ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра прикладной математики и информатики

МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ЗАДАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ ПО КУРСУ "АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ "

(направление подготовки 6.050103 ”Программная инженерия”)

Донецк 2009

ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра прикладной математики и информатики

МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ЗАДАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ ПО КУРСУ "АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ "

(направление подготовки 6.050103 "Программная инженерия")

Рассмотрено на заседании кафедры
Прикладной математики и информатики
Протокол № 2 от 26.09.09

Утверждено на заседании
учебно- издательского совета ДонНТУ
протокол № 5 от 21.12.09

Донецк 2009

УДК 518.551071

Методические указания и задания к лабораторным работам по курсу "Алгоритмы и структуры данных" (направление подготовки 6.050103 ”Программная инженерия”). Сост. Г.Г.Шалдырван, Н.С.Костюкова. – Донецк, 2009. – 108 с.

Излагаются вопросы, связанные с теоретическими основами структур данных, способами представления данных в оперативной памяти компьютера, методами обработки различных структур данных. Приведены описания алгоритмов и процедур обработки таблиц, списковых структур, деревьев.

В приложениях методических указаний приводятся примеры программ по темам лабораторных работ.

Методические указания предназначены для усвоения теоретических основ и формирования практических навыков по выбору рациональных структур данных и их обработке.

Составители: доц. Г.Г. Шалдырван
доц. Н.С. Костюкова

ЛАБОРАТОРНАЯ РАБОТА 1
ПОСТОЯННЫЕ ТАБЛИЦЫ

Цель работы: ознакомление с различными методами сортировки таблиц и методами поиска в упорядоченных таблицах.

Задание к лабораторной работе

Разработать приложение, содержащее процедуры, реализующие основные операции над таблицами: формирование таблицы, сортировка или поиск (согласно указанному варианту), вывод таблицы. В качестве варианта выбрать номер по журналу

В качестве языка программирования следует использовать ТурбоПаскаль, как наиболее подходящее средство для задач обработки данных. В **Приложении А** приведен пример программы, выполняющей сортировку таблицы методом линейного выбора.

Варианты заданий

Упорядочить таблицу, используя:

1. Метод линейного выбора с подсчетом.
2. Метод шейкер - сортировки.
3. Метод вставки с бинарным включением.
4. Метод двухпутевой вставки.
5. Метод Шелла.
6. Метод "быстрой" сортировки (в качестве разделяющего использовать первый элемент таблицы).
7. Метод "быстрой" сортировки (в качестве разделяющего использовать средний элемент таблицы).
8. Метод простого двухпутевого слияния.
9. Метод естественного слияния.

Упорядочить таблицу указанными в варианте методами и выполнить сравнительный анализ этих методов:

10. Метод вставки с прямым включением и метод двухпутевой вставки.
11. Метод вставки с прямым включением и метод Шелла.
12. Метод "пузырька" и метод шейкер-сортировки.
13. Метод линейного выбора и метод линейного выбора с подсчетом.
14. Метод линейного выбора и метод "быстрой" сортировки с первым разделяющим элементом.
15. Метод вставки с прямым включением и метод “быстрой” сортировки со средним разделяющим элементом.
16. Метод линейного выбора и метод “быстрой” сортировки со средним разделяющим элементом.
17. Метод вставки с прямым включением и метод "быстрой" сортировки с первым разделяющим элементом.

18. Метод вставки с прямым включением и метод вставки с бинарным включением.
19. Метод “пузырька” и метод вставки с бинарным включением.
20. Метод “пузырька” и метод простого двухпутевого слияния.
21. Методы "пузырька", линейного выбора и вставки с прямым включением.
22. Метод "пузырька" и метод вставки с прямым включением.
23. Метод линейного выбора и метод вставки с бинарным включением.
24. Метод "быстрой" сортировки с первым разделяющим элементом и метод “быстрой” сортировки со средним разделяющим элементом.
25. Метод вставки с прямым включением и метод шейкер-сортировки.

Выполнить следующие виды дихотомического поиска:

26. Поиск по совпадению при наличии в таблице нескольких записей с заданным значением ключа (предварительно упорядочить таблицу методом вставки с бинарным включением).
27. Поиск по близости (предварительно упорядочить таблицу методом “пузырька”).
28. Поиск по интервалу (предварительно упорядочить таблицу методом вставки с прямым включением).
29. Поиск записей, ключи которых больше заданного значения (предварительно упорядочить таблицу методом простого двухпутевого слияния).
30. Поиск записей, ключи которых меньше заданного значения (предварительно упорядочить таблицу методом линейного выбора).

Пояснения:

Сравнительный анализ методов сортировки (варианты 10-25) можно осуществить подсчетом числа сравнений, выполненных в процессе сортировки таблицы разными методами.

Требования к выполнению лабораторной работы

Разрабатываемое приложение должно удовлетворять следующим требованиям:

1. Алгоритмы, реализующие операции над таблицами, нужно оформить в виде процедур (функций) с **соответствующими формальными параметрами.**

2. Содержимое контрольной таблицы может считываться из файла, но должна быть предусмотрена также возможность ввода записей таблицы в интерактивном режиме. Таблица должна иметь не менее двух полей, одно из которых - ключевое.

3. На экран выводить:

а) исходную таблицу;

б) **результаты выполнения шагов сортировки (поиска), демонстрирующие конкретный метод** (при этом рекомендуется выполнять вывод данных **различным цветом**);

в) окончательный результат.

4. В контрольной таблице должно быть **20- 25** записей (чтобы выполнилось несколько шагов метода).

5. **Привести примеры “ручной” сортировки (поиска) таблицы заданными методами.**

Контрольные вопросы

1. Определение таблицы, записи, ключа; классификация таблиц.
2. Определение операции сортировки; основные методы сортировки и их характеристики.
3. Определение операции поиска; дихотомический поиск по совпадению, близости, интервалу.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ РАБОТЫ

Определение таблицы

Таблица является наиболее употребительной структурой данных. *Таблица* – это множество записей одинаковой структуры. *Запись* – это логическое объединение разнотипных данных в единую конструкцию. Каждое данное представляется *полем*. *Поле* – минимальная единица, на которую можно ссылаться при обращении к записи.

Поля записи содержат в общем случае алфавитно – цифровую информацию. Одно из полей определяет *ключ* записи. *Ключ* – это поле, значение которого однозначно идентифицирует запись в таблице. *Ключ* используется в операциях упорядочения таблиц и поиска записей в таблицах.

При классификации таблиц наиболее существенным является признак, который характеризует *способ* создания таблицы. По этому признаку различают *постоянные* и *переменные* таблицы.

Если таблица формируется один раз (сразу заносятся все записи), а поиск записей в ней должен происходить многократно, то эту таблицу следует организовать как *постоянную*. Для эффективного выполнения операции поиска постоянную таблицу нужно сначала отсортировать, а затем использовать для нее *дихотомический поиск*.

На *логическом уровне* таблица представляется, например, так:

Name	Capital	Area	Population
Россия	Москва	225110	2400000
Украина	Киев	15630	400000
. . .	. . .	. . .	. . .

Здесь **Name, Capital, Area, Population** – поля таблицы.

Физический уровень: постоянная таблица отображается на вектор, причем записи следуют одна за другой без промежутков. Графическое представление вектора:

В программе физический уровень определяется описанием таблицы. Например, так

```
Const  Nmax=100;  {максимальное количество записей в таблице}
Type
    Rec=Record                {тип записи }
          kl : integer;       { ключ записи }
          inf : string [10];
        End;
    Table=Array [1..Nmax] Of Rec;  { тип таблицы }
```

Описание показывает, что таблица отображается на вектор.

Характерный признак постоянных таблиц: **обращение к записям осуществляется по ключу.**

Если операции занесения и поиска записей должны выполняться поочередно, то таблицу нужно организовать как *переменную* (*хеш-таблицу,* таблицу *с вычисляемым входом*).

Методы сортировки таблиц

Сортировка – это расстановка записей по позициям таблицы в порядке, определяемом некоторым критерием упорядочения. Сортировка может выполняться по *возрастанию* или по *убыванию* значений ключевого поля (ключа). Для символьных ключей используется лексикографическая упорядоченность.

Цель сортировки – упростить операцию *поиска записи* (записей) по заданному критерию.

Для оценки методов сортировки используются следующие показатели:

C - среднее количество операций сравнения пар ключей,
P - число перестановок записей,
ΔS - дополнительный объем памяти.

Показатели **C** и **P** зависят от метода сортировки и связаны с *n* – размером таблицы (размер таблицы – это количество содержащихся в ней записей).

Существует достаточно много методов *внутренней сортировки* (случаи, когда упорядочиваемая таблица полностью размещается в **ОП**).

Для малых таблиц можно использовать простые методы сортировки: линейный выбор, "пузырек", шейкер-сортировка, вставка с прямым включением. При больших *n* эффективны более сложные алгоритмы: метод Шелла, сортировка слиянием, "быстрая" сортировка, вставка с бинарным включением.

Однако не существует метода, являющегося оптимальным в любом случае, так как эффективность методов существенно зависит от типа ключей и их предварительной упорядоченности.

С точки зрения экономичности использования памяти методы внутренней сортировки можно разделить на две группы:

– методы, выполняющие упорядочение записей "на том же месте", то есть на участке памяти, где располагается сортируемая таблица;

– методы, требующие для проведения сортировки дополнительного объема памяти (двухпутевая вставка, методы слияния).

Замечание. В приводимых далее методах рассматривается упорядочение таблиц по **возрастанию** значений ключевого поля. При описании алгоритмов используются следующие обозначения:

R[j] – j-я запись таблицы;

K[j] – значение ключевого поля (ключа) j-й записи.

В иллюстрационных примерах для простоты рассматриваются только значения ключевого поля таблицы.

Линейный выбор (сортировка выбором)

Идея метода: при каждом проходе по таблице запись с наименьшим значением ключа становится в начальные позиции таблицы.

Согласно методу сортировки *линейным выбором*, начиная с первой записи таблицы *(i=1)*, осуществляется поиск записи, имеющей наименьшее значение ключа *(j=i, i+1, ...n)*. После того, как эта запись найдена, она меняется местами с первой записью таблицы. В результате такой перестановки запись с наименьшим значением ключа помещается в первую

позицию и в дальнейшем не рассматривается. Если же наименьшее значение ключа имеет первая запись, то перестановка не нужна.

Затем, начиная со второй записи таблицы *(i=2)*, осуществляется поиск второго наименьшего ключа. Найденная запись меняется местами со второй записью. Этот процесс заканчивается поиском места для предпоследней записи таблицы (*i=n-1*). После этого все записи будут расположены в порядке возрастания значений ключевого поля.

Число сравнений в данном методе равно *n*(n-1)/2* (в среднем порядка $n^2/2$). Максимальное количество перестановок – *(n-1)*, причем эти величины не зависят от исходного состояния ключей.

В качестве примера рассмотрим сортировку таблицы со следующими значениями ключей: **23 11 4 56 9 35 7.**
Проходы по таблице (ключи перемещаемых записей подчеркнуты):

1:	**23**	11	**4**	56	9	35	7
2:	4	**11**	23	56	9	35	**7**
3:	4	7	**23**	56	**9**	35	11
4:	4	7	9	**56**	23	35	**11**
5:	4	7	9	11	**23**	35	56
6:	4	7	9	11	23	**35**	56

Программа сортировки таблицы методом линейного выбора приведена в **Приложении А**.

Линейный выбор с подсчетом

Идея метода: число перестановок, необходимое для сортировки выбором, можно значительно сократить, если использовать для каждого ключа счетчик относительной позиции. Но для такой модификации требуется дополнительная память для результирующей (упорядоченной) таблицы и массива счетчиков.

Просмотр таблицы начинается с первой записи (*i=1,2, …n-1)*. Ее ключ сравнивается с ключами последующих записей *(j=i+1, i+2,…n)*. При этом счетчик большего из сравниваемых ключей увеличивается на 1. При втором проходе таблицы первый ключ уже не рассматривается, второй ключ сравнивается со всеми последующими. Результаты сравнений фиксируются в счетчиках. Для таблицы, содержащей *n* записей, этот процесс повторяется *(n-1)* раз.

После выполнения всех проходов счетчик каждого элемента указывает, какое количество ключей таблицы меньше ключа этого элемента. Эти счетчики используются затем в качестве индексов элементов

результирующей таблицы. Поместив записи в результирующую таблицу в соответствии со значениями их счетчиков, получим упорядоченную таблицу.

В процессе сортировки выполняется *(n-1)* проходов таблицы с *n/2* сравнениями в среднем при каждом проходе. Значения счетчиков изменяются столько раз, сколько выполняется сравнений. Рассмотрим пример использования этого метода. Ключи записей и массив счетчиков имеют вид:

Индексы	Ключи	Счетчики
0	9	0
1	5	0
2	10	0
3	2	0

Изменение массива счетчиков в результате проходов:

1-й проход	2-й проход	3-й проход	Результирующая таблица (ключи)
2	2	2	2
0	1	1	5
1	2	3	9
0	0	0	10

Метод “пузырька”

Идея метода: при каждом проходе по таблице сравниваются ключи рядом стоящих записей, и, если упорядоченность нарушена, то записи меняются местами. В результате прохода запись с наибольшим значением ключа становится в конечные позиции таблицы (на свое место) и в дальнейшем не рассматривается.

В течение первого прохода таблицы сравниваются ключи первой (*R[1]*) и второй (*R[2]*) записей, и, если порядок между ними нарушен, записи *R[1]* и *R[2]* меняются местами. Затем этот процесс повторяется для записей *R[2]* и *R[3]*, *R[3]* и *R[4]*,..., *R[n-1]* и *R[n]*, причем при выполнении сравнения записи с меньшими ключами будут "всплывать" (то есть перемещаться на позицию с меньшим индексом).

В результате первого прохода запись с наибольшим значением ключа "осядет" в позиции *n*. При втором проходе таблицы сравниваются ключи записей *R[1]* и *R[2]*, *R[2]* и *R[3]*,...,*R[n-2]* и *R[n-1]*. Запись с наибольшим значением ключа "осядет" в позиции *(n-1)* и т.д.

Для сортировки таблицы требуется максимально *(n-1)* проходов, причем при каждом проходе один "пузырек" располагается на уровне, соответствующем его весу.

Рассмотренный алгоритм можно сделать эффективнее, если после каждого прохода выполнять проверку: были ли совершены перестановки в течение данного прохода. Если перестановок не было, то это означает, что таблица уже отсортирована, и процесс можно завершить.

Кроме того, нужно запоминать не только сам факт, что обмен имел место, но и положение (индекс) последнего обмена. Это позволит уменьшить на следующем шаге просматриваемую часть таблицы.

Характеристики сортировки методом "пузырька" в худшем случае составляют *n*(n-1)/2* сравнений и *n*(n-1)/2* перестановок (худшим считается случай, когда записи наиболее удалены от своих конечных позиций). Среднее число сравнений и перестановок пропорционально $n^2/2$.

Рассмотрим пример сортировки методом "пузырька" (процедура, реализующая метод "пузырька" на языке Паскаль, приведена на рисунке 1).

Начальные ключи: 15 18 11 46 50 12 13 14 54 60 51

Проходы

1: 15 11 18 46 12 13 14 50 54 51 **60**

2: 11 15 18 12 13 14 46 50 51 **54 60**

3: 11 15 12 13 14 **18 46 50 51 54 60**

4: 11 12 13 14 **15 18 46 50 51 54 60**

5: 11 12 13 14 **15 18 46 50 51 54 60**

Пояснения:

1. Благодаря тому, что в алгоритме учитывается индекс последнего обмена записей (переменная *m*), на третьем проходе к упорядоченной части таблицы присоединяется сразу несколько записей.

2. На пятом проходе перестановок записей не было (переменная *pr* осталась со значением *false*), поэтому завершение работы алгоритма произошло досрочно.

Шейкер – сортировка

Идея метода: метод "пузырька" можно улучшить, если менять направления следующих один за другим проходов таблицы. Этот вариант метода назван *шейкер – сортировкой*. Его выгодно использовать в тех случаях, когда "легкие" пузырьки расположены в исходной таблице на противоположном ("тяжелом") конце.

```
{описание типов}
Const  Nmax=100;
Type
Rec=Record
          kl : integer;
         inf : string [10];
       End;
 Table-=Array [1..Nmax] Of Rec;
. . . . . .
Procedure bubble (Var T:Table;  n:integer);
{T – имя таблицы;  n – размер таблицы (количество записей)}
{kl – ключ записи (поле сортировки)}
Var
      i, k, m: integer;
           pr: boolean;
          zap: rec;
 Begin
      k:=n-1; pr:=true;   m:=0;
      while (k>=1) and pr do
          begin
            pr:=false;
            for i:=1 to k do
              if T[i].kl > T[i+1].kl then
                  begin
                      zap:=T[i];
                      T[i]:=T[i+1];
                      T[i+1]:=zap;
                       m:=i;
                       pr:=true;
                  end;
             k:=m-1;
          end
 end;
```

Рисунок 1 – Процедура сортировки таблицы методом “пузырька”

Рассмотрим работу *шейкер – сортировки* на следующем примере:

Начальные ключи	**Проходы**			
	1	2	3	4
15	15	1	1	1
23	8	15	8	3
8	16	8	15	8
16	3	16	3	15
3	1	3	16	16
1	20	20	20	20
20	23	23	23	23

Шейкер – сортировка выполняется быстрее, чем сортировка "пузырьком" (число сравнений равно, примерно, *(n*n)/4+n*).

Метод вставки с прямым включением

Идея метода: для каждой записи, начиная со второй, отыскивается соответствующее место среди впереди стоящих уже упорядоченных записей, то есть, перед рассмотрением записи *R[i]* (*i=2, 3, ..., n*) предыдущие записи *R[1], R[2], ..., R[i-1]* уже упорядочены (*K[1]<=K[2]<=...<=K[i-1]*) и для *R[i]* среди них должно быть найдено соответствующее место.

Рассмотрим *i*-й шаг метода *вставки с прямым включением*. Если *K[i]<K[i-1]*, то запись *R[i]* запоминается в рабочей переменной (например, *RAB*) и ее ключ сравнивается поочередно с ключами записей *R[j] (j=i-1, i-2, ..., 1)* до тех пор, пока выполняется условие *K[i]<K[j]* или достигнут левый конец упорядоченной подтаблицы (*j=0*). Выполнение условия *K[i]>=K[j]* означает, что запись *RAB* нужно вставить между записями *R[j]* и *R[j+1]*. Чтобы подготовить место для вставки (позиция (*j+1*)), операции сравнения нужно чередовать с операциями перемещения записей на одну позицию (*R[j+1]=R[j]*). Завершается *i*-й шаг операцией *R[j+1]=RAB*.

Ниже показано выполнение *i*-го шага вставки. Ключевое значение записи *R[i]*, равное 6, сравнивается с предыдущими ключами, и при необходимости записи перемещаются:

5	8	10	[14]	_6_	2	11	*i=5, j=4, 6< 14*
5	8	[10]	14	14	2	11	*j=3, 6<10*
5	[8]	10	10	14	2	11	*j=2, 6<8*
[5]	8	8	10	14	2	11	*j=1, 6>5*
5	_6_	8	10	14	2	11	*запись RAB вставлена*

Процедура сортировки таблицы методом вставки с прямым включением имеет вид:

```
{описание типов}
Const Nmax=100;
Type   rec=record
          kl:  integer;
          inf: string[10]
        end;
       Table=Array[1..Nmax] of rec;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
procedure VST (var T:Table; n:integer);
{T- имя таблицы, n – размер таблицы, сортировка по полю kl}
var
   i, j: integer;
   rab: rec;
begin
  for i:=2 to n do
      if T[i].kl<T[i-1].kl then
        begin
          rab:=T[i];
          j:=i-1;
          while (j>0) and (rab.kl<T[j].kl) do
           begin
             T[j+1]:=T[j];
             j:=j-1
           end;
         T[j+1]:= rab;
        end;
  end;
```

Количество операций сравнения для метода вставки определяется как $C = n*(n - 1)/4$. При достаточно большом n можно принять $C = n^2/4$.

Максимальное количество перестановок при использовании этого метода равно, примерно, $n^2/4$.

Метод вставки обычно используется тогда, когда нужно внести записи в упорядоченную таблицу. Новая запись должна быть вставлена в таблицу таким образом, чтобы существующая упорядоченность не нарушилась.

Метод вставки с бинарным включением

Идея метода: место для вставляемой записи в упорядоченной подтаблице отыскивается с помощью *дихотомического* поиска (деление подтаблицы на две части).

Метод вставки с прямым включением можно улучшить, если отыскивать место для вставляемой записи в упорядоченной подтаблице с помощью метода *бинарного* (*дихотомического, двоичного, логарифмического)* поиска. Эта модификация метода вставки названа *вставкой с бинарным включением.*

Рассмотрим *j*-й шаг сортировки (*j=2, 3, ..., n*). Если *K[j]>=K[j-1]*, то упорядоченность не нарушилась и следует перейти к *R[j+1]*–ой записи. Если же *K[j]<K[j-1]*, то *R[j]* запоминается в рабочей переменной *(Rab=R[j])* и для нее ищется место в упорядоченной части таблицы – в подтаблице. Обозначим нижнюю границу индекса этой подтаблицы через *ng*, верхнюю - через *vg* (первоначально *ng=1. vg=j-1*).

Согласно бинарному поиску ключ *K[j]* рассматриваемой записи *R[j]* должен сначала сравниться с ключом *K[i]* записи *R[i]*, находящейся в середине упорядоченной подтаблицы *(i=(ng+vg) div 2)*. Если *K[j]>K[i]*, то отбрасывается (то есть больше не рассматривается) левая часть подтаблицы-записи с меньшими ключами *(ng=i+1)*. Если *K[j]<K[i]*, то отбрасывается правая часть подтаблицы - записи с большими ключами *(vg=i-1).* В оставшейся части подтаблицы поиск продолжается.

Процесс деления частей подтаблицы пополам продолжается до тех пор, пока не возникнет одна из следующих ситуаций:

1) *K[j]=K[i]*, следовательно, *(i+1)*-я позиция является местом для рассматриваемой записи. Сдвинем записи *R[i+1], R[i+2], ..., R[j-1]* на одну позицию вправо и освободим тем самым место для вставки *(R[i+1]=Rab).*

2) *K[j]<>K[i]* и *ng>vg* – ключи не совпали, а длина последней подтаблицы равна 1. В этом случае местом для вставки является позиция *ng*, поэтому записи *R[ng], R[ng+1], ... , R[j-1]* должны быть сдвинуты на одну позицию вправо *(R[ng]=Rab).*

Алгоритм бинарного поиска подробно описан в разделе "Дихотомический поиск по совпадению".

Рассмотрим на примере *j*-й шаг сортировки: определяется место записи с ключом, равным **9** (*j=7, K[j]=9).*

5	6	**8**	10	14	**18**	9	2	ng=1, vg=6, i=3, 9>8;
.	.	.	**10**	**14**	**18**	9	2	ng=4, vg=6, i=5, 9<14;
.	.	.	**10**	14	18	9	2	ng=4, vg=4, i=4, 9<10;
.	.	.	9	10	14	18	2	ng=4, vg=3, ng-место для вставки

Среднее число сравнений для данного метода составляет $n^* log_2(n)$.

Метод двухпутевой вставки

Идея метода: для улучшения характеристики сортировки методом вставки с прямым включением нужно использовать дополнительный объем памяти (зона вывода *T*), равный объему, занимаемому таблицей.

