

2021

# Глава 6 Использование динамически выделяемой памяти

МГТУ им. Н.Э. Баумана

Факультет Информатика и системы управления

Кафедра Компьютерные системы и сети

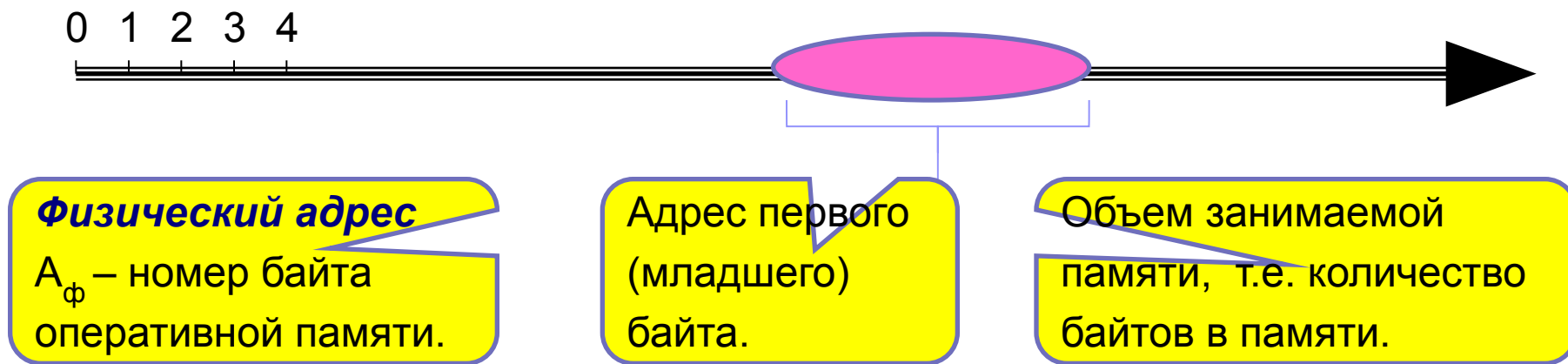
Лектор: д.т.н., проф.

Иванова Галина Сергеевна

# 6.1 Адресация оперативной памяти.

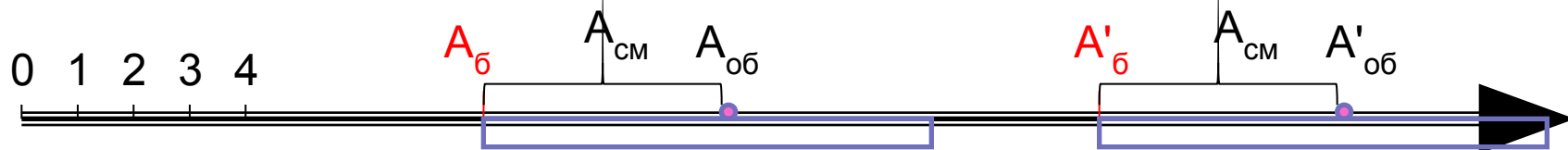
## Указатели и операции над ними

Минимальная адресуемая единица памяти большинства современных процессоров – *байт*. Байты памяти нумеруют, начиная с нуля.



Размещаемая в памяти компьютера информация: числовые и текстовые данные, а также машинные коды программы - обычно имеют размер более 1 байта, т.е. занимают в памяти некоторое количество байтов.

# Относительная адресация памяти или адресация по схеме "База + смещение"



$$A_{\text{ф}} = A_{\text{б}} + A_{\text{см}},$$

где  $A_{\text{б}}$  – адрес базы – адрес, относительно которого считают остальные адреса;

$A_{\text{см}}$  – смещение – разность физического и базового адресов.

Адресация по схеме "База + смещение" обеспечивает

"перемещаемость" программ и данных в памяти, т.е. загрузку в разные места оперативной памяти в зависимости от того, какой участок памяти свобод.

**Указатель** – тип данных, используемый для хранения **смещений**.

В памяти занимает 4 байта, адресует сегмент размером  $V = 2^{32} = 4$  Гб.

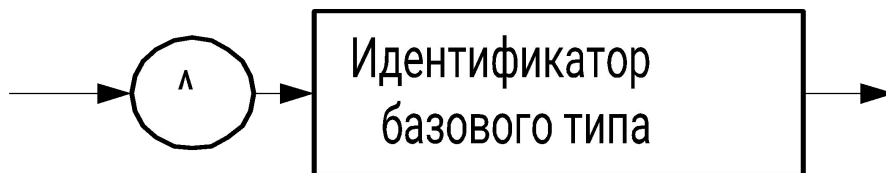
При сегментной модели памяти базовый адрес = адрес сегмента. 3

# Типизированные и нетипизированные указатели

Различают указатели:

- **типизированные** – адресующие данные конкретного типа;
- **нетипизированные** – не связанные с данными определенного типа.

Объявление типизированного указателя:



Объявление нетипизированного указателя: `pointer`

Примеры:

а) `Type tp1 = ^integer;`  
`Var pi: tp1;`

ИЛИ

`Var pi: ^integer;`

б) `Var p: pointer;`

в) `Type pp = ^person;`

`person = record`

`name: string;`

`next: pp;`

`end;`

г) `Var r: ^integer = nil;`

# Операции присваивания и получения адреса

1. *Присваивание.* Допускается присваивать указателю только значение того же или неопределенного типа.

Пример:

```
Var      p1,p2: ^integer;  
        p3: ^real;  
        p: pointer;...
```

{допустимые операции}

```
p1:=p2;    p:=p3;    p1:=p;    p1:=nil;    p:=nil; ...
```

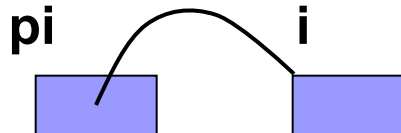
{недопустимые операции}

```
p3:=p2;    p1:=p3;
```

2. *Получение адреса.* Результат операции – указатель типа pointer – может присвоен любому указателю.

Пример:

```
Var i:integer;  
    pi: ^integer;  
    ... pi:=@i;
```



# Операции доступа и отношения

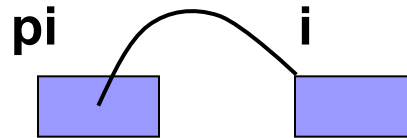
3. *Доступ к данным по указателю (операция разыменования).*  
Полученное значение имеет тип, совпадающий с базовым типом данных указателя.

*Нетипизированные указатели разыменовывать нельзя.*

**Примеры:**

```
j := pi;
```

```
pi := pi+2;
```



4. *Операции отношения: проверка равенства (=) и неравенства (< >).*

**Примеры:**

```
sign := p1 = p2;
```

```
if p1<>nil then ...
```

# Подпрограммы, работающие с указателями

1. Функция **ADDR(x) : pointer** – возвращает адрес объекта x, в качестве которого может быть указано имя переменной, функции, процедуры.

**Пример:**

```
Data := Addr (NodeNumber) ; ↔ Data := @NodeNumber ;
```

2. Функция **Assigned(const P) : Boolean** – проверяет присвоено ли значение указателю (true – если присвоено).

**Пример:**

```
if Assigned (P) then Writeln ('You won't see this') ;
```

3. Функция **Ptr(Address: Integer) : Pointer** – преобразует число в указатель.

