового уровня в области аэрокосмических и геоинформационных технологий"

Рецензенты: д-р техн. наук, проф. Н.А. Ковардцев,
д-р техн. наук, проф. С. В. Смирнов

Симонова Е.В.

С 375 **Структуры данных. Часть II. Нелинейные динамические структуры**: учеб. пособие / *Е.В.Симонова.* – Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2007. – 81 с. : ил.

ISBN 978-5-7883-0523-3

Учебное пособие включает разделы, которые подробно описывают методы совмещения типов, рекурсивные алгоритмы обработки структур данных, иерархические структуры данных (деревья и графы). Теоретический материал иллюстрируется большим количеством программных фрагментов, реализующих алгоритмы обработки различных структур данных. Учебное пособие содержит контрольные вопросы и упражнения по всем разделам.

Учебное пособие предназначено для студентов, обучающихся по направлениям 010500 - "Прикладная математика и информатика", 010600 – "Прикладные математика и физика", 010400 – "Информационные технологии".

УДК 004.9 (075)
ББК 32.97

ISBN 978-5-7883-0523-3

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В учебном пособии описаны структуры данных и алгоритмы, которые широко используются при решении разнообразных задач в широком спектре предметных областей.

Учебное пособие разработано в соответствии с федеральным компонентом ГОС подготовки бакалавров по направлениям 010500 – Прикладная математика и информатика, 010600 – Прикладные математика и физика, 010400 – Информационные технологии, 010200 – Автоматизированные системы обработки информации и управления. Учебное пособие предназначено для студентов, обучающихся по данным направлениям.

Рекомендуется использовать учебное пособие при изучении следующих дисцилин:

- “Языки программирования и методы трансляции”, “Практикум на ЭВМ (языки программирования)” для направления 010500;
- “Информатика”, “Прикладная информатика” для направления 010600;
- “Основы программирования”, “Языки программирования”, “Практикум на ЭВМ” для направления 010400;
- “Программирование на языке высокого уровня”, “Информационные технологии” для направления 010200.

Содержание учебного пособия соответствует разделам рабочих программ по дисциплинам, перечисленным выше.

В главе 1 представлены сведения о методах совмещения типов, позволяющих изменять вид интерпретации данных, находящихся в оперативной памяти.

Глава 2 посвящена рассмотрению рекурсии как метода организации множества объектов и вычислительных процессов. Подробно описывается структура рекурсивного вычислительного процесса, особенности его реализации. Приводится большое количество примеров рекурсивных алгоритмов, нашедших широкое применение при решении задач различной природы.

В главе 3 описываются рекурсивные иерархические древовидные структуры. Рассматриваются алгоритмы обработки наиболее распространенных древовидных структур, таких, как сбалансированные деревья, дихотомические деревья, деревья выражений.

В главе 4 подробно рассматриваются особенности организации структур графов. Особое внимание уделено рассмотрению алгоритмов решения распространных в практике задач обхода графов и нахождения кратчайших путей в графе.

Программы, реализующие алгоритмы обработки структур данных различных типов, соответствуют современному уровню информационных технологий. Разделы, рассмотренные в пособии, имеют большое учебно-методическое значение и необходимы при самостоятельной работе студентов

во время выполнения ими домашних заданий, лабораторного практикума и курсовой работы.

В конце каждой главы приведены упражнения и контрольные вопросы, с помощью которых можно проверить усвоение изложенного материала.

В учебном пособии содержатся сведения, недостаточно освещенные в учебной литературе, а также реализуются новые методики преподавания, активизирующие самостоятельную работу студентов.

Учебное пособие может быть полезно широкому кругу читателей, практикующихся в области компьютерных наук.

ВВЕДЕНИЕ

В [10] представлены разнообразные структуры данных, позволяющие моделировать линейные отношения на множестве объектов единой природы. Объекты при этом рассматриваются как принадлежащие одному и тому же абстрактному типу. Однако, на практике часто возникают задачи, для эффективного решения которых требуется применять различные методы интерпретации данных в динамике работы программы. Для этого используются методы совмещения типов данных.

Для описания сложных отношений между объектами используются иерархические нелинейные структуры данных.

Деревья представляют собой иерархическую структуру некоторого множества объектов. Древовидные структуры широко используются для организации хранения информации в системах управления базами данных и для представления синтаксических структур в компиляторах программ. Специальные методы представления множеств основаны на использовании структур деревьев двоичного поиска, нагруженных и сбалансированных деревьев.

Во многих задачах, возникающих в технических дисциплинах, в математике и в компьютерных науках часто требуется наглядно представить отношения между какими-либо объектами. Наиболее адекватной моделью для представления таких отношений являются ориентированные и неориентированные графы. В пособии рассматриваются основные структуры данных, которые применяются для представления ориентированных и неориентированных графов, а также описываются основные алгоримы определения связности ориентированных графов, построения минимальных остовных деревьев и нахождения кратчайших путей в графе.

Две части пособия, посвященные рассмотрению линейных и нелинейных структур данных, охватывают широкий спектр представления объектов различной природы, используемых при решении разнообразных практических задач.

1. МНОЖЕСТВЕННАЯ ИНТЕРПРЕТАЦИЯ ОБЪЕКТОВ

1.1. Совместимость типов. Приведение и преобразование типов

Языки программирования со строгой типизацией построены на основе соблюдения концепции типов, согласно которой все операции, определенные в языке, могут применяться только к операндам совместимых типов. Два типа считаются ***совместимыми***, если:

- оба они есть один и тот же тип,
- оба вещественные,
- оба целые,
- оба логические,
- один тип есть тип-диапазон второго типа,
- оба являются типами-диапазонами одного и того же базового типа,
- оба являются множественными типами, составленными из элементов одного и того же базового типа,
- один тип – есть тип-строка, а другой – тип-строка или символ,
- один тип есть любой указатель, а другой – нетипизированный указатель,
- оба есть процедурные типы с одинаковым типом результата (для функций), количеством параметров и типом взаимно соответствующих параметров.

Два объекта являются ***совместимыми по представлению***, если размеры их элементов хранения равны. Значение некоторого типа и тип переменной, которой может быть присвоено это значение, если совместимость разрешается, называются ***совместимыми по присваиванию***. Совместимость по присваиванию необходима, если имеет место присваивание значения, например, в операторе присваивания или при передаче значений параметров. Совместимость по присваиванию реализуется через приведение и преобразование типов.

Приведение типа возможно только для объектов, совместимых по представлению. Приведение типа состоит в следующем: определяя тип объекта, мы определяем представление (структуру) элемента хранения объекта данного типа. Но если “взглянуть” на образ объекта в памяти с точки зрения машинного представления другого типа, то можно трактовать тот же самый элемент хранения как принадлежащий другому типу. Для этого используются функции приведения типов:

Имя_типа (Имя_переменной)
Имя_типа (Выражение).

Функции приведения типа определены как для стандартных, так и для пользовательских типов. Функции приведения типов могут использоваться как в левой, так и в правой части оператора присваивания, т.к. приведение

типа не изменяет внутреннего представления объекта, а изменяет только его интерпретацию:

```
Type
  Rec = array [1…4] of  byte;
Var A: Rec; L: longint; b: byte; w: word;
begin
  A [1]: = …;  A [2]: =…;  A [3]: =…;  A[4]: =…;          { инициализация массива }
  L: = LongInt ( A );      { приведение типа Rec к типу LongInt в правой части оператора
                                                                          присваивания }
  L: = 123456;                                          { инициализация переменной L }
  b: = Rec ( L ) [3];      { приведение типа LongInt к типу Rec  в правой части оператора
                                                                          присваивания }
  Rec ( L ) [1]: = 10;     { приведение типа LongInt к типу Rec  в левой части оператора
                                                                          присваивания }
  . . .
  w:=word( -6 );           { приведение типа Integer к типу Word: w = 65530 ( см. 1.1.1 ) }
  . . .
```

Преобразование типа изменяет внутреннее представление объектов, т.е. их элементы хранения. В результате преобразования типа длина внутреннего представления объекта может как увеличиться, так и уменьшиться. При выполнении преобразования к типу большей мощности, у которого размер элемента хранения больше, значения атрибутов объекта будут целиком записаны в младшие байты. В ряде случаев при выполнении преобразования к типу меньшей мощности, например, если “длинный” целый тип преобразуется к “короткому”, от элемента хранения объекта берутся только младшие байты.

Преобразование типа может быть явным, неявным и автоопределенным. ***Неявное преобразование*** возможно в выражениях, составленных из вещественных и целочисленных переменных, переменные типа *INTEGER* автоматически преобразуются к типу *REAL*, и все выражение в целом приобретает вещественный тип. ***Явные преобразования*** связаны с использованием специальных функций преобразования, определенных в языке, аргументы которых принадлежат одному типу, а значения – другому:

```
function Chr( X: byte ): char;                  { возвращает символ с заданным
                                                          порядковым номером X }
function Ord( X: любой порядковый               { возвращает порядковый номер, соответ-
              тип): longint;                    ствующий значению X порядкового типа }
function Round( R: real ): longint;             { округляет значение R вещественного типа
                                                          до ближайшего целого }
function Trunc( R: real ): longint;             { усекает значение вещественного типа
                                                          путем отбрасывания дробной части }
```

Автоопределенное преобразование связано с использованием только в правой части оператора присваивания функций преобразования вида:

Имя_типа (Имя_переменной)

Имя_типа (Выражение).

Функции преобразования типа определены как для стандартных, так и для пользовательских типов:

```
Var L: longint; W: word;
begin
  writeln( integer ( 'd' ) );          { преобразование типа Char к типу Integer }
  writeln( byte ( 534 ) );             { преобразование типа Word к типу Byte }
  . . .
  W:=word ( L );                       { преобразование типа LongInt к типу Word }
  . . .
```

1.2. Методы совмещения типов

Различные методы совмещения типов связаны с маскированием. ***Маскирование*** позволяет изменить интерпретацию данных, содержащихся в элементах хранения объектов. ***Маска*** – это некоторый ***тип данных***, определяющий вид интерпретации. Накладывая различные маски на одну и ту же область памяти, можно рассматривать размещенные в ней данные как принадлежащие объектам различных типов. Маскирование может быть реализовано с помощью средств, встроенных в язык программирования, или с помощью указателей. Средства языка *PASCAL*, реализующие совмещение типов:

- записи с вариантной частью;
- директива *ABSOLUTE*;
- параметры процедур и функций без типа;
- открытые массивы.

1.2.1. Запись с вариантной частью

Запись с вариантной частью используется для работы с объектами различной структуры как с переменными одного и того же типа. Запись с вариантной частью состоит из одного или нескольких фиксированных полей (полей фиксированного размера) и единственной вариантной части, которая должна располагаться в конце записи:

```
Type
  Rec =record
      f1: integer;
      case  f2: boolean  of
            false: (f3, f4, f5: char);
            true: (f6: word)
  end;
Var r1, r2: rec;
```

Вариантная часть задается предложением вида:

Case селектор_вариантов of ...

В конце вариантной части не следует ставить *END* как пару к *CASE...OF*. (Поскольку вариантная часть - всегда последняя в записи, за ней все же стоит *END*, но лишь как пара к *RECORD*). Селектор вариантов может быть именованным или неименованным, должен иметь любой порядковый тип (стандартный или пользовательский). Вариантная часть состоит из нескольких вариантов. Каждый вариант определяется константой выбора, за которой следует двоеточие и список полей данного варианта, заключенный в круглые скобки. Селектор варианта может принимать значение любой константы выбора.

Особенностью вариантной части является то, что все заданные в ней варианты "накладываются" друг на друга и используют одну и ту же область памяти (рис. 1). В элементе хранения конкретного объекта типа записи с вариантами всегда присутствует единственный вариант.

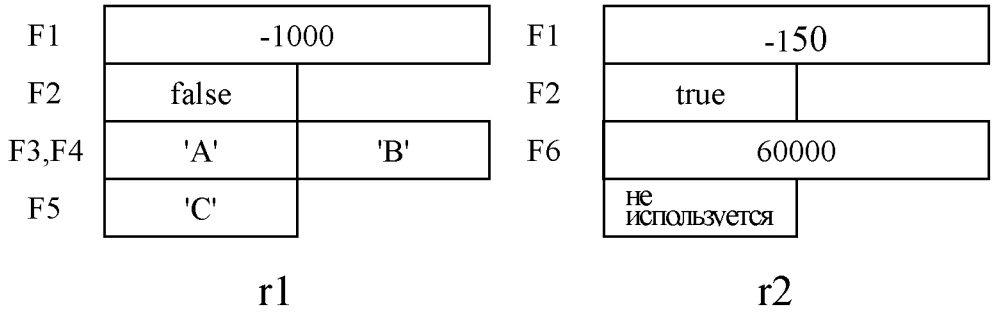


Рис. 1. Элементы хранения объектов, использующих запись с вариантной частью

Размер элемента хранения объектов, содержащих запись с вариантами, определяется размером самого длинного варианта:

$$Sizeof(Rec) = Sizeof(f1) + Sizeof(f2) + max\{Sizeof(f3) + Sizeof(f4) + Sizeof(f5),\ Sizeof(f6)\} = 6.$$

Таким образом, компилятор не отводит отдельную область памяти для каждого варианта, а выделяет участок памяти, достаточный для размещения варианта самого большого размера.

При использовании именованного селектора в целях дополнительного контроля следует присваивать значение селектору перед тем, как присвоить значения полям варианта, а также проверять значение селектора перед обращением к полям варианта.

Записи с вариантами применяются для описания узлов разнородных списков. Рассмотрим разнородный список, содержащий информацию о геометрических фигурах, составляющих некоторый проект. Элемент хранения узла разнородного списка содержит фиксированные поля, которые располагаются в начале записи: связь в списке, координаты точки, относительно которой строится фигура. Далее располагается поле селектора

вариантов и поля вариантов, соответствующие трем типам геометрических фигур:

```
Type
 Figure = (circle, rectangle, triangle);                    { тип фигуры }
 PPolygon = ^ Polygon;        { указатель на эл-т хранения узла разнородного списка }
 Polygon = record                    { эл-т хранения узла разнородного списка }
  link: PPolygon;                                                { связь в списке }
  x,y: word;          { координаты точки, относительно которой строится фигура }
  case kind: Figure of          { селектор варианта, определяющий тип фигуры }
   circle: ( radius: word );                                     { окружность }
   rectangle: ( height, width: word );                        { прямоугольник }
   triangle: ( x1,y1,x2,y2: word )                             { треугольник }
end;

Var
  f: PPolygon;                                  { указатель на первый узел списка }
  Poly: Polygon;                                                  { фигура }

Procedure Input_Polygon( var z: Polygon; t: Figure );          { t – тип фигуры }
begin                                  { ввод значения информационного поля - фигуры }
  with z do begin
   writeln( 'введите координаты точки, относительно которой строится фигура' );
   write( 'x= ' ); readln( x );   write( 'y= ' ); readln( y );
   kind:=t;                                        { заполнение поля селектора вариантов }
   case t of          { заполнение полей вариантов в зависимости от типа фигуры: }
     circle:                                                       { окружность }
     begin write( 'введите значение радиуса = ' ); readln( radius ) end;
     rectangle:                                                 { прямоугольник }
       begin
        write( 'введите значение высоты = ' ); readln( height );
        write( 'введите значение ширины = ' ); readln( width )
       end;
     triangle:                                                   { треугольник }
       begin
        writeln( 'введите координаты второй вершины' );
         write( 'x= ' ); readln( x1 );
         write( 'y= ' ); readln( y1 );
         writeln( 'введите координаты третьей вершины' );
         write( 'x= ' ); readln( x2 );
         write( 'y= ' ); readln( y2 );
       end;
   end;                                                              { case }
  end                                                                { with }
end;
{ создание элемента хранения узла и занесение его в разнородный список }
```

```
Procedure Create_Node( var first: PPolygon; z: Polygon );
Var p: PPolygon;                                  { first – указатель на первый узел списка }
begin
  new( p );   p^:=z;          { создание элемента списка и заполнение информац. поля }
  p^.link:=first;  first:=p;                  { включение элемента хранения узла в список }
end;

Procedure Print( first: PPolygon );      { просмотр содержимого разнородного списка }
Var p: PPolygon;
begin
 p:=first;
 while p <> nil do begin
 writeln( 'координаты точки, относительно которой построена фигура' );
 writeln( 'x= ',  p^.x,  'y= ',  p^.y );
 case  p^.kind  of                        { проверка значения поля селектора вариантов: }
  circle:                                                                  { окружность }
    writeln( 'радиус окружности = ', p^.radius );
  rectangle:                                                          { прямоуглольник }
    writeln( 'высота прям-ка = ' , p^.height, 'ширина прям-ка = ', p^.width );
  triangle:                                                             { треугольник }
  begin
    writeln( 'координаты второй вершины треугольника' );
    writeln( 'x= ',  p^.x1,  'y= ',  p^.y1 );
    writeln( 'координаты третьей вершины треугольника' );
    writeln( 'x= ',  p^.x2,  'y= ',  p^.y2 );
  end;
 end;
 p:=p^.link
end;

Procedure Destroy( var first: PPolygon );          { разрушение разнородного списка }
var p: Ppolygon;
begin
  while (first<>nil) then
   begin
     p:=first; first:=first^.link; dispose( p )
   end
end;

begin     f:=nil;                                          { первоначально список пуст }
 Input_Polygon( Poly, rectangle ); Create_Node( f, Poly );
 Input_Polygon( Poly, triangle );  Create_Node( f, Poly );
 Input_Polygon( Poly, circle );     Create_Node( f, Poly );
 Print( f );  . . . Destroy( f );
end.
```

Неименованный селектор вариантов используется для приведения типов (память для него в элементе хранения объекта не отводится):

```
Type
  Mem = record
     case byte of
        0: ( byt: array [1..4] of  byte  );
        1: ( wo: array [1..2] of  word  );
        2: ( l: longint )
  end;
Var m: Mem; w: word; b: byte;
begin
  m.l:=123456;          { инициализация переменной типа Mem значением типа LongInt }
  b:=m.byt[4];                                  { преобразование типа Mem к типу Byte }
  w:=m.wo[1];                                   { преобразование типа Mem к типу Word }
  . . .
```

Sizeof (Mem) = 4 .

Объект *m* имеет три варианта, каждый из которых может быть размещен в одном и том же элементе хранения размером 4 байта. В зависимости от того, к какому полю записи происходит обращение в программе, элемент хранения объекта *m* трактуется как массив из четырех байтов, массив из двух слов или длинное целое число.

1.2.2. Использование директивы *ABSOLUTE*

Совмещение данных в памяти может быть достигнуто при явном размещении данных разного типа по одному и тому же ***абсолютному адресу***. Для размещения объекта по нужному абсолютному адресу после идентификатора и типа объекта необходимо указать стандартную директиву ***ABSOLUTE***, за которой следует либо абсолютный адрес, либо идентификатор ранее определенного объекта. Абсолютный адрес задается парой чисел типа *WORD* (как правило, в шестнадцатиричном формате), разделенных двоеточием, причем первое число трактуется как сегмент, а второе – как смещение адреса:

```
Var b: byte absolute $0000 : $0100;
```

Тип объекта, описанного с директивой *absolute*, является маской и определяет вид интерпретации данных, находящихся по заданному адресу. При определении переменной с директивой *absolute* фактически происходит наложение маски на область памяти по заданному адресу. Обращение к этой переменной в программе позволяет соответствующим образом интерпретировать данные, находящиеся по этому адресу.

Если за директивой *absolute* указан идентификатор ранее определенного объекта, то в памяти происходит совмещение данных различных типов, причем младшие байты внутреннего представления этих данных будут располагаться по одному и тому же абсолютному адресу:

```
Type MI = array [1..3] of integer;                    { маска }
Var x: real;                          { интерпретируемая область памяти }
    y: MI  absolute x;                                { наложение маски }
begin
 x:=…;                                    { инициализация переменной x }
 writeln( y[2] );      { интерпретация данных, хранящихся в переменной x,
                                                          через маску }
```

Если происходит совмещение объектов, элементы хранения которых имеют разную длину, следует размещать меньший объект по адресу большего, а не наоборот (рис. 2):

```
Var st: string [20];                  { размещение объекта меньшего размера }
    stlen: byte absolute st;            { по адресу объекта большего размера }
```

Рис. 2. Совмещение объектов с помощью директивы *ABSOLUTE*

```
Var stlen: byte; w: word;      { ошибка: размещение объекта большего размера }
    st: string [20] absolute stlen;          { по адресу объекта меньшего размера }
```

При размещении объекта большего размера по адресу объекта меньшего размера запись в переменную *st* может испортить данные, располагающиеся за переменной *stlen*.