На первом шаге сортировки м*етодом двухпутевой вставки* в середину зоны вывода (позиция *m=(n div 2)+1*) помещается первая запись таблицы *R[1]*. Остальные позиции *T* пока пусты. На последующих шагах сортировки ключ очередной записи *R[j]* (*j=2, 3, …, n*) сравнивается с ключом записи *T[m]*, и, в зависимости от результатов сравнения, место для *R[j]* отыскивается в *T* слева или справа от *T[m]* методом вставки. При этом должны запоминаться позиции самого левого (*l*) и самого правого (*r*) внесенных в зону вывода элементов. Конечные значения *l* и *r* равны *1* и *n* соответственно.

В алгоритме должны быть учтены также следующие ситуации:
- ключ записи *R[j]* меньше ключа записи *T[m]*, но *l=1*;
- ключ записи *R[j]* больше ключа записи *T[m]*, но *r=n*.

В этих случаях для вставки записи *R[j]* необходимо осуществлять сдвиг записей подтаблицы вместе с записью *T[m]* вправо или влево (используется метод вставки с прямым включением).

Рассмотрим пример сортировки с использованием этого метода. Пусть исходная последовательность ключей таблицы имеет вид:

24, 1, 28, 7, 25, 3, 6, 18, 8 *(n=9, m=(n div 2)+ 1=5)*

Номер шага j	Зона вывода									Поясне-ния
	1	2	3	4	5	6	7	8	9	
1					**24**					r=5, l=5
2				**1**	24					1<24, l=4
3				1	24	**28**				28>24, r=6
4			1	**7**	24	28				7<24, l=3
5			1	7	24	**25**	28			25>24, r=7
6		1	**3**	7	24	25	28			3<24, l=2
7	1	3	**6**	7	24	25	28			6<24, l=1
8	1	3	6	7	**18**	24	25	28		18<24, r=8
9	1	3	6	7	**8**	18	24	25	28	8<18, r=9

Пояснение: в каждой строке зоны вывода подчеркнут вставленный элемент.

Метод Шелла

Идея метода: таблица разбивается на группы записей, каждая группа упорядочивается методом прямой вставки. В группу входят не непосредственные соседи, а записи, отстоящие друг от друга на заданное расстояние - шаг группы. При каждом новом просмотре таблицы шаг группы уменьшается, пока не станет равным 1.

Метод Шелла (прямое включение с убывающим шагом) является эффективным усовершенствованием метода вставки. Суть метода состоит в том, что упорядочиваемая таблица разбивается на группы записей, каждая из которых упорядочивается методом вставки. В группу входят записи, отстоящие друг от друга на заданное расстояние h (шаг группы), то есть группой является совокупность записей, номера которых образуют арифметическую прогрессию с разностью h.

В начале процесса упорядочения выбирается первый шаг группы, который Шелл предложил брать в виде $h_1=[n/2]$, где n – размер таблицы. Однако Хиббард показал, что сортировка дает лучшие результаты, если первый шаг будет степенью двойки: $h_1=2^k-1$, где $2^k <= n < 2^{k+1}$.

Последующие шаги определяются по формуле: $h_i = [h_{i-1}/2]$.

Таким образом, при каждом новом просмотре таблицы расстояние между элементами группы уменьшается, количество же элементов в группе увеличивается. На последнем просмотре шаг становится равным 1.

На ранних просмотрах записи совершают большие скачки к нужным позициям, а не передвигаются на одну позицию, как в методе вставки. Это позволяет сократить число повторных сравнений и обменов на более поздних просмотрах. На последнем просмотре таблицы записи будут расположены близко к своим позициям, и, следовательно, не потребуют значительных перемещений при окончательном размещении.

Рассмотрим подробнее алгоритм метода.

После выбора h_1 методом вставки упорядочиваются группы, содержащие записи с номерами позиций i, $i+h_1$, $i+2*h_1$, ..., $i+m_i*h_1$. При этом $i=1,2,...,h_1$ (не более h_1); m_i – наибольшее целое, удовлетворяющее неравенству $i+m_i*h_1 <= n$.

Затем выбирается шаг h_2 ($h_2<h_1$), и упорядочиваются группы, содержащие записи с номерами позиций i, $i+h_2$, ..., $i+m_i*h_2$. Эта процедура с уменьшающимися шагами продолжается до тех пор, пока очередной шаг h_l станет равным единице ($h_1>h_2>...>h_l$). Этот последний этап представляет собой упорядочение всей таблицы методом вставки. Но так как исходная таблица упорядочивалась отдельными группами с последовательным объединением этих групп, то общее количество сравнений значительно меньше, чем $n^2/4$, требуемое при методе вставки. Число операций сравнения пропорционально $n\ (\log_2 n)^2$.

Рассмотрим пример сортировки методом Шелла:

Номера позиций упорядочиваемой таблицы															Шаг
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
6	1	3	15	8	14	13	**11**	10	4	9	2	5	12	**7**	h=7
6	**1**	3	15	8	14	13	7	**10**	4	9	2	5	12	11	
6	1	**3**	15	8	14	13	7	10	**4**	9	2	5	12	11	
6	1	3	**15**	8	14	13	7	10	4	**9**	2	5	12	11	
6	1	3	9	**8**	14	13	7	10	4	15	**2**	5	12	11	
6	1	3	9	2	**14**	13	7	10	4	15	8	**5**	12	11	
6	1	3	9	2	5	**13**	7	10	4	15	8	14	**12**	11	
6	1	3	**9**	2	5	**12**	7	10	**4**	15	8	**14**	13	11	h=3
4	**1**	3	6	**2**	5	9	**7**	10	12	**15**	8	14	**13**	11	
4	1	**3**	6	2	**5**	9	7	**10**	12	13	**8**	14	15	**11**	
4	1	3	6	2	5	9	7	8	12	13	10	14	15	11	h=1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

“Быстрая” сортировка (обменная сортировка с разделением)

Идея метода: на каждом шаге данного метода таблица с помощью разделяющего элемента делится на две подтаблицы таким образом, что в левой подтаблице размещаются записи, ключи которых меньше или равны ключу разделяющего элемента, а в правой – больше или равны.

Аналогичное разделение применяется к каждой из полученных подтаблиц и повторяется до тех пор, пока все записи не будут установлены на их конечные позиции.

"Быстрая" сортировка является одним из лучших методов внутренней сортировки и весьма эффективна для больших таблиц (предложена Ч. Хоаром). Среднее число сравнений для данного метода составляет $n \log_2(n)$.

Существуют два варианта "быстрой" сортировки: в качестве разделяющего элемента могут быть взяты ключ **первой** записи таблицы или ключ **средней** записи.

1. Рассмотрим первый шаг "быстрой" сортировки, взяв в качестве разделяющего элемента ключ **первой** записи таблицы. Используются два индекса i и j с начальными значениями границ таблицы ($i=1$, $j=n$). Сравниваются ключи $K[i]$ и $K[j]$, и, если перестановка записей таблицы не требуется, то j уменьшается на 1 и сравнение продолжается. В том случае, когда $K[i] > K[j]$, записи $R[i]$ и $R[j]$ меняются местами. Затем сравниваются записи с i, увеличиваемым на 1, и фиксированным j до тех пор, пока не возникнет следующая перестановка. Далее j снова будет уменьшаться на 1, а i оставаться фиксированным.

Процесс выполняется до тех пор, пока $i<j$. В итоге выполнения первого шага разделяющий элемент станет на свою конечную позицию, а таблица разобьется на две подтаблицы, записи которых имеют индексы в интервалах $[1, (i-1)]$ и $[(i+1), n]$.

После каждого шага подтаблица разбивается на две части, одна из которых обрабатывается, а границы второй должны быть сохранены для дальнейшей обработки. Среднее число сравнений для данного метода составляет $n \log_2(n)$.

В качестве примера рассмотрим записи со следующими ключами:

42 23 30 40 74 11 12 65 36 50 58 94

(начальные значения индексов: $i=1$, $j=n$).

Ниже приведена последовательность перестановок (обменов) при перемещении разделяющей записи (ключ **42)** на ее конечную позицию (первый шаг сортировки). Ключи, значения которых сравниваются, подчеркнуты:

индексы:	1	2	3	4	5	6	7	8	9	10	11	12	
	42	23	30	40	74	11	12	65	36	50	58	**94**	
	42	23	30	40	74	11	12	65	36	50	**58**	94	
	42	23	30	40	74	11	12	65	36	**50**	58	94	
	42	23	30	40	74	11	12	65	**36**	50	58	94	*i=1, j=9 - обмен*
	36	**23**	30	40	74	11	12	65	**42**	50	58	94	
	36	23	**30**	40	74	11	12	65	**42**	50	58	94	
	36	23	30	**40**	74	11	12	65	**42**	50	58	94	
	36	23	30	40	**74**	11	12	65	**42**	50	58	94	*i=5, j=9 - обмен*
	36	23	30	40	**42**	11	12	**65**	74	50	58	94	
	36	23	30	40	**42**	11	**12**	65	74	50	58	94	*i=5, j=7 - обмен*
	36	23	30	40	12	**11**	**42**	65	74	50	58	94	
	36	23	30	40	12	11	**42**	65	74	50	58	94	*i=7, j=7 - шаг завершен*

В результате исходная таблица разбита на две подтаблицы:

1) [36, 23, 30, 40,12, 11] с границами элементов *l=1*, *r=6,*
2) [65, 74, 50, 58, 94] с границами *l=8*, *r=12*.

Запись с ключом **42** в дальнейшей сортировке не участвует.

Для продолжения сортировки таблицы нужно применить этот процесс к получившимся двум частям, затем к частям частей и так до тех пор, пока каждая из частей не будет состоять из одного элемента. Запоминать специально границы разделенных частей не нужно, так как современные системы программирования позволяют реализовывать рекурсивные процедуры.

Рекурсивная процедура "быстрой" сортировки с первым разделяющим элементом может выглядеть так:

```
Procedure SORT_1 (l, r: integer);
  Var i,j:integer;
  . . . . . .
  begin
    i:=l;  j:=r;
    {процесс разделения}
. . . . . .
    if (i-1)>l  then  SORT_1 ( l, i-1 );
    if (i+1)<r then  SORT_1 ( i+1, r );
 end;
```

2. Возьмем в качестве разделяющего элемента ключ **средней** записи таблица и обозначим его через ***X***. На первом шаге сортировки ***X***=*K[n/2]*.

Сравниваем с *X* сначала ключи *K[i]* записей, расположенных слева от *X*. Если *K[i]<X*, то *i:=i+1* (*i=1* – начальное значение) и сравнения продолжаются. Как только *K[i]>=X*, *i* фиксируется, и начинается просмотр ключей справа от *X*. Если *K[j]>X*, то *j:=j-1* (*j=n* – начальное значение) и сравнения продолжаются. При *K[j]<=X* фиксируется *j*. Как только *i* и *j* зафиксированы, происходит обмен местами записей *R[i]* и *R[j]* и изменение значений индексов (*i:=i+1, j:=j-1*).

Процесс сравнения ключей с *X* и обмена записей продолжается до тех пор, пока *i<=j*. При *i>j* первый шаг алгоритма завершается, и таблица становится разделенной на две подтаблицы: левую - с ключами, меньше или равными *X,* и правую – с ключами, больше или равными *X*. Элементы этих подтаблиц имеют индексы в интервалах *[l, j]* и *[i, r]*. Далее метод разделения применяется к полученным подтаблицам, каждая из которых также делится на две части.

Сортировка продолжается до тех пор, пока каждая из подтаблиц не будет состоять из одного элемента. Среднее число сравнений для данного метода составляет $n \log_2(n)$.

В качестве примера рассмотрим записи со следующими ключами:

47 16 22 89 11 30 85 40 13 90 94 20 99 87

Первый шаг сортировки: разделяющий элемент *X= K[7]=* **85**; *i= 1, j= 14*.

индексы:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
47	16	22	**89**	11	30	85	40	13	90	94	20	99	87	
47	16	22	89	11	30	85	40	13	90	94	**20**	99	87	- *обмен*
47	16	22	**20**	11	30	85	40	13	90	94	**89**	99	87	
47	16	22	20	11	30	**85**	40	13	90	94	89	99	87	
47	16	22	20	11	30	85	40	**13**	90	94	89	99	87	- *обмен*
47	16	22	20	11	30	**13**	40	**85**	90	94	89	99	87	
47	16	22	20	11	30	13	40	**85**	90	94	89	99	87	
47	16	22	20	11	30	13	40	**85**	90	94	89	99	87	*i=9, j=8, i>j - шаг завершен*

В результате выполнения первого шага сортировки заданная таблица будет разбита на две подтаблицы с границами элементов *l=1, r=j* и *l=i, r=n.*

Рассмотренную разновидность метода "быстрой" сортировки также нужно реализовать методом рекурсии:

```
Procedure SORT_sr (l, r: integer);
  var i,j:integer;
  . . . . . .
begin
  {процесс разделения}
. . . . . .
  if  j>l  then  SORT_sr ( l, j );
  if  i<r then  SORT_sr ( i, r );
end;
```

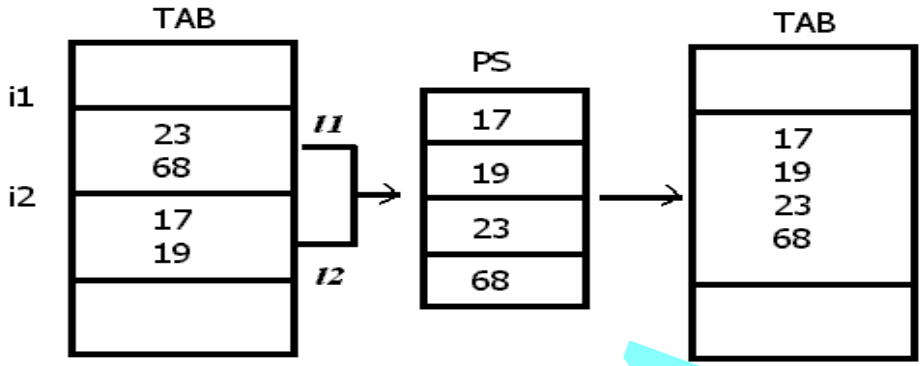
Метод простого двухпутевого слияния (сортировка слиянием)

Идея метода *простого двухпутевого слияния* состоит в том, что упорядочиваемая таблица на каждом этапе сортировки представляется равными упорядоченными группами записей, которые попарно *сливаются*, образуя новые группы, содержащие вдвое больше записей.

Слияние - объединение двух упорядоченных таблиц таким образом, чтобы получилась одна упорядоченная таблица. На первом этапе группы состоят из одной элемента (шаг слияния (*h*) равен 1), на втором этапе – из двух элементов (*h=2*), на третьем этапе – из четырех элементов (*h=4*) и так далее. Таким образом, количество групп на каждом этапе сортировки уменьшается вдвое, пока не будет получена одна упорядоченная группа.

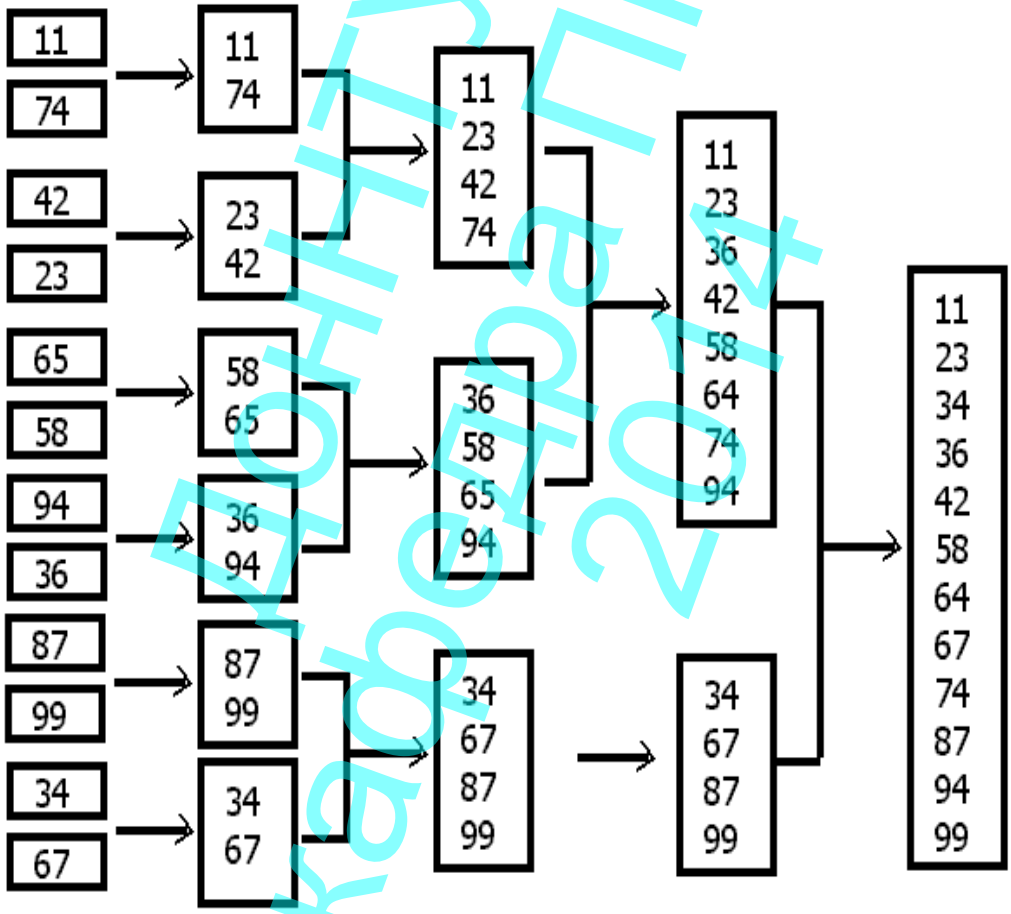
Особые случаи метода:
1) если число групп, сформированных на некотором этапе сортировки, нечетно, то последняя (непарная) группа не участвует в процессе слияния на данном этапе;
2) длины сливаемых групп одинаковы кроме, быть может, последней группы.

Процедура слияния двух групп должна иметь такие параметры:
TAB – имя сортируемой таблицы;
n – длина таблицы (количество записей);
i1, i2 – начальные индексы сливаемых групп;
l1, l2 – длины сливаемых групп.
В процедуре для слияния групп должен быть выделен дополнительный объем памяти *PS* (таблица слияния.).



Рассмотрим пример сортировки слиянием таблицы со следующими ключами:

11 74 42 23 65 58 94 36 87 99 34 67



Шаг слияния, первоначально равный 1, удваивается при переходе к каждому последующему этапу сортировки *(h=2*h)*.

На любом этапе сортировки первые две сливаемые группы должны иметь такие параметры: *i1=1, l1=h, i2=i1+h, l2=h* (*l2* может быть равным *(n-i2+1)* – для последней группы таблицы).

При переходе к слиянию следующих двух групп параметры изменяются следующим образом: *i1=i1+2*h, l1=h, i2=i1+h, l2=h* или, возможно, *l2=n-i2+1*.

Сортировка продолжается до тех пор, пока *h<n*.

Рассмотренный метод сортировки двухпутевым слиянием весьма эффективен. Поскольку при сортировке нужно выполнить $log_2 n$ этапов, то необходимое суммарное число сравнений равно, примерно, $n*log_2 n$. Одним из недостатков данного метода является требование дополнительного резерва памяти, равного объему исходной таблицы

Метод естественного слияния (сортировка слиянием)

Идея метода: в методе *естественного слияния* учитывает тот факт, что в таблице могут изначально находиться упорядоченные последовательности записей произвольной длины, которые можно сразу объединить в одну упорядоченную группу (при простом двухпутевом слиянии эти упорядоченные последовательности разорвутся, и их элементы будут включены в разные группы).

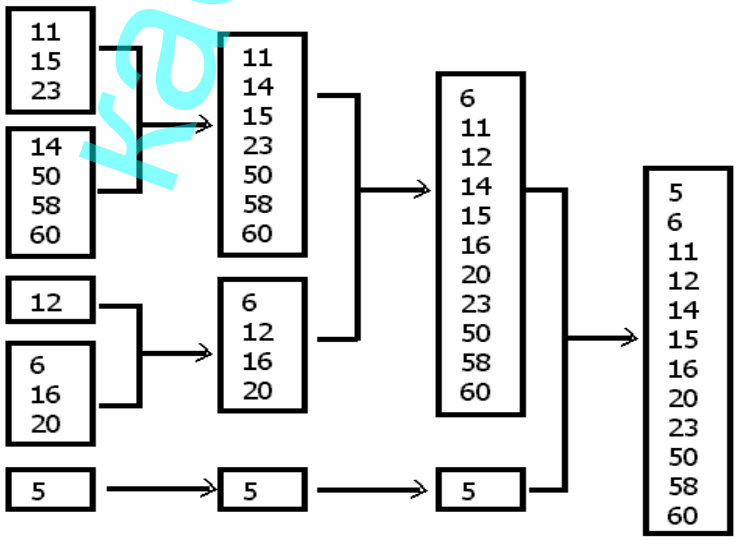
Цель естественного слияния – исключить лишние просмотры.

Процедура слияния двух групп такая же, как и в методе простого слияния, но длины *l1* и *l2* сливаемых групп нужно каждый раз подсчитывать.

На первом этапе сортировки длины групп определяются сравнением ключей двух рядом стоящих записей. При этом нужно составить массив длин упорядоченных групп. Элементы этого массива будут использованы для вычисления длин групп на последующих этапах сортировки.

Рассмотрим пример сортировки таблицы со следующими ключами:

11, 15, 23, 14, 50, 58, 60, 12, 6, 16, 20, 5.



Вспомогательный массив длин групп :

1)	2)	3)
3	7	11
4	4	1
1	1	
3		
1		

Определение операции поиска

Поиском называется операция выделения из множества записей подмножества, записи которого удовлетворяют заранее поставленному условию. Возможны следующие критерии поиска:

- по совпадению,
- по интервалу,
- по близости,
- по арифметическому условию,
- по семантическому условию,
- по нескольким условиям.

Алгоритмы указанных разновидностей поиска можно получить из алгоритма *поиска по совпадению*, при котором задается поисковый признак (*ключ поиска*) и требуется выделить все записи, удовлетворяющие ему.

Линейный поиск

Простейшим методом поиска записи (записей) с заданным ключом является *последовательный* (*линейный*) поиск. Он применим как к упорядоченным, так и к неупорядоченным таблицам. Метод заключается в том, что последовательно, начиная с первой, просматриваются записи таблицы, пока не будет найдена запись (записи) с заданным ключом.

Число сравнений ключей (*длина поиска*), требующееся для нахождения определенной записи, при последовательном поиске равно, в среднем, $(n+1)/2$, максимальная длина поиска (в наихудшем случае) равна n. Очевидно, что затраты времени на такой поиск весьма существенны, поэтому он используется в основном для небольших неупорядоченных таблиц.

Для уменьшения времени поиска нужно сначала **упорядочить таблицу**, а затем использовать методы, более эффективные по сравнению с последовательным поиском.

