

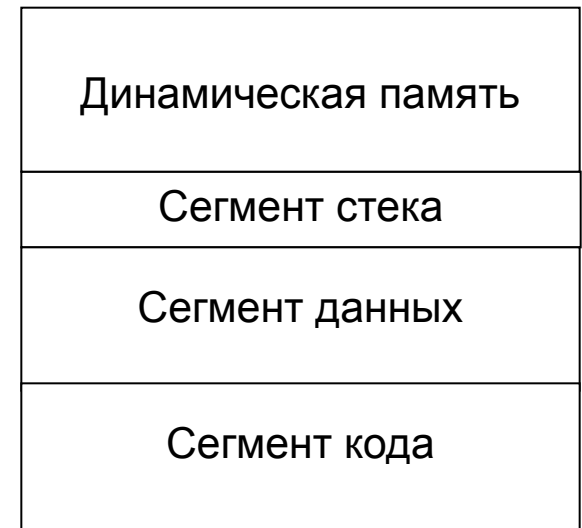
Лекция

Работа с динамической памятью

Указатели: виды, описание, использование. Динамические переменные. Динамические структуры данных: стек, очередь, линейный список, бинарное дерево.

Основные понятия

- Переменные для хранения адресов областей памяти называются *указателями*.
- В указателе можно хранить адрес данных или программного кода.
- Адрес занимает четыре байта и хранится в виде двух слов, одно из которых определяет сегмент, второе — смещение.



Виды указателей



```
graph TD; A[Виды указателей] --> B[стандартные:]; A --> C[определяемые программистом:];
```

стандартные:

```
var p : pointer;
```

определяемые программистом:

```
type pword = ^word;
```

```
var pw : pword;
```

или:

```
var pw : ^word;
```

Операции с указателями

- присваивание;

`p1 := p2;`

- проверка на равенство и неравенство:

`if p1 = p2 then ...`

Правила присваивания указателей

- Любому указателю можно присвоить стандартную константу **nil**, которая означает, что указатель **не ссылается** на какую-либо конкретную ячейку памяти: `p1 := nil;`
- Указатели стандартного типа `pointer` совместимы с указателями любого типа.
- Указателю на конкретный тип данных можно присвоить только значение указателя того же или стандартного типа.

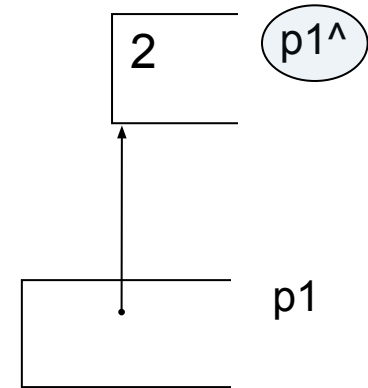
Операция разадресации

применяется для обращения к значению переменной, адрес которой хранится в указателе:

```
var p1: ^word;    // НЕ разадресация
```

```
...
```

```
p1^ := 2; inc(p1^);  writeln(p1^);  // ОНА!
```



С величинами, адрес которых хранится в указателе, можно выполнять любые действия, допустимые для значений этого типа.

Операция @ и функция addr

позволяют получить адрес переменной:

```
var w : word;
```

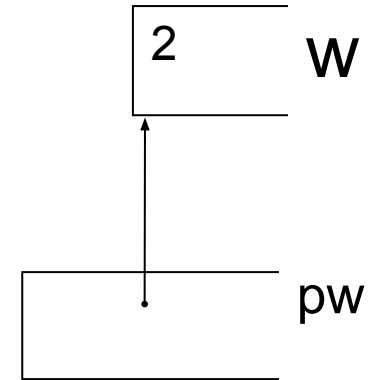
```
    pw : ^word;
```

```
...
```

```
pw := @w;
```

```
{ или pw := addr(w); }
```

```
pw^ := 2;
```



Стандартные функции

для работы с указателями:

- `seg(x) : word` — возвращает адрес сегмента для `x`;
- `ofs(x) : word` — возвращает смещение для `x`;
- `cseg : word` — возвращает значение регистра сегмента кода `CS`;
- `dseg : word` — возвращает значение регистра сегмента данных `DS`;
- `ptr(seg, ofs : word) : pointer` — по заданному сегменту и смещению формирует адрес типа `pointer`.

Динамические переменные

создаются в хипе во время выполнения программы с помощью подпрограмм `new` или `getmem`:

- Процедура `new(var p : тип_указателя)`
- Функция `new(тип_указателя) : pointer`

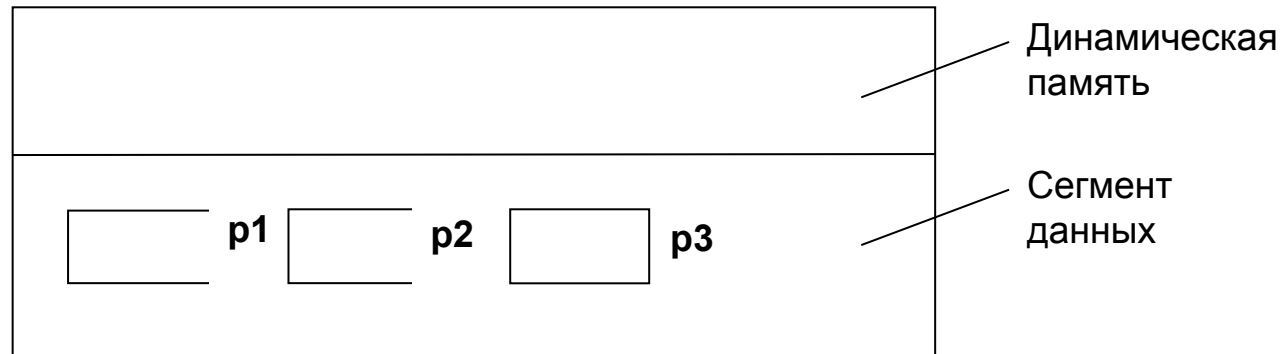
Процедура и функция `new` обычно применяются для типизированных указателей.

- Процедура `getmem(var p : pointer; size : word)`

Эту процедуру можно применять и для указателей типа `pointer`.