**Пример:**

```
p := Ptr (a) ;
```

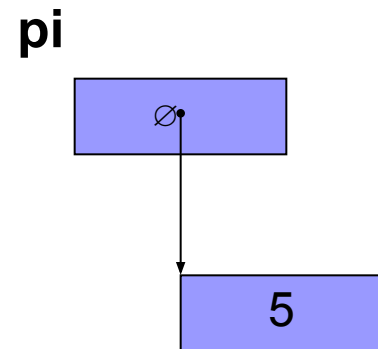
## 6.2 Динамическое распределение памяти

Управление выделением и освобождением памяти осуществляется посредством специальных процедур и функций:

1. Процедура **New**(var P: ^<тип>) – выделяет память для размещения переменной, размер определяется типом указателя.
2. Процедура **Dispose**(var P: ^<тип>) – освобождает выделенную память.

Пример:

```
Var pi:^integer;...  
  if Assigned(pi) then ... {false}  
  New(pi);  
  pi^:=5;  
  Dispose(pi);  
  if Assigned(pi) then ... {true}  
  ...
```





## Подпрограммы динамического распределения (2)

3. Процедура `GetMem(var P: Pointer; Size: Integer)` – выделяет указанное количество памяти и помещает ее адрес в указатель.
4. Процедура `FreeMem(var P: Pointer[; Size: Integer])` – освобождает выделенную память.
5. Функция `SizeOf(X) : Integer` – возвращает размер переменной в байтах.

### Пример:

```
Var Buffer: ^array[1..100] of byte;
```

```
...
```

```
GetMem(Buffer, SizeOf(Buffer));
```

```
Buffer^[1] := 125;
```

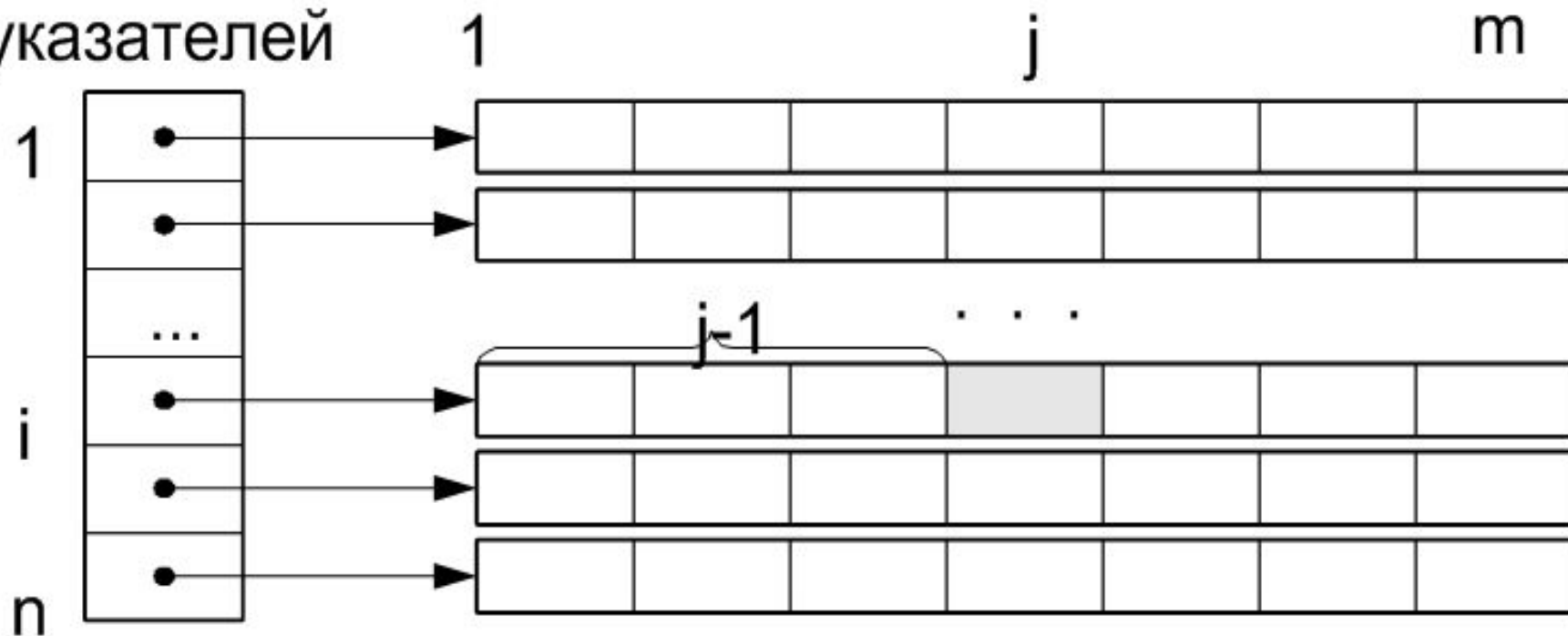
```
...
```

```
FreeMem(Buffer, sizeof(Buffer));...
```

# Динамическая матрица

Статический массив указателей

Динамические массивы строк



# Программа

```
Program Ex6_1;  
{$APPTYPE CONSOLE}  
Uses SysUtils;  
Var i,j,n,m:word; s:single;  
    ptrstr:array[1..10000] of pointer;  
Type tpsingle=^single;  
  
{Функция вычисления адреса}  
Function AddrR(i,j:word):tpsingle;  
begin  
    AddrR:=Ptr(Integer(ptrstr[i])+(j-1)*SizeOf(single));  
end;
```

# Программа (2)

Begin

```
Randomize;
```

```
WriteLn('Input n,m');           ReadLn(n,m);
```

```
for i:=1 to n do
```

```
begin
```

```
    GetMem(ptrstr[i],m*sizeof(single));
```

```
    for j:=1 to m do AddrR(i,j)^:=Random;
```

```
end;
```

```
s:=0;
```

```
for i:=1 to n do
```

```
    for j:=1 to m do    s:=s+AddrR(i,j)^;
```

```
WriteLn('Summa =',s:15:10);
```