Дихотомический поиск по совпадению

Существует группа методов поиска, в основе которых лежит деление упорядоченной таблицы на части. На практике наибольшее распространение получил метод деления на две части (подтаблицы) - *дихотомический*

(*бинарный, двоичный, логарифмический) поиск*. В зависимости от результата сравнения ключа срединной записи и ключа поиска одна из подтаблиц отбрасывается, и поиск продолжается в другой подтаблице. В процессе поиска участвуют границы рассматриваемых подтаблиц: *ng* – нижняя граница индекса, *vg* –верхняя граница индекса. Первоначально *ng=1, vg=n.*

Пусть таблица упорядочена по возрастанию значений ключа (*K[1] <=K[2]<=...<=K[n]*), и нужно найти запись с ключом ***KL***. Предполагается, что в таблице может быть только **одна** запись с заданным ключом (или ни одной).

Согласно бинарному поиску ключ ***KL*** должен сначала сравниться с ключом *K[i]* записи *R[i]*, находящейся в середине таблицы *(i=(ng+vg) div 2)*. Если ***KL*** *>K[i]*, то отбрасывается (то есть больше не рассматривается) левая подтаблица - записи с меньшими ключами *(ng=i+1)* и поиск продолжается в правой подтаблице. Если ***KL***<*K[i]*, то отбрасывается правая подтаблица - записи с большими ключами *(vg=i-1)* и дальнейший поиск осуществляется в левой подтаблице.

Процесс деления подтаблиц пополам продолжается до тех пор, пока не возникнет одна из следующих ситуаций:

1) *KL=K[i]* - запись найдена;

2)*KL<>K[i], ng>vg* – ключи не совпали, а длина последней подтаблицы равна 1; это означает, что искомой записи в таблице нет.

Средняя длина дихотомического поиска равна $log_2 n$ - *1*. В худшем случае понадобится $log_2 n$ + *1* сравнений. Это значительно лучше, чем при последовательном поиске.

В качестве примера рассмотрим поиск записи с ключом равным **30** *(KL=30)* в упорядоченной по возрастанию таблице (приведены ключи записей):

индексы:	1	2	3	4	5	6	7	8	9	10	11	
ключи:	10	13	18	22	25	27	30	33	38	40	45	ng=1, vg=11,i=6, 30>27
			.	.	.		30	33	38	40	45	ng=7, vg=11,i=9, 30<38
			.	.	.		30	33	.	.	.	ng=7, vg=8, i=7, 30=30
												запись найдена

Процедура дихотомического поиска приведена на рисунке 2 (предполагается, что в таблице может быть только **одна** запись с искомым ключом или ни одной).

Замечание

На практике используется также следующая модификация метода: выбирается наибольшая степень двойки, не превосходящая длину таблицы, и на первом шаге обращаются к записи с номером, равным полученной степени. Если по результатам сравнения одна часть таблицы отбрасывается, то в оставшейся части также выбирается запись с номером, равным наибольшей степени двойки, и так далее.

```
Const Nmax=100;
 Type
        Rec=Record
              kl : integer;       {ключ записи}
              inf :string[10];  {информационное поле}
            End;
        Table = Array[1..Nmax] Of Rec;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Procedure Search (Var T: Table; n, key: integer;  var k: integer);

{ T – таблица, n-число записей  в таблице,  key- значение ключа поиска}
{  k-номер найденной записи (k=0 при неуспешном поиске)}

Var
      ng ,vg, i: integer;
              fl: boolean;
   Begin
      ng:=1; vg:= n;
      fl:=false; k:=0;
     while (ng<=vg) and not fl do
       begin
          i:=(ng+vg) div 2;
          if key<T[i].kl then
             vg:=i-1
         else
             if key>T[i].kl then
               ng:=i+1
             else
                begin
             fl:=true;
             k:=i
               end
       end
    end;
```

Рисунок 2 – Процедура дихотомического поиска записи

Дихотомический поиск нескольких записей

Если в таблице может быть **несколько** записей с заданным значением ключа, то в процедуре поиска нужно определить индекс первой (*i1*) и индекс последней (*i2)* записей с этим ключом.

В качестве примера рассмотрим поиск записей с ключом равным **33** ***(KL=33)*** в упорядоченной по возрастанию таблице (приводятся ключи записей):

индексы:	1	2	3	4	5	6	7	8	9	10	11	
ключи:	10	13	18	22	25	27	30	33	33	33	33	ng=1, vg=11,i=6, 33>27
			.	.	.		30	33	33	33	33	ng=7, vg=11,i=9, 33=33
			.	.	.		30	33	33	33	33	i1=8, i2=11
												записи найдены

Процедура поиска в этом случае имеет вид:

```
Const Nmax=100;
 Type
        Rec=Record
              kl : integer;      {ключ записи}
              inf :string[10];  {информационное поле}
            End;
        Table = Array[1..Nmax] Of Rec;
. . . . . .
Procedure search_2 (Var T: Table; key, n: integer;  var i1, i2: integer);
{ T – таблица, n-число записей  в таблице,  key- значение ключа поиска}
{ i1-индекс  первой найденной записи, i2 - индекс последней}
Var
      ng ,vg, i: integer;
              fl: boolean;
   Begin
      ng:=1;  vg:= n;   fl:=false;
      i1:=0;  i2:=0;
     while (ng<=vg) and not fl do
       begin
          i:=(ng+vg) div 2;
          if key<T[i].kl then
             vg:=i-1
         else
             if key>T[i].kl then
```

```
                ng:=i+1
            else
                fl:=true;
        end ;
    if fl then
     begin
        i1:=i;   i2:=i;
        while (i1>1) and (T[i1-1].kl=key) do
          i1:=i1-1;
        while (i2<n) and (T[i2+1].kl=key) do
          i2:=i2+1;
       end;
end;
```

Дихотомический поиск по близости

Задачей поиска *по близости* является определение одной или двух записей, ключи которых наиболее близки к значению ключа поиска (в частности, возможно точное совпадение ключей записи и поиска). Алгоритм такого поиска основывается на алгоритме поиска по совпадению.

В качестве примера рассмотрим следующую последовательность ключей (ключи записей упорядоченной таблицы):

индексы записей	1	2	3	4	5	6	7	8	9	10
ключи	5	8	12	15	18	20	21	25	28	30

Определить записи, **наиболее близкие** к задаваемому ключу ***KL*:**

1) ***KL***=22

Результат поиска: запись с ключом 21 (индекс равен 7).

2) ***KL***=27

Результат поиска: запись с ключом 28 (индекс равен 9).

3) ***KL***=19

Результат поиска: записи с ключами 18 и 20 (индексы равны 5, 6).

4) ***KL***=12

Результат поиска: запись с индексом 3 (точное совпадение).

Используя поиск по близости, можно также найти записи, ключи которых **больше (меньше) заданного.** Например:

1) Определить записи, ключи которых >=22.

Результат поиска: записи с индексами от 8 до 10.

2) Определить записи, ключи которых <=15.
Результат поиска: записи с индексами от 1 до 4.

Дихотомический поиск по интервалу

Задачей поиска *по интервалу* является определение записей таблицы, ключи которых входят в **задаваемый интервал значений.**

Поиск заключается в определении положения границ интервала. Сначала нужно найти элемент, **больший** или **равный** значению нижней границы интервала, затем – элемент, **меньший** или **равный** значению верхней границы. Индексы найденных элементов и определяют границы для элементов, попадающих в заданный интервал значений ключей. Алгоритм такого поиска основывается на алгоритме поиска по совпадению.

В качестве примера рассмотрим следующую последовательность ключей (ключи записей упорядоченной таблицы):

индексы записей	1	2	3	4	5	6	7	8	9	10
ключи	5	8	12	15	18	20	21	25	28	30

Определить записи, ключи которых попадают в **задаваемый интервал:**

1) интервал [10, 20]
Результат поиска: записи с индексами 3..6.
2) интервал [16, 40]
Результат поиска: записи с индексами 5..10.
3) интервал [4, 35]
Результат поиска: записи с индексами 1..10 (все записи таблицы попадают в заданный интервал).
4) интервал [16, 17]
Результат поиска: в таблице нет записей с ключами из заданного интервала.
5) интервал [35, 40]
Результат поиска: в таблице нет записей с ключами из заданного интервала.

ЛАБОРАТОРНАЯ РАБОТА 2

ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Цель работы: ознакомление со списковыми структурами данных и операциями их обработки.

Задание к лабораторной работе

Написать программу создания и обработки односвязного списка. В качестве варианта выбрать номер по журналу.

Варианты операций обработки

* 1. Слить два упорядоченных списка в один, тоже упорядоченный.
2. Включить в список M задаваемых элементов перед K-м по счету элементом списка.
3. Удалить из списка M элементов, стоящих после K-го по счету элемента списка.
**4.Удалить из упорядоченного списка элементы с уникальными значениями ключа.
* 5. Оставить в упорядоченном списке только элементы с уникальными значениями ключа (из нескольких элементов с одинаковыми значениями ключа оставить только один элемент).
6. Удалить из неупорядоченного списка все элементы с отрицательными значениями ключа.
** 7. В заданном списке поменять местами K-й и M-й по счету элементы.
8. Инвертировать заданный список (то есть перестроить в списке связи между элементами в обратном порядке).
* 9. В неупорядоченном списке удалить те элементы, для которых выполняется условие: ключ элемента меньше ключа следующего за ним элемента.
10. В заданном списке поменять местами последний и предпоследний элементы.
11.Разделить упорядоченный список на два списка по заданному значению ключа: второй список должен начинаться с элемента, ключ которого задан.
12. Включить в список M задаваемых элементов после K-го по счету элемента списка.
13.Удалить из неупорядоченного списка все элементы с четными значениями ключей.
14. Включить в список M задаваемых элементов перед предпоследним элементом списка.
* 15. Поменять местами второй и предпоследний элементы списка.
* 16. Разделить заданный список на два списка: в первый список включить элементы с четными значениями ключей, во второй - с нечетными

значениями.
* 17. Разделить заданный список на два списка: в первый список включить элементы с положительными значениями ключей, во второй - с отрицательными значениями.
18. Удалить из неупорядоченного списка все элементы с заданным значением ключа.
19. Удалить из заданного списка два элемента, стоящие перед последним элементом списка.
20. Поменять местами первый и последний элементы списка.
**21. Удалить из упорядоченного списка элементы, ключи которых находятся в задаваемом интервале.
22. Удалить из упорядоченного списка все элементы с заданным значением ключа.
23. Удалить из неупорядоченного списка все элементы с ключами, меньшими заданного значения.
24. Поменять местами второй и последний элементы списка.
25. Включить в список M задаваемых элементов перед последним элементом списка.
*26. В неупорядоченном списке поменять местами элементы с минимальным и максимальным значениями ключа.
27. Удалить из упорядоченного списка элементы с ключами, больше или равными заданному значению.
28. Включить в список задаваемый элемент перед K-м по счету элементом списка.
29. Включить в список задаваемый элемент после K-го по счету элемента списка.
30. Удалить из списка элемент, стоящий после K-го по счету элемента списка.
31. Удалить из списка элемент, стоящий перед K-ым по счету элементом списка.
32. Включить в список задаваемый элемент перед предпоследним элементом списка.
33. Удалить из списа элемент, стоящий перед предпоследним элементом списка.

Требования к выполнению лабораторной работы

1. Алгоритмы операций обработки односвязного списка необходимо оформить в виде процедур (функций) с соответствующими формальными параметрами. В алгоритме заданной операции нужно предусмотреть все особые случаи.
2. **Элементы списка вводить в интерактивном режиме.**
3. На экран выводить:
   - заданный список;
   - результаты выполнения операции.

4. Привести примеры заданной операции обработки списка (общий и особые случаи).

В **Приложении Б** приведен пример программы, выполняющей обработку списка.

Контрольные вопросы

1. Определение списка, стека, очереди; отображение в оперативной памяти; операции над этими структурами.

2. Определение динамической переменной; процедуры создания и освобождения динамической переменной, их действие.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ РАБОТЫ

Связные списки

Отображение структур данных на вектор, то есть на последовательное распределение памяти, возможно лишь в определенных приложениях. Существует большой класс задач, в которых такое отображение приводит к неэффективному использованию памяти, потере машинного времени или вообще неприемлемо. Например, если над данными выполняются многочисленные операции **включения** и **исключения**, то они потребуют при последовательном размещении данных сдвигов значительных объемов информации и, кроме того, для таких данных заранее невозможно точно определить размер памяти, необходимый для их хранения.

В указанных случаях удобнее пользоваться *связным* (*динамическим*) распределением памяти, базирующимся на использовании *указателей* (*ссылок*) при обращении к данным. *Динамическое распределение* приводит к более эффективному использованию памяти, так как в этом случае память под отдельные данные выделяется в процессе выполнения программы, а не во время компиляции.

К структурам данных, которые удобно отображать в динамической памяти, относятся *список, стек, очередь, дерево*. Простейшей структурой среди перечисленных является *список*.

Определение:

Список – это линейная динамическая структура данных, элементы которой связаны указателями. Каждый элемент списка состоит из двух частей:

INF	*UKZ*

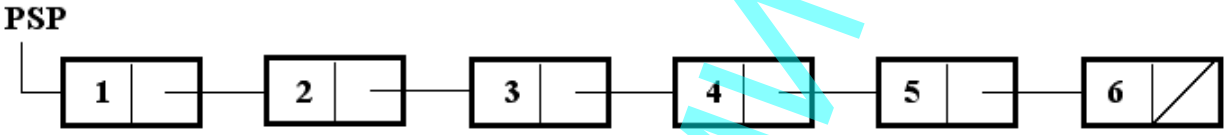
В *INF* содержится информация, представляемая одним полем или группой полей, среди которых одно является ключевым. В более сложных

случаях в *INF* может содержаться ссылка на участок памяти, где хранится информация, или на другой список.

Поле *UKZ* содержит *указатель* (*ссылку*) на элемент списка, следующий за данным элементом в логической последовательности, то есть *UKZ* содержит адрес следующего элемента.

(**конец определения**)

Таким образом, на *логическом уровне* список можно представить, например, так:



Здесь

PSP – *указатель списка* (содержит адрес первого элемента);

1, 2, … - информация;

“ / ” означает конец списка.

Список может содержать любое количество элементов.

Линейная структура - все элементы структуры находятся на одном уровне.

Различают следующие виды списков:

- односвязные (однонаправленные): каждый элемент содержит указатель на следующий элемент списка (рис.3);
- двусвязные (двунаправленные): в каждом элементе имеются указатели как на следующий, так и на предыдущий элементы того же списка (рис.4);
- кольцевые: в последнем элементе списка содержится указатель на первый элемент (рис.5).

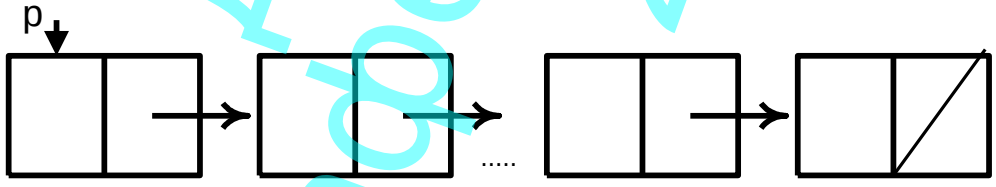


Рисунок 3-Односвязный список

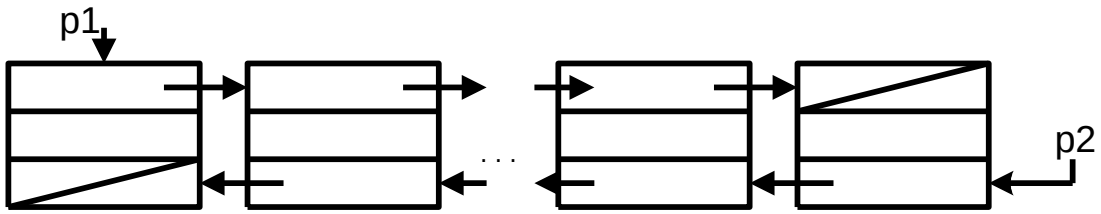


Рисунок 4- Двусвязный список

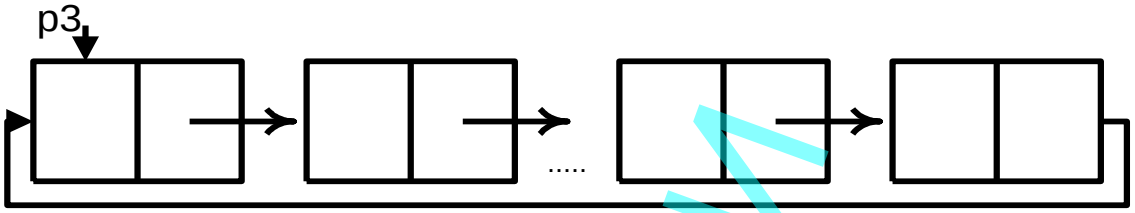


Рисунок 5-Кольцевой список

Элементы списка – *динамические переменные* (**ДП**). Они создаются (т.е. им выделяется память) во время выполнения программы, следуя потребностям решаемой задачи.

ДП можно также “уничтожать“ во время выполнения программы: отведенная ранее память освобождается (становится недоступной) .

Так как **ДП** невозможно обозначить идентификатором, то единственным средством доступа к ней является *указатель* на место ее текущего расположения в памяти. Доступ к элементам списка осуществляется через *указатель списка* (на рисунках - *P*, *P1*, *P2*, *P3*).

В системе Pascal *динамически распределяемая область памяти* (“куча”) поддерживается с помощью специальной программы. Введены так называемые *ссылочные переменные* (переменные типа “указатель”).

Физический уровень: список отображается в динамически распределяемой области памяти. В программе тип элемента списка указывается так:

```
TYPE            { описание типов}
     U = ^ZAP;
     ZAP = Record
           inf : integer;
           ukz : u;
         End;
VAR             { описание переменных}
    PSP, P : u;
```

ZAP– имя типа; определяет тип элемента списка – *запись*.

U – имя типа (тип – *указатель*); знак “^” (каррет) показывает, что это тип элементов, которые будут находиться в динамической памяти.

Запись ***U = ^ZAP*** показывает, что тип – указатель *и* связан с переменной типа ***ZAP***, определяющей структуру элемента списка.

PSP, ***P*** – указатели; они связаны с типом ***ZAP*** (и только с этим типом).

Выделение памяти для **ДП** осуществляется при выполнении системной процедуры *New(Psp)* (*Psp* – параметр процедуры, имеет тип - *указатель*):

под переменную типа ***ZAP*** выделяется участок памяти, и его адрес присваивается указателю *Psp* (в *Psp* засылается адрес первого байта выделенного участка памяти).

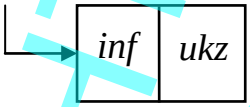
Обращение к полям записи, адресуемой с помощью указателя *Psp*, осуществляется посредством точечной нотации:

Psp^.inf – обращение к полю *inf*;

Psp^.ukz – обращение к полю *ukz*.

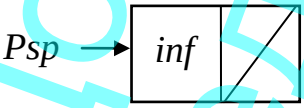
Первый элемент списка можно создать следующей последовательностью операторов:

New(Psp) – выделяется память под переменную типа *ZAP*, и ее адрес присваивается переменной *Psp*:



ReadLn (*Psp^.inf*);

Psp^.ukz:=nil;



Получили список, состоящий из одного элемента, *Psp* - указатель списка.

Системная константа *Nil* определяет пустой указатель, что означает отсутствие ссылки на другой элемент. Если в поле *Ukz* не будут занесены некоторый указатель или *Nil*, то будем иметь неопределенный указатель, использование которого приводит к непредсказуемым последствиям.

Так как переменные-указатели содержат адреса, то для них допустимы только операции сравнения (“=” и “<>”) и оператор присваивания.

При исключении элемента из списка нужно вернуть занимаемую им память в "кучу" с тем, чтобы она снова могла быть использована (например, для создания нового элемента, включаемого в список).

При выполнении системной процедуры *Dispose(Psp)* область памяти, адресуемая указателем *Psp*, освобождается (становится недоступной) и возвращается в "кучу". При этом указатель *Psp* не должен иметь значение *Nil* или быть неопределенным.

Связное представление облегчает операции включения и исключения элементов из списка. Также значительно проще выполнить объединение двух списков или разделение списка на два по сравнению с выполнением таких же операций над массивами или таблицами. Это осуществляется простой заменой значений указателей и не требует перемещения данных в памяти.

Над списками определены следующие **основные** операции:
- *включение* элемента в список,
- *поиск* элемента по заданному ключу,
- *исключение* элемента из списка.

Следует помнить, что при выполнении операций над списками не должно происходить физическое перемещение элементов в памяти: должны изменяться только связи между ними.

На рис. 6 приведена процедура **создания неупорядоченного списка** (связывание элементов в "цепочку" по мере их поступления).

Формальный параметр процедуры *PSP* – указатель списка, является выходным параметром.

```
Procedure Create_List (Var PSP:U);
Var  Q,      {указатель на предыдущий элемент списка}
       P : U;  {указатель на текущий элемент списка }
      Ch : Char;
Begin
        New(Q);
        WriteLn('Введите информационное поле элемента');
        ReadLn (Q^.INF);
        PSP:=Q;
        WriteLn('Продолжить ввод? y/n');
        Ch:=ReadKey;
        While Ch in ['y','Y','н','Н'] do
        Begin
           New(P);
           WriteLn ('Введите информационное поле элемента');
           ReadLn (P^.INF);
           Q^.UKZ:=P;
           Q:=P;
           WriteLn ('Продолжить ввод? y/n');
           Ch:=ReadKey;
        End;
        Q^.UKZ:=Nil
End;
```

Рисунок 6 - Процедура создания неупорядоченного списка

Рассмотрим пример выполнения процедуры Create_List. Для чисел **5, 3, 4, 2** получается следующий список:

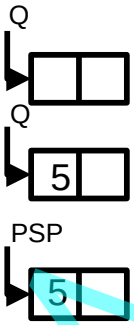
Этапы выполнения процедуры:

New(Q);

ReadLn (Q^.Inf);

PSP:=Q;

Ch:=ReadKey; {Ch='y'}



Первый шаг цикла:

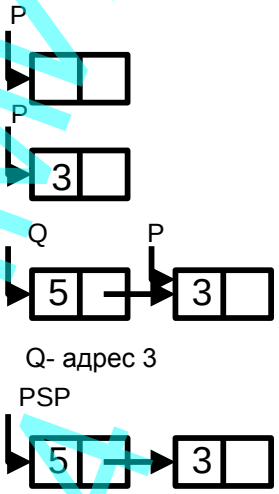
New(P);

ReadLn(P^.Inf);

Q^.Ukz:=P;

Q:=P;

Ch:=ReadKey; {Ch='y'}



Второй шаг цикла:

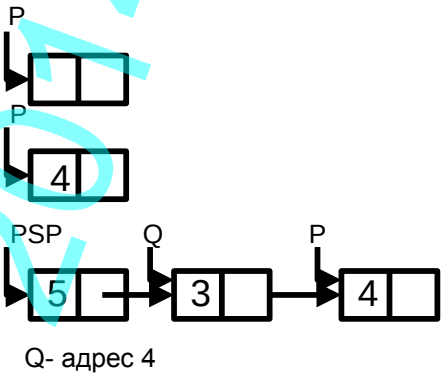
New(P);

ReadLn(P^.Inf);

Q^.Ukz:=P;

Q:=P;

Ch:=ReadKey; {Ch='y'}



Третий шаг цикла:

New(P);

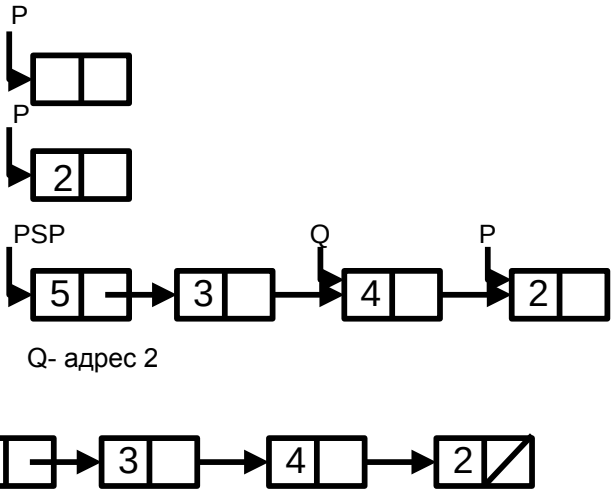
ReadLn(P^.Inf);

Q^.Ukz:=P;

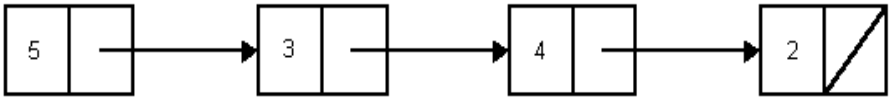
Q:=P;

Ch:=ReadKey; {Ch<>'y'

– выход из цикла}



Q^.Ukz:=Nil;



Алгоритм построения неупорядоченного списка очень прост. Однако, при поиске записи по заданному ключу нужно рассматривать в таком списке все элементы подряд и, возможно, до конца списка. Чтобы операция поиска была эффективной, следует создавать упорядоченный список - список, в котором элементы упорядочены по возрастанию значений ключевого поля.

