Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра «Вычислительные методы и программирование»

# ПРОГРАММИРОВАНИЕ АЛГОРИТМОВ В СРЕДЕ DELPHI

Лабораторный практикум
по курсам «Программирование»
и «Основы алгоритмизации и программирование»
для студентов 1–2-го курсов всех специальностей БГУИР
дневной и вечерней форм обучения

В 2-х частях

Часть 2

Под общей редакцией А.К. Синицына

Минск 2003

УДК 681.3.06 (075.8)
ББК 32.973 я 73
П 78

А в т о р ы :
А.К. Синицын, С.В. Колосов, А.А. Навроцкий,
А.В. Гуревич, В.Т. Карцев, А.А. Лавренов, А.В. Щербаков

**П 78** **Программирование алгоритмов в среде DELPHI:** Лаб. практикум по курсам «Программирование» и «Основы алгоритмизации и программирование» для студентов 1–2-го курсов всех специальностей БГУИР дневной и вечерней форм обучения. В 2 ч. Ч. 2 / А.К. Синицын, С.В. Колосов, А.А. Навроцкий и др.; Под общ. ред. А.К. Синицына. — Мн.: БГУИР, 2003. — 72 с.: ил.

ISBN 985-444-471-6 (ч. 2).

В лабораторном практикуме приведены краткие теоретические сведения по алгоритмам и структурам данных, а также даны примеры их реализации на языке Object Pascal в среде Delphi. После каждой темы приведен набор индивидуальных заданий.

В практикум вошло 10 лабораторных работ.

**УДК 681.3.06 (075.8)**
**ББК 32.973 я 73**

Часть 1 издана в БГУИР в 2002 г.

**ISBN 985-444-471-6 (ч. 2)**
**ISBN 985-444-400-7**

# СОДЕРЖАНИЕ

# ТЕМА 1. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ РЕКУРСИИ

***ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:*** ИЗУЧИТЬ СПОСОБЫ ПРОГРАММИРОВАНИЯ АЛГОРИТМОВ С ИСПОЛЬЗОВАНИЕМ РЕКУРСИИ.

## *1.1. Понятие рекурсии*

Рекурсия — это такой способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения составляющих ее операторов обращается сама к себе. Классический пример программирования вычисления $n$-го члена ($n>0$) рекуррентной последовательности $x_n = j\ (x_{n-1})$ при $x_0 = a$

```
...
Function XR(n:Word):extended;
    begin
      if n=0 then  XR:=a
             else  XR:=φ(XR(n-1));
    end;
```

Здесь φ — функция, определяющая закон рекуррентности и определяемая в общем случае как: function Fi(x:extended):extended; или явно, например, как XR:=(XR(n – 1) + 4/XR(n – 1))/2.

При выполнении правильно организованной рекурсивной подпрограммы осуществляется многократный переход от некоторого текущего уровня организации алгоритма к нижнему уровню последовательно, до тех пор, пока наконец не будет получено тривиальное решение задачи (в вышеприведенном примере $n=0$).

РЕКУРСИВНАЯ ФОРМА ЗАПИСИ АЛГОРИТМА ОБЫЧНО ВЫГЛЯДИТ ИЗЯЩНЕЕ ИТЕРАЦИОННОЙ И ДАЕТ БОЛЕЕ КОМПАКТНЫЙ ТЕКСТ ПРОГРАММЫ, НО ПРИ ВЫПОЛНЕНИИ, КАК ПРАВИЛО, МЕДЛЕННЕЕ И МОЖЕТ ВЫЗВАТЬ ПЕРЕПОЛНЕНИЕ ПРОГРАММНОГО СТЕКА.

Рекурсивный вызов может быть прямым, как в вышеприведенном примере, и косвенным. В этом случае подпрограмма обращается к себе опосредованно, путем вызова другой подпрограммы, в которой содержится обращение к первой, например:

```
Procedure B(j:byte);   Forward;
Procedure A(i:byte);
  begin
       ...
       B(i);
  end;
Procedure B;
  begin
       ...
       A(j);
  end;
```

В этом примере обращение к процедуре B(i) записано раньше, чем ее описание, что в принципе запрещено. Для разрешения этой ситуации используется **опережающее описание** с помощью стандартной директивы **Forward.**

### 1.2. Порядок выполнения работы

Задание: Написать программу, содержащую две подпрограммы, вычисляющие $S=\sum_{i=1}^{n}(i+1)^2/i$. Одна вычисляет сумму без использования рекурсии, другая — рекурсивная.

### 1.2.1. Пример рекурсивной и нерекурсивной подпрограммы

Написать две подпрограммы, вычисляющие $S=\sum_{i=1}^{n}(i+1)^2/i$. Одна вычисляет сумму с использованием рекурсии, другая — рекурсивная.

```
Function sr(n:word):extended;  // Программа с использованием рекурсии
  begin
    if n=1 then sr:=4
           else  sr:=sr(n-1)+sqr(n+1)/n;
  end;
Function s(n:word):extended;  // Программа, не использующая рекурсию
  var i:byte;
  begin
    Result:=4;
     for i:=2 to n do Result:=result+sqr(i+1)/i;
  end;
```

### 1.3. Варианты задач

Решить поставленные задачи двумя способами — с применением рекурсии и без нее.

1. Для заданного целого десятичного числа N получить его представление в p-ичной системе счисления (p<10).

2. В упорядоченном массиве целых чисел $a_i$, $i=1...n$ найти номер элемента ***c***, используя метод двоичного поиска. Предполагается, что элемент *c* находится в массиве.

3. Найти наибольший общий делитель чисел *M* и *N*. Используйте теорему Эйлера: Если *M* делится на *N*, то *НОД (N, M)=N*, иначе *НОД (N, M)= =НОД (M mod N, N)*.

4. Вычислить число Фибоначчи *Fb(n)*. Числа Фибоначчи определяются следующим образом: *Fb(0)=1; Fb(1)=1; Fb(n)=Fb(n-1)+Fb(n-2)*.

5. Найти значение функции Аккермана *A(m, n)*, которая определяется для всех неотрицательных целых аргументов m и n следующим образом:

*A(o, n)=n+1;*

*A(m, o)=A(m-1, 1); (m>o);*

*A(m, n)=A(m-1, A(m, n-1)); (m>o; n>o).*

6. Вычислить *m*-ю производную полинома степени *n*, $P_n = \sum_{i=0}^{n} a_i x^{i-1}, \quad (m < n)$.

7. Вычислить значение $x = \sqrt{a}$, используя рекуррентную формулу $x_n = \frac{1}{2}\left(x_{n-1} + a / x_{n-1}\right)$, в качестве начального приближения использовать значение $x_0=0.5(1+a)$.

8. Найти максимальный элемент в массиве $a_1 \ldots a_n$, используя очевидное соотношение $max\ (a_1 \ldots a_n)=max\ (max\ (a_1 \ldots a_{n-1}), a_n)$.

9. Найти максимальный элемент в массиве $a_1 \ldots a_n$, используя соотношение (метод деления пополам) $max\ (a_1 \ldots a_n)=max\ (max\ (a_1 \ldots a_{n/2}), max\ (a_{n/2+1}, a_n))$.

10. Вычислить $y(n) = \sqrt{1+\sqrt{2+\ldots+\sqrt{n}}}$ .

11. Вычислить $y(n) = \cfrac{1}{n+\cfrac{1}{(n-1)+\cfrac{1}{(n-2)+\cfrac{1}{\ldots\ \ldots \\ \ldots+\cfrac{1}{1+\frac{1}{2}}}}}}$.

12. Вычислить произведение n≥2 (n-четное) сомножителей $y = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \ldots$

13. Вычислить $y = x^N$ по следующему алгоритму: $y = (x^{N/2})^2$, если N четное; $y = x \cdot y^{N-1}$, если N нечетное.

14. Вычислить *n!*.

15. Выполнить сортировку массива целых чисел $a_i$, $i=1, n$ с помощью разделения (QuickSort). См. Вирт Н. Алгоритмы и структура данных. С. 108–113.

# ТЕМА 2. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ДЕРЕВЬЕВ РЕШЕНИЙ

***Цель лабораторной работы:*** изучить способы программирования алгоритмов с использованием деревьев решений.

## 2.1. Задача оптимального выбора и дерево решений

МНОГО ВАЖНЫХ ПРИКЛАДНЫХ ЗАДАЧ (В ЧАСТНОСТИ ЗАДАЧ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА) МОЖНО СВЕСТИ К СЛЕДУЮЩЕЙ:

Имеется набор из n элементов $a_1 \ldots a_n$. Каждый элемент $a_i$ характеризуется определенными свойствами, например, вес $w_i$ и цена $c_i$ (это могут быть размер и время, объем инвестиций и ожидаемый доход, и т.д.). Требуется найти оптимальную выборку $a_{i_1} \ldots a_{i_k}$ из этого набора, т.е. такую, для которой, например, при заданном ограничении на суммарный вес $\sum_{j=1}^{k} w_{i_j} \le Wmax$ достигается максимальная стоимость $\sum_{j=1}^{k} c_{i_j}$.

С такой проблемой сталкивается, например, путешественник, упаковывающий чемодан, суммарный вес которого ограничен $Wmax$ кг, а ценность вещей должна быть побольше, или инвестор, которому надо выгодно вложить $Wmax$ млн р. в какие-то из $n$ возможных проектов, каждый из которых имеет свою стоимость и ожидаемый доход.

Решение данной задачи состоит в переборе всех возможных выборок $\{(i_1,\ldots,i_k), 0 \le k \le n\}$ из n значений $\{1, 2, \ldots, n\}$. Например, для n=3 таких выборок $2^n = 8$: $\{1,2,3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, $\{1\}$, $\{2\}$, $\{3\}$, $\{\ \}$. Наглядной моделью решения данной задачи служит двоичное дерево вида:

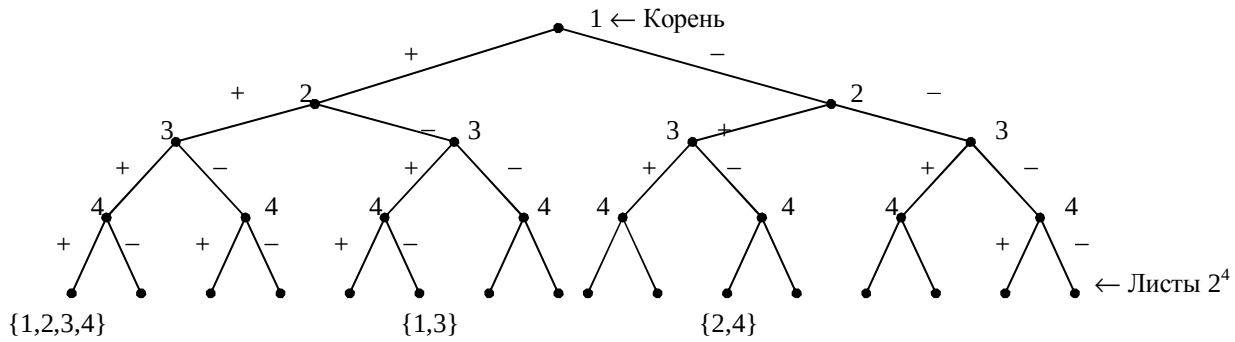



РИС. 2.1

КАЖДЫЙ УЗЕЛ В ДЕРЕВЕ ПРЕДСТАВЛЯЕТ СОБОЙ ОДИН ШАГ РЕШЕНИЯ ЗАДАЧИ ПЕРЕБОРА ВАРИАНТОВ. ВЕТВЬ В ДЕРЕВЕ СООТВЕТСТВУЕТ РЕШЕНИЮ, КОТОРОЕ ВЕДЕТ К БОЛЕЕ ПОЛНОМУ РЕШЕНИЮ. В ИГРОВЫХ ДЕРЕВЬЯХ, НАПРИМЕР, КАЖДАЯ ВЕТВЬ СООТВЕТСТВУЕТ ХОДУ ИГРОКА. ЛИСТЬЯ ПРЕДСТАВЛЯЮТ СОБОЙ ОКОНЧАТЕЛЬНЫЕ РЕШЕНИЯ (ВАРИАНТЫ).

ЦЕЛЬ СОСТОИТ В ТОМ, ЧТОБЫ ИЗ ВСЕХ ВОЗМОЖНЫХ ВАРИАНТОВ (РЕШЕНИЙ) НАЙТИ НАИЛУЧШИЙ ПУТЬ ОТ КОРНЯ ДО ЛИСТА ПРИ ВЫПОЛНЕНИИ НЕКОТОРЫХ УСЛОВИЙ.

ЕСТЕСТВЕННО, ЧТО ФОРМА ДЕРЕВА, УСЛОВИЯ НАЛАГАЕМЫЕ НА РЕШЕНИЯ ОПРЕДЕЛЯЮТСЯ КОНКРЕТНОЙ ЗАДАЧЕЙ. ТАК, ДЛЯ ЗАДАЧИ ОПТИМАЛЬНОГО ВЫБОРА ПОДХОДИТ ВЫШЕПРИВЕДЕННОЕ ДВОИЧНОЕ ДЕРЕВО. ДЛЯ ЗАДАЧИ МОДЕЛИРОВАНИЯ ИГРЫ В КРЕСТИКИ-НОЛИКИ ДЕРЕВО ПОЛУЧАЕТСЯ С БОЛЬШИМ КОЛИЧЕСТВОМ ВЕТВЕЙ, ВЫХОДЯЩИХ ИЗ УЗЛА:

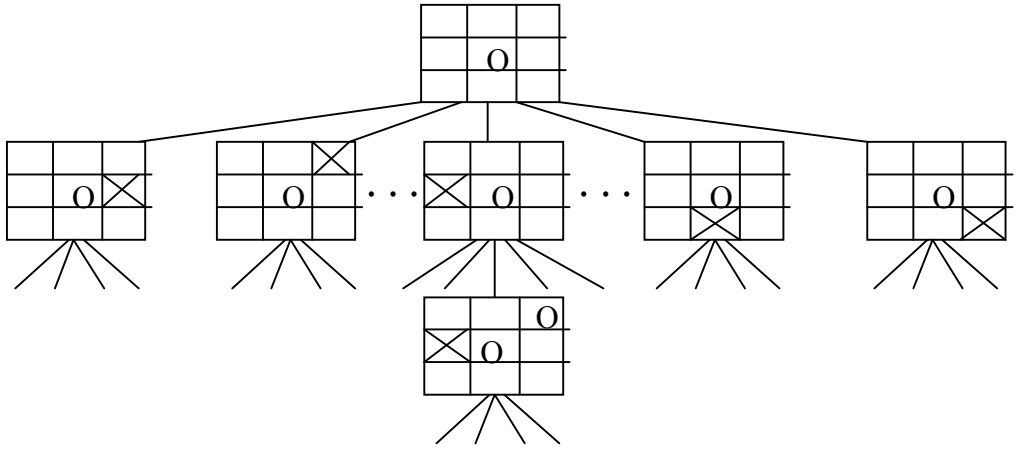


РИС. 2.2

В ЭТОЙ ЗАДАЧЕ ВЫБОРКИ ДЕЛАЮТСЯ ИЗ ДВУХМЕРНОГО ИСХОДНОГО МАССИВА $\{1...n,\ j_i = 1...m_i\}$ — *J*-Й КАНДИДАТ НА *I*-М ХОДУ.

ДЕРЕВЬЯ РЕШЕНИЙ ОБЫЧНО ОГРОМНЫ. ВСЕ ДЕРЕВО ДЛЯ ИГРЫ В КРЕСТИКИ-НОЛИКИ СОДЕРЖИТ БОЛЕЕ 500 ТЫС. ВАРИАНТОВ. БОЛЬШИНСТВО ЖЕ РЕАЛЬНЫХ ЗАДАЧ НЕСРАВНЕННО СЛОЖНЕЕ. СООТВЕТСТВУЮЩИЕ ИМ ДЕРЕВЬЯ РЕШЕНИЙ МОГУТ СОДЕРЖАТЬ БОЛЬШЕ ЛИСТЬЕВ, ЧЕМ АТОМОВ ВО ВСЕЛЕННОЙ.

## 2.2. РЕКУРСИВНЫЕ ПРОЦЕДУРЫ ПОЛНОГО ПЕРЕБОРА ИГРОВОГО ДЕРЕВА

ИМЕЕТСЯ ДОВОЛЬНО МНОГО МЕТОДОВ РАБОТЫ С ТАКИМИ ОГРОМНЫМИ ДЕРЕВЬЯМИ. ДЛЯ САМЫХ МАЛЕНЬКИХ ДЕРЕВЬЕВ (ТИПА КРЕСТИКИ-НОЛИКИ) МОЖНО ИСПОЛЬЗОВАТЬ **МЕТОД ПОЛНОГО ПЕРЕБОРА** ВСЕХ ВОЗМОЖНЫХ РЕШЕНИЙ.

РЕКУРСИВНАЯ ПРОЦЕДУРА ПОЛНОГО ПЕРЕБОРА ЛИСТЬЕВ ПРОИЗВОЛЬНОГО ДЕРЕВА ВЫГЛЯДИТ СЛЕДУЮЩИМ ОБРАЗОМ:

```
S:=0;  // МАССИВ (МНОЖЕСТВО), В КОТОРОМ ЗАПИСЫВАЕТСЯ РЕШЕНИЕ
Procedure Vbr(i:Index);
    Begin
     repeat
  <выбор и включение в выборку j-го кандидата на i-м ходу S[i]:=aij>;
        if i<n then  Vbr(i+1)
              else <лист решения сформирован, проверка
                    приемлемости и оптимальности, печать S[1..n]>;
        <исключение j-го кандидата на i-м ходу из выборки S[i]:=0>;
   until < список кандидатов ai1..aim_i исчерпан>;
    END;  // VBR
```

ЗДЕСЬ В КАЖДОМ УЗЛЕ ДЕРЕВА ИМЕЕТСЯ НАБОР *M* ВЕТВЕЙ (КАНДИДАТОВ), ПО КОТОРЫМ ВОЗМОЖНО ПРОДВИЖЕНИЕ К ПОЛНОМУ РЕШЕНИЮ. ВЕЛИЧИНА *M* МОЖЕТ БЫТЬ РАЗНАЯ И ОПРЕДЕЛЯЕТСЯ УСЛОВИЕМ **<** СПИСОК КАНДИДАТОВ ***AI1...AIM****i* ИСЧЕРПАН**>**. НАПРИМЕР, В РАНЕЕ РАССМОТРЕННОЙ ЗАДАЧЕ НА КАЖДОМ ХОДУ ПРЕДЛАГАЕТСЯ ДВА КАНДИДАТА {AI1=I, AI2=0}, В РЕЗУЛЬТАТЕ ПОЛУЧИМ РЕКУРСИВНЫЙ ПЕРЕБОР ДВОИЧНОГО ДЕРЕВА.

**Полный перебор** всех вариантов двоичного дерева реализуется следующей рекурсивной подпрограммой:

```
 type index=1..10;
 var s: Set of index;
Procedure Vbr(i:Index);
        begin
            < включение i-го элемента в выборку Include (s,i) >;
            if i<n then Vbr(i+1)
                    else < проверка приемлемости и оптимальности >;
                    < исключение i-го элемента из выборки Exclude (s,i) >;
            if i<n then Vbr(i+1)
                    else < проверка приемлемости и оптимальности >;
        end;  // Vbr
```

ПОЛНЫЙ ПЕРЕБОР ИСПОЛЬЗУЕТСЯ РЕДКО.

ДЛЯ РАБОТЫ С БОЛЬШИМИ ДЕРЕВЬЯМИ БОЛЕЕ ПОДХОДИТ РАССМОТРЕННЫЙ НИЖЕ **МЕТОД ВЕТВЕЙ И ГРАНИЦ**. ИДЕЯ ЭТОГО МЕТОДА В ТОМ, ЧТО НА КАЖДОМ ШАГЕ РЕШЕНИЯ ДЕЛАЕТСЯ ОЦЕНКА ЦЕЛЕСООБРАЗНОСТИ ДАЛЬНЕЙШЕГО СПУСКА ПО ТОЙ ИЛИ ИНОЙ ВЕТВИ И ПРЕКРАЩЕНИЕ (ОТСЕЧЕНИЕ) ПРОСМОТРА ПО ПУТИ ЗАВЕДОМО НЕ ОПТИМАЛЬНОМУ. ТАКОЕ ОТСЕЧЕНИЕ ПОЗВОЛЯЕТ ЗНАЧИТЕЛЬНО УМЕНЬШИТЬ КОЛИЧЕСТВО ДОПУСКАЕМЫХ РЕШЕНИЙ.