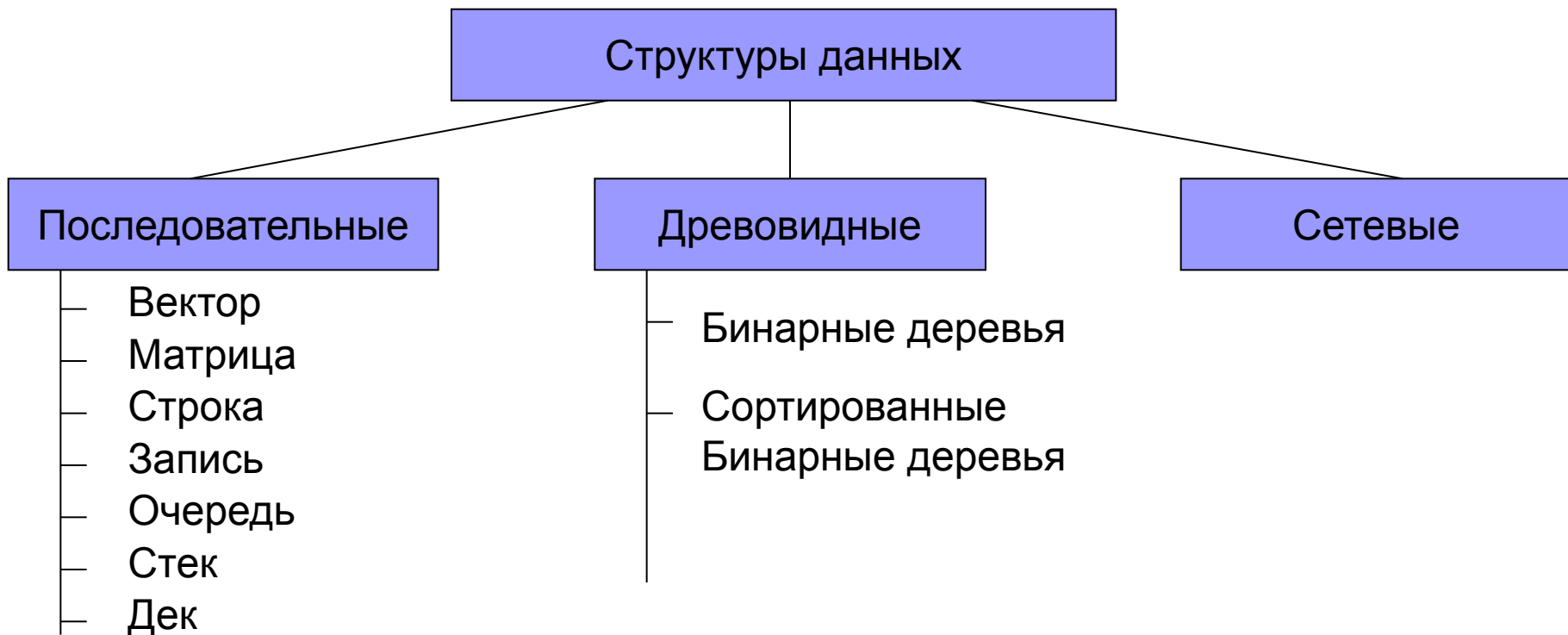
```
WriteLn('Middle value =',s/(n*m):15:10);
```

```
for i:=1 to n do    FreeMem(ptrstr[i],m*SizeOf(single));
```

```
ReadLn;
```

End.

## 6.3 Динамические структуры данных



### Динамические линейные структуры

1. Очередь – структура данных, реализующая: добавление – в конец, а удаление – из начала.
2. Стек – структура данных, реализующая: добавление и удаление с одной стороны.
3. Дек – структура данных, реализующая: добавление и удаление с двух сторон.



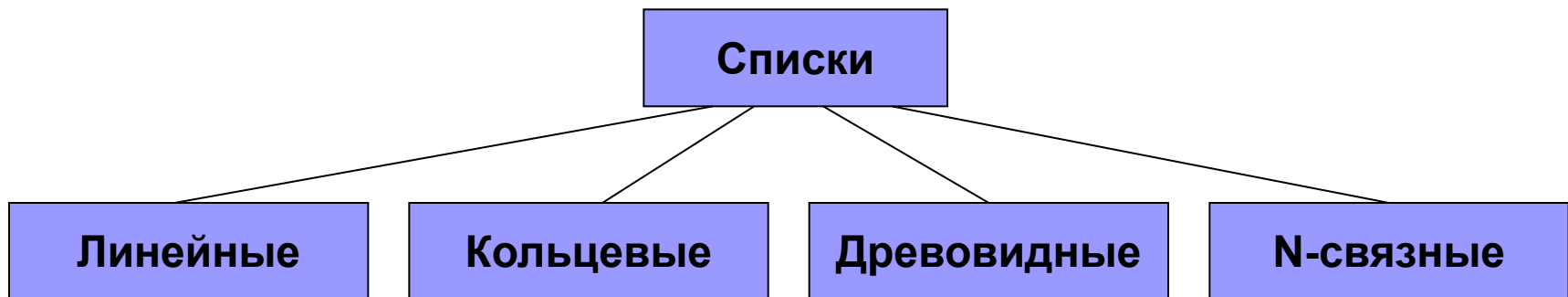
# Списки

**Список** – способ организации данных, предполагающий использование указателей для определения следующего элемента.

Элемент списка состоит из двух частей: *информационной* и *адресной*.

**Информационная часть** содержит поля данных.

**Адресная** – включает от одного до  $n$  указателей, содержащих адреса следующих элементов. Количество связей, между соседними элементами списка определяет его связность: односвязные, двусвязные,  $n$ -связные.



# Виды списков

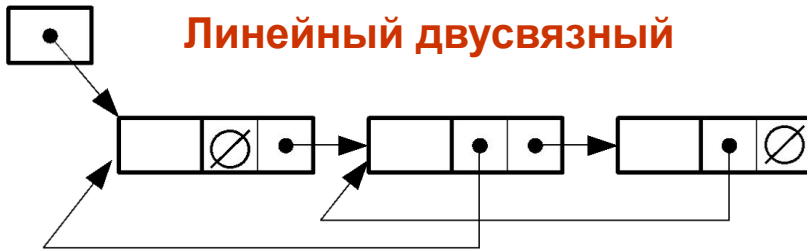
Линейный односвязный



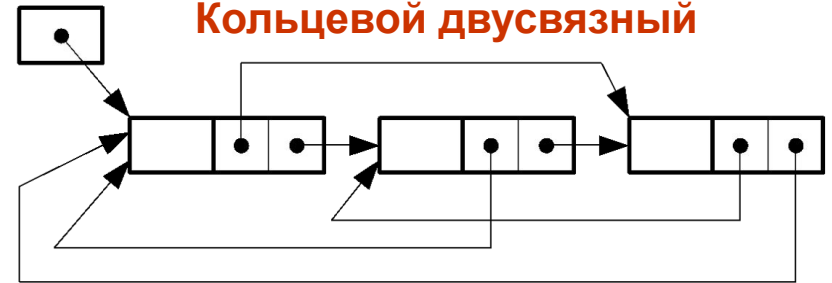
Кольцевой односвязный



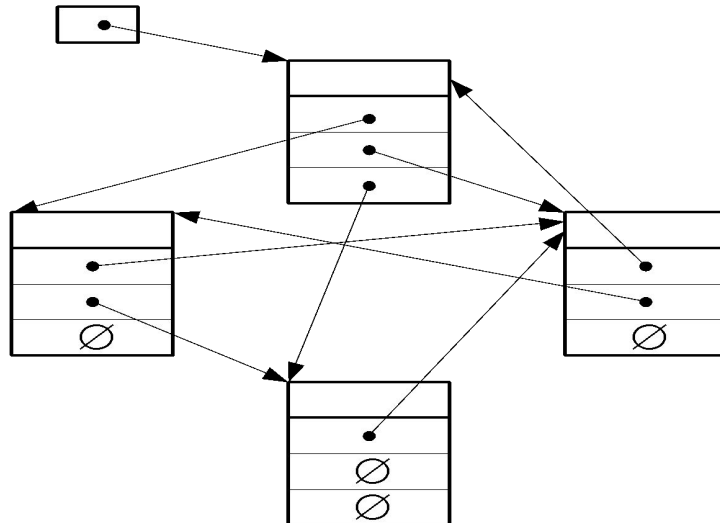
Линейный двусвязный



Кольцевой двусвязный



Сетевой n-связный



# Примеры описания элементов списка

Односвязный список:

```
Type pe = ^element;           {тип указателя}
    element = record
        name: string[16];      {информационное поле 1}
        telefon:string[7];     {информационное поле 2}
        p: pe;                 {адресное поле}
    end;
```