С помощью точечной нотации можно обратиться не только к текущему элементу списка. Например, синтаксически правильными будут следующие обращения:

Psp^.ukz^.inf – к информационной части, содержащей число 3;

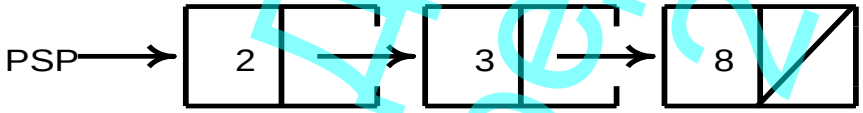
Psp^.ukz^.ukz – к полю-указателю, содержащему адрес третьего элемента;

Psp^.ukz^.ukz^.inf – к информационной части третьего элемента списка, содержащего число 4.

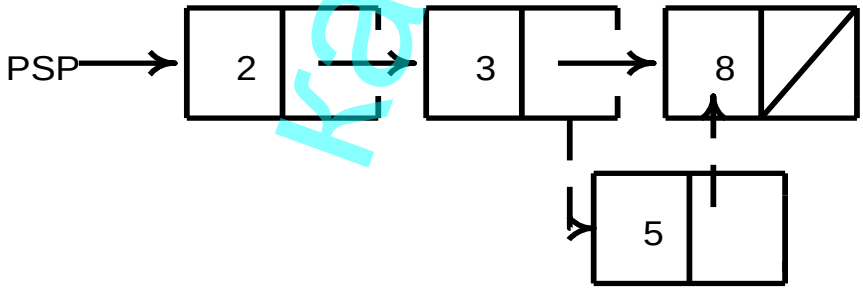
На рис. 7 приведена процедура *VKLSP*, реализующая **включение элемента в упорядоченный список** (элементы списка упорядочены по возрастанию значений ключевого поля).

Формальные параметры процедуры: *PSP* – указатель списка; *VKL* – указатель (адрес) на включаемый элемент, который уже подготовлен:

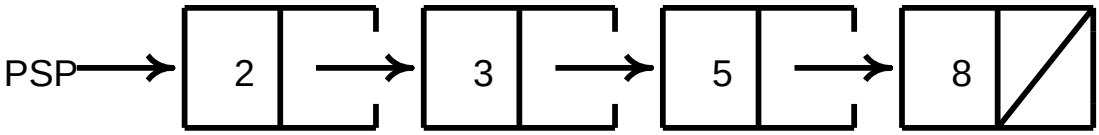
Например, исходное состояние списка:



Включаемый элемент: **5**



Результат включения:



```
Procedure VKLSP (Var PSP,VKL : U);
    Var  P,Q : U;
       Begin
          If PSP=nil Then
            Begin
               PSP:=VKL;
               exit
            End;
          If VKL^.INF<=PSP^.INF  Then
          { включение перед первым элементом }
          Begin
             VKL^.UKZ:=PSP;
              PSP:=VKL;
              Exit
          End;
          {общий случай}
          P:=PSP;
         While (VKL^.INF>P^.INF) And (P^.UKZ<>Nil) Do
          Begin
             Q:=P; { Q- указатель предыдущего элемента списка }
             P:=P^.UKZ; { P- указатель текущего элемента списка }
          End;
          If VKL^.INF<=P^.INF  Then
           { включение в середину списка }
            Begin
              VKL^.UKZ:=Q^.UKZ;
              Q^.UKZ:=VKL
            End
          Else  { включение в конец списка }
             P^.UKZ:=VKL
          End;
```

Рисунок 7- Процедура включения элемента в упорядоченный список

Особый случай: включение перед первым элементом списка (указатель списка должен поменять значение).

На рисунке 8 приведена процедура *ISKL*, реализующая **исключение по заданному ключу элемента из упорядоченного списка**.

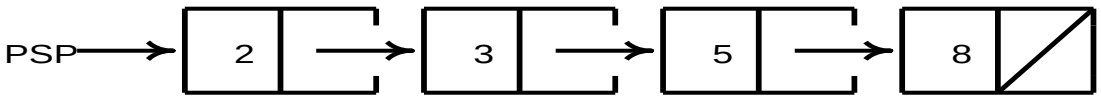
Формальные параметры процедуры:

PSP – указатель списка;
KL – заданный ключ;
PR–признак: если элемент с заданным ключом отсутствует, то *PR=0*.

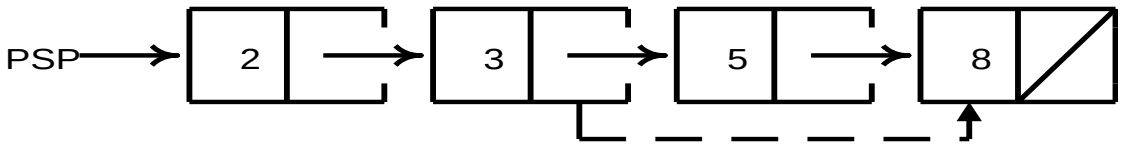
```
Procedure ISKL( Var PSP:U; Var PR: integer; KL: integer);
   Var     P, Q: U;
    Begin
      PR:=0;
      If  PSP=nil  Then
         Begin
            Writeln ( ‘Список пуст! ‘);
            Readkey;
            Exit;
        End;
      PR:=1;
     If PSP^.INF=KL     Then
     { исключаемый элемент-первый }
        Begin
          Q:=PSP;
          PSP:=PSP^.UKZ;
          Dispose(Q);
          Exit;
       End;
     {общий случай}
     P:=PSP;
     While (KL>P^.Inf) and (P^.Ukz<>Nil) do
       Begin
         Q:=P;
         P:=P^.UKZ;
       End;
      If P^.INF=KL    Then
       Begin
         Q^.UKZ:=P^.UKZ;
         Dispose(P);
       End
      Else      { в списке нет элемента с заданным ключом }
        PR:=0;
    End;
```

Рисунок 8 - Процедура исключения элемента из упорядоченного списка

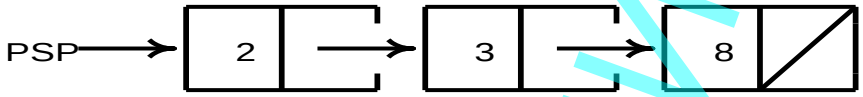
Например, исходное состояние списка:



Заданный ключ KL=5



Результат исключения:



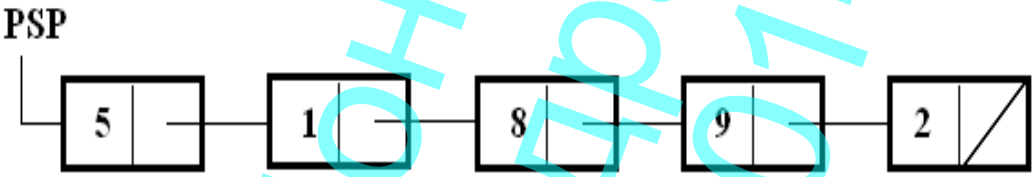
Особый случай: исключение первого элемента списка (указатель списка должен поменять значение).

Пример 1

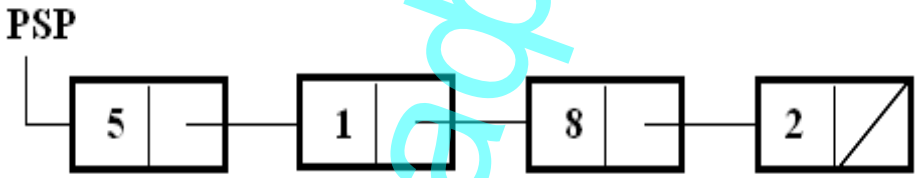
Написать процедуру удаления из заданного списка *К-го* по счету элемента (например, *К=4).*

Общий случай

Пусть задан следующий список:



Результат удаления 4-го по счету элемента:



Особые случаи:

1) список пуст (Psp=nil) или *К<=0* – операция не выполнима;
2) *К=1* – удаление первого элемента (изменится значение указателя списка);
3) *К* больше количества элементов списка.

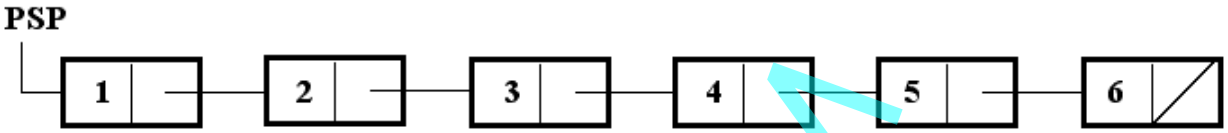
Следует также проверить пограничные случаи при *К=5* и *К=6.*

```
Procedure ISKL_K (Var Psp:u;  K: integer);
  Var     P, Pt: u;
  Begin
    If  (Psp=nil) or  (K<=0) Then
     Begin
       Writeln ( 'Операция не выполнима! ');
       Readkey;
       Exit;
     End;
   If K=1   Then
    {удаление первого элемента}
      Begin
         Pt:=Psp;
         Psp:=Psp^.ukz;
         Dispose(Pt);
         Exit;
      End;
    {общий случай}
     Pt:=Psp;
     i:=1;
     While  (Pt^.ukz<>nil) and (i < (k-1)) do
      Begin
        Pt:=Pt^.ukz; {на след. элемент}
        i:=i+1;
      End;
   If Pt^.ukz = nil    Then
    Begin
      Writeln('В списке <K элем.! ');
      Readkey;
      Exit;
    End;
       { i=k-1}
         P:=Pt^.ukz;
    Pt^.ukz:= Pt^.ukz^.ukz;
    Dispose(P);
  End;
```

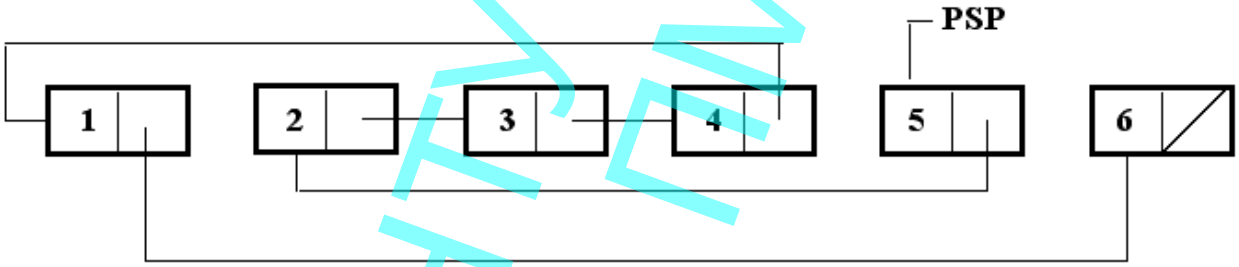
Пример 2

Написать программу, выполняющую следующую операцию обработки односвязного списка: поменять местами первый и предпоследний элементы списка (программа приведена в **Приложении Б**).

В **общем случае** для списка, состоящего из шести элементов,



результат обмена будет выглядеть так:



Особые случаи:

1) если список пуст (Psp=nil) или состоит из одного (Psp^.ukz=nil), двух элементов (Psp^.ukz^.ukz=nil), то поставленная задача не может быть выполнена;
2) если список состоит из трех элементов (Psp^.ukz^.ukz^.ukz=nil), то нужно поменять местами первый и второй элементы, то есть элементы, стоящие рядом.

Стек

Определение

Стек – линейная динамически изменяющаяся структура данных, предназначенная для добавления элементов к множеству и удалению их из него. Стек может удлиняться, укорачиваться и на время становиться пустым.

Над стеком определены следующие операции:

- занесение нового элемента;
- выборка и удаление элемента (одна операция).

Операции выполняются с одного и того же конца, называемого *вершиной стека*. Принцип выполнения операций - *LIFO*: ("последним пришел – первым обслуживаешься").

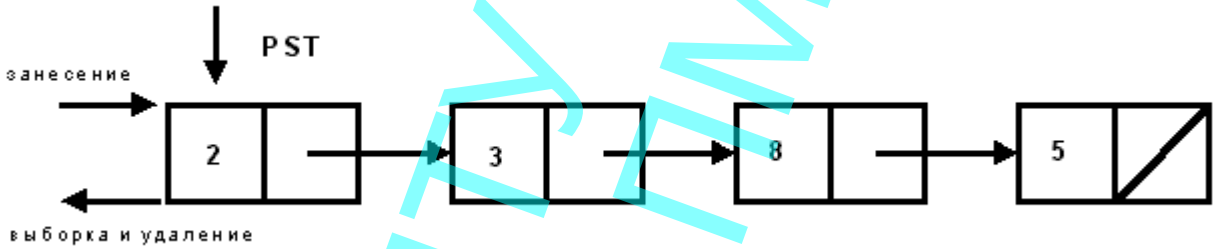
{ конец определения }

На *логическом* и *физическом* уровнях стек представляется так же, как и список, но операции над этими структурами выполняются по-разному.

При работе со стеком понятие ключа записи не используется, имеет значение только порядок занесения элемента в структуру (поэтому операция **поиска** для стека не определена).

Стеки используются обычно не для представления исходных данных, а при реализации различных алгоритмов, например, при разработке компиляторов и операционных систем.

Пусть, например, в стек последовательно поступают такие элементы: 5, 8, 3, 2. Получим следующее состояние стека:



На рис. 9 приведена процедура *ZSTEK*, реализующая **занесение элемента в стек.**

Формальные параметры процедуры:
PST – указатель на вершину стека (указатель стека);
VKL – указатель на включаемый элемент (элемент уже подготовлен).

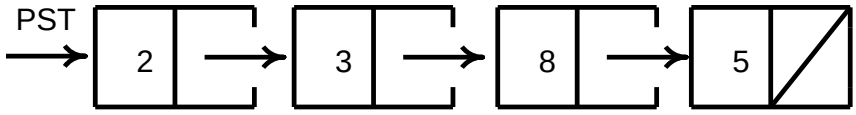
```
Procedure ZSTEK (Var PST, VKL: U);
Begin
    VKL^.Ukz:=PST;
    PST:=VKL
End;
```

Рисунок 9 - Процедура занесения элемента в стек

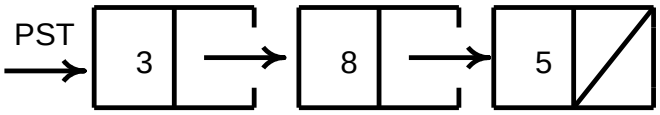
На рис. 10 приведена процедура *VSTEK*, реализующая **выборку и удаление элемента из стека.**

Формальные параметры процедуры:
PST – указатель стека;
ELEM – возвращает информационную часть удаляемого элемента (если стек пуст, то из процедуры возвращается *ELEM* =0).

Например, если перед удалением элемента состояние стека было таким:



то после удаления содержимое стека будет иметь вид:



(выполнилась выборка и удаление элемента 2).

```
Procedure VSTEK (Var PST: U; Var ELEM: integer);
Var P: U;
Begin
 If PST<>nil Then
  Begin
     P:=PST;
     ELEM:=PST^.Inf;
     PST:=PST^.Ukz;
     Dispose(P);
   End
  Else
     ELEM:=0;
End;
```

Рисунок 10 - Процедура выборки и удаления элемента из стека

Очередь

Определение

Очередь – линейная динамически изменяющаяся структура данных, предназначенная для добавления элементов к множеству и удалению их из него. Очередь может удлиняться, укорачиваться и на время становиться пустой.

Над очередью определены следующие операции:

- занесение нового элемента;
- выборка и удаление элемента (одна операция).

При работе с очередью используются два ее конца:

- “голова” (начало очереди), *PN* – указатель на "голову";
- “хвост” (конец очереди), *PK* – указатель на "хвост".

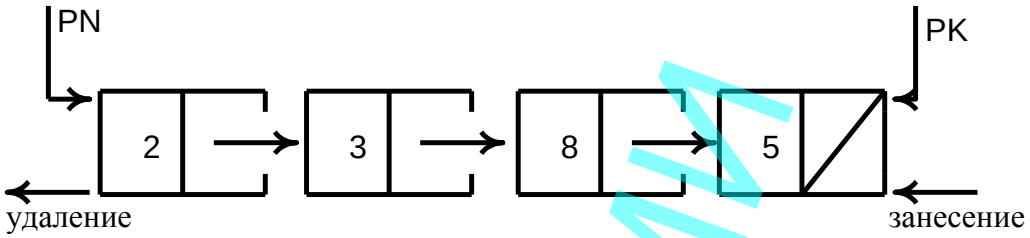
Занесение элемента в очередь осуществляется с “хвоста”, выборка и удаление – с “головы” (по принципу *FIFO*: “первым пришел – первым обслуживаешься”).

{ конец определения }

При работе с очередью понятие ключа записи не используется, имеет значение только порядок занесения элемента в структуру. Поэтому операция **поиска** для очереди не определена.

На *логическом* и *физическом* уровнях очередь представляется так же, как и список, но операции над этими структурами выполняются по-разному.

Пусть, например, в очередь последовательно поступают такие элементы: 2, 3, 8, 5. Получим следующее состояние очереди:



На рис. 11 приведена процедура занесения элемента в очередь, а на рис. 12 - выборки и удаления из очереди.

```
Procedure ZOCH(Var PN, PK,
                        VKL : U);
Begin
  If PK=nil Then
  Begin
    PK:=VKL;
    PN:=VKL;
  End
  Else
  Begin
     PK^.Ukz:=VKL;
     PK:=VKL;
  End;
End;
```

Рисунок 11 - Процедура занесения элемента в очередь

```
Procedure VOCH (Var PN, PK: U;
                    Var ELEM: integer);
Var P: U;
Begin
  If PN<>nil Then
  Begin
     P:=PN;
     ELEM:=PN^.Inf;
     PN:=PN^.Ukz;
     Dispose(P);
     If  PN=nil Then
        PK:=nil
  End
  Else
    ELEM:=0;
End;
```

Рисунок 12 - Процедура выборки и удаления элемента из очереди

Одной из областей, в которой применяются очереди, является моделирование. Термин "моделирование" обозначает экспериментирование с моделями систем для изучения их возможного поведения; при этом не требуется создавать реальные системы. Например, моделирование вычислительной системы с разделением времени, работы аэропорта, сети дорог, связывающих несколько городов, транспортного движения. Но особенно широко очереди используются при разработке операционных систем.

ЛАБОРАТОРНАЯ РАБОТА 3

НЕЛИНЕЙНЫЕ СТРУКТУРЫ

Цель работы: ознакомление с древовидными структурами данных и операциями над ними.

Задание к лабораторной работе

Написать на языке Паскаль программу создания и обработки **упорядоченного бинарного дерева.** Вариант задания выбрать в соответствии с номером по журналу.

Варианты операций обработки

1. Определить количество элементов с заданным значением ключа.
2. Проверить, является ли заданный элемент листом дерева.
3. Удалить заданный элемент, если он имеет одного сына.
4. Определить высоту дерева и сумму его элементов.
5. Найти всех потомков заданного узла.
6. Определить количество элементов в правом поддереве.
7. Определить среднее арифметическое элементов дерева.
8. Проверить, находится ли элемент **B** в однов из поддеревьев элемента **A.**
9. Определить глубину (номер уровня), на которой находится элемент с заданным значением ключа.
10. Определить количество и сумму листьев.
11. Удалить заданный элемент, если он является листом.
12. Найти отца для заданного элемента.
13. Определить количество элементов, имеющих двух сыновей.
14. Определить сумму элементов в правом поддереве заданного узла.
15. Определить количество элементов и высоту дерева.
16. Определить количество и сумму отрицательных элементов дерева.
17. Проверить, является ли элемент **A** отцом элемента **B.**
18. Определить количество элементов, имеющих одного сына.
19. Определить количество элементов, превышающих заданное значение.
20. Определить количество и сумму положительных элементов дерева.
21. Проверить, является ли элемент **A** сыном элемента **B.**
22. Определить количество и сумму элементов левого поддерева.
23. Удалить из дерева заданный элемент.
24. Определить сумму элементов, значения которых меньше заданного.
25. Определить количество элементов в левом поддереве заданного узла.
26. Определить количество элементов дерева, ключевое поле которых содержит **0.**

Требования к выполнению лабораторной работы

1. Алгоритмы операций над деревом необходимо оформить в виде процедур (функций) с соответствующими формальными параметрами.

2. Элементы дерева вводить в интерактивном режиме.
3. На экран выводить:
   - заданное дерево;
   - результаты выполнения операций.

В **Приложении В** приведен пример программы, выполняющей операции над деревом.

Контрольные вопросы

1. **Определения**: дерево, корень, листья, поддерево, ветвь дерева, бинарное дерево, упорядоченное бинарное дерево.
2. Системные процедуры создания и удаления элемента дерева.
3. Основные операции над деревом; их выполнение.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ РАБОТЫ

Определение дерева

Списки выражают линейные (одномерные) отношения. Но во многих приложениях отношения между данными носят более сложный, нелинейный характер. Наиболее важным типом нелинейных структур данных являются *деревья*. Чаще всего *деревья* используются в нечисленных алгоритмах (компиляторы, диалоговые и операционные системы, …).

Определение:

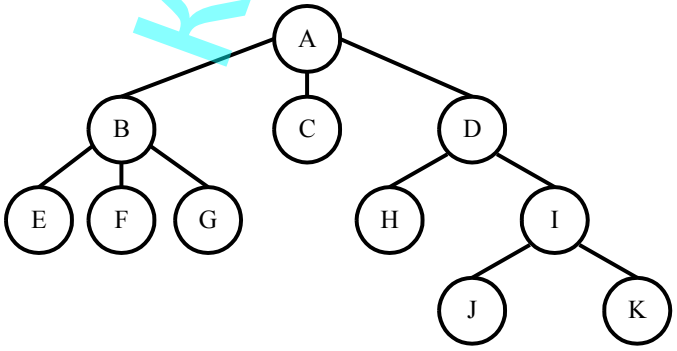
Дерево-это структура, состоящая из множества *вершин* (*элементов, узлов*), не имеющая контуров и петель. Вершины дерева организованы по уровням. Вершина самого верхнего уровня называется *корнем* дерева, вершины самого нижнего уровня – *листьями*.