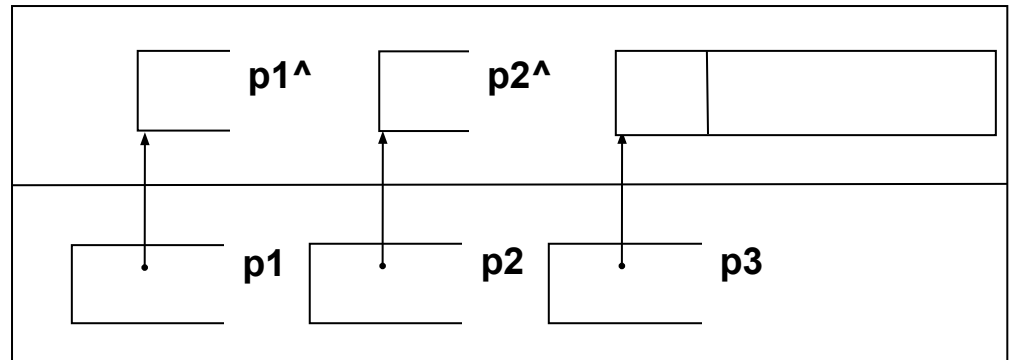
Пример работы с динамическими переменными

```
type rec = record  
    d : word;  
    s : string;  
end;  
pword = ^word;  
  
var p1, p2 : pword;  
    p3      : ^rec;
```



...

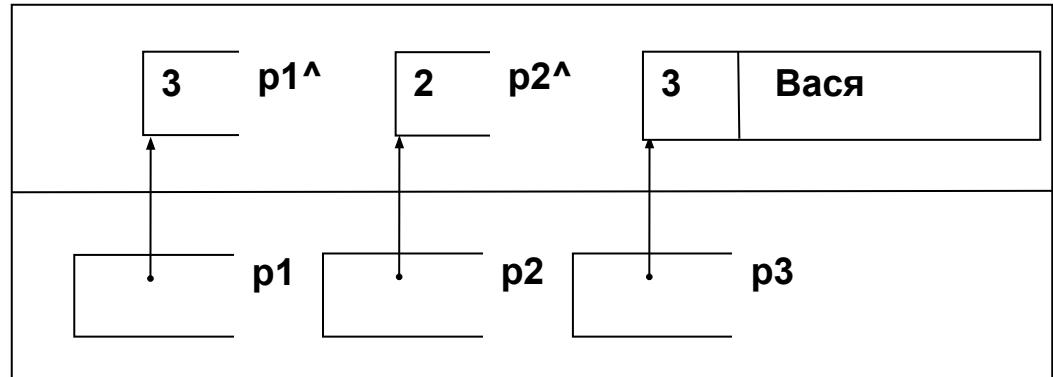
```
new(p1);  
p2 := new(pword);  
new(p3);
```



$p1^{\wedge} := 3;$ $p2^{\wedge} := 2;$

$p3^{\wedge}.d := p1^{\wedge};$

$p3^{\wedge}.s := \text{'Вася'};$

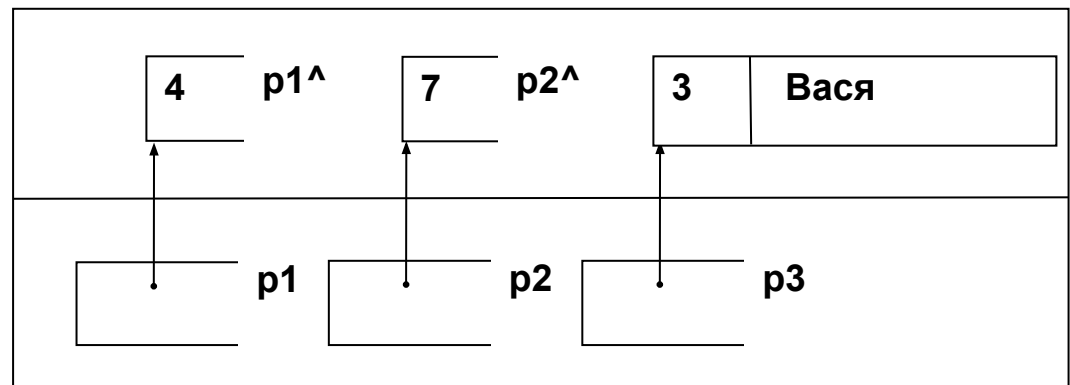


Динамические переменные можно использовать в операциях, допустимых для величин соответствующего типа:

$\text{inc}(p1^{\wedge});$

$p2^{\wedge} := p1^{\wedge} + p3^{\wedge}.d;$

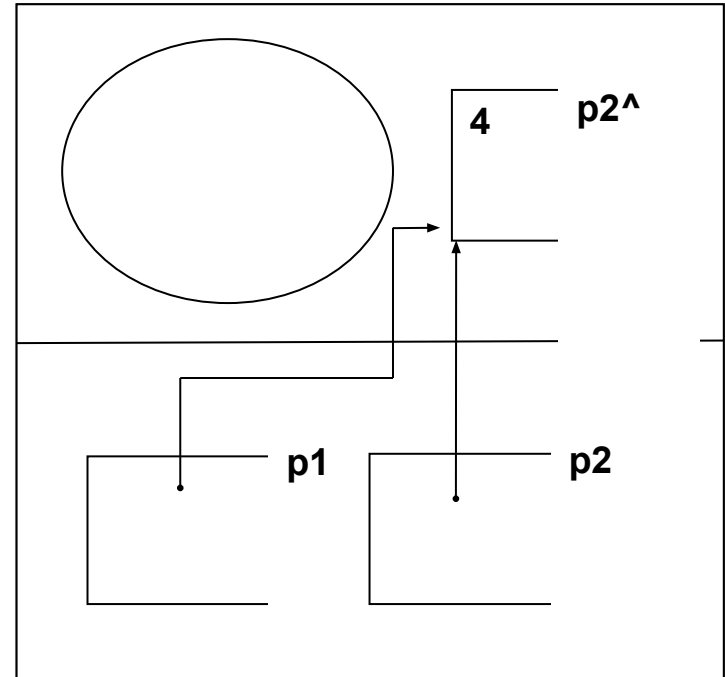
$\text{with } p3^{\wedge} \text{ do writeln (d, s);}$



Мусор

При присваивании указателю
другого значения старое
значение теряется.

Это приводит к появлению так
называемого мусора
(обозначен овалом), когда
доступа к участку
динамической памяти нет, а
сам он помечен как занятый.



`new(p1);...`

`new(p2); ...`

~~`p1 := p2;`~~

Освобождение памяти

- Процедура **Dispose**(var p : pointer)

освобождает участок памяти, выделенный New.