Рассмотрим пример совмещения данных с использованием директивы *ABSOLUTE*. Образ текстового экрана (25 строк на 80 столбцов) в области оперативной памяти, называемой видеопамятью, начинается с адреса $B800 : $0000 и занимает 4000 байтов. Структура видеопамяти проста: видимые строки экрана хранятся последовательно. Один символ на экране занимает два байта в памяти: первый байт – ASCII код символа, второй байт – атрибут символа, задающий цвет символа, цвет фона и признак мерцания символа. Поэтому все четные адреса видеопамяти, начиная с 0, содержат символы, а нечетные – атрибуты. На образ текстового экрана в видеопамяти можно наложить маску - структуру двумерного массива, который рассчитан на 25 * 80 = 2000 элементов, включающих символы, находящиеся на экране, и их атрибуты. Использование такой маски облегчает работу с видеопамятью, т.к. позволяет получить доступ к отдельному символу и его атрибутам.

Ниже приведен пример установки значения текстового атрибута в видеопамяти для символа с координатами (*x,y*). Заметим, что координата *x* фактически задает номер столбца, а координата *y* – номер строки в двумерном массиве – маске:

```
Type
  VideoWord= record                    { образ одного символа в видеопамяти: }
    symbol: char;                                  { ASCII код символа }
    attr: byte                                     { атрибут символа }
   end;

{ маска – образ текстового экрана в видеопамяти  - тип данных }
 TextScreen = array [1..25, 1..80] of  VideoWord;

{ наложение маски по абсолютному адресу в видеопамяти }
Var T: TextScreen absolute $B800 : $0000;

{ интерпретация данных, хранящихся по абсолютному адресу, через маску }
Procedure Change_Attr( x,y: byte;          { x,y – координаты символа на экране }
                       textattr: byte);          { textattr – атрибут символа }
begin
  T[y,x].attr:=textattr          { установка атрибута символа с использованием маски }
end;
```

1.2.3. Параметры без типа

Формальный параметр без типа внутри процедуры или функции не имеет типа и его перед использованием следует преобразовать к конкретному типу с помощью *автоопределенного преобразования*. Параметр без типа передается только по ссылке. Внутри процедуры или функции должна быть определена маска, задающая вид интерпретации элементов фактического параметра, передаваемого в процедуру или функцию при вызове. Напомним, что маска – это тип. Размерность маски рекомендуется выбирать так, чтобы она была рассчитана на максимальное количество данных интерпретируемого типа, которое может разместиться в пределах сегмента. Если размер маски превышает 65520 байт, на этапе компиляции выдается сообщение об ошибке. Тип фактического параметра может быть любым, но дополнительно в процедуру или функцию следует передавать истинную размерность фактического параметра.

Ниже приведен фрагмент программы суммирования N элементов одномерных числовых массивов произвольной размерности:

```
. . .
Var                                  { фактические параметры - }
  Ar_Byte1: array[1..200] of  byte;          { интерпретируемые }
```

```
  Ar_Byte2: array[1..2000] of byte;                    { области }
  Ar_Int: array[1..500] of integer;                    { памяти }

Function Sum( var X; n: word ):        { X – параметр без типа, n – размерность
longint;                                                          массива }
Type
  Xtype = array[1..65520] of     {Маска – одномерный массив типа Byte }
byte;
Var
  summa: longint; i: word;
begin
  summa:=0;
  for i:=1 to n do
    { XType( X )[i] – автоопределенное преобразование элемента массива-
                      параметра X к типу элемента массива-маски Byte }
    summa:=summa + XType( X )[i];      { интерпретация элементов массива X
                                                              через маску }
  Sum:=summa;
end;

begin
  . . .
  writeln( Sum( Ar_Byte1, 200 ) );
  writeln( Sum( Ar_Int, 1000 ) );
  writeln( Sum( Ar_Byte2, 2000 ) );
  . . .
```

В качестве первого параметра функции *Sum* можно использовать массив любого типа и любой размерности. В теле функции выполняется автоопределенное преобразование элемента массива-параметра *X* к типу элемента массива-маски *Byte*. Если бы в теле функции была определена маска другого типа, например, *Word*, преобразование выполнялось бы к этому типу.

1.2.4. Открытые массивы

В качестве параметров-переменных процедуры или функции могут использоваться массивы и строки открытого типа, у которых не задаются размеры. В качестве фактического параметра можно использовать массив любого размера, но содержать он должен элементы того же типа, что и открытый массив. Истинный размер массива – фактического параметра определяется с помощью встроенной функции *HIGH*. Открытый массив задается как обычный массив, но без указания индексов и трактуется как маска. Индексирование элементов массива – фактического параметра начинается с нуля, а максимальный индекс равен значению функции *HIGH* минус единица:

```
. . .
var                                              { фактические параметры - }
  Ar1: array[0..99] of integer;                  {  интерпретируемые }
  Ar2: array[0..999] of integer;                 { области памяти }
  s: integer;

Procedure Sum( var X: array of integer;     { X – окрытый массив, его тип – }
                 var summa: longint );                    { маска }
Var i: word;
begin
  summa:=0;
  for i:=0 to high( X ) –1 do              { интерпретация элементов массива }
    summa:=summa + X[i];                                  { через маску }
end

begin
  . . .
  Sum( Ar1,s );  writeln( s );
  Sum( Ar2,s );  writeln( s );
  . . .
```

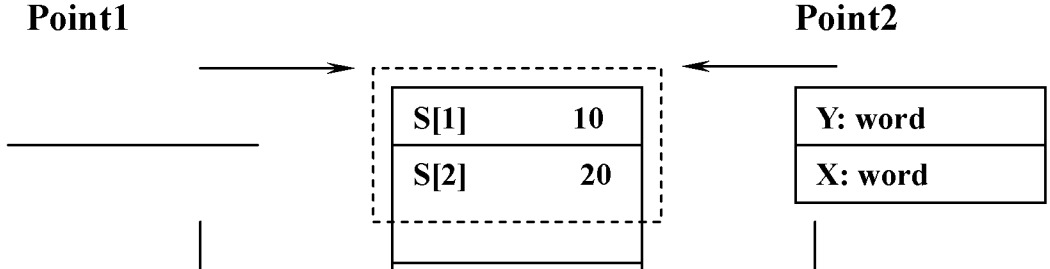
1.2.5. Наложение масок с помощью указателей

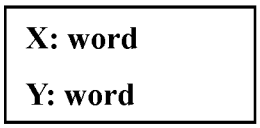
Наиболее универсальным методом совмещения типов является ***явное наложение масок с помощью указателей*** без использования специальных средств, встроенных в язык.

Алгоритм наложения маски с помощью указателя:

- определить тип маски;
- определить тип указателя, связанного с маской;
- определить переменную - типизированный указатель, связанный с маской;
- задать интерпретируемую область памяти;
- произвести наложение маски с использованием типизированного указателя. Наложение маски выполняется с помощью занесения адреса интерпретируемой области памяти в переменную-типизированный указатель, связанный с соответствующей маской. Через этот указатель открывается доступ к области памяти;
- выполнить интерпретацию области памяти через маску с использованием операции раскрытия ссылки.

Выполнение алгоритма наложения маски с помощью указателя представлено на рис. 3.





X: =10
Y: =20

X: = 20
Y: = 10

Рис. 3. Наложение маски с помощью указателя

```
Type
  Point1 = record                                          { маска 1 }
    x,y : word
  end;
  Pnt1 = ^ Point1;                        { тип указателя, связанного с маской 1 }

  Point2 = record                                          { маска 2 }
    y,x : word
  end;
  Pnt2 = ^ Point2;                        { тип указателя, связанного с маской 2 }

Var
  S: array[1..2] of word;                        { интерпретируемая область памяти }
  p1: Pnt1;  p2: Pnt2;      { типизированные указатели для наложения масок }
  x, y: word;

begin
  S[1]:=10;  s[2]:=20;       { инициализация интерпретируемой области памяти }
  p1:=@ S;                                                 { наложение маски 1 }
  x:=p1^.x;  y:=p1^.y;          { интерпретация области памяти через маску 1:
                                                           x =10; y = 20; }
  . . .

  p2:=@ S;                                                 { наложение маски 2 }
  x:=p2^.x;  y:=p2^.y;          { интерпретация области памяти через маску 2:
                                                           x =20; y = 10; }
  . . .
```

Наложение маски с помощью указателя широко используется для создания ***динамических массивов переменной длины***. При создании динамических массивов переменной длины память выделяется динамически, в зависимости от размера массива. Интерпретация области памяти, отведенной под динамический массив, осуществляется через маску. Размер маски рекомендуется выбирать так, чтобы она была рассчитана не более чем

на максимальное количество данных интерпретируемого типа, которое может разместиться в пределах сегмента.

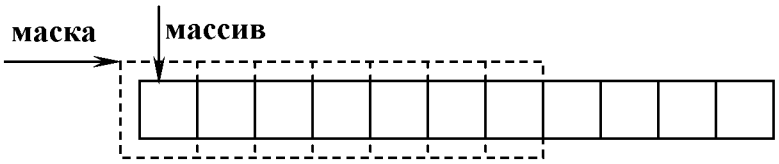
```
Type
  Mas = array[1..10000] of  word;                          { маска }
  PMas = ^ Mas;          { типизированный указатель, связанный с маской }

var
  P: PMas;                                  { указатель для наложения маски }
  A: Pointer;                  { адрес области памяти, выделенной для массива
                                                          переменной длины }
  n: word;                                          { истинный размер массива }
  i: word;

begin
  write ( “введите размерность массива = ” );  readln( n );
  getmem( A, n * sizeof( word ) );              { динамическое выделение области
                                                          памяти для массива }
  p:=A;                                                  { наложение маски }
  for i:=1 to n do readln( p^[i] );          { интерпретация области памяти через
                                                                  маску }
  . . .
```

При использовании динамических массивов переменной длины следует обратить особое внимание на соотношение между верхней границей маски и верхней границей массива. При этом возможны две ситуации:

1. Число элементов динамического массива N больше верхней границы маски.



При обращении к элементу массива с номером, большим верхней границы маски, будет зафиксирована ошибка времени исполнения – выход за пределы верхней границы массива.

2. Число элементов динамического массива N меньше верхней границы маски.

При обращении к элементу массива с номером, большим N, ошибка времени исполнения не фиксируется, а происходит попадание в область памяти, не принадлежащую массиву размером N, но интерпретируемую через маску. В результате данные, которые располагаются сразу за массивом, могут быть испорчены.

1.3. Контрольные вопросы к главе 1

1. Дайте определение понятия совместимости типов.

2. Что такое приведение типа? Приведите примеры.

3. Что такое преобразование типа? Какие виды преобразования типов Вам известны? Приведите примеры.

4. Какие методы совмещения типов Вам известны? Дайте определение понятия маски.

5. Для чего следует использовать запись с вариантной частью? Какова ее структура? В чем отличия именованного и неименованного селектора вариантов?

6. Каким образом выполняется совмещение объектов с помощью директивы ABSOLUTE?

7. В чем особенности использования параметров без типа?

8. В чем особенности использования открытых массивов?

9. Каким образом выполняется множественная интерпретация объектов с помощью указателей?

10. Каким образом используется наложение масок с помощью указателей при работе с динамическими массивами?

1.4. Упражнения к главе 1

1. Напишите процедуру, меняющую местами значения старшего и младшего байтов аргумента типа WORD. Стандартные функции не использовать.

2. Реализуйте процедуру, которая распечатывает значения младшего и

старшего байтов каждого из элементов массива натуральных чисел. Стандартные функции не использовать.

3. Напишите процедуру динамического создания массива для хранения координат N точек и определите максимальное значение координаты по оси Y (массив заполняется значениями, вводимыми с клавиатуры).

4. Динамически создайте массив для хранения N измерений одного из параметров физического процесса, заполните этот массив значениями. Затем выполните сжатие результатов следующим образом: в каждой группе из 10 значений определите максимальное и среднее значения, запишите эти показания в новый динамический массив соответствующего размера, после чего исходный массив уничтожьте.

5. Напишите процедуры динамического создания и уничтожения массива для хранения результатов физического эксперимента. Экспериментальная зависимость представляет собой последовательность пар аргумент/функция.

6. Переменная **size : longint** содержит нормализованную длину свободного блока: в старшем слове - количество параграфов, в младшем - количество свободных байтов в диапазоне 0..15. Преобразуйте нормализованную длину свободного блока в фактическую длину, выраженную в байтах. Стандартные функции не использовать.

7. Напишите процедуры динамического создания (с заполнением значениями) и разрушения двумерной квадратной матрицы размера 200 * 200. Матрица содержит вещественные числа и является неразреженной.

2. РЕКУРСИВНЫЕ АЛГОРИТМЫ ОБРАБОТКИ СТРУКТУР ДАННЫХ

2.1. Итерация и рекурсия в программировании

2.1.1. Понятие рекурсии

Рекурсия – есть метод определения множества объектов или процесса в терминах самого себя.

Рекурсивные определения

Любое рекурсивное определение содержит две части: ***базисную часть*** и собственно ***рекурсивную***. Например, понятие нечетного целого числа определяется следующим образом:

- базисная часть: число 1 является нечетным целым числом;
- рекурсивная часть: если какое либо число К является нечетным целым числом, то нечетными целыми будут числа, определяемые выражениями К-2 и К+2.

Это определение состоит из двух независимых частей: базисной (или базы) и рекурсивной. Базисная часть является нерекурсивным утверждением, которое задает определение для некоторой фиксированной группы объектов. Рекурсивная часть определения записывается таким образом, чтобы при цепочке повторных применений утверждение из рекурсивной части приводилось бы к базисной части. Таким образом, базисная часть задает один или более случаев, которые удовлетворяют определению, а рекурсивная часть показывает, как применить определение, чтобы проверить, удовлетворяют ли ему другие случаи.

Например, докажем, что число К = 7 является нечетным целым числом. Для этого применим рекурсивную часть определения:

число К = 7 является нечетным целым числом, если нечетным целым является число К - 2 = 7 - 2 = 5;

число К = 5 является нечетным целым числом, если нечетным целым является число К - 2 = 5 - 2 = 3;

число К = 3 является нечетным целым числом, если нечетным целым является число К - 2 = 3 - 2 = 1;

число К = 1 является нечетным целым числом. Таким образом, рекурсивное утверждение удалось привести к базе, следовательно, первоначальное утверждение о том, что число К = 7 - нечетное целое число является истинным.

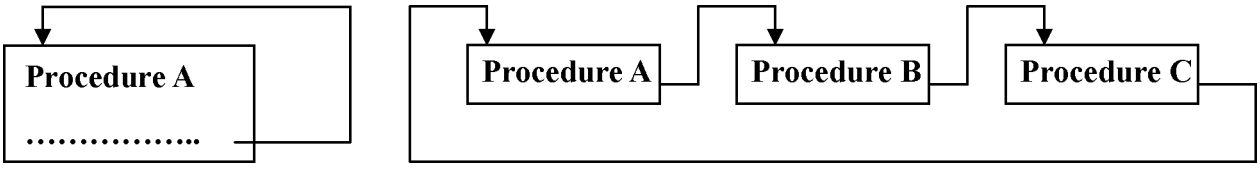
Рекурсивные алгоритмы

Рекурсивный алгоритм реализует какое-либо рекурсивное определение посредством разбиения решаемой задачи на подзадачи меньшей размерности, выполняемые с помощью одного и того же алгоритма. Процесс разбиения завершается при достижении простейших возможных решаемых задач

минимальной размерности, которые называются условиями завершения. Таким образом, рекурсивное определение алгоритма включает две части:

- ***условия завершения*** (одно или несколько), которые могут быть вычислены для определенных параметров. Условия завершения соответствуют базисной части рекурсивного определения;
- ***шаг рекурсии***, в котором текущие значения некоторых переменных в алгоритме могут быть определены в терминах их предыдущих значений. В конечном итоге шаг рекурсии должен приводить к выполнению условий завершения.

Рекурсивные алгоритмы реализуются через ***рекурсивные процедуры (функции)***. Рекурсивной называется процедура, которая в процессе выполнения явно или неявно вызывает саму себя. ***Прямая (явная) рекурсия*** характеризуется наличием в теле процедуры оператора обращения к ней же самой (рис. 4.а). В случае ***косвенной (неявной) рекурсии*** одна процедура обращается к другой, которая (возможно через цепочку вызовов других процедур) вновь обращается к первой (рис. 4.б). Далее будем рассматривать только прямую рекурсию.



а б

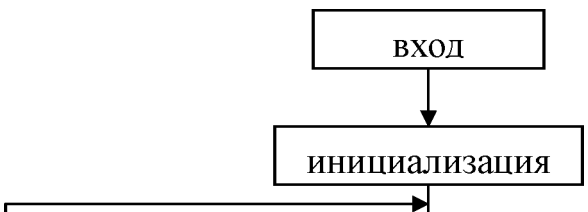
Рис. 4. Схема прямой и косвенной рекурсии:
а – прямая рекурсия; б – косвенная рекурсия

2.1.2. Итеративная и рекурсивная схема организации вычислительного процесса

Для того чтобы лучше понять особенности рекурсивных алгоритмов, полезно сопоставить итеративнную и рекурсивную огранизацию процесса вычислений в программе. Особенности итеративного и рекурсивного вычислительного процесса рассмотрим на примере вычисления значения факториала некоторого натурального числа N.

Итеративная схема организации вычислительного процесса

Итеративный процесс можно проиллюстрировать с помощью схемы, приведенной на рис. 5. Этот процесс состоит из четырех блоков: инициализации, принятия решения (о продолжении вычислений), вычисления и модификации.



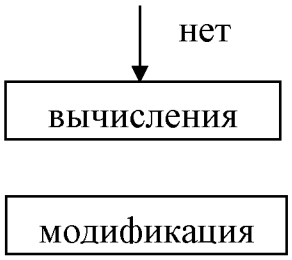


Рис. 5. Схема итеративного вычислительного процесса

В основе итеративного вычислительного процесса лежит ***итеративный цикл*** *While*, *Repeat-Until*, *For*. Наиболее универсальным является цикл *While*:

While < условие цикла > do < тело цикла >;

Итеративная схема вычисления факториала:

N! = 1 * 2 * 3 * … * N.

Процедура, реализующая итеративную схему вычисления факториала, приведена ниже:

```
Procedure Iter_Fact ( n: byte; var f: longint );
var i: word;
begin
  i:=1;  f:=1;                                  { инициализация }
  while i < = n do begin                        { решение о завершении }
    f:= f * i;                                  { вычисления }
    inc( i );                                   { модификация }
  end;
end;
```

Существует два важных положения, известных в математике и в программировании, определяющих соотношение между итерацией и рекурсией:

1. любой итеративный цикл может быть заменен рекурсией;

2. рекурсия не всегда может быть заменена итерацией.

Рекурсивная схема организации вычислительного процесса

Общая схема рекурсивного вычислительного процесса представлена на рис. 6.

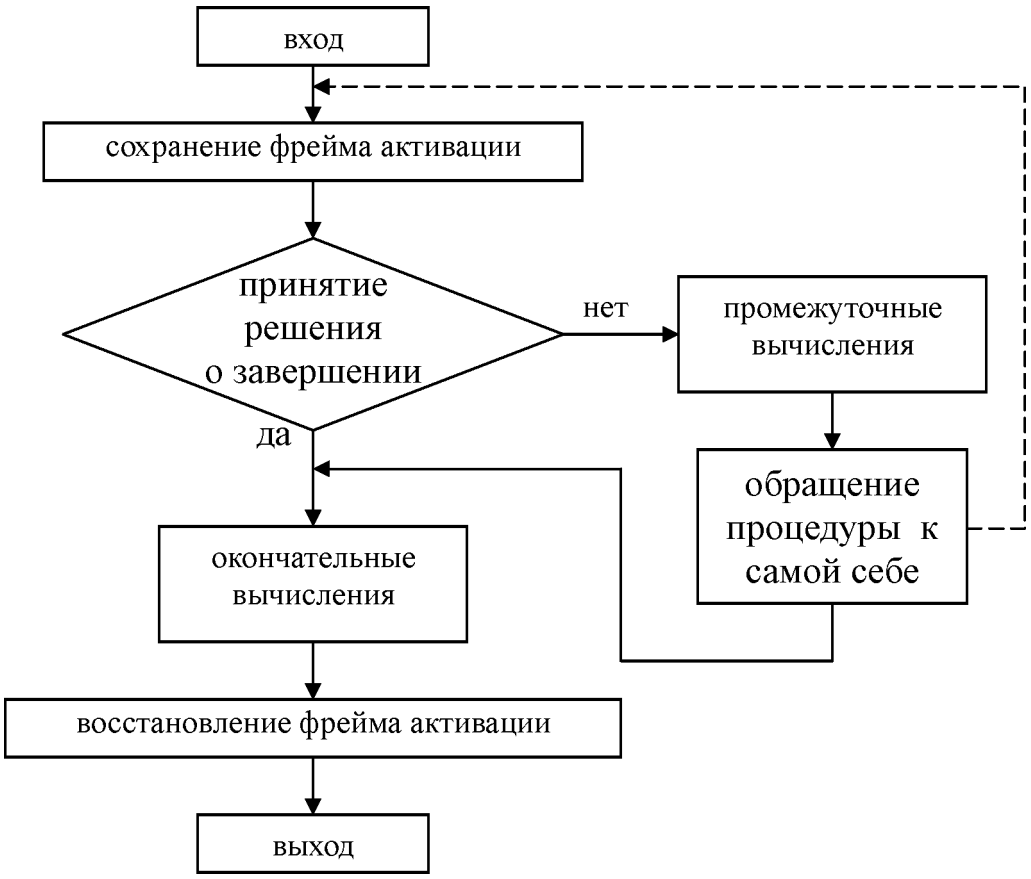


Рис. 6. Схема рекурсивного вычислительного процесса

Так как обращаться к рекурсивной процедуре можно как из нее самой, так и извне, каждое обращение к рекурсивной процедуре вызывает ее независимую активацию. При каждой активации образуются копии всех локальных переменных и формальных параметров рекурсивной процедуры, в которых “оставляют следы” операторы текущей активации. Таким образом, для рекурсивной процедуры может одновременно существовать несколько активаций. Для обеспечения правильного функционирования рекурсивной процедуры необходимо сохранять адреса возврата в таком порядке, чтобы возврат после завершения каждой текущей активации выполнялся в точку, соответствующую оператору, непосредственно следующему за оператором рекурсивного вызова. Совокупность локальных переменных, значений фактических параметров, подставляемых на место формальных параметров рекурсивной процедуры, и адреса возврата однозначно характеризует текущую активацию и образует ***фрейм активации***. Фрейм активации необходимо сохранять при очередной активации и восстанавливать после завершения текущей активации.