### 2.3. Рекурсивная процедура метода ветвей и границ

Стратегия отсечения заведомо неприемлемых и неоптимальных решений позволяет резко сократить количество просматриваемых вариантов. В общем случае для двоичного дерева эта стратегия программируется с помощью следующей рекурсивной процедуры:

```
Procedure Vbr(i:Index);
  begin
      if < приемлемо включение i-го элемента > then
        begin
            <включение i-го элемента в выборку>;
            if i<n then Vbr(i+1)
                    else <проверка оптимальности>;
            <исключение i-го элемента из выборки>;
        end;
      if <приемлемо невключение i-го элемента > then
        if i<n then Vbr(i+1)
                 else <проверка оптимальности>;
  end;  // Vbr
```

С помощью приведенной рекурсивной процедуры Vbr(i) описывается процесс исследования на пригодность *i*-го элемента к включению в выборку и генерацию всех выборок с проверкой на оптимальность. При рассмотрении каждого элемента (кандидата на включение) возможны два заключения: включать или не включать элемент в текущую выборку. Причем условия включения и невключения в общем случае разные.

Ниже приведен листинг программы, реализующей решение сформулированной задачи методом ветвей и границ. Здесь введен массив элементов *a*, имеющих тип записи с полями *w* и *c*. При генерации текущей и оптимальной выборок используются множества S и optS из индексов, входящих в выборку элементов. В процедуре Vbr(i,tw,ac) введены дополнительно два формальных параметра: tw — суммарный вес текущей выборки и ac – общая ее стоимость, т.е. стоимость, которую еще можно достичь, продолжая текущую выборку на данном пути. Критерием <приемлемо включение *i*-го элемента> является тот факт, что он подходит по весовым ограничениям, т.е. tw+a[i].w≤Wmax.

Если элемент не подходит по этому критерию, то попытки добавить еще один элемент в текущую выборку можно прекратить. Если речь идет об исключении, то критерием <приемлемости, невключения>, т.е. возможности продолжения построения текущей выборки, будет то, что после данного исключения общая стоимость текущей выборки ac будет не меньше полученной до этого стоимости maxC оптимальной выборки, находящейся в optS: ac-a[i].c>maxC.

ВЕДЬ ЕСЛИ СУММА МЕНЬШЕ, ТО ПРОДОЛЖЕНИЕ ПОИСКА, ХОТЯ ОН И МОЖЕТ ДАТЬ НЕКОТОРОЕ РЕШЕНИЕ, НЕ ПРИВЕДЕТ К ОПТИМАЛЬНОМУ РЕШЕНИЮ. СЛЕДОВАТЕЛЬНО, ДАЛЬНЕЙШИЙ ПОИСК НА ТЕКУЩЕМ ПУТИ БЕСПОЛЕЗЕН.

## 2.1. ЭВРИСТИЧЕСКИЕ МЕТОДЫ

МЫ УЖЕ ВИДЕЛИ, ЧТО ДЛЯ ПРОГРАММИРОВАНИЯ ПОИСКА В ДЕРЕВЕ РЕШЕНИЙ РЕКУРСИВНЫЕ ПРОЦЕДУРЫ ОКАЗЫВАЮТСЯ ДОВОЛЬНО ЭФФЕКТИВНЫМ СРЕДСТВОМ.

ЕСЛИ, ОДНАКО, ДЕРЕВО ОЧЕНЬ БОЛЬШОЕ, НАПРИМЕР, ЕСЛИ В ЗАДАЧЕ ОПТИМАЛЬНОГО ВЫБОРА $N$=100 ДЕРЕВО СОДЕРЖИТ БОЛЕЕ $10^{20}$ УЗЛОВ, ТО ДАЖЕ СОВРЕМЕННЫЙ КОМПЬЮТЕР МЕТОДОМ ВЕТВЕЙ И ГРАНИЦ БУДЕТ РЕШАТЬ ЭТУ ЗАДАЧУ БОЛЕЕ 1 МЛН ЛЕТ.

ДЛЯ ОГРОМНЫХ ДЕРЕВЬЕВ ОСТАЕТСЯ ЕДИНСТВЕННЫЙ СПОСОБ — ИСПОЛЬЗОВАТЬ ЭВРИСТИЧЕСКИЙ МЕТОД. НАЙДЕННЫЙ ЭВРИСТИЧЕСКИМ МЕТОДОМ ВАРИАНТ МОЖЕТ ОКАЗАТЬСЯ НЕ НАИЛУЧШИМ ИЗ ВОЗМОЖНЫХ, НО ЗАЧАСТУЮ БЛИЗКИМ К НЕМУ. ЭВРИСТИЧЕСКИЕ МЕТОДЫ ПОЗВОЛЯЮТ ИССЛЕДОВАТЬ ПРАКТИЧЕСКИ ЛЮБОЕ ДЕРЕВО.

НЕКОТОРЫЕ ЭВРИСТИЧЕСКИЕ МЕТОДЫ РЕШЕНИЯ ЗАДАЧИ ОПТИМАЛЬНОГО ВЫБОРА:

**МЕТОД МАКСИМАЛЬНОЙ СТОИМОСТИ:**

ПРИ КАЖДОМ ВКЛЮЧЕНИИ НОВОГО ЭЛЕМЕНТА В ВЫБОРКУ ВЫБИРАЕТСЯ ЭЛЕМЕНТ, ИМЕЮЩИЙ МАКСИМАЛЬНУЮ ЦЕНУ, ДО ТЕХ ПОР, ПОКА СУММАРНЫЙ ВЕС МЕНЕЕ *WMAX*.

РЕШАЕТСЯ ЗАДАЧА ОЧЕНЬ ПРОСТО, БЕЗ ВСЯКИХ РЕКУРСИЙ, ОСОБЕННО ЕСЛИ МАССИВ ЭЛЕМЕНТОВ ВНАЧАЛЕ ОТСОРТИРОВАТЬ ПО СТОИМОСТИ *C*.

**МЕТОД НАИМЕНЬШЕГО ВЕСА:**

ЭТА СТРАТЕГИЯ В НЕКОТОРОМ СМЫСЛЕ ПРОТИВОПОЛОЖНА ПРЕДЫДУЩЕЙ.

НА КАЖДОМ ШАГЕ ВЫБИРАЕТСЯ ЭЛЕМЕНТ С МИНИМАЛЬНЫМ ВЕСОМ. В ЭТОМ СЛУЧАЕ МЫ СФОРМИРУЕМ ВЫБОРКУ С МАКСИМАЛЬНЫМ КОЛИЧЕСТВОМ ЭЛЕМЕНТОВ.

**МЕТОД СБАЛАНСИРОВАННОЙ СТОИМОСТИ:**

ЭТА ЭВРИСТИКА СОСТОИТ В ТОМ, ЧТО ПРИ ВКЛЮЧЕНИИ ОЧЕРЕДНОГО ЭЛЕМЕНТА В ВЫБОРКУ СРАВНИВАЕТСЯ КАК СТОИМОСТЬ, ТАК И ВЕС, А ИМЕННО, ЭЛЕМЕНТ С САМЫМ БОЛЬШИМ ОТНОШЕНИЕМ СТОИМОСТИ К ВЕСУ.

**МЕТОД СЛУЧАЙНОГО ПОИСКА:**

ИСПОЛЬЗУЯ ДАТЧИК СЛУЧАЙНЫХ ЧИСЕЛ, АЛГОРИТМ ВЫБИРАЕТ ОЧЕРЕДНОЙ ЭЛЕМЕНТ СЛУЧАЙНЫМ ОБРАЗОМ. ПОИСК ОСУЩЕСТВЛЯЕТСЯ НЕСКОЛЬКО РАЗ. ДАННЫЙ МЕТОД ДАЕТ УДИВИТЕЛЬНО ХОРОШИЕ РЕЗУЛЬТАТЫ.

СРАВНЕНИЕ:

РАЗЛИЧНЫЕ ЭВРИСТИЧЕСКИЕ МЕТОДЫ ВЕДУТ СЕБЯ ПО-РАЗНОМУ В РАЗЛИЧНЫХ ЗАДАЧАХ. РЕКОМЕНДУЕТСЯ ПОПРОБОВАТЬ РЕШИТЬ ЗАДАЧУ ВСЕМИ ВАМ ИЗВЕСТНЫМИ ЭВРИСТИЧЕСКИМИ МЕТОДАМИ И ВЫБРАТЬ НАИЛУЧШЕЕ ИЗ ПОЛУЧЕННЫХ РЕШЕНИЙ.

СЛЕДУЕТ ОТМЕТИТЬ, ЧТО УСЛОВИЯ ОПТИМАЛЬНОСТИ В КАЖДОЙ ЗАДАЧЕ ЗАДАЮТСЯ ПО-РАЗНОМУ И НЕ ВСЕГДА ТАК ПРОСТО СФОРМУЛИРОВАТЬ УСЛОВИЯ ОТСЕЧЕНИЯ ПЕРСПЕКТИВНЫХ ВЕТВЕЙ В МЕТОДЕ ВЕТВЕЙ И ГРАНИЦ. БОЛЕЕ ТОГО, ДАЖЕ МЕТОД СБАЛАНСИРОВАННОЙ СТОИМОСТИ НЕ ВСЕГДА ТАК ПРОСТО РЕАЛИЗОВАТЬ. ЧАСТО ПРИХОДИТСЯ ПРИДУМЫВАТЬ СВОИ ЭВРИСТИКИ ПРИ РЕШЕНИИ ОЧЕРЕДНОЙ СЛОЖНОЙ ЗАДАЧИ. ВОТ ЗДЕСЬ УЖЕ НАЧИНАЕТСЯ ИСКУССТВО ПРОГРАММИРОВАНИЯ.

## 2.1. Порядок написания программы

Задание: Составить программу решения задачи оптимального выбора различными методами.

Текст программы приведен ниже.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, Buttons, Grids;

type
  TForm1 = class(TForm)
    Button1: TButton;
    StringGrid1: TStringGrid;
    Memo1: TMemo;
    Edit1: TEdit;
    Label1: TLabel;
```

```
    Edit2: TEdit;
    Label2: TLabel;
    BitBtn1: TBitBtn;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
Const     nmax=10;  // <=256
Type
          Telem=Record c,w:Cardinal end;
Var       i,n:byte;
          a:array[1..nmax] of Telem;
          S,optS:Set of byte;
          Wmax,maxC,acm:Cardinal;
procedure TForm1.FormCreate(Sender: TObject);
begin
   Edit1.Text:='3';
   Edit2.Text:='10';
   StringGrid1.Cells[1,0]:='вес';
   StringGrid1.Cells[2,0]:='цена';
    for i:=1 to 10 do StringGrid1.Cells[0,i]:=IntToStr(i);
    for i:=1 to 3 do begin
          StringGrid1.Cells[1,i]:=IntToStr(i*i);
          StringGrid1.Cells[2,i]:=IntToStr(i+5);
                end;
    Memo1.Clear;
end;

// Метод ветвей и границ
Procedure Vbr(i:byte;tw,ac:Cardinal);
        Var ac1:Cardinal;
        Begin  // Попытка включения
```

```
        if tw+a[i].w<=Wmax then begin
              Include(s,i);
              if i<n then Vbr(i+1,tw+a[i].w,ac)  // Продвижение влево
              else if ac>maxC then begin maxC:=ac;optS:=S end;
              Exclude(S,i);
                                                  end;
          //Попытка исключения
        ac1:=ac-a[i].c;
        if ac1>maxC then
              if i<n then Vbr(i+1,tw,ac1)   // Продвижение вправо
                        else begin maxC:=ac1; optS:=S end;
      End;//Vbr

procedure TForm1.Button1Click(Sender: TObject);
begin

  n:=strToInt(Edit1.Text); acm:=0;
    for i:=1 to n do begin
     a[i].w:=StrToInt(StringGrid1.Cells[1,i]);
     a[i].c:=StrToInt(StringGrid1.Cells[2,i]);
     acm:=acm+a[i].c end;
        Wmax:=strToInt(Edit2.Text);
        maxC:=0; S:=[]; optS:=[];
        Vbr(1,0,acm);
       memo1.Lines.Add('Оптимальный вариант');
       memo1.Lines.Add(' i       w      c');
        for i:=1 to n do
        if i IN optS then
       memo1.Lines.Add(IntToStr(i)+'       '+
                  Inttostr(a[i].w)+'      '+
                  Inttostr(a[i].c));
        memo1.Lines.Add('maxC='+Inttostr(maxC)+
                   ' Wmax='+Inttostr(Wmax));
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
Memo1.Clear;
end;

end.
```

Панель диалога будет иметь вид, показанный на рис. 2.3.

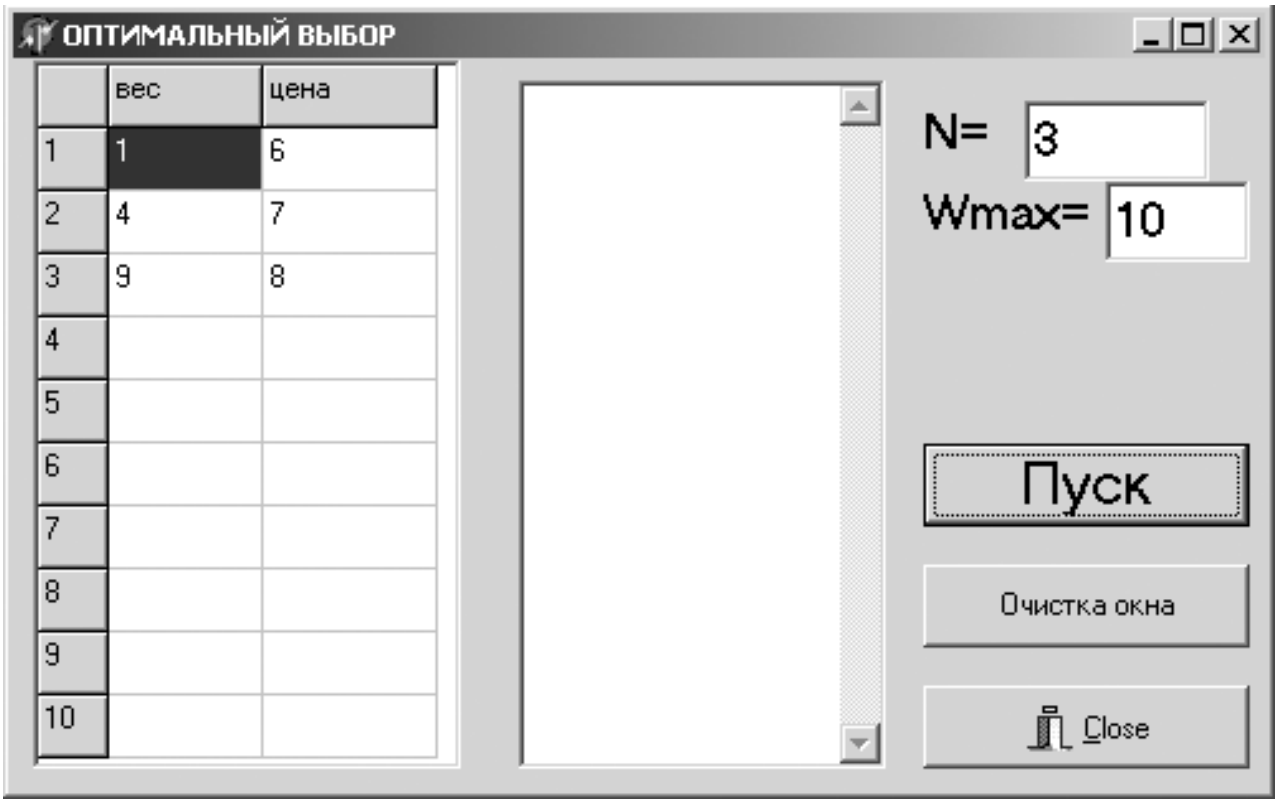



Рис. 2.3

### 2.3. Варианты задач

Задана таблица из 10 элементов:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Вес | $10+N_v$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| Цена | 18 | 20 | 17 | 19 | $28-N_v$ | 21 | 27 | 23 | 25 | 24 |

Здесь $N_v$ — номер варианта 1–15.

1. Для двух значений *Wmax*=50 и *Wmax*=10 найти оптимальные варианты и построить дерево поиска, поясняющее работу алгоритма, для чего в нужных местах вставить вывод промежуточных значений.

2. Решить эту же задачу методом полного перебора для n=5, построить полное дерево поиска, получить оценку эффективности метода ветвей и границ по отношению к методу полного перебора.

3. Решить методом максимальной стоимости.

4. Решить методом минимального веса.

5. Решить методом сбалансированной стоимости.

6. Решить методом случайного поиска (метод Монте-Карло).

Сравнить полученные решения и время выполнения для n=10, 20, 40.

ТЕМА 3. ПОИСК И СОРТИРОВКА МАССИВОВ

***Цель лабораторной работы:*** изучить способы сортировки и поиска в массивах записей и файлах

## 3.1. Организация работы с базами данных

Для создания и обработки всевозможных баз данных широко применяются массивы записей [1, тема 10].

Обычно база данных накапливается и хранится на диске. К ней часто приходится обращаться, обновлять, перегруппировывать. Работа с базой может быть организована двумя способами:

1. Вносить изменения и осуществлять поиск можно прямо на диске, используя специфическую технику работы с записями на файлах, при этом временные затраты на обработку данных (поиск, сортировку) значительно возрастают, но нет ограничений на оперативную память.
2. В начале работы вся база (или ее необходимая часть) считывается в массивы записей и обработка производится в оперативной памяти, что значительно сокращает ее время, однако требует затрат памяти.

Наиболее частыми операциями при работе с базами данных являются «поиск» и «сортировка». При этом алгоритмы решения этих задач существенно зависят от того, организованы записи в массивы или размещены на диске.

Обычно запись содержит некое ключевое слово (ключ), по которому ее находят среди множества других аналогичных записей. В вышеприведенном примере в зависимости от решаемой задачи ключом может служить фамилия или номер группы или адрес. Основное требование к ключу в задачах поиска состоит в том, чтобы операция проверки на равенство была корректной. Поэтому, например, в качестве ключа не следует выбирать действительное число, т.к. из-за всегда возможной ошибки округления поиск нужного ключа может оказаться безрезультатным, хотя этот ключ в массиве имеется.

## 3.2. Поиск в массиве записей

Задача поиска требуемого элемента в массиве записей *a[i], i=1...n* заключается в нахождении индекса *i,* удовлетворяющего условию a[i].k=x. Здесь ключ *k* выделен в отдельное поле, x - аргумент поиска того же типа что и *k*. После нахождения *i* обеспечивается доступ ко всем другим полям найденной записи a[i].

**Линейный поиск** используется, когда нет никакой дополнительной информации о разыскиваемых данных. Он представляет собой последовательный перебор массива до обнаружения требуемого ключа или до конца, если ключ не обнаружен:

```
i:=1;   n1:=n+1;
While(i<n1) and (a[i].k<>x) do i:=i+1;
if i=n1 then “элемент не найден” else элемент i’;
```

Видно, что на каждом шаге требуется увеличивать индекс и вычислять логическое выражение. А можно ли уменьшить затраты на поиск? Единствен-

ная возможность – попытаться упростить логическое выражение с помощью введения вспомогательного элемента — *барьера*, который предохраняет от перехода за пределы массива:

```
a[n+1].k=x; i=1;
While a[i].k<>x do i:=i+1;
if i=n+1 then ‘элемент не найден’ else ‘элемент i’;
```

**Поиск делением пополам** используется, когда данные упорядочены, например, по возрастанию ключа *k*, т.е. a[i]. k≤a[i+1]. k. *Основная идея* — возьмем «средний» (*m*) элемент. Если a[m]. k<x, то все элементы $i \le m$ можно исключить из дальнейшего поиска, если a[m]. k≥x, то можно исключить все $i > m$:

```
i=1; j:=N;
While i<j do
  begin
       m:=(i+j) div 2;
       if a[m].k<x then i:=m+1 else j:=m
  end;
  if a[i].k=x then ‘элемент i найден’ else ‘нет’;
```

В этом алгоритме отсутствует проверка внутри цикла совпадения a[m].k=x. На первый взгляд это кажется странным, однако тестирование показывает, что в среднем выигрыш от уменьшения количества проверок превосходит потери от нескольких «лишних» вычислений до выполнения условия $i=j$.

### 3.3. Сортировка массивов

Под сортировкой понимается процесс перегруппировки элементов массива, приводящий к их упорядоченному расположению относительно ключа.

Цель сортировки — облегчить последующий поиск элементов. Метод сортировки называется устойчивым, если в процессе перегруппировки относительное расположение элементов с равными ключами не изменяется. Основное условие при сортировке массивов — это не вводить дополнительных массивов, т.е. все перестановки элементов должны выполняться «на том же месте» в исходном массиве. Сортировку массивов принято называть **внутренней** в отличие от сортировки файлов (списков), которую называют **внешней**.

Методы внутренней сортировки классифицируются по времени их работы. Хорошей мерой эффективности может быть число сравнений ключей — *C* и число пересылок элементов — *P*. Эти числа являются функциями *C(n), P(n)* от числа сортируемых элементов *n*. **Быстрые** (но сложные) алгоритмы сортировки требуют (при $n \to \infty$) порядка $n \log n$ сравнений, **прямые** (простые) методы — $n^2$.

Прямые методы коротки, просто программируются, быстрые, усложненные, методы требуют меньшего числа операций, но эти операции обычно сами более сложны, чем операции прямых методов, поэтому для достаточно малых *n* $(n \le 50)$ прямые методы работают быстрее. Значительное преимущество быстрых методов (в $n/\log(n)$ раз) начинает проявляться при $n \gtrsim 100$.

Среди простых методов наиболее популярны:

1) **Метод прямого обмена** (пузырьковая сортировка [2, лекция 6]).

2) **Метод прямого выбора**:

```
for i=1 to n begin
m=i; for j=i+1 to n do
  if a[j].k<a[m].k then m=j;
     r=a[m];  a[m]:=a[i];  a[i]=r;
              end;
```

Реже используются [3]:

3) Сортировка с помощью прямого (двоичного) включения;

4) **Шейкерная** сортировка (модификация пузырьковой).

Улучшенные методы сортировки [3]:

1) **Метод Д. Шелла** (1959), усовершенствование метода прямого включения.

2) Сортировка с помощью дерева, метод **HeapSort**, Д. Уильямсон (1964).

3) Сортировка с помощью разделения, метод **QuickSort**, Ч. Хоар (1962), улучшенная версия пузырьковой сортировки. На сегодняшний день это самый эффективный метод сортировки.

Идея метода разделения **QuickSort** в следующем.

Выберем значение ключа среднего $m$-го элемента x=a[m].k. Будем просматривать массив слева до тех пор, пока не обнаружим элемент a[i].k>x. После этого будем просматривать массив справа, пока не обнаружим a[j].k<x. Поменяем местами элементы a[i] и a[j] и продолжим такой процесс просмотра (слева и справа, обмен), пока оба просмотра не встретятся где-то внутри массива. В результате массив окажется разбитым на левую часть $a[l],\ 1 \le l \le j$ с ключами меньше (или равными) $x$ и правую $a[p],\ i \le p \le n$ с ключами больше (или равными) $x$.

Алгоритм такого разделения очень прост и эффективен:

```
 i:=1, j:=n;
Repeat
  While a[i].k<x do i:=i+1;
  While x<a[j].k do j:=j-1;
    if i<=j then begin
    w:=a[i];a[i]:=a[j];a[j]:=w;i:=i+1;j:=j-1 end
  Until i>j;
```

Чтобы отсортировать массив, остается применять алгоритм разделения к левой и правой частям, затем к частям частей и так до тех пор, пока каждая из частей не будет состоять из одного единственного элемента. Алгоритм получается итерационным. На каждом этапе возникают две задачи по разделению. К решению одной из них можно приступить сразу, для другой следует заполнить начальные условия в список (номер разделения, границы и отложить ее решение до момента окончания сортировки выбранной половины. Требования из списка выполняются несколько специфическим образом, в обратном порядке (записанное последним выбирается первым). В нижеприведенном алгоритме для организации списка введен массив $stak[s],\ 0 \le s \le M$.

Алгоритм этого метода можно записать следующим образом:

```
Unit Sort;
Intarface
        const Mr=10000;
        type
          Tzp=record
           zp:Tinf;   // Поле информации
           k:Tk;      // Поле ключа
          end;
    Mas=array[1..Mr] of Tzp;   // Массив записей
Procedure QuickSort(Var a:mas;n:Word);    // n≤Mr
Implementation
Procedure QuickSort;
    const M=12;
    Var i,j,L,R:Word; x:Tk; w:Tzp;
        s:0..M;
        stak:array[1..M] of record L,R:word end;
    begin
          s:=1; stak[1].L:=1; stak[1].R:=n;
    Repeat    // Выбор из Stak последнего запроса
          L:=stak[s].L; R:stak[s].R; s:=s-1;
      Repeat  // Разделение a[L]...a[R]
          i:=L; j:=R; x:=a[(L+R) div 2].k;
       Repeat
        While a[i].k<x do i:=i+1;
        While x<a[j].k do j:=j-1;
        if i<=j then begin
        w:=a[i];a[i]=a[j];a[j]:=w;i=i+1;j=j-1 end;
       until i>j;
        if j-L<R-i then   // Выбор, какая половина длиннее?
        begin
        if i<R then       // Запись в Stak запроса из правой части
         begin s:=s+1; stak[s].L:=i; stak[s].R:=R; end
          R:=j;
        end               // Теперь L и R ограничивают левую часть
                    else
        begin
        if L<j then       // Запись в Stak запроса из левой части
         begin s:=s+1;stak[s].L:=L;stak[s].R:=j end
         L:=i
        end;
      until L>=R;     // Конец разделения очередной части
    until s=0;//stak пуст
    End;              // Конец процедуры QuickSort.
    End.
```

Сравнение методов сортировок показывает, что при n>100 наихудшим является метод пузырька, метод QuickSort в 2–3 раза лучше, чем HeapSort, и в 3–7 раз, чем метод Шелла.

### 3.4. Индивидуальные задания

Написать программу, записывающую в файл и читающую из файла массив из записей. Написать следующие процедуры и организовать их вызов: процедуру линейного поиска в файле; процедуру сортировки массива и файла методами прямого выбора и QuickSort; процедуру двоичного поиска в отсортированном массиве. При создании процедур предусмотреть передачу массива через формальные параметры, а тип ключа – используя глобальную переменную.

1. В МАГАЗИНЕ ФОРМИРУЕТСЯ СПИСОК ЛИЦ, ЗАПИСАВШИХСЯ НА ПОКУПКУ ТОВАРА. КАЖДАЯ ЗАПИСЬ ЭТОГО СПИСКА СОДЕРЖИТ: ПОРЯДКОВЫЙ НОМЕР, Ф.И.О., ДОМАШНИЙ АДРЕС ПОКУПАТЕЛЯ И ДАТУ ПОСТАНОВКИ НА УЧЕТ. УДАЛИТЬ ИЗ СПИСКА ВСЕ ПОВТОРНЫЕ ЗАПИСИ, ПРОВЕРЯЯ Ф.И.О. И ДОМАШНИЙ АДРЕС. КЛЮЧ: ДАТА ПОСТАНОВКИ НА УЧЕТ.

2. СПИСОК ТОВАРОВ, ИМЕЮЩИХСЯ НА СКЛАДЕ, ВКЛЮЧАЕТ В СЕБЯ НАИМЕНОВАНИЕ ТОВАРА, КОЛИЧЕСТВО ЕДИНИЦ ТОВАРА, ЦЕНУ ЕДИНИЦЫ И ДАТУ ПОСТУПЛЕНИЯ ТОВАРА НА СКЛАД. ВЫВЕСТИ В АЛФАВИТНОМ ПОРЯДКЕ СПИСОК ТОВАРОВ, ХРАНЯЩИХСЯ БОЛЬШЕ МЕСЯЦА, СТОИМОСТЬ КОТОРЫХ ПРЕВЫШАЕТ 1000000 Р. КЛЮЧ: НАИМЕНОВАНИЕ ТОВАРА.

3. ДЛЯ ПОЛУЧЕНИЯ МЕСТА В ОБЩЕЖИТИИ ФОРМИРУЕТСЯ СПИСОК СТУДЕНТОВ, КОТОРЫЙ ВКЛЮЧАЕТ Ф.И.О. СТУДЕНТА, ГРУППУ, СРЕДНИЙ БАЛЛ, ДОХОД НА ЧЛЕНА СЕМЬИ. ОБЩЕЖИТИЕ В ПЕРВУЮ ОЧЕРЕДЬ ПРЕДОСТАВЛЯЕТСЯ ТЕМ, У КОГО ДОХОД НА ЧЛЕНА СЕМЬИ МЕНЬШЕ ДВУХ МИНИМАЛЬНЫХ ЗАРПЛАТ, ЗАТЕМ ОСТАЛЬНЫМ В ПОРЯДКЕ УМЕНЬШЕНИЯ СРЕДНЕГО БАЛЛА. ВЫВЕСТИ СПИСОК ОЧЕРЕДНОСТИ ПРЕДОСТАВЛЕНИЯ МЕСТ В ОБЩЕЖИТИИ. КЛЮЧ: ДОХОД НА ЧЛЕНА СЕМЬИ.

4. В СПРАВОЧНОЙ АВТОВОКЗАЛА ХРАНИТСЯ РАСПИСАНИЕ ДВИЖЕНИЯ АВТОБУСОВ. ДЛЯ КАЖДОГО РЕЙСА УКАЗАНЫ ЕГО НОМЕР, ТИП АВТОБУСА, ПУНКТ НАЗНАЧЕНИЯ, ВРЕМЯ ОТПРАВЛЕНИЯ И ПРИБЫТИЯ. ВЫВЕСТИ ИНФОРМАЦИЮ О РЕЙСАХ, КОТОРЫМИ МОЖНО ВОСПОЛЬЗОВАТЬСЯ ДЛЯ ПРИБЫТИЯ В ПУНКТ НАЗНАЧЕНИЯ РАНЬШЕ ЗАДАННОГО ВРЕМЕНИ. КЛЮЧ: ВРЕМЯ ПРИБЫТИЯ.

5. НА МЕЖДУГОРОДНОЙ АТС ИНФОРМАЦИЯ О РАЗГОВОРАХ СОДЕРЖИТ ДАТУ РАЗГОВОРА, КОД И НАЗВАНИЕ ГОРОДА, ВРЕМЯ РАЗГОВОРА, ТАРИФ, НОМЕР ТЕЛЕФОНА В ЭТОМ ГОРОДЕ И НОМЕР ТЕЛЕФОНА АБОНЕНТА. ВЫВЕСТИ ПО КАЖДОМУ ГОРОДУ ОБЩЕЕ ВРЕМЯ РАЗГОВОРОВ С НИМ И СУММУ. КЛЮЧ: ОБЩЕЕ ВРЕМЯ РАЗГОВОРОВ.

6. ИНФОРМАЦИЯ О СОТРУДНИКАХ ФИРМЫ ВКЛЮЧАЕТ: Ф.И.О., ТАБЕЛЬНЫЙ НОМЕР, КОЛИЧЕСТВО ПРОРАБОТАННЫХ ЧАСОВ ЗА МЕСЯЦ, ПОЧАСОВОЙ ТАРИФ. РАБОЧЕЕ ВРЕМЯ СВЫШЕ 144 Ч СЧИТАЕТСЯ СВЕРХУРОЧНЫМ И ОПЛАЧИВАЕТСЯ В ДВОЙНОМ РАЗМЕРЕ. ВЫВЕСТИ РАЗМЕР ЗАРАБОТНОЙ ПЛАТЫ КАЖДОГО СОТРУДНИКА ФИРМЫ ЗА ВЫЧЕТОМ ПОДОХОДНОГО НАЛОГА, КОТОРЫЙ СОСТАВЛЯЕТ 12% ОТ СУММЫ ЗАРАБОТКА. КЛЮЧ: РАЗМЕР ЗАРАБОТНОЙ ПЛАТЫ.

7. ИНФОРМАЦИЯ ОБ УЧАСТНИКАХ СПОРТИВНЫХ СОРЕВНОВАНИЙ СОДЕРЖИТ: НАИМЕНОВАНИЕ СТРАНЫ, НАЗВАНИЕ КОМАНДЫ, Ф.И.О. ИГРОКА, ИГРОВОЙ НОМЕР, ВОЗРАСТ, РОСТ, ВЕС. ВЫВЕСТИ ИНФОРМАЦИЮ О САМОЙ МОЛОДОЙ КОМАНДЕ. КЛЮЧ: ВОЗРАСТ.

8. ДЛЯ КНИГ, ХРАНЯЩИХСЯ В БИБЛИОТЕКЕ, ЗАДАЮТСЯ: РЕГИСТРАЦИОННЫЙ НОМЕР КНИГИ, АВТОР, НАЗВАНИЕ, ГОД ИЗДАНИЯ, ИЗДАТЕЛЬСТВО, КОЛИЧЕСТВО СТРАНИЦ. ВЫВЕСТИ СПИСОК КНИГ С ФАМИЛИЯМИ АВТОРОВ В АЛФАВИТНОМ ПОРЯДКЕ, ИЗДАННЫХ ПОСЛЕ ЗАДАННОГО ГОДА. КЛЮЧ: АВТОР.

9. РАЗЛИЧНЫЕ ЦЕХА ЗАВОДА ВЫПУСКАЮТ ПРОДУКЦИЮ НЕСКОЛЬКИХ НАИМЕНОВАНИЙ. СВЕДЕНИЯ О ВЫПУЩЕННОЙ ПРОДУКЦИИ ВКЛЮЧАЮТ: НАИМЕНОВАНИЕ, КОЛИЧЕСТВО, НОМЕР ЦЕХА. ДЛЯ ЗАДАННОГО ЦЕХА НЕОБХОДИМО ВЫВЕСТИ КОЛИЧЕСТВО ВЫПУЩЕННЫХ ИЗДЕЛИЙ ПО КАЖДОМУ НАИМЕНОВАНИЮ В ПОРЯДКЕ УБЫВАНИЯ КОЛИЧЕСТВА. КЛЮЧ: КОЛИЧЕСТВО ВЫПУЩЕННЫХ ИЗДЕЛИЙ.

10. ИНФОРМАЦИЯ О СОТРУДНИКАХ ПРЕДПРИЯТИЯ СОДЕРЖИТ: Ф.И.О., НОМЕР ОТДЕЛА, ДОЛЖНОСТЬ, ДАТУ НАЧАЛА РАБОТЫ. ВЫВЕСТИ СПИСКИ СОТРУДНИКОВ ПО ОТДЕЛАМ В ПОРЯДКЕ УБЫВАНИЯ СТАЖА. КЛЮЧ: ДАТА НАЧАЛА РАБОТЫ.

11. ВЕДОМОСТЬ АБИТУРИЕНТОВ, СДАВШИХ ВСТУПИТЕЛЬНЫЕ ЭКЗАМЕНЫ В УНИВЕРСИТЕТ, СОДЕРЖИТ: Ф.И.О., АДРЕС, ОЦЕНКИ. ОПРЕДЕЛИТЬ КОЛИЧЕСТВО АБИТУРИЕНТОВ, ПРОЖИВАЮЩИХ В Г. МИНСКЕ И СДАВШИХ ЭКЗАМЕНЫ СО СРЕДНИМ БАЛЛОМ НЕ НИЖЕ 4.5, ВЫВЕСТИ ИХ ФАМИЛИИ В АЛФАВИТНОМ ПОРЯДКЕ. КЛЮЧ: Ф.И.О.

12. В СПРАВОЧНОЙ АЭРОПОРТА ХРАНИТСЯ РАСПИСАНИЕ ВЫЛЕТА САМОЛЕТОВ НА СЛЕДУЮЩИЕ СУТКИ. ДЛЯ КАЖДОГО РЕЙСА УКАЗАНЫ: НОМЕР РЕЙСА, ТИП САМОЛЕТА, ПУНКТ НАЗНАЧЕНИЯ, ВРЕМЯ ВЫЛЕТА. ВЫВЕСТИ ВСЕ НОМЕРА РЕЙСОВ, ТИПЫ САМОЛЕТОВ И ВРЕМЕНА ВЫЛЕТА ДЛЯ ЗАДАННОГО ПУНКТА НАЗНАЧЕНИЯ В ПОРЯДКЕ ВОЗРАСТАНИЯ ВРЕМЕНИ ВЫЛЕТА. КЛЮЧ: ПУНКТ НАЗНАЧЕНИЯ.

13. У АДМИНИСТРАТОРА ЖЕЛЕЗНОДОРОЖНЫХ КАСС ХРАНИТСЯ ИНФОРМАЦИЯ О СВОБОДНЫХ МЕСТАХ В ПОЕЗДАХ НА БЛИЖАЙШУЮ НЕДЕЛЮ В СЛЕДУЮЩЕМ ВИДЕ: ДАТА ВЫЕЗДА, ПУНКТ НАЗНАЧЕНИЯ, ВРЕМЯ ОТПРАВЛЕНИЯ, ЧИСЛО СВОБОДНЫХ МЕСТ. ОРГКОМИТЕТ КОНФЕРЕНЦИИ ОБРАЩАЕТСЯ К АДМИНИСТРАТОРУ С ПРОСЬБОЙ ЗАРЕЗЕРВИРОВАТЬ $M$ МЕСТ ДО ГОРОДА $N$ НА $K$-Й ДЕНЬ НЕДЕЛИ С ВРЕМЕНЕМ ОТПРАВЛЕНИЯ ПОЕЗДА НЕ ПОЗДНЕЕ $T$ ЧАСОВ ВЕЧЕРА. ВЫВЕСТИ ВРЕМЯ ОТПРАВЛЕНИЯ ИЛИ СООБЩЕНИЕ О НЕВОЗМОЖНОСТИ ВЫПОЛНИТЬ ЗАКАЗ В ПОЛНОМ ОБЪЕМЕ. КЛЮЧ: ЧИСЛО СВОБОДНЫХ МЕСТ.

14. ВЕДОМОСТЬ АБИТУРИЕНТОВ, СДАВШИХ ВСТУПИТЕЛЬНЫЕ ЭКЗАМЕНЫ В УНИВЕРСИТЕТ, СОДЕРЖИТ: Ф.И.О. АБИТУРИЕНТА, ОЦЕНКИ. ОПРЕДЕЛИТЬ СРЕДНИЙ БАЛЛ ПО УНИВЕРСИТЕТУ И ВЫВЕСТИ СПИСОК АБИТУРИЕНТОВ, СРЕДНИЙ БАЛЛ КОТОРЫХ ВЫШЕ СРЕДНЕГО БАЛЛА ПО УНИВЕРСИТЕТУ. ПЕРВЫМИ В СПИСКЕ ДОЛЖНЫ ИДТИ СТУДЕНТЫ, СДАВШИЕ ВСЕ ЭКЗАМЕНЫ НА 5. КЛЮЧ: СРЕДНИЙ БАЛЛ.

15. В радиоателье хранятся квитанции о сданной в ремонт радиоаппаратуре. Каждая квитанция содержит следующую информацию: наименование группы изделий(телевизор, радиоприемник и т. п.),марку изделия, дату приемки в ремонт, состояние готовности заказа (выполнен, не выполнен). Вывести информацию о состоянии заказов на текущие сутки по группам изделий. Ключ: дата приемки в ремонт.

ТЕМА 4. Указатели и их использование при работе со списками на основе динамических массивов

***Цель лабораторной работы:*** изучить способы работы с динамическими массивами данных и организацию работы со списками, используя указатели.

### 4.1. Динамическое распределение памяти

В языке Паскаль возможна организация динамического распределения памяти, при которой оперативная память для размещения данных выделяется непосредственно во время выполнения программы по мере надобности, после чего освобождается для работы с другими данными. Для этой цели используются переменные специального типа — **указатели**, которые обеспечивают работу непосредственно с адресами ячеек оперативной памяти.

Вводятся указатели следующим образом:

```
Type Puk=^<тип переменной>;
Var    p, q : pointer;     // Нетипизируемые указатели
       a, b : Puk;         // Типизированные указатели
       c    : extended; // Обычная переменная
begin
 ...
 p:=Addr(c);        // В p заносится адрес переменной c
 a:=p;
 p:=Nil;            // Очистка адреса
 ...
```

ВСЯ СВОБОДНАЯ ОТ ПРОГРАММ ПАМЯТЬ КОМПЬЮТЕРА ПРЕДСТАВЛЯЕТ СОБОЙ МАССИВ БАЙТ, КОТОРЫЙ НАЗЫВАЕТСЯ ***КУЧЕЙ***. КОГДА ВОЗНИКАЕТ НЕОБХОДИМОСТЬ ИСПОЛЬЗОВАНИЯ ПРОГРАММОЙ ДОПОЛНИТЕЛЬНОЙ ПАМЯТИ, ЭТО ОСУЩЕСТВЛЯЕТСЯ ОДНОЙ ИЗ ПРОЦЕДУР NEW ИЛИ GETMEM.

Процедура

```
New(a:<типизированный указатель>);
```

находит в куче свободный участок памяти, размер которого позволяет разместить тип данных *a* и присваивает указателю *a* значение адреса первого байта этого участка. После этого данный участок памяти закрепляется за программой и с ним можно работать через возникшую в программе переменную *a^*. Такие переменные называются **динамическими**. После того как необходимость работы с этой переменной отпала, данный участок памяти освобождается с помощью процедуры

```
Dispose(a);
```

При работе как с типизированными, так и с нетипизированными указателями аналогичные действия выполняют процедуры

```
GetMem(P:pointer;size:Word);
FreeMem(P,size);
```

здесь *size* — количество байт выделяемой памяти начиная с адреса, помещаемого в указатель *P*.

### 4.2. Организация динамических массивов

ОБЫЧНО ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ И ОСВОБОЖДЕНИЕ ПАМЯТИ ИСПОЛЬЗУЕТСЯ ПРИ РАБОТЕ С БОЛЬШИМИ МАССИВАМИ ДАННЫХ.

В случае, **когда максимальные размеры массива заранее известны**, динамическое распределение памяти может быть организовано следующим образом:

```
type Tms=array[1..30,-5..5] of byte;
      Pms=^Tms;
Var b:Pms;      // Указатель на массив
      m,n,i,j:integer;
begin
      ....
      New(b);   // Выделение памяти
      Read(m,n);//1<=m<=30; -5<=n<=5
      for i:=1 to m do
      for j:=-5 to n do read(Fl,b[i,j]);
      <работа с массивом>
      ....
      Dispose(b);  // Освобождение памяти
....
```

ПРИ ТАКОЙ ОРГАНИЗАЦИИ ПАМЯТЬ ВЫДЕЛЯЕТСЯ НЕРАЦИОНАЛЬНО И ЗАВЕДОМО С ИЗБЫТКОМ, ПРИ ЭТОМ ВСЕГДА НУЖНО ПОМНИТЬ ОБ ОГРАНИЧЕНИИ НА ИНДЕКСЫ.

С помощью процедур Getmem и Freemem можно создавать **массивы с изменяемым размером – динамические массивы**. Для этого определим тип указателя на массив с небольшим размером, а затем выделим памяти столько, сколько необходимо:

```
type
      Tms=array[2..3] of <тип элементов>;  // Индексы начинаются с 2
      pms=^ms;                             // Указатель на массив
 Var a:pms; mt:word;
begin
      mt:=sizeof(<тип элемента>);  // Определение количества байт,
                                   // которое требуется для размещения элемента
  Read(n);
  GetMem(a,mt*n);            // Выделение памяти под n чисел
  for i:=2 to n do Read(Fl,a[i]);
 <добавление, удаление и обработка элементов при работе со списком>
  Sort(a,n);     // Сортировка
  for i:=2 to n do Write(Fl,a[i]);
  FreeMem(a, mt*n);    // Освобождение памяти
```

При работе с такой программой необходимо отключать проверку выхода индекса за пределы массива и внимательно следить за тем, чтобы индекс не вышел за пределы выделенной памяти.

### 4.3. Работа со списками

**СПИСОК** — ЭТО ГРУППА ОБЪЕКТОВ (ДАННЫХ), С КОТОРЫМИ НАДО РАБОТАТЬ В ОПЕРАТИВНОЙ ПАМЯТИ: ДОБАВЛЯТЬ В ГРУППУ НОВЫЕ ОБЪЕКТЫ, УДАЛЯТЬ ИСПОЛЬЗУЕМЫЕ, СОРТИРОВАТЬ, НАХОДИТЬ НУЖНЫЕ. В ПРОЦЕССЕ РАБОТЫ СПИСОК МОЖЕТ ВОЗРАСТАТЬ И УМЕНЬШАТЬСЯ.

ПРОСТЕЙШАЯ ФОРМА ОРГАНИЗАЦИИ СПИСКА — ЭТО МАССИВ ДАННЫХ. ДАННЫЕ, РАЗМЕЩЕННЫЕ В МАССИВЕ, ИМЕЮТ СВОЙ НОМЕР (ИНДЕКС) И ЭТО ПРИДАЕТ ЭФФЕКТ «ВИДИМОСТИ» КАЖДОМУ ЭЛЕМЕНТУ. МЫ УЖЕ НАУЧИЛИСЬ НЕКОТОРЫМ ПРИЕМАМ РАБОТЫ С МАССИВАМИ И ОЦЕНИЛИ ИХ ЭФФЕКТИВНОСТЬ.

РАССМОТРИМ, КАК ОРГАНИЗУЕТСЯ РАБОТА СО СПИСКАМИ НА ОСНОВЕ МАССИВА ПЕРЕМЕННОГО РАЗМЕРА.

Предположим, что мы создали начальный список в массиве *a* размером *n* с помощью вышеописанного фрагмента программы.

При добавлении нового элемента в список необходимо выделить память на 1 элемент большую, затем скопировать старый список в новый раздел памяти, добавить туда еще 1 элемент, после чего освободить ранее выделенную память и установить указатель **a** на новый раздел памяти.

Для организации добавления элемента an в позицию ipos напишем следующую процедуру:

```
procedure AddMs(an:<тип элемента>,ipos:<тип индекса>);
        var
          new_a:Pms;
        begin
          GetMem(new_a,(n+1)* mt);
          j:=1;
        for i:=2 to n+1 do  // Копирование старых элементов
        if i<>ipos then begin j:=j+1;  new_a^[i]:=a^[j]end
                    else new_a[i]:=an;   // Добавление нового
        FreeMem(a,mt*n); // Освобождение ранее выделенной памяти
        a:=new_a; n:=n+1; // Теперь имя опять а
end;
```

Это простая схема хорошо работает для небольших списков, но у нее есть существенный недостаток: при каждом добавлении приходится менять размер выделенной памяти и, что еще хуже, копировать.

Процедура удаления *i*-го элемента тоже требует кроме выделения новой памяти и копирования еще и «схлопывания», т.е. сдвига каждого элемента на одну позицию, чтобы заполнить освободившуюся *i*-ю позицию.

Иногда хороший выход из этой ситуации — организация изменения размера порциями (например по 20 элементов). Можно сделать размер порций зависимыми от текущей длины массива (например 10% от n).

### 4.4. Индивидуальные задания

ОРГАНИЗОВАТЬ РАБОТУ СО СПИСКАМИ ЗАПИСЕЙ НА ОСНОВЕ ДИНАМИЧЕСКОГО МАССИВА. ИНДИВИДУАЛЬНЫЕ ВАРИАНТЫ ТИПОВ ЗАПИСЕЙ ИСПОЛЬЗОВАТЬ ТЕ ЖЕ, ЧТО И В ЛАБОРАТОРНОЙ РАБОТЕ № 3.

1.СОСТАВИТЬ ПРОЦЕДУРЫ ДОБАВЛЕНИЯ ЭЛЕМЕНТА В СПИСОК И УДАЛЕНИЯ ИЗ СПИСКА. ПРИ ДОБАВЛЕНИИ ИЛИ УДАЛЕНИИ НОВЫХ ЗАПИСЕЙ ПРИ НЕОБХОДИМОСТИ ПАМЯТЬ ВЫДЕЛЯТЬ (УМЕНЬШАТЬ) ПОРЦИЯМИ ЗАДАННЫХ РАЗМЕРОВ.

2. СОСТАВИТЬ ПРОЦЕДУРЫ СОРТИРОВКИ СПИСКА ПО КЛЮЧУ, МОДИФИЦИРУЯ И ИСПОЛЬЗУЯ СТАНДАРТНЫЕ ПРОЦЕДУРЫ ИЗ ТЕМЫ 3.

3. ОРГАНИЗОВАТЬ ЗАПИСЬ СПИСКА В ФАЙЛ И ЧТЕНИЕ СПИСКА ИЗ ФАЙЛА.

4. НАЧАЛЬНЫЙ СПИСОК ПРОЧЕСТЬ ИЗ ФАЙЛА.

5. РЕЗУЛЬТАТЫ ОТОБРАЗИТЬ В ОКНО TMEMO.

## ТЕМА 5. ОРГАНИЗАЦИЯ ОДНОНАПРАВЛЕННОГО СПИСКА НА ОСНОВЕ РЕКУРСИВНЫХ ДАННЫХ В ВИДЕ СТЕКА

***Цель лабораторной работы:*** изучить возможности работы со стеком в динамической памяти.

### 5.1. Определение стека

Стек — это структура данных, организованная по принципу «первый вошел — последний вышел». Образно это можно представить, как запаянную с одной стороны трубку, в которую закатываются шарики. Первый шарик всегда будет доставаться из трубки последним. Элементы в стек можно добавлять или извлекать только через его вершину. Программно стек реализуется в виде однонаправленного списка с одной точкой входа (вершиной стека).

Для работы со стеком вводится следующий рекурсивный тип:

```
Type
  TSel=^Sel; // Указатель на запись
  Sel=Record;    // Сама запись элемента стека
    inf:TInf;      // Информационная часть элемента стека,
                   // например Tinf=String
    A : TSel;      // Указатель на предыдущий элемент стека
  End;
```

Для работы со стеком целесообразно организовывать библиотеку, в которую входят следующие подпрограммы.

#### Добавление элемента в стек

```
Procedure AddStack(var Sp:TSel;S:TInf); // Здесь Sp – указатель на
                    // предыдущий элемент стека, s – добавляемая строка.
Var SPt:TSel;    // Временный указатель на элемент стека
Begin
  New(SPt);    // Выделение памяти под новый элемент стека
  SPt^.inf:=s; // Запись информационной части
```

```
  SPt^.A:=Sp; // Запоминание указателя на предыдущий элемент стека
  SP:=Spt;       // Sp – теперь новый указатель на вершину стека
End;
```

**Извлечение информации из стека с освобождением памяти**

```
Procedure ReadStack(var Sp:TSel; var S:TInf); //Здесь Sp - указатель на
                                  // вершину стека, S - извлекаемая информация.
Var SPt:TPz;
Begin
       if SP=nil then ShowMessage('Стек пуст')
                     else Begin
  S:=Sp^.inf;       // Извлечение из стека строки
  SPt:=Sp;          // Запоминание указателя на элемент стека
  SP:=Sp^.A;      // Sp теперь указывает на предыдущий элемент стека
  Dispose(Spt);  // Освобождение памяти под элемента стека
                              End;
End;
```

**Чтение элемента стека с заданным номером *n* без освобождения памяти**

```
  Procedure ReadNStack(Sp:TSel;n:integer;var S:TInf); // Здесь Sp –
                  // указатель на вершину стека, n - номер элемента стека
                 // (с вершины), значения которого нужно прочитать.
Var i:integer; // Текущий номер элемента стека
Begin
 i:=1;
While (I<>n) or (Sp<>Nil) do
  begin
    Inc(i);
     Sp:=Sp^.A;
  end;
 if I=n then S:=Sp^.inf
       else ShowMessage('Элемент < n');
end;
```

### 5.2 Сортировка однонаправленных списков

При поиске информации в списке, выводе данных обычно список упорядочивают (сортируют) по ключу.

***Метод пузырьковой сортировки*** основан на переставлении местами двух соседних элементов. Поменять местами два соседних элемента в однонаправленном списке можно двумя способами.

Первый способ основан на перестановке адресов двух соседних элементов, следующих за элементом с известным указателем *spi*. В этом случае пер-

вый элемент стека не используется и называется **меткой**. В него при необходимости записывается служебная информация.

```
Procedure Rev12After(spi:Tsel);
 Var sp:Tsel;
  begin
        sp:=spi^.A^.A;
        spi^.A^.A:=sp^.A;
        sp^.A:=spi^.A;
        spi^.A:=sp;
   end;
```

Второй способ основан на обмене информации между ячейкой с текущим указателем spi и следующей за ней.

```
Procedure Rev12Inf(spi:Tsel);
 Var Inf:TInf;
  begin
        Inf:=spi^.Inf;
        spi^.Inf:=sp^.A^.Inf;
        spi^.A^.inf:=Inf;
   end;
```

Первый способ более приемлем, если элементы содержат большие массивы информации.

Теперь можно записать два способа пузырьковой сортировки:

Вариант сортировки стека с пересылкой ключей:

```
Procedure SortBublAfter(spm1:Tsel);
 Var sp,spt:Tsel;
  begin
        if spm1^.A=Nil then exit;
        spt:=Nil;
        Repeat
          sp:=spm1;
          While sp^.A^.A<>spt do begin
          if sp^.A^.Inf.key>sp^.A^.A^.Inf.key
            then Rev12After(sp);
          sp:=sp^.A;
                                                end;
            spt:=sp^.A;
        Until spm1^.A^.A=spt;
   end;
```

ВАРИАНТ СОРТИРОВКИ СТЕКА С ПЕРЕСЫЛКОЙ ИНФОРМАЦИИ:

```
Procedure SortBublInf(sp1:Tsel);
 Var sp,spt:Tsel;
  begin
        spt:=Nil;
```

```
      Repeat
        sp:=sp1;
        While sp^.A<>spt do begin
      if sp^.Inf.key>sp^.A^.Inf.key then Rev12Inf(sp);
        sp:=sp^.A;
                                                end;
          spt:=sp
      Until sp1^.A:=spt;
end;
```

### 5.3.Индивидуальные задания

В заданиях (1–15) реализовать сортировку стека двумя методами пузырька. Результат формирования и преобразования стека показывать в компонентах TlistBox.

СОЗДАТЬ СТЕК СО СЛУЧАЙНЫМИ ЦЕЛЫМИ ЧИСЛАМИ И ВЫЧИСЛИТЬ СРЕДНЕЕ АРИФМЕТИЧЕСКОЕ И СРЕДНЕКВАДРАТИЧНЫЙ РАЗБРОС ВОЗЛЕ СРЕДНЕГО.

1. Создать стек со случайными целыми числами в диапазоне –50 до +50 и преобразовать его в два стека. Первый должен содержать только положительные числа, а второй – отрицательные. Порядок чисел должен быть сохранен, как в первом стеке.

2. Создать стек из случайных целых чисел и удалить из него записи с четными числами.

3. Создать стек из случайных целых чисел в диапазоне от –10 до 10 и удалить из него записи с отрицательными числами.

4. Создать стек из случайных целых чисел и поменять местами крайние элементы.

5. Подсчитать, сколько элементов стека, построенного из случайных чисел, превышает среднее значение от всех элементов стека.

6. Создать стек из случайных целых чисел и найти в нем максимальное и минимальное значение.

7. Создать стек из случайных целых чисел и определить, сколько элементов стека, начиная с вершины, находится до элемента с максимальным значением.

8. Создать стек из случайных целых чисел и определить, сколько элементов стека, начиная с вершины, находится до элемента с минимальным значением.

9. Создать стек из случайных чисел и определить, сколько элементов стека находится между минимальным и максимальным элементами.

10. Создать стек из случайных чисел и определить, сколько элементов стека имеют значения меньше среднего значения от всех элементов стека.

11. Создать стек из случайных чисел и поменять местами минимальный и максимальный элементы.

12. Создать стек из случайных целых чисел и из него сделать еще два стека. В первый поместить все четные, а во второй — нечетные числа.

13. Создать стек из случайных целых чисел в диапазоне от 1 до 10 и определить наиболее часто встречающееся число.

14. Создать стек из случайных целых чисел и удалить из него каждый второй элемент.

15. Создать стек из 10 произвольных строк. Замкнуть его в кольцо и, задав произвольное целое число (n), организовать выбывание элементов кольца через n шагов по кольцу. Определить последнюю оставшуюся строку. Аналог игры «На золотом крыльце сидели ...».

16. Создать стек из произвольного числа строк и реверсировать его, т.е. порядок следования элементов стека должен быть обратным.

17. Создать стек из произвольного числа строк и упорядочить его элементы в алфавитном порядке, используя метод «пузырька» (можно менять местами только два соседних элемента).

18. Создать два стека из упорядоченных по возрастанию целых случайных чисел. Объединить их в один стек с сохранением упорядоченности по возрастанию значений.

19. Создать стек из произвольного числа строк. Из него создать новый стек, куда бы входили в алфавитном порядке буквы, входящие в строки первого стека.

20. Создать стек из произвольного числа строк. Из него создать новый стек, в котором бы строки располагались в порядке увеличения числа символов в строке.

21. Создать стек из произвольного числа строк. Из него создать новый стек, в котором бы строки располагались в алфавитном порядке.

22. Создать стек из произвольного числа случайных комплексных чисел. Определить, сколько элементов стека имеют модуль комплексного числа больше среднего значения.

23. С помощью стека создать игру «минная дорожка». В стек случайным образом занести 100 нулей и 10 единиц. 1 означает наличие мины, 0 — ее отсутствие на дорожке. Игрок должен последовательно вводить произвольные числа от 1 до 10, пока не достигнет дна стека. Если на его пути встречаются мины (значение 1), он проигрывает и игра начинается сначала.

24. Создать два упорядоченных по алфавиту стека со словами строкового типа. Объединить их в один стек с сохранением упорядоченности по алфавиту.

25. Создать три стека со случайными целыми числами без перекрытия их значений. Объединить их в один стек, соединив дно одного с вершиной другого и т.д.

26. Создать стек, содержащий координаты точек на плоскости. Найти наиболее удаленные между собой точки на плоскости.

27. Создать стек, содержащий координаты точек на плоскости. Найти геометрический центр этих точек и найти точку, к нему ближайшую.

28. Создать стек, содержащий координаты точек на плоскости. Найти геометрический центр этих точек и исключить из стека элемент, наиболее удаленный от центра.

29. Создать стек из случайных чисел. Преобразовать стек в кольцо. Посмотреть, к какому результату приведет циклическая операция, когда значение каждого элемента стека будет определяться как умноженное на 2 и с вычетом значения предыдущего элемента стека.

## ТЕМА 6. ОРГАНИЗАЦИЯ ОДНОНАПРАВЛЕННОГО И ДВУНАПРАВЛЕННОГО СПИСКОВ В ВИДЕ ОЧЕРЕДИ НА ОСНОВЕ РЕКУРСИВНЫХ ДАННЫХ

***Цель лабораторной работы:*** изучить возможности работы со списками, организованными в виде очереди.

### 6.1. Очередь на основе односвязанного списка

Очередь по своей структуре очень похожа на стек, но только она работает по принципу «первый вошел — первый вышел». Ее образно можно представить в виде трубки, открытой с двух сторон. С одной стороны мы закатываем шарики, а с другого конца извлекаем.

Для элемента очереди вводится такой же рекурсивный тип, как и для стека.

```
Type
      TSel=^Sel;
       Sel=Record
         Inf:TInf;    // TInf – тип информации об элементе списка
         A:TSel;  // Адрес следующей ячейки
       end;
```

***Добавление новой записи*** в конец очереди осуществляется следующей процедурой:

```
Procedure Dobk(var sp1,spk:Tsel;Inf:TInf);
begin
  if spk=Nil then begin
    New(spk);
    Spk^.A:=Nil;
    Spk^.Inf:=inf;
    Sp1:=spk;
                end
          else begin
    New(spk^.A);
    spk:=spk^.A;
    spk^.Inf:=Inf;
    spk^.A:=Nil;
              end;
End;
```

В результате первого обращения к этой подпрограмме (Spk=Nil) в списке размещается одна заявка. При этом инициируются два указателя: sp1 — адрес 1-й записи, spk — адрес последней.

***Добавление новой записи*** в начало очереди:

```
Procedure Dob1(var sp1,spk:Tsel;Inf:TInf);
  Var sp:Tsel;
begin
        if sp1=Nil then begin
                    New(sp1);
                    sp1^.A:=Nil;
                    sp1^.Inf:=Inf;
                    spk:=sp1;
                        end
                    else begin
                  New(sp);
                  sp^.Inf:=Inf;
                  sp^.A:=sp1;
                  sp1:=sp;
                  end;
end;
```

***Прочесть и стереть информацию из начала очереди*** можно с помощью следующей процедуры:

```
        Procedure Red1(var sp1,Spk:Tsel;var Inf:Tinf);
         Var sp:Tsel;
        begin
                 Inf:=sp1.Inf;
                  sp:=sp1;
                  sp1:=sp1^.A;
                  Dispose(sp);
            if sp1=nil then spk:=nil;
        end;
```

***Распечатать очередь*** можно, используя следующую процедуру:

```
        Procedure Wrt(sp1:Tsel);
         Var sp:Tsel;
                begin
                  sp:=sp1;
                  While sp <> Nil do
                        begin
                          Writeln(sp^.Inf);
                          sp:=sp^.A;
                        end;
                end;
```

***Чтение и удаление элемента из середины однонаправленного списка*** сделать просто, если известен адрес ячейки spi, находящейся **перед** удаляемой:

```
Procedure ReadAfter(spi:Tsel;var Inf:TInf);
 Var sp:Tsel;
  begin
        Inf:=spi^.Inf;
        sp:=spi^.A;  //  В sp адрес, следующий за spi
        spi^.A:=sp^.A;
        Dispose(sp);
  end;
```

***Добавление элемента в середину однонаправленного списка*** после элемента с адресом *spi*:

```
Procedure DobAfter(spi:Tsel,Inf:TInf);
 Var sp:Tsel;
  begin
   New(sp);
        sp^.Inf:=Inf;
        sp^.A:=spi^.A;
        spi^.A:=sp;
  end;
```

***Чтение и удаление ячейки с адресом spi и добавление перед spi*** осуществляется более сложно, с использовнаием ReadAfter; DobAfter:

```
Procedure Redi(spi:Tsel;Inf:TInf);
 Var Inf:Tinf;
 begin;
   Inf:spi^.A^.Inf;        // Запоминание информации, следующей за spi
   ReadAfter(spi,Inf);  // Удаление следующей за spi
   spi^.Inf:=Inf;          // Восстановление Inf уже в spi
 end;
Procedure DobBefore(spi:Tsel;Inf:Tinf);
 begin
   DobAfter(spi,spi^.Inf);   // Вставка spi после spt
   spi^.Inf:=Inf;
 end;
```

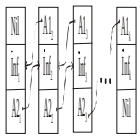
## 6.2. Двухсвязанные списки

**Двухсвязанные списки** организуются, когда требуется просматривать список как в одном, так и в обратном направлении. Мы видели, что в однонаправленном списке довольно просто удалить (вставить) новую ячейку после заданной, но довольно сложно удалить саму заданную ячейку или предыдущую. Эта проблема легко решается, если ввести рекурсивный тип с двумя адресными ячейками:

```
Type
        Tseld=^seld;
        seld=record
          Inf:Tinf;
```

```
        A1:Tseld;
        A2:Tseld;
      End;
Var spdm1,spdmk,spd:Tseld;
```

В A1 засылается адрес предыдущего элемента, в A2 — последующего, причем в первой ячейке с адресом spdm1 — значение A1=Nil, в последней ячейке с адресом spdmk — значение A2=Nil:



***Создание двухсвязанного пустого списка*** с двумя метками входа реализуется с помощью следующей процедуры:

```
Procedure Newspd(spdm1,spdm2:Tseld);
 begin
      New(spdm1); New(spdmk);
      Spdm1^.A1:=Nil; Spdm1^.A2:=spdmk;
      Spdmk^.A1:=Spdm1; Spdmk^.A2:=Nil;
 end;
```

***Добавление элемента после заданного в двухсвязанный список***:

```
Procedure DobdAfter(Inf:TInf; spdi:Tseld);
 Var spd:Tseld;
 begin
      New(spd);
      Spd^.Inf:=Inf;
      Spd^.A1:=Spdi;
      Spd^.A2:=Spdi.A2;
      Spdi^.A2:=Spd;
      Spdi^.A2.A1:=spd;
 end;
```

***Добавление элемента перед заданным***:

```
Procedure DobdBefor(Inf:TInf; spdi:Tzapd);
 Var spd:Tseld;
 begin
      New(spd);
      Spd^.Inf:=Inf;
      Spd^.A2:=Spdi;
      Spd^.A1:=Spdi.A1;
      Spdi^.A1:=Spd;
      Spdi^.A1.A2:=spd;
 end;
```

***Чтение и удаление элемента с адресом spdi***:

```
Procedure Read(Spdi:Tzapd;var Inf:TInf);
  begin
        Inf:=spdi^.Inf;
        spdi^.A1^.A2:=spdi^.A2;
        spdi^.A2^.A1:=spdi^.A1;
        Dispose(spdi);
  end;
```

### 6.3. Сортировка очереди слиянием

Допустим, что есть два отсортированных в порядке возрастания списка *sq1, sqk, sr1, srk*. Построим алгоритм их слияния в один отсортированный список *spk*:

```
 Procedure Slip (Var sq1,sqk,sr1,srk,sp1,spk:Tsel);
 Var Infq,Infr:TInf;
 begin
   While(sq1<>Nil) and (sr1<>Nil) do begin
              Red1(sq1,sqk,Infq);
              Red1(sr1,srk,Infr);
             if Infq.Key<Infr.Key then begin
                  Dobk(sp1,spk,Infq);
                  Dob1(sr1,srk,Infr);
                        end
                             else begin
                  Dobk(sp1,spk,Infr);
                   Dob1(sq1,sqk,Infq);
                        end;
                               end;
       while sq1<>nil do begin
             Red1(sq1,sqk,Infq);
             Dobk(sp1,spk,Infq);
                       end;
      while sr1<>nil do begin
            Red1(sr1,srk,Infr);
             Dobk(sp1,spk,Infr);
               end;
       end;
```

***Разбиение списка на 2 списка.***

```
Procedure Div2s(var sp1,spk,sq1,sqk,sr1,srk:Tsel);
  Var Inf:TInf;bl:boolean;
  begin
  sr1:=nil; srk:=Nil; sq1:=Nil; sqk:=nil;
    bl:=true;
```

```
        While bl do begin
          Red1(sp1,spk,Inf);
         Dobk(sq1,sqk,Inf);
            bl:=(sp1<>Nil);
           If bl then begin
              Red1(sp1,spk,Inf);
             Dobk(sr1,srk,Inf);
                 bl:=(sp1<>Nil);
                   end;
                  end;
        end;
```

***Сортировка очереди слиянием (рекурсивная):***

```
Procedure Sortsl(var sp1,spk:Tsel
  Var sq1,sr1,sqk,srk:Tsel;
   begin
        if sp1<>spk then begin
        Div2s(sp1,spk,sq1,sqk,sr1,srk
        sortsl(sq1,sqk);
        sortsl(sr1,srk);
        slip(sq1,sqk,sr1,srk,sp1,spk);
                      end;
end;
```

### 6.4. Индивидуальные задания

Во всех задачах создать список из случайных чисел в виде очереди и отсортировать рекурсивным методом слияния, после чего выполнить задание в соответствии с вариантом. Отображать список в компонентах TListBox.

1. СОЗДАТЬ ОЧЕРЕДЬ ИЗ СЛУЧАЙНЫХ ЦЕЛЫХ ЧИСЕЛ. НАЙТИ МИНИМАЛЬНЫЙ ЭЛЕМЕНТ И СДЕЛАТЬ ЕГО ПЕРВЫМ.
2. СОЗДАТЬ ДВЕ ОЧЕРЕДИ ИЗ СЛУЧАЙНЫХ ЦЕЛЫХ ЧИСЕЛ. В ПЕРВОЙ НАЙТИ МАКСИМАЛЬНЫЙ ЭЛЕМЕНТ И ЗА НИМ ВСТАВИТЬ ЭЛЕМЕНТЫ ВТОРОЙ ОЧЕРЕДИ.
3. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК ИЗ СЛУЧАЙНЫХ ЦЕЛЫХ ЧИСЕЛ. УДАЛИТЬ ИЗ СПИСКА ВСЕ ЭЛЕМЕНТЫ, НАХОДЯЩИЕСЯ МЕЖДУ МАКСИМАЛЬНЫМ И МИНИМАЛЬНЫМ.
4. УПОРЯДОЧИТЬ ЭЛЕМЕНТЫ ДВУХСВЯЗАННОГО СПИСКА СЛУЧАЙНЫХ ЦЕЛЫХ ЧИСЕЛ В ПОРЯДКЕ ВОЗРАСТАНИЯ МЕТОДОМ «ПУЗЫРЬКА», КОГДА МОЖНО ПЕРЕСТАВЛЯТЬ МЕСТАМИ ТОЛЬКО ДВА СОСЕДНИХ ЭЛЕМЕНТА.
5. ПРЕДСТАВИТЬ ТЕКСТ ПРОГРАММЫ В ВИДЕ ДВУХСВЯЗАННОГО СПИСКА. ЗАДАТЬ НОМЕРА НАЧАЛЬНОЙ И КОНЕЧНОЙ СТРОК. ЭТОТ БЛОК СТРОК СЛЕДУЕТ ПЕРЕМЕСТИТЬ В ЗАДАННОЕ МЕСТО СПИСКА.
7. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК ИЗ СЛУЧАЙНЫХ ЦЕЛЫХ ЧИСЕЛ. ИЗ ЭЛЕМЕНТОВ, РАСПОЛОЖЕННЫХ МЕЖДУ МАКСИМАЛЬНЫМ И МИНИМАЛЬНЫМ, СОЗДАТЬ ПЕРВОЕ КОЛЬЦО. ОСТАЛЬНЫЕ ЭЛЕМЕНТЫ ДОЛЖНЫ СОСТАВИТЬ ВТОРОЕ КОЛЬЦО.
8. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК ИЗ СЛУЧАЙНЫХ ЦЕЛЫХ, ПОЛОЖИТЕЛЬНЫХ И ОТРИЦАТЕЛЬНЫХ ЧИСЕЛ. ИЗ ЭТОГО СПИСКА ОБРАЗОВАТЬ ДВА СПИ-

СКА, ПЕРВЫЙ ИЗ КОТОРОГО ДОЛЖЕН СОДЕРЖАТЬ ОТРИЦАТЕЛЬНЫЕ ЧИСЛА, А ВТОРОЙ — ПОЛОЖИТЕЛЬНЫЕ. ЭЛЕМЕНТЫ СПИСКОВ НЕ ДОЛЖНЫ ПЕРЕМЕЩАТЬСЯ В ПАМЯТИ.

9. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК ИЗ СТРОК ПРОГРАММЫ. ПРЕОБРАЗОВАТЬ ЕГО В КОЛЬЦО. ОРГАНИЗОВАТЬ ВИДИМУЮ В КОМПОНЕНТЕ МЕМО ЦИКЛИЧЕСКУЮ ПРОКРУТКУ ТЕКСТА ПРОГРАММЫ.

10. СОЗДАТЬ ДВА ДВУХСВЯЗАННЫХ СПИСКА ИЗ СЛУЧАЙНЫХ ЦЕЛЫХ ЧИСЕЛ. ВМЕСТО ЭЛЕМЕНТОВ ПЕРВОГО СПИСКА, ЗАКЛЮЧЕННЫХ МЕЖДУ МАКСИМАЛЬНЫМ И МИНИМАЛЬНЫМ, ВСТАВИТЬ ВТОРОЙ СПИСОК.

11. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК ИЗ СЛУЧАЙНЫХ ЦЕЛЫХ ЧИСЕЛ. УДАЛИТЬ ИЗ СПИСКА ЭЛЕМЕНТЫ С ПОВТОРЯЮЩИМИСЯ БОЛЕЕ ОДНОГО РАЗА ЗНАЧЕНИЯМИ.

12. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК И ПОМЕНЯТЬ В НЕМ ЭЛЕМЕНТЫ С МАКСИМАЛЬНЫМ И МИНИМАЛЬНЫМ ЗНАЧЕНИЯМИ, ПРИ ЭТОМ ЭЛЕМЕНТЫ НЕ ДОЛЖНЫ ПЕРЕМЕЩАТЬСЯ В ПАМЯТИ.

13. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК ИЗ НАРИСОВАННЫХ ВАМИ КАРТИНОК. ПРЕОБРАЗОВАТЬ ЕГО В КОЛЬЦО И ОРГАНИЗОВАТЬ ЕГО ЦИКЛИЧЕСКИЙ ПРОСМОТР В КОМПОНЕНТЕ IMAGE.

14. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК ИЗ СЛУЧАЙНЫХ ЧИСЕЛ. ПРЕОБРАЗОВАТЬ ЕГО В КОЛЬЦО. ПРЕДУСМОТРЕТЬ ВОЗМОЖНОСТЬ ДВИЖЕНИЯ ПО КОЛЬЦУ В ОБЕ СТОРОНЫ С ОТОБРАЖЕНИЕМ МЕСТА ПОЛОЖЕНИЯ ТЕКУЩЕГО ЭЛЕМЕНТА С ПОМОЩЬЮ КОМПОНЕНТЫ TGAUGE (GKPIE) И ЧИСЛОВОГО ЗНАЧЕНИЯ — С ПОМОЩЬЮ TLABEL.

15. СОЗДАТЬ ДВУХСВЯЗАННЫЙ СПИСОК ИЗ ТЕКСТА ВАШЕЙ ПРОГРАММЫ И ОТОБРАЗИТЬ ЕГО В TLISTBOX. ВЫДЕЛИТЬ В TLISTBOX ЧАСТЬ СТРОК И ОБЕСПЕЧИТЬ ЗАПОМИНАНИЕ ЭТИХ СТРОК. ДАЛЕЕ ВЫДЕЛИТЬ ЛЮБУЮ СТРОКУ И НАЖАТЬ КНОПКУ, КОТОРАЯ ДОЛЖНА ОБЕСПЕЧИВАТЬ ПЕРЕМЕЩЕНИЯ ВЫДЕЛЕННЫХ РАНЕЕ СТРОК ПЕРЕД ТЕКУЩЕЙ СТРОКОЙ. ПРИ ЭТОМ В TLISTBOX ДОЛЖНЫ ОТОБРАЖАТЬСЯ СТРОКИ ИЗ ДВУХ СВЯЗАННОГО СПИСКА.

# ТЕМА 7. ИСПОЛЬЗОВАНИЕ СТЕКА ДЛЯ ПРОГРАММИРОВАНИЯ АЛГОРИТМА ВЫЧИСЛЕНИЯ АЛГЕБРАИЧЕСКИХ ВЫРАЖЕНИЙ

***Цель лабораторной работы:*** изучить возможности работы со списками и очередями.

## 7.1. Задача вычисления арифметических выражений

ОДНОЙ ИЗ ЗАДАЧ ПРИ РАЗРАБОТКЕ ТРАНСЛЯТОРОВ ЯВЛЯЕТСЯ ЗАДАЧА РАСШИФРОВКИ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ, НАПРИМЕР:

$$r:=(a+b) * (c+d)-e;$$

В выражении ***a+b*** ***a*** и ***b*** – операнды, + операция. Такая запись называется **инфиксной** формой. Возможны также обозначения ***+ab*** — **префиксная**, ***ab+*** — **постфиксная** форма. В наиболее распространенной инфиксной форме для указания последовательности выполнения операций необходимо расставлять скобки. Польский математик Я. Лукашевич обратил внимание на тот факт, что при записи выражений в постфиксной форме скобки не нужны, а последователь-

ность операндов и операций удобна для расшифровки, основанной на применении эффективных методов. Поэтому постфиксная запись выражений получила название **обратной польской записи**. Например, в ОПЗ вышеприведенное выражение выглядит следующим образом:

***r=ab+cd+ * e–;***

АЛГОРИТМ ВЫЧИСЛЕНИЯ ТАКОГО ВЫРАЖЕНИЯ ОСНОВАН НА ИСПОЛЬЗОВАНИИ СТЕКА. ПРИ ПРОСМОТРЕ ВЫРАЖЕНИЯ СЛЕВА НАПРАВО КАЖДЫЙ ОПЕРАНД ЗАНОСИТСЯ В СТЕК. В РЕЗУЛЬТАТЕ ДЛЯ КАЖДОЙ ВСТРЕЧЕННОЙ ОПЕРАЦИИ ОТНОСЯЩИЕСЯ К НЕЙ ОПЕРАНДЫ БУДУТ ДВУМЯ ВЕРХНИМИ ЭЛЕМЕНТАМИ СТЕКА. БЕРЕМ ИЗ СТЕКА ЭТИ ОПЕРАНДЫ, ВЫПОЛНЯЕМ ОЧЕРЕДНУЮ ОПЕРАЦИЮ НАД НИМИ И РЕЗУЛЬТАТ ПОМЕЩАЕМ В СТЕК.

Алгоритм **преобразования выражения из инфиксной формы в форму обратной польской записи** (постфиксную) был предложен Дейкстрой. При его реализации вводится понятие стекового приоритета операций:

**Операции )( + - * / ^ (^ — возведение в степень)**

Приоритет 0 1 2 3

Просматривается слева направо исходная строка символов, в которой записано выражение в **инфиксной форме**, причем операнды переписываются в выходную строку, в которой формируется постфиксная форма выражения, а знаки операций заносятся в стек следующим образом:

1. Если стек пуст, то операция записывается в стек.
2. Открывающаяся скобка дописывается в стек.
3. Очередная операция выталкивает в выходную строку все операции из стека с большим или равным приоритетом.
4. Закрывающая скобка выталкивает все операции из стека до ближайшей открывающейся скобки в выходную строку, сами скобки уничтожаются.
5. Если после выборки из исходной строки последнего символа в стеке остались операции, то все они выталкиваются в выходную строку.

### 7.2. Порядок написания программы

Задание: Написать программу расшифровки и вычисления арифметических выражений с использованием стека.

Панель диалога будет иметь вид, показанный на рис. 7.1.

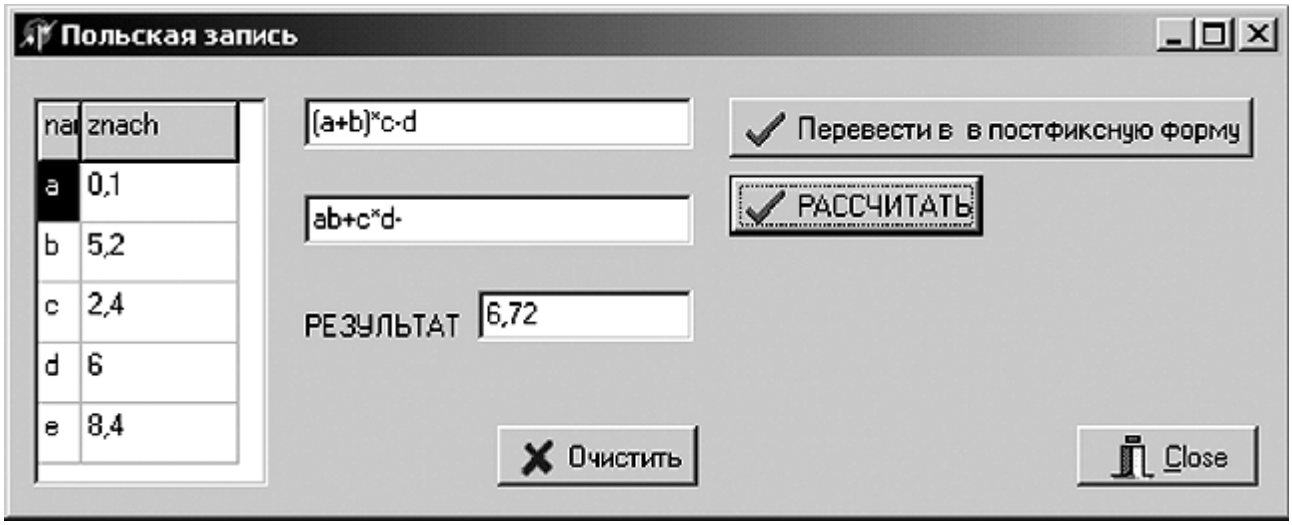



Рис. 7.1

Текст программы приведен ниже.

```
unit polsk;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, Buttons, Grids;

type
  TForm1 = class(TForm)
    StringGrid1: TStringGrid;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    BitBtn3: TBitBtn;
    BitBtn4: TBitBtn;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
    procedure BitBtn4Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }

  end;
```

```
  var Form1: TForm1;

implementation

{$R *.dfm}
 Type Tstek=^stek;
        stek=record
            Inf : Char;
            A   : Tstek;
         end;

  Var
   str      : string;
   mszn : array[97..200] of extended; //[a..z]
   stk     : Tstek;

Procedure Dob1(var P:Tstek;S:char); // Добавление элемента в стек
Var Pt:Tstek;    // Временный указатель на элемент стека
Begin
     if p=nil then begin
                    New(P);
                    P^.inf:=s;
                    P^.A:=nil;
                    end
      else begin
         New(Pt);    // Выделение памяти под новый элемент стека
         Pt^.inf:=s;  // Запись информационной части
         Pt^.A:=p;   // Запоминание указателя на предыдущий элемент стека
         P:=pt;
           end
End;

Procedure Red1(var p:Tstek; var S:char);
  Var Pt:Tstek;
 Begin
         S:=p^.inf;    // Извлечение строки из стека
         Pt:=p;
         p:=p^.A;       // p указывает на предыдущий элемент стека
        Dispose(pt);  // Освобождение памяти от элемента стека
End;

Function AV(strp:string):extended;
  var ch,ch1,ch2,chr:char;
```

```
      op1,op2,rez:extended;
      i:byte;
 begin
 chr:=Succ('z');
   for i:=1 to Length(strp)do begin
     ch:=strp[i];
      if not (ch in ['*','/','+','-','^']) then Dob1(stk,ch)
                                        else begin
                    Red1(stk,ch1);
                    Red1(stk,ch2);
                         op1:=mszn[ord(ch1)];
                          op2:=mszn[ord(ch2)];
                         case ch of
                         '+':rez:=op2+op1;
                         '-':rez:=op2-op1;
                         '*':rez:=op2*op1;
                         '/':rez:=op2/op1;
                         '^':rez:=exp(op1*ln(op2));
                         end;
                         mszn[ord(chr)]:=rez; Dob1(stk,chr);Inc(Chr);
                                        end;
                          end;
 Result:=rez;
end;

Function prior(ch:char):byte;
 begin
    case ch of
     '(',')' : Result:=0;
     '+','-' : Result:=1;
     '*','/' : Result:=2;
     '^'     : Result:=3;
     end;
end;

Procedure OBP(var stri,strp : string);
 Var stk:Tstek; pc:0..3; n,i:byte;ch,ch1:char;bl:boolean;
  begin
     strp:='';
     n:=length(stri);
     stk:=nil;
         for i:=1 to n do begin
                  ch:=stri[i];
           if ch in['+','-','*','/','(',')','^'] then
```

```
            if ch='(' then Dob1(stk,ch)
                       else
           if ch=')' then
                    Red1(stk,ch);
                    While ch<>'('do begin
                           strp:=strp+ch;
                           Red1(stk,ch);
                                  end
                           End
                      else
            if stk=Nil then Dob1(stk,ch) // Если стек пуст
                     else begin        // Выталкивание менее приоритетных
                         PC:=prior(ch);
                          Repeat
                            Red1(stk,ch1);
                            bl:=prior(ch1)>=pc;
                            if bl then strp:=strp+ch1
                                  else Dob1(stk,ch1);
                          Until(stk=Nil) or (Not bl);
                            Dob1(stk,ch);
                             end
                                       else
                              strp:=strp+ch; ..
          end;//for
       While stk<>nil do begin Red1(stk,ch);strp:=strp+ch;end;
  end; // Конец процедуры OBP

procedure TForm1.FormCreate(Sender: TObject);
begin
 with StringGrid1 do begin
 Cells[0,0]:='name'; Cells[1,0]:='znach';
 Cells[0,1]:='a';  Cells[1,1]:='0,1';
 Cells[0,2]:='b';  Cells[1,2]:='5,2';
 Cells[0,3]:='c';  Cells[1,3]:='2,4';
 Cells[0,4]:='d';  Cells[1,4]:='6';
 Cells[0,5]:='e';  Cells[1,5]:='8,4';
                  end;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
 var stri,strp:string;
begin
  stri:=Edit1.Text;
  OBP(stri,strp);
```