- Процедура **Freemem**(var p : pointer; size : word)

освобождает участок памяти размером size, начиная с адреса p.

- Если память выделялась с помощью New, следует применять Dispose, в противном случае — Freemem.
- Значение указателя после вызова этих процедур становится неопределенным.

Освобождение памяти из-под группы переменных

- Если требуется освободить память из-под нескольких переменных одновременно, можно применять процедуры Mark и Release.
- Процедура **Mark**(var p : pointer) записывает в указатель p адрес начала участка свободной динамической памяти на момент ее вызова.
- Процедура **Release**(var p : pointer) освобождает участок динамической памяти, начиная с адреса, записанного в указатель p процедурой Mark.

Вспомогательные функции

- Функция **Maxavail** : longint возвращает длину в байтах самого длинного свободного участка динамической памяти.
- Функция **Memavail** : longint возвращает полный объем свободной динамической памяти в байтах.
- Вспомогательная функция **Sizeof(x)** : word возвращает объем в байтах, занимаемый x, причем x может быть либо именем переменной любого типа, либо именем типа

Лекция

Динамические структуры данных

Виды динамических структур

В программах чаще всего используются:

- линейные списки
- стеки
- очереди
- бинарные деревья

Элемент динамической структуры состоит из двух частей:

1. *информационной*;
2. *указателей*:

type

d	s	p
---	---	---

pnode = ^node;

node = record

d : word; { | информационная | }

s : string; { | часть | }

p : pnode; { указатель на следующий элемент }

end;

Стек

Реализует принцип обслуживания LIFO
(Last In – First Out).

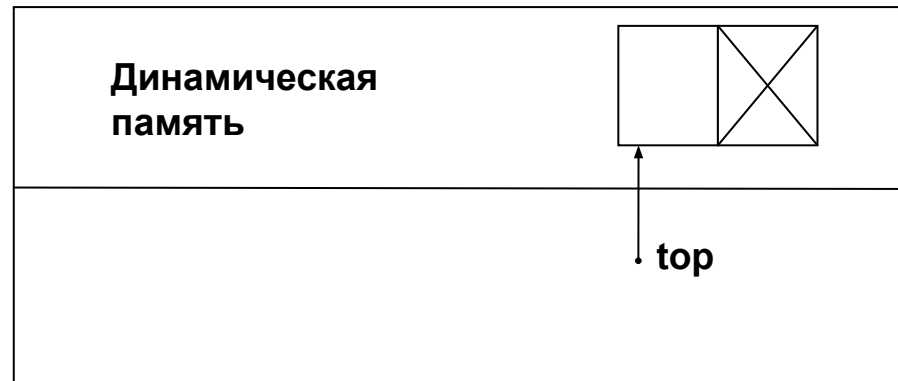
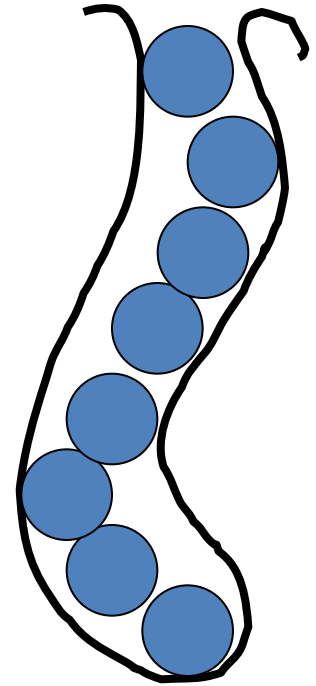
Для работы со стеком используются две статические переменные:

- указатель на вершину стека;
- вспомогательный указатель:

```
var top, p : pnode;
```

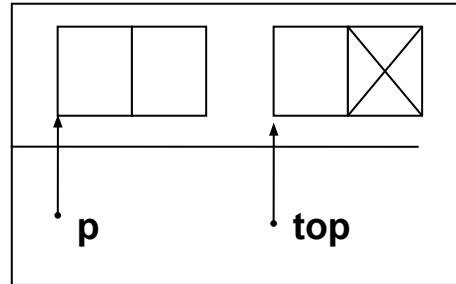
Создание первого элемента стека:

```
new(top);  
top^.d := 100;  
top^.s := 'Вася';  
top^.p := nil;
```

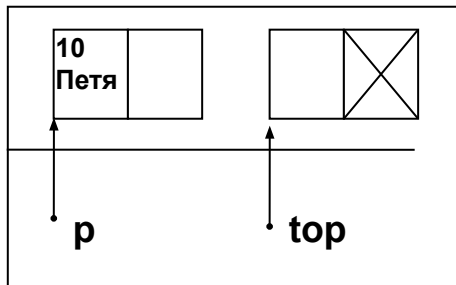


Добавление элемента в стек

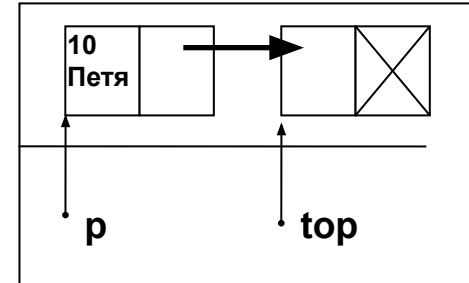
1. Выделение
памяти
 $\text{new}(p);$



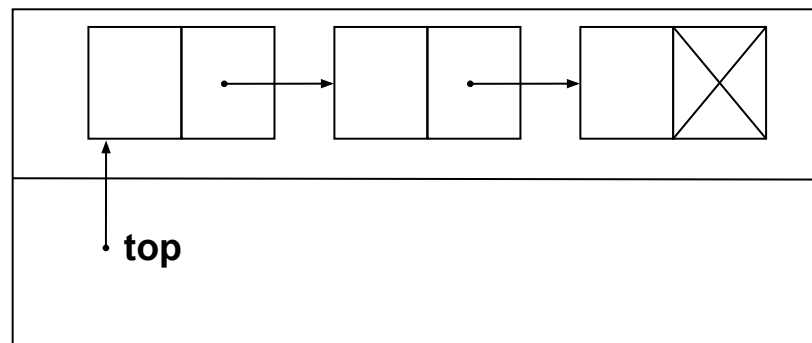
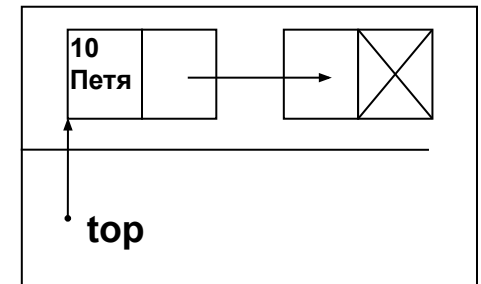
2. Занесение данных
 $p^{\wedge}.d := 10;$
 $p^{\wedge}.s := \text{'Петя'};$