В блоке принятия решения (о продолжении вычислений) производится проверка, являются ли значения входных параметров такими, для которых

возможно вычисление значений выходных параметров в соответствии с базисной частью рекурсивного определения. На основании этой проверки принимается решение о выполнении промежуточных или окончательных вычислений. Блок промежуточных вычислений можно объединить с блоком обращения к процедуре, если промежуточные вычисления очень просты. В блоке окончательных вычислений производится явное определение параметров-переменных процедуры для конкретных значений входных параметров, соответствующих текущей активации процедуры.

В основе рекурсивного вычислительного процесса лежит ***рекурсивный цикл***, который реализуется через вызов рекурсивной процедуры, причем каждая активация рекурсивной процедуры эквивалентна одному проходу итеративного цикла *While*.

Общая схема рекурсивного цикла:

```
Procedure Рекурсивный_Цикл (...);
begin
  if  < условие цикла > then
  begin
     < тело рекурсивного цикла; >
     Рекурсивный_Цикл (…);
   end;
end;
```

В теле рекурсивного цикла (в блоке промежуточных вычислений) обязательно должны содержаться операторы, изменяющие значения переменных, от которых зависит условие завершения рекурсивного цикла. Напомним, что выполнение условия завершения рекурсивного цикла соответствует достижению базиса рекурсивного определения. Если значения этих переменных не успевают измениться в теле цикла до очередной активации рекурсивной процедуры, то возникает ***бесконечный рекурсивный цикл***.

Общая схема бесконечного рекурсивного цикла:

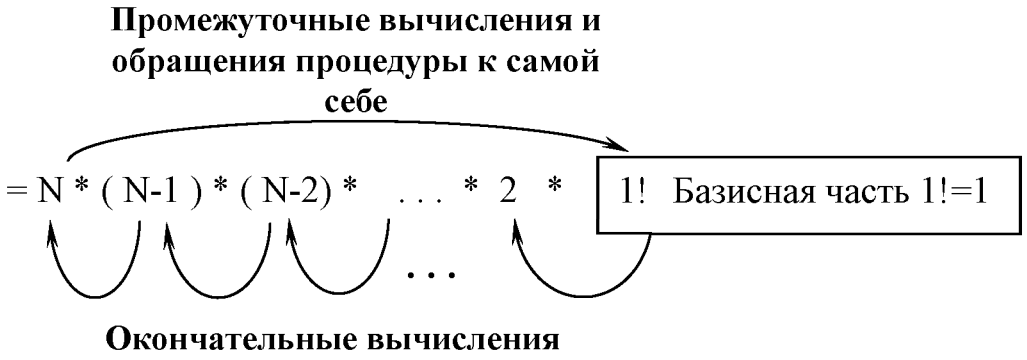
```
Procedure Бесконечный_Рекурсивный_Цикл (...);
begin
   if  < условие цикла > then
   begin
      Бесконечный_Рекурсивный_Цикл (…);
    < тело рекурсивного цикла; >
   end;
end;
```

Рекурсивная схема вычисления факториала:

Базисная часть:
0! = 1; 1! = 1;

Рекурсивная часть:

N! = N * (N-1)! = N * (N-1) * (N-2)! = N * (N-1) * … * (N – (N-1))! =



Процедура, реализующая рекурсивную схему вычисления факториала:

```
Procedure Recurs_Fact (n : byte; var f: longint );
begin
 if  n <= 1 then                    { принятие решения о завершении  вычислений:}
  f:=1                              { да - окончательные вычисления для базисной части }
 else  begin
   Recurs_Fact ( n-1, f );          { нет - промежуточные вычисления и обращение процедуры
                                      к самой себе }
  { 1 }  f:=f * n;                  { окончательные вычисления для рекурсивной части }
  end;
{ 2 } end;
```

{ 1 } – адрес возврата после завершения активации,

{ 2 } – завершение активации.

С каждым обращением к рекурсивной процедуре ассоциируется ***номер уровня рекурсии*** (номер фрейма активации). Считается, что при первоначальном вызове рекурсивной процедуры из основной программы номер уровня рекурсии равен единице. Каждый следующий вход в процедуру имеет номер уровня на единицу больше, чем номер уровня процедуры, из которой производится это обращение.

Другой характеристикой рекурсивной процедуры является ***глубина рекурсии***, определяемая максимальным уровнем рекурсии в процессе вычисления при заданных аргументах. В общем случае эта величина неочевидна, исключение составляют простые рекурсивные функции: например, при вычислении значения N! глубина рекурсии равна N.

Так как при выходе из текущей активации самым первым должен быть восстановлен фрейм, который был позже всех сохранен, для хранения фреймов используется автоматическая память, т.е. область системного стека ([10]: с.31). Таблица 1 и рис. 7 поясняют механизм рекурсивных вычислений.

Таблица 1.

Трасса вычисления 3!

№ уровня рекурсии	Знач-е вход. пар-ра n	Содержимое стека		Знач-е выход. пар-ра F	Выход из уровня
		До вызова N,F,(*)возврата	После вызова N,F,(*)возврата		
1	3	–,–,–	3, □ ,(*1*)		
2	2	3, □ ,(*1*)	2, □ ,(*1*) 3, □ ,(*1*)		
3	1	2, □ ,(*1*) 3, □ ,(*1*)	1, □ ,(*1*) 2, □ ,(*1*) 3, □ ,(*1*)		
		До возврата из уровня	После возврата из уровня		
3	1	1, □ ,(*1*) 2, □ ,(*1*) 3, □ ,(*1*)	2, □ ,(*1*) 3, □ ,(*1*)	1	(*2*)
2	2	2, □ ,(*1*) 3, □ ,(*1*)	3, □ ,(*1*)	2	(*2*)
1	3	3, □ ,(*1*)	–,–,–	6	(*2*)

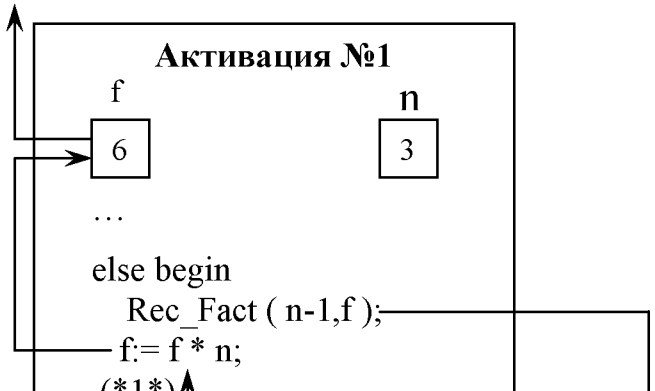


Рис. 7. Фреймы активации при вычислении 3!

2.2 Виды рекурсивных алгоритмов

2.2.1. Вычислительные алгоритмы

а) Вычисление n-го по счету члена числовой последовательности.

Примером числовой последовательности являются числа Фибоначчи. Числа Фибоначчи составляют последовательность, очередной элемент которой вычисляется по двум предыдущим значениям:

$$F_n = F_{n-1} + F_{n-2}.$$

Нулевое и первое значения должны быть заданы и равны единице. Последовательности такого рода применяются, например, в программных генераторах случайных чисел:

```
Uses Crt;
var n: word;

Function Fib( n: word ): longint;
```

```
begin
   if ( n = 0 ) or ( n = 1 ) then Fib:=1                { условие завершения }
   else Fib:= Fib( n-2 ) + Fib( n-1 )                   { шаг рекурсии }
end;

begin
   write('Введите значение n ');  readln( n );
   writeln( 'Fib( ', n, ' ) =  ', Fib( n )  );
end.
```

Каждое обращение при $n > 1$ приводит к двум дальнейшим обращениям, т.е. общее число обращений растет экспоненциально. Очевидно, что числа Фибоначчи более эффективно можно вычислять по итеративной схеме.

Однако существуют такие числовые последовательности, для которых применение рекурсии – единственный способ решения. Например:

$$a(1) = 1;\ a(n) = n - a(a(n-1)),\ n > 1.$$

б) Вычисление значений полиномов порядка n в заданной точке x. Например, полиномы Чебышева определяются следующим образом:

$$T_0(x) = 1; \qquad T_1(x) = x; \qquad T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x).$$

Таким образом, текущее значение полинома определяется по двум предыдущим значениям.

в) Решение разностных уравнений. Рассмотрим, например, уравнение следующего вида:

$$A(0) = 1; \qquad A(n) = A(n \text{ div } 2) + A(n \text{ div } 3).$$

Простого и очевидного способа итерационного программирования функции $A(n)$ не существует. Решение разностных уравнений выполняется с помощью рекурсии.

г) Сортировки.

В качестве примера рассмотрим ***алгоритм быстрой сортировки.***

1. Выбрать в массиве некоторый элемент, называемый опорным элементом, желательно, чтобы опорный элемент разбивал множество элементов массива на две примерно равные части.
2. Выполнить операцию разделения массива таким образом, чтобы все элементы, меньшие или равные опорного элемента, оказались слева от него, а все элементы, большие опорного – справа от него. Это

достигается путем просмотра массива попеременно с обоих концов, причем каждый элемент сравнивается с опорным. Элементы из левой части, не меньшие опорного, меняются местами с элементами из правой части, не большими опорного. После завершения процедуры разделения опорный элемент оказывается на своем окончательном месте. Все элементы, которые меньше опорного, будут стоять слева от него, а все элементы, которые больше опорного, - справа от него.

3. Выполнить пункты 1 и 2 над частями массива слева и справа от опорного элемента.

Базой рекурсии являются массивы, состоящие из одного или двух элементов, которые уже отсортированы. Алгоритм всегда завершается, поскольку за каждую итерацию по крайней мере один элемент размещается на его окончательном месте.

д) Бинарный поиск элемента с заданным значением в упорядоченном множестве объектов (массиве).

При бинарном поиске берется некоторый ключ и просматривается упорядоченный массив из *N* элементов на предмет совпадения с этим ключом (элементы индексируются от *1* до *N*). Результатом поиска является индекс совпавшего с ключом элемента или *0* при отсутствии такового. Алгоритм бинарного поиска может быть описан рекурсивно. Пусть упорядоченный массив *A* характеризуется нижним индексом *low* и верхним индексом *high*. Поиск совпадения с ключом начинается в середине массива (индекс *mid*):

mid = (low + high) / 2; сравнить *A[mid]* с ключом.

Если совпадение произошло, условие останова достигнуто, что позволяет прекратить поиск и возвратить индекс *mid*. Если совпадения не происходит, можно воспользоваться тем фактом, что массив упорядочен, и ограничить диапазон поиска “нижним подмассивом” (слева от *mid*) или “верхним подмассивом” (справа от *mid*).

Если ключ меньше *A[mid]*, совпадение может произойти только в левой половине массива в диапазоне индексов от *low* до *mid – 1*. Если ключ больше *A[mid]*, совпадение может произойти только в правой половине массива в диапазоне индексов от *mid + 1* до *high*.

Шаг рекурсии направляет бинарный поиск для продолжения в один из подмассивов. Рекурсивный процесс просматривает массивы все меньшего размера. Поиск заканчивается неудачей, если подмассивы исчезли, т.е. если нижняя граница превосходит верхнюю. Условие *low > high* – второе условие останова рекурсивного процесса. В этом случае алгоритм возвращает *0*.

2.2.2. Перебор с возвратами

Перебор с возвратами (бектрекинг - back tracking – обратное следование) – это способ поиска решения, когда при возврате после рассмотрения “пробного” варианта решения на один шаг назад все переменные программы восстанавливают свои значения. Классическим примером перебора с возвратами является ***алгоритм прохождения лабиринта***, идея которого состоит в следующем. Лабиринт есть множество перекрестков, связанных между собой. Предположим, что каждый перекресток имеет три атрибута, определяющих возможность перемещения налево, прямо и направо. Если значение некоторого атрибута равно единице, то движение в данном направлении возможно. Нуль показывает, что движение в данном направлении заблокировано. Три нулевых значения атрибутов перемещения из некоторого перекрестка представляют тупик. Проход по лабиринту считается успешным при достижении точки “Выход”.

Процесс поиска выхода включает ряд рекурсивных шагов. В каждом перекрестке необходимо исследовать возможные варианты. Если возможно, сначала следует пойти налево. Достигнув очередного перекрестка, необходимо снова рассмотреть варианты и попытаться пойти налево. Если выбор левого направления заводит в тупик, следует отступить на один шаг назад и выбрать движение прямо, если это направление не заблокировано. Если этот выбор заводит в тупик, вновь необходимо отступить на один шаг и пойти направо. Если все альтернативы движения из данного перекрестка ведут в тупики, следует вернуться к предыдущему перекрестку и сделать новый выбор. Если произошел возврат к исходной точке, и из нее нет новых путей, задача поиска выхода из лабиринта не имеет решения. Рассмотрим варианты в лабиринте, изображенном на рис. 8.

Перекресток	Выбор	Результирующий перекресток
1	прямо	2
2	налево	3
3	налево	7
7	тупик: вернуться к 3	3
3	прямо	4
4	прямо	6
6	тупик: вернуться к 4	4
4	направо	5
5	тупик: вернуться к 4, 3, 2	2
2	прямо	8
8	налево	9
9	прямо	10 - выход

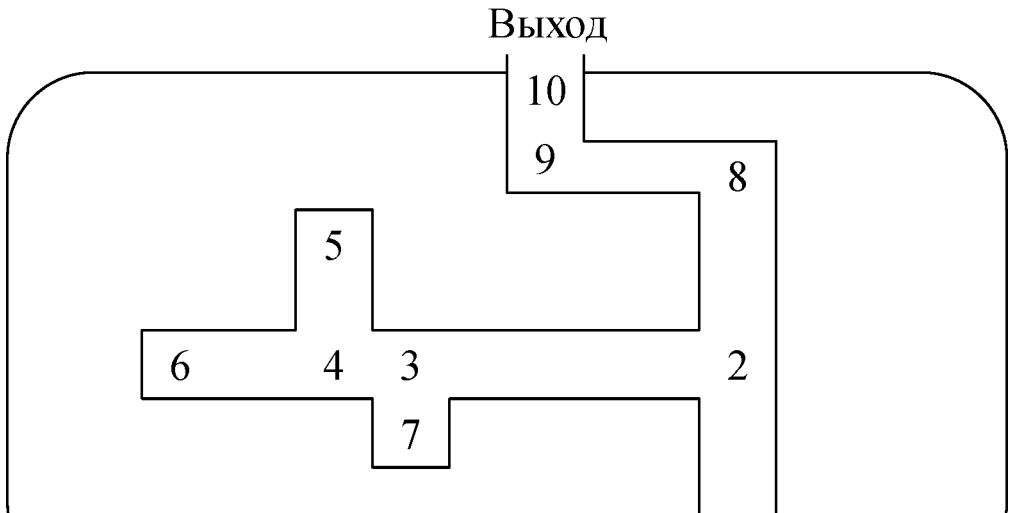


Рис. 8. Лабиринт

С помощью алгоритма бектрекинга решаются разнообразные ***шахматные задачи***. Наиболее известными являются следующие из них:

- задача о расстановке восьми ферзей. Требуется найти все возможные способы расстановки восьми ферзей на шахматной доске так, чтобы ни один из них не нападал на любого другого;
- задача о расстановке ладей. Требуется определить все возможные способы расстановки *n* ладей на шахматной доске размером *n* * *n* так, чтобы они не угрожали друг другу;
- задача обхода шахматной доски размером 8 * 8 ходом коня так, чтобы конь побывал на каждом поле один раз.

2.2.3. Комбинаторика

Рекурсия находит широкое применение в комбинаторике – разделе математики, занимающемся подсчетом объектов.

Перестановка из N элементов (1,2, …, N) есть упорядоченное расположение этих элементов. Для N = 3 (1 3 2), (3 2 1), (1,2,3) – различные перестановки. В комбинаторике установлено, что число перестановок равно N!. Для позиции 1 существуют N вариантов, т.к. все N элементов доступны. Для позиции 2 имеются N – 1 вариантов, т.к. один элемент зафиксирован в позиции 1. Число вариантов уменьшается на 1 по мере продвижения по позициям. Общее число перестановок есть произведение числа вариантов в каждой позиции.

$$Permulation(N) = N * (N-1) * (N-2) * \ldots * 2 * 1 = N!$$

Рекурсивный алгоритм генерирует перечень всех перестановок из N элементов для N > 1. Сформируем 24 (4!) перестановки из четырех элементов. Перестановки с одинаковыми первыми элементами записываются в отдельный столбец. Каждый столбец затем делится на три пары с одинаковыми вторыми элементами:

1	**2**	**3**	**4**
1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1

1 4 2 3	2 4 1 3	3 4 1 2	4 3 1 2
1 4 3 2	2 4 3 1	3 4 2 1	4 3 2 1

Алгоритм вычисления числа перестановок иллюстрируется иерархическим деревом (рис. 9), которое содержит упорядоченные пути, соответствующие перестановкам. В исходном положении имеется четыре варианта – 1, 2, 3 и 4, соответствующие четырем столбцам. По мере продвижения вниз по дереву нижние уровни разделяются на 3, 2 и 1 элемент, соответственно. Алгоритм генерации всех перестановок моделирует проход по путям дерева. Продвигаясь от уровня к уровню, мы тем самым указываем очередную позицию в перестановке. Этот процесс представляет собой шаг рекурсии.

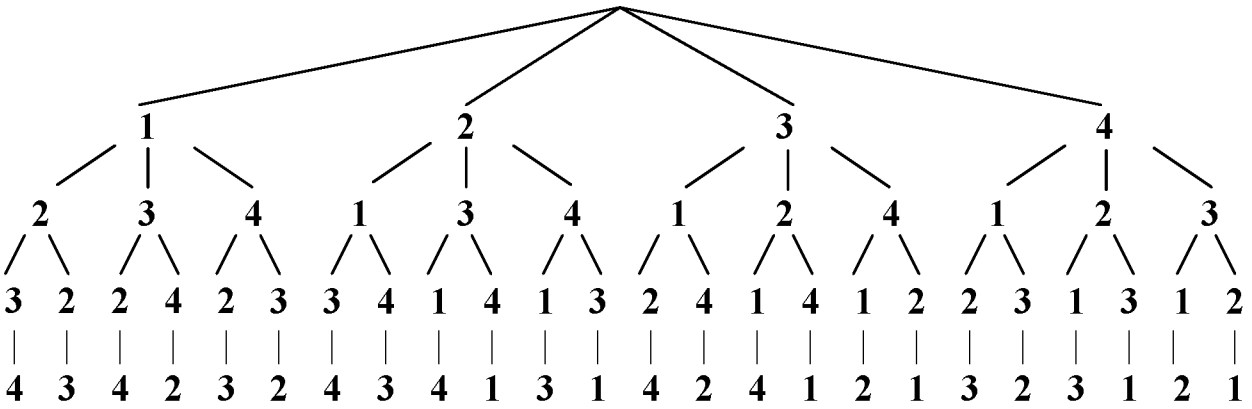


Рис. 9. Иллюстрация работы алгоритма перестановок

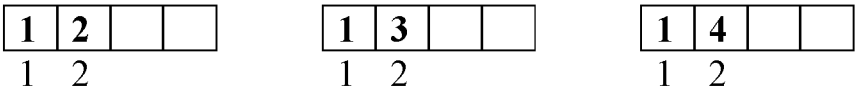
Шаг рекурсии выполняется следующим образом. Представим перестановку в виде массива из N элементов. Индекс элемента массива – это номер позиции перестановки. В каждом элементе массива хранится значение, соответствующее позиции перестановки, определяемой индексом элемента.