Двусвязный список:

```
Type pe = ^element;           {тип указателя}
    element = record
        name: string[16];      {информационное поле 1}
        telefon:string[7];     {информационное поле 2}
        prev: pe;              {адресное поле «предыдущий»}
        next: pe;              {адресное поле «следующий»}
    end;
```



## 6.4 Односвязные списки

Аналогично одномерным массивам реализуют последовательность элементов. Однако в отличие от одномерных массивов позволяют:

- работать **с произвольным количеством элементов**, добавляя и удаляя их по мере необходимости;
- осуществлять вставку и удаление записей, **не перемещая остальных элементов последовательности**;

НО

- **не допускают прямого обращения к элементу по индексу**;
- **требуют больше памяти для размещения.**

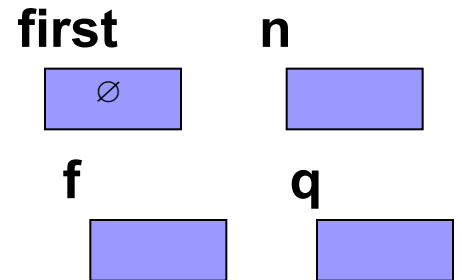
# Объявление типа элемента и переменных

Описание типа элемента:

```
Type tpe1 = ^element;    {тип «указатель на элемент»}
    element = record
        num: integer;    {число}
        p: tpe1;         {указатель на следующий элемент}
    end;
```

Описание переменной – указателя списка и несколько переменных-указателей в статической памяти:

```
Var first,                {адрес первого элемента}
    n, f, q: tpe1;        {вспомогательные указатели}
```



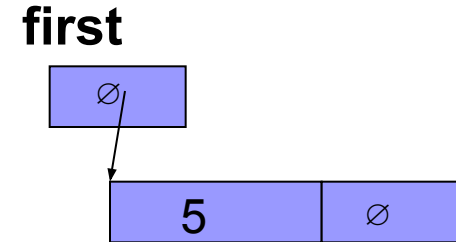
Исходное состояние – «список пуст»:

```
first := nil;
```

# Добавление элементов к списку

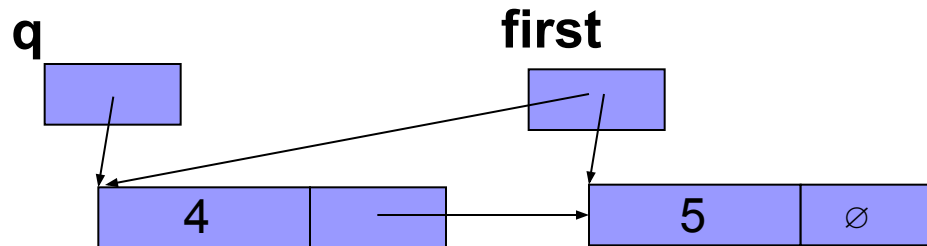
1 Добавление элемента к пустому списку:

```
new(first) ;  
first ^ .num:=5;  
first ^ .p:=nil;
```



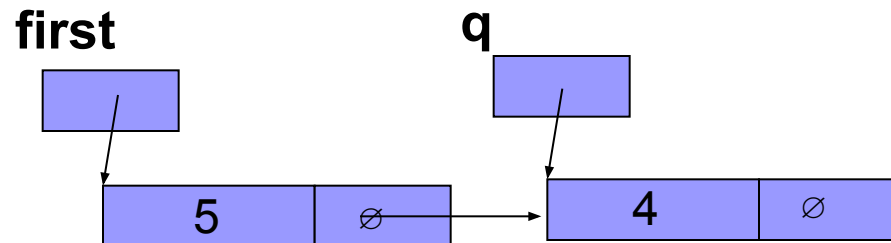
2 Добавление элемента перед первым (по типу стека):

```
new(q) ;  
q ^ .num:=4;  
q ^ .p:=first;  
first:=q;
```



3 Добавление элемента после первого (по типу очереди):

```
new(q) ;  
q ^ .num:=4;  
q ^ .p:=nil;  
first ^ .p:=q;
```



# «Стек» записей

```
Program Ex6_2;
```

```
{$APPTYPE CONSOLE}
```

```
Uses SysUtils;
```

```
Type point=^zap;
```

```
zap=record
```

```
det:string[10];
```

```
diam:real;
```

```
p:point;
```

```
end;
```

```
Var r,q,f:point;
```

```
a:zap;
```

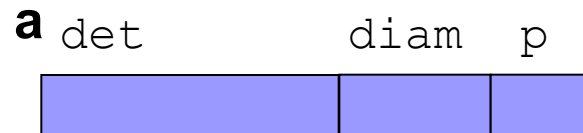
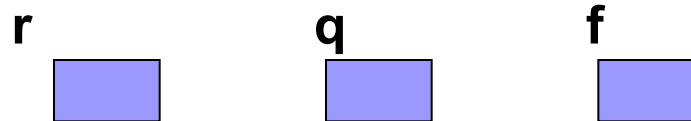
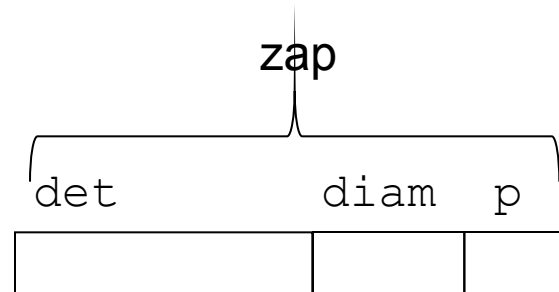
```
c:integer;
```

```
Begin new(r);
```

```
r^.p:=nil;
```

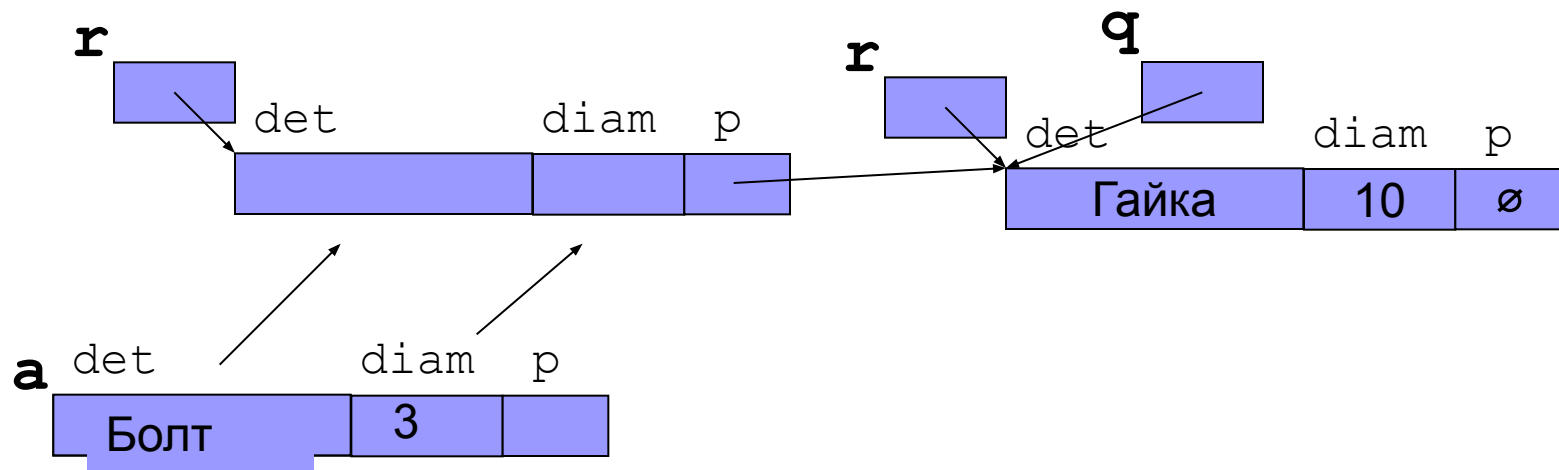
```
Writeln('Input name, diam');
```

```
Readln(r^.det,r^.diam);
```



