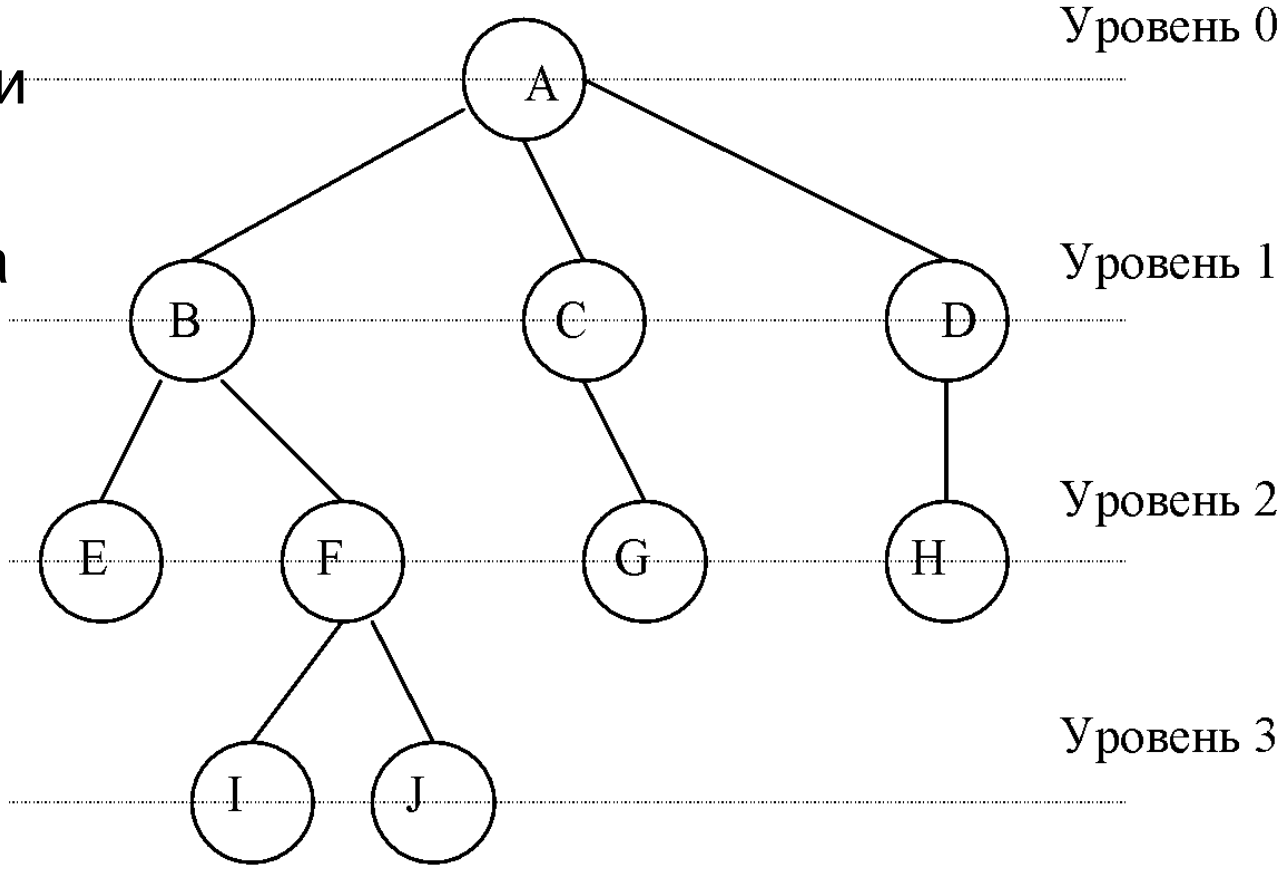
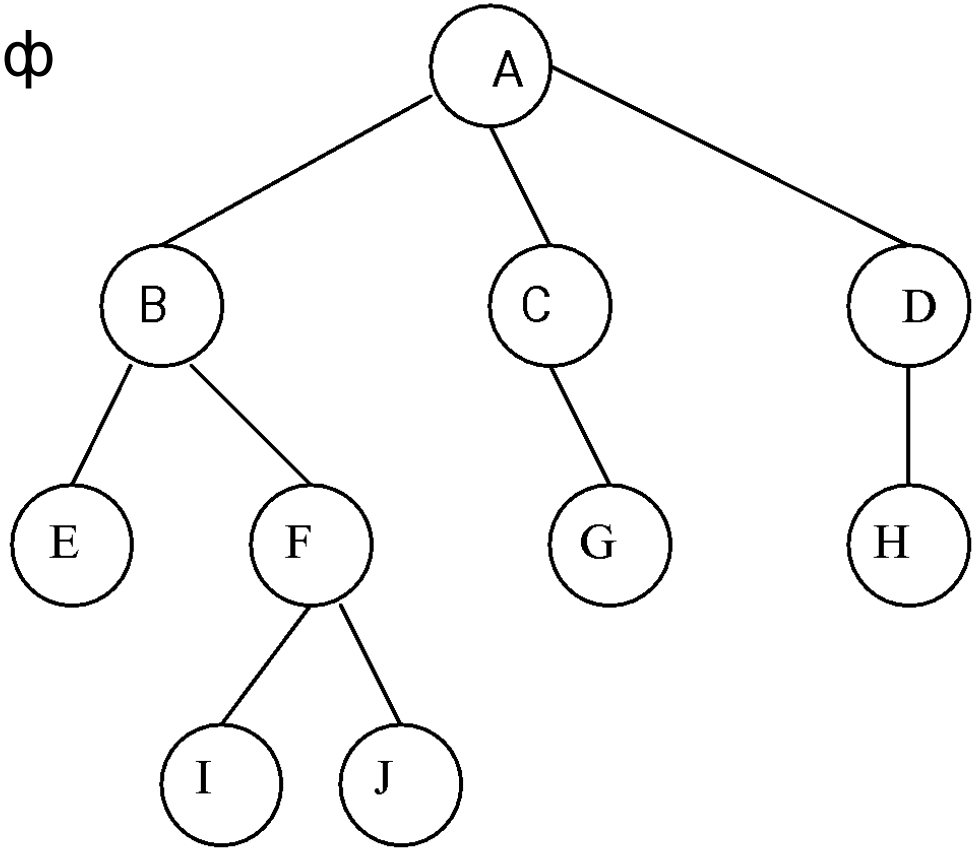
# Деревья

- Дерево
- Корень дерева
- Листья
- Пустое дерево
- Предки, потомки
- Уровень узла
- Глубина дерева



# *Способы изображения древовидных структур*

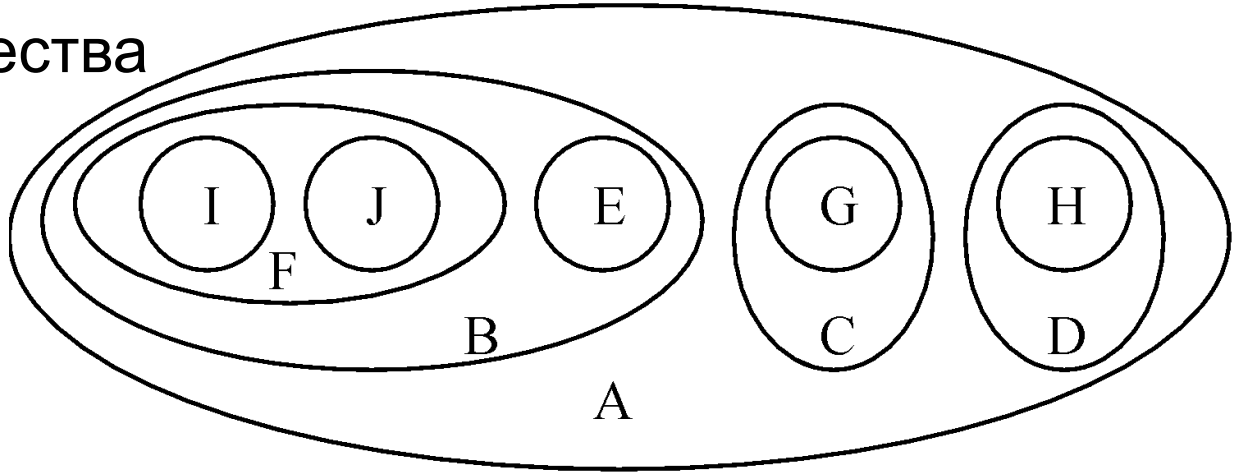
- Граф




- Вложенные скобки

(A(B(E,F(I,J)),C(G),D(H)))