```
  Edit2.Text:=strp;
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
  var  i:byte;  Ch:Char;  strp:string;
begin
 strp:=Edit2.Text;
  For i:=1 to 5 do begin
     Ch:=StringGrid1.Cells[0,i][1];
     mszn[ord(Ch)]:=StrToFloat(StringGrid1.Cells[1,i]);
                 end;
  Edit3.Text:=FloatToStr(AV(strp));
end;

procedure TForm1.BitBtn4Click(Sender: TObject);
begin
Edit1.Clear;
Edit2.Clear;
Edit3.Clear;
end;

end.
```

### 7.3. Индивидуальные задания

В основе программы должны лежать два алгоритма: а) преобразование арифметического выражения из инфиксной формы в постфиксную (форму обратной польской записи); б) расшифровка и вычисление выражения в постфиксной форме.

У каждого варианта должно быть свое оригинальное арифметическое выражение, включающее операции + – ∗ / ^. Для контроля вычислить это же выражение обычным образом. Разработать удобный интерфейс ввода и вывода данных.

## ТЕМА 8. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ДЕРЕВЬЕВ НА ОСНОВЕ РЕКУРСИВНЫХ ТИПОВ ДАННЫХ

***Цель лабораторной работы:*** изучить способы программирования алгоритмов обработки данных с использованием древовидных структур. Получить навыки работы с компонентом TTreeView.

## 8.1. Понятие древовидной структуры

ДРЕВОВИДНАЯ МОДЕЛЬ ОКАЗЫВАЕТСЯ ДОВОЛЬНО ЭФФЕКТИВНОЙ ДЛЯ ПРЕДСТАВЛЕНИЯ ДИНАМИЧЕСКИХ ДАННЫХ С ЦЕЛЬЮ БЫСТРОГО ПОИСКА ИНФОРМАЦИИ.

Древовидное размещение списка данных (*abcdefgkl*) можно изобразить, например, следующим образом:

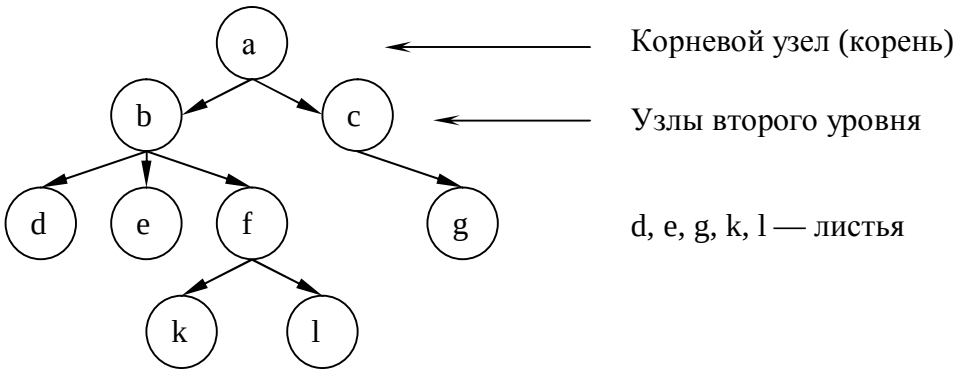



Рис. 8.1

Как видим, данные размещаются в узлах дерева, соединенных направленными дугами (ветвями). Если два узла соединены направленной дугой (*X*®*Y*), то узел *X* называется **предшественником** (родителем), а узел *Y* — **преемником** (дочерью). Деревья имеют единственный узел (корень), *у* которого нет родителя. Узел, не имеющий дочерей называется **листом**. **Внутренний** узел — это узел, не являющийся листом или корнем. **Порядок узла** — количество его дочерних узлов. **Степень дерева** — максимальный порядок его узлов. **Глубина узла** равна числу его предков плюс 1. **Глубина дерева** — это наибольшая глубина его узлов. Дерево, представленное выше, имеет степень 3 (троичное), глубину 4, корневой узел **a**.

Для реализации древовидных структур данных степени m используется следующая конструкция рекурсивного типа данных:

```
Type Ttree=^tree
        Tree=Record
         Inf:TInf;
          A1:Ttree;          // A1...Am – указатели на адреса,
       ...                   // по которым расположены сестры
          Am:Ttree;          // Если сестра отсутствует, то
        end;                 // соответствующий адрес равен Nil
```

РАЗМЕЩЕНИЕ ДАННЫХ В ВИДЕ ДРЕВОВИДНОЙ СТРУКТУРЫ, ПРЕДСТАВЛЕННОЙ НА РИС. 8.1, МОЖНО ОСУЩЕСТВИТЬ, НАПРИМЕР, СЛЕДУЮЩИМ ОБРАЗОМ:

```
Var proot, p  : Ttree;  // Указатели: на корень и текущий
...
New(proot);      P=Proot;    P^.Inf:='a'; P^.A2:=Nil;
New(P^.A3);      P:=P^.A3;   P^.Inf:='c'; P^.A1:=Nil;   P^.A2:=Nil;
New(P^.A3);      P:=P^.A3;   P^.Inf:='g'; P^.A1:=Nil;   P^.A2:=Nil;
                 P=Proot;
New(P^.A1);      P:=P^.A1;   P^.Inf:='b';
```

```
New(P^.A1);
        P^.A1.Inf:='d';  P^.A1^.A1:=Nil;  P^.A1^.A2:=Nil; P^.A1^.A3:=Nil;
New(P^.A2);
        P^.A2.Inf:='e';  P^.A2^.A1:=Nil;  P^.A2^.A2:=Nil;  P^.A2^.A3:=Nil
New(P^.A3);       P:=P^.A3;  P^.Inf:='f';   P^.A2:=Nil;
New(P^.A1);
        P^.A1.Inf:='k';  P^.A1^.A1:=Nil;  P^.A1^.A2:=Nil;  P^.A1^.A3:=Nil;
New(P^.A3);
        P^.A3.Inf:='l';  P^.A3^.A1:=Nil;  P^.A3^.A2:=Nil;  P^.A3^.A3:=Nil
```

После того как дерево заполнено информацией, его нужно уметь просмотреть, распечатать, осуществить поиск. Как видно из вышеприведенного примера, непосредственное заполнение даже небольшого дерева требует довольно громоздкой последовательности команд. Поэтому для работы с деревьями используют набор специфических алгоритмов. Последовательное обращение ко всем узлам называется **обходом дерева**. Следующая рекурсивная процедура осуществляет такой обход с распечаткой каждого узла:

```
Procedure WrtTree(var p:Ttree);   // p – указатель на текущий узел дерева
  begin
        if p<>Nil then begin   // Если дочь отсутствует, то выход
        <Print(p^.Inf);>         // При прямом обходе печать ставить здесь
        Wrttree(p^.A1);
        WrtTree(p^.A2);
        ...
        WrtTree(p^.Am);
        <Print(p^.Inf);>       // При обратном обходе печать ставить здесь
                               end;
  end;
```

ПРИ ВЫЗОВЕ ПРОЦЕДУРЫ :

```
        Wrttree(proot);
```

данные, изображенные на рисунке, будут распечатаны в следующем порядке:
Прямой обход (печать стоит в начале): *a, b, d, e, f, k, l, c, g.*
Обратный обход (печать стоит в конце): *d e k l f b g c a.*

Аналогично можно составить процедуру нахождения информации в дереве по заданному ключу, который находится в поле Inf.key:

```
 var blpoisk : Boolean;
 blpoisk:=True;
        …
Procedure PoiskTree(var p:Ttree; var Inf:TInf);
  begin
        if (p<>Nil) or blpoisk then
              Inf.key<>p^.Inf.key then begin
                          Wsttree(p^.A1,Inf);
                          ....
```

```
                    Wsttree(p^.Am,Inf);
                            end;
                    else begin
                    Inf:=p^.Inf;
                    blpoisk:=False;
                            end;
end;
```

### 8.2. Компонент TTreeView

Компонент TTreeView предназначен для отображения ветвящихся иерархических структур в виде горизонтальных деревьев, например, каталогов файловой системы дисков. Основным свойством этого компонента является Items, которое представляет собой массив элементов типа TTreeNode, каждый из которых описывает один узел дерева. Всего в Items — count узлов. Рассмотрим некоторые методы класса TTreeNodes.

Function AddFirst(Node:TTreeNode; const S:String):TTreeNode; — создает первый дочерний узел в узле Node. Если Node=Nil, то создается корневой узел. S — это строка содержания узла. Функция возвращает указатель на созданный узел.

Function AddChild(Node:TTreeNode; const S:String):TTreeNode; — добавляет очередной дочерний узел к узлу Node.

Function AddChildFirst(Node:TTreeNode; const S:String):TTreeNode; — добавляет новый узел как первый из дочерних узлов узла Node.

Procedure Clear; — очищает список узлов, описанных в свойстве Items.

Function FullExpand; — раскрывает все узлы при отображении дерева.

### 8.3. Бинарное дерево поиска

НАИБОЛЕЕ ЧАСТО ДЛЯ РАБОТЫ СО СПИСКАМИ ИСПОЛЬЗУЕТСЯ БИНАРНОЕ (ИМЕЮЩИЕ СТЕПЕНЬ 2) ДЕРЕВО ПОИСКА. ПРЕДПОЛОЖИМ, ЧТО У НАС ЕСТЬ НАБОР ДАННЫХ, УПОРЯДОЧЕННЫХ ПО КЛЮЧУ:

*key*: 1, 6, 8, 10, 20, 21, 25, 30.

Если организовать древовидную структуру со следующим распределением ключей (*N=Nil*):

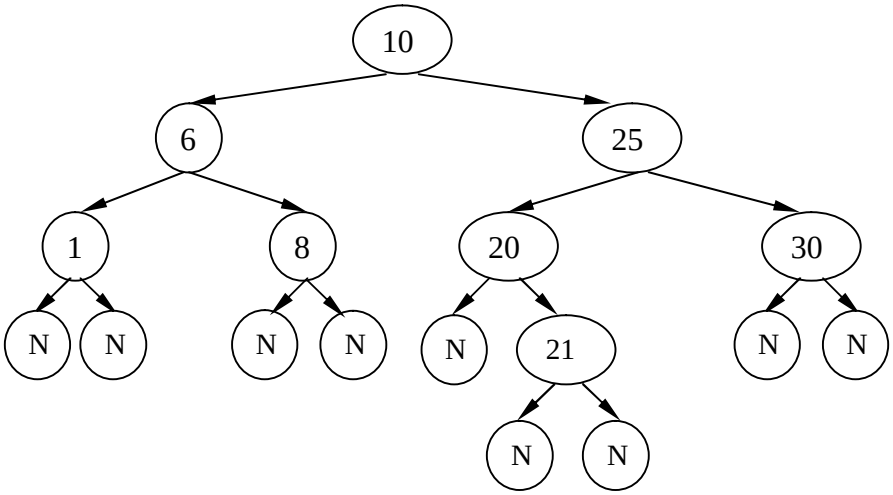



Рис. 8.2.

то обход дерева в **симметричном порядке** организуется следующей процедурой:

```
WrtS1(p:Ttree);
  begin
       if p<>nil then
         begin
              WrtS1(p^.A1);
              Print(p^.Inf);  // Печать ставится здесь
              WrtS1(p^.A2);
         end;
   end;
```

ПРИ ВЫЗОВЕ WRTS1(PROOT); БУДЕТ НАПЕЧАТАНА ИНФОРМАЦИЯ В ПОРЯДКЕ ВОЗРАСТАНИЯ КЛЮЧА. ЕСЛИ В ПРОЦЕДУРЕ ПОМЕНЯТЬ МЕСТАМИ A1 И A2, ТО ИНФОРМАЦИЯ БУДЕТ НАПЕЧАТАНА В ПОРЯДКЕ УБЫВАНИЯ КЛЮЧА.

В дереве на рис. 8.2 ключи расположены таким образом, что для любого узла значения ключа у левого преемника меньше, чем у правого. Таким образом, организованное дерево получило название **двоичное дерево поиска**. Ввиду его своеобразной организации эффективность поиска информации в такой динамической структуре данных сравнима с эффективностью двоичного поиска в массиве, т.е. $O(\log_2 n)$. Заметим, что двоичный поиск в линейном связанном списке организовать невозможно, а эффективность линейного поиска имеет порядок $O(n/2)$.

Конечно, оценка $O(\log_2 n)$ справедлива для **сбалансированного** дерева, т.е. такого, у которого узлы, имеющие только одну дочь, располагаются не выше двух последних уровней.

### 8.4. Основные операции с двоичным деревом поиска

ПРИ ОРГАНИЗАЦИИ СПИСКОВ В ВИДЕ ДВОИЧНОГО ДЕРЕВА НЕОБХОДИМ ПАКЕТ ПРОГРАММ, РЕАЛИЗУЮЩИХ СЛЕДУЮЩИЕ ДЕЙСТВИЯ:

- поиск заданного ключа;
- поиск минимального (максимального) ключа;
- вставка нового значения ключа, не изменяя свойств дерева поиска;
- удаление заданного ключа;

— формирование дерева поиска;
— балансировка дерева.

**Поиск в двоичном дереве** по заданному ключу Inf.key:

```
Procedure poisk(proot:Ttree;var Inf:TInf);
        P:Ttree;
  begin p:=Proot;
        While(p<>Nil) and (p^.Inf.key<>Inf.key) do
          Inf.key<p^.Inf.key then p:=p^.A1
                                  else p:=p^.A2;
  if p<>Nil then Inf:=p^.Inf
              else Write('Поиск без результата');
  end;
```

**Поиск информации с минимальным (максимальным) ключом**:

```
Procedure Minkey(Proot:Ttree;var Inf:TInf);
        P:Ttree;
  begin p:=Proot;
        While p^.A1<>Nil do p:=p^.A1;
        Inf:=p^.Inf;
  end;
```

При поиске максимального ключа нужно спуститься по правой ветке (p:=p^.A2) до самого правого листа.

**Вставить новый элемент с ключом key,** не совпадающим ни с одним из ключей:

Вначале напишем процедуру создания нового листа дерева:

```
Procedure MakeList(var p:Ttree;Inf:Tinf);  // Создание нового листа дерева
           begin
                 New(p);
                 p^.Inf:=Inf;
                 p^.A1:=Nil;
                 p^.A2:=Nil;
end;
Procedure Dobtree(Proot:Ttree;Inf:TInf);   // Добавление листа в дерево
   var P,q:Ttree;bl:Boolean;
  begin
  p:=Proot;
        While p<>Nil do
          begin                              // q – указатель на предыдущий элемент
                q:=p; bl:=Inf.key<p^.Inf.key;
          if bl then  p:=p^.A1
                else  p:=p^.A2;
          end;
        MakeList(p,Inf);
```

```
    if bl then q^.A1:=p else q^.A2:=p;
end;
```

**Построение дерева поиска**:

Пусть имеется некоторый массив из n значений данных с разными ключами a:array[1..n] of Tinf. Построение дерева поиска можно осуществить следующим образом:

```
Procedure Makeproot(var proot:Ttree);
 begin
      MakeList(proot,a[1]);
      For i:=2 to n do Dobtree(proot,a[i]);
 end;
```

При случайном чередовании ключей в массиве **a** обычно формируется дерево, неплохо сбалансированное. Однако, если ключи в исходном массиве **a** частично упорядочены, то дерево поиска оказывается сильно разбалансированным и его эффективность для организации поиска оказывается сравнимой с линейным поиском в массиве.

Этот способ построения дерева поиска можно использовать для **печати элементов массива** (например расположенного в файле) **в порядке возрастания**. Для этого достаточно построить и распечатать полученное дерево процедурой WrtS1(proot).

**Построение сбалансированного дерева поиска** для заданного массива **a** можно осуществить, если этот массив предварительно отсортирован в порядке возрастания ключа.

Следующая рекурсивная процедура строит идеально сбалансированное дерево поиска по отсортированному массиву:

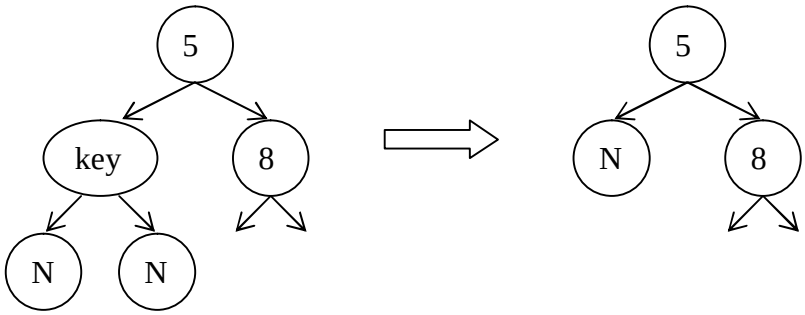
```
Var a:array[1..n] of TInf;
Function prootBLns(K,L:Word):Ttree;
Var p:Ttree; m:Word;
 begin
      if K>L then P:=Nil
            else begin
           m:=K+L div 2;
           New(p); P^.Inf:=a[m];
           P^.A1:=ProotBLns(K,m);
           P^.A2:=ProotBLns(m+1,L);
                 end;
      Result:=p;
 end;
....
for i:=1 to n do
a[i]:=<отсортированные по ключу данные>;
Proot:=ProotBLns(1,n);
```

С помощью процедуры MakeProot, WrtS1 и ProotBLns можно легко построить идеально сбалансированное дерево для любого списка данных.

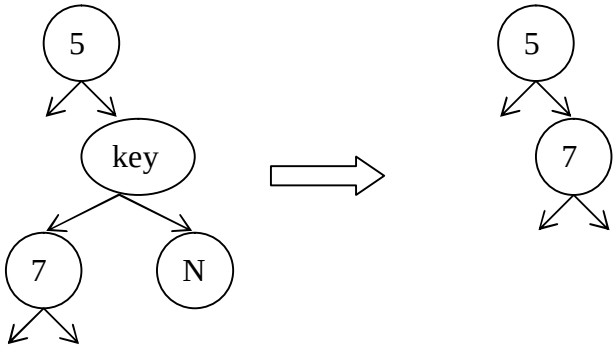
**Удаление узла с заданным ключом из дерева поиска,** сохраняя его свойства.

При решении этой задачи следует рассматривать три разных ситуации:

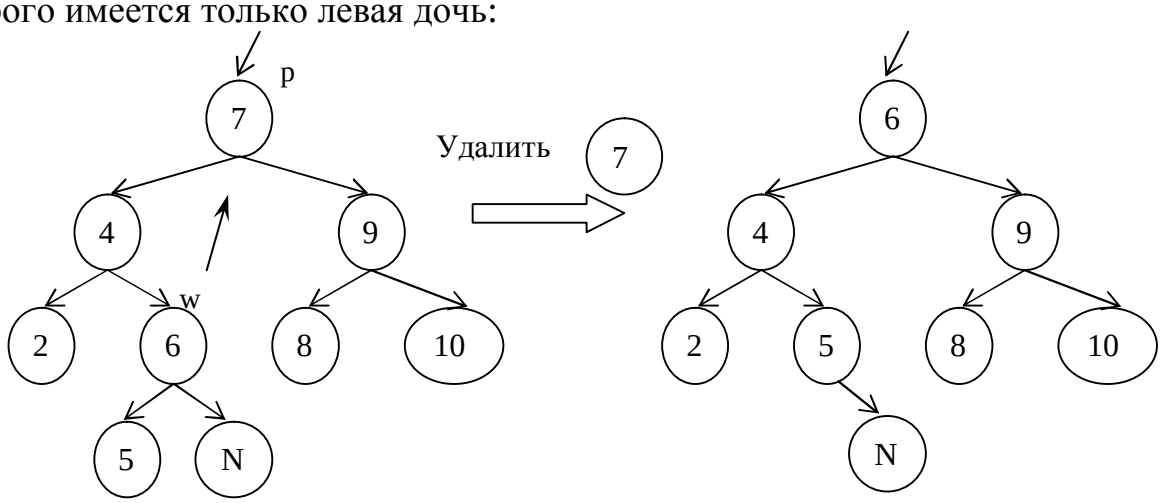
1. Удаление листа с ключом key:




2. Удаление узла, имеющего одну дочь:




3. Удаление узла, имеющего двух дочерей, значительно сложнее. Если *p* — исключаемый узел, то его следует заменить узлом *w*, который содержит наибольший ключ в левом поддереве p (либо наименьший ключ в правом поддереве). Такой узел *w* является либо листом, либо самым правым узлом поддерева p, у которого имеется только левая дочь:




```
Procedure Udtree(Proot:Ttree; Key:Tkey);
    v,w,q:Ttree;
```

```
begin
        p:=Proot;
    While (p<>Nil) and (p^.Inf.key<>key) do
        begin                                          // Поиск
          q:=p;                                        // удаляемого узла
          if P^.Inf.key>key then P:=P^.A1
                                 else P:=P^.A2;
        end;
    if p=Nil then <нет узла key>
      else//исключение найденного узла p:
    begin
    if (P^.A1=Nil) and (P^.A2=Nil) then                    // P лист
    if q^.A1=P then q^.A1:=Nil                        // случай 1
               else q^.A2:=Nil
                                 else

    if P^.A1=Nil then
      if q^.A1=P then q^.A1:=P^.A2                     // P имеет
                 else q^.A2:=P^.A2                     // 1 дочь
                 else                                  // случай 2
    if P^.A2=Nil then
      if q^.A1=P then q^.A1:=P^.A1
                 else q^.A2:=P^.A1
                 else

        begin
        v:=p; w:=p.A1;
        While w^.A2<>Nil do
          begin v=w; w:=w^.A2 end;                     // P имеет
                 v^.A2:=w^.A1;                         //2 дочери
                 w^.A1:=p^.A1;                         //случай 3
                 w^.A2:=p^.A2;
          if q.A1=P then q.A1:=w
                    else q.A2:=w;
        end;
        Dispose(p);
end;
```

### 8.5. Порядок написания программы

Задание: В качестве примера рассмотрим проект, который создает дерево, отображает его в компонентах TTreeView и Memo и удаляет дерево. Панель диалога будет иметь вид, представленный на рис. 8.3.


Рис. 8.3

Текст программы приведен ниже.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, ComCtrls, StdCtrls, Buttons, Grids;
type
  TForm1 = class(TForm)
    StringGrid1: TStringGrid;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    BitBtn3: TBitBtn;
    TreeView1: TTreeView;
    Memo1: TMemo;
    BitBtn5: TBitBtn;
    BitBtn4: TBitBtn;
    Label1: TLabel;
    Edit1: TEdit;
    BitBtn6: TBitBtn;
    procedure BitBtn1Click(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
    procedure BitBtn5Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure BitBtn4Click(Sender: TObject);
    procedure BitBtn6Click(Sender: TObject);
  private
```

```
    { Private declarations }
  public
    { Public declarations }
  end;
Type
 TInf=record
       s   : string[80];     // Ф.И.О.
       key : longword;   // Номер паспорта
      end;

 Ttree=^tree;
 Tree=Record
    Inf:TInf;
    A1:Ttree;
    A2:Ttree;
      end;

var
  Form1: TForm1;
  Proot:Ttree;
  n : integer;
implementation
{$R *.dfm}

Procedure MakeList(var p:Ttree;Inf:Tinf);  // Создание нового листа дерева
          begin
                New(p);
                p^.Inf:=Inf;
                p^.A1:=Nil;
                p^.A2:=Nil;
end;

Procedure Dobtree(Proot:Ttree;Inf:TInf);   // Добавление листа в дерево
   var P,q:Ttree;bl:Boolean;
  begin
  p:=Proot;
        While p<>Nil do  begin
         q:=p; bl:=Inf.key<p^.Inf.key; // q – указатель на предыдущий элемент
          if bl then  p:=p^.A1
                else  p:=p^.A2;
                              end;
        MakeList(p,Inf);
          if bl then q^.A1:=p else q^.A2:=p;
end;
```

```
Procedure ViewTree(p:Ttree; var kl : Integer);
 begin
 if  p <> Nil then begin
   if kl=-1 then
     Form1.TreeView1.Items.AddFirst(Nil, p^.Inf.s+' '+IntToStr(p^.Inf.key))
          else
     Form1.TreeView1.Items.AddChildFirst(Form1.TreeView1.Items[kl],
p^.Inf.s+' '+IntToStr(p^.Inf.key));
        Inc(kl);
      ViewTree(p^.A1,kl);
      ViewTree(p^.A2,kl);
          Dec(kl);
                end;
end;

 Procedure WrtTree(var p:Ttree);    // Прямой обход
  begin
        if p<>Nil then begin
        Form1.Memo1.Lines.Add(p^.Inf.s);
        WrtTree(p^.A1);
        WrtTree(p^.A2);
                    end;
  end;

 Procedure DeleteTree(var p:Ttree);  // Удаление дерева из памяти
Begin
   If p=nil then Exit;
     DeleteTree(p^.A1);
     DeleteTree(p^.A2);
        Dispose(p);
        p:=Nil;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 n:=9;      Edit1.Text:=IntToStr(n);
Memo1.Clear;
  With StringGrid1 do begin
 Cells[0,0]:='Ф.И.О.';
 Cells[1,0]:='Номер';
   Cells[0,1]:='Иванов А.А.';          Cells[1,1]:='100005';
   Cells[0,2]:='Петров С.И.';          Cells[1,2]:='100002';
   Cells[0,3]:='Сидоров К.М.';                Cells[1,3]:='100004';
   Cells[0,4]:='Васильев М.К.';       Cells[1,4]:='100001';
```

```
   Cells[0,5]:='Смирнов В.К.';              Cells[1,5]:='100007';
   Cells[0,6]:='Мишин Т.В.';          Cells[1,6]:='100006';
   Cells[0,7]:='Долин А.К.';          Cells[1,7]:='100008';
   Cells[0,8]:='Катаев А.М.';         Cells[1,8]:='100000';
   Cells[0,9]:='Рубан В.В.';          Cells[1,9]:='100009';
                    end;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);  // Ввод данных в дерево
 var i : integer;
     A : array[1..10] of  TInf;
begin
    For i:=1 to n do
       with  A[i] do begin
          s:=StringGrid1.Cells[0,i];   // Ф.И.О.
          key:=StrToInt(StringGrid1.Cells[1,i]); // Номер паспорта
                       end;
                 MakeList(proot,a[1]);
   For i:=2 to n do Dobtree(proot,a[i]);
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
 var kl : integer;
begin
 kl:=-1;
  TreeView1.Items.Clear;
    ViewTree(Proot,kl);
   TreeView1.FullExpand;
end;

procedure TForm1.BitBtn5Click(Sender: TObject);
begin
Memo1.Clear;
  Wrttree(proot);
end;

procedure TForm1.BitBtn4Click(Sender: TObject);
begin
 n:=StrToInt(Edit1.Text);
  StringGrid1.RowCount:=N+1;
end;

procedure TForm1.BitBtn6Click(Sender: TObject);
begin
```

```
 DeleteTree(proot);
   Memo1.Clear;
   TreeView1.Items.Clear;
end;
end.
```

### 8.6. Индивидуальные задания

Разработать проект для работы с деревом поиска, содержащий следующие обработчики, которые должны:

— ввести информацию из компонента StringGrid в массив. Каждый элемент массива должен содержать строку текста и целочисленный ключ (например Ф.И.О. и номер паспорта);

— внести информацию из массива в дерево поиска;

— сбалансировать дерево поиска;

— добавить в дерево поиска новую запись;

— по заданному ключу найти информацию в дереве поиска и отобразить ее;

— удалить из дерева поиска информацию с заданным ключом;

— распечатать информацию прямым, обратным обходом и в порядке возрастания ключа;

— решить одну из следующих задач:

1. Поменять местами информацию, содержащую максимальный и минимальный ключи.
2. Подсчитать число листьев в дереве. (Лист — это узел, из которого нет ссылок на другие узлы дерева).
3. Удалить из дерева ветвь с вершиной, имеющей заданный ключ.
4. Определить максимальную глубину дерева, т.е. число узлов в самом длинном пути от корня дерева до листьев.
5. Определить число узлов на каждом уровне дерева.
6. Удалить из левой ветви дерева узел с максимальным значением ключа и все связанные с ним узлы.
7. Определить количество символов во всех строках дерева.
8. Определить число листьев на каждом уровне дерева.
9. Определить число узлов в дереве, в которых есть указатель только на один элемент дерева.
10. Определить число узлов в дереве, у которых есть две дочери.
11. Определить количество записей в дереве, начинающихся с определенной буквы (например «а»).
12. Найти среднее значение всех ключей дерева и найти строку, имеющую ближайший к этому значению ключ.
13. Найти запись с ключом, ближайшим к среднему значению между максимальным и минимальным значениями ключей.
14. Определить количество записей в левой ветви дерева.
15. Определить количество записей в правой ветви дерева.

# ТЕМА 9. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

*Цель лабораторной работы:* **ознакомиться с понятиями объекта и класса в Delphi. Написать программу, использующую классы-предки и классы-потомки.**

## 9.1. Понятие объекта и класса

Ключевым понятием ***объектно-ориентированного программирования (ООП)*** является ***объект***, который представляет собой особый структурированный тип переменных. Подобно обыкновенной записи типа ***Record*** переменная типа **объект** под одним именем объединяет как данные различных типов (называемые, как и у записей, ***полями*** объекта), так и процедуры и функции обработки этих данных (называемые ***методами*** объекта).

В Delphi для описания объектов (вместо Object в Паскале) введен специальный тип — ***Class***. Важным отличием типа ***Class*** от ***Object*** является то, что переменные типа ***Class*** представляют собой указатели на объекты. Объявление объектов и работа с ними осуществляется следующим образом:

```
type
 TVect=class(TObject)
    Nazv:string[30];
    X,Y:extended;
    Procedure Povorot;
    Function ScalarProizvedenie(V2:TVect):extended
  End;

var Vec1,Vec2:TVect;

. . .

Vec1.X:=3; Vec1.Y:=4; Vec2.X:=0; Vec2.Y:=5;
Vec1.Povorot;
Vec1.ScalarProizvedenie(Vec2);
```

Как видно, обращение к полям и методам объекта подобно обращению к полям записи. Однако метод является подпрограммой; в описании объекта указывается лишь его заголовок, а его тело должно быть описано ниже:

```
Function TVect.ScalarProizvedenie(V2:TVect):extended;
  Begin
   Result:=X*V2.X+Y*V2.Y
  End;
```

В теле метода при обращении к полям и методам самого вызывающего их объекта имя объекта опускается, а при обращении к полям и методам других

объектов — указывается явно. (Для явной записи указателя на вызывающий объект существует слово ***Self*** ).

В каждом экземпляре объекта содержатся все его поля, но подпрограммы-методы класса хранятся в памяти в одной-единственной копии. Присваивание значений переменным типа ***Class***:

```
Vec2:=Vec1;
```

приводит к копированию только указателя, а не самого объекта, а переменные Vec1 и Vec2 будут указывать на один и тот же объект.

### 9.2. Создание и уничтожение объектов

Как всякая динамическая переменная, объект перед началом работы с ним должен быть создан. Нужно выделить под него динамическую область памяти и ***инициализировать*** (т.е. подготовить к работе) созданный объект. Для этого вDelphi служит ***конструктор Create*** :

*Имя-переменной-типа-класс*> := <*тип-класса*> . Create ;

Например Vec:=TVec.Create;

После окончания работы с объектом выделенную под него память необходимо освободить (***деструктор Destroy)***. Вместо деструктора удобнее использовать метод ***Free***, который перед освобождением памяти проверяет, не была ли она уже освобождена:

<*Имя-переменной-типа-класс*> . Free ;

Например Vec:=TVec.Free;

### 9.3. Свойство наследственности

Свойство наследственности заключается в том, что любой класс может быть порожден от другого класса. В приведенной ниже программе класс TFace порожден от класса TKrug. Поэтому принято называть TKrug — «класс-родитель», а TFace — «класс-потомок». Порожденный класс ( TFace ) автоматически наследует поля и методы своего родителя и обогащает их новыми. Таким образом, принцип наследования позволяет эффективно использовать уже наработанный задел программ и на его основе создать классы для решения все более сложных задач.

Прародителем всех классов в Delphi является класс ***TObject***. Все компоненты Delphi, которыми вы оперировали (***TForm, TEdit, TCanvas*** и др.), представляют собой созданные разработчиками классы, имеющие довольно большое генеалогическое дерево.

Переменной родительского типа ( TKrug ) можно присвоить значение переменной типа-потомка ( TFace ). Обратное же присваивание запрещено, т.к. при этом некоторые поля или методы могли бы оказаться несуществующими.

Однако, если переменная типа TKrug указывает на объект, фактически соответствующий TFace, то при необходимости ее можно преобразовать к TFace с помощью оператора ***as*** :

```
Var K:TKrug; F1,F2:TFace;
. . .
K:=F1;
. . .
F2:=K as TFace;
```

### 9.4. Понятие полиморфизма

Свойство ***полиморфизма*** позволяет использовать одинаковое название метода для решения сходных, но несколько отличающихся в разных родственных классах задач. Например, в программе (см. ниже) метод Show рисует изображение на экране; но это изображение в классе-родителе и классе-потомке различное. Обеспечивается это тем, что в классе-потомке метод переписывается по новому алгоритму, т.е. ***перекрывается***. В результате в объекте-родителе и объекте-потомке будут действовать два одноименных метода с разными алгоритмами. Это и называется полиморфизмом объектов.

Метод потомка может вызывать методы предков. Для вызова методов родителя служит слово ***Inherited*** :

```
Inherited Show;
```

В случае, если у потомка неизмененный родительский метод должен вызывать измененный метод потомка, вызываемый метод должен быть описан как ***виртуальный***: в родительском классе с помощью слова ***virtual*** , а в классах-потомках с помощью слова ***override*** . Типы параметров (и, если есть, результата) в перекрываемых виртуальных методах должны совпадать. Указатели на эти методы будут помещены в ***таблицы виртуальных методов***. Каждому классу соответствует такая таблица, а в каждый экземпляр объекта ***конструктор*** при ***инициализации*** автоматически записывает «невидимый» для программиста указатель на таблицу своего класса. Такой механизм значительно расширяет возможности полиморфизма.

Конструкторы и деструкторы, подобно другим методам, могут быть переопределены; однако рекомендуется, чтобы при этом они всегда обращались к одноименным родительским методам.

### 9.5. Пример написания программы

РАССМОТРИМ ЗАДАЧУ, ВКЛЮЧАЮЩУЮ СЛЕДУЮЩИЕ ПОДЗАДАЧИ:

А) ОПИСАТЬ КЛАСС — КРУГ НА ЭКРАНЕ (ЗАДАННОГО ПОЛОЖЕНИЯ И ЦВЕТА), ВКЛЮЧАЮЩИЙ МЕТОД ДЛЯ ПЕРЕМЕЩЕНИЯ ПО ЭКРАНУ;

Б) НА ЕГО ОСНОВЕ ОПИСАТЬ КЛАСС — «РОЖИЦУ», ВКЛЮЧАЮЩИЙ ДОПОДНИТЕЛЬНО МЕТОД «ПОДМИГИВАНИЯ»;

С) НАПИСАТЬ ПРОГРАММУ, ДЕМОНСТРИРУЮЩУЮ РАБОТУ С ОБЪЕКТАМИ ТАКИХ КЛАССОВ.

КРУГ (КЛАСС TKRUG) ОПИСЫВАЕТСЯ В МОДУЛЕ UKRUG. ЧТОБЫ НЕ ПРИВЯЗЫВАТЬ КЛАСС TKRUG К КОНКРЕТНОЙ ФОРМЕ, В ЧИСЛО ЕГО ПОЛЕЙ ВКЛЮЧЕНО ПОЛЕ CANVAS — ***УКАЗАТЕЛЬ НА ОБЪЕКТ КЛАССА TCANVAS,*** ОПИСЫВАЮЩИЙ ОБЛАСТЬ РИСОВАНИЯ. В КАЧЕСТВЕ ЗНАЧЕНИЯ ПОЛЯ CANVAS В ГОЛОВНОМ МО-

ДУЛЕ UNIT1 ПОДСТАВЛЯЕТСЯ FORM1.CANVAS (С ПОМОЩЬЮ КОНСТРУКТОРА CREATE).

ПОЛЯ X, Y, R, COLOR ХРАНЯТ СООТВЕТСТВЕННО КООРДИНАТЫ ЦЕНТРА КРУГА, ЕГО РАДИУС И ЦВЕТ. КОНСТРУКТОР TKRUG.CREATE, ПЕРЕОПРЕДЕЛЯЮЩИЙ TOBJECT.CREATE, ОБЕСПЕЧИВАЕТ НЕ ТОЛЬКО СОЗДАНИЕ ОБЪЕКТА, НО И ПРИСВОЕНИЕ ЕГО ПОЛЯМ НАЧАЛЬНЫХ ЗНАЧЕНИЙ. МЕТОД SHOW РИСУЕТ КРУГ НА ЭКРАНЕ, А МЕТОД MOVE ПЕРЕМЕЩАЕТ ЕГО НА ЗАДАННОЕ РАССТОЯНИЕ (ПРИ ЭТОМ ПРЕЖНЕЕ ИЗОБРАЖЕНИЕ ЗАТИРАЕТСЯ ЦВЕТОМ ФОНА, КОТОРЫЙ СЧИТАЕТСЯ РАВНЫМ СТАНДАРТНОМУ ЦВЕТУ ФОРМЫ ***CLBTNFACE*** ).

МЕТОДЫ TKRUG ИСПОЛЬЗУЮТ ГРАФИЧЕСКИЕ ПРОЦЕДУРЫ (МОДУЛЬ GRAPHICS ).

```
UNIT UKRUG;
INTERFACE

USES GRAPHICS;

TYPE TKRUG=CLASS(TOBJECT)
 X,Y,R:INTEGER;
 COLOR:TCOLOR;
 CANVAS:TCANVAS;
 CONSTRUCTOR
CREATE(X0,Y0,R0:INTEGER;COLOR0:TCOLOR;CANVAS0:TCANVAS);
 PROCEDURE MOVE(DELTAX,DELTAY:INTEGER);
 PROCEDURE SHOW; VIRTUAL;
END;

IMPLEMENTATION

CONSTRUCTOR TKRUG.CREATE;
BEGIN
 INHERITED CREATE;
 X:=X0;
 Y:=Y0;
 R:=R0;
 COLOR:=COLOR0;
 CANVAS:=CANVAS0;
END;

PROCEDURE TKRUG.SHOW;
BEGIN
 CANVAS.PEN.COLOR:=CLBLACK;
 CANVAS.BRUSH.COLOR:=COLOR;
 CANVAS.ELLIPSE(X-R,Y-R,X+R,Y+R)
END;

PROCEDURE TKRUG.MOVE;
BEGIN
 CANVAS.BRUSH.COLOR:=CLBTNFACE;  // СТИРАНИЕ ПРЕДЫДУЩЕГО ИЗОБРА-
ЖЕНИЯ
```

```
 CANVAS.PEN.COLOR:=CLBTNFACE;
 CANVAS.ELLIPSE(X-R, Y-R, X+R, Y+R);

 X:=X+DELTAX;
 Y:=Y+DELTAY;
 SHOW
END;
END.
```

«РОЖИЦА» (КЛАСС TFACE) ОПИСЫВАЕТСЯ В МОДУЛЕ UFACE. ПОЛЕ EYECOLOR ОПИСЫВАЕТ ЦВЕТ ГЛАЗ, HEIGHTEYE1 И HEIGHTEYE2 — ВЫСОТУ (ПОЛОВИННУЮ) КАЖДОГО ИЗ ГЛАЗ. КОНСТРУКТОР CREATE СОЗДАЕТ «РОЖИЦУ» СТАНДАРТНОГО РАЗМЕРА И ЦВЕТА. МЕТОДЫ PODMIGNUT И NORMEYES МЕНЯЮТ ВИД «РОЖИЦЫ». МЕТОД MOVE УНАСЛЕДОВАН БЕЗ ИЗМЕНЕНИЙ, НО ОН ОБРАЩАЕТСЯ К ИЗМЕНЕННОМУ МЕТОДУ SHOW.

```
UNIT UFACE;
INTERFACE

USES GRAPHICS,
    UKRUG;

TYPE TFACE=CLASS(TKRUG)
 EYECOLOR:TCOLOR;
 HEIGHTEYE1,HEIGHTEYE2:BYTE;
 CONSTRUCTOR CREATE(X0,Y0:INTEGER;CANVAS0:TCANVAS);
 PROCEDURE SHOW; OVERRIDE;
 PROCEDURE PODMIGNUT;
 PROCEDURE NORMEYES;
END;

IMPLEMENTATION

CONSTRUCTOR TFACE.CREATE;
BEGIN
 INHERITED CREATE(X0, Y0, 25, CLYELLOW, CANVAS0);
 NORMEYES
END;

PROCEDURE TFACE.NORMEYES;
BEGIN
 EYECOLOR:=CLAQUA;
 HEIGHTEYE1:=3;
 HEIGHTEYE2:=3;
 SHOW
END;

PROCEDURE TFACE.SHOW;
VAR
 XI,YI,H,I:INTEGER;
BEGIN
```

```
 INHERITED SHOW;
 CANVAS.BRUSH.COLOR:=EYECOLOR;
 FOR I:=1 TO 2 DO BEGIN
  CASE I OF
1: BEGIN
    XI:=X-10;
    H:=HEIGHTEYE1
   END;
2: BEGIN
    XI:=X+10;
    H:=HEIGHTEYE2
   END
  END;    // CASE I OF
  YI:=Y-10;
  CANVAS.ELLIPSE(XI-5, YI-H, XI+5, YI+H)
 END;          // FOR I
 CANVAS.ARC(X-20, Y-25, X+20, Y+15,
           X-25, Y+10, X+25, Y+10);
END;

PROCEDURE TFACE.PODMIGNUT;
BEGIN
 HEIGHTEYE1:=1;
 EYECOLOR:=CLBLUE;
 SHOW
END;
END.
```

ДЛЯ РАБОТЫ С ОБЪЕКТАМИ СЛУЖИТ МОДУЛЬ UNIT1, ЧЬЯ ФОРМА (СМ. РИС. 9.1) СОДЕРЖИТ ВСЕ НЕОБХОДИМЫЕ ПОЛЯ И КНОПКИ. В НЕМ СОЗДАЮТСЯ ОДИН ОБЪЕКТ ТИПА TKRUG И ДВА ОБЪЕКТА ТИПА TFACE (ПЕРЕМЕННАЯ FACEACTIV МОЖЕТ ПОПЕРЕМЕННО УКАЗЫВАТЬ НА ЭТИ 2 ОБЪЕКТА)

ЧТОБЫ ОБЪЕКТЫ ПРОРИСОВЫВАЛИСЬ СРАЗУ ЖЕ И НЕ ПРОПАДАЛИ С ЭКРАНА ПРИ СВОРАЧИВАНИИ И РАЗВОРАЧИВАНИИ ОКОН, ВВЕДЕНА ПРОЦЕДУРА FORMPAINT, ОБРАБАТЫВАЮЩАЯ СОБЫТИЕ ONPAINT (НЕОБХОДИМОСТЬ ПЕРЕРИСОВКИ ЭКРАНА) ДЛЯ FORM1. УНИЧТОЖАЮТСЯ ОБЪЕКТЫ В ПРОЦЕДУРЕ FORMCLOSE (НАПОМИНАЕМ, ЧТО ДЛЯ СОЗДАНИЯ ПРОЦЕДУР FORMPAINT И FORMCLOSE НУЖНО ВЫБРАТЬ В OBJECT INSPECTOR СВОЙСТВА ФОРМЫ (FORM1) И НА ВКЛАДКЕ EVENTS ЩЕЛКНУТЬ ДВАЖДЫ НА СТРОКЕ СОБЫТИЯ ONPAINT И ONCLOSE СООТВЕТСТВЕННО).