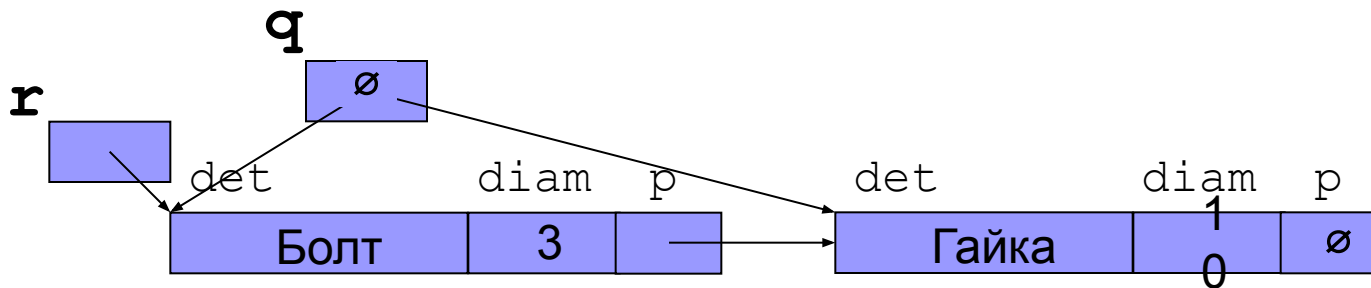
# Создание списка по типу стека

```
ReadLn (a.det) ;  
while a.det<>'end' do  
  begin ReadLn (a.diam) ;  
        q:=r ;  
        new (r) ;  
        r^.det:=a.det ;  
        r^.diam:=a.diam ;  
        r^.p:=q ;  
        ReadLn (a.det) ;  
  end ;
```

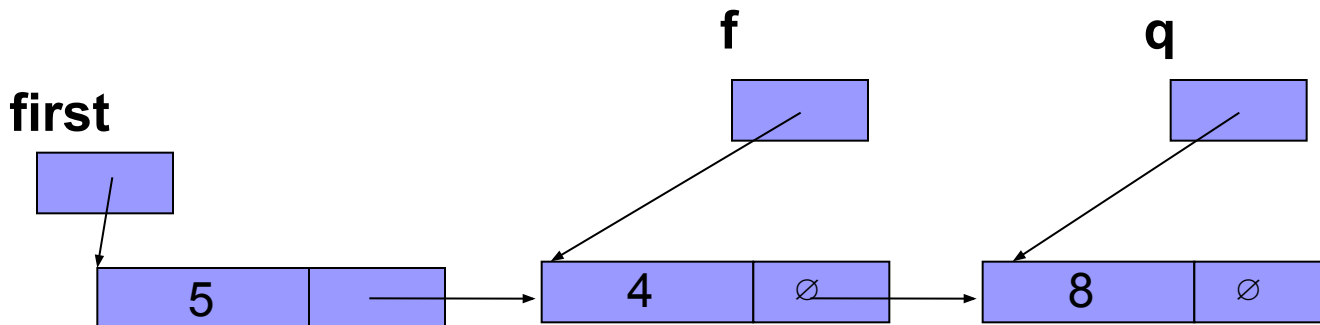
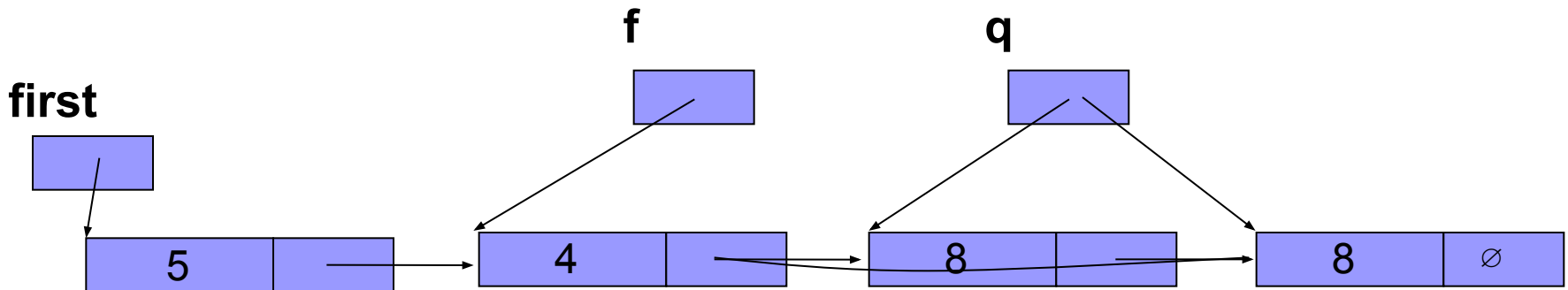
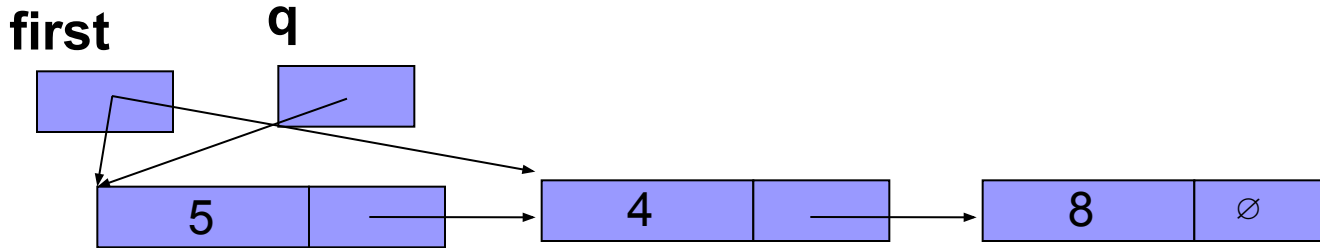


# Вывод полученного списка

```
q:=r;  
if q=nil then WriteLn('No records')  
else  
  while q<>nil do  
    begin  
      WriteLn(q^.det:11,q^.diam:5:1);  
      q:=q^.p;  
    end;
```