- Вложенные множества




- Ломаная последовательность

A

B

E

F

I

J

C

G

D

H

# Двоичные (бинарные) деревья

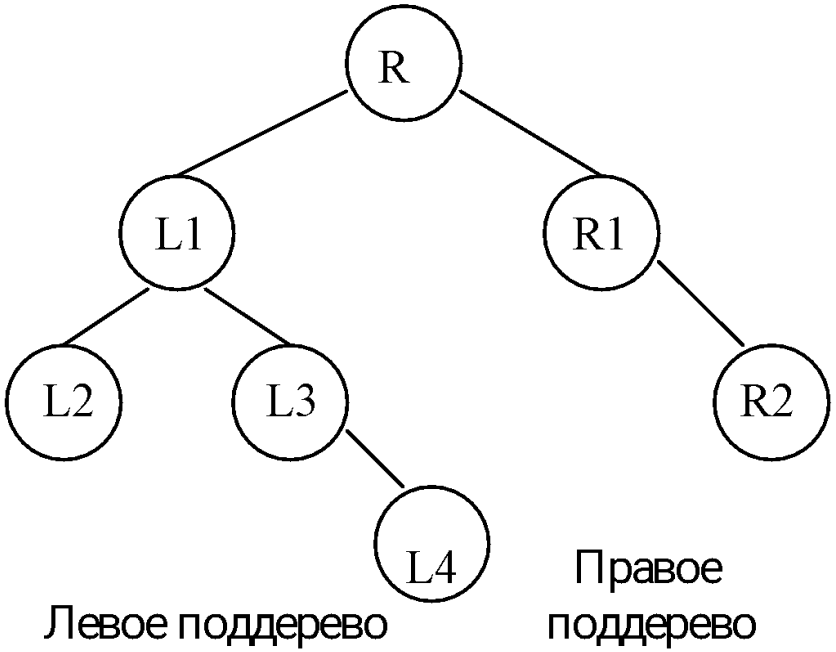
{R} - корневой узел

{L1, L2, ..., Lm} - левое поддерево R

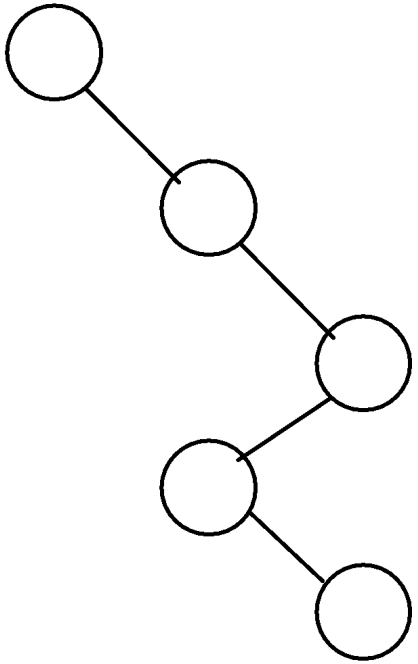
{R1, R2, ..., Rm} - правое поддерево R

n

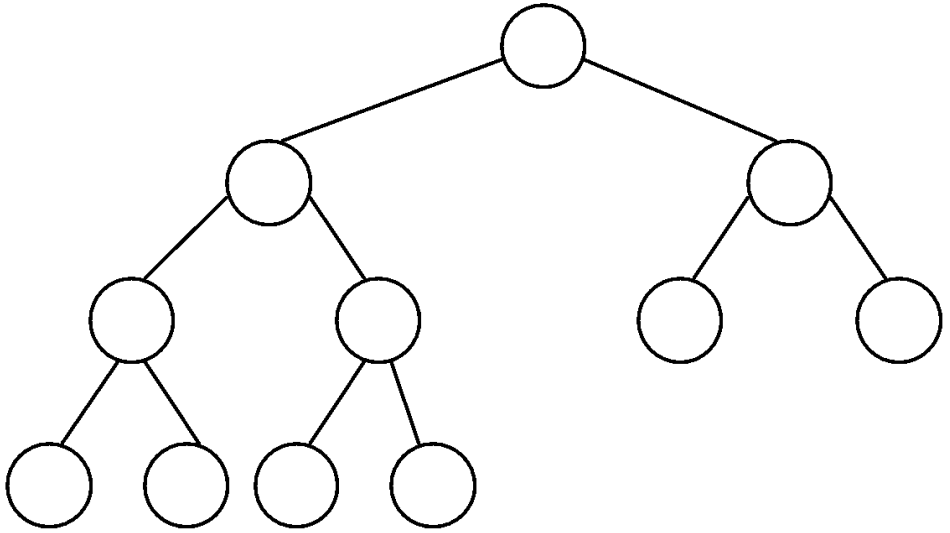
1 … $2^n$



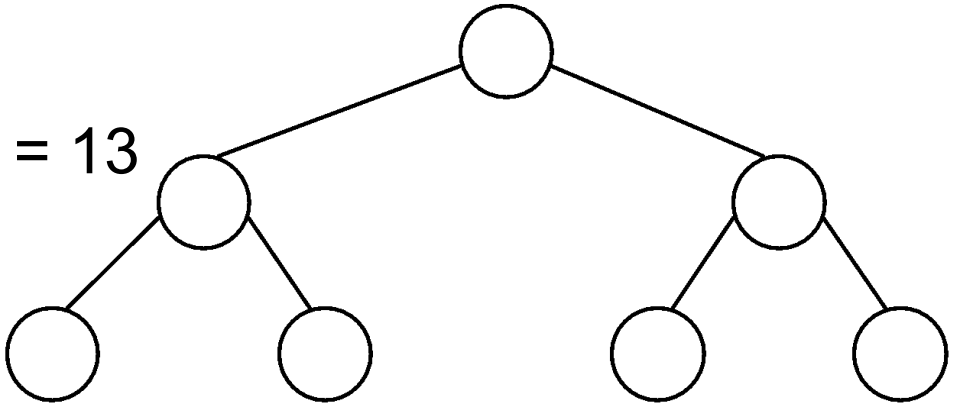
Вырожденное дерево
(глубина 4)



Законченное дерево
(Глубина 3)



Полное дерево
(Глубина 2)



N

глубина = int ($\log_2 N$).

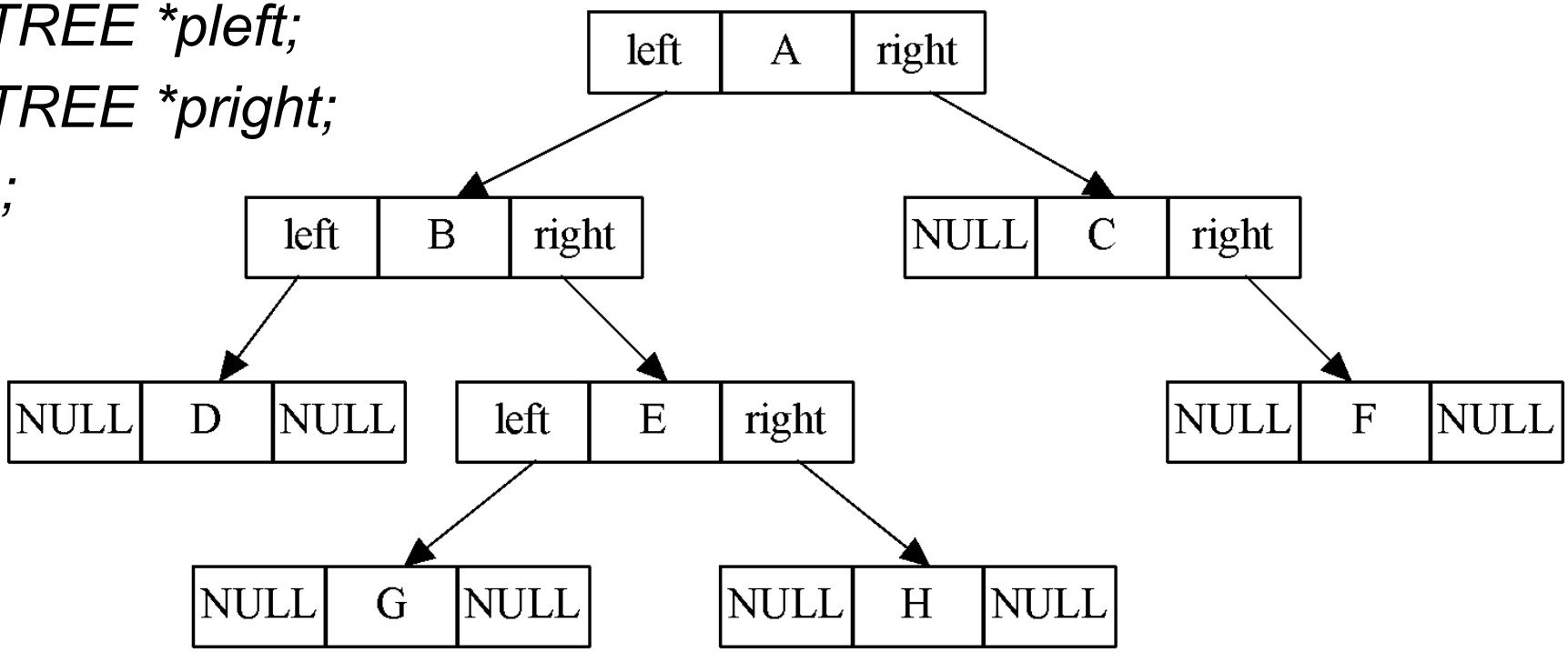
N=10 000

int(log210 000) = int(13.28) = 13

# *Структура двоичного дерева*

- С использованием массива
  *TREE[n];*
  *TREE[K] - TREE[2K+1], TREE[2K+2]*
- В виде динамической структуры