3. Связь с предыдущим
 $p^{\wedge}.p := \text{top};$



4. Обновление
указателя на вершину
 $\text{top} := p;$

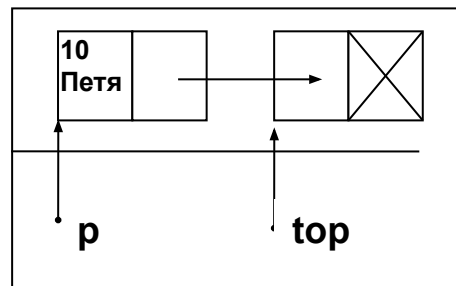
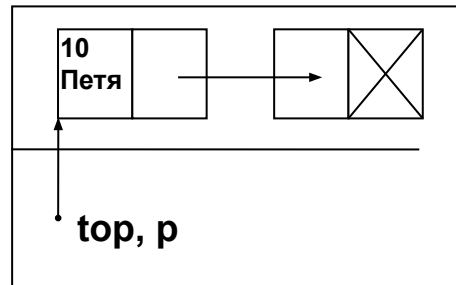


Выборка из стека

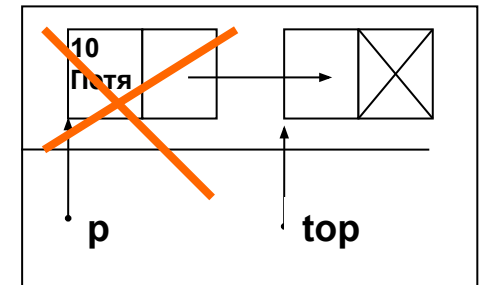
1. Выборка данных
with top^\wedge do writeln (d, s);

2. Запомнить ук-ль
 $p := \text{top}$;

3. Обновить ук.
на вершину
 $\text{top} := \text{top}^\wedge.p$;



4. Освободить память
 $\text{dispose}(p)$;



Пример работы со стеком

Программа формирует стек из пяти целых чисел и их текстового представления и выводит его на экран.

```
program stack;
const n = 5;
type      pnode = ^node;
          node = record                                d : word;
                s : string;
                p : pnode;
          end;
var top : pnode;
    i   : word;
    s   : string;
const  text : array [1 .. n] of string = ('one', 'two', 'three', 'four', 'five');
```

```

{ ----- ЗАНЕСЕНИЕ В СТЕК ----- }
function push(top : pnode; d : word; const s : string) : pnode;
var p : pnode;
begin
    new(p);
    p^.d := d;    p^.s := s;    p^.p := top;
    push := p;
end;

```

```

{ ----- ВЫБОРКА ИЗ СТЕКА ----- }
function pop(top : pnode; var d : word; var s : string) : pnode;
var p : pnode;
begin
    d := top^.d;    s := top^.s;
    pop := top^.p;
    dispose(top);
end;

```

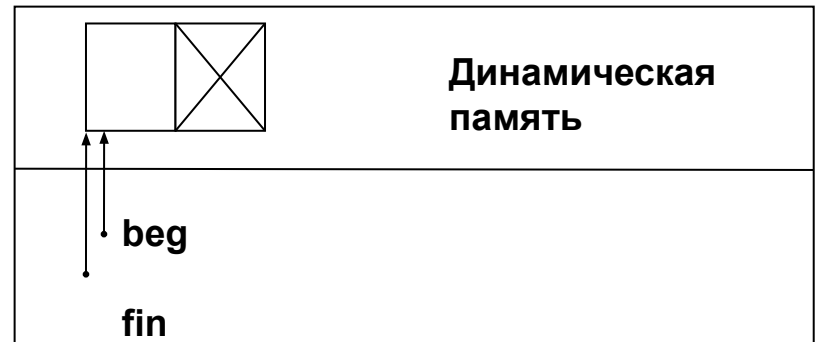
```
{ ----- главная программа ----- }  
begin  
    top := nil;  
    { занесение в стек: }  
    for i := 1 to n do top := push(top, i, text[i]);  
  
    { выборка из стека: }  
    while top <> nil do begin  
        top := pop(top, i, s);  writeln(i:2, s);  
    end;  
end.
```

Очередь

Реализует принцип обслуживания FIFO

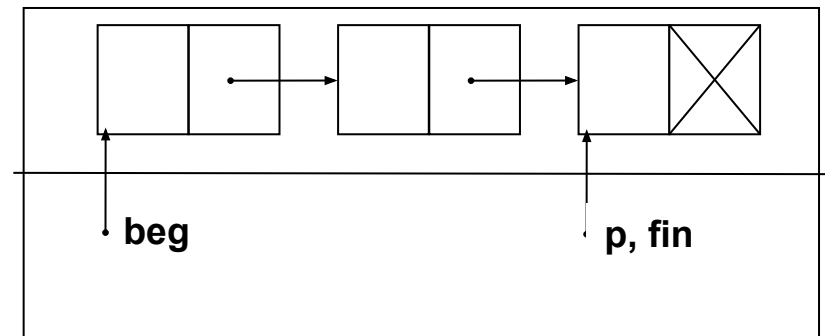
Начальное формирование очереди:

```
new(beg);  
beg.d := 100; beg.s := 'Вася';  
beg.p := nil;  
fin := beg;
```



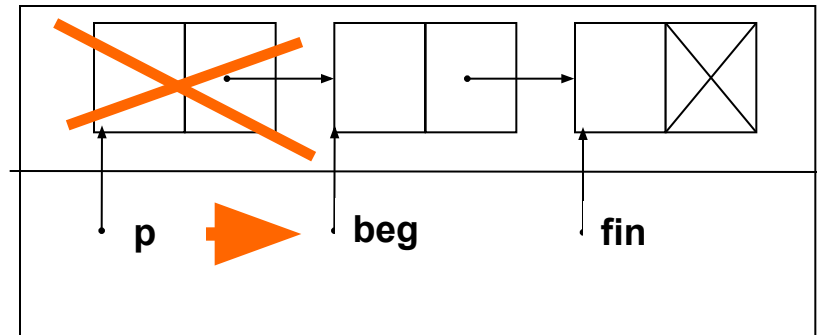
Добавление элемента в конец:

```
new(p);  
p.d := 10; p.s := 'Петя';  
p.p := nil;  
fin.p := p;    fin := p;
```

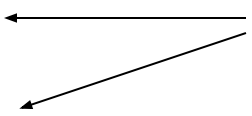


Выборка элемента из начала

```
with beg^ do writeln (d, s);  
p := beg;  
beg := beg^.p;  
dispose(p);
```



Линейные списки

- односвязные
 - двусвязные
 - кольцевые
- 

Каждый элемент списка содержит ключ, идентифицирующий этот элемент.