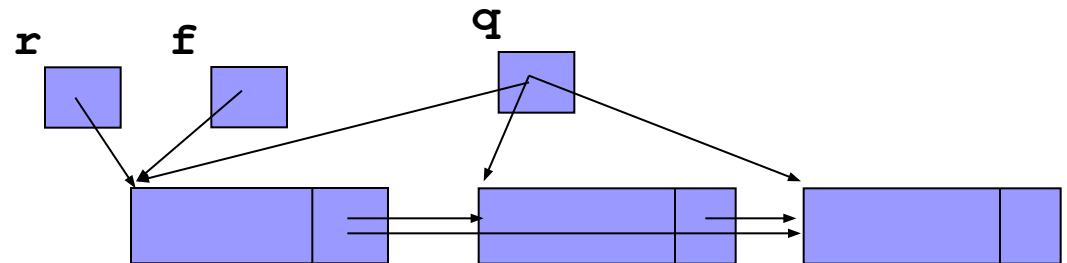
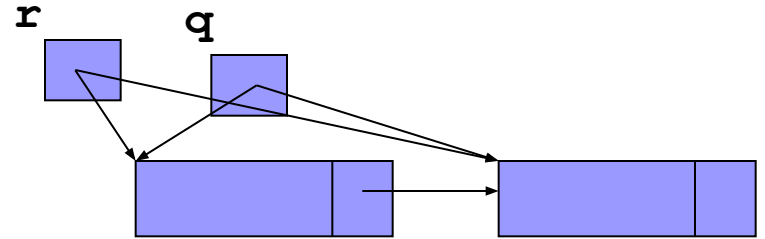


# Варианты удаления элементов



# Удаление записей

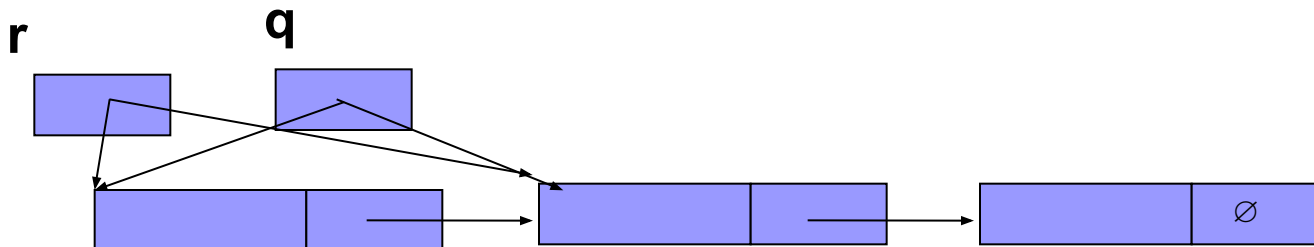
```
q:=r;  
c:=0;  
repeat  
  if q^.diam<1 then  
    if c=0 then  
      begin r:=r^.p;  
            dispose(q); q:=r  
    end  
    else  
      begin q:=q^.p;  
            dispose(f^.p); f^.p:=q  
    end  
  end  
else  
  begin f:=q;  
        q:=q^.p;  
        c:=1  
  end;  
until q=nil;
```





# Вывод результата и освобождение памяти

```
q:=r;  
if q=nil then WriteLn('No records')  
else  
  while q<>nil do  
    begin  
      WriteLn(r^.det:11,r^.diam:5:1);  
      r:=r^.p;  
      dispose(q);  
      q=r;  
    end;  
  ReadLn;  
End.
```



# КОЛЬЦЕВОЙ СПИСОК

1 2 3 4 5

```
Program Ex6_3;
```

```
{ $APPTYPE CONSOLE }
```

```
Uses SysUtils;
```

```
Var y:integer;
```

```
Function Play(n,m:integer):integer;
```

```
Type child_ptr=^child;
```

```
child=record
```

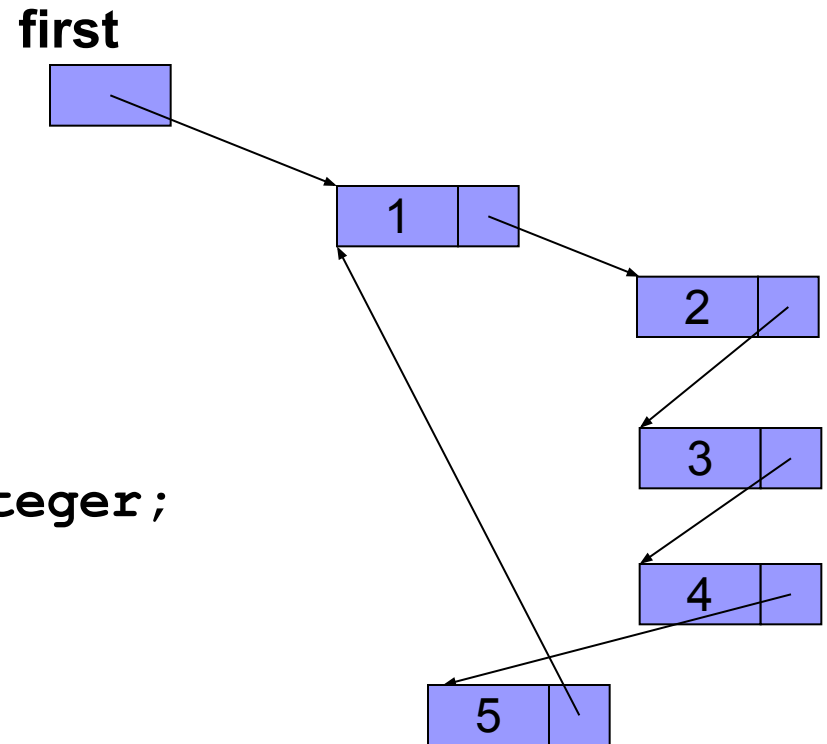
```
    name:integer;
```

```
    p:child_ptr;
```

```
end;
```

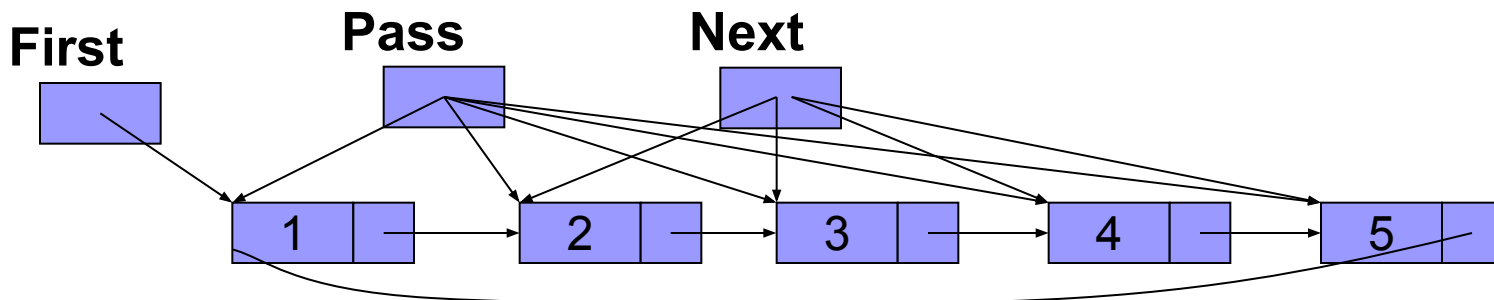
```
Var First,Next,Pass:child_ptr;
```

```
    j,k:integer;
```