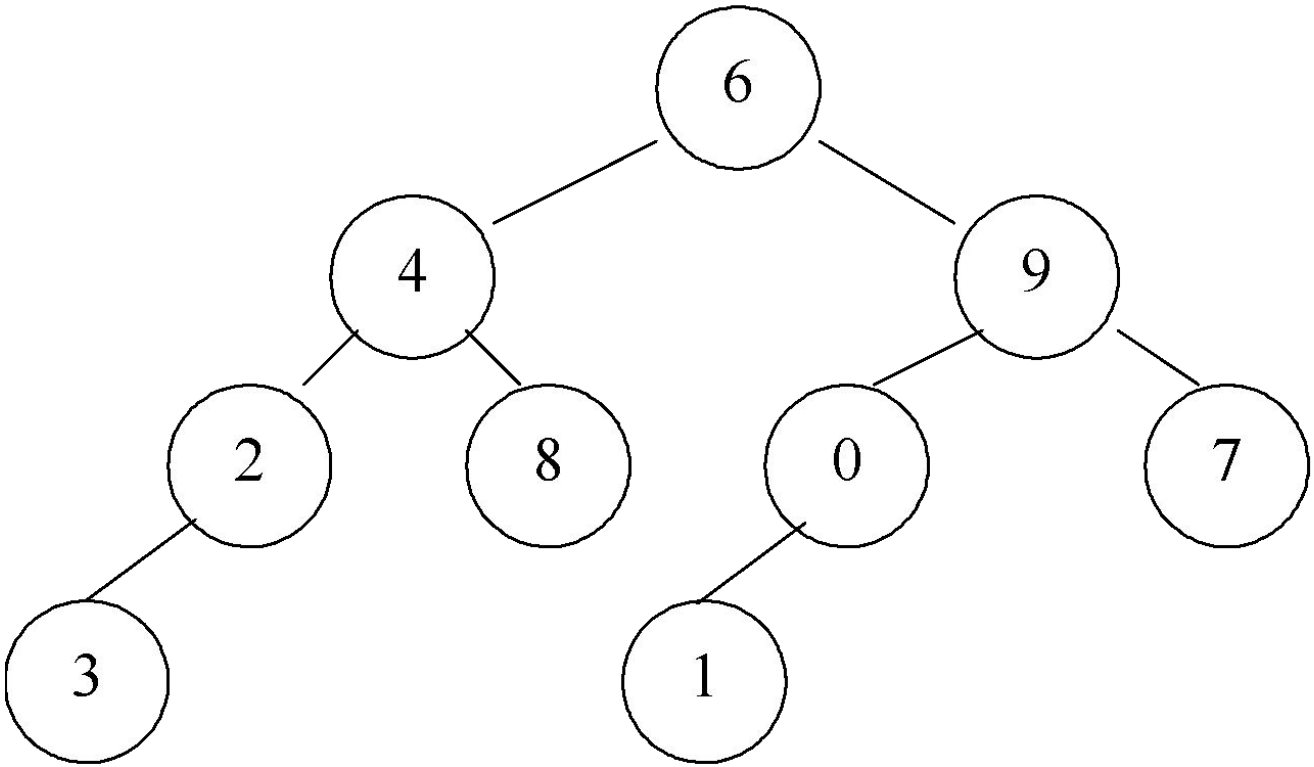
```
struct TREE{
  int dann;
  TREE *pleft;
  TREE *pright;
  };
```



# *Идеально сбалансированное дерево*

- Взять один узел в качестве корня.
- Левое поддерево: nl=n/2.
- Правое поддерево: nr=n-nl-1.

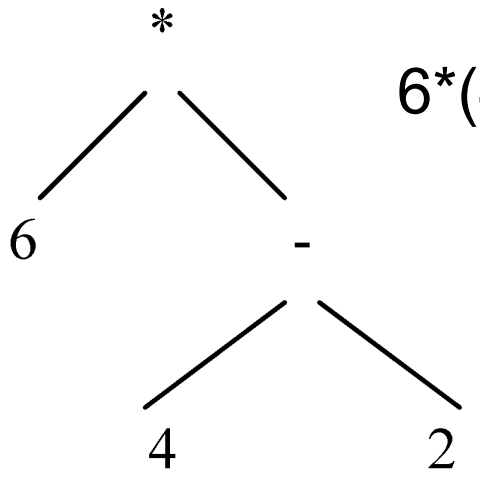
Пример: 6, 4, 2, 3, 8, 9, 0, 1, 7



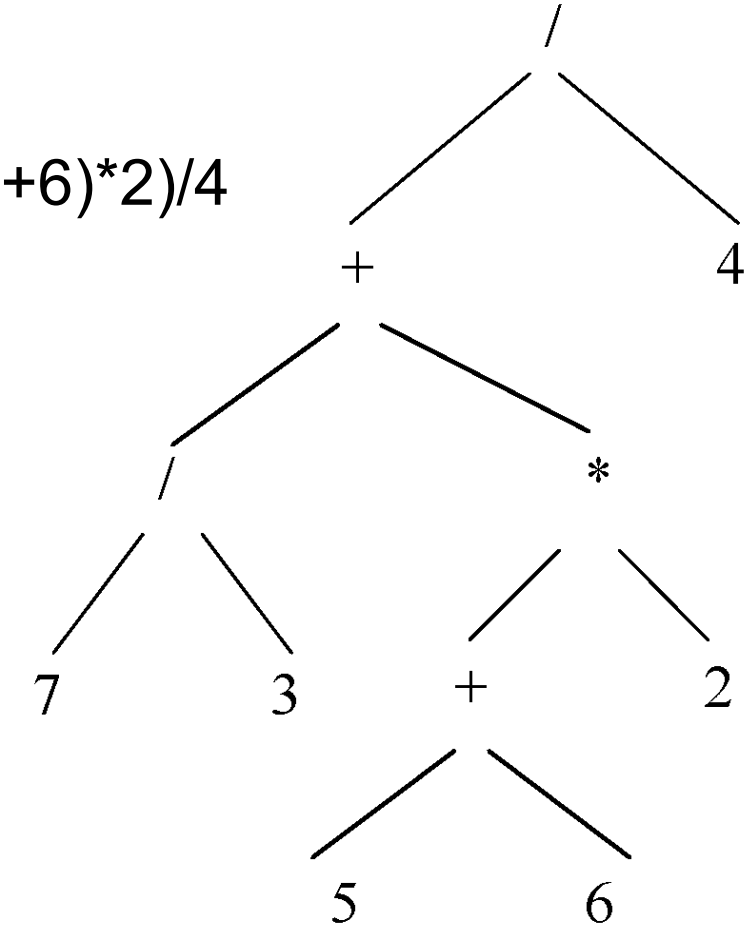
```
TREE* maketree(int n)
{TREE *ptr;
int nl, nr, x;
if (n==0) return NULL;
nl=n/2;
nr=n-nl-1;
ptr=new (TREE);
cout << "Input ";
cin >> ptr->dann;
ptr->pleft=maketree(nl);
ptr->pright=maketree(nr);
return ptr;
}

TREE *root;
root=maketree(n);
```

```
void print(TREE *ptr, int h)
{
if (ptr) {
  print(ptr->pright,h+1);
  for (int i=1;i<=h;i++) cout << "   ";
  cout << ptr->dann << endl;
  print(ptr->pleft,h+1);
  }
}
```

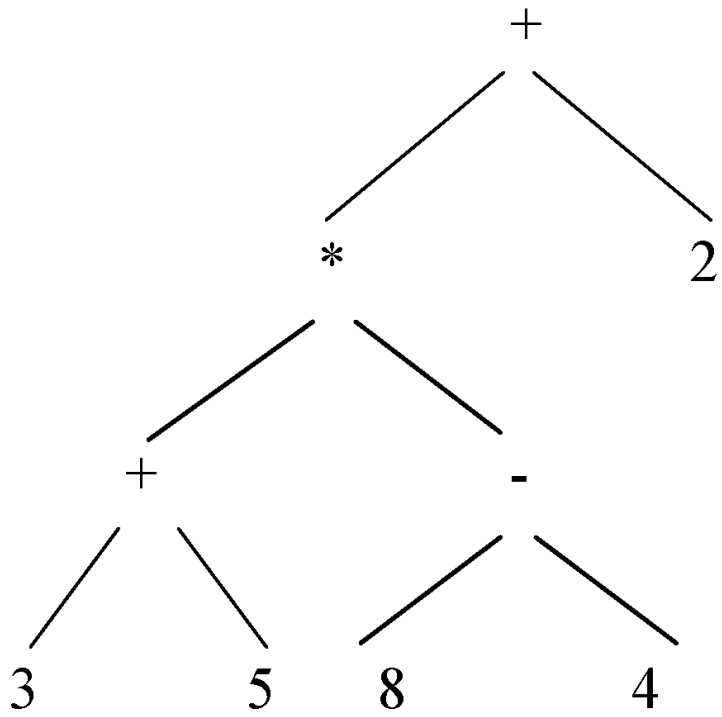
# *Двоичные деревья выражений*




6*(4-2)