Между вершинами дерева имеет место отношение “исходный – порожденный” (“отец – сын”), которое действует только в одном направлении – сверху вниз (от корня к листьям).

{конец определения}

На логическом уровне принято изображать дерево с корневой вершиной вверху:

A – корень дерева;



C, E, F, G, H, J, K -листья.

Для узла **D**: **A** – "отец" ("исходный"), **H** и **I** - "сыновья" ("порожденные"). В свою очередь узел **I** является "отцом" для **J** и **K**.

Корень не имеет "отца", листья - "сыновей".

Степень вершины – определяется числом порожденных узлов.
Степень дерева - определяется максимальной степенью вершины.
Высота (глубина) дерева – максимальный уровень его вершин.

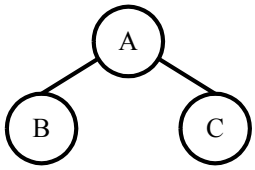
Для дерева, приведенного на рисунке: *степень* равна 3, *высота* равна 4.

Бинарные деревья

Наибольший интерес при решении практических задач представляют *бинарные (двоичные)* деревья, которые определяются следующим условием:

любой узел, кроме листа, может иметь не более двух порожденных узлов.

Для бинарного дерева (**БД**) используются понятия *левого поддерева* (*левой ветви*) и *правого поддерева* (*правой ветви*). При отображении **БД** в оперативной памяти удобно использовать связное размещение узлов, при этом каждый узел имеет следующую структуру:



(структура узла **A**)

указатель на **B**	**A**	указатель на **C**

B – левый порожденный узел,
C – правый порожденный узел.

Связное представление структуры узла:

Ukzl	*Inf*	*Ukzp*

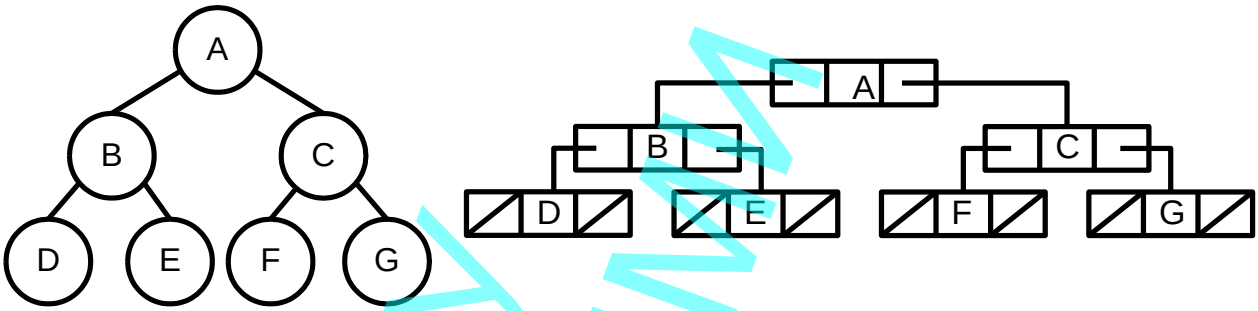
Inf - информационное поле или поля, среди которых одно является ключевым (в дальнейшем берем только одно информационное поле, которое считаем ключевым);

Ukzl - содержит указатель на *левое поддерево;*
Ukzp - содержит указатель на *правое поддерево*.

Левое поддерево (*левая ветвь*) элемента **A** - совокупность всех элементов дерева, к которым имеется путь от **A** через его *левый* указатель.

Правое поддерево (*правая ветвь*) элемента **A** - совокупность всех элементов дерева, к которым имеется путь от **A** через его *правый* указатель.

Пример **БД** дерева и соответствующей ему связной структуры на *логическом уровне*:



Для представления *физического уровня* нужно привести описание элемента дерева. Например,

```
Type u = ^zap;
     zap = Record
             ukzl : u;
              inf : integer;
              ukzp : u
           end;
```


Упорядоченное бинарное дерево

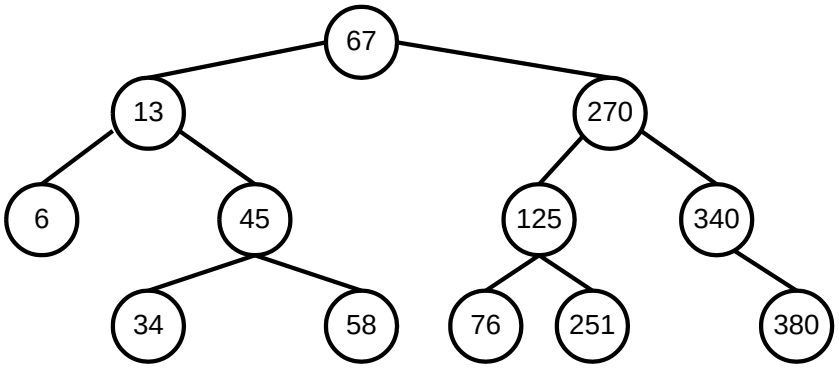
Для поиска записей по заданному ключу можно представить данные не в виде таблицы (последовательное представление), а в виде *упорядоченного бинарного дерева* - ***УБД*** (связное представление).

УБД определяется так:

для каждого узла все ключи в левой ветви меньше, а в правой ветви больше или равны значению ключа данного узла.

Основными операциями, выполняемыми над деревьями, являются:

- **включение** элемента в **УБД**;
- **поиск** в **УБД** элемента по заданному значению ключа;
- **удаление** элемента из **УБД** по заданному значению ключа;
- **обход БД**.



1) Включение

Задача поиска записей по заданному ключу может решаться более эффективно, если записи организовать не в виде таблицы (последовательное представление), а в **УБД** (связное представление).

Построение **УБД** осуществляется по следующему алгоритму (для простоты изложения совместим понятия информационной части узла и его ключевого поля).

Ключ, поступивший первым, располагается в корне дерева (пусть это будет a_1). Если a_2 (ключ второго узла) больше или равен a_1, то a_2 помещается в правую ветвь узла a_1, в противном случае – в левую ветвь:

Выбор места для a_i производится следующим образом:

- если $a_i < a_1$ – выполняется переход по левой ветви корня;
- если $a_i >= a_1$ – переход по правой ветви корня;
- если $a_i < a_j$ – переход по левой ветви узла a_j;
- если $a_i >= a_j$ – переход по правой ветви узла a_j.

Переходы по ветвям (спуск по уровням дерева) выполняются до тех пор, пока не будет найден соответствующий пустой указатель.

Например, если ключи поступают в последовательности:

67, 13 , 45, 270, 34, 58, 6, 340, 125, 76, 251, 380,

то упорядоченное двоичное дерево будет иметь вид:

Описание элемента дерева на языке Паскаль имеет вид:

```
Type u = ^zap;
     zap = Record
             ukzl : u;
              inf : integer;
              ukzp : u
           end;
```

Процедура включения элемента в **УБД** приведена на рис. 13. Формальные параметры этой процедуры:

PDR – указатель на корень дерева;

UVKL – указатель включаемого элемента (элемент подготовлен):


```
Procedure VKL (Var PDR,UVKL : U);
Var
    PT : U;
Begin
 If  PDR=nil Then   {включаемый элемент становится первым элементом}
  Begin
    PDR:=UVKL;
    exit;
  End;
  PT:=PDR;
 While True Do
    If UVKL^.INF>=PT^.INF Then
     { переход по правой ветви }
      If PT^.UKZP=NIL Then
        Begin
          PT^.UKZP:=UVKL;
          Exit;
        End
      Else
        PT:=PT^.UKZP
    Else
     { переход по левой ветви }
     If PT^.UKZL=NIL Then
       Begin
         PT^.UKZL:=UVKL;
         Exit;
       End
     Else
      PT:=PT^.UKZL
End;
```

Рисунок 13 - Процедура включения элемента в УБД

2) Поиск

Алгоритм поиска элемента дерева с заданным ключом аналогичен алгоритму включения элемента. Поиск считается неудачным, если при достижении листьев совпадение не обнаруживается (то есть элемент с заданным ключом отсутствует).

Процедура поиска приведена на рис. 14. Ее формальные параметры:

PDR – указатель корня дерева;

ELEM- указатель (адрес) найденного элемента; если поиск неуспешен, то *ELEM* возвращается равным *NIL*;
KL – ключ поиска.

```
Proceduse Poisk (Var PDR, ELEM:U; KL:integer);
Var PT : U;
Begin
   PT:=PDR;
  ELEM:=Nil;  {если элемент не найден}
  While PT <> nil do
     If KL=PT^.INF Then        { элемент найден }
      Begin
        ELEM:=PT;
        Exit;
       End
     Else
      If KL >PT^.INF Then
         PT:=PT^.UKZP   { переход по правой ветви }
      Else
         PT:=PT^.UKZL ;  { переход по левой ветви }
End;
```

Рисунок 14 - Процедура поиска элемента в УБД

Средняя длина поиска $C_{ср}$ зависит от структуры дерева. Если дерево сбалансировано (то есть **все уровни заполнены, кроме, может быть, последнего**), то $C_{ср} = log_2(n+1) - 2$ (*n*- число внутренних узлов).

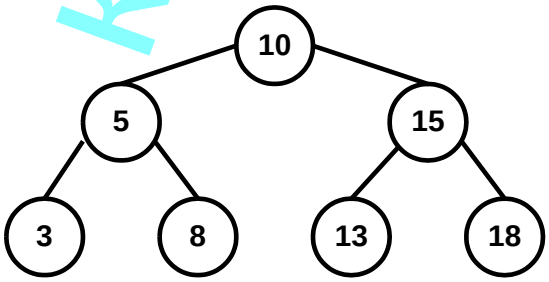
Для несбалансированного дерева (данные включаются в случайном порядке) $C_{ср} = 1{,}44*log_2(n)$; для больших *n*: $C_{ср} = 2*log_2(n)$

3) Удаление

При удалении элемента из дерева возможны три случая:

а) Удаляемый элемент - лист.

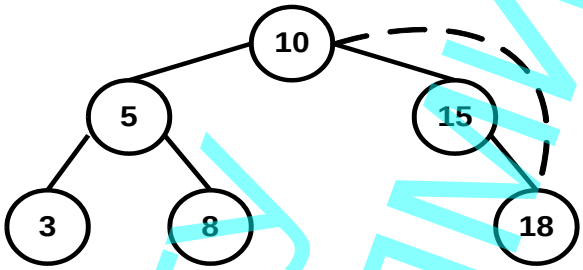
Например:



(удаляемые элементы: 3, 8, 13, 18)

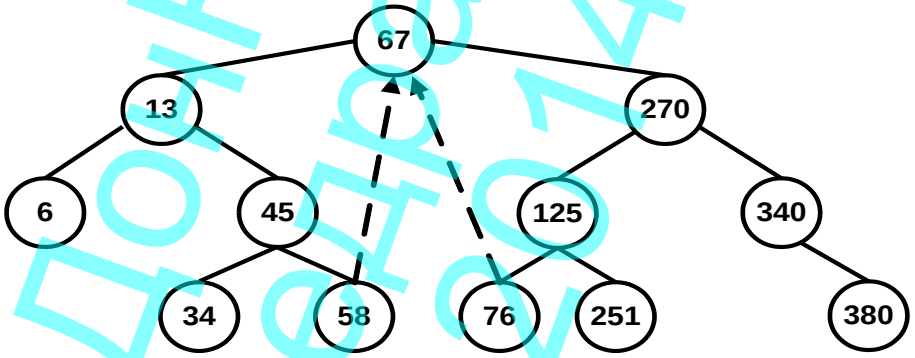
Алгоритм удаления основывается на алгоритме поиска, но должны использоваться два указателя: *Pt* – указатель текущего элемента; *Ptp* – указатель предыдущего элемента. При нахождении листа, подлежащего удалению (указатель *Pt*), левому или правому указателю его "отца" нужно присвоить *nil,* а память листа – освободить (Dispose(*Pt*)).

б) Удаляемый элемент имеет одного "сына".
Например, удалить элемент **15:**



(удаление выполняется так же, как и удаление элемента списка).

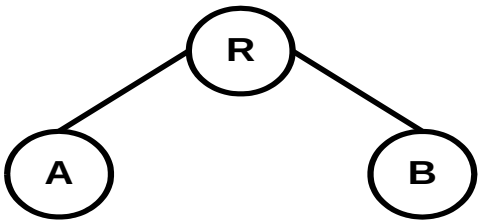
в) Удаляемый элемент имеет двух "сыновей".
Например, удаляемый элемент **67**:



Удаляемый элемент должен быть заменен наиболее близким по значению элементом (чтобы упорядоченность дерева не нарушилась). Это будет самый правый по левой ветви или самый левый по правой ветви элемент.

4) Обход

При работе с древовидными структурами часто возникает задача: выполнить некоторую операцию ***P*** над каждым элементом дерева (например, распечатать информационную часть). Для решения этой задачи нужно "посетить" каждую вершину, то есть выполнить обход дерева.



Существует три метода обхода дерева:

1. *Сверху вниз*: **R, A, B** (порядок обхода вершин).
2. *Слева направо*: **A, R, B**.
3. *Снизу вверх*: **A, B, R**.

Эти методы удобно реализовать с помощью рекурсивных процедур, так как дерево является рекурсивной структурой данных.

Рекурсивное определение дерева: дерево с базовым типом ***T***- это либо пустое дерево, либо некоторый элемент типа ***T*** с конечным числом связанных с ним отдельных деревьев с базовым типом ***T***, называемых поддеревьями.

Рекурсивные процедуры **обхода дерева** приведены на рис. 15, 16, 17. В качестве формального параметра берется ***T***- указатель (адрес) текущего элемента дерева (передается по значению). При обращении к процедуре обхода указывается адрес корня дерева: *D1 (Pdr), D2 (Pdr), D3 (Pdr).*

В качестве примера рассмотрим дерево, приведенное в пункте "Удаление" (в).

Последовательность вывода ключей при обходе этого дерева имеет вид:

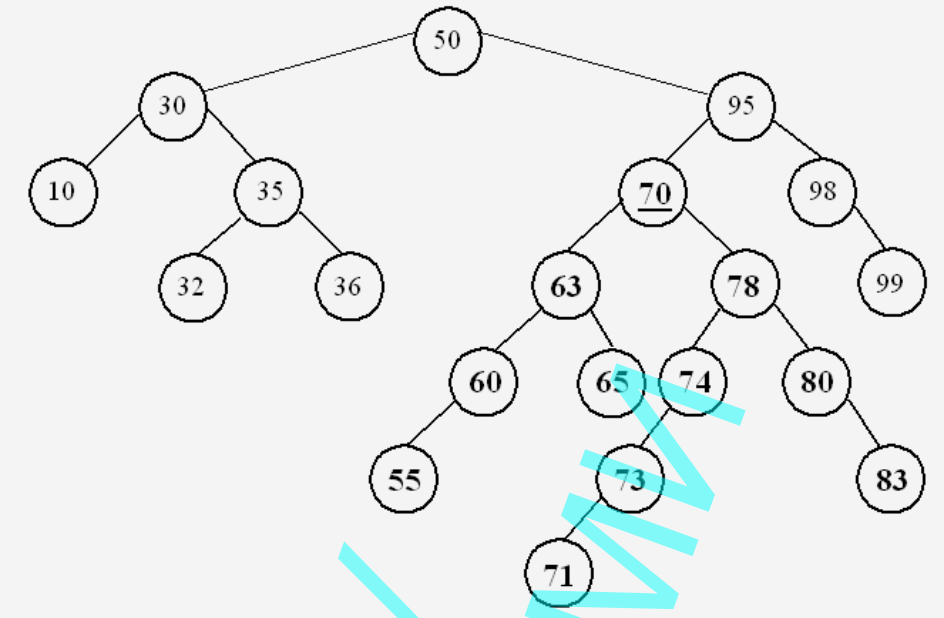
Сверху вниз	Слева направо	Снизу вверх
67, 13, 6; 45, 34; 58; 270, 125, 76; 251; 340, 380;	6, 13; 34, 45, 58; 67; 76, 125, 251; 270; 340, 380;	6; 34, 58, 45, 13; 76, 251, 125; 380, 340, 270; 67

Пример

В **УБД** найти *поддерево* с заданным корнем *kl*, определить его высоту и количество элементов.

В качестве примера рассмотрит дерево, в которое заносятся элементы

50, 30, 10, 95, 70, 98, 63, 35, 60, 55, 99, 65. 78, 80, 83, 36, 32, 74, 73, 71



В результате выполнения операций для *kl=**70*** получим:

- высота поддерева равна ***5***,
- количество элементов равно ***11***.

```
Procedure D1 (T: U);
{ обход дерева сверху вниз }
Begin
    If T<>Nil Then
    Begin
       Writeln(T^.INF);
       D1 (T^.UKZL);
       D1 (T^.UKZP);
     End;
End;
```

Рисунок 15 - Процедура обхода двоичного дерева сверху вниз

```
Procedure D2 (T: U);
{ обход дерева слева направо }
Begin
    If T<>Nil Then
    Begin
        D2 (T^.UKZL);
       Writeln(T^.INF);
       D2 (T^.UKZP);
    End;
End;
```

Рисунок 16 - Процедура обхода двоичного дерева слева направо

```
Procedure D3 (T: U);
{ обход дерева снизу вверх }
Begin
    If T<>Nil Then
    Begin
        D3 (T^.UKZL);
        D3 (T^.UKZP);
        Writeln(T^.INF);
    End;
End;
```

Рисунок 17 - Процедура обхода двоичного дерева снизу вверх

ЛИТЕРАТУРА

1. Вирт Н. Алгоритмы+структуры данных=программы: Пер. с англ. -М.:Мир, 1985.-406 с.
2. Вирт Н. Алгоритмы и структуры данных: Пер.с англ. – 2-е изд., испр.- СПб.:Невский Диалект, 1989.-360с., 2005.-352 с.: ил.
3. Вирт Н. Алгоритмы и структуры данных: Пер.с англ. – 2-е изд., испр.- СПб.:Невский Диалект, 2005.-352 с.
4. Кнут Д. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск.- М.: Мир,1978.-843 с.
5. Лорин Г. Сортировка и системы сортировки, Пер. с англ.-М.: Наука, 1983. -384 с.
6. Трамбле Ж., Соренсон П. Введение в структуры данных.-М.: Машиностроение, 1982. -832 с.
7. Альфред Ахо, Джон Хопкрофт, Джеффри Ульман. Структуры данных и алгоритмы: Пер. с англ. -М.: Уч. пос. –М., Издательский дом “Вильям”,2000.-384с., 2006. -400с. :ил.
8. Берж К. Теория графов и ее применения. – М.: ИЛ, 1962. – 319с.
9. Харари Ф. , Теория графов. -М., Мир, 1973. -300с.
10. Оре О. Теория графов. – М.: Наука, 1980. –336с.
11. Грин Д., Кнут Д. Математические методы анализа алгоритмов. –М., Мир, 1987.
12. Зуев Е.А. Язык программирования TurboPascal 6.0.-М.: Унитех, 1992 - 298 с., ил.-(Мир TurboPascal. Вып. 1)
13. Епанешников А., Епанешников В. Программирование в среде TurboPascal 7.0. – М.: ДИАЛОГ-МИФИ, 1993. – 288с.
14. Поляков Д.Б., Круглов И.Ю. Программирование в среде ТурбоПаскаль: Справочно – методическое пособие. – М.: Изд-во МАИ, 1

Приложение А

Пример программы, выполняющей сортировку таблицы методом линейного выбора

```
Program Lin_Sort;
{сортировка таблицы методом линейного выбора}
Uses Crt;
Const  Nmax=100;  {максимальное количество записей в таблице}
Type
      Rec=Record                { тип записи }
               kl : integer;        { ключ записи }
               inf : string [10];  {информационное поле }
             End;
      Table=Array [1..Nmax] Of Rec;  { тип таблицы }

Var T: Table;          {сортируемая таблица}
   n: integer;         {количество записей в таблице}
  ch: byte;

Procedure InTab (Var T: Table; Var n: integer);
 {ввод записей таблицы с клавиатуры}
Var     ch: char;
     fname: string;
         i: integer;
         f: file of rec;
       cod: byte;
Begin
   ClrScr;
   n:=0;
   Repeat
     n:=n+1;
     WriteLn ('Введите ',n,'-ю запись таблицы');
     Writeln ('Ключ:');
     ReadLn (t[n].kl);
     WriteLn ('Информация: ');
     Readln (t[n].inf);
     WriteLn ('Продолжить ввод? y/n');
     ch:=ReadKey;
  until ch in ['n','N','т','Т'];
```

```
WriteLn ('Сохранить введенные данные в файле? y/n');
 ch:=ReadKey;
 if ch in ['n', 'N', 'т', 'Т'] then
    exit;
 WriteLn ('Введите имя файла');
 ReadLn (fname);
 Assign (f, fname);
 {$I-}              { отключение контроля ошибок ввода/вывода }
 Rewrite (f);
 {$I+}              { включение контроля ошибок ввода/вывода }
 cod:=IOResult; {IOResult – сист. функция, определяет код заверш.}
 If cod=0 then
   begin            {занесение таблицы в файл}
      for i:=1 to n do
         Write (f, t[i]);
       close(f);
   end
  else
    begin
       WriteLn ('Ошибка при создании файла');
       ReadKey;
    end;
End;

Procedure InFileTab (Var T: Table; Var n: integer);
 {чтение записей  из файла в таблицу}
Var  cod: byte;
          f: file of rec;
    fname: string;
         ch: char;
Begin
 ClrScr;
 Repeat
    WriteLn ('Введите имя файла');
    ReadLn (fname);
    Assign (f, fname);
      {$I-}
    Reset (f);
     {$I+}
```

```
  cod:=IOResult;
    If cod <>0 then  { файл не найден }
       begin
         WriteLn ('Не могу найти файл  ', fname);
         WriteLn  ('Продолжить работу? y/n');
         ch:=ReadKey;
         If ch in ['n', 'N', 'т', 'Т'] then
            exit;      {выход из процедуры}
       end;
  until cod=0;
  n:=0;
  While not eof(f)  do
      begin
        n:=n+1;
        Read (f, T[n]);
     end;
  Close(f);
  WriteLn ('Таблица загружена из файла ', fname);
  Write ('Нажмите любую клавишу... ');
  ReadKey;
End;

Procedure ShowTab (Var T: Table; n: integer);
 {печать таблицы}
Var i:integer;
Begin
    ClrScr;
    WriteLn ('      Записи таблицы:');
    WriteLn ('-----------------------------');
    WriteLn ('¦    Ключ   ¦ Информация ¦');
    WriteLn ('-----------------------------');
    For i:=1 to n do
       WriteLn ('¦     ', T[i].kl, ' ¦       ', T[i].inf' ¦');
   WriteLn ('-----------------------------');
   WriteLn ('Нажмите любую клавишу ...');
   ReadKey;
End;
```

```
Procedure PrintTab (Var T:Table; n, i, imin:integer);
 {промежуточная печать ключей для  демонстрации метода    сортировки}
Var k:integer;
Begin
   For k:=1 to n do
      begin
        If k=i then       { i - позиция переставляемой записи }
          textattr:=red*16+white       {белые на красном фоне}
        else
          If k=imin then   { imin - позиция записи с миним.ключом}
            textattr:=blue*16+white       {белые на синем фоне}
          else
            textattr:=black*16+white;     {белые на черном фоне}
   Write (t[k].kl);
   textattr:=black*16+white;
   Write ('  ');
  end;
  WriteLn;
End;

Procedure SortTab (var T:Table; n:integer);
 { сортировка таблицы методом линейного выбора}
Var i, imin, j : integer;
            tmp : rec;
Begin
    ClrScr;
    WriteLn ('Шаги сортировки: ');
    For i:=1 to n-1 do
       begin
          imin:=i;        {определение индекса мин. элемента}
          For j:=i+1 to n do
             If t[j].kl < t[imin].kl then
                imin:=j;
          If imin <> i then
             begin
               PrintTab (T, n, i, imin);   {промежут. печать ключей}
               ReadKey;
               { перестановка записей}
               tmp:= t[i];
```

```
                t[i]:= t[imin];
                t[imin]:= tmp;
              end;
        end;
End;
Begin
  n:=0;
  Repeat
    ClrScr;
    WriteLn ('1. Ввод данных с клавиатуры');
    WriteLn ('2. Загрузка таблицы из файла');
    WriteLn ('3. Вывод таблицы');
    WriteLn ('4. Сортировка таблицы линейным  выбором');
    WriteLn ('-------------------------------------------------');
    WriteLn ('0. Выход');
    WriteLn;
    Write ('Ваш выбор: ');
    ReadLn(ch);
    Case ch of
     1: InTab (T,n);
     2: InFileTab (T,n);
     3: If  n<>0 then
          ShowTab (T,n)
        else
          begin
            WriteLn ('Таблица не создана');
             ReadKey;
          end;
     4: if n<>0 then
          SortTab(T,n)
        else
          begin
            Writeln('Таблица не создана');
            ReadKey;
          end;
    end;
  until ch=0;
End.
```