Итерационный процесс проверяет все возможные значения первого элемента перестановки. В нашем примере существует N=4 возможных значения первого элемента:

Индекс 1:

На следующем уровне дерева каждый узел дает начало N-1 перестановкам, содержащим N-1 отличных от первого элементов. Напроимер, узел 1 соответствует перестановкам, которые начинаются с единицы в первой (индекс 1) позиции. Путь, ведущий к узлу 2 следующего уровня, соответствует перестановкам со значениями 1 и 2 в первых двух позициях и т.д. Можно итерационно определить второй элемент перестановки перебором узлов 2, 3 и 4:

Индекс 2:



На следующем уровне дерева каждый узел разветвляется на два пути, которые представляют перестановки из двух элементов. Например, если во второй позиции (индекс 2) находится элемент 2, в третьей позиции (индекс 3) могут быть только элементы 3 или 4:

Индекс 3:

Так как третий элемент перестановки зафиксирован, последний элемент определяется однозначно, поскольку перестановка не допускает повторяющиеся значения:

Индекс 4:

Определение всех N элементов перестановки является ***условием завершения*** работы рекурсивного алгоритма.

Алгоритм перестановок можно запрограммировать, оперируя массивом A[1] ,…, A[n], в котором хранится перестановка чисел 1..n. Рекурсивная процедура [11] распечатывает все перестановки, которые в первых t позициях совпадают с перестановкой A:

```
Var A: array[1..10] of word;                    { Массив для хранения перестановок }
   i, t, k, n: byte;

Procedure Generate;                             { Перестановки чисел 1..N }
  var i,j : byte;
begin
  if t = n then begin                           { Условие завершения шага }
    for i:=1 to n do write( A[i], ' ');  readkey; writeln end     { Перестановка готова }
    else
      for j:=t+1 to n do begin                  { Формирование перестановки: t < n }
        k:=A[t+1]; A[t+1]:=A[j]; A[j]:=k;       { Поменять местами a[t+1] и a[j]  }
        inc( t ); Generate; dec( t );
        k:=A[t+1]; A[t+1]:=A[j]; A[j]:=k;       { Поменять местами a[t+1] и a[j]  }
      end;
end;

begin
```

```
writeln('введите значение n = '); readln( n );
for i:=1 to n do A[i]:=i; t:=0; Generate;
```

Алгоритм перестановок можно запрограммировать также с использованием типа данных МНОЖЕСТВО ([10]: с.16). Обозначим k номер текущей позиции перестановки, для которой выбирается значение. В переменной S множественного типа удобно хранить элементы, которые еще не зафиксированы в позициях перестановки с индексами, меньшими k. Элемент i, который должен быть записан в текущую позицию с индексом k, может быть выбран среди элементов множества S с использованием операции определения принадлежности элемента множеству – *in (i in S)*. После того как сформирована текущая позиция перестановки, элемент i, который в ней зафиксирован, должен быть исключен из множества с использованием операции вычитания элемента из множества ($S - [i]$). Номер позиции k увеличивается на единицу. Алгоритм рекурсивно выполняется, чтобы выбрать следующий элемент перестановки среди оставшихся в множестве S элементов и зафиксировать его в позиции с индексом $k+1$:

```
Procedure Permulation( n: byte );                    { Перестановки чисел 1..N }
  const Max_Set = 10;                                { Количество элементов в множестве }
  type Set_Per = set of 1..Max_Set;                  { Множественный тип для хранения
                                                       незафиксированных элементов }
  var A: array[1..Max_Set] of byte;                  { Тип массива для хранения перестановок }

  Procedure Pere( S: Set_Per; k: byte );
  { S – множество для хранения элементов, которые не зафиксированы в позициях
        перестановки, k – текущая позиция перестановки }
   var i: byte;
    begin
      if k = n+1 then begin write ("перестановка  ");      { Условие завершения шага }
       for i:=1 to n do write( A[i], ' '); readkey; writeln end    { Перестановка готова }
      else
        for i:=1 to n do     { Формирование k-й позиции перестановки и вызов процедуры }
          if i in S then begin A[k]:=i; Pere( S-[i], k+1 ) end;      { для (k+1)-й позиции }
    end;   { Pere }
  begin  S:=[1..n];  Pere( S,1 );
end;       {Permulation }
```

2.2.4. Игры и головоломки: задача о "ханойских башнях"

Согласно легенде, у жрецов храма Брахмы в горах Тибета есть медная платформа с тремя алмазными стержнями A, B и C. На одном стержне A нанизано 64 золотых диска, каждый из которых немного меньше того, что

под ним. Конец света наступит, когда жрецы переместят диски со стержня A на стержень C. Задача имеет следующие условия:

- за один раз можно перемещать только один диск,
- больший диск нельзя класть на меньший,
- снятый диск нельзя отложить, его необходимо сразу же надеть на другой стержень.

Несомненно, жрецы все еще работают, т.к. задача включает в себя $2^{64} - 1$ ходов. Если тратить по одной секунде на ход, то потребуется примерно 500 миллиардов лет.

Решение данной задачи носит рекурсивный характер. Задача разбивается на последовательность подзадач одного и того же типа, но меньшей размерности. Условием завершения является выполнение простой задачи перемещения одного диска. Сформулируем алгоритм решения данной задачи для N дисков. Обозначим стержни: начальный (*start*), промежуточный (*temp*), конечный или целевой (*destination*). На начальный стержень нанизано N дисков в порядке возрастания размера, т.е. самый большой диск лежит внизу. Цель задачи – переместить все N дисков с начального стержня на конечный, следуя определенным выше условиям, и распечатать перечень ходов. Предполагается, что промежуточный стержень будет использоваться для временного хранения дисков.

Иллюстрация решения задачи для N = 1 диска приведена на рис. 10, для N = 2 дисков - на рис. 11, для N = 3 дисков – на рис. 12. Алгоритм требует $2^N - 1$ ходов.

Если N = 1, выполняется условие завершения (базисное условие рекурсивного алгоритма), которое может быть обработано путем перемещения единственного диска с начального стержня на конечный и распечатки соответствующего хода. Для N > 1 выполняется трехшаговый процесс перемещения N дисков с начального стержня на конечный, причем стержень A является начальным (*A – start*), стержень B – промежуточным (*B – temp*), стержень C – конечным (*C – dest*).

На первом шаге алгоритма перемещаются N – 1 дисков с начального стержня на промежуточный с использованием конечного стержня для временного хранения. При этом стержень A выполняет роль начального (*A – start*), стержень B выполняет роль конечного (*B – dest*). Конечный стержень C используется как промежуточный (*C – temp*).

На втором шаге самый большой диск просто перемещается с начального стержня на конечный: *start -> dest*.

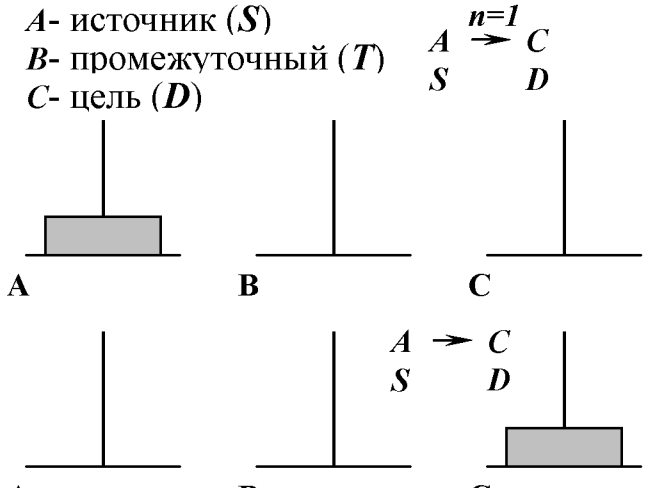


Рис. 10. Решение задачи о “ханойских башнях” для N = 1 диска

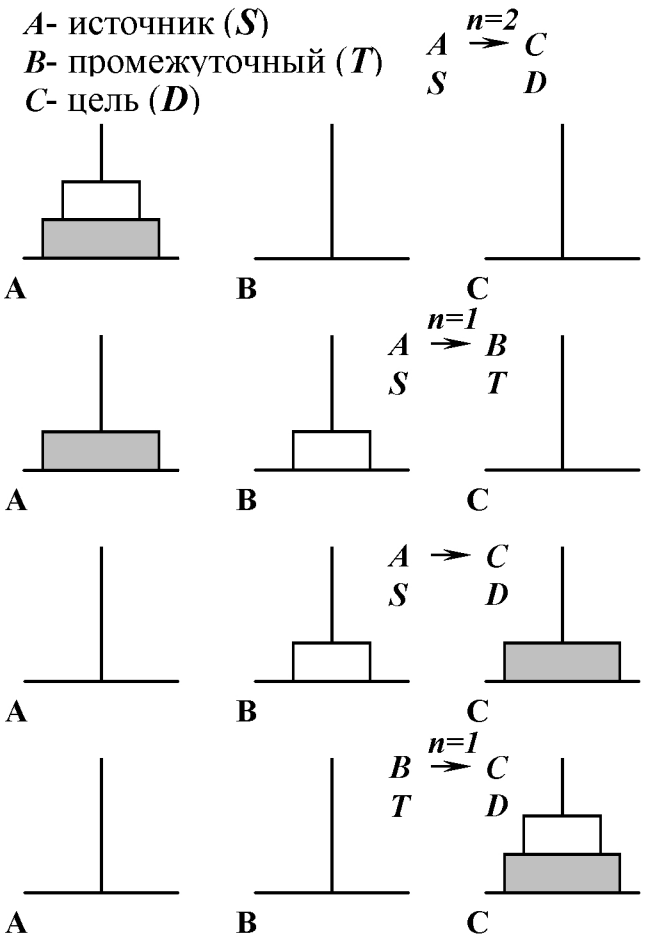


Рис. 11. Решение задачи о “ханойских башнях” для N = 2 дисков

На третьем шаге N – 1 дисков перемещаются со среднего стержня на конечный с использованием начального стержня для временного хранения. При этом стержень B выполняет роль начального (*B – start*), стержень C выполняет роль конечного (*C – dest*). Начальный стержень A используется как промежуточный (*A – temp*).

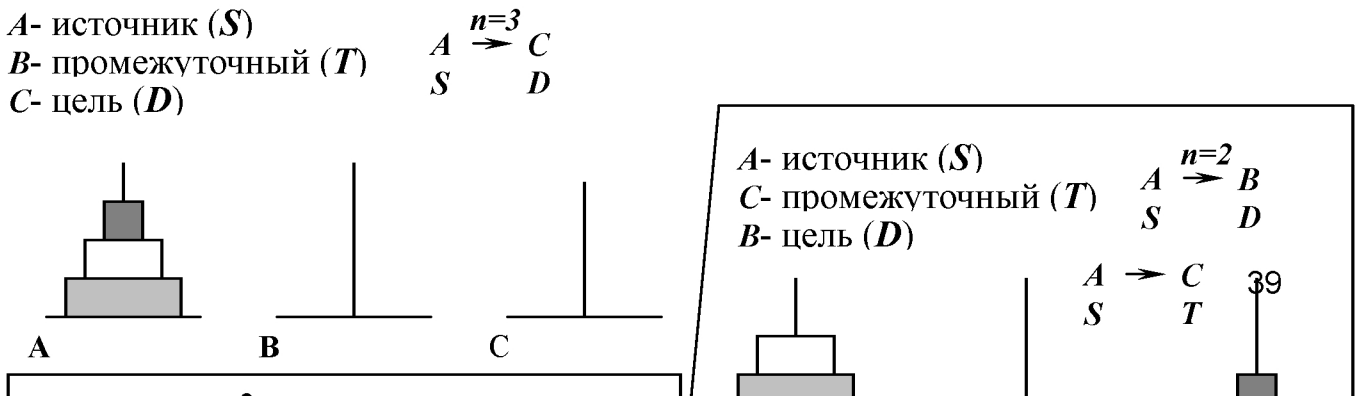


Рис. 12. Решение задачи о “ханойских башнях” для N = 3 дисков

Программная реализация алгоритма решения задачи о “ханойских башнях” следует ниже:

```
{ перенести N дисков с начального стержня на конечный, используя промежуточный
стержень для временного хранения дисков }
Procedure Hanoi( n: byte; start, tmp, dest: char );
begin
 if  ( n=1 ) then                    { условие завершения: перемещение одного диска }
   writeln( start, '-->', dest )

 else begin
   { перенести N–1 дисков с начального стержня на промежуточный, используя
     конечный  стержень для временного хранения дисков }
  Hanoi( n-1, start, dest, temp );
```

```
    { перенести нижний диск с начального стержня на конечный }
    writeln( start, '-->', dest );
    { перенести N–1 дисков с промежуточного стержня на конечный, используя
      начальный  стержень для временного хранения дисков }
    Hanoi( n-1, temp, start, dest )
  end;
end;
```

Более короткий вариант решения задачи о “ханойских башнях”:

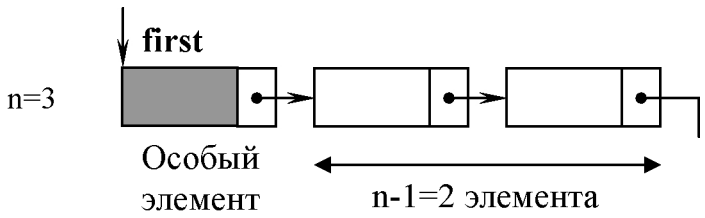
```
{ перенести N дисков с начального стержня на конечный, используя промежуточный
стержень для временного хранения дисков }
Procedure Hanoi( n: byte; start, tmp, dest: char );
begin
  if  ( n>0 ) then begin                                  { условие продолжения }
    { перенести N–1 дисков с начального стержня на промежуточный, используя
      конечный стержень для временного хранения дисков }
    Hanoi( n-1, start, dest, temp );
    { перенести нижний диск с начального стержня на конечный }
    writeln( start, '-->', dest );
    { перенести N–1 дисков с промежуточного стержня на конечный, используя
      начальный стержень для временного хранения дисков }
    Hanoi( n-1, temp, start, dest )
  end;
end;
```

Рекурсивные алгоритмы наиболее эффективны и удобны для обработки рекурсивных структур данных.

2.3. Рекурсивные алгоритмы обработки динамических линейных структур данных на примере списков

Рекурсивное определение списка: список - это пустая структура или структура, состоящая из особого элемента (первого или головного) и списка, на который указывает особый элемент. Рекурсивную интерпретацию линейного односвязного списка иллюстрирует рис. 13.

Как следует из рекурсивного определения списка, условием завершения его обработки является достижение пустого списка в процессе применения шагов рекурсии.



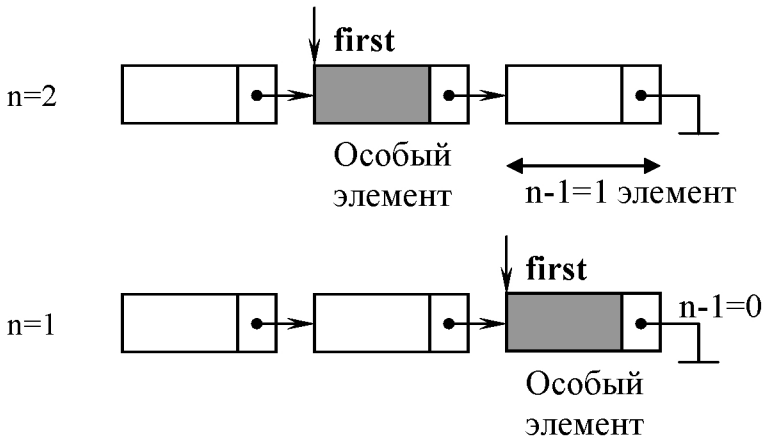


Рис. 13. Рекурсивная интерпретация списка

Рассмотрим две рекурсивные процедуры, одна из которых распечатывает содержимое информационных полей линейного односвязного списка в прямом порядке (*Print_Front*), а вторая – в обратном порядке (*Print_Back*):

```
Procedure Print_Front( first: PList );          { распечатка содержимого списка }
begin                                                  { в прямом направлении }
  If  first <> nil  then begin
    writeln( first^.info );
    Print_Front( first^.link )                    { задняя рекурсия (см. 6.4) }
  end;
end;
```

```
Procedure Print_Back( first: PList );           { распечатка содержимого списка }
begin                                                  { в обратном направлении }
  if  first <> nil  then begin
    Print_Back( first^.link )
    writeln( first^.info );
  end;
end;
```

2.4. Эффективность рекурсивных вычислений

Так как для хранения фреймов активации рекурсивной процедуры используется автоматическая память, при обработке данных нерекурсивной природы следует использовать итеративные алгоритмы, не требующие дополнительного расхода памяти, если только рекурсивные алгоритмы не оказываются более ясными и понятными. Например, при вычислении

значения факториала натурального числа N несомненно следует предпочесть итеративную процедуру.

Если процедура содержит единственный рекурсивный вызов и он является последним действием процедуры, то говорят, что имеет место ***хвостовая рекурсия*** (*tail recursion*) (см. процедуру *Print_Front*, приведенную в разделе 2.3). Этот рекурсивный **вызов требует затрат на создание фреймов активации и запоминание** их в стеке. Когда рекурсивный вычислительный процесс доходит до условия завершения, выполняется серия возвратов, выталкивающих фреймы активации из стека. Если при наличии хвостовой рекурсии фреймы активации не используются для окончательных вычислений, следует предпочесть итеративную реализацию (см. процедуру *Print*, приведенную в [10]: с.52). Рекурсивная процедура *Print_Back*, не содержащая хвостовой рекурсии, имеет более эффективную реализацию, чем итеративная процедура, выполняющая те же действия.

2.5. Контрольные вопросы к главе 2

1. В чем заключаются особенности работы с автоматической памятью?

2. Из каких частей состоит рекурсивное определение?

3. Из каких частей состоит рекурсивный алгоритм?

4. Что такое прямая и косвенная рекурсия?

5. Сравните итеративную и рекурсивную организацию вычислительного процесса. Чем они отличаются?

6. Что такое бесконечная рекурсия? Какова причина ее возникновения?

7. Что такое уровень рекурсии? Что такое глубина рекурсии?

8. Каким образом используются фреймы активации в процессе работы рекурсивного алгоритма?

9. Что такое бектрекинг? Как реализуются алгоритмы поиска с возвратом?

10. В чем особенности рекурсивной реализации комбинаторных алгоритмов?

11. Что такое “хвостовая” рекурсия? Почему при реализации алгоритмов ее следует избегать?

2.6. Упражнения к главе 2

1. Реализуйте рекурсивный алгоритм вычисления последовательности *n* вложенных корней

$$x(m,i)=\sqrt{m+\sqrt{m+...+\sqrt{m}}}, \qquad m\geq 0,\ i=\overline{1,...,n.}$$

2. Реализуйте рекурсивный алгоритм вычисления суммы *n* слагаемых. *i* – количество синусов в каждом слагаемом.

$$y(x,i)=\sin x+\sin(\sin x)+...+\sin(\sin...(\sin x)), \quad i=\overline{1,...,n.}$$

3. Реализуйте рекурсивный алгоритм вычисления суммы n первых членов ряда

$$x+\frac{x^3}{2!}+\frac{x^5}{3!}+\frac{x^7}{4!}+...+\frac{x^{2n-1}}{n!}+...$$

4. Пусть в алгебраической записи выражения имеется единственная операция – умножение, обозначаемое обычным образом (два сомножителя следуют непосредственно друг за другом). Выражение состоит из строки символов и скобок- ограничителей: ( ) [ ] { }. Реализуйте рекурсивный алгоритм, выполняющий проверку соответствия открывающих и закрывающих скобок. Например, запрещены выражения вида (ab] или a(b[c])d].

5. Задана строка S из N символов. Реализуйте рекурсивный алгоритм проверки, является ли симметричной часть строки, начинающаяся i-м и заканчивающаяся j-м ее элементом.

6. Реализуйте рекурсивный алгоритм, распечатывающий различные представления заданного натурального числа N в виде суммы не менее двух натуральных слагаемых. Представления, отличающиеся лишь порядком слагаемых, различными не считаются.

7. Реализуйте рекурсивный алгоритм, распечатывающий по одному разу в лексикографическом порядке все последовательности длины N, составленные из натуральных чисел 1..K.

8. Реализуйте рекурсивный алгоритм вычисления значения многочлена вида

$$P_n(x) = a_0 * x^n + a_1 * x^{n-1} + a_2 * x^{n-2} + ...+ a_{n-1} * x + a_n$$

в заданной целочисленной точке согласно формуле

$$P_n(x) = x * P_{n-1}(x) + a_n, \quad \text{где}$$

$$P_{n-1}(x) = a_0 * x^{n-1} + a_1 * x^{n-2} + \ldots + a_{n-2} * x + a_{n-1}.$$

9. Реализуйте рекурсивный алгоритм нахождения наибольшего общего делителя последовательности N натуральных чисел.