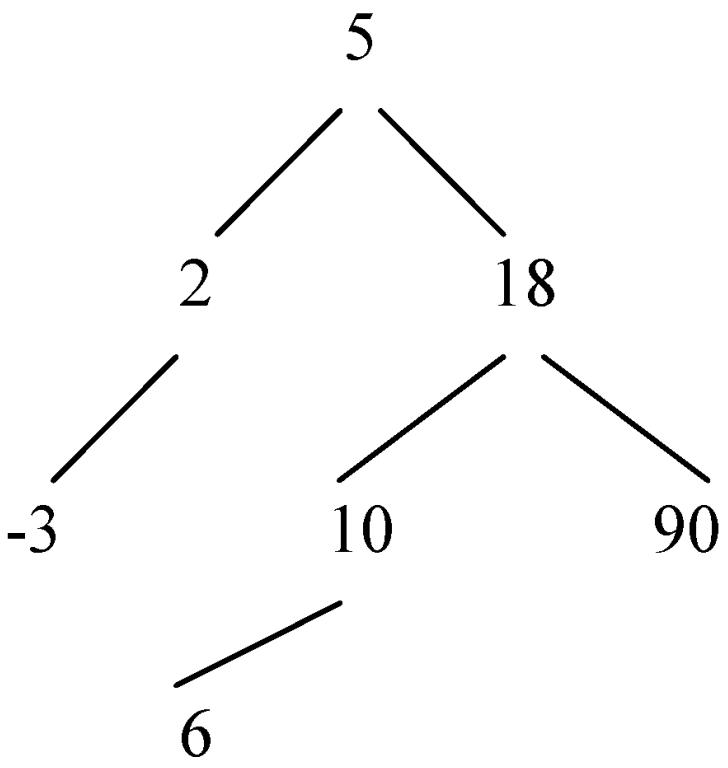


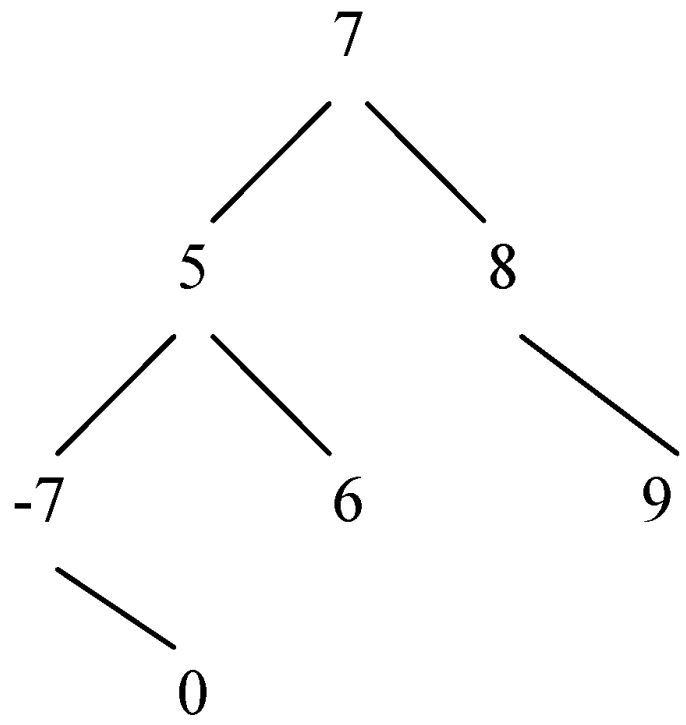

(7/3+(5+6)*2)/4




(3+5)*(8-4)+2

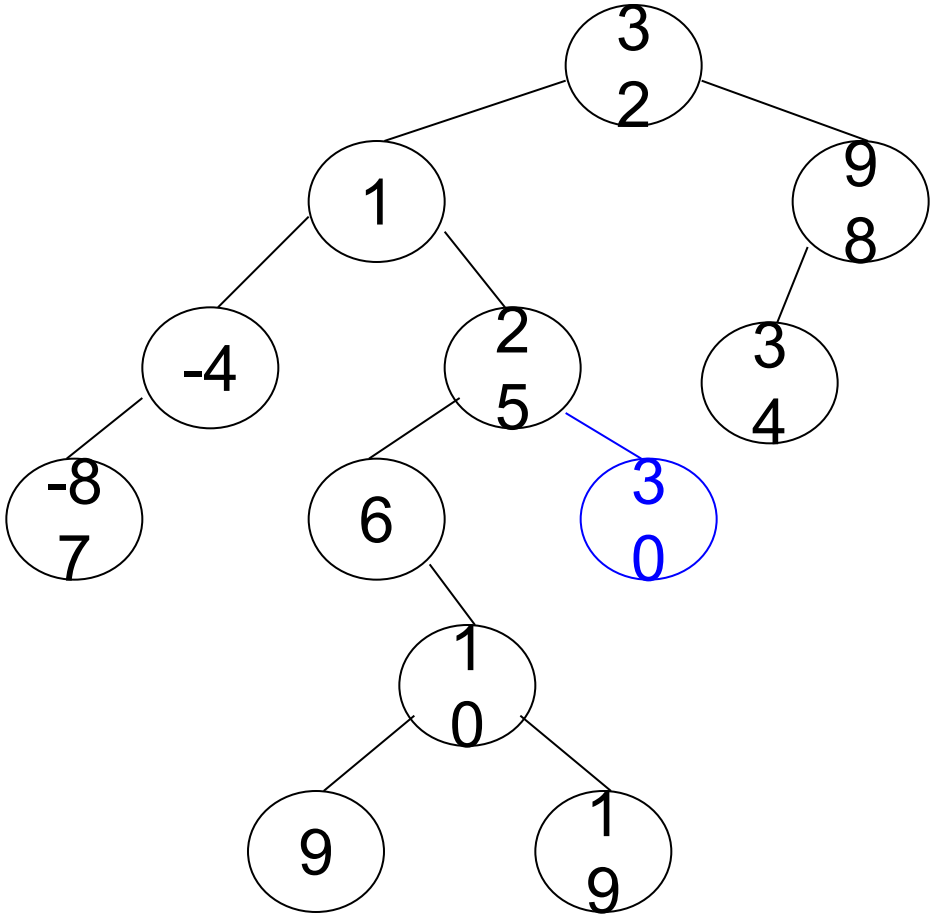
# *Деревья двоичного поиска*




O(N)  $O(\log_2 N)$

Пример: 32, 1, 98, -4, 34, -87, 25, 6, 10, 9, 19




30

```
void build(TREE *tt, char ch)
{
if (ch<tt->dann)
    if (tt->pleft==NULL)
       {left=new TREE;
        tt->pleft=left;
        left->dann=ch;
        left->pleft=NULL;
        left->pright=NULL;
        }
    else build(tt->pleft,ch);
else
    if (tt->pright==NULL)
       {right=new TREE;
        tt->pright=right;
        right->dann=ch;
        right->pleft=NULL;
        right->pright=NULL;
        }
    else build(tt->pright,ch);
}
```