Операции со списком:

- начальное формирование списка (создание первого элемента);
- добавление элемента в конец списка;
- чтение элемента с заданным ключом;
- вставка элемента в заданное место списка (до или после элемента с заданным ключом);
- удаление элемента с заданным ключом;
- упорядочивание списка по ключу.

Пример работы со списком

Программа, формирующая односвязный список из пяти элементов, содержащих число и его текстовое представление. Выполняет вставку и удаление заданного элемента. В качестве ключа используется число.

d	s	p
---	---	---

```
program linked_list;  
const n = 5;  
type      pnode = ^node;  
      node = record      { элемент списка }  
        d : word;  
        s : string;  
        p : pnode;  
      end;
```

```
var beg : pnode; { указатель на начало списка }  
    i, key   : word;  
    s       : string;  
    option  : word;  
const text: array [1 .. n] of string =  
    ('one', 'two', 'three', 'four', 'five');
```

{ добавление элемента в конец списка }

```
procedure add(var beg : pnode; d : word; const s : string);
```

```
var p : pnode;      { указатель на создаваемый элемент }  
    t : pnode;      { указатель для просмотра списка }
```

```
begin
```

```
    new(p);          { создание элемента }
```

```
    p^.d := d; p^.s := s; { заполнение элемента }
```

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

```
    if beg = nil then beg := p { список был пуст }
```

```
    else begin        { список не пуст }
```

```
        t := beg;
```

```
        while t^.p <> nil do { проход по списку до конца }
```

```
            t := t^.p;
```

```
        t^.p := p; { привязка нового элемента к последнему }
```

```
    end
```

```
end;
```

{ ----- ПОИСК ЭЛЕМЕНТА ПО КЛЮЧУ ----- }

```
function find(beg : pnode; key : word; var p, pp : pnode) : boolean;  
begin
```

```
    p := beg;
```

```
    while p <> nil do begin
```

```
        if p^.d = key then begin
```

```
            find := true; exit end;
```

```
            pp := p;
```

```
            p := p^.p;
```

```
        end;
```

```
        find := false;
```

```
end;
```

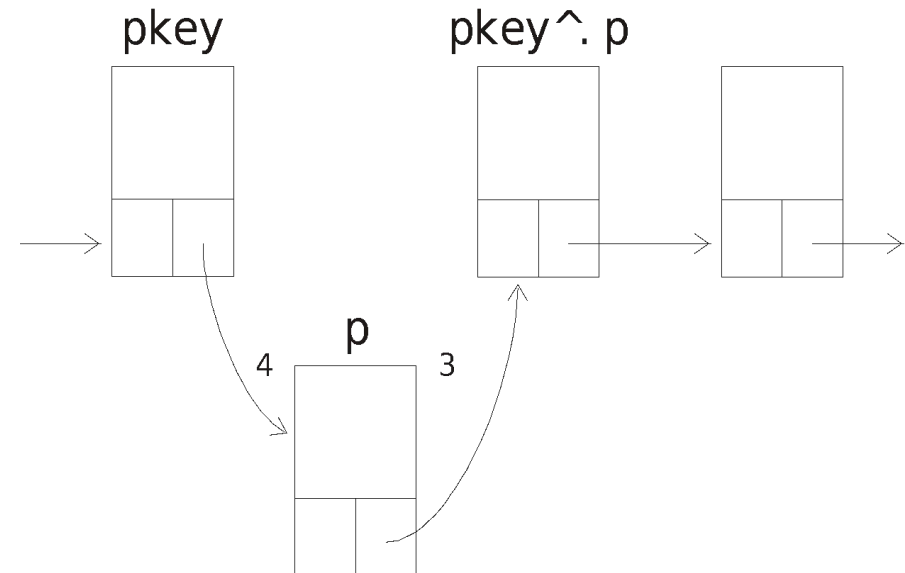
{ ----- ВСТАВКА ЭЛЕМЕНТА ----- }

```

procedure insert(beg : pnode; key, d : word; const s : string);
var p      : pnode;  { указатель на создаваемый элемент }
    pkey   : pnode;  { указатель на искомый элемент }
    pp     : pnode;  { указатель на предыдущий элемент }

begin
    if not find(beg, key, pkey, pp) then begin
        writeln(' вставка не выполнена');
        exit;
    end;
    new(p);           {1}
    p^.d := d; p^.s := s; {2}
    p^.p := pkey^.p;   {3}
    pkey^.p := p;      {4}
end;

```



{ ----- удаление элемента ----- }

```
procedure del(var beg : pnode; key : word);  
var p   : pnode;  { указатель на удаляемый элемент }  
    pp  : pnode;  { указатель на предыдущий элемент }  
begin  
    if not find(beg, key, p, pp) then begin  
        writeln(' удаление не выполнено'); exit; end;  
    if p = beg then  
        beg := beg^.p { удаление первого элемента }  
    else pp^.p := p^.p;  
    dispose(p);  
end;
```

{ ----- ВЫВОД СПИСКА ----- }

```
procedure print(beg : pnode);  
var p : pnode;  { указатель для просмотра списка }  
begin  
    p := beg;  
    while p <> nil do begin  { цикл по списку }  
        writeln(p^.d:3, p^.s);  { вывод элемента }  
        p := p^.p  { переход к следующему элементу списка }  
    end;  
end;
```



```
{ ----- главная программа ----- }
```

```
begin
```

```
  for i := 1 to 5 do add(beg, i, text[i]);
```

```
  while true do begin
```

```
    writeln('1 - вставка, 2 - удаление,  
           3 - вывод, 4 - выход');
```

```
    readln(option);
```

```
  case option of
```

```
    1: begin                                { вставка }
```

```
      writeln('Ключ для вставки?');
```

```
      readln(key);
```

```
      writeln('Вставляемый элемент?');
```

```
      readln(i); readln(s);
```

```
      insert(beg, key, i, s);
```

```
    end;
```