10. Реализуйте рекурсивный алгоритм двоичного поиска элемента с заданным значением в упорядоченном целочисленном массиве.

3. ИЕРАРХИЧЕСКИЕ НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ. ДЕРЕВЬЯ

3.1. Деревья общего вида

Нелинейные структуры данных выражают более сложные отношения порядка между объектами, чем отношения предшествования и следования. Наиболее важным видом нелинейных структур являются **деревья**. Древовидные структуры позволяют определить такие отношения, как предок, потомок, брат и т.п.

Дерево (непустое) – конечное множество объектов *T*, состоящее из одного или более узлов, для которых выполняются следующие условия:

- имеется один специально выделенный узел, называемый корнем данного дерева;
- остальные узлы (исключая корень) содержатся в *m* попарно непересекающихся множествах *T1, …,Tm,* каждое из которых в свою очередь является деревом. Деревья *T1, ...,Tm* называются ***поддеревьями*** данного корня. Структура дерева общего вида представлена на рис. 14.

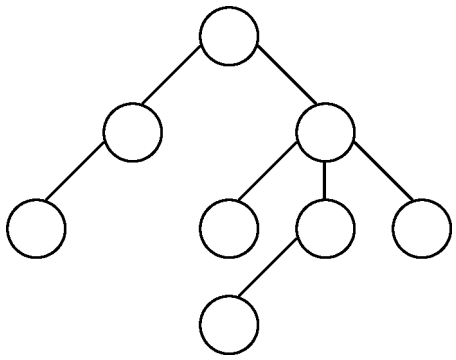


Рис. 14. Дерево общего вида

Пустым называется дерево, не содержащее узлов.

Число поддеревьев данного корня (узла) называется ***степенью*** этого ***узла***. Максимальная степень всех узлов дерева называется ***степенью дерева*** (обозначим *d*). Узел с нулевой степенью называется ***терминальным узлом или листом***. ***Уровень узла*** – выраженная в числе ребер длина пути, ведущего из узла в корень дерева плюс единица. Считается, что корень дерева находится на уровне 1. Если некоторый узел A располагается на уровне *i*, то находящийся непосредственно ниже его на уровне *i+1* узел B называется ***непосредственным потомком*** узла A, а узел A называется ***непосредственным предком*** узла B. Максимальный уровень всех узлов дерева называется ***высотой или глубиной дерева*** (обозначим *h*). Высота пустого дерева равна нулю, высота дерева, состоящего из одного корня, равна 1. Дерево, содержащее максимальное число узлов, называется ***полным***. В полном дереве у всех узлов, за исключением терминальных, число

непосредственных потомков равно степени дерева. Количество узлов в полном дереве степени d высотой h вычисляется по формуле (i – номер уровня без единицы)

$$N_d(h) = \sum_{i=0}^{h-1} d^i .$$

Основные свойства деревьев общего вида:

- корень не имеет предков;
- каждый узел, за исключением корня, имеет только одного предка;
- каждый узел связан с корнем единственным путем, т.е. в деревьях отсутствуют замкнутые контуры (циклы).

Если в определении дерева имеет значение относительный порядок поддеревьев T1,…,Tm, дерево является ***упорядоченным***. Поэтому два упорядоченных дерева на рис. 15 – это разные, отличные друг от друга объекты.

Рис. 15. Два различных дерева

3.2. Бинарные деревья

Особенно важное практическое значение имеют упорядоченные деревья второй степени. Их называют ***двоичными или бинарными деревьями***. ***Бинарное дерево*** – это конечное множество узлов, которое или пусто, или состоит из корня и двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня. Примеры бинарных деревьев приведены на рис. 16.

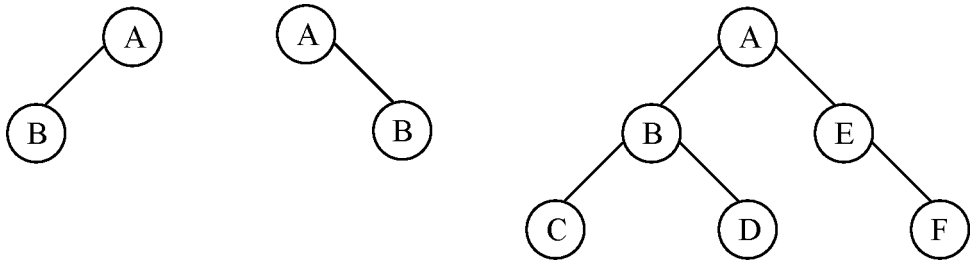


Рис. 16. Примеры бинарных деревьев

Максимальное число узлов в бинарном дереве высотой h *(d=2)*:

$$N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1 .$$

Максимальное число узлов на уровне i в бинарном дереве:

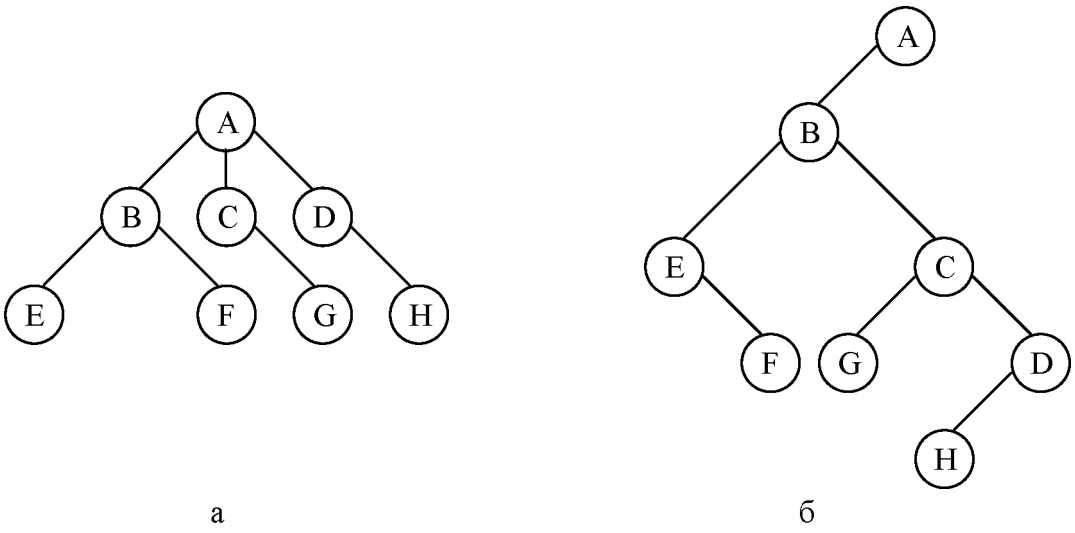
$$n_i = 2^{i-1}.$$

Высота полного бинарного дерева, содержащего N узлов:

$$h = \lfloor \log_2 N \rfloor + 1,$$

где $\lfloor\ \rfloor$ означает взятие целой части числа.

Существуют различные алгоритмы преобразования дерева произвольной степени, т.е. дерева общего вида, к виду бинарного. ***Левосторонний алгоритм*** формулируется следующим образом: у каждого узла дерева произвольной степени необходимо сохранить самую левую связь, а узлы – потомки одного и того же узла следует соединить правой связью. На рис. 17.а представлено исходное дерево общего вида, а на рис. 17.б – эквивалентное бинарное.



а б

Рис. 17. Преобразование произвольного дерева к виду бинарного:
а – произвольное дерево; б – бинарное дерево

3.3. Представление бинарных деревьев

3.3.1. Представление бинарных деревьев в памяти с последовательной организацией

Если известен максимальный размер дерева (т.е. высота, а следовательно, и количество узлов, соответствующих полному дереву), то структуру дерева можно хранить в виде массива. При этом корень дерева будет храниться в элементе массива с индексом [1]. Для каждого узла с номером k его левый потомок будет храниться в элементе с индексом [2 * k], а правый - в элементе с индексом [2 * k + 1] (см. рис. 18).

Достоинством представления бинарных деревьев на статической памяти является простота доступа как от предка к потомку, так и от потомка к предку, а недостатком – то, что если дерево не является полным, в массиве появляется большое число пустых элементов. Ограниченный размер массива не позволяет включать в дерево новые узлы, т.е. дерево не может “расти”. Удаление узла из дерева и соответствующее изменение его структуры потребует модификации содержимого массива.

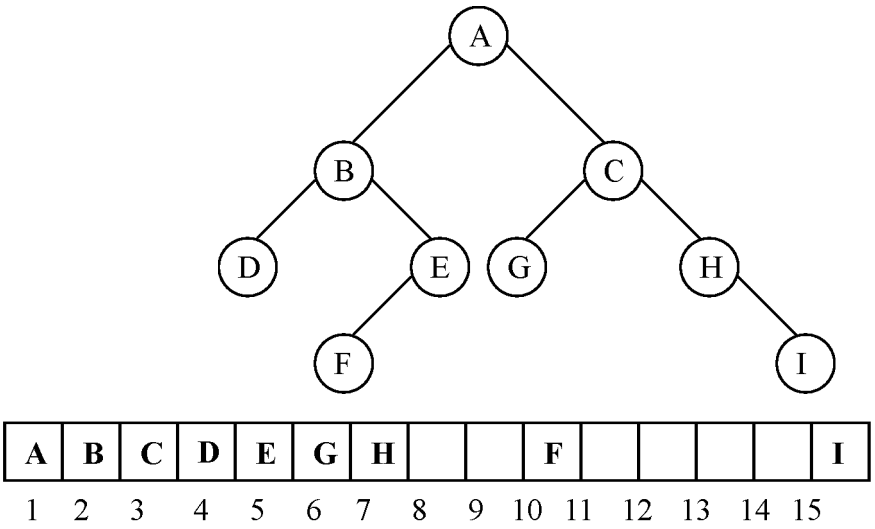


Рис. 18. Представление бинарного дерева в статической памяти

3.3.2. Связанное представление бинарных деревьев в динамической памяти

Элемент хранения узла бинарного дерева состоит из одного или нескольких информационных полей и двух полей связи, указывающих соответственно на левое и правое поддеревья данного узла:

```
Type
   PTree = ^ Tree;                    { тип – указатель на узел дерева }
   Tree = record                      { тип – элемент хранения узла дерева}
     info: char;                      { информационное поле }
     left, right: PTree               { ссылки на поддеревья }
   end;
Var Root: PTree;                      { указатель на корень дерева }
```

Дерево задается указателем на его корень. Если дерево пусто, указатель на его корень равен *NIL*. Связанное представление бинарного дерева **произвольного вида** иллюстрирует рис. 19. Бинарное дерево произвольного вида создается, начиная с корня, с помощью последовательного включения узлов либо в левое, либо в правое поддерево корневого узла.

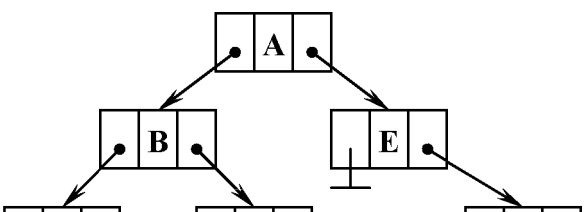


Рис. 19. Представление бинарного дерева в динамической памяти

3.4. Алгоритмы обхода бинарных деревьев

Наиболее типичная операция, выполняемая над бинарными деревьями – обход дерева. ***Обход*** – это процедура, при выполнении которой каждый узел обрабатывается ровно один раз одинаковым образом. Обход дерева заключается в разбиении дерева на корень, левое и правое поддеревья и применении к каждому из поддеревьев соответствующей процедуры обработки до тех пор, пока в процессе разбиения не будет получено пустое дерево. Полный обход дерева дает линейную расстановку узлов, что облегчает выполнение многих алгоритмов. Существует несколько принципов упорядочения, которые естественно вытекают из структуры дерева. Как и саму древовидную структуру, их удобно выразить с помощью рекурсии. Различным принципам упорядочения соответствуют три левосторонних алгоритма обхода (и три симметричных правосторонних алгоритма): нисходящий, восходящий, смешанный, а также обход в ширину.

3.4.1. Нисходящий, восходящий и смешанный обходы

Рассмотрим три ***левосторонних*** алгоритма обхода.

Нисходящий (прямой) обход выполняется согласно алгоритму ***“корень-левый-правый” (К-Л-П)***:

1. обработать корневой узел;
2. обойти левое поддерево в нисходящем порядке;
3. обойти правое поддерево в нисходящем порядке.

Трасса нисходящего обхода дерева рис. 19: *A B* ***C D*** *E* ***F***.

Восходящий (концевой) обход выполняется согласно алгоритму ***“левый-правый-корень” (Л-П-К)***:

1. обойти левое поддерево в восходящем порядке;
2. обойти правое поддерево в восходящем порядке;
3. обработать корневой узел.

Трасса восходящего обхода дерева рис. 19: ***C D*** *B* ***F*** *E A* .

Смешанный (обратный) обход выполняется согласно алгоритму ***“левый-корень-правый” (Л-К-П)***.

1. обойти левое поддерево в смешанном порядке;
2. обработать корневой узел;
3. обойти правое поддерево в смешанном порядке.

Трасса смешанного обхода дерева рис. 19: ***C*** *B* ***D*** *A E* ***F*** .

Заметим, что при различных алгоритмах обхода порядок обработки терминальных узлов не изменяется.

Ниже приведена процедура, распечатывающая трассу нисходящего обхода бинарного дерева – последовательность информационных полей узлов дерева.

```
Procedure KLP( root: PTree );                { root – указатель на корень дерева }
          begin
            if  root <> nil then begin                    { дерево не пусто? }
              writeln( root^.info );          { распечатать информ. поле корневого узла }
              KLP( root^.left );          { обойти левое поддерево в нисходящем порядке }
(* 1 *)       KLP( root^.right )         { обойти правое поддерево в нисходящем порядке }
(* 2 *)     end;
(* 3 *)    end;
```

Для иллюстрации работы этой процедуры и построения трассы состояний стека будем использовать следующие обозначения:

^A – адрес узла дерева, в информационном поле которого хранится символ “A”;

(* 1 *) и (* 2 *) – адреса возврата из уровня рекурсивной процедуры, номер которого на 1 больше текущего уровня;

(* 3 *) – адрес возврата из текущего уровня рекурсивной процедуры.

Трасса нисходящего обхода дерева, приведенного на рис. 19, содержится в таблице 2.

Таблица 2.

Трасса состояний стека
при работе процедуры нисходящего обхода бинарного дерева

Номер входа в процедуру	Номер уровня рекурсии	Значение фактического параметра	Стек		Выходная трасса	Выход из уровня
			Исходное состояние	Конечное состояние		
1	1	^A	(-,-)	(^A, *1*)	A	
2	2	^B	(^A, *1*)	(^B, *1*) (^A, *1*)	B	
3	3	^C	(^B, *1*) (^A, *1*)	(^C, *1*) (^B, *1*) (^A, *1*)	C	
4	4	NIL	(^C, *1*) (^B, *1*) (^A, *1*)	(NIL,*1*) (^C, *1*) (^B, *1*) (^A,*1*)		
	4	NIL	(NIL,*1*) (^C, *1*) (^B, *1*) (^A,*1*)	(^C, *1*) (^B, *1*) (^A, *1*)		(*3*)

Номер входа в процедуру	Номер уровня рекурсии	Значение фактического параметра	Исходное состояние стека	Конечное состояние стека	Выходная трасса	Выход из уровня
4	3	^C	(^C, *1*) (^B, *1*) (^A, *1*)	(^C, *2*) (^B, *1*) (^A, *1*)		
5	4	NIL	(^C, *2*) (^B, *1*) (^A, *1*)	(NIL,*2*) (^C, *2*) (^B, *1*) (^A, *1*)		
	4	NIL	(NIL,*2*) (^C, *2*) (^B, *1*) (^A, *1*)	(^C, *2*) (^B, *1*) (^A, *1*)		(*3*)
	3	^C	(^C, *2*) (^B, *1*) (^A,*1*)	(^B, *1*) (^A, *1*)		(*3*)
	2	^B	(^B, *1*) (^A,*1*)	(^B, *2*) (^A, *1*)		
6	3	^D	(^B,*2*) (^A,*1*)	(^D, *1*) (^B, *2*) (^A, *1*)	D	
7	4	NIL	(^D, *1*) (^B, *2*) (^A, *1*)	(NIL,*1*) (^D, *1*) (^B, *2*) (^A, *1*)		
	4	NIL	(NIL,*1*) (^D, *1*) (^B, *2*) (^A, *1*)	(^D, *1*) (^B, *2*) (^A, *1*)		(*3*)
	3	^D	(^D, *1*) (^B, *2*) (^A, *1*)	(^D, *2*) (^B, *2*) (^A, *1*)		
8	4	NIL	(^D, *2*) (^B, *2*) (^A, *1*)	(NIL,*2*) (^D, *2*) (^B, *2*) (^A, *1*)		
	4	NIL	(NIL,*2*) (^D, *2*) (^B, *2*) (^A, *1*)	(^D, *2*) (^B, *2*) (^A, *1*)		(*3*)
	3	^D	(^D, *2*) (^B, *2*) (^A, *1*)	(^B, *2*) (^A, *1*)		(*3*)
	2	^B	(^B, *2*) (^A, *1*)	(^A, *1*)		(*3*)
	1	^A	(^A, *1*)	(^A, *2*)		
9	2	^E	(^A, *2*)	(^E, *1*) (^A, *2*)	E	
Номер	**Номер**	**Значение**	**Исходное**	**Конечное**	**Выход-**	**Выход из**

входа в процедуру	уровня рекурсии	фактического параметра	состояние стека	состояние стека	ная трасса	уровня
10	3	NIL	(^E, *1*) (^A, *2*)	(NIL,*1*) (^E, *1*) (^A, *2*)		
	3	NIL	(NIL,*1*) (^E, *1*) (^A, *2*)	(^E, *1*) (^A, *2*)		(*3*)
	2	^E	(^E, *1*) (^A, *2*)	(^E, *2*) (^A, *2*)		
11	3	^F	(^E, *2*) (^A, *2*)	(^F, *1*) (^E, *2*) (^A, *2*)	F	
12	4	NIL	(^F, *1*) (^E, *2*) (^A, *2*)	(NIL, *1*) (^F, *1*) (^E, *2*) (^A, *2*)		
	4	NIL	(NIL, *1*) (^F, *1*) (^E, *2*) (^A, *2*)	(^F, *1*) (^E, *2*) (^A, *2*)		(*3*)
12	3	^F	(^E, *2*) (^A, *2*)	(^F, *2*) (^E, *2*) (^A, *1*)		
13	4	NIL	(^F, *2*) (^E, *2*) (^A, *2*)	(NIL, *2*) (^F, *2*) (^E, *2*) (^A, *2*)		
	4	NIL	(NIL, *2*) (^F, *2*) (^E, *2*) (^A, *2*)	(^F, *2*) (^E, *2*) (^A, *2*)		(*3*)
	3	^F	(^F, *2*) (^E, *2*) (^A, *2*)	(^E, *2*) (^A, *2*)		(*3*)
	2	^E	(^E, *2*) (^A, *2*)	(^A, *2*)		(*3*)
	1	^A	(^A, *2*)	(-, -)		(*3*)

3.4.1. Обход в ширину

При обходе в ширину узлы посещаются уровень за уровнем, начиная с корня, причем каждый уровень обходится слева направо.

Для реализации данного алгоритма используется структура очереди, в которой хранятся указатели на поддеревья каждого узла. Для структуры очереди определены операции: In_Queue – включить в очередь; Out_Queue – исключить из очереди. Ниже приведена схема алгоритма обхода дерева в ширину. Q – указатель на голову очереди, Root – указатель на корень дерева:

```
…
 In_Queue( Q, Root );                { Занести в очередь указатель на корень дерева }
```

```
while ( Q<>nil ) do begin          { Выполнять до тех пор, пока очередь не пуста }
   Out_Queue( Q, p );              { P – указатель на узел, извлеченный из очереди }
   Work( p );                      { Обработать узел P^ }
   if  p^.left <> nil then In_Queue( p^.left );   { Если узел P^ имеет левого
                                   потомка, занести в очередь указатель на левого потомка }
   if  p^.right <> nil then In_Queue( p^.right )  { Если узел P^ имеет правого
                                   потомка, занести в очередь указатель на правого потомка }
end;
```

Трасса обхода в ширину дерева рис. 67: *A B E* ***C D F***.

3.5. Виды бинарных деревьев

3.5.1. Сбалансированные деревья

Дерево называется ***идеально сбалансированным***, если для каждой вершины число узлов в ее правом и левом поддеревьях отличается не более чем на единицу (далее будем называть такие деревья сбалансированными). ***Сбалансированное дерево***, состоящее из N узлов, имеет минимальную высоту среди всех бинарных деревьев. Высоту сбалансированного дерева можно определить по формуле:

$$h = \lfloor \log_2 N \rfloor + 1.$$

Минимальная высота при заданном числе узлов достигается, если на всех уровнях, кроме последнего, размещается максимально возможное число узлов. Это происходит за счет равномерного размещения узлов поровну слева и справа от каждого узла.