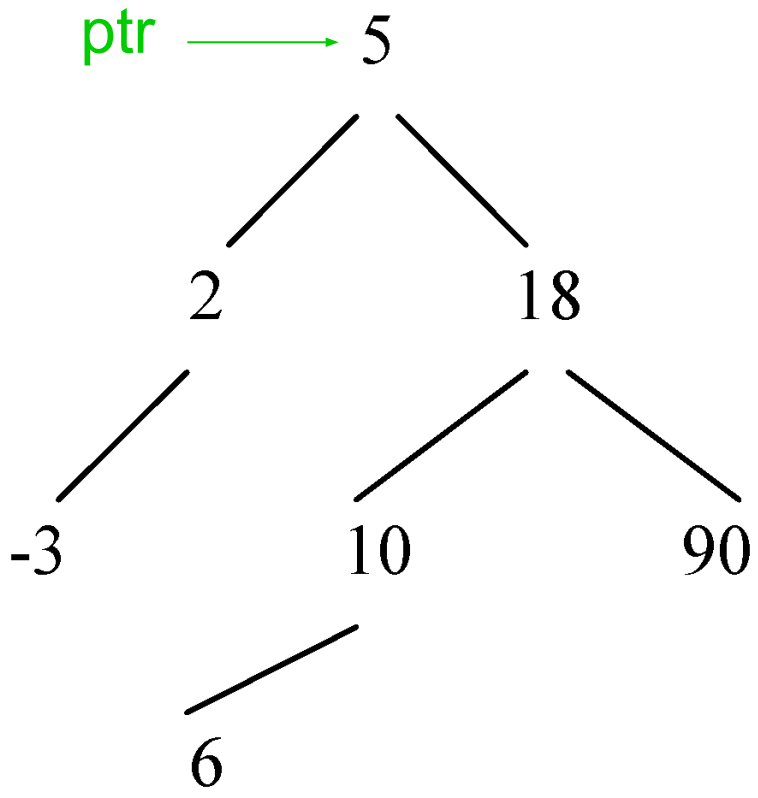
```
root=new TREE;
root->dann=str[0];
root->pright=NULL;
root->pleft=NULL;

for (int i=1; i<strlen(str); i++)
      build(root,*(str+i));
```

# Операции с двоичными деревьями

## *Алгоритмы поиска*

```
TREE * poisk(TREE *ptr, char ch)
{
  while(ptr->dann!=ch && ptr)
     if (ch<ptr->dann) ptr=ptr->pleft;
      else ptr=ptr->pright;
return ptr;
}
```

10



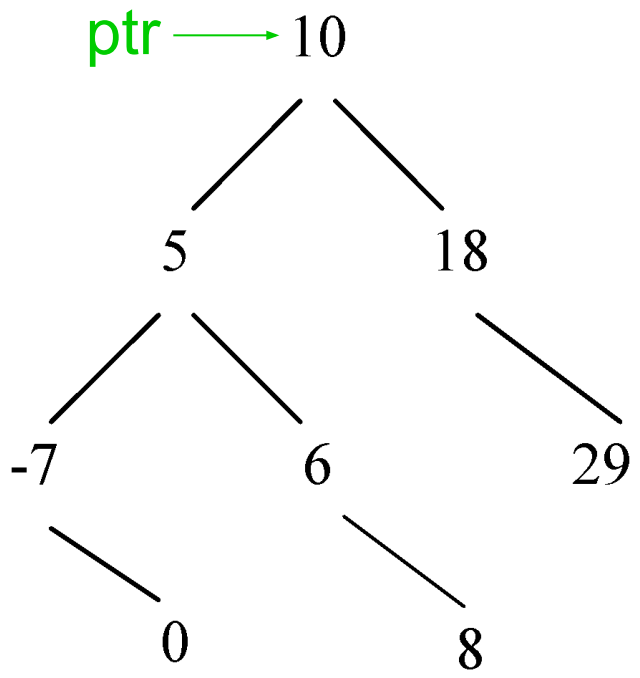
```
TREE * poisk(TREE *ptr,
    char ch,)
{
 if (ch<ptr->dann)
    ptr=poisk(ptr->pleft, ch);
 if (ch>ptr->dann)
    ptr=poisk(ptr->pright, ch);
 else return ptr;
}
```

```
TREE *parent=NULL;
TREE *find(char x, TREE* ptr)
{
while (ptr!=NULL)
{   if (x==ptr->dann)
     break;
   else
    {parent=ptr;
     if (x<ptr->dann)
      ptr=ptr->pleft;
     else ptr=ptr->pright;
     }
}
return ptr;

}
```

# *Алгоритмы обхода дерева*

- Прямой

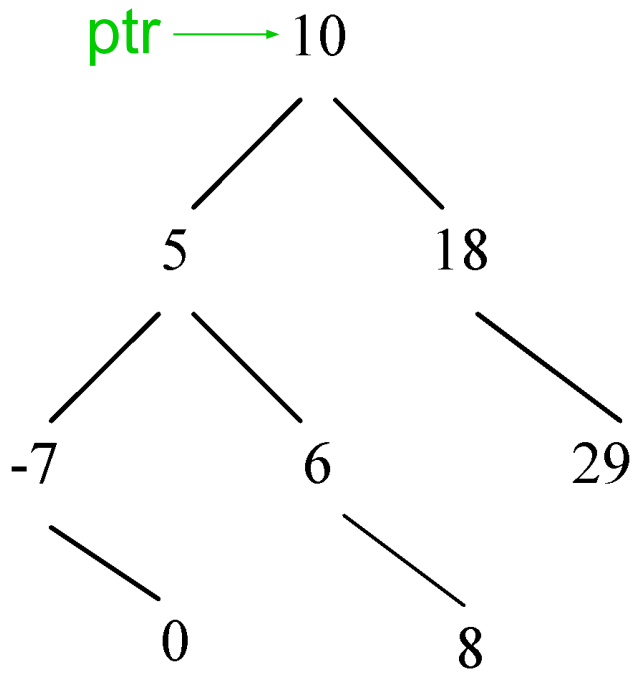
```
void preorder(TREE *ptr)
{
if (ptr) {
putchar(ptr->dann);
if (ptr->pleft) preorder(ptr->pleft);
if (ptr->pright)
    preorder(ptr->pright);
}
}
```




10 5 -7 0 6 8 18 29

- Симметричный

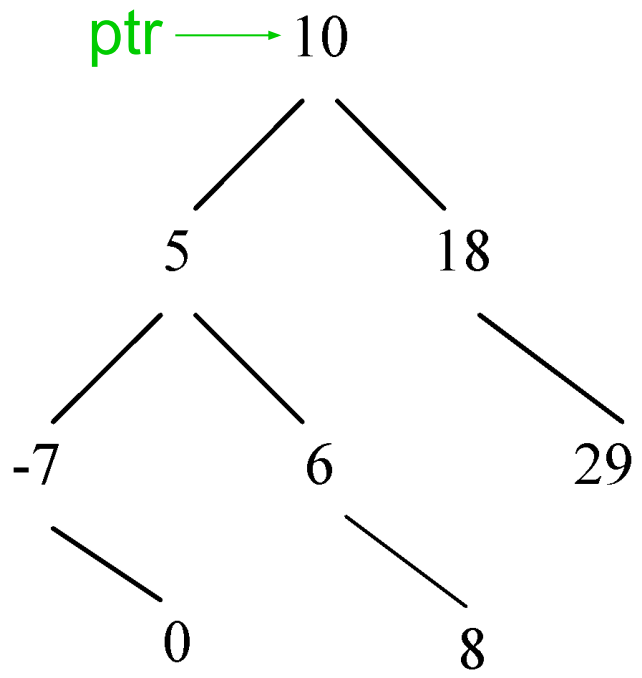
```
void inorder(TREE *ptr)
{
if (ptr) {
if (ptr->pleft) inorder(ptr->pleft);
putchar(ptr->dann);
if (ptr->pright) inorder(ptr->pright);
}
}
```




-7 0 5 6 8 10 18 29

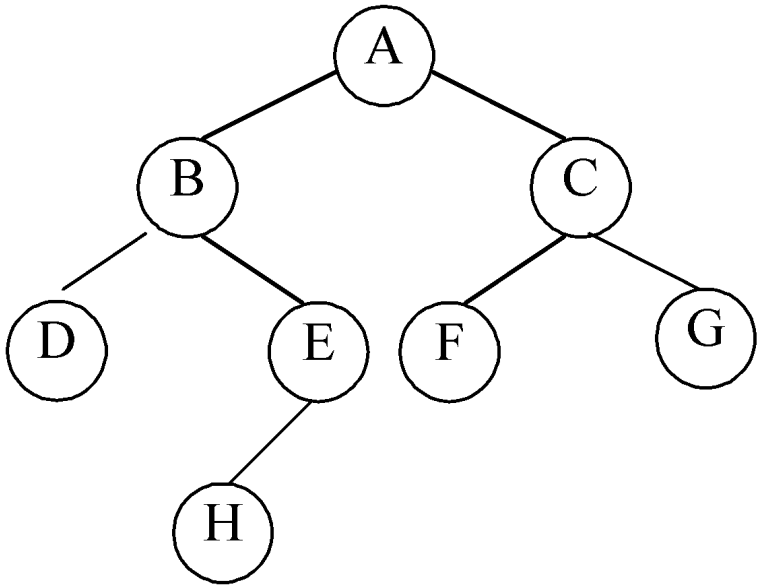
- Обратный

```
void postorder(TREE *ptr)
{
if (ptr) {
if (ptr->pleft) postorder(ptr->pleft);
if (ptr->pright) postorder(ptr->pright);
putchar(ptr->dann);
}
}
```