Приложение Б
Пример программы обработки списка

```
Program List;
 {обмен местами первого и предпоследнего элементов списка}
Uses crt;
Type  u =^zap;
      zap = record      {структура элемента списка}
               inf : integer;
               ukz : u;
             end;
Var PSP : u;            {указатель списка}
        ch : char;

Procedure Create_List (Var PSP:u);
 {создание неупорядоченного списка}
Var  q,              {указатель на предыдущий элемент списка}
       p : u         { указатель на текущий  элемент списка }
       ch : char;
Begin
       New(q);
       WriteLn ('Введите информационное поле элемента:');
       ReadLn (q^.inf);
       PSP:=q;
       WriteLn ('Продолжить ввод? y/n');
       ch:=ReadKey;
       While ch In ['y','Y',’н’,’Н’] do
           Begin
              New(p);
              WriteLn ('Введите информационное поле элемента');
              ReadLn (p^.inf);
              q^.ukz:=p;
              q:=p;
              WriteLn ('Продолжить ввод?  y/n');
              ch:=ReadKey;
            End;
      q^.ukz:=nil;
End;
```

```
Procedure Print_list(PSP:u);
      { печать элементов списка, PSP - указатель списка  }
Var t:u;
Begin
  If PSP=nil then
    begin
      WriteLn ('Список пуст');
       ReadKey;
       Exit;
    end;
  Writeln ('  Список:');
  t:=PSP;
  While t <> nil do
    begin
       Write (t^.inf,' --> ');
        t:=t^.ukz;
    end;
  WriteLn ('NIL');
  WriteLn ('Нажмите любую клавишу');
  Readkey;
End;

Procedure Destroy_list(Var PSP:u);
 { освобождение памяти, занятой элементами списка}
Var t:u;
Begin
   While PSP <> nil do
      begin
         t:=PSP;
         PSP:=PSP^.ukz;
         Dispose(t);   {освобождение памяти}
      end;
End;

Procedure Change_node (Var PSP:u);
{обмен первого и предпоследнего элементов списка}
Var q, s, t : u;
Begin
   If (PSP=nil) or (PSP^.ukz=nil) or (PSP^.ukz^.ukz=nil)  then
```

```
    begin
       WriteLn ('В списке меньше трех элементов');
       Print_list(PSP);
       exit;
    end;
       {поиск предпоследнего элемента}
   q:=PSP;
   s:=PSP^.ukz;   {s - указатель на предпоследний элемент}
   While s^.ukz^.ukz<>nil do
       begin
          q:=s;
          s:=s^.ukz;
       end;
    If q=PSP then    {особый случай - обмен 1-го и 2-го элементов}
      begin
          q^.ukz:=s^.ukz;
          s^.ukz:=q;
          PSP:=s;
          WriteLn ('Обмен 1-го и 2-го элементов выполнен');
          Readkey;
          Exit
      end;
     {общий случай}
    t:=PSP^.ukz;
    PSP^.ukz:=s^.ukz;
    s^.ukz:=t;
    q^.ukz:=PSP;
    PSP:=s;
    WriteLn ('Обмен элементов выполнен');
    Readkey;
End;


Begin
   PSP:=nil;
   Repeat
    ClrScr;
```

```
      WriteLn ('1 - Создать список');
      WriteLn ('2 - Просмотреть список');
      WriteLn ('3 - Поменять местами первый и предпоследний
                        элементы ');
      WriteLn ('4 - Удалить список');
      WriteLn ('-------------------------------------------------------');
      WriteLn ('0 - Выход');
      WriteLn;
      Write ('Ваш выбор:');
      ch:=readkey;
      WriteLn (ch);
      WriteLn;
      Case ch of
        '1' : begin
                PSP:=nil;
                Create_list(PSP);
              end;
        '2' : Print_list(PSP);
        '3 ': Change_node(PSP);
        '4 ': begin
                Destroy_list(PSP);
                Writeln ('Список удален!');
                Readkey;
              end;
      end; {case}
    until ch='0';
End.
```

Приложение В

Пример программы, реализующей операции с деревом

```
Program Derevo;
 {определение высоты и кол-ва элементов заданного поддерева}
Uses Crt;
Type  u=^zap;                 {тип элемента дерева}
        zap=record
              ukzl : u;       {указатель на левую ветвь}
              inf : integer;  {информационное поле}
              ukzp : u        {указатель на правую ветвь}
             end;
Var  Pdr,                     {указатель на корень дерева}
      Poddr : u;              {указатель на корень поддерева}
      ch:byte;
      k,                      {количество элементов поддерева}
      v,                      {высота искомого поддерева}
      kl : integer;           {ключ поиска}

Procedure Vkl (Var Pdr, uvkl:u);
  {включение элемента в упорядоченное бинарное дерево;
   uvkl -  указатель на включаемый элемент, элемент подготовлен}
Var pt : u;      {текущий указатель}
Begin
    If Pdr=nil then
      begin
        Pdr:=uvkl;
        exit;
      end;
    pt:=Pdr;
  While  true  do
     If uvkl^.inf >=pt^.inf then
        If pt^.ukzp=nil then
           begin
             pt^.ukzp:=uvkl;
            exit;
          end
        else
           pt:=pt^.ukzp
```

```
    else
      if pt^.ukzl=nil then
        begin
          pt^.ukzl:=uvkl;
          exit;
        end
      else
        pt:=pt^.ukzl;
  End;

Procedure Create_der (Var Pdr :u);
 {создание упорядоченного двоичного дерева}
Var   q: u;      {текущий указатель}
      ch: char;
Begin
     Repeat
          New(q);
          WriteLn(' Введите информационное поле элемента ');
          ReadLn (q^.inf);
          q^.ukzl:=nil;
          q^.ukzp:=nil;
          Vkl(pdr,q);   {включение элемента в дерево}
          WriteLn ('Продолжить ввод?  y/n');
          ch:=ReadKey;
     until ch='n';
End;

Procedure Obhod (t:u);
 {обход сверху вниз – подсчет количества элементов поддерева;
  t -  указатель текущего элемента дерева}
Begin
  If  t <>  nil then
     begin
       Write (t^.inf, ' ');
        Readkey;
        k:=k+1;
        Obhod (t^.ukzl);
        Obhod (t^.ukzp);
```

```
      end;
End;

Procedure Poisk (Var Pdr, Poddr :u; kl:integer);
 {поиск поддерева с заданным корнем}
Var pt:u;     {указатель текущего элемента дерева}
Begin
   pt:=pdr;
   Poddr:=nil;
   While pt <> nil do
      If kl=pt^.inf then
         begin
            Poddr:=pt;     {поддерево с заданным корнем  найдено}
            exit;
         end
      else
         If kl>pt^.inf then
            pt:=pt^.ukzp
         else
           pt:=pt^.ukzl;
End;

Function Visota (t:u) : integer;
 {обход снизу вверх – определение высоты поддерева;
  t -  указатель текущего элемента дерева}
Var v1, v2: integer;
Begin
   If t=nil then
      visota:=0
  else
      begin
         v1:=Visota(t^.ukzl);
         v2:=Visota(t^.ukzp);
         If v1>v2 then
            Visota:=v1+1
         else
            Visota:=v2+1
      end;
End;
```

```
Begin
   Pdr:=nil;
   Repeat
        ClrScr;
        WriteLn ('1 - Создать дерево');
        WriteLn ('2 - Просмотреть дерево');
        WriteLn ('3 - Найти поддерево');
        WriteLn ('4 - Удалить дерево');
        WriteLn ('-----------------------------');
        WriteLn ('0 - Выход');
        WriteLn;
        Write ('Ваш выбор:');
        Read (ch);
        WriteLn;
        Case ch of
          1: begin
               Pdr:=nil;
               Create_der(Pdr);
             end;
          2: begin
               k:=0;
               WriteLn;
               Obhod (Pdr);
               WriteLn;
               WriteLn ('Количество элементов в дереве = ', k);
               Readkey;
             end;
         3: begin
              WriteLn ('Введите корень поддерева: ');
              ReadLn (kl);
              Poisk (Pdr, Poddr, kl);
              If Poddr <> nil then
                 begin
                    Obhod (Poddr);
                    WriteLn;
                    WriteLn('Количество элементов в поддереве = ', k);
                    Readkey;
                    v:=Visota (Poddr);
```

```
                    WriteLn;
                    WriteLn ('Высота поддерева = ',v);
                  end
                else
                  begin
                    WriteLn (‘Поддерево не найдено’);
                    Readkey;
                  end;

        end;
    until ch= 0;
End.
```

Приложение Д

КРАТКОЕ ОПИСАНИЕ СИСТЕМЫ TURBO PASCAL

1. СТРУКТУРА PASCAL-ПРОГРАММЫ

Язык Pascal был разработан Н. Виртом как инструмент для обучения программированию. Однако вскоре он показал себя весьма пригодным для разработки самых разнообразных приложений. Существуют многочисленные реализации языка практически для всех машинных архитектур. Наиболее известной реализацией языка Pascal является система программирования Turbo Pascal 6.0 (TP), разработанная фирмой Borland International для персональных компьютеров. Она широко используется для создания программ производственного и научного назначения, разработки компиляторов, баз данных и операционных систем.

Схематично структуру Pascal-программы можно представить так:

< Заголовок программы >
< USES – часть >
< Описание >
< Программный блок >

PROGRAM < имя программы > - заголовок программы, рассматривается TP как комментарий (то есть выполняет декоративные функции).

USES определяет отдельно транслируемые программные единицы (Unit), которые будут использоваться в данной программе. В TP существуют 6 стандартных Unit: System, Dos, Crt, Printer, Graph, Overlay.

System используется всегда и содержит процедуры и функции ввода-вывода, манипулирования строками, динамического управления памятью, арифметические и ряд других. System подключается автоматически, поэтому его можно не указывать в списке USES.

Dos содержит процедуры обращения к функциям MS-DOS.

Crt содержит переменные, процедуры и функции, используемые при управлении выводом информации на экран монитора: окна, цвет, звук, определение положения курсора, стирание информации и другие. Например,

- black, blue, green, cyan, red, magenta, brown, lightgray – константы для цвета букв и фона;
- textattr – системная переменная, определяющая текущий цвет букв и фона;
- ClrScr – системная процедура очистки экрана текущим цветом;
- Readkey- системная функция, запоминающая код нажатой клавиши и приостанавливающая выполнение программы до очередного нажатия клавиши;

- IOResult – системная функция, определяющая код ошибки ввода-вывода.

Overlay содержит константы и процедуры, используемые при построении программ с перекрытиями.

Printer содержит описание файла LST, который используется для вывода информации на принтер.

Graf включает в себя большое количество процедур для построения графических изображений.

< **Описание** > - каждая переменная, процедура или функция должна быть описана до своего первого применения. В блоке < Описание > содержится в общем случае пять секций:

LABEL – описание меток, посредством которых можно передавать управление из разных точек программы с помощью оператора безусловного перехода GOTO <метка>. Однако идеология создания структурированных программ не поддерживает безусловные переходы, поэтому их использование в программах не рекомендуется.

CONST- определение констант; позволяет указать идентификаторами некоторые значения. Идентификатор константы может использоваться в любых конструкциях соответствующего типа. В TP определены также так называемые типизированные константы, значения которых могут изменяться в программе.

TYPE – описание нестандартных типов переменных; позволяет программисту конструировать собственные типы, расширяя тем самым исходный набор типов языка Pascal применительно к потребностям решаемой задачи.

VAR – описание глобальных переменных, областью действия которых являются главная программа и все описанные в программе процедуры и функции (если в них эти переменные не описаны повторно).

PROCEDURE/FUNCTION – описания процедур и функций, представляющих собой самостоятельные и законченные фрагменты программы. Процедуры и функции являются основным средством структурирования программ.

< **Программный блок** > - операторная часть Pascal – программы (принято называть ее основной или главной программой). Операторы заключаются между ключевыми словами Begin … End, и после End ставится точка. Между собой операторы должны разделяться символом “;” :

```
Begin
   оператор1;
   оператор2;
      …
End.
```

Выполнение Pascal – программы начинается с основной программы.

Замечания

1. В TP соответствующие заглавные и строчные буквы в идентификаторах и служебных (ключевых) словах не различаются. Например, PROCEDURE, Procedure, procedure – это различные написания одного и того же ключевого слова.

2. Комментарии в TP заключаются в фигурные скобки '{' и '}' или в разделители вида '(*' и '*)'. Например,

```
{ короткий комментарий}
(* длинный комментарий,
   расположенный на
   нескольких строках  *)
```

Комментарий может находиться между любыми двумя лексическими единицами.

3. Длина программной строки ограничена 126 символами.

Пример **развернутой** структуры программы:

```
Program
Uses Crt;
Const
   Nmax = 100;  ESC = #27;
   Stroka = '----------------';
   Err = 'Нажмите любую клавишу';
   Pi : real=3.14;
   Max : integer=9999;
Type
   Rec=Record
         Name : string[15];
         Area : integer;
       end;
   Table=array [1..Nmax] of rec;
Var
    n: integer;
    T: Table;
    ch: char;
    rab: rec;
Procedure  InTab (Var T:Table; Var n: integer);
   Var i, j : integer;
    < операторы >
 End;
   . . .
Function hfunc (s: string): integer;
     <  операторы  >
```

End;
Begin
< операторы основной программы >
End.

2. ПРОСТЫЕ ТИПЫ

Язык Pascal является статическим языком. Это означает, что тип переменной определяется при ее описании и не может быть изменен. Тип переменной определяет множество допустимых для нее значений и, следовательно, виды операций, которые могут выполняться над этой переменной.

Базовыми в системе типов являются **простые** типы: скалярные (стандартные скалярные и перечислимые) и ограниченные. Стандартные скалярные делятся на четыре группы:

- целые типы (Byte, Integer, Shorting, Word, Longint),
- вещественные типы (Real – основной),
- символьный тип (Char),
- булевский тип (Boolean).

Целые типы

Byte – позволяет представить целые значения в диапазоне 0..255 (занимает 1 байт);
Shorting – в диапазоне -128..127 (1 байт);
Integer – в диапазоне -32768..32767 (2 байта);
Word – в диапазоне 0..65535 (2 байта);
Longint – для значений | x |<=2*(10^9) (4 байта).

Над целыми значениями определены операции ‘+’, ‘-‘, ‘*’ (дают целый результат), ‘/’ (вещественный результат), а также дополнительные операции деления:

Div – деление нацело (дробная часть отбрасывается),
Mod – остаток от целочисленного деления.

Основной вещественный тип

Real – служит для представления значений с плавающей запятой в интервале 2,9E-39<=|x|<=1,7E+38 (занимает 6 байт). Мантисса содержит до 11 значащих цифр. Для выполнения операций с переменными типа Real математический сопроцессор не требуется.

Над значениями вещественного типа допустимы операции ‘+’, ‘-‘, ‘*’, ‘/’. Все они дают вещественный результат, если хотя бы один операнд вещественный.

Символьный тип

Char – служит для представления символов ASCII – кода (американский стандартный код для обмена информацией). Это множество состоит из 256 различных символов, упорядоченных определенным образом (символы цифр, заглавных и строчных букв, специальные управляющие символы). Каждый символ имеет порядковый номер, который можно определить с помощью специальной функции. Значение переменной типа Char занимает 1 байт.

Для символьных переменных возможны только операции сравнения.

Булевский тип

Имеет два значения, обозначаемые ключевыми словами **true** (истина, “да”) и **false** (ложь, “нет”). Над булевскими переменными допустимы операции сравнения, причем считается, что false<true. Кроме того, определены четыре стандартные логические операции:

and – конъюнкция (логическое умножение, “и”),

or - дизъюнкция (логическое сложение, “или”),

xor – сложение по модулю 2 (исключающее “или”),

not – логическое отрицание (“не”).

Значения булевского типа занимают 1 байт.

Перечислимые типы

В TP имеется возможность определения нового скалярного типа путем явного перечисления всех возможных его значений, причем каждое такое значение будет определяться только именем.

Например,

```
TYPE
  Cvet  = (красный, желтый, зеленый);
  Days  = (понедельник, вторник, среда, четверг,  пятница, суббота,
           воскресенье);
```

Считается, что значения перечислимого типа указаны в списке в порядке возрастания.

Ограниченные типы

Можно образовать новый тип, указав допустимый диапазон значений некоторого стандартного скалярного типа. Например,

```
TYPE
    ind1 = 1..10;
    ind2 = -100..100;
    simv = ‘a’..’z’;
```

Значения переменных, описанных такими типами, не должны выходить за границы указанного диапазона.

Все скалярные типы, кроме вещественных, а также перечислимые типы называются дискретными.

Стандартные функции преобразования типов (типы аргумента и результата выполнения функции различны):

- odd (x)

x – целого типа; возвращает true, если аргумент нечетный, и false, если аргумент четный;

- trunc (x)

x – вещественного типа; результатом является целая часть аргумента (дробная часть отбрасывается); результат – целого типа;

- round (x)

x – вещественного типа; округляет значение аргумента до ближайшего целого (например, round(1.5) →2; round(1.45) →1; round(-1.5) →-2; round(-1.45) →-1);

- ord (x)

x – скалярного типа; возвращает порядковый номер (нумерация от 0) значения аргумента в данном типе (например, ord('a')→ 97, ord (‘d’)→100);

- chr (x)

x – целый тип (byte); возвращает ASCII-символ, соответствующий заданному номеру (например, chr(100)→’d’).

Описание переменных

Описание (определение, объявление) должно содержать имя переменной и ее тип, разделяемые двоеточием. Заканчивается описание символом ' ; '. Если нужно описать несколько переменных одного и того же типа, то перед двоеточием следует перечислить их имена.

Пример описаний:

```
VAR
    x, y, z : real;
    i, j : integer;
    fl : Boolean;
    ind1, ind2 : 1..100;
   cvet : (красный, желтый, зеленый);
```

3. СОСТАВНЫЕ ТИПЫ

На основе небольшого числа стандартных типов Pascal позволяет конструировать типы разной структуры и сложности, позволяя тем самым представлять данные для решения любой задачи.

В TP определены следующие составные типы:

- строковые,
- регулярные (массивы),
- комбинированные (записи),
- множественные,
- файловые,
- объектовые.

Строковые типы

Строковый тип данных определяет множество символьных цепочек произвольной длины: от 0 символов (пустая строка) до 255 (максимальная длина строки). При описании строкового типа нужно указать служебное слово string и за ним в квадратных скобках количество символов, то есть длину строки.

Например,

```
TYPE
   line : string [80];
VAR
    stroka : line;
    stroka1: string;
```

Переменная stroka может иметь в качестве своего значения любую последовательность символов длиной от 0 до 80. Например,

```
stroka :='Курсовая работа';
        . . .
stroka := 'работа';
        . . .
stroka :='';     {пустая строка}
```

Пример показывает, что строка может в текущий момент занимать лишь часть отведенной под нее памяти. Фактическая длина строки указывается в нулевом байте, поэтому каждая строка занимает на один байт больше, чем указано в описании.

Если при описании длина строки не указана, то предполагается максимально возможная длина – 255 символов.

Над строками определены операция конкатенация ('+') и обычные операции сравнения: =, <>, <, … Например,

```
stroka1 := 'Дипломная '+'работа';
```

(результат: 'Дипломная работа').

Смысл операций сравнения для строковых данных:

- более короткая строка всегда меньше длинной строки,
- если длины сравниваемых строк равны, то выполняется поэлементное сравнение символов этих строк с учетом лексикографической упорядоченности символов.

Если строковой переменной присваивается строка с большей, чем указано в описании длиной, то лишняя часть отбрасывается (при этом не происходит прерывание выполнения программы).

Системные функции строк:

- length (s)

s – строкового типа; определяет число символов в строке (длину строки); тип результата – integer;

- copy (s, pos, num)

s – string; pos, num – integer; результат – integer; из строки s, начиная с позиции pos, выделяется подстрока, длиной num символов;

- pos (str, s)

str, s - string; результат – byte; определяет позицию, с которой подстрока str первый раз входит в строку s (если подстрока не найдена, то функция возвращает 0).

Системные процедуры для строк:

- Delete (s, pos, num)

s – string; pos, num – integer; удаляет num символов из строки s, начиная с позиции pos;

- Insert (str, s pos)

str, s – string; pos – integer; вставляет строку str в строку s, начиная с позиции pos строки s;

- **Str (value, s)**

s – string; value – real или integer; преобразует числовое значение value в строку символов и запоминает результат в s.

Массивы

Массив – это совокупность однотипных элементов. Описание типа массива имеет вид:

TYPE

<имя типа>= **Array**[<тип индекса>, <тип индекса>, …] **Of** <тип элемента>;

Здесь **Array** и **Of** – служебные слова. В квадратных скобках указывается список индексов. Количество индексов определяет размерность массива (одномерный, двумерный, трехмерный, …), причем размерность массива не ограничивается. В качестве индекса могут выступать любые дискретные типы (в том числе перечислимые и ограниченные типы), кроме longint. Элементами массивов могут быть значения любых типов, например, составных.

Описать массив можно также в разделе **VAR**.

Примеры описаний:

```
CONST
    Nmax=100;

TYPE
    mas1=Array [1..Nmax] Of integer;
    mas2=Array[1..100] Of real;
VAR
    A : mas1;
    B : mas2;
    C : Array [1..10, 1..20] Of real;   {двумерный массив}
```

D : **Array** [0..Nmax] **Of** byte;

Выбирая размерность массива и тип его элементов, следует помнить, что общий объем памяти, отводимый TP под все глобальные переменные, не должен превышать 64К.