Правило равномерного размещения для N узлов (*правило 1*) формулируется с помощью рекурсии:

- взять один узел в качестве корня;
- по *правилу 1* построить левое поддерево с числом узлов *nl = N div 2;*
- по *правилу 1* построить правое поддерево с числом узлов *nr = N - nl - 1.*

Структура сбалансированного дерева из N узлов определяется количеством узлов (рис. 20).

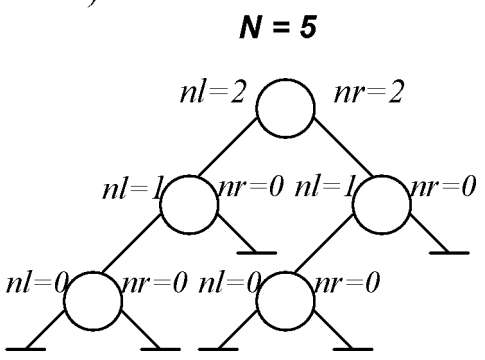


Рис. 20. Сбалансированное дерево

Процедура построения сбалансированного дерева из N узлов:

```
Type
 PTree = ^ Tree;                    { тип – указатель на узел сбалансированного дерева }
 Tree = record                      { тип – элемент хранения узла сбалансированного дерева }
  info: char;
  left, right: PTree
 end;

Procedure Balance( var root: PTree;          { root – указатель на корень дерева, }
                   n: byte );                  { n – количество узлов в дереве }
begin
 if  n = 0 then root:=nil                    { если n = 0, построить пустое дерево }
 else begin
      new( root );                           { создать корень дерева }
      write( 'Значение инф. поля', i, '-го узла дерева = ' );
      readln( root^.info );                  { заполнить информационное поле корня }
      Balance( root^.left, n div 2);         { построить левое поддерево}
(*1*) Balance( root^.right, n – n div 2 – 1) { построить правое поддерево }
(*2*) end
(*3*)end;
```

Для иллюстрации работы этой процедуры и построения трассы состояний стека будем использовать следующие обозначения:

^A – адрес узла дерева, в информационном поле которого хранится символ "A";

(* 1 *) и (* 2 *) – адреса возврата из уровня рекурсивной процедуры, номер которого на 1 больше текущего уровня;

(* 3 *) – адрес возврата из текущего уровня рекурсивной процедуры.

Трасса построения сбалансированного дерева, содержащего 3 узла, приведена в таблице 3.

Сбалансированное дерево, как и дерево любого другого вида, можно построить в процессе нисходящего обхода. Особенность работы процедуры построения дерева заключается в том, что сначала создаются корневые узлы, адреса которых запоминаются в стеке. Установить ссылки из корневых узлов дерева на их поддеревья можно только после того, как эти поддеревья будут созданы и адреса их корневых узлов возвращены в точки вызова. Точками вызова являются поля ссылок на левое поддерево (*root^.left*) или правое поддерево (*root^.right*) корневого узла (в таблице установка этих ссылочных связей показана стрелками). Подобная возможность обеспечивается за счет того, что указатель на корень дерева является параметром-переменной процедуры.

Таблица 3.

Трасса состояний стека при работе процедуры построения сбалансированного дерева

Номер уровня рекур-сии	Знач-я входного парам-ра n	Стек		Знач-я вых. пар-ра			Выход из уровня
		Исходное состояние (n,root, (.) возврата)	Конечное состояние (n,root, (.) возврата)	root	root^. left	root^. right	
1	3	(-, -, -)	(3,^A, *1*)	^A	-	-	-
2	1	(3,^A, *1*)	(1, ^B, *1*) (3, ^A, *1*)	^B	-	-	-
3	0	(1, ^B, *1*) (3, ^A, *1*)	(0,NIL,*1*) (1, ^B, *1*) (3, ^A, *1*)	NIL	-	-	-
	0	(0,NIL,*1*) (1, ^B, *1*) (3, ^A, *1*)	(1, ^B, *1*) (3, ^A, *1*)	NIL	-	-	(* 3 *)
2	1	(1, ^B, *1*) (3, ^A, *1*)	(1, ^B, *2*) (3, ^A, *1*)	^B	NIL	-	-
3	0	(1, ^B, *2*) (3, ^A, *1*)	(0,NIL,*2*) (1, ^B, *2*) (3, ^A, *1*)	NIL	-	-	-
	0	(0,NIL,*2*) (1, ^B, *2*) (3, ^A, *1*)	(1, ^B, *2*) (3, ^A, *1*)	NIL	-	-	(* 3 *)
2	1	(1, ^B, *2*) (3, ^A, *1*)	(3, ^A, *1*)	^B	NIL	NIL	(* 3 *)
1	3	(3, ^A, *1*)	(3, ^A, *2*)	^A	^B	-	-
2	1	(3,^A, *2*)	(1, ^C, *1*) (3, ^A, *2*)	^C	-	-	-
3	0	(1, ^C, *1*) (3, ^A, *2*)	(0,NIL,*1*) (1, ^C, *1*) (3, ^A, *2*)	NIL	-	-	-
	0	(0,NIL,*1*) (1, ^C, *1*) (3, ^A, *2*)	(1, ^C, *1*) (3, ^A, *2*)	NIL	-	-	(* 3 *)
2	1	(1, ^C, *1*) (3, ^A, *2*)	(1, ^C, *2*) (3, ^A, *2*)	^C	NIL		
3	0	(1, ^C, *2*) (3, ^A, *2*)	(0,NIL,*2*) (1, ^C, *2*) (3, ^A, *2*)	NIL	-	-	-
	0	(0,NIL,*2*) (1, ^C, *2*) (3, ^A, *2*)	(1, ^C, *2*) (3, ^A, *2*)	NIL	-	-	(* 3 *)
2	1	(1, ^C, *2*) (3, ^A, *2*)	(3, ^A, *2*)	^C	NIL	NIL	(* 3 *)
1	3	(3, ^A, *2*)	(-, -, -)	^A	^B	^C	(* 3 *)

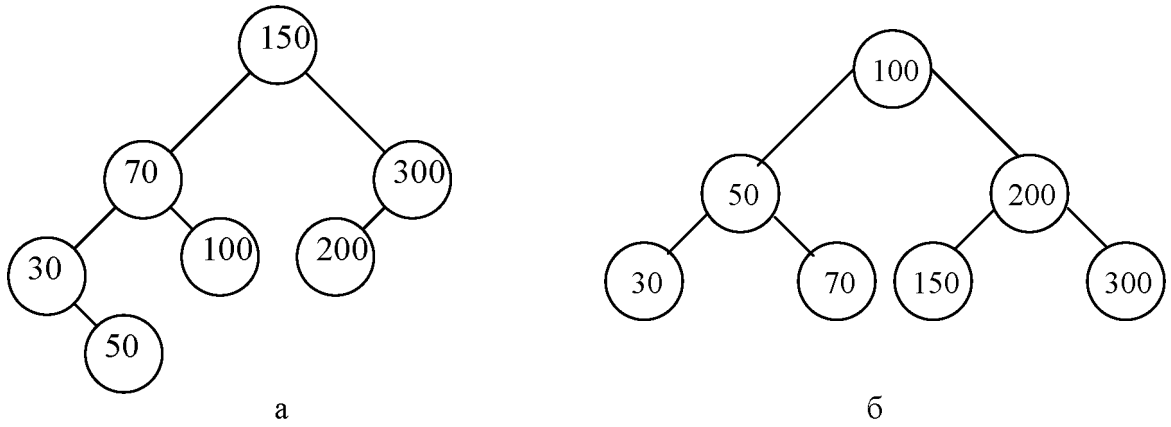
3.5.2. Дихотомические деревья (деревья поиска)

Бинарные деревья часто используются для представления множества объектов, среди которых идет поиск по уникальному значению некоторого атрибута, называемого ***ключом***. При этом на множестве объектов определено особое отношение порядка – ***дихотомия***, заключающееся в поэтапном разбиении множества информационных полей объектов на два непересекающихся подмножества. ***Дихотомическим деревом (деревом поиска)*** называется бинарное дерево, организованное так, что для каждого узла, имеющего ключ T, справедливо утверждение о том, что в его левом поддереве содержатся узлы с ключами, меньшими T, а в его правом поддереве содержатся узлы с ключами, большими T.

Описание элемента хранения узла дихотомического дерева:

```
Type
   PDtree = ^ Dtree;          { тип – указатель на узел дихотомического дерева }
   Dtree = record             { тип – элемент хранения узла дихотомического дерева}
     key: word;                                                  { поле ключа }
     info: char;                                        { информационное поле }
     left, right: PDtree                                { ссылки на поддеревья }
   end;
Var Root: PDtree;             { указатель на корень дихотомического дерева }
```

Пример дихотомических деревьев приведен на рис. 21:



а б

Рис. 21. Дихомические деревья:
а – несбалансированное; б – сбалансированное

Поиск в дихотомическом дереве узла с заданным значением ключа осуществляется на основе сравнения заданного ключа с ключом корня. Единственное сравнение позволяет перейти к левому или к правому поддереву корня. Если дихотомическое дерево является сбалансированным, то поиск узла с заданным значением ключа требует не более чем $[log_2 N]+1$ сравнений, где N – количество узлов дерева. В частном случае, когда множество ключей является линейно упорядоченным, дихотомическое дерево фактически вырождается в линейный список (рис. 22). При этом

поиск среди N узлов выполняется максимум за N сравнений, а в среднем – за $N / 2$ сравнений:

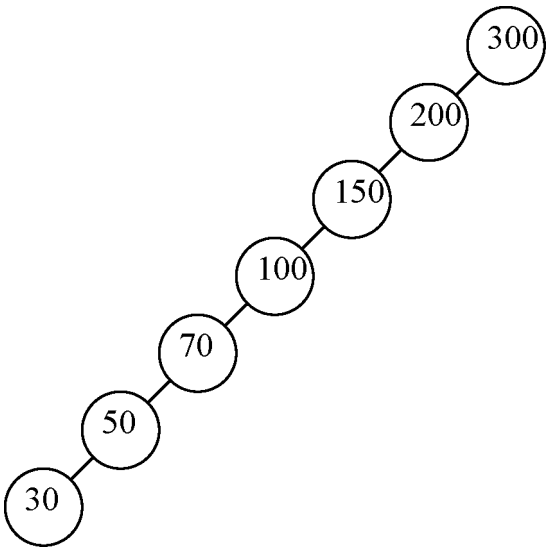


Рис. 22. Вырожденное дихотомическое дерево

Структура дихотомического дерева определяется последовательностью задания ключей. Последовательности задания ключей для

- дерева рис. 21 а) – 150, 70, 300, 100, 30, 200, 50;
- дерева рис. 21 б) – 100, 50, 30, 70, 200, 150, 300;
- дерева рис. 22 – 300, 200, 150, 100, 70, 50, 30.

Включение в дихотомическое дерево узла с заданным значением ключа производится следующим образом: если поиск узла привел в тупик (т.е. к пустому поддереву, обозначенному ссылкой со значением *NIL*), то новый узел необходимо включить в дерево на место пустого поддерева. Таким образом, узел включается в дихотомическое дерево всегда в качестве листа:

```
Procedure Ins_Node( var root: PDtree;          { root – указатель на корень дерева }
                    k: word );                 { k – ключ вставляемого узла }
begin
  if  root = nil then begin                    { дерево пусто – вставка узла в качестве листа }
    new( root );                               { создать элемент хранения узла }
    readln( root^. Info);                      { заполнить информационное поле узла }
    root^.key:=k;                              { заполнить поле ключа }
    root^.left:=nil;   root^.right:=nil;       { заполнить ссылки на поддеревья }
  end
  else                                         { дерево не пусто – продолжить поиск: }
  if  k < root^.key then Ins_Node( root^.left,k )          { поиск в левом поддереве }
  else if  k > root^.key then Ins_node( root^.right,k )    { поиск в правом поддереве}
  else writeln('узел с ключом ', k, 'уже есть в дереве' )  { ключ должен быть }
end;                                           { уникальным }
```

Если корневой узел не содержит искомого ключа, процедура рекурсивно вызывает саму себя для продолжения поиска либо в правой, либо в левой половине дерева. В том случае, если достигнут конец ветви и не обнаружен искомый ключ, создается новый узел с этим значением ключа. Рекурсия заканчивается, когда искомый ключ найден (при этом выдается сообщение о невозможности повторного включения узла) или достигнут конец ветви (при этом новый узел включается в дерево).

??? *Какой алгоритм обхода лежит в основе процедуры включения узла в дихотомическое дерево?*

При ***построении дихотомического дерева*** из N узлов на каждом из N шагов цикла осуществляется вставка узла с заданным значением ключа (вызов процедуры *Ins_Node*). Поэтому сложность создания дихотомического дерева из N узлов оценивается как $N * log_2 N$ операций**.**

При ***удалении узла***, с заданным значением ключа из дихотомического дерева различают три случая:

1. узла с заданным значением ключа в дереве нет;
2. узел с заданным значением ключа имеет не более одного потомка. В этом случае исключаемый узел заменяется своим потомком;
3. узел с заданным значением ключа имеет двух потомков. В этом случае исключаемый узел заменяется либо на *самый правый узел его левого поддерева*, имеющий не более одного потомка, либо на *самый левый узел его правого поддерева*, имеющий не более одного потомка.

Пример исключения узлов из дихотомического дерева приведен на рис. 23.

Разрушение бинарного дерева любого вида заключается в последовательном освобождении участков памяти, в которых размещены элементы хранения узлов с возвратом их в кучу. В результате разрушения дерево становится пустым.

??? *Какой алгоритм обхода необходимо использовать для разрушения бинарного дерева?*

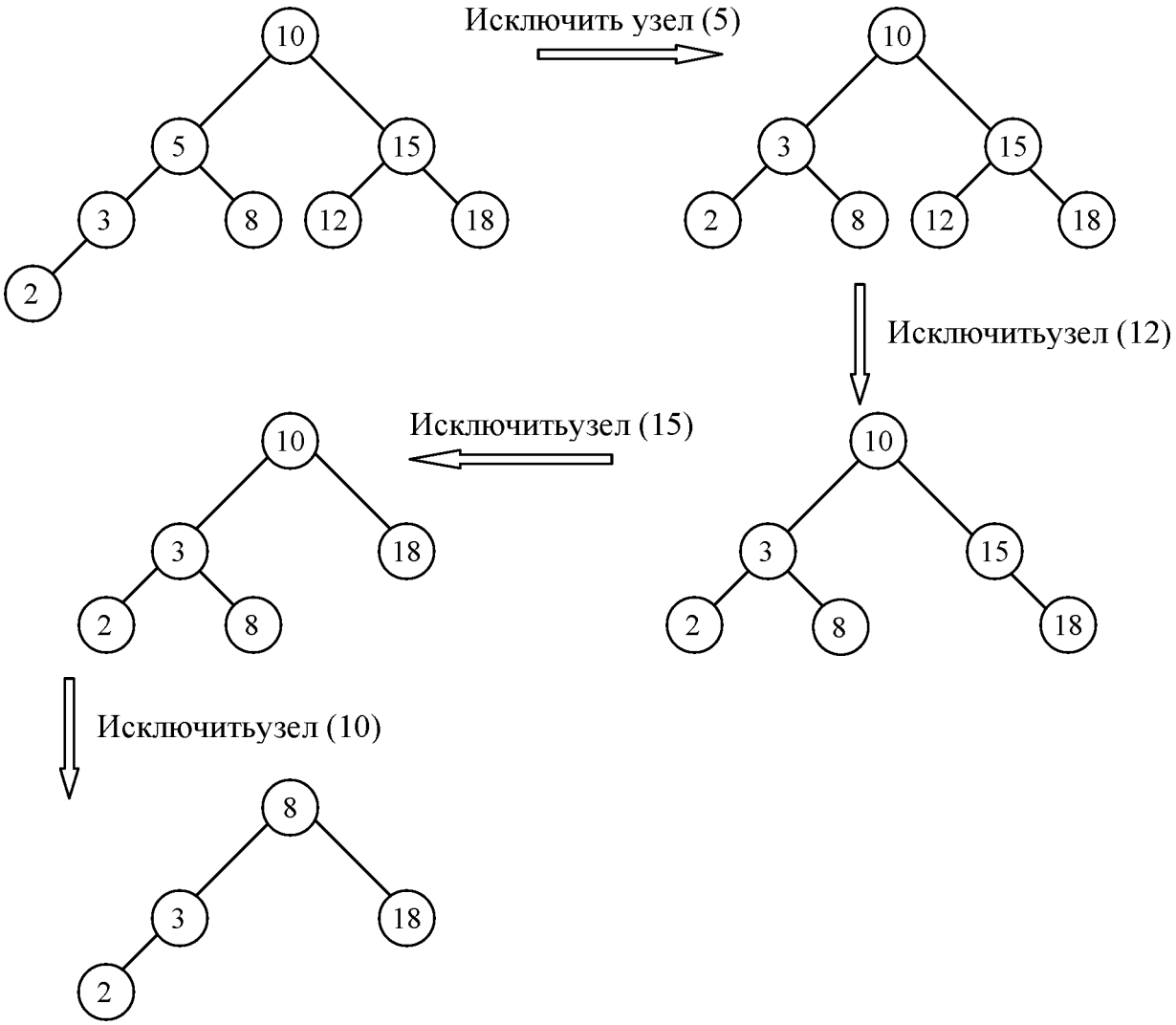


Рис. 23. Исключение узлов из дихотомического дерева

3.5.3. Деревья выражений

Дерево выражений – бинарное дерево, в корневых узлах которого хранятся признаки операций, а в терминальных узлах – операнды выражения (переменные или константы). Дерево выражений представлено на рис. 24.

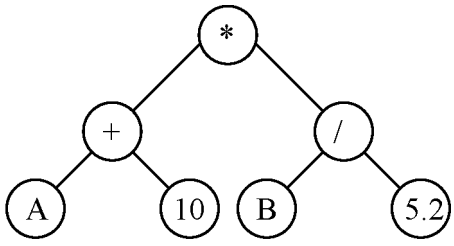


Рис. 24. Дерево выражений

Различные алгоритмы обхода дерева выражений соответствуют различной структуре представления выражения в виде строки.

Нисходящему обходу соответствует ***префиксная форма*** представления, т. к. в ней знак операции предшествует операнду:

** + A 10. / B 5.2 .*

Восходящему обходу соответствует ***постфиксная форма*** представления, т. к. в ней знак операции находится после операндов:

*A 10. + B 5.2 / * .*

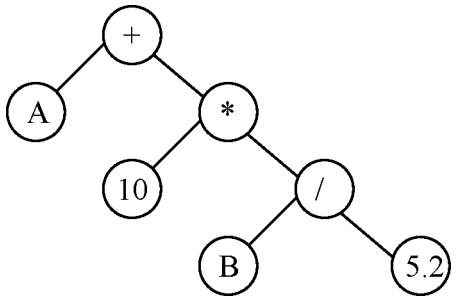
Смешанному обходу соответствует ***инфиксная форма*** представления, т. к. в ней знак операции находится между операндами:

*A + 10. * B / 5.2 .*

Для того чтобы задать приоритеты операций, используется абсолютно скобочная форма, в которой каждое подвыражение заключается в круглые скобки:

*((A + 10.) * (B / 5.2)) .*

На рис. 25 приведено дерево, смешанный обход которого позволяет получить бесскобочную инфиксную форму, эквивалентную дереву рис. 24, а скобочная инфиксная форма задает совсем другие приоритеты операций. Заметим, что подобная проблема не может возникнуть при использовании префиксной или постфиксной формы представления выражения.