0 -7 8 6 5 29 18 10

# Вертикальная печать дерева

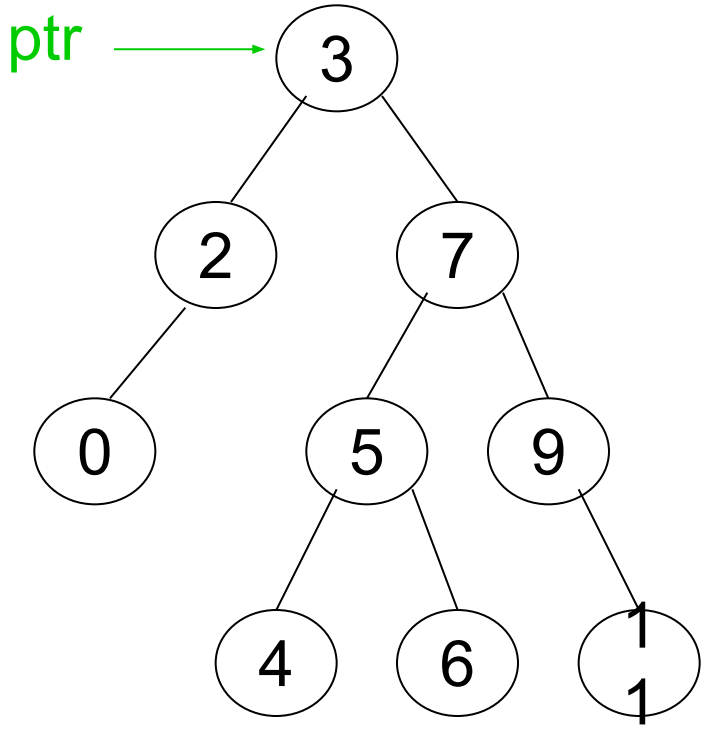



A B C D E F G H

| | | | | |
|---|---|---|---|---|
| A | | | | |
| B | C | | | |
| C | D | E | | |
| D | E | F | G | |
| E | F | G | | |
| F | G | H | | |
| G | H | | | |
| H | | | | |
| | | | | |

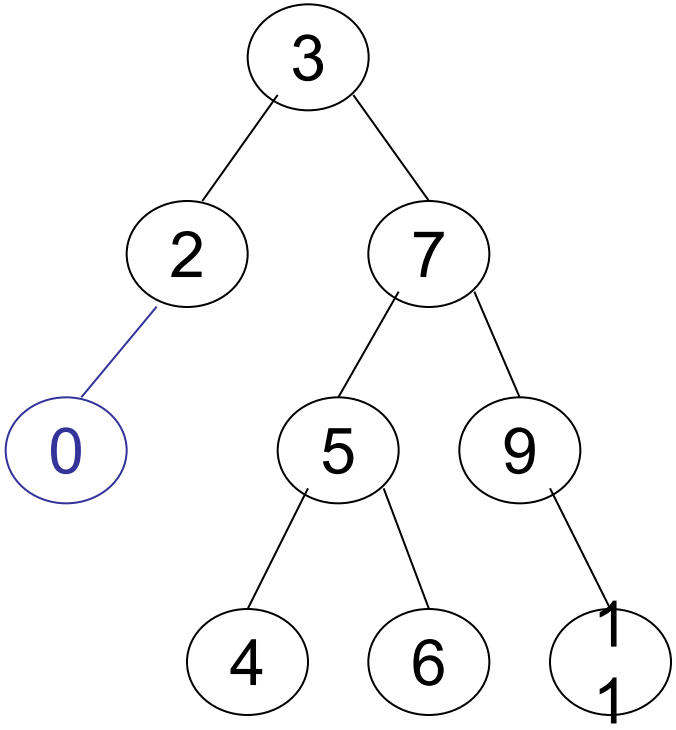
# Удаление дерева

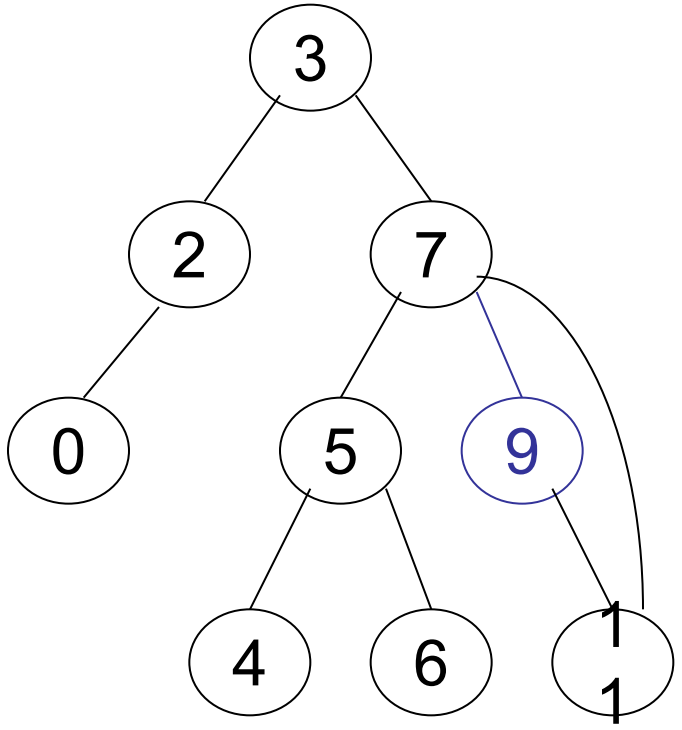
```
TREE* del(TREE *t)
{
if (t!=NULL)
{
  del(t->pleft);
  del(t->pright);
  delete t;
}
return NULL;
}

root=del(root);
```



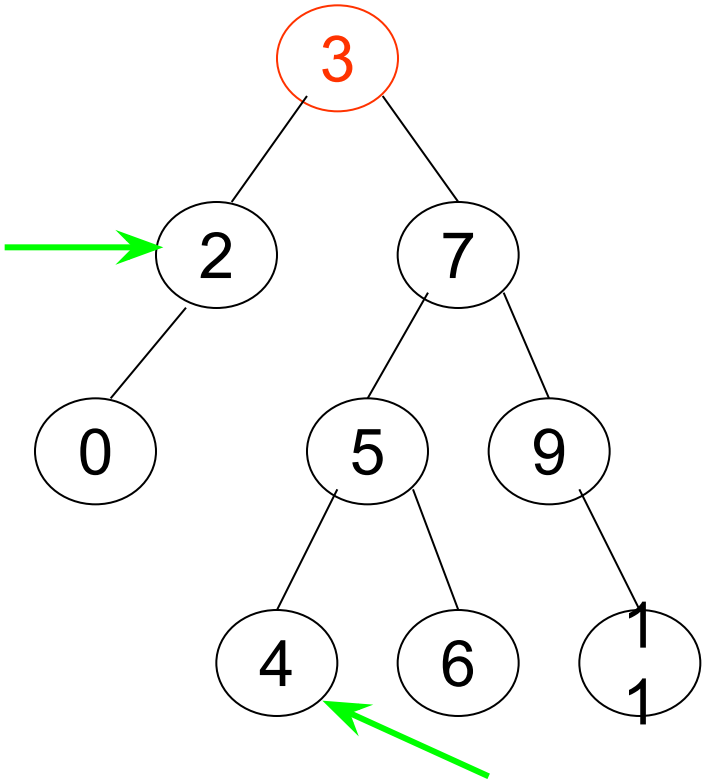
# *Удаление элемента из дерева*

- Элемента со значением, равным x, в дереве не существует;
- Элемент со значением x является терминальным узлом;
- Элемент со значением x имеет одного потомка;

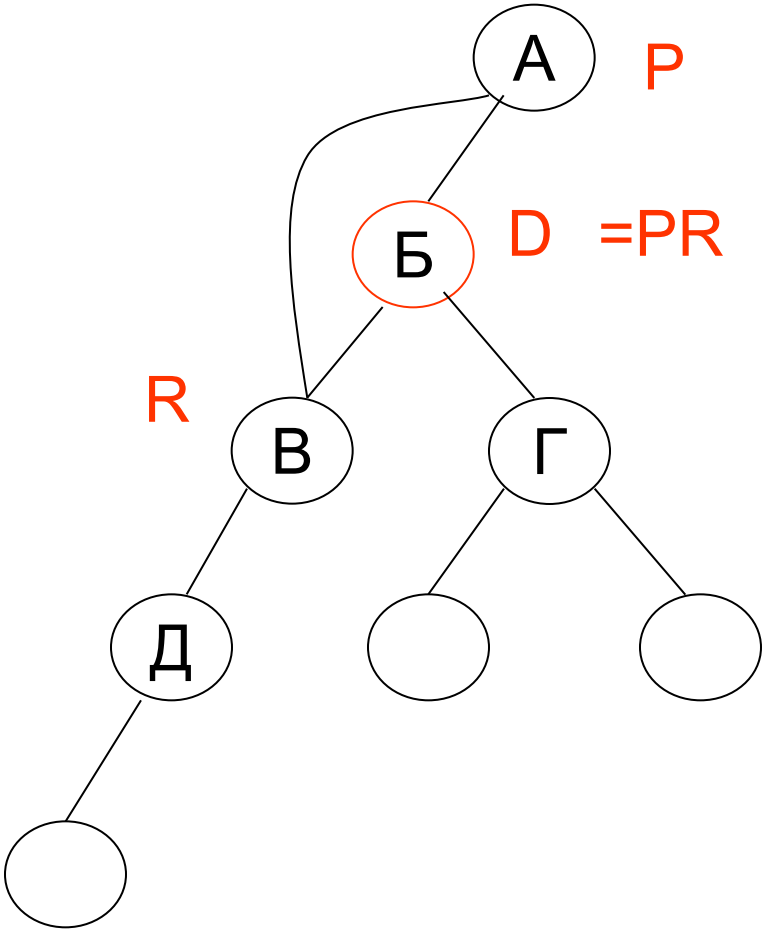


- Элемент со значением x имеет двух потомков.