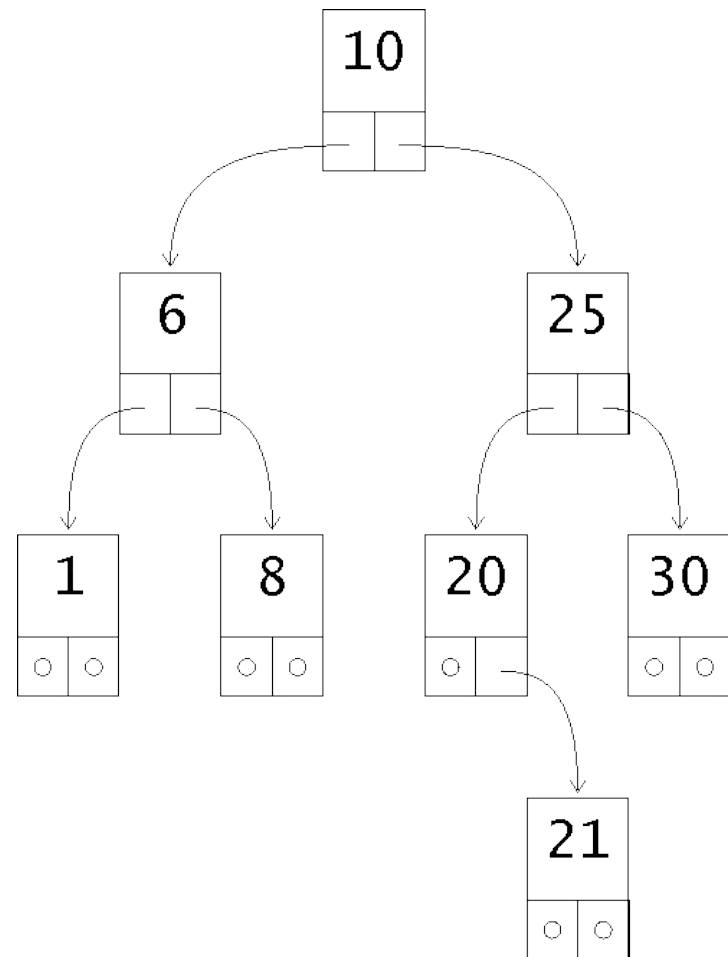
```
2: begin                                { удаление }
    writeln('Ключ для удаления?');
    readln(key);
    del(beg, key);
end;
3: begin                                { вывод }
    writeln('Вывод списка:');
    print(beg);
end;
4: exit;                                { выход }
    end
    writeln;
end
end.
```

Бинарное дерево

Бинарное дерево — динамическая структура данных, состоящая из узлов, каждый из которых содержит кроме данных не более двух ссылок на различные бинарные деревья.

На каждый узел имеется ровно одна ссылка.

Начальный узел называется *корнем* дерева.



Операции

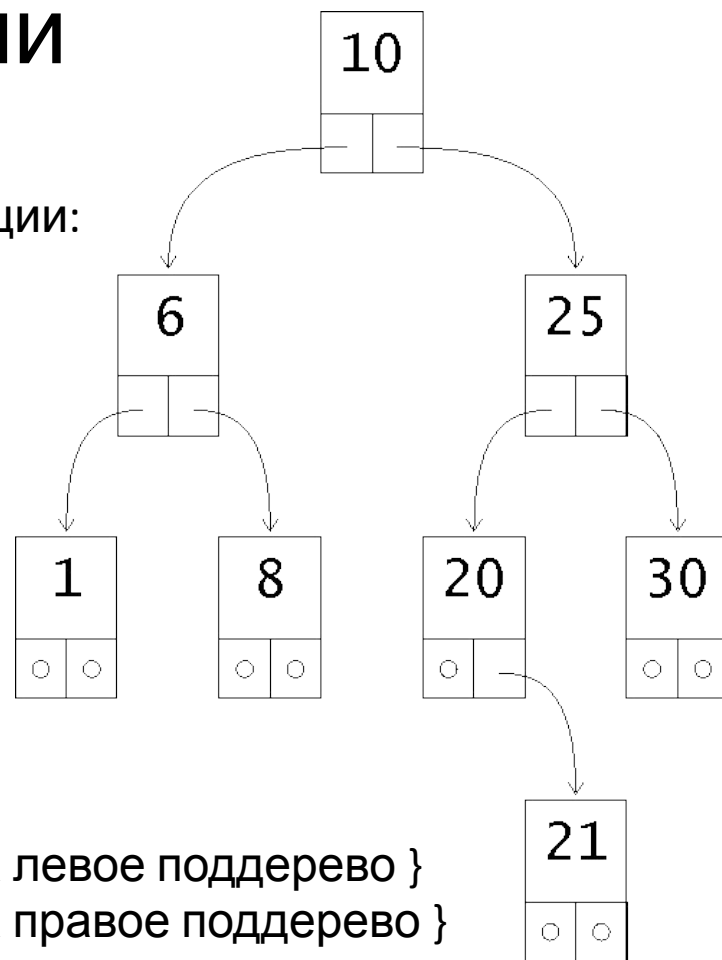
Для бинарных деревьев определены операции:

- включения узла в дерево;
- поиска по дереву;
- обхода дерева;
- удаления узла.

Элемент дерева:

```
type pnode = ^node;  
  node = record  
    data : word;  
    left : pnode;  
    right : pnode  
  end;
```

{ ключ }
{ указатель на левое поддерево }
{ указатель на правое поддерево }



Поиск по дереву

```
function find(root : pnode; key : word; var p,
              parent : pnode) : boolean;
```

```
begin
```

```
  p := root;    { поиск начинается от корня }
```

```
  while p <> nil do begin
```

```
    if key = p^.data then    { такой узел есть }
      begin find := true; exit end;
```

```
    parent := p; { запомнить ук-ль перед спуском }
```

```
    if key < p^.data
```