*(A + (10. * (B / 5.2)))*

Рис. 25. Дерево выражений

??? *Какой алгоритм обхода необходимо использовать для того, чтобы подсчитать значение арифметического выражения, представленного в виде дерева?*

3.6. Демонстрационная программа, реализующая операции создания, обработки, просмотра содержимого бинарного дерева (на примере сбалансированного дерева)

```
Uses Crt;
Type
 PTree = ^ Tree;                    { тип – указатель на узел сбалансированного дерева }
 Tree = record                      { тип – элемент хранения узла сбалансированного дерева }
  info: word;
  left, right: PTree
 end;

var f: PTree; cod,n: byte; sum: word;

Procedure Balance( var root: PTree;          { root – указатель на корень дерева, }
                   n: byte );                  { n – количество узлов в дереве }
begin
 if  n = 0 then root:=nil                    { если n = 0, построить пустое дерево }
 else begin
       new( root );                          { создать корень дерева }
       write( 'Значение информационного поля узла дерева =  ' );
       readln( root^.info );                 { заполнить информационное поле корня }
       Balance( root^.left, n div 2);        { построить левое поддерево}
       Balance( root^.right, n – n div 2 – 1)  { построить правое поддерево }
 end
end;

Procedure Print( root: PTree );              { просмотр информац. полей узлов дерева - }
begin                                        { нисходящий обход }
  if  root <> nil then begin
      writeln( 'Информационное поле узла дерева =  ', root^.info );
      Print( root^.left );                   { просмотреть левое поддерево }
      Print( root^.right );                  { просмотреть правое поддерево }
  end;
end;

Procedure Work( root: PTree; var s: word );  { суммирование значений информ. }
begin                                        { полей узлов дерева – смешанный обход }
  if  root <> nil then begin
    Work( root^.left,s );                    { обработать левое поддерево }
    s:=s + root^.info;                       { суммирование значений инф. полей  узлов }
    Work( root^.right,s );                   { обработать правое поддерево }
  end;
end;
Procedure Destroy( var root: PTree );        { разрушение дерева }
```

```
begin
   if  root <> nil then begin
      Desrtoy(root^.left); Desrtoy(root^.right);
      dispose(root); root:=nil
end;

Procedure Message;                                    { вспомогательная процедура }
begin
   writeln( ‘Дерево пусто‘ ); write( ‘Нажмите любую клавишу‘ ); readkey
end;

begin
   f:=nil;                                            { первоначально дерево пусто }
   repeat Clrscr;
     writeln(‘1-Создание 2-Просмотр 3–Обработка 4–Разрушение 5-Выход‘);
     write( ‘Код действия = ‘ ); readln( cod  );
     case cod of
     1: begin                                  { создание сбалансированного дерева }
          write( ‘Количество узлов в дереве = ‘ ); readln( n );
          Balance( f,n ); write( ‘Нажмите любую клавишу‘ ); readkey
        end;
     2:                                                  { просмотр дерева }
          if f=nil then Message
          else  begin
            Print( f  ); write( ‘Нажмите любую клавишу‘ ); readkey
          end;
     3:                                                  { обработка дерева }
          if f=nil then Message
          else  begin
            sum:=0;                  { инициализация значения глобальной переменной }
            Work( f,sum ); writeln( ‘Сумма значений инф. полей =  ‘, sum );
            write( ‘Нажмите любую клавишу‘ ); readkey
          end;
     4:                                                  { разрушение дерева }
          if f=nil then Message
          else  begin
            Destroy( f ); writeln( ‘Дерево разрушено‘ );
            write( ‘Нажмите любую клавишу‘ ); readkey
          end;
     5: Destroy( f  )                                    { выход }
     end;
   until ( cod = 5 ); Clrscr
end.
```

3.7. Контрольные вопросы к главе 3

1. Дайте определение дерева общего вида.

2. Что такое степень дерева и глубина дерева?

3. Перечислите свойства деревьев общего вида.

4. Дайте определение бинарного дерева.

5. Сформулируйте алгоритм преобразования дерева произвольного вида к виду бинарного дерева.

6. Каким образом бинарное дерево представляется в памяти с последовательной и связанной организацией?

7. Какие алгоритмы обхода бинарных деревьев Вы знаете?

8. Дайте определение сбалансированного дерева. В чем отличительные особенности сбалансированных деревьев? Сформулируйте алгоритм построения сбалансированного дерева.

9. Дайте определение дихотомического дерева. Приведите примеры применения дихотомических деревьев.

10. Сформулируйте алгоритм создания дихотомического дерева.

11. Сформулируйте алгоритм исключения узла из дихотомического дерева.

12. Дайте определение дерева выражений.

13. Какой алгоритм обхода необходимо использовать для вычисления значения выражения, представленного в виде дерева?

14. Какой алгоритм обхода необходимо использовать для разрушения дерева?

3.8. Упражнения к главе 3

1. Создать сбалансированное дерево. Подсчитать количество узлов дерева с положительными и отрицательными значениями информационных полей.

2. Создать сбалансированное дерево. Подсчитать количество узлов дерева с заданным значением информационных полей.

3. Создать дерево поиска. Подсчитать сумму значений информационных полей узлов дерева.

4. Создать два дерева поиска. Скопировать в линейный список информационные поля элементов с совпадающими в первом и втором деревьях значениями ключей.

5. Создать дерево поиска. Построить копию этого дерева.

6. Создать дерево поиска. Скопировать в линейный список узлы дерева, выбранные по заданным значениям ключей.

7. Создать дерево поиска. Скопировать в новое дерево поиска узлы дерева, выбранные по заданным значениям ключей.

8. Создать два сбалансированных дерева. Определить эквивалентность двух деревьев.

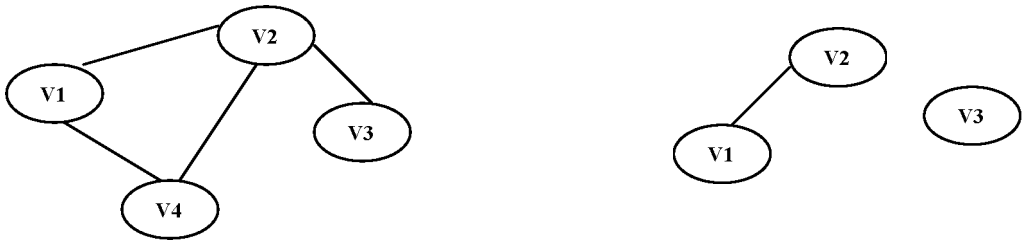
9. Создать дерево поиска. Определить глубину дерева.

10. Создать дерево поиска. Написать процедуру исключения произвольного узла из дерева.

4. ИЕРАРХИЧЕСКИЕ НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ. ГРАФЫ

4.1. Основные понятия и определения

Граф G = (V, E) включает конечное множество вершин V и конечное множество ребер E, соединяющих эти вершины. Любые две вершины, соединенные в графе ребром, называются смежными. Ребро графа, направленное от одной вершины к другой, называется ориентированным. Ребро, не имеющее определенного направления, является неориентированным. Граф, в котором все ребра ориентированы, называется ***орграфом***. Ребро графа, соединяющее некоторую вершину саму с собой, называется петлей. Граф, в котором между любой парой вершин существует не более одного ребра (не более одного ребра данного направления для орграфа), называется ***простым***. Примеры неориентированных и ориентированных графов приведены на рис. 26 и 27.



а б

Рис. 26. Неориентированные графы
а – граф G1; б – граф G2

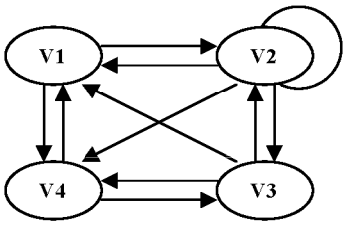


Рис. 27. Ориентированный граф G3

Пусть G = (V, E) – ***простой граф*** (простой орграф). Любая последовательность ребер графа (ориентированных ребер орграфа) такая, что конечная вершина любого ребра этой последовательности является начальной вершиной следующего ребра, задает ***путь*** P в графе. Число ребер в последовательности, задающей путь, является ***длиной пути***.

$P = ((v_1, v_2), (v_2, v_3), \ldots, (v_{K-2}, v_{K-1}), (v_{K-1}, v_K))$ – путь в графе.

$P = (v_1, v_2, \ldots, v_{K-1}, v_K)$ - сокращенное обозначение пути в графе.

Пути в графе G_1: $P_1 = (1,4,2,3)$; $P_2 = (1,4,2,1)$; $P_3 = (2,3,2)$; $P_4 = (1,4,2,1,4,1)$.
Различные пути из вершины 2 в вершину 4 в графе G_3: $P_1 = (2,4)$; $P_2 = (2,3,4)$; $P_3 = (2,3,1,4)$; $P_4 = (2,1,4)$; $P_5 = (2,3,2,4)$; $P_6 = (2,2,4)$;

Путь в графе, все ребра которого различны, называется ***простым путем***. Путь в графе, все вершины которого различны, называется ***элементарным путем***. Все элементарные пути являются также и простыми. Для графа G_3 пути P_1 , P_2 , P_3 , P_5 – простые, а пути P_4 , P_6 - простые, но не элементарные.

Простой путь, начинающейся и заканчивающейся в одной вершине, называется ***циклом***. В неориентированном графе длина цикла больше или равна 3. Для графа G1: путь $C_1 = (1,2,4,1)$ – цикл; путь $P = (2, 3,2)$ – не цикл. В орграфе длина цикла больше или равна 2. Простой граф без цикла называется ***ациклическим***.

Граф называется ***связным***, если для любой его вершины i существует путь к любой другой вершине j: $P = (i = v_1 , v_2 , \ldots, v_{k-1} , v_k = j)$ Если в графе какие либо два подмножества вершин не соединены ребрами, граф не является связным. Например, граф G_2 не является связным.

Граф $G' = (V', E')$ является ***подграфом*** графа $G = (V, E)$, если $V' \subset V$, $E' \subset E$. Максимальный связный подграф графа называется его ***компонентой связности***. Ориентированный граф называется ***сильно связным***, если для каждой пары вершин i и j существует хотя бы один ориентированный путь из i в j и хотя бы один ориентированный путь из j в i.

Дерево – ациклический связный граф, одна вершина которого (корень) не имеет входящих ребер, а все другие вершины имеют единственное входящее ребро. В дереве n вершин, (n-1) ребер. Каждому ребру графа можно сопоставить некоторое число, называемое ***весом***, физический смысл которого определяется предметной областью решаемой задачи. Такой граф называется ***взвешенным***.

4.2. Представление графов

4.2.1. Матричное представление графов

Пусть для простого графа $G = (V, E)$ задан некоторый вид упорядоченности вершин $\{V_1 , V_2 , \ldots, V_n\}$, такой, что некоторая вершина названа первой, другая – второй и т.д., где n – количество вершин.

Матрица $A (n, n) = |a_{ij}|$ $i = 1..n$, $j = 1.. n$ такая, что

$$a_{ij} = \begin{cases} 1 - \text{ребро } (v_i, v_j) \text{ из вершины i в вершину j существует,} \\ 0 - \text{ребро } (v_i, v_j) \text{ из вершины i в вершину j не существует,} \end{cases}$$

называется ***матрицей смежности***. Для простых неориентированных графов матрица смежности симметрична относительно главной диагонали. Для графа без петель главная диагональ содержит только нулевые элементы. Для взвешенных графов задается ***весовая матрица*** W.

Матрица $W (n, n) = |w_{ij}|$ $i = 1..n$, $j = 1.. n$ такая, что

$$\begin{cases} \text{вес} - \text{ребро } (v_i, v_j) \text{ из вершины i в вершину j существует,} \\ \end{cases}$$

w_{ij} =

∞ – ребро (v_i, v_j) из вершины i в вершину j не существует,

называется ***весовой матрицей.***

Граф G4 (рис. 28) представлен матрицей смежности (рис. 29). Элементы i-й строки матрицы определяются ребрами, исходящими из вершины v_i. Элементы j-го столбца определяются ребрами, входящими в вершину v_j.

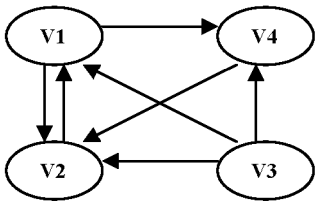


Рис. 28. Граф G4

	V1	V2	V3	V4
V1	0	1	0	1
V2	1	0	0	0
V3	1	1	0	1
V4	0	1	0	0

Рис. 29. Матрица смежности графа G4

Описание графа с помощью весовой матрицы:

```
Const
  maxV = 100;                                  { макс. количество вершин }
  maxW = 1000;                                 { максимальный вес }
Type
  Vertex = 1… maxV;                            { тип “вершина” }
  Weight = 0 ... maxW;                         { тип “вес” }
  Graph = record                               { тип “граф” }
    Size : Vertex;                             { количество вершин в графе }
    Matrix : array [Vertex, Vertex]  of Weight { весовая матрица графа }
  End;
```

В матрице смежности единица на пересечении i-й строки и j-го столбца матрицы A означает наличие ребра (v_i, v_j), т.е., прямого пути длины 1 из вершины i вершину j. Прямой путь длины 1 между вершинами графа существует не всегда. Поэтому необходимо определить, существует ли в графе какой либо путь (не обязательно простой) длины, не превышающей n, из вершины i в вершину j. Это можно определить с помощью матрицы достижимости P.

Матрица $P(n, n) = |p_{ij}|$ i =1..n , j = 1.. n такая, что

$$p_{ij} = \begin{cases} 1 - \text{существует простой путь длины} \le n \text{ из вершины } v_i \text{ в } v_j, \\ 0 - \text{не существует простой путь длины} \le n \text{ из вершины } v_i \text{ в } v_j, \end{cases}$$

называется ***матрицей достижимости***.

Для определения матрицы достижимости используются булевы матричные операции над квадратными матрицами размера n * n: булева сумма ($\vee$) и булево произведение ($\wedge$). Для вычисления матрицы P задается выражение:

$$P = A \vee A^2 \vee A^3 \vee \ldots \vee A^{n-1} \vee A^n = \bigcup_{k=1}^{n} A^k .$$

Каждый элемент матрицы A^K, k =1,…,n, находящийся на пересечении i-й строки и j –го столбца, равен 1, если существует не обязательно простой путь длины k, ведущий из вершины i в вершину j. Например, элементы матрицы A^2 на пересечении строки i и столбца j задают наличие или отсутствие пути длины 2 между v_i и v_j. Элементы этой матрицы определяются выражением

$$A^2 = |a^2_{ij}| = |\bigcup_{k=1}^{n}(a_{ik} \cap a_{kj})| \quad i,j = 1,\ldots,n.$$

Для любого k условие $a_{ik} \wedge a_{kj} = 1$ выполняется тогда и только тогда, когда оба элемента a_{ik} и a_{kj} равны 1, т.е., в графе имеются ребра (v_i, v_k) и (v_k, v_j). Таким образом,

$$A^2 = A \wedge A ; \; A^3 = A \wedge A^2 ; \; A^4 = A \wedge A^3 ; \; \ldots \; A^n = A \wedge A^{n-1} .$$

Для графа G4 матрица достижимости вычисляется следующим образом.

$$A^2 = \begin{vmatrix} 1\,1\,0\,0 \\ 0\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,0\,0\,0 \end{vmatrix} \quad A^3 = \begin{vmatrix} 1\,1\,0\,1 \\ 1\,1\,0\,0 \\ 1\,1\,0\,1 \\ 0\,1\,0\,1 \end{vmatrix} \quad A^4 = \begin{vmatrix} 1\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,0\,0 \end{vmatrix} \quad P = \begin{vmatrix} 1\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,0\,1 \end{vmatrix}$$

Диагональные элементы в матрицах A^K, k =1,…,n указывают наличие цикла длины k в вершине v_i, i = 1,…,n. Диагональные элементы в матрице P указывают наличие цикла длины, не превышающей n, в вершинах v_i, i=1,…,n. Для ациклических графов диагональные элементы равны 0.

Матрица $Q(n, n) = |q_{ij}|$ i =1..n , j = 1.. n такая, что

$$q_{ij} = \begin{cases} 1 - \text{существует простой путь длины} \le n \text{ из вершины } v_j \text{ в } v_i, \\ 0 - \text{не существует простой путь длины} \le n \text{ из вершины } v_j \text{ в } v_i, \end{cases}$$

называется ***матрицей контрдостижимости***. Столбец j матрицы Q совпадает со строкой j матрицы P, т.е., матрица контрдостижимости является транспонированной к матрице достижимости $Q = P^T$.

4.2.2. Представление графа в виде списка смежности

Представление графа G4 (рис. 28) в виде списка смежности приведено на рис. 30.

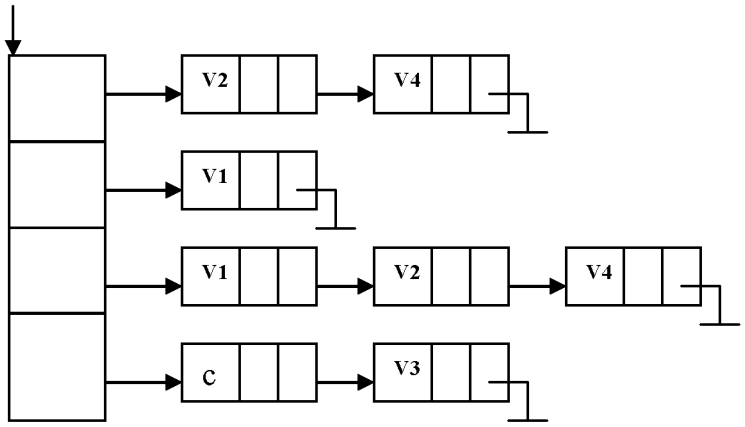


Рис. 30. Представление графа в виде списка смежности

Описание графа с помощью списка смежности:

```
Const
  maxV = 100;                                   { макс. количество вершин }
  maxW = 1000;                                  { максимальный вес }
Type
  Vertex = 1… maxV;                             { тип “вершина” }
  Weight = 1 ... maxW;                          { тип “вес” }
  P_Node = ^Node;                               { тип “указатель на элемент списка вершин”
                                                }
  Node = record                                 { тип “элемент списка вершина” }
    Id : Vertex;                                { идентификатор смежной вершины }
    We : Weight;                                { вес ребра }
    Link : P_Node                               { указатель на соседний элемент списка вершин }
End;

P_List = ^Array [Vertex] of P_Node;             тип “указатель на список смежности”}
  Graph = record                                { тип “граф” }
    Size : Vertex;                              { количество вершин в графе }
    List : P_List                               { указатель на список смежности вершин графа}
  End;
```

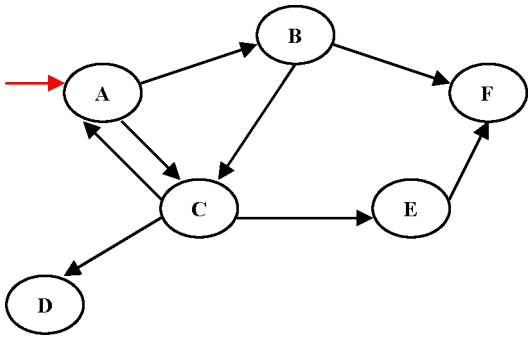
Выбор представления графа в виде списка смежности или матрицы смежности зависит от того, “разреженный” граф или “плотный”. Плотным называется граф, у которого количество ребер приближается к числу (n^2-n), где n – количество вершин. Разреженным называется граф, у которого количество ребер намного меньше числа вершин. Для представления плотных графов рекомендуется использовать матрицы смежности, а для представления разреженных графов – списки смежности.

4.3. Алгоритмы обхода графов

Обход графа легко выполнить, если принять одну из его вершин в качестве первой посещаемой, т.е. “корня”. Затем следует рекурсивно обрабатывать все остальные вершины достижимые от первой. Существует два основных алгоритма обхода графа.

Обход графа в глубину

Этот алгоритм обобщает алгоритм нисходящего обхода деревьев. Рассмотрим алгоритм нисходящего обхода на примере графа G5 (рис. 31), представленного матрицей смежности.



	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	1	0	0	1
C	1	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Рис. 31. Граф G5, представленный матрицей смежности

В качестве корневой выбираем вершину A. Обработав корневую вершину A, находим в списке смежности этой вершины первый элемент не равный нулю. Он соответствует вершине B, поэтому далее рассматриваем строку матрицы, соответствующую вершине B.