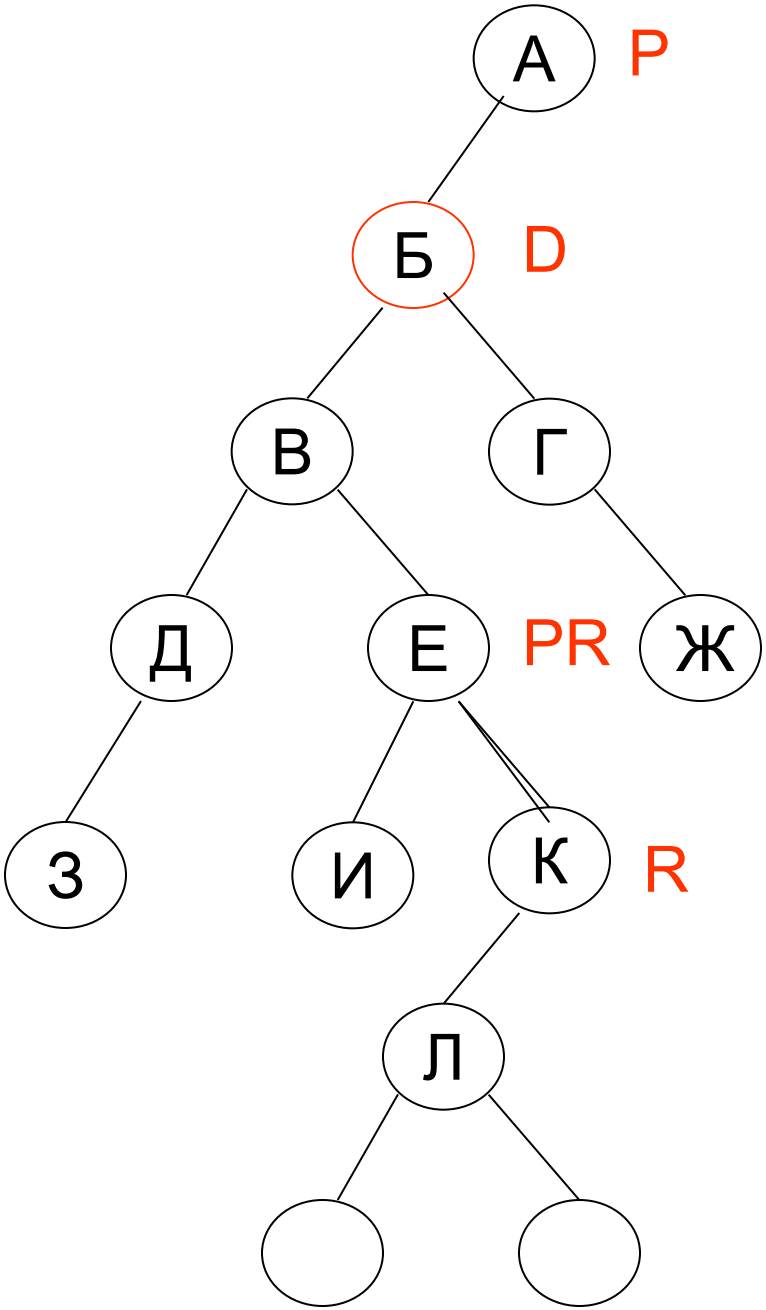



D
P
R
PR




R->right=D->right;

P->left=R;




PR->right=R->left;

# Двоичные деревья, представляемые массивами

A[i]

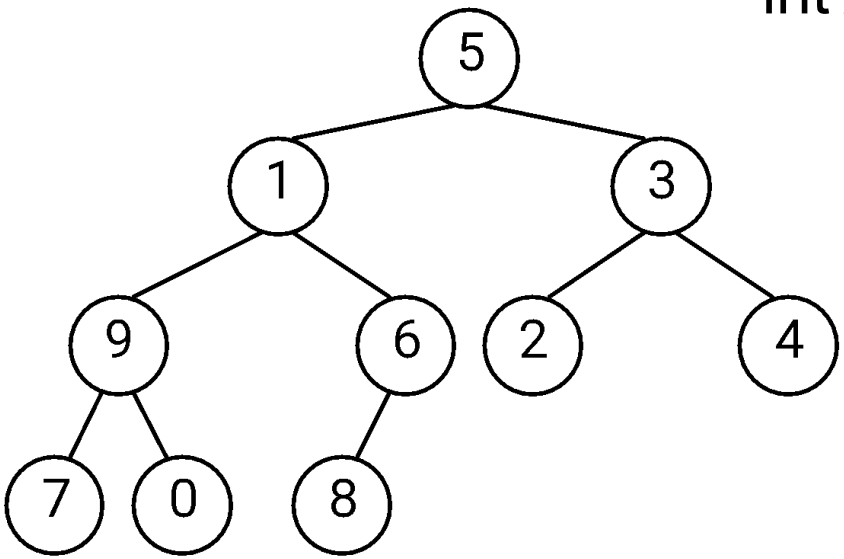
Индекс левого наследника = 2*i+1.

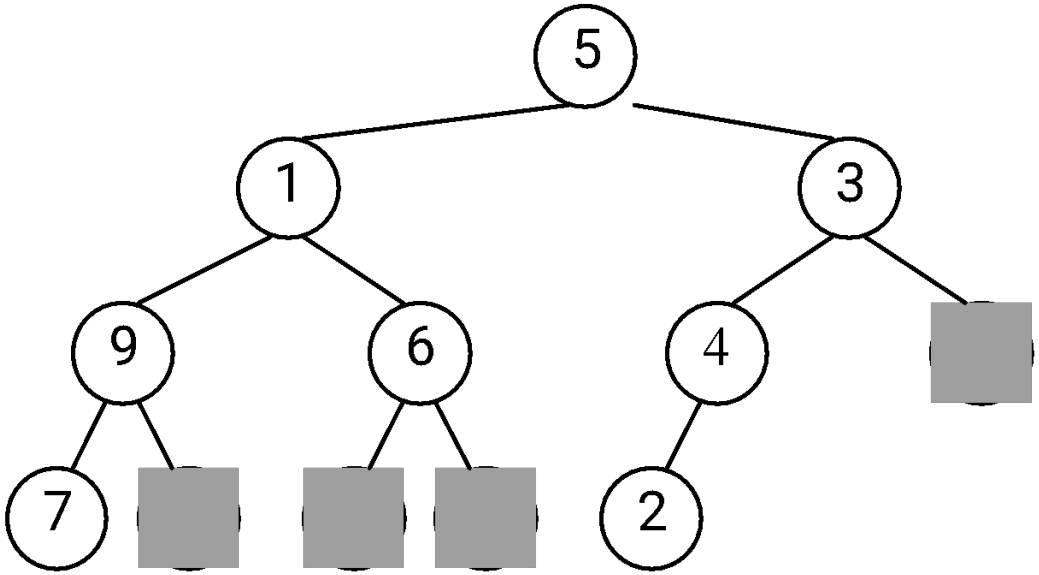
Индекс правого наследника = 2*i+2.

Индекс родителя = (i-1)/2.