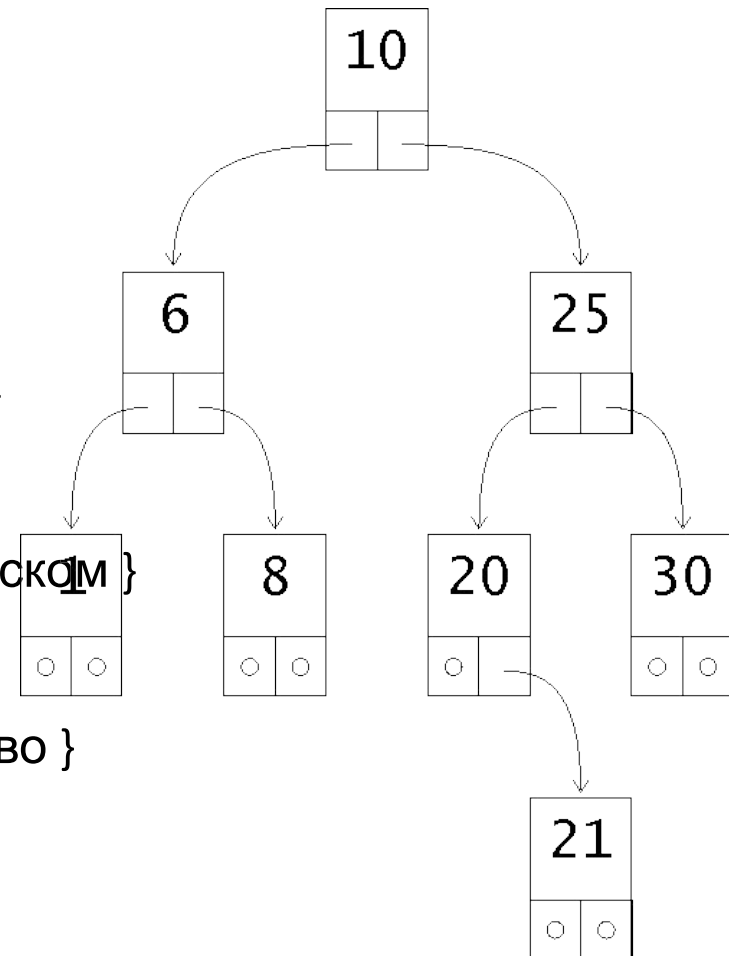
```
      then p := p^.left { спуститься влево }
```

```
      else p := p^.right; { спуститься вправо }
```

```
  end;
```

```
  find := false;
```

```
end;
```



Включение в дерево

```
procedure insert(var root : pnode; key : word);
```

```
var p, parent : pnode;
```

```
begin
```

```
  if find(root, key, p, parent) then begin
```

```
    writeln(' такой элемент уже есть'); exit; end;
```

```
  new(p);    { создание нового элемента }
```

```
  p^.data := key;
```

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

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

```
  if root = nil then root := p { первый элемент }
```

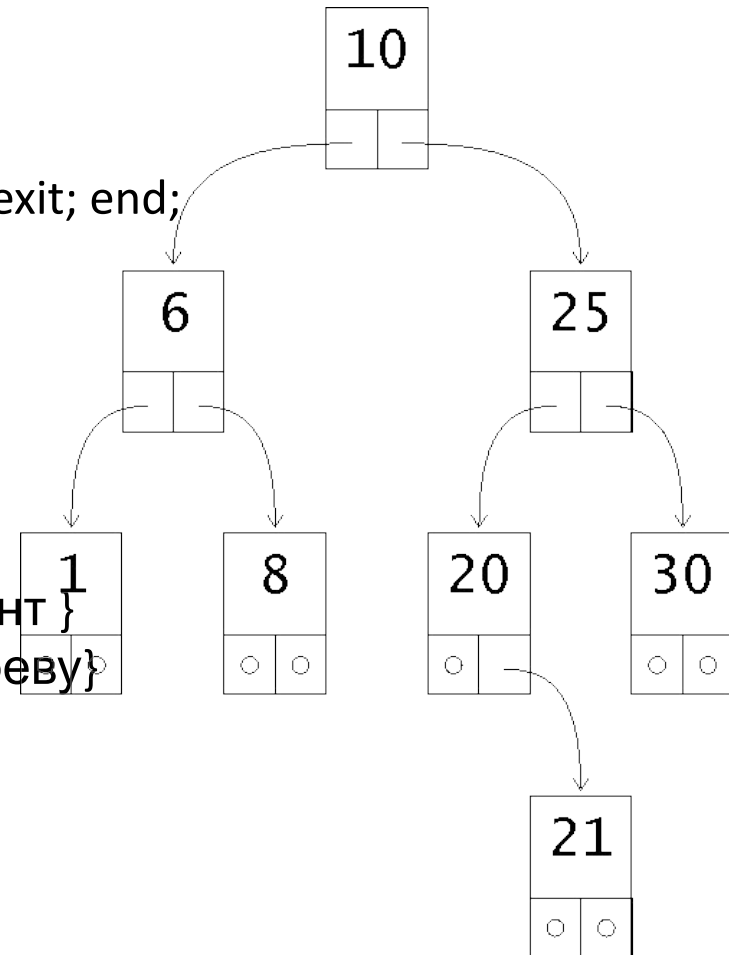
```
  else { присоед-е нового элемента к дереву }
```

```
    if key < parent^.data
```

```
    then parent^.left := p
```

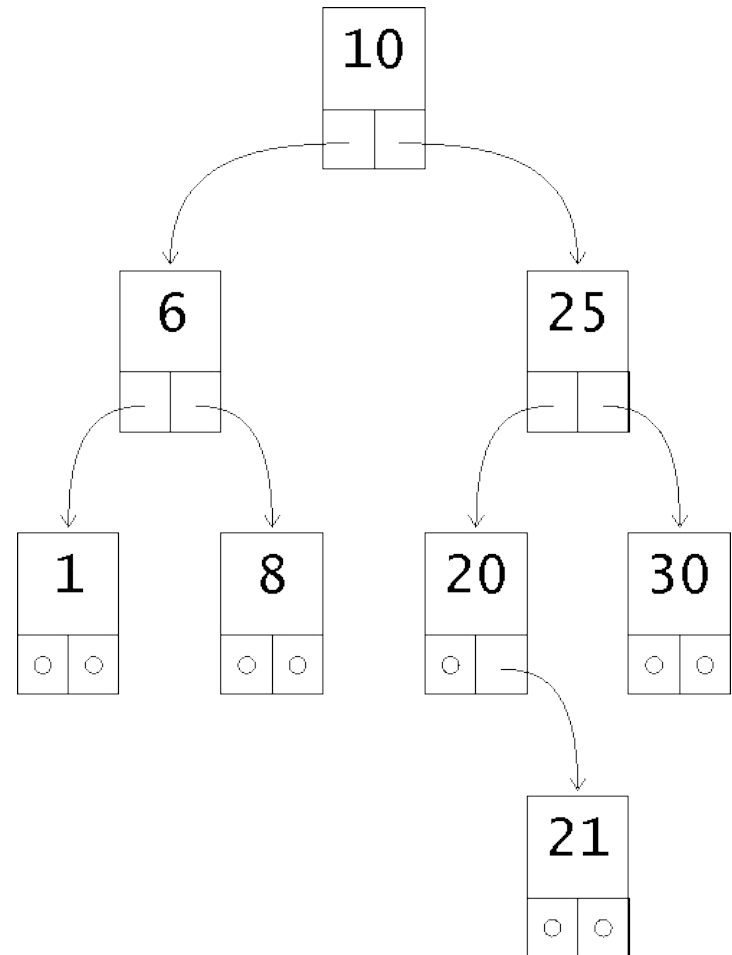
```
    else parent^.right := p;
```

```
end;
```