В языке Pascal поддерживается принцип строгой типизации, поэтому переменные A и B, описанные в виде

VAR
A : **Array**[1..5] **Of** real;
B : **Array**[1..5] **Of** real;

считаются переменными разных типов (присваивания между ними не допустимы).

Для обеспечения совместимости необходимо объявить эти переменные следующим образом:

TYPE
mas=**Array** [1..5] **Of** real;
VAR
A, B : mas;

В этом случае возможно выполнение следующего оператора присваивания:

A:=B;

Присваивание – это единственно возможная операция над массивом в целом.

Обращение к элементам массива осуществляется с помощью следующей конструкции: после имени массива указывается в квадратных скобках индекс нужного элемента или список индексов, определяющий элемент многомерного массива. В качестве индекса может выступать произвольное выражение, тип которого должен соответствовать типу индекса в описании массива. Например,

A[Nmax] – обращение к последнему элементу массива,
B[1] – к первому элементу,
C[i,j] – к элементу двумерного массива (при этом значения i и j не должны выходить за указанные в описании границы),
D[2*k] – (2*k) должно быть <= Nmax.

Множество операций над элементами массива определяется типами этих элементов.

Символьные массивы: система TP позволяет представлять символьные данные не только строками, но и в виде одномерных массивов с элементами типа char. Например,

Var
S: **Array** [1..16] **Of** char; {символьный массив}

Для этого массива допустима такая операция присваивания:

S :=’Структуры данных’;

Эта операция выполнится только в случае точного соответствия длины строки и размера массива.

Недопустимой является операция присваивания вида S := S1+S2, где S, S1, S2 – символьные массивы (вне зависимости от размера массива S).

К элементам строк можно обращаться так же, как и к элементам символьных массивов. Например, S[1]→'C' , S[11]→'д'.

Записи

Запись – это объединение разнотипных элементов в единую конструкцию. Описание записи представляет собой список описаний ее элементов, называемых полями. Список полей размещается между ключевыми словами **record** и **end.** Каждое поле должно иметь собственное, уникальное в пределах записи, имя и собственный тип.

Например, при описании объекта "Студент" необходимо указать такие данные: ФИО, дата рождения, пол, адрес и другие. Тип такого объекта удобно представить записью следующего вида:

```
Type
   Stud = record
              Name: string[25];
              Day: 1..31;
              Montsh: 1..12;
              Year: integer;
              Sex: (мужской, женский);
              Address: string[50];
                  . . . . . .
          end;
```

Определение переменных типа Stud:

```
Var
   student1,  student2: Stud;
```

Обращение к полям этих переменных осуществляется посредством точечной нотации, имеющей такую конструкцию:

< имя переменной >.< имя поля >

Например,

```
Student1.Name:='Иванов И.И.';
Student1.Day:=10;
Student1.Montsh:=2;
    . . . . . .
```

Более сложной структурой, содержащей записи, является массив записей:

```
Type
   Group=Array [1..25] Of Stud;
Var
    Students: group;
```

Доступ к полям записей, составляющих массив Students, производится следующим образом:

```
Students [1].Name:=’Иванов И.И.’;
Students [j].Name:=’Степанов С.С.’;
. . . . .
```

В состав записи могут входить поля, также имеющие тип записи. Если информацию о дате рождения представить типом

```
Type
   Date=record
          Day: 1..31;
          Montsh: 1..12;
          Year: integer;
        End;
```

то описание типа Stud примет вид:

```
Stud = record
          Name: string[25];
          Birtdate:date;
          Sex: (мужской, женский);
          Address: string[50];
             . . . . . .
        end;
Var
   st:Stud;
```

Обращение к элементу Birtdate строится по общему правилу:

```
St.birtdate.day:=25;
St.birtdate.montch:=2;
St.birtdate.year:=1990;
```

(дата рождения – 25.02.1990 г.).

Множества

Множество строится из значений одного (базового) типа и может содержать до 256 **различных** элементов.

Описание типа множества (множественного типа) имеет вид:

TYPE

< имя типа > = **Set Of** < базовый тип >;

Set, Of - служебные слова.

Базовым типом могут быть:

- целый тип (значения должны находиться в диапазоне от 0 до 255),
- тип **char**,

- перечислимый тип,
- ограниченный тип.

Явное изображение множества в программе: список элементов в квадратных скобках. Порядок расположения элементов во множестве не фиксируется. Многократное включение элемента во множество эквивалентно однократному его упоминанию.

Над множествами определены такие операции:

A*B – пересечение множеств (результат: множество, состоящее только из тех элементов, которые содержатся одновременно и в A, и в B);

A+B – объединение множеств (результат: множество, полученное слиянием элементов множеств A и B; повторяющиеся элементы исключаются);

A-B –разность множеств (результат: множество, состоящее из элементов, входящих в A, но не входящих в B);

A=B – проверка на равенство множеств (результат: True, если A и B состоят из одинаковых элементов независимо от порядка следования и False – в противном случае);

A<>B – проверка на неравенство множеств (результат: True, если A и B отличаются хотя бы одним элементом и False – в противном случае);

A<=B – проверка на подмножество (результат: True, если все элементы A содержатся и в B независимо от порядка их следования и False – в противном случае);

A>=B – проверка на надмножество (результат: True, если все элементы B содержатся в A и False – в противном случае);

X **In** A (или X **In** […]) – проверка вхождения элемента X во множество A (результат: True, если X принадлежит базовому типу множества и входит в A (или в […]), False, если A не содержит в себе значение X)

Операция **In** бывает очень полезна при проверке попадания в диапазоны перечислимых типов. Например,

```
. . .
Writeln (‘Продолжить работу? y/n’);
Ch:=Readkey;   { запоминается код нажатой клавиши}
If  Ch In [‘n’, ‘N’, ‘т’, ‘Т’] then
      Exit;     {работа окончена}
. . .
```

(в условии оператора **If** использовано множество – константа).

Примеры описаний множеств:

```
Type
    Al=Set of 1..5;     {множество из чисел от 1 до 5}
    Bt=Set of byte;     {множество из чисел }
    Ch=Set of char;     {множество из символов}
Var
S, S1, S2 : Al;
Sim1, Sim2 : Ch;
B1, B2 : Bt;
```

Можно опустить описание типа а разделе TYPE и сразу задавать его в разделе описания переменных:

```
Var
   Col = Set of (Red, Blue, Green);
  Mas = Set of [1, 2, 3, 4, 10, 20, 30, 40];
```

Примеры изображений множеств:

```
S:= [];      {пустое множество}
S1:=[1, 2];
Sim1:=['a'..'z'];
Sim2:=['a'..'z', '0'..'9'];
Col1:=[Red,Green];
```

4. ОПЕРАТОРЫ

Оператор – это синтаксическая конструкция, предписывающая выполнить определенные действия. Язык Pascal содержит набор различных операторов, с помощью которых любой алгоритм решения задачи можно однозначно отобразить в программу.

Операторы могут быть простыми или сложными (структурными). Простой оператор не содержит внутри себя никаких других операторов. Простыми являются операторы присваивания, процедуры, перехода и пустой оператор. В состав структурных операторов могут входить другие операторы, которые, в свою очередь, также могут быть структурными, причем глубина вложенности операторов не ограничивается.

К сложным операторам относятся условные операторы (**IF, CASE**), циклические (**FOR, DO WHILE, REPEAT**), а так же оператор **WITH.** Любые операторы должны разделяться между собой символом ";".

Оператор присваивания

Это самый простой и наиболее используемый оператор языка. Имеет следующую структуру:

< **переменная** > := < **выражение** >

<переменная> – это имя переменной или имя функции;

<выражение> – синтаксическая конструкция, в которой задается порядок вычисления некоторого значения.

Выполнение оператора: "выражение" вычисляется, и его значение присваивается переменной, указанной в левой части оператора. При этом тип выражения должен быть совместим по присваиванию с типом переменной (допускается случай, когда тип переменной –**real**, а тип выражения - **integer**).

Выражение строится из операндов, знаков операций и круглых скобок. Операндами могут быть переменные (простые переменные, поля записей, элементы массивов, вызовы функций) и константы. Виды операций зависят от типов операндов.

Порядок выполнения выражения определен приоритетами операций, входящих в выражение. Сначала выполняются операции высшего приоритета. Если же подряд стоят операции одного приоритета, то они выполняются поочередно слева направо. В случае, когда установленный порядок должен быть изменен, используются круглые скобки. При вычислении выражения в первую очередь выполняются операции, указанные в круглых скобках.

Для быстрого увеличения или уменьшения значений целых переменных можно использовать следующие **арифметические процедуры:**

Inc(X [, n]) – увеличивает значение первого аргумента на величину второго аргумента, а при его отсутствии – на 1. Например, оператор i:=i+1 и Inc(i) – эквивалентны, но второй оператор выполняется быстрее первого.

Dec(X [, n]) – уменьшает значение первого аргумента на величину второго аргумента, а при его отсутствии – на 1. Например, оператор i:=i-1 и Dec(i) – эквивалентны, но второй оператор выполняется быстрее первого.

Составной оператор

Составной оператор – это последовательность операторов, заключенная между служебными словами **Begin** и **End** (перед End не обязательно ставить ”;”). Составной оператор применяется в тех случаях, когда синтаксис языка Pascal допускает использование только одного оператора, в то время как семантика программы требует задания некоторой последовательности действий.

Условные операторы

Оператор IF предназначен для реализации в программе разветвлений. Полная форма оператора представляется так:

IF < логическое выражение > **THEN** оператор1
ELSE оператор2;

Логическое выражение задает условие разветвления. Если условие удовлетворяется (логическое выражение принимает значение True), то выполняется ”оператор1”, если не удовлетворяется (False) - “оператор2”.

Логическое выражение в общем случае строится с помощью логических операций **Not**, **And, Or, Xor** и операций сравнения **<, <=, >, >=, =, <>, In** (проверка принадлежности множеству). В частных случаях в логических выражениях используются только операции сравнения.

Краткая форма оператора

IF < логическое выражение > **THEN** оператор;

используется для проверки условия, а затем либо выполнения, либо пропуска “оператора”.

Если по ветвям **IF** должен выполняться не один оператор, а несколько, то эти операторы нужно оформить составным оператором.

Примеры оператора IF:

1) { краткая форма оператора }

```
IF imin <>i   then
  begin
     tmp:=t[i];
     t[i]:=t[imin];
     t[imin]:=tmp;
end;
```

Внимание! Для того чтобы текст программы был понятным и удобным для восприятия, рекомендуется при написании сложных операторов делать отступы.

2) { краткая форма оператора }

```
IF (j>0) and (key <t[j].kl)   then
  begin
     t[j+1]:=t[j];
     j:=j+1;
end;
```

3) { полная форма оператора }

```
 IF x<y  then
    z:=sin(x)
 else          {перед ключевым словом else  символ ";" не ставится }
    begin
        z:=cos(x);
        dec(j);
    end;
```

4). { вложенные if}

```
IF key< t[i].kl  then
    vg:=i-1
 else
   if  key> t[i].kl then
      ng:=i+1
    else
       begin
         fl:=true;
         k:=i;
       end;
```

При использовании вложенных условных операторов может возникнуть ситуация синтаксической неоднозначности при определении, к какому **IF** относится **ELSE**. Эта неоднозначность разрешается так: служебное слово **ELSE** всегда ассоциируется с ближайшим по тексту **IF,** которое еще не связано с **ELSE.**

Оператор CASE (оператор варианта) предназначен для организации разветвления по нескольким направлениям. Имеет следующую структуру:

CASE <селектор> **OF**

```
    <const1> : < оператор1>;
    <const2> : < оператор2>;
        . . .
    <constN> : < операторN>
        [ ELSE < оператор>; ]
  END;  { конец CASE}
```

<селектор> - переменная (допускается также выражение) дискретного типа, причем для целого типа ее значения должны находиться в следующем диапазоне : -32768..32767.

<const> - все указанные в операторе константы должны иметь тип, совместимый с типом селектора, и быть уникальными в пределах этого оператора.

Кроме одиночных констант ветви CASE могут помечаться также списками и/или диапазонами значений, которые в этом случае должны разделяться запятыми.

Выполнение CASE: значение селектора сравнивается последовательно с указанными константами, и при совпадении выполняется соответствующий константе оператор. На этом выполнение CASE завершается. Если значение селектора не совпало ни с одной из указанных констант, то выполняется оператор, следующий за ELSE. Если же в операторе варианта ELSE отсутствует, то оператор CASE окажется пустым, то есть не выполнит никаких действий.

При необходимости по ветвям CASE может находиться не один оператор, а несколько. В этом случае их нужно сгруппировать в составной оператор.

Примеры:

1)
```
case ch of    {селектор ch должен иметь тип char }
  '1': begin
         psp:=nil;
         create_list(PSP);
       end;
  '2': print_list(PSP);
  '3': change_node(PSP);
  '4': begin
         destroy_list(PSP);
         Writeln('Список удален!');
         Readkey;
       end;
end; {case}
```

2)
```
case i of     {i - целого типа }
      1, 2, 3 : Proc1;
   4, 5, 8..12 : Proc2;
      13..20 : Proc3
else
```

```
        Proc4;
end; {case}
```

Proc1, Proc2, Proc3 и Proc4 являются операторами обращения к процедурам.

Диапазоны констант не должны пересекаться и не должны содержать констант, указанных по другим ветвям CASE.

Операторы цикла

Операторы цикла используются в программировании в случаях, когда нужно многократно повторять выполнение некоторой группы операторов. В языке Pascal существуют три вида операторов цикла: **FOR, WHILE, REPEAT.**

Оператор FOR используется для циклов с известным числом повторений, то есть в тех случаях, когда число повторений цикла может быть определено перед его началом.

Синтаксис оператора:

FOR < переменная > := <нач. значение> **TO** <кон. значение> **DO**
< оператор > ;

Здесь

<переменная> - параметр цикла (управляющая переменная); в качестве параметра должна использоваться простая переменная дискретного типа.

<начальное значение> и <конечное значение> - диапазон изменения значения параметра цикла; при каждом повторении значение параметра цикла автоматически увеличивается на 1.

<оператор> - многократно выполняемый оператор (тело цикла); если в цикле должна выполняться группа операторов, то она оформляется составным оператором.

Выполнение **FOR:** параметру цикла присваивается начальное значение, затем выполняется тело цикла. Далее значение параметра увеличивается на 1, и снова выполняется тело цикла. Этот процесс продолжается до тех пор, пока значение параметра цикла не превысит конечного значения. При превышении происходит выход из цикла, то есть завершается выполнение оператора **FOR.**

Если начальное значение параметра цикла равно конечному значению, то тело цикла выполнится только один раз. Если же начальное значение превышает конечное, то операторы тела цикла не выполнятся ни разу.

Ограничения:

1. Начальное и конечное значения должны иметь тип, совместимый с типом параметра цикла. Они могут задаваться не только константами, но и выражениями, которые вычисляются только один раз - перед первым выполнением тела цикла.

2. В теле цикла запрещается явное изменение значения параметра цикла.

Пример:

```
for i:=1 to n do
  begin
    s:=s+x[i];
    p:=p*x[i];
  end;
```

Если параметр цикла должен изменять значения от большего к меньшему, то используется такая форма оператора:

FOR < переменная > := <нач. значение> **DOWNTO** <кон. значение> **DO**
< оператор > ;

(“начальное значение” >= “конечное значение”).

Выполнение: при каждом повторении значение параметра цикла уменьшается на 1. Например,

```
for i:=n downto 2 do
  x[i]:=x[i-1];
```

Замечание: при написании операторов цикла также рекомендуется использовать отступы.

Оператор WHILE используется в тех случаях, когда заранее неизвестно, сколько раз должно повториться выполнение операторов тела цикла. Такой цикл продолжает свою работу до тех пор, пока выполняется определенное условие. Синтаксис оператора:

WHILE < логическое выражение > **DO**
< оператор >;

< логическое выражение > - определяет условие выполнения < оператора>.
< оператор > - тело цикла, обычно телом цикла является составной оператор.

Выполнение **WHILE**: сначала проверяется условие выполнения; если оно удовлетворяется (логическое выражение принимает значение **True**), то тело цикла выполняется. Если условие не удовлетворяется (**False**), то оператор **While** завершает работу. Возможна ситуация, когда тело цикла ни разу не выполнится.

Так как логическое выражение определяет условие продолжения работы цикла, то в его состав должны входить переменные, изменяющие свое значение при каждой новой итерации. Например,

```
. . .
ng:=1;  vg:= n;
fl:=false;
while (ng<=vg) and not fl do
  begin
    i:=(ng+vg) div 2;
    if key<T[i].kl then
      vg:=i-1
```

```
  else
     if key>T[i].kl then
       ng:=i+1
     else
       fl:=true;
end;
```

Оператор **WHILE** называют оператором цикла с предусловием.

Оператор REPEAT также используется в случаях, когда заранее неизвестно число повторений цикла, но требуется, чтобы тело цикла выполнилось хотя бы один раз. Синтаксис оператора:

REPEAT
< оператор1 >**;**
< оператор2 >**;**
. . .
UNTIL < логическое выражение >;

Последовательность операторов между ключевыми словами **REPEAT** и **UNTIL** составляет тело цикла.

< логическое выражение > - определяет условие выхода из цикла.

Выполнение оператора: сначала выполняется тело цикла, затем проверяется Until – условие. Если оно не удовлетворяется (то есть логическое выражение принимает значение False), то итерация повторяется и снова проверяется условие. Процесс итераций продолжается до тех пор, пока логическое выражение примет значение True (условие выполнится). После этого произойдет выход из цикла.

Оператор **REPEAT** называют оператором цикла с постусловием.

Пример:

```
i:=1;  j:=n;
Repeat
    . . .
   i:=i+1;
    . . .
   j:=j-1;
    . . .
Until  i>j ;
```

Оператор над записями

Известно, что при обращении к полям переменных, имеющих тип записи, нужно использовать составные имена следующей конструкции:

< имя переменной > . < имя поля >

Рассмотрим фрагмент программы:

```
Type
   Stud = record
            Name: string[25];
            Day: 1..31;
            Montsh: 1..12;
```

```
            Year: integer;
                . . . . . .
        end;
Var
   student: Stud;
   Group: Array [1..25] Of Stud;
        . . .
Begin
      Student.Name:=’Иванов И.И.’;
      Student.Day:=10;
      Student.Montsh:=2;
           . . . . . .
End;
```

(в программе использованы составные имена).

В языке Pascal существует возможность упростить программу с переменными-записями, указав имя переменной, являющееся общим для составных имен, в операторе **WITH**.

Структура оператора:

```
WITH  < имя переменной >  DO
        < оператор >;
```

Или

```
WITH  < имя переменной >  DO
   Begin
        < оператор1 >;
        < оператор2 >;
           . . .
   End;
```

С использованием **WITH** предыдущие операторы будут выглядеть так:

```
WITH  Student  DO
   begin
      Name:=’Иванов И.И.’;
      Day:=10;
      Montsh:=2;
        . . .
   end;
```

Теперь операторы, манипулирующие с полями записи, выглядят компактнее и нагляднее, чем в предыдущем варианте. Кроме того, эти операторы будут выполняться быстрее, так как адрес записи Student вычисляется только один раз при обработке конструкции

```
WITH  Student  DO
```

Использование **WITH** дает еще большую эффективность при обработке массива записей. Например,

```
For i:=1 to 25 do
  Begin
 . . .
```

```
WITH  Group[i]  DO
  Begin
    . . .
    Writeln (‘Фамилия:’, Name);
    Writeln (‘Год рождения:’, Year);
        . . .
  end;
 . . .
end;
```

Если фрагмент программы содержит операции над записями разных видов, то в операторе **WITH** можно указать список необходимых переменных-записей:

```
WITH  < имя1 >[,  < имя2 > . . . ]  DO
        Begin
         < оператор1 >;
         < оператор2 >;
. . .
        End;
```

5. ПРОЦЕДУРЫ И ФУНКЦИИ

Если программу, реализующую весь алгоритм решения задачи, представить единым блоком, то ее будет трудно отлаживать и изменять. Нужно выделить в алгоритме части, соответствующие определенным функциям вычислительного процесса, и согласно этому выделению разбить программу на самостоятельные фрагменты – подпрограммы.

Подпрограмма снабжается именем и становится отдельной синтаксической единицей программы. Представление программы как совокупности относительно обособленных фрагментов делает ее наглядной, легко проверяемой и в итоге - более эффективной.

Вызов подпрограммы (то есть выполнение действий, заданных в подпрограмме в виде последовательности операторов) может быть произведен в некоторой точке программы посредством указания имени этой подпрограммы. Для определенной настройки работы подпрограммы можно передать ей из точки вызова некоторую информацию. Такая настройка подпрограммы реализуется с помощью понятия параметров. Через параметры подпрограмме задаются исходные данные для ее работы (входные параметры) и возвращаются результаты выполнения подпрограммы (выходные параметры).

Использование механизма параметров характеризуется следующими преимуществами:

- способствует большей гибкости и универсальности подпрограммного механизма;

- ослабляет взаимную зависимость подпрограмм и точек их вызова (то есть делает подпрограммы самостоятельными, завершенными частями вычислительного процесса);
- способствует разработке структурированных программ.

В системе ТР представлены два вида подпрограмм:

- процедуры.
- функции.

Они различаются **назначением** и **способом использования.**

Если результатом выполнения подпрограммы является несколько значений (в частном случае – одно значение), то она оформляется в виде процедуры. Если же подпрограмма возвращает одно значение, которое может быть использовано далее при вычислении выражения, то она оформляется в виде функции.

Структуру процедуры (описание процедуры) можно представить так:

< заголовок процедуры >
< тело процедуры >
< конец процедуры >

Заголовок процедуры:

Procedure < имя > (список формальных параметров);

(все формальные параметры должны указываться с типами данных).

Тело процедуры практически повторяет структуру всей Pascal – программы: содержит описания объектов, с которыми осуществляются действия, и последовательность операторов, составляющих смысл процедуры.

Конец процедуры: ключевое слово **END** и **";"**.

Пример 1

```
Program Lin_Sort;
Uses Crt;
Const Nmax=100;
Type Rec=Record
       Name: String;
       Area: Integer;
     end;
     Table=array [1..Nmax] of Rec;

Var T: Table;                    {таблица}
    n: integer;                  {количество записей}
   ch: byte;

    .........
procedure SortTab(var T:Table; n:integer);
{процедура сортировки таблицы методом линейного выбора}
var i, imin,j:integer;
          tmp:rec;
```

```
begin
    for i:=1 to n-1 do
  begin
    imin:=i;
    for j:=i+1 to n do
      if t[j].Area<t[imin].Area then
        imin:=j;
    if imin<>i then
    begin
      {обмен записей}
      tmp:=t[i];
      t[i]:=t[imin];
      t[imin]:=tmp;
    end;
  end;
end; { конец процедуры}

begin
 . . .
SortTab(T, n);
 . . .
End. { конец программы}
```

Переменные, описанные в процедуре (в данном примере это переменные *i, j, imin, tmp*), называются **локальными** переменными. Они размещаются в оперативной памяти (в сегменте стека) при активизации процедуры. После завершения выполнения процедуры память, отведенная для локальных переменных, освобождается (то есть связь с этими переменными разрывается). Кроме того, в сегменте стека размещаются и формальные параметры процедуры.