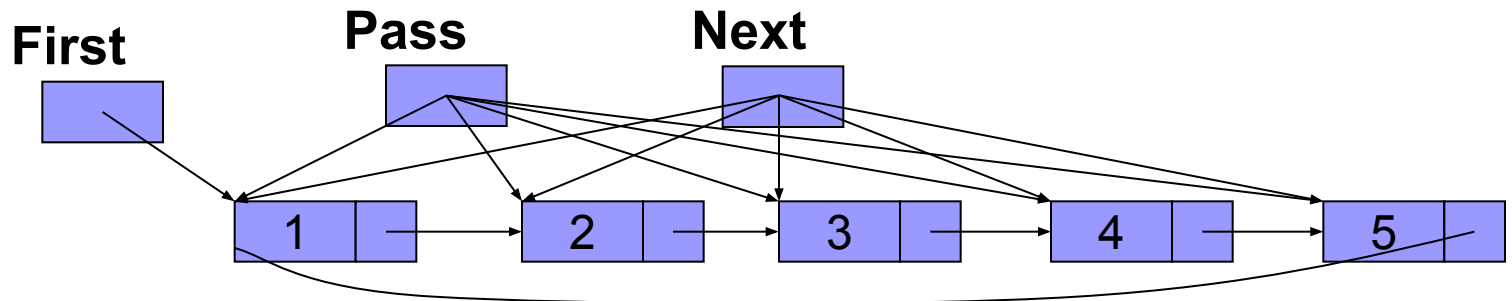
# Создание списка

```
begin { Создание списка }  
  new(First);  
  First^.name:=1;  
  Pass:=First;  
  for k:=2 to N do  
    begin new(Next);  
      Next^.name:=k;  
      Pass^.p:=Next;  
      Pass:=Next;  
    end;  
  Pass^.p:=First; {Замыкание круга}
```



# Проход по кольцу n-1 раз

```
Pass:=First;  
for k:=n downto 2 do  
  begin  
    for j:=1 to m-1 do  
      begin  
        Next:=Pass;  
        Pass:=Pass^.p;  
      end;  
    end;  
  end;
```



# Удаление m-го элемента. Основная программа

```
WriteLn (Pass^.name) ; Begin  
Next^.p:=Pass^.p;      y:=Play (5,7) ;  
dispose (Pass) ;      WriteLn ('Result =' ,y:2) ;  
Pass:=Next^.p;        ReadLn ;  
End.
```

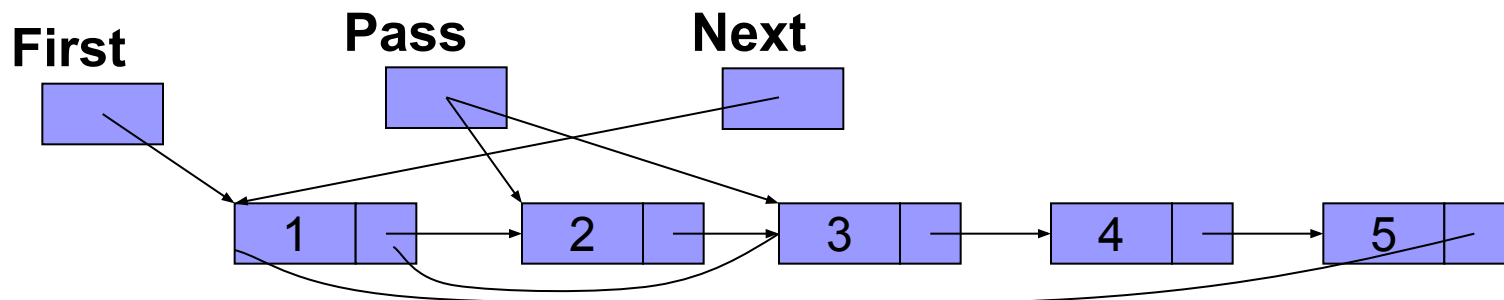
end;

{Возврат результата}

```
Play:=Pass^.name;
```

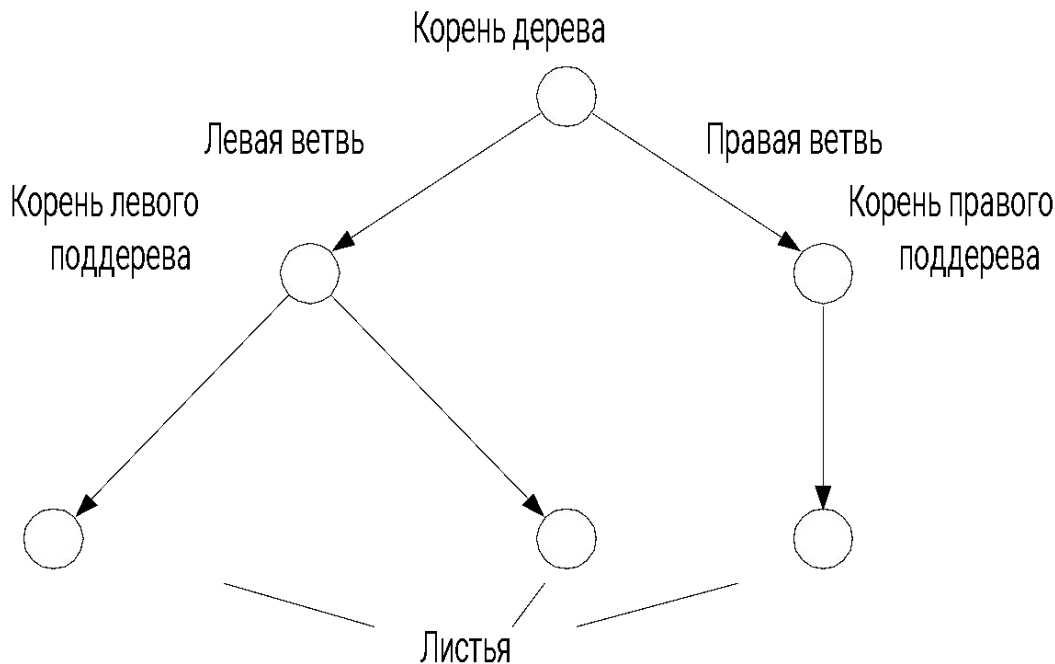
```
dispose (Pass) ;
```

end;



## 6.5 Бинарные сортированные деревья

В математике **бинарным деревом** называют конечное множество вершин, которое либо пусто, либо состоит из корня и не более чем двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня.