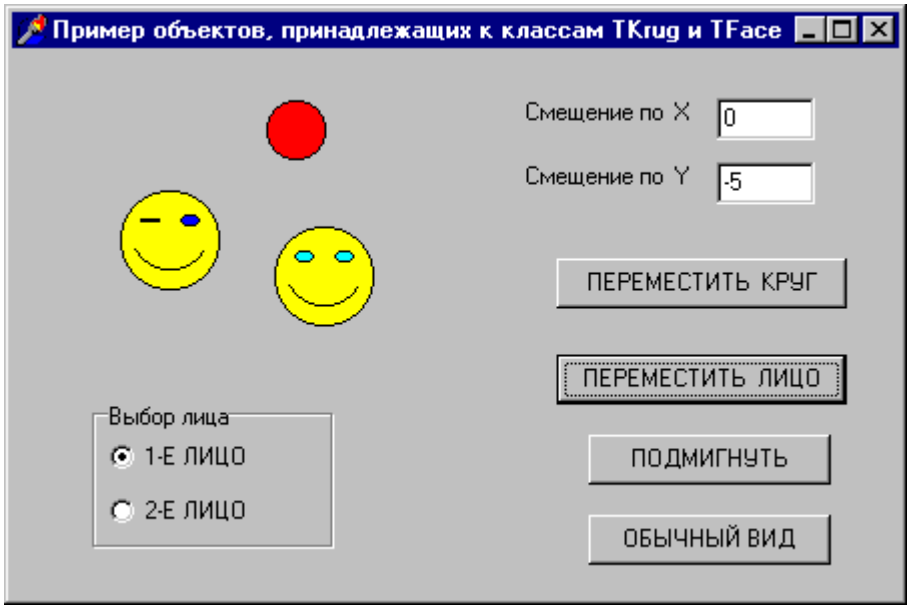



Рис. 9.1

Текст программы:

```
unit Unit1;
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls,
  UKrug, UFace;

type
  TForm1 = class(TForm)
    Button2: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Button3: TButton;
    Button4: TButton;
    RadioGroup1: TRadioGroup;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);
```

```
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  Krug1:TKrug;
  Face1,Face2,FaceActiv:TFace;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
 Krug1:=TKrug.Create(180, 40, 15, clRed, Form1.Canvas);
 Face1:=TFace.Create(40, 40, Form1.Canvas);
 Face2:=TFace.Create(110, 40, Form1.Canvas);
 FaceActiv:=Face1
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
 Face1.Show;
 Face2.Show;
 Krug1.Show
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 Krug1.Move(StrToInt(Edit1.Text), StrToInt(Edit2.Text))
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
 FaceActiv.Move(StrToInt(Edit1.Text), StrToInt(Edit2.Text))
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
 FaceActiv.Podmignut
```

```
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
 FaceActiv.NormEyes
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
 if RadioGroup1.ItemIndex=0
  then FaceActiv:=Face1
  else FaceActiv:=Face2
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 Krug1.Free;
 Face1.Free;
 Face2.Free
end;
end.
```

Таким образом, ООП позволяет создавать единые методы для различных, хотя и родственных между собой, объектов. Оно также позволяет постепенно усложнять программу, не изменяя уже созданные классы, а наращивая их путем введения потомков.

### 9.6. Варианты заданий

Описать класс-родитель и класс-потомок, имеющие методы, указанные в соответствующем варианте задания (потомок наследует или переопределяет методы родителя и приобретает новые). Предусмотреть необходимое количество кнопок для демонстрации каждого из методов объектов.

1. Родитель — круг (перемещение).

Потомок — колесо (вращение).

2. Родитель — прямоугольник (перемещение).

Потомок — повозка (прямоугольник на 2 колесах) (перемещение вперед и назад с поворотом колес).

3. Родитель — отрезок (перемещение в произвольную сторону, поворот вокруг начального конца).

Потомок — ракета (включение/выключение двигателя с появлением/ исчезновением пламени из сопла, смещение вперед).

4. Родитель — эллипс (перемещение).

Потомок — рожица (открывание и закрывание рта, открывание и закрывание глаз).

5. Родитель — неподвижный человечек без головного убора с подвижными, сгибающимися в локтях руками (шевеление руками).

Потомок — «солдатик» (ввести поле — наличие пилотки) (отдание чести).

6. Родитель — трапеция с горизонтальными основаниями (перемещение).

Потомок — кораблик (смещение вперед, поднятие/спуск флага).

7. Родитель — квадрат (перемещение).

Потомок — квадратная рожица (поворот глаз направо и налево).

8. Родитель — кружок радиусом 3 пикселя (перемещение).

Потомок — выходящая из кружка стрелка (поворот, сдвиг вперед).

9. Родитель — трапеция (перемещение).

Потомок — автомобиль (движение вперед и назад, открывание дверцы).

10. Родитель — неподвижный человечек с подвижными руками (шевеление руками).

Потомок — сигнальщик (подъем/опускание флажка заданного цвета, выбираемого из нескольких цветов).

11. Родитель — грузовик (смещение вперед/назад).

Потомок — самосвал (ввести поле — наличие груза) (загрузка, откидывание/поднятие кузова).

12. Родитель — домик с дверцей (открывание/закрывание дверцы).

Потомок — домик с дверцей и 2 окнами (открывание/закрывание каждого из окон).

13. Родитель — прямоугольник на 2 колесах (смещение влево/вправо).

Потомок — автофургон (разворот в обратном направлении).

14. Родитель — самолетик (вид сбоку) (перемещение).

Потомок — самолетик с шасси (выпуск/убирание шасси.)

15. Родитель — светофор (смена предыдущего сигнала на очередной (в последовательности: красный -> красный+желтый -> зеленый -> желтый -> красный -> красный+желтый ->...) ).

Потомок — светофор со стрелкой поворота (активизация ее с удлинением последовательности (желтый без стрелки -> красный без стрелки -> красный со стрелкой -> красный+желтый без стрелки -> ...); отключение стрелки поворота с возвратом к прежней последовательности).

# ТЕМА 10. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ХЕШИРОВАНИЯ

*Цель лабораторной работы:* **изучить способы программирования алгоритмов с использованием хеширования.**

## 10.1. Понятие хеширования

Имеется глобальная проблема: поиск данных в списке.

а) Если данные расположены беспорядочно в массиве (линейном списке, файле), то осуществляют **линейный поиск** его эффективность *0(n/2).*

б) Если имеется упорядоченный массив (двоичное дерево), то возможен **двоичный поиск** с эффективностью $0(\log_2 n)$;

ОДНАКО ПРИ РАБОТЕ С ДВОИЧНЫМ ДЕРЕВОМ И УПОРЯДОЧЕННЫМ МАССИВОМ ЗАТРУДНЕНЫ ОПЕРАЦИИ ВСТАВКИ И УДАЛЕНИЯ ЭЛЕМЕНТОВ, ДЕРЕВО РАЗБАЛАНСИРУЕТСЯ, И ТРЕБУЕТСЯ БАЛАНСИРОВКА. ЧТО МОЖНО ПРИДУМАТЬ БОЛЕЕ ЭФФЕКТИВНОЕ?

Придумали алгоритм **хеширования** (*hashing* — перемешивание), при котором ключи данных записываются в хеш-таблицу. При помощи некой функции $i=h(key)$ алгоритм хеширования определяет положение искомого элемента в таблице по значению его ключа.

В простейшем случае алгоритм хэширования реализуется следующим образом.

Допустим, имеется набор $m$ записей с ключами в диапазоне $0 \le key \le K, \quad m < K$.

Создадим массив:

```
H : array[0..K] of <тип записей>;
```

Вначале его очистим **`H[i]:=0`** и наши $m$ записей помещаем в этот массив в соответствии со значением ключа $i=h(key)=key$. Затем, используя операторы

```
zp:=H[key]; // Извлекаем из массива
H[key]:=zp; // Записываем в массив
```

Хорошо, если ключи располагаются от 1 до 100. А если это номер страховой карточки с 9 значащими цифрами? Потребуется массив из $10^9$ ячеек!!! Для одного города с миллионным населением этот массив будет использоваться только на 0,1%.

Чтобы все-таки использовать этот изящный способ, и придумывают различные **схемы хеширования**, т.е. установление такой функциональной связи между значением ключа $key$ и местоположением $i$ записи в некотором массиве, при котором размер таблицы был бы пропорционален количеству записей, а не величине ключа.

Пусть $M$ — предельный размер массива записей $m < M$. Позиции в массиве будем нумеровать начиная с нуля. Задача состоит в том, чтобы подобрать функцию $i=h(key)$, которая по возможности равномерно отображает значения ключа $key$ на интервал $0 \le i \le M-1$. Чаще всего, если нет информации о вероятности распределения значений ключей по записям, в предположении равновероятности используют $i=h(key)=key \bmod M$.

Функция $h(key)$ называется **функцией расстановки,** или **хеш-функцией**. Ввиду того, что число возможных значений ключа $K$ обычно значительно превосходит возможное количество записей $K>>M$, любая функция расстановки может для нескольких значений ключа давать одинаковое значение позиции $i$ в таблице. В этом случае $i=h(key)$ только определяет позицию, начиная с которой нужно искать запись с ключом $key$. Поэтому схема хеширования должна включать **алгоритм разрешения конфликтов**, определяющий порядок действий, если позиция $i=h(key)$ оказывается занятой записью с другим ключом.

Имеется множество схем хеширования, различающихся как выбором удачной функции *h(key),* так и алгоритма разрешения конфликтов. Эффективность решения реальной практической задачи будет существенно зависеть от выбираемой стратегии.

**Алгоритм размещения** записей в хеш-таблицу содержит следующие действия: сначала ключ *key* очередной записи отображается на позицию *i=h(key)* таблицы. Если позиция свободна, то в нее размещается элемент с ключом *key*, если занята, то отрабатывается алгоритм разрешения конфликтов, который находит место для размещения данного элемента.

**Алгоритм поиска** сначала находит по ключу позицию *i* в таблице, и если ключ не совпадает, то последующий поиск осуществляется в соответствии с алгоритмом разрешения конфликтов, начиная с позиции *i*.

Часто ключами таблиц являются не числа, а последовательность букв (строки). При выборе функции *i=H(key)* важно, чтобы ее значения вычислялись, как можно проще и быстрее. Поэтому символьные ключи обычно заменяют целыми числами из некоторого диапазона. Например, если *key* — слово, то суммой кодов первых 2-3-х букв, если фраза (Ф.И.О.), то сумма кодов первых букв. При написании алгоритма разрешения конфликтов поиск ведут уже по полному слову.

## 10.2. Хеш-таблица на основе массива связанных списков

ОДИН ИЗ САМЫХ ИЗЯЩНЫХ СПОСОБОВ РАЗРЕШЕНИЯ КОНФЛИКТОВ СОСТОИТ В ТОМ, ЧТОБЫ СОХРАНЯТЬ ЗАПИСИ, ОТОБРАЖАЕМЫЕ НА ОДНУ ПОЗИЦИЮ В СВЯЗАННЫЕ СПИСКИ (ОБЫЧНО СТЕКИ). ДЛЯ ЭТОГО ВВОДИТСЯ МАССИВ ИЗ *M* УКАЗАТЕЛЕЙ НА СВЯЗАННЫЕ СПИСКИ.

*M* — простое число, немного большее, чем предполагаемое количество записей.

Хэш-функция выбирается в виде *i:=Key mod M.*

РАБОТУ С ТАБЛИЦЕЙ МОЖНО ОРГАНИЗОВАТЬ ПОСРЕДСТВОМ КЛАССА TH, ПРЕДСТАВЛЕННОГО НИЖЕ:

```
Type
      TInf=Record
        In:Tin;       // Поле информации
        Key:Word; // Поле ключа
      end;
        Tsel=^sel
         sel=Record
         Inf:TInf;
         A:Tsel;
        end;

TH=class(Tobject);
```

```
  protected iM:word; p:Tsel;    // Защищенная область полей
       H:^array[0..1] of Tsel;     // Указатель на массив указателей
  public
      Constructor create(M:word);          // Создание таблицы
      Destructor Free(M);                   // Освобождение таблицы
      Procedure Add(Inf:TInf);              // Добавление элемента
      Procedure Del(key:Tkey);              // Удаление элемента
      Procedure Read(key:Tkey;var Inf); // Чтение элемента без удаления
  end;

       Constructor TH.create;
         begin
          Inherited create;       // Вызов конструктора
                                  // родительского класса Tobject
              GetMem(H,4*M);          // Создание массива из M указателей
              for i:=0 to M-1 do H[i]:=Nil;    // и его очистка
         end;

       Procedure TH.Add;
         begin
              i:=Inf.key mod M;       // Хэш-функция
              New(P);P^.Inf:=Inf;    // Добавление элемента в стека
              P^.A:=H[i];             // с вершиной H[i]
              H[i]:=P;
         end;

Procedure TH.Del;              // Поиск и удаление записи
 Var p1:Tsel;
  begin
       i:=key mod M;
       p:=H[i];
       if p=Nil then exit else     // Нет записи
       if P^.Inf.key=key then    // Ключ в первой записи
                  begin H[i]:=P.^A;  Dispose(p)  end
       else
  begin p1:=p^.A                 // Переход ко второй записи
       While P1<>Nil do begin
         if P1^.Inf.key=key then
              begin
                   P^.A:=P1^.A;
                   Dispose(P1);
                   Exit
              End;
          P:=P1;P1:=P1^.A
```

```
                        end;
 end;

Procedure TH.Read;   // Поиск и чтение записи без удаления
 Var bl:boolean;         // Сравнение с предыдущей
 begin
        i:=key mod M; p:=H[i];bl:=false;
    if p=Nil then bl:=false;
    else
        Repeat
            bl:=P^.Inf.Key=Key;
            p1:=p; P:=P^.A;
        Until bl or P=Nil;
    If bl then Inf:=P1^.Inf else Inf:=<отсутствует>;
 end;
```

После описания этого класса работа с хэш-таблицей осуществляется следующим образом:

```
Var H1:TH;   // Экземпляр таблицы
..
begin
    ...
        H1.create(Mzap);
        Repeat
          <ввод Inf>;
          H1.Add(Inf);
        Until <пока не закончилась информация>;
        <ввод key>
        H1.read(key,Inf);
        ....
        <ввод key>
        H1.Del(key);
        ....
        H1.free(Mzap);
```

Преимущество этого метода заключается в том, что связанные хэш-таблицы никогда не переполняются, довольно просто осуществляется вставка, удаление и поиск элементов. Недостаток таких таблиц в том, что если данные недостаточно равномерно перемешаны по ключу, то некоторые стеки могут оказаться очень длинными, в то время как большинство других будут пустыми, при этом поиск будет замедляться.

Для избавления от этого недостатка нужно придумать другую функцию хеширования используя информацию о распределении записей по значению

ключа. Например, можно выбрать двухмерный массив указателей, и подбирая простые числа $p$ и $q$, $p * q \gtrsim M$ , добиваться равномерного распределения значений по таблице.

H[i,j], i=Key mod p; j:=(key div p) mod q.

### 10.3. Индивидуальные задания

СОСТАВИТЬ КЛАСС ДЛЯ РАБОТЫ С ХЭШ-ТАБЛИЦЕЙ НА ОСНОВЕ МАССИВА СТЕКОВ. В ВЫШЕПЕРЕЧИСЛЕННЫХ МЕТОДАХ МОДИФИЦИРОВАТЬ ДЕСТРУКТОР ТАК, ЧТОБЫ ПРИ ОБРАЩЕНИИ К НЕМУ ПРОИСХОДИЛА ЗАПИСЬ ВСЕХ ДАННЫХ ИЗ ХЭШ-ТАБЛИЦЫ В ФАЙЛ С ОСВОБОЖДЕНИЕМ ПАМЯТИ. СОЗДАТЬ ОБРАБОТЧИК, ПРИ ОБРАЩЕНИИ К КОТОРОМУ ВСЕ ДАННЫЕ ИЗ ФАЙЛА ЗАПИСЫВАЮТСЯ В ХЭШ-ТАБЛИЦУ. СОЗДАТЬ ПРОГРАММУ ЗАПИСИ ВВОДИМОЙ ИНФОРМАЦИИ В ХЭШ-ТАБЛИЦУ И ПОИСКА ТРЕБУЕМОЙ ЗАПИСИ ПО КЛЮЧУ.

В качестве индивидуального задания использовать задания по теме 6.

# ЛИТЕРАТУРА


1. Программирование алгоритмов в среде Delphi. Лаб. практикум по курсам «Программирование» и «Основы алгоритмизации и программирование» для студентов 1–2 курсов всех специальностей БГУИР. В 2 ч. Ч. 1. / Под общ. ред. А.К. Синицына. — Мн.: БГУИР, 2002.

2. Синицын А.К. Конспект лекций по курсу «Программирование» для студентов 1–2 курсов всех специальностей БГУИР. — Мн.: БГУИР, 2001.

3. Вирт Никлаус. Алгоритмы и структуры данных. — СПб.: «Невский диалект», 2001.

4. Стивенс Род. Delphi. Готовые алгоритмы. — М.: ДМК Пресс, 2001.

5. Морозов А.А. Структуры данных и алгоритмы. Учеб. пособие. В 2 ч. — Мн. БГПУ им. М. Танка, Ч. 1. — 2000, Ч. 2. — 2001.

6. Фаронов В.В. DELPHI 3. Учебный курс. — М.: Нолидж, 1998.

7. Дарахвелидзе П.Г., Марков Е.П. Delphi — среда визуального программирования. — СПб.: BHV–Санкт-Петербург, 1996.

8. Федоров А.Г. Delphi 3.0. для всех. — М.: КомпьютерПресс, 1998.

9. Дарахвелидзе П.Г., Марков Е.П. Delphi 4. — СПб.: BHV–Санкт-Петербург, 1999.

10. Брукшир Дж. Гленн. Введение в компьютерные науки. — М., «Вильямс», СПб., Киев. 2001.

11. Архангельский А.Я. Все о Delphi. — М: Бином. 1999.

12. Фаронов В.В. Delphi 4. Учебный курс. — М., 1999.

Св. план 2002, поз. 28

**Учебное издание**

**Синицын** Анатолий Константинович,
**Колосов** Станислав Васильевич,
**Навроцкий** Анатолий Александрович и др.

# ПРОГРАММИРОВАНИЕ АЛГОРИТМОВ В СРЕДЕ DELPHI

Лабораторный практикум по курсу
«Основы алгоритмизации и программирование»
для студентов 1–2-го курсов всех специальностей БГУИР
дневной и вечерней форм обучения

В 2-х частях

Часть 2

Редактор Е.Н. Батурчик

---

Подписано в печать 27.01.2003. Формат 60x84 1/16.
Бумага офсетная. Печать ризографическая. Гарнитура «Таймс». Усл. печ. л. 4,3.
Уч. изд. л. 3,7. Тираж 500 экз. Заказ 681.

---

Издатель и полиграфическое исполнение:
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия ЛП № 156 от 30.12.2002.
Лицензия ЛП № 509 от 03.08.2001.
220013, МИНСК, П. БРОВКИ, 6