В некоторый момент выполнения программы в сегменте стека может присутствовать несколько групп локальных переменных, соответствующих цепочке вызванных и не завершенных процедур. Отведение и освобождение памяти для локальных переменных выполняется по принципу стека. По умолчанию сегменту стека назначается размер, равный 16 К.

Формальные параметры процедуры могут быть:

- входными (передают в процедуру данные, необходимые для выполнения алгоритма, реализованного в процедуре),
- выходными (возвращают результаты выполнения процедуры).

Возможны частные случаи:

- входной параметр может быть одновременно и выходным,
- отсутствуют входные параметры,
- отсутствуют выходные параметры,

- вообще отсутствуют параметры (процедуры без параметров).

Формальные параметры не являются реальными объектами программы: при вызове процедуры они заменяются фактическими параметрами. Именно фактические параметры передаются процедуре и обрабатываются в ней.

В ТР существуют два способа передачи параметров:

- по ссылке,
- по значению.

Передача параметра по ссылке (в списке формальных параметров такой параметр должен указываться с описателем **Var**) означает передачу в процедуру адреса фактического параметра, указанного при обращении к процедуре. При этом любые изменения формального параметра внутри процедуры вызывают изменения значения фактического параметра.

Параметры, которые возвращаются процедурой, обязательно должны передаваться по ссылке.

Передача параметра по значению обеспечивает сохранность величины переданного параметра, то есть все изменения параметра внутри процедуры никак не отражаются на значении фактического параметра. По значению могут передаваться только входные параметры.

Вызов процедуры (обращение к процедуре) выполняется с помощью оператора процедуры:

< имя процедуры > (список фактических параметров);

Фактические параметры – это реальные объекты программы; при компиляции программы им выделяются участки памяти.

Требования к фактическим параметрам:

- между фактическими и формальными параметрами должно соблюдаться соответствие в количестве, порядке следования, типах;
- фактический параметр, передаваемый по значению, может быть константой, переменной, выражением;
- фактический параметр, передаваемый по ссылке, не может быть константой или выражением;
- параметры - массивы рекомендуется всегда передавать по ссылке.

При обращении к процедуре происходит передача управления на первый оператор процедуры, а по завершении работы процедуры управление передается на оператор, следующий за оператором вызова процедуры.

По определению работа процедуры завершается после выполнения последнего оператора ее тела. Однако в любой точке процедуры можно прервать ее выполнение с помощью системной процедуры **Exit**. Exit завершает работу процедуры и возвращает управление в точку вызова. Если же вызов Exit был произведен из главной программы , то программа

завершит выполнение. Кроме того, можно выйти из программы в операционную систему с помощью системной процедуры Halt.

Пример 2

```
Program  Summa;
Var  i, j, k : integer;
Procedure Sum1 (x, y, z : integer);
  { все параметры передаются по значению}
Begin
   z:=x+y;
         Writeln ('Sum1:  сумма=', z);
      End;
Procedure Sum2 (x, y : integer;  Var z:integer);
{ x, y передаются по значению, z – по ссылке}
Begin
    z:=x+y;
    Writeln ('Sum2:  сумма=', z);
End;
Begin      { главная  программа }
  i:=1;
  j:=2;
 Sum1 (i, j, k);
 Writeln ( 'После обращения к Sum1:  сумма=', k);
 Sum2 (i, j, k);
 Writeln ( 'После обращения к Sum2:  сумма=', k);
End.
```

Проанализируем работу приведенной программы.

В процедуре *Sum1* все параметры передаются по значению. Механизм передачи по значению: значение фактического параметра копируется и заносится в некоторую системную переменную, например, в *S1*. В процедуре формальный параметр, соответствующий указанному фактическому параметру, заменяется на *S1*, и все действия будут выполняться над *S1* (следовательно, значения фактического параметра не изменятся).

Таким образом, значения фактических параметров *i, j, k* заносятся в системные переменные, например, в *s1, s2, s3* соответственно.

После обращения к *Sum1* на экран выведутся такие сообщения:

Sum1: сумма = 3

После обращения к Sum1: сумма = ???

что означает, что значение переменной *k* не определено.

После обращения к *Sum2* на экран будет выведено:

Sum2: сумма = 3

После обращения к Sum2: сумма = 3

Процедура *Sum1* организована неправильно, так как параметр-результат должен передаваться по ссылке.

Пример 3

Если в программе, приведенной в примере 1, написать заголовок процедуры в виде

```
procedure SortTab(T:Table; n:integer);
```

то при обращении к процедуре возникнут следующие ошибки.

Так как параметр *T* процедуры *SortTab* передается по значению, то при активизации *SortTab* создается копия таблицы *T*, которая помещается в сегмент стека. Сразу же возникнет прерывание **Переполнение стека,** так как объем памяти, требуемый для *T*, равен (2+255+1)*100=25800 байт, что превышает размер сегмента стека, назначаемый по умолчанию (16 К). Кроме того, процедура *SortTab* будет работать с копией таблицы *T* и, следовательно, сама таблица *T* останется неотсортированной.

Параметр *T* является входным – выходным, поэтому его нужно передавать по ссылке (пример 1).

Как было сказано ранее, подпрограмму можно оформить в виде **функции**. Назначение функции заключается в задании алгоритма вычисления некоторого значения и организации передачи этого значения в точку вызова.

Заголовок функции имеет следующую структуру:

Function <имя функции> (список формальных параметров) :<тип результата>;

Среди операторов тела функции должен присутствовать оператор присваивания, в левой части которого указывается имя функции, а в правой части – выражение, вычисляющее возвращаемое значение, причем значение может быть только простого, строкового или ссылочного типа.

Таких операторов присваивания может быть несколько, но в процессе выполнения тела функции обязательно должен выполниться один из них (иначе результат функции считается неопределенным).

Выход из функции осуществляется или после выполнения последнего оператора ее тела или с помощью Exit.

Пример 4

```
function hfunc(s:string; n:integer):integer;
{преобразование символьного ключа в целое число}
var len:integer;
begin
 len:=length(s); {длина ключа}
 hfunc:=((ord(s[1])+ord(s[len]))*len) mod n+1;
end;
```

6. ФАЙЛЫ

Определение файлов

Для хранения данных на внешних носителях языки программирования содержат определенные средства взаимодействия с внешними устройствами.

Область памяти на внешнем носителе, способную хранить некоторую совокупность информации, называют *файлом***.** В файл можно как помещать определенные данные, так и извлекать их из него. Эти действия имеют общее название ввод – вывод. Для организации ввода – вывода нужно определить специальные переменные *файловых типов,* которые считаются представителями файлов в Pascal-программе.

Например, описание переменной вида

Var

F :**File Of** integer;

понимается как определение под именем *F* последовательности целых чисел, расположенных, например, на магнитном диске.

С каждой переменной файлового типа связано понятие *текущего указателя* файла. Текущий указатель можно понимать как внутреннюю (системную) переменную, которая обозначает некоторый конкретный элемент файла (”указывает” на этот элемент).

При выполнении операций ввода – вывода текущий указатель может перемещаться по файлу, настраиваясь на тот или иной элемент файла. Этот элемент становится активным (текущим) и, как следствие, доступным для обработки.

В системе TP поддерживаются три типа файлов (файловых переменных):

- файлы с типом (типизированные файлы),
- файлы без типа (нетипизированные или бестиповые файлы),
- текстовые файлы.

Тип файла определяет способ хранения информации в файле.

Файл с типом состоит из однотипных компонент, причем их число при определении файла не указывается. При чтении/записи указатель файла перемещается к следующей компоненте. Поскольку все компоненты файла имеют одинаковую длину, то возможен произвольный (прямой) доступ к компонентам файла.

Объявление файла с типом имеет вид:

Var < имя переменной > : #**File Of** < тип >;

File, Of – служебные слова;

“тип” - тип элементов файла (базовый тип).

Базовый тип может быть любым типом, кроме файлового. Не допускается также комбинированный тип (тип-запись), одним из полей которого является файл.

Файл без типа состоит из последовательности элементов одинакового размера, структура которых не известна или не определена. В такой файл можно записать значение любой переменной, имеющей заданный размер. При чтении из бестипового файла допускается произвольная интерпретация содержимого очередного элемента.

Бестиповые файлы применяются обычно в процедурах копирования или при обработке файлов базы данных. Описание файла:

Var < имя переменной > : **File**;

Текстовый файл - это последовательность строк переменной длины, каждая из которых завершается маркером конца строки (CR/LF): управляющий код "перевод каретки"/ управляющий код "перевод строки" (#13/#10). Заканчивается текстовый файл специальным кодом "конец файла" (^Z или #26).

Следует заметить, что в большинстве случаев знание конкретной кодировки управляющих символов необязательно, так как системные файловые операции автоматически учитывают эти символы.

Представителем текстового файла в Pascal-программе является переменная файлового типа, которая должна быть описана стандартным типом **text**. . Например,

Var inf: text;

Текстовые файлы можно обрабатывать только построчно и строго последовательно, причем ввод и вывод нельзя выполнять одновременно для одного и того же файла.

Поддержка системой ТР текстовых файлов позволяет использовать язык Pascal при программировании широкого класса задач, имеющих нечисловой характер и связанных с обработкой текстовой информации.

Примеры описания файловых переменных:

```
Type
   zap=record
            kl: integer;
           inf: string [60];
         end;
  str80= file of strinr[80];
Var
   f1: file of char;   {  с типом   }
   f2: text;           {текстовый}
   f3: file;           {  без типа  }
   f4: str80;          {   с типом  }
   f1: file of zap;    {   с типом  }
```

Операции над файлами с типами

Язык Pascal не содержит встроенных операций над файловыми переменными. Операции над файлами полностью реализованы в виде системных процедур и функций.

Для файлов с типами используются такие системные процедуры: **Assing, Reset, Rewrite, Close, Read, Write, Seek** и системные функции **Eof, FileSize, FilePos:**

- Assing (F, < имя файла >
предназначена для установления связи между файлом на магнитном диске и переменной файлового типа, которая будет являться представителем файлового типа в программе.

Имя файла строится по правилам, принятым в операционной системе: обозначение дисковода, цепочка каталогов, приводящих к нужному файлу, полное имя файла. Полное имя записывается так:

имя . расширение

(имя<=8 символов, расширение = 3 символам; в имени не допускаются символы ' ' (пробел), ‘.’, ‘,’, ‘:’, ‘*’, ‘?’, ‘\’).

Максимальная длина имени вместе с путем не должна превышать 79 символов.

Например, пусть в программе имеется оператор

Assing (F, 'd:\mydir\test.dta');

После выполнения данной процедуры файловая переменная *F* будет связана с файлом *test.dta,* расположенным на диске *d* в каталоге *mydir*. Для файла, находящегося в текущем каталоге, обращение к процедуре будет иметь вид:

Assing (F, 'test.dta');

Следует помнить, что в программах редко используются конкретные имена файлов. Обычно второй параметр процедуры указывается строковым выражением или строковой переменной, содержащей имя файла.

- Reset (F)
предназначена для открытия файла. Открытие файла в данном случае предполагает поиск файла, связанного с файловой переменной *F* с помощью **Assign**. Эту процедуру нужно использовать в случаях, когда открываемый файл уже существует, то есть для операции чтения из файла. Текущий указатель файла устанавливается при этом на его начало (на первую компоненту, имеющую номер 0).

С помощью **Reset** можно также открывать файл для занесения в него дополнительной информации. И так как занесение возможно только в конец файла, то сначала необходимо позиционироваться в конец файла, а затем производить запись.

- Rewrite (F)
также предназначена для открытия файла, но при этом создается новый файл и текущий указатель устанавливается на его начало. Если файл с таким

именем уже существует на диске, то он удаляется, а затем создается новый файл. Процедуру **Rewrite** нужно использовать для выполнения операции занесения записей в файл.

- Close (F)
предназначена для закрытия файла (то есть все действия с файлом завершаются). При этом ликвидируются внутренние буферы, образованные при открытии этого файла. После этого файловую переменную *F* можно связать посредством процедуры **Assign** с другим файлом.

- Read (F, P1 [, P2…])
предназначена для чтения компонент из файла в указанные переменные P1, P2, … . Тип файла и тип переменных должны совпадать. Выполнение операции чтения начинается с текущей позиции файла. Значения, содержащиеся в файле, последовательно считываются и поочередно присваиваются переменным P1, P2, … После считывания каждого значения указатель файла смещается на следующую позицию. Если в процессе чтения будет достигнут *конец файла* (то есть указатель установится на позицию, не содержащую информацию), то операция будет завершена. Возникновение состояния ”конец файла” проверяется с помощью системной функции **Eof.**

- Write (F, P1 [, P2…])
предназначена для записи в файл содержимого переменных P1, P2, … Тип файла и тип переменных должны совпадать. Информация записывается в позицию файла, определяемую текущим указателем. После записи очередной переменной указатель файла передвигается на одну позицию.

- Seek (F, n)
устанавливает указатель на *n-ую* компоненту файла (*n*- выражение типа longint). Первая компонента имеет номер 0, поэтому **Seek** (F, 0) установит указатель на начало файла.

- Eof (F)
используется для поверки состояния ”конец файла”. Функция принимает логическое значение True, если достигнут ”конец файла”, и False – в противоположном случае.

- FileSize (F)
возвращает количество компонент в файле (предварительно файл нужно открыть). Результат имеет тип longint.

- FilePos (F)
возвращает номер текущей компоненты файла (первая компонента имеет номер 0). Тип результата - longint.

Обработка ошибок ввода-вывода

Системные процедуры и функции, работающие с файлами, проверяют код завершения операции ввода-вывода и при обнаружении ошибки аварийно завершают программу. На экран дисплея при этом выводится краткое диагностическое сообщение, содержащее код (условный номер) ошибки.

Ошибки ввода-вывода можно обрабатывать в пользовательской программе. Для этого используется директива компилятора **{$I-},** отключающая автоматическую проверку кода завершения, и системная функция **IOResult**, которая передает код в программу. В зависимости от полученного значения можно указать в программе те или иные действия.

Нулевое значение функции **IOResult** свидетельствует об успешном выполнении операции ввода-вывода, ненулевое – о возникшей ошибке. Например,

“Файл не найден” (код = 2)
“Путь не найден” (код = 3)
“Слишком много открытых файлов” (код = 4)
“Файл не открыт” (код = 103)
“Файл не открыт для ввода” (код =104)
“ Файл не открыт для вывода ” (код =105)

При использовании программной обработки ошибок нужно помнить следующее положение: если какая-либо операция завершилась аварийно, то до обращения к функции **IOResult** все последующие операции ввода-вывода с любыми файлами будут игнорироваться. Поэтому обработку ошибки нужно организовать так, чтобы обращение к **IOResult** и анализ кода ошибки осуществлялись сразу после выполнения операции с файлом.

Следует также помнить, что после обращения к функции **IOResult** код возврата последней операции устанавливается в 0. То есть все последующие обращения к **IOResult** будут давать нулевой результат, пока какая-либо файловая операция не закончится аварийно.

Замечание

Для того чтобы расширить файл, открытый процедурой **Reset**(F), нужно сначала выполнить позиционирование с помощью обращения к процедуре **Seek** (F, **FileSize**(F)). И так как записи нумеруются с нуля, то функция **FileSize**(F) вернет номер позиции, следующей за последней записью. На эту позицию установится текущий указатель, и с нее начнется добавление записей.

Операции над файлами без типов

Для файлов без типа системные процедуры **Reset** и **Rewrite** используются с двумя параметрами:

Reset (F [,n]), **Rewrite** (F [,n])

n – параметр типа word, задает размер записи в байтах. Если второй параметр опущен, то предполагается размер, равный 128 б.

Используются также процедуры **Seek** (F, n) и функции **Eof, FileSize, FilePos.**

Операции над текстовыми файлами

Работа с текстовыми файлами организуется построчно. Характер чтения и записи является строго последовательным, причем нельзя одновременно читать и записывать в текстовый файл.

Для работы с текстовыми файлами также используются процедуры **Assign, Reset** (только для чтения из файла), **Rewrite** (только для записи в файл), **Close.**

Дополнительно для текстовых файлов определены процедуры **Append, Read, ReadLn, Write, WriteLn** и функция **Eoln:**

- **Append** (F)

предназначена для открытия файла с последующей записью в него. Отличается от процедуры **Rewrite** тем, что не очищает открываемый файл, а только устанавливает текущий указатель в его конец (то есть процедуру **Append** удобно использовать, когда нужно добавить новые строки в конец уже существующего файла).

- **Read** ([F,] P1 [, P2…])

читает информацию из файла *F* (если файл задан, иначе – с клавиатуры) в переменные *P1, P2,…*Процедура чтения зависит от типа переменной, стоящей в списке.

- **ReadLn** ([F,] [P1, P2, …])

отличается от процедуры **Read** тем, что после прочтения данных в переменные пропускаются все оставшиеся символы данной строки и маркер конца строки. Если в процедуре отсутствует список переменных, то происходит переход к следующей строке.

- **Write** ([F,] P1 [, P2…])

записывает информацию в файл (если он задан, иначе – на экран) из задаваемых переменных *P1, P2,…* Процедура записи зависит от типа переменной, стоящей в списке. Для символьных, арифметических и строковых переменных выводится их значение, для Boolean – строка TRUE или FALSE. Если поле задает больше позиций, чем требуется для вывода, то оно слева дополняется пробелами, иначе выводятся самые правые символы из представления переменной.

- **WriteLn** ([F,] [P1, P2, …])

отличается от процедуры **Write** тем, что после вывода всех переменных в файл записывается маркер конца строки.

- **Eoln** (F)

используется для проверки конца строки. Если конец строки достигнут, то функция **Eoln** возвращает значение TRUE, иначе – FALSE. Если **Eof** (F) возвращает TRUE, то значение **Eoln** (F) также будет TRUE.

Пример

Процедура чтения записей из файла в таблицу:

```
    .  .  .
Const Nmax=100;
Type Rec=Record
           Name: String[15];
           Area: Integer;
         end;
     Table=array [1..Nmax] of Rec;

Var T: Table          {таблица}
    n: integer;       {количество записей в таблице}
    .  .  .
procedure InFileTab( var T: Table; var n: integer);
var cod: byte;
      f: file of rec;
  fname: string;
     ch: char;
begin
  ClrScr; {очистка экрана}
  repeat
    Writeln('Введите имя файла');
    ReadLn(fname);
    Assign(f,fname);
{$I-}       {отключение контроля ошибок ввода/вывода}
    Reset(f);
{$I+}
    cod:=IOResult;
    if cod<>0 then  { файл не найден }
    begin
      WriteLn('Не могу найти файл ', fname);
      Writeln('Продолжить работу? y/n');
      ch:=ReadKey;
      if ch in ['n','N','т','Т'] then
        exit;       {выход в меню}
    end;
  until cod=0;
  n:=0;
  while not eof(f)do
  begin
    n:=n+1;
```

```
    read(f,T[n]);
  end;
  close(f);
end;
    .   .   .
```

7. УПРАВЛЕНИЕ КОНСОЛЬЮ

Консоль оператора физически состоит из двух устройств: клавиатуры и монитора. Программы управления консолью содержатся в Unit Crt. Они позволяют считывать символы с клавиатуры без отображения на экране, устанавливать различные режимы отображения текста, определять “окна”, позиционировать курсор, определять положение курсора внутри окна, устанавливать цвет фона и символов, включать-выключать звуковой динамик IBM PC.

Для управления консолью используются следующие системные функции:

- **KeyPressed**

возвращает TRUE в случае когда на клавиатуре нажата клавиша, порождающая символ. На клавиши Alt, Ctrl, NumLock, ScrollLock, Shift, CapsLock функция не реагирует.

- **ReadKey**

возвращает символ, соответствующий нажатой клавише, или ждет нажатия. Введенный символ на экране не отображается.

- **Window** (X1, Y1, X2, Y2)

определяет окно (X1, Y1 – координаты левого верхнего угла, X2, Y2 – правого нижнего; тип координат – байт). Курсор устанавливается в позицию (1,1) окна, то есть в левый верхний угол. Применение процедур и функций из Unit Crt в дальнейшем будет оказывать влияние только на эту область экрана. При запуске Pascal – программы начальное окно совпадает со всем физическим экраном. На экране в каждый момент времени активно только одно (последнее) окно.

- **GotoXY** (X, Y)

перемещает курсор в позицию (X, Y) (координаты определяются в пределах окна)

- **WhereX**

определяет X – координату положения курсора относительно текущего окна.

- **WhereY**

определяет Y – координату положения курсора относительно текущего окна.

Для управления цветовым режимом вывода символов на экран удобнее всего пользоваться системной переменной **TextAttr**, определенной в секции Interface CRT.

Переменная **TextAttr** имеет следующую битовую структуру:

Номер бита	7	6 5 4	3 2 1 0
Что определяет	Мерцание 1-да, 0-нет	Цвет фона (8 значений)	Цвет символов (16 значений)
Цвет		Крас Зел Гол	Ярк Крас Зел Гол

Общая формула для значений типа Byte имеет вид:

Значение = $B7*2^7 + B6*2^6 + B5*2^5 + B4*2^4 + B3*2^3 + B2*2^2 + B1*2^1 + B0*2^0$

Множители B0, B1, … B7 – это значения соответствующих битов, равные 0 или 1, а числовые множители – это два в степени номера бита.

Выражение можно упростить:

Значение=2^4 *(B7*2^3 +B6*2^2 + B5*2^1 + B4*2^0)+(B3*2^3+B2*2^2+ B1*2^1+ B0*2^0)=

= 16* (< цвет фона >) + (< цвет символа>)

(константа, определяющая цвет фона, умножается на 16).

В Unit CRT введены следующие цветовые константы:

```
  {константы для цвета букв и фона}
Black = 0;          {0000 - черный}
Blue = 1;           {0001 - голубой}
Green = 2;          {0010 - зеленый}
Cyan = 3;           {0011 - бирюзовый}
Red = 4;            {0100 - красный}
Magenta = 5;        {0101 - малиновый}
Brown = 6;          {0110 - коричневый}
LightGray = 7;      {0111 – свело-серый}

  {константы для цвета букв}
DarkGray = 8;       {1000 – темно-серый}
LightBlue=9;        {1001 –  светло-голубой}
LightGreen=10;      {1010 – светло-зеленый}
LightCyan=11;       {1011 - салатный}
LightRed=12;        {1100 – светло-красный}
LightMagenta=13;    {1101 - розовый}
Yellow = 14;        {1110 - желтый}
```

```
White = 15;          {1110 - белый}
Blink = 128;         {мерцание букв}
```

Пример цветовых установок:
TextAttr:= 16*4 + 15=79; или TextAttr:=16*Red + White;
(на красном фоне белые символы)

Для работы с цветом используются также следующие процедуры:

- **ClrScr**

очищает текущее окно, установленное процедурой Window или взятое по умолчанию (то есть весь экран). При этом окно как бы закрашивается установленным цветом фона.

- **TextColor** (Color)

устанавливает цвет выводимых на экран символов (параметр может принимать значения от 0 до 31).

- **TextBackground** (Color)

устанавливает цвет фона под символом.

- **HighVideo**

устанавливает повышенную яркость выводимых символов.

- **LowVideo**

устанавливает обычную яркость выводимых символов.

СОДЕРЖАНИЕ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ЗАДАНИЯ

к лабораторным работам по курсу

«Алгоритмы и структуры данных»

(направление подготовки 6.050103 ”Программная инженерия”)

Составители: **Галина Георгиевна Шалдырван**

Наталья Стефановна Костюкова