Обрабатываем вершину B. Для нее первым ненулевым элементом в строке матрицы смежности является вершина C. Переходим к строке, соответствующей вершине C, обрабатываем вершину C. Для вершины C первым ненулевым элементом в строке матрицы смежности является A, но вершина A уже обработана, значит, для вершины C выбираем следующий элемент не равный нулю, т.е. D. Строка матрицы смежности, соответствующая вершине D, пуста. Поэтому опять возвращаемся к предыдущей вершине C и находим в строке матрицы смежности следующий элемент не равный нулю, он соответствует вершине E. От вершины E переходим к вершине F. В результате, получилась следующая трасса обхода графа G5 в глубину, начиная с вершины A: ***A B C D E F***. Чтобы при обходе графа избегать зацикливаний, которые могут возникнуть из–за многократного посещения вершин, имеющих несколько исходящих ребер, обработанные вершины следует помечать. В результате обхода графа в глубину, пройденные ребра вместе с обработанными вершинами образуют одно или несколько деревьев (если граф имеет несколько компонент связности). Для графа G5 дерево, полученное в результате обхода в глубину, показано на рис. 32:

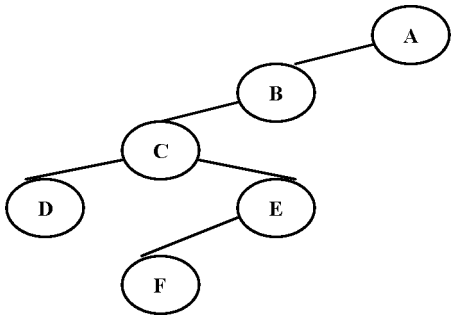


Рис. 32. Дерево, полученное в результате обхода графа G5 в глубину

Для реализации обхода графа в глубину необходимо использовать структуру стека. Реализация алгоритма обхода графа в глубину из заданной вершины приведена ниже. Граф задан с помощью матрицы смежности. Используется вектор посещенных вершин, в котором отмечаются обработанные вершины. Распечатываются номера вершин в порядке их посещения:

```
Const
  maxV = 100;                                  { максимальное количество вершин }
Type
  Vertex = 1… maxV;                                          { тип “вершина” }
  Graph = array [Vertex, Vertex] of  boolean;                { тип “граф” }
  Visited = array [Vertex] of  boolean;          { вектор посещенных вершин }
Var
   G: Graph; V: Visited; j: Vertex;

Procedure Depth_first Search (i: Vertex);      { i – вершина, с которой начинается
                                                обход}
Var k: Vertex;
 begin
   V[i]:=true;                                 { отметить вершину i как обработанную }
    writeln( i );                              { распечатать номер посещенной вершины }
    for k:=1 to maxV do                        { найти первую встретившуюся ранее не
                                               посещенную вершину k, смежную с вершиной i }
      if G[i,k] and (not V[k]) then
        Depth_first Search( k )                { перейти к обработке вершины k }
 end;

 begin
   for j:=1 to maxV do V[j]:=false;    { инициализация вектора посещенных вершин }
```

Обход графа в ширину

Этот вид обхода соответствует прохождению узлов в горизонтальном порядке (слева направо), в соответствии с матрицей смежности в порядке возрастания длины пути от корня к вершине. Трасса обхода в ширину графа G6 (рис. 33), начиная с вершины A, - ***A B C G H D E F I L J K***.

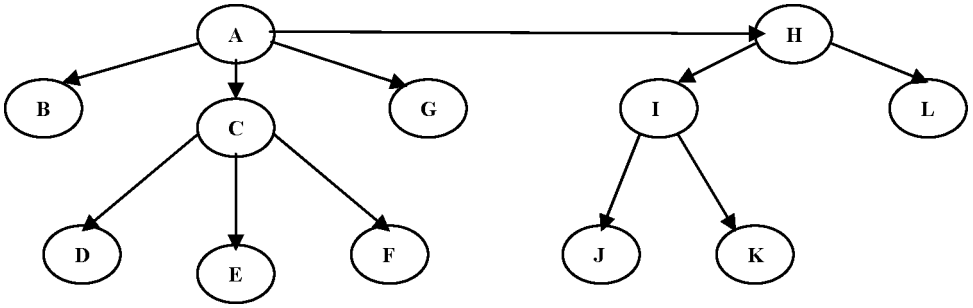
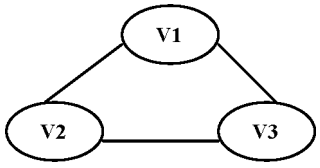


Рис. 33. Граф G6

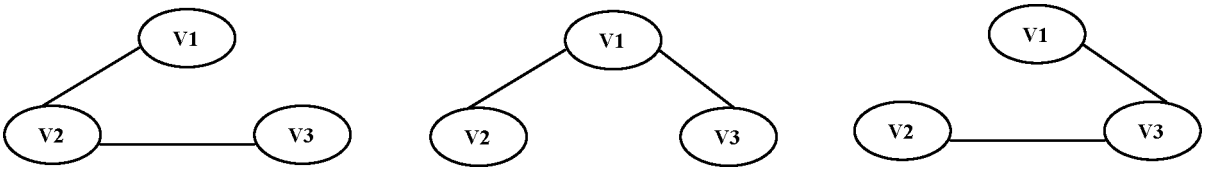
Для реализации обхода графа в ширину используется структура очереди, в которой должны временно храниться вершины, имеющие смежные вершины на более низком уровне, чтобы на более низком уровне вершины также можно было бы обрабатывать слева направо.

4.4. Остовные деревья

Остовным деревом (или каркасом) G' = (V', E') графа G = (V, E) называется подграф графа G, который является деревом и содержит все вершины графа G, т.е., V' = V, E' ⊂ E. Число различных каркасов полного, связного, неориентированного графа с n вершинами равно n^{n-2}. На рис. 34 показаны различные остовные деревья графа G7.



а



б

Рис. 34. Граф G7 и его остовные деревья
а – граф G7; б – остовные деревья графа G7

Алгоритм построения остовного дерева G = (V',E') неориентированного графа G = (V, E)

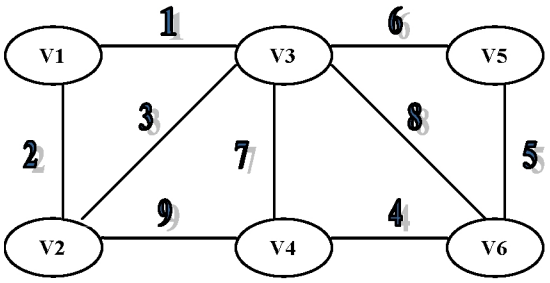
1. Пусть v ∈V – произвольная вершина. V' = [v], E' = [ ] ;
2. Если V' = V, то G ' = (V', E') – остовное дерево построено, иначе переход к пункту 3.
3. Найти ребро (i,j)∈ E такое, что (i∈ V') & (j∈ V-V').
   Обновление V' и E': V' = V' ∨ [j]; E' = E' ∨ [(i,j)]. Переход к пункуту 2.

Данный алгоритм реализуется на основе обхода графа в глубину с использованием матрицы смежности. Вершины, подключаемые к остовному дереву, помечаются, чтобы избежать зацикливаний. Остовное дерево всегда имеет N вершин и (N–1) ребро. Для хранения остовного дерева, следует использовать структуру двумерного массива следующего вида, позволяющую перечислить ребра остовного дерева:

```
Var Cover: array [1 .. maxV-1, 1..2] of  Vertex;
```

4.4.1 Остовные деревья минимального веса

Пусть задан связный неориентированный граф G = (V, E). $W = |w_{ij}|$ i=1..n, j=1..n. В весовой матрице обозначим бесконечно большим числом отсутствие ребра между вершинами i и j. ***Остовным деревом минимального веса*** называется остовное дерево с минимальной суммой весов ребер. Существует два алгоритма построения остовных деревьев минимального веса, оба они выполняются за (n-1) итерацию. Рассмотрим алгоритмы построения остовных деревьев на примере графа G8 (рис. 35).



	V1	V2	V3	V4	V5	V6
V1	0	2	1	∞	∞	∞
V2	2	0	3	9	∞	∞
V3	1	3	0	7	6	8
V4	∞	9	7	0	∞	4
V5	∞	∞	6	∞	0	5
V6	∞	∞	8	4	5	0

Рис. 35. Граф G8, представленный матрицей смежности

Алгоритм Краскала

Построение начинается одновременно со всех вершин, которые рассматриваются в качестве отдельных фрагментов. Затем какие либо два фрагмента последовательно соединяются ребром, имеющим минимальный вес среди всех ребер, добавление которых не создает цикл (рис. 36).

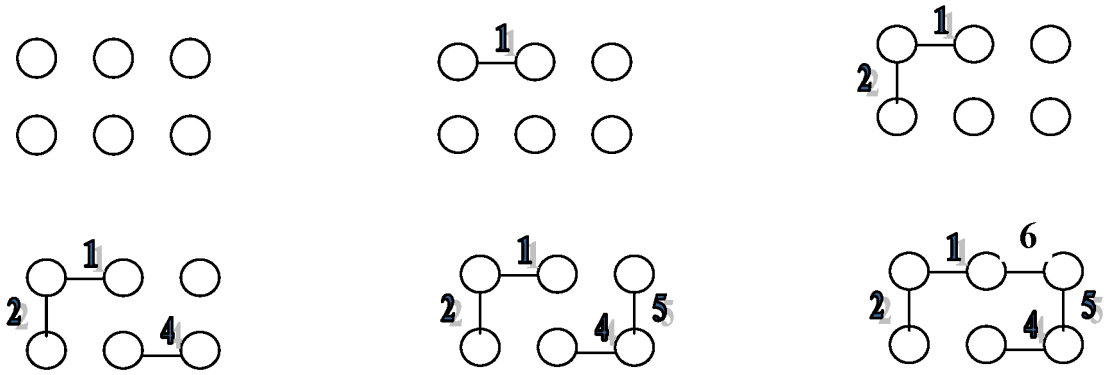


Рис. 36. Алгоритм Краскала

Алгоритм Прим – Дейкстра

Этот алгоритм начинает работу с произвольно выбранной вершины, которая затем соединяется с ближайшей вершиной. Соединенные вершины образуют связное множество, остальные вершины - несвязное множество. На каждом шаге в множество несвязных вершин добавляется вершина, находящаяся на кратчайшем расстоянии к любой из вершин связного множества. Множества связных и несвязных вершин корректируются, алгоритм заканчивает работу, когда все вершины попадут в множество связных вершин (рис. 37). Построение остовного дерева начнем с вершины V5.

Если веса некоторых ребер совпадают, предпочтение отдается ребру, исходящему из вершины с меньшим номером, а, если ребра исходят из одной и той же вершины, предпочтение отдается ребру, входящему в вершину с меньшим номером.

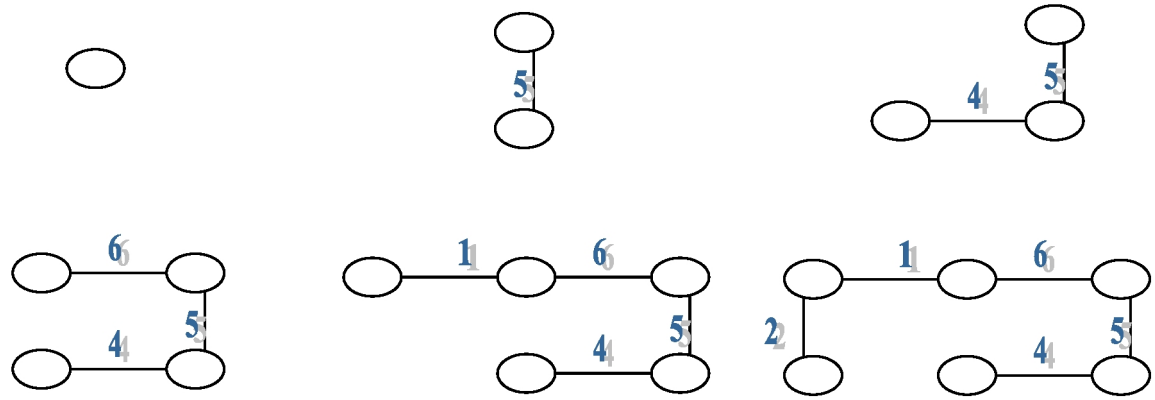


Рис. 37. Алгоритм Прим-Дейкстра

4.5. Алгоритмы нахождения кратчайших путей в графе

Взвешенный орграф G = (V, E) задан весовой матрицей W = $|w_{ij}|$ i=1..n, j=1..n. Заданы начальная и конечная вершины s, t ∈ V. Длина любого

ориентированного пути между вершинами s и t определяется как сумма весов ребер, принадлежащих этому пути:

$$P = (s, i, j, \ldots, k, t) = W_{si} + W_{ij} + \ldots + W_{kt}.$$

Кратчайший путь между вершинами s и t имеет минимальную длину пути между заданными вершинами s и t. Ребра (i,j) принадлежат пути от вершины s к вершине t:

$$P_{min} = \min \sum W_{ij}$$

Все алгоритмы нахождения кратчайших путей определяют оценки от начальной вершины до всех остальных вершин графа и осуществляют построение матрицы кратчайших путей.

Матрица $D(n, n) = |d_{ij}|$ $i = 1..n$, $j = 1..n$ такая, что

$$d_{ij} = \begin{cases} \text{длина кратчайшего пути - путь из вершины i в вершину j существует,} \\ \infty \text{ – путь из вершины i в вершину j не существует,} \end{cases}$$

называется ***матрицей кратчайших путей.***

Наиболее широко распространены два алгоритма нахождения кратчайших путей в графе: алгоритм Флойда и алгоритм Дейкстра.

4.5.1. Алгоритм Флойда

Алгоритм Флойда состоит в последовательном поиске длин кратчайших путей из вершины i в вершину j для всех пар (i,j) при ограничении, что промежуточной вершиной может быть только вершина 1, затем при ограничении, что промежуточными могут быть только вершины 1 и 2 и т.д. Таким образом, поиск кратчайших путей выполняется в порядке увеличения количества промежуточных вершин:

1. [Инициализация]. Установить D = W;
2. [Выполнение прохода]. Повторять шаги 3,4 при k = 1,2,…n;
3. [Обработка строк]. Повторять шаг 4 при i = 1,2,…n;
4. [Обработка столбцов]. Повторять при j = 1,2,…n:
   Установить $D_{ij} = \min [D_{ij}, D_{ik} + D_{kj}]$.
   Два варианта соответствуют использованию и неиспользованию вершины k в качестве промежуточной;
5. [Завершение алгоритма].

Так как каждый из n шагов алгоритма выполняется для каждой пары вершин, объем вычислений составляет $O(n^3)$.

4.5.2. Алгоритм Дейкстра

Алгоритм Дейкстра выполняет поиск длины кратчайшего пути от некоторого выделенного узла (узла 1) до всех остальных узлов графа. Пути определяются в порядке возрастания их длины. Обозначим через D_i длину пути из узла 1 в узел i. S – множество связных вершин. d_{ij} – длина прямого пути из вершины i в вершину j.

Перед началом работы алгоритма S = [1]; D1 =0; $Dj = d_{1j}$ для $\forall j \neq 1$.

1. [Поиск следующего ближайшего узла]. Найти вершину $i \notin S$ такую, что $D_i = \min D_j$, $j \notin S$;
   [Обновление множества связных вершин]. $S = S \vee [i]$. Если S = V, то кратчайший путь найден, иначе переход к пункту 2;
2. [Обновление длин кратчайших путей]. Для $\forall j \notin S$
   $$D_j = \min [D_j , D_i + d_{ij}].$$
   Переход к пункту 1.

Так как в алгоритме Дейкстра число операций, выполняемых на каждом шаге, пропорционально количеству вершин n, а каждый шаг выполняется (n-1) раз, объем вычислений составляет $O(n^2)$. Применение алгоритма Дейкстра для нахождения всех кратчайших путей в графе потребует объема вычислений $O(n^3)$.

4.6. Контрольные вопросы к главе 4

1. Дайте определение графа.

2. Какие способы представления графов Вам известны?

3. Дайте определение матрицы смежности, весовой матрицы, матрицы достижимости.

4. Дайте определение дерева как частного случая графа.

5. Сформулируйте алгоритм обхода графа в глубину.

6. Сформулируйте алгоритм обхода графа в ширину.

7. Дайте определение остовного дерева графа.

8. Сформулируйте алгоритм Краскала построения остовного дерева графа.

9. Сформулируйте алгоритм Прим-Дейкстра построения остовного дерева графа.

10. Приведите общую постановку задача нахождения кратчайшего пути в графе.

11. Сформулируйте алгоритм Флойда нахождения кратчайшего пути в графе.

12. Сформулируйте алгоритм Дейкстра нахождения кратчайшего пути в графе.

13. Сравните эффективность алгоритмов Флойда и Дейкстра нахождения кратчайшего пути в графе.

4.7. Упражнения к главе 4

1. На основании заданной матрицы смежности графа постройте матрицу достижимости этого графа.

2. Найдите все вершины графа, недостижимые от заданной его вершины.

3. **Обод** - это граф, вершины которого V_0, V_1, ... , V_n (n >= 2) можно занумеровать так, что для всех i (1 <= i <= n-1) вершина V_i соединена ребрами с V_{i-1} и V_{i+1}, вершина V_0 - с V_n, а других ребер нет. Определите, является ли ободом заданный граф.

4. Граф называется ***связным***, если для любой пары вершин существует соединяющий их путь (не обязательно прямой). Определите, является ли связным заданный граф.

5. Определите, является ли заданный граф ***ациклическим*** (т.е., не содержащим циклов).

6. ***Источником*** орграфа называется вершина, от которой достижимы все другие вершины; ***стоком*** - вершина, достижимая от всех других вершин. Найдите все источники и стоки данного орграфа.

7. Реализуйте алгоритм обхода графа в глубину из всех вершин графа.

8. Реализуйте алгоритм обхода графа в ширину из заданной вершины.

9. Реализуйте алгоритм построения основных деревьев заданного графа.

10. Напишите программу, реализующую алгоритм Флойда поиска в графе длин кратчайших путей, ведущих из вершины i в вершину j для всех пар (i,j).

11. Напишите программу, реализующую алгоритм Дейкстра поиска длин кратчайших путей, ведущих из заданной вершины во все остальные вершины.

ЗАКЛЮЧЕНИЕ

Учебное пособие “Структуры данных. Часть II. Нелинейные динамические структуры” посвящено рассмотрению структур данных и алгоритмов, которые являются фундаментом современной методологии разработки программ.

Учебное пособие включает разделы, которые подробно описывают методы множественной интерпретации объектов, рекурсивные структуры данных, иерархические структуры (деревья, графы). Учебное пособие отличается методикой изложения всех разделов с точки зрения особенностей внутреннего представления структур данных различных видов. Теоретический материал иллюстрируется большим количеством примеров программ, реализующих алгоритмы обработки различных структур данных.

Каждый раздел содержит большое количество примеров программ обработки данных различной структуры на языке Pascal. Логическим завершением каждого раздела являются контрольные вопросы и упражнения для самостоятельной работы студентов.

Вопросы, имеющие практическое значение для студентов при выполнении лабораторных работ и курсового проекта, освещены в учебном пособии с необходимой для использования полнотой.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Ахо А. Структуры данных и алгоритмы: пер. с англ. / Ахо А., Хопкрофт Д., Ульман Д. –М.: Вильямс, 2002. – 384 с.
2. Бобровский С.И. Delphi 7. Учебный курс : учебное пособие для вузов /Бобровский С.И. – СПб.: Питер, 2006. – 736 с.
3. Вирт Н. Алгоритмы + структуры данных = программы: пер. с англ. / Н.Вирт. -М.: Мир, 1985. – 452 с.
4. Вирт Н. Алгоритмы и структуры данных: пер. с англ. / Н.Вирт. Изд. 2-е, испр. -СПб.: Невский диалект, 2005. – 352 с.
5. Иванова Г.С. Основы программирования: учебник для вузов / Иванова Г.С. Изд. 3-е, испр. –М.: Изд-во МГТУ им. Н.Э. Баумана, 2004. – 416с.
6. Кнут Д. Искусство программирования для ЭВМ: в 3 т. Т 1: пер. с англ. / Кнут Д. – М.: Вильямс, 2000. – 720 с.
7. Кормен Т. Алгоритмы: построение и анализ: пер. с англ. / Кормен Т., Лейзерсон Ч., Ривест Р. –М.: МЦНМО, 2000. – 960 с.
8. Павловская Т.А. Паскаль. Программирование на языке высокого уровня: учебник для вузов / Павловская Т.А. – Спб.: Питер, 2004. – 393 с.
9. Павловская Т.А. Паскаль. Программирование на языке высокого уровня: Практикум : учебник для вузов / Павловская Т.А. – Спб.: Питер, 2006. – 317 с.
10. Симонова Е.В. Структуры данных. Часть I. Линейные динамические структуры. Учебное пособие для вузов / Симонова Е.В. – Самара: Изд-во СГАУ, 2006. – 82 с.
11. Топп У. Структуры данных в С++: пер. с англ. / Топп У., Форд У. –М.: ЗАО “Издательство БИНОМ”, 1999. – 815 с.
12. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс: учебное пособие / Фаронов В.В. –М.: Нолидж, 2006. – 575 с.
13. Фаронов В.В. Delphi. Программирование на языке высокого уровня: учебник для вузов / Фаронов В.В. – СПб.: Питер, 2005. – 640 с.
14. Шень А. Программирование. Теоремы и задачи: учебное пособие для вузов / А.Шень. – М.: МЦНМО, 2004. – 262 с.

Учебное издание

Симонова Елена Витальевна

СТРУКТУРЫ ДАННЫХ
ЧАСТЬ II
НЕЛИНЕЙНЫЕ ДИНАМИЧЕСКИ СТРУКТУРЫ

Учебное пособие

Технический редактор В. А. Ф у р с о в
Редакторская обработка Н. С. К у п р и я н о в а
Корректорская обработка Н. С. К у п р и я н о в а
Доверстка Н. С. К у п р и я н о в а

Подписано в печать 25.09.07. Формат 60 84 1/16.
Бумага офсетная . Печать офсетная .
Усл. печ. л. 5,25.
Тираж 120 экз . Заказ . ИП-97/2007

Самарский государственный
аэрокосмический университет .
443086 Самара , Московское шоссе , 34.

Изд-во Самарского государственного
аэрокосмического университета .
443086 Самара , Московское шоссе , 34.