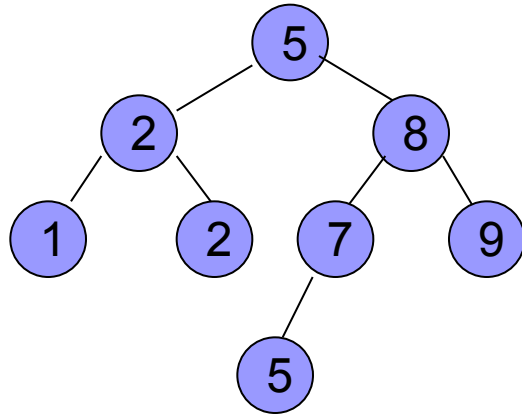


Вершины, из которых не выходит ни одной ветви, называют **листьями**

**Сортированные** бинарные деревья, строятся по правилу: *ключевое поле левого поддерева должно содержать значение меньше, чем в корне, а ключевое поле правого поддерева – значение больше или равное значению в корне.*

# Построение бинарного дерева

Рассмотрим последовательность целых чисел: {5, 2, 8, 7, 2, 9, 1, 5}



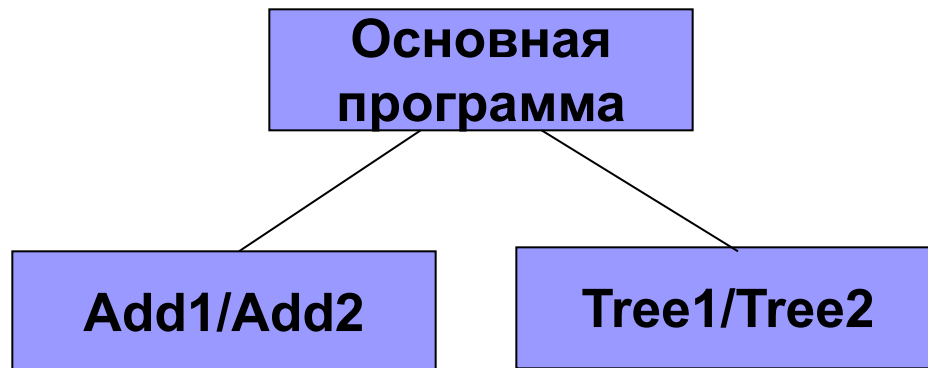
**Пример.** Разработать программу сортировки последовательности чисел с использованием бинарного дерева.

# Описание элемента древовидного списка

```
Program Ex6_4;  
{ $APPTYPE CONSOLE }  
Uses SysUtils;  
Const lim=100;  
Type top_ptr=^top;  
      top=record  
          value:integer;  
          left,right:top_ptr;  
      end;  
Var next_number:integer;  
    r,pass:top_ptr;
```

1 – нерекурсивный вариант;  
2 – рекурсивный вариант

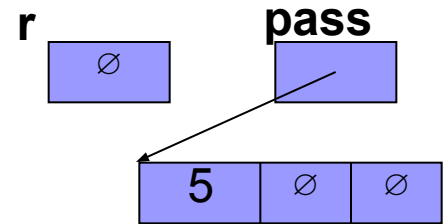
## Схема структурная ПО





# Основная программа

```
Begin WriteLn('Input numbers (End - 1000)');  
  r:=nil;  
  Read(next_number);  
  while next_number<>1000 do  
    begin new(pass);  
      with pass^ do  
        begin value:=next_number;  
              left:=nil; right:=nil;  
            end;  
        Add1(r,pass);  
        Read(next_number)  
      end;  
  end;  
  ReadLn;  
  WriteLn('Result:');  
  Tree1(r); ReadLn;  
End.
```



# Нерекурсивная процедура построения дерева

```
Procedure Add1 (Var r:top_ptr; pass:top_ptr);
```

```
Var next,succ:top_ptr;
```

```
Begin if r=nil then r:=pass
```

```
else
```

```
begin succ:=r;
```

```
while succ<>nil do
```

```
begin next:=succ;
```

```
if pass^.value<succ^.value then
```

```
succ:=succ^.left
```

```
else succ:=succ^.right;
```

```
end;
```

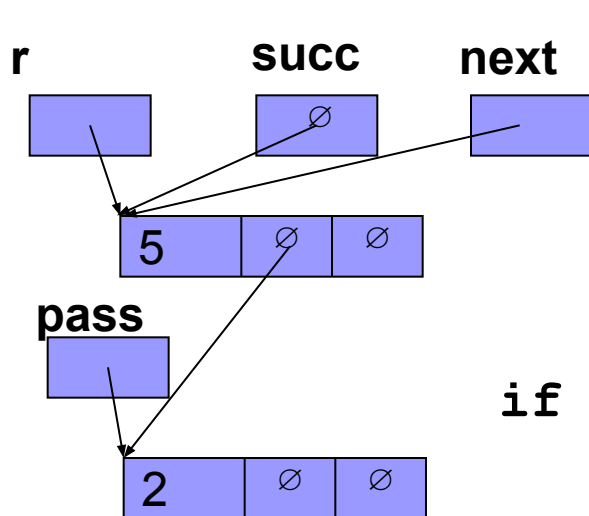
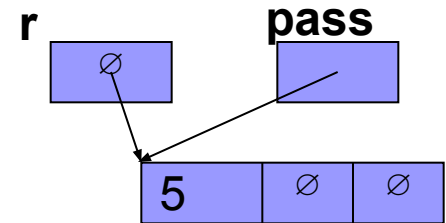
```
if pass^.value<next^.value then
```

```
next^.left:=pass
```

```
else next^.right:=pass;
```

```
end;
```

```
End;
```



# Рекурсивная процедура построения дерева

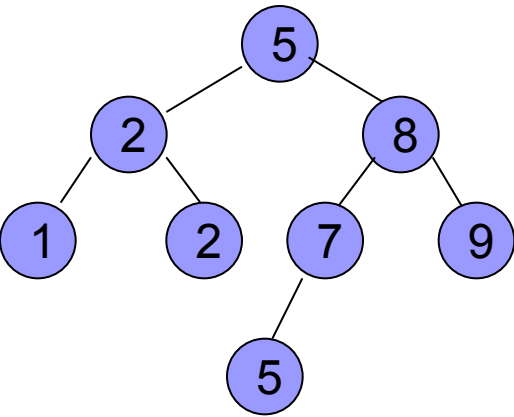
```
Procedure Add2 (Var r:top_ptr; pass:top_ptr) ;  
begin  
    if r=nil then r:=pass  
    else  
        if (pass^.value<r^.value) then  
            Add2 (r^.left,pass)  
        else Add2 (r^.right,pass) ;  
end;
```

# Нерекурсивная процедура обхода дерева

```
Procedure Tree1 (r:top_ptr) ;  
var pass:top_ptr;  
    mem_top:record  
        nom:0..lim;  
        adres:array[1..lim] of top_ptr;  
    end;  
begin pass:=r;
```

# Нерекурсивная процедура обхода дерева (2)

```
with mem_top do
  begin nom:=0;
    while (pass<>nil) or (nom<>0) do
      if pass<>nil then
        begin
          if nom=lim then
            begin Writeln('Error lim'); exit;
            end;
          nom:=nom+1;
          adres[nom]:=pass;
          pass:=pass^.left;
        end
      else begin pass:=adres[nom];
              nom:=nom-1;
              writeln(pass^.value);
              pass:=pass^.right;
            end;
        end;
      end;
    end;
```



# Рекурсивная процедура обхода дерева

```
Procedure Tree2 (r:top_ptr) ;  
begin  
  if r<>nil then  
    begin  
      Tree2 (r^.left) ;  
      Write (r^.value:4) ;  
      Tree2 (r^.right) ;  
    end ;  
end ;
```

Кроме этого при работе с деревом необходимо предусмотреть процедуру освобождения памяти, занятой двоичным деревом.