Обход дерева

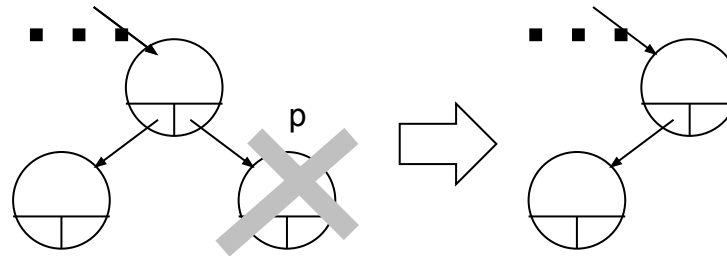
```
procedure print_tree( дерево );  
begin  
    print_tree( левое_поддерево )  
    посещение корня  
    print_tree( правое_поддерево )  
end;
```



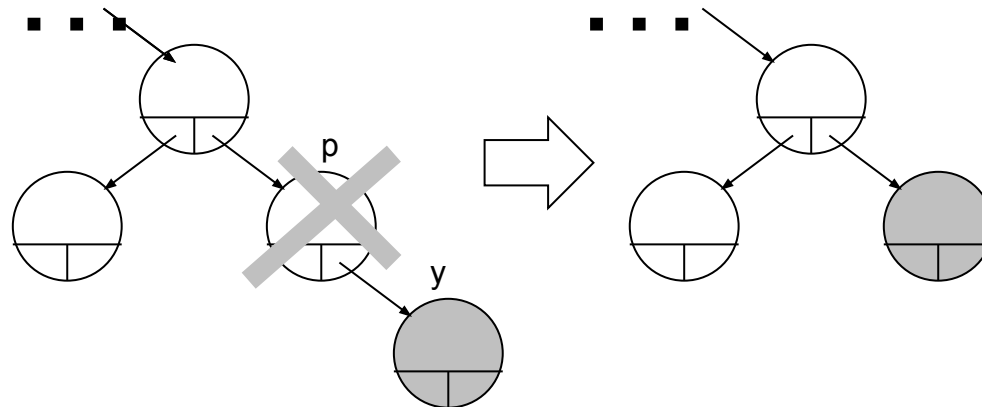
Удаление из дерева

1. Найти узел, который будет поставлен на место удаляемого.
2. Реорганизовать дерево так, чтобы не нарушились его свойства.
3. Присоединить новый узел к узлу-предку удаляемого узла.
4. Освободить память из-под удаляемого узла.

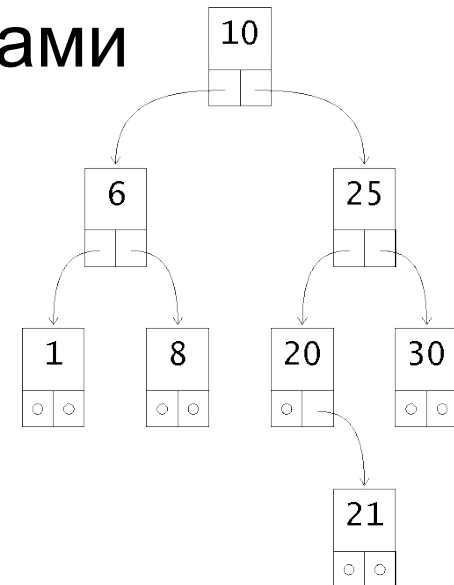
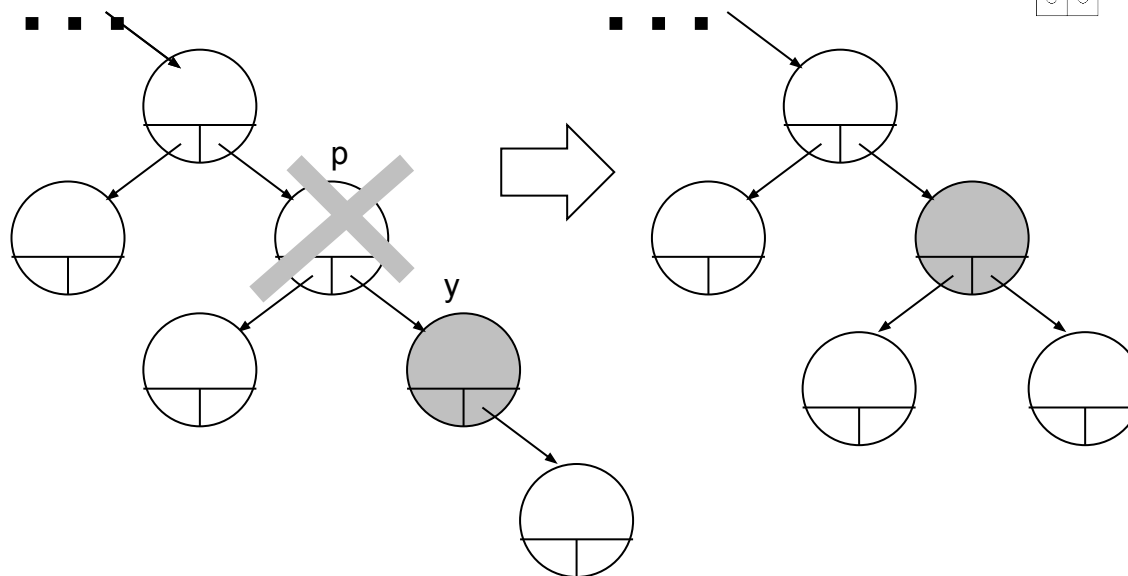
Удаление узла, не имеющего потомков



Удаление узла с одним потомком



Удаление узла с двумя потомками



Удаление узла (общий случай)