Пример:

int A[]={5, 1, 3, 9, 6, 2, 4, 7, 0, 8};





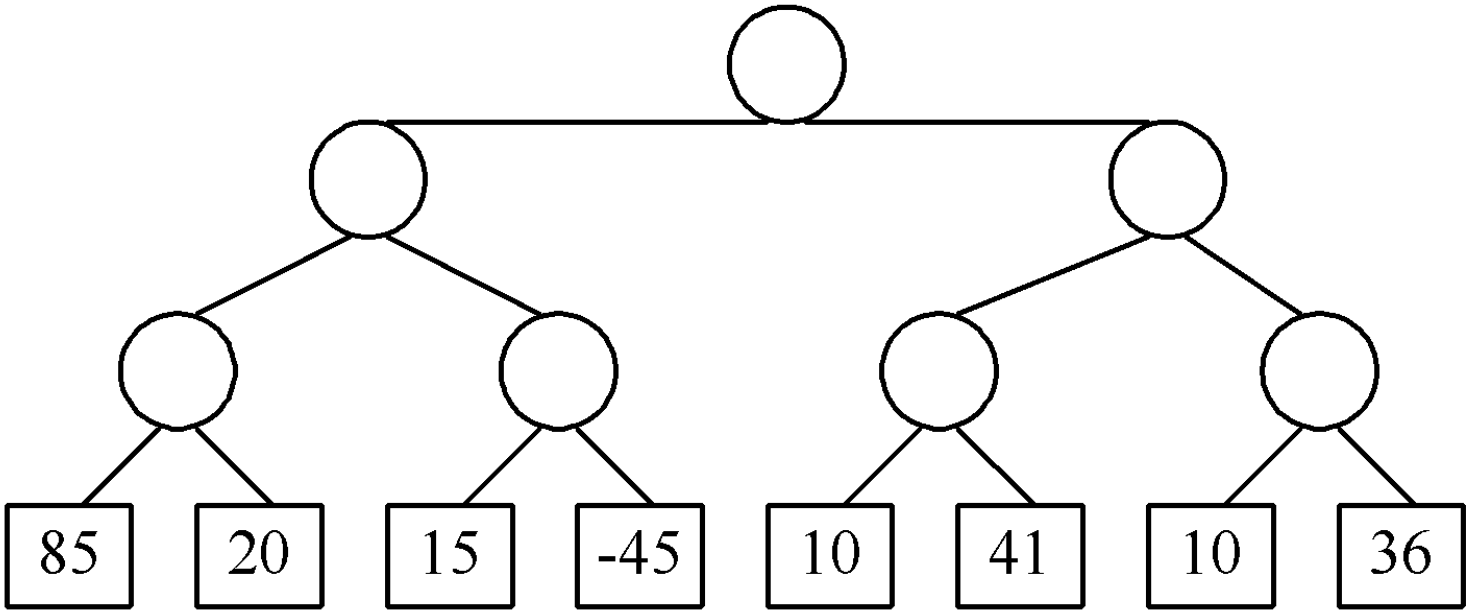

int A[]={5, 1, 3, 9, 6, 4, #, 7, #, #, #, 2};

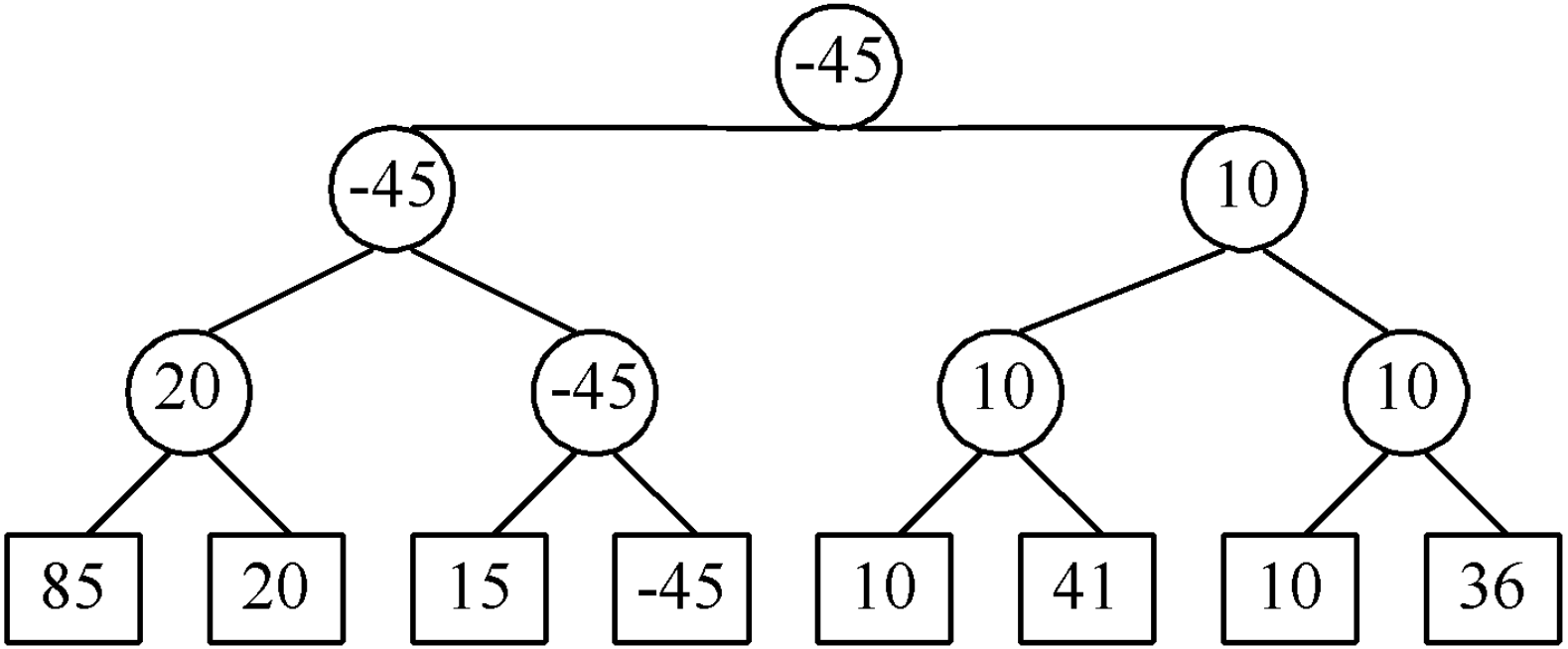
# *Турнирная сортировка*

A[8]={85, 20, 15, -45, 10, 41, 10, 36}

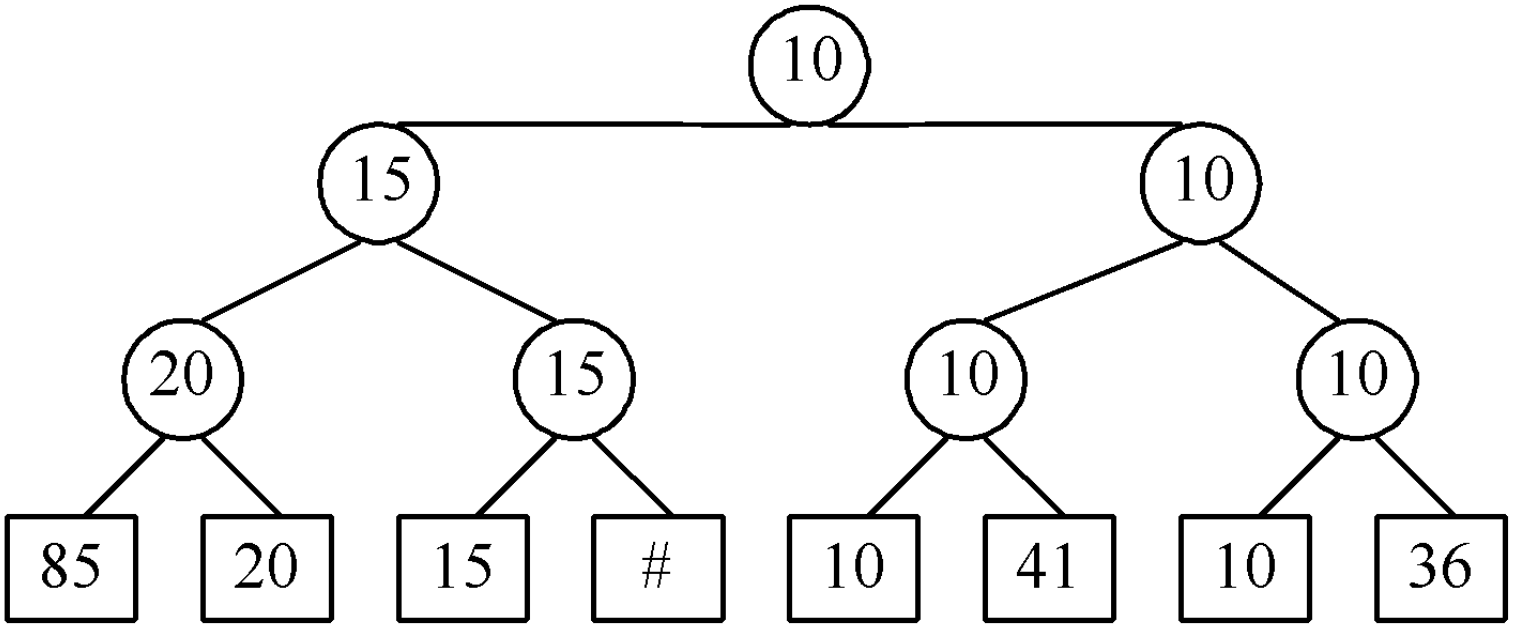
$2^k \geq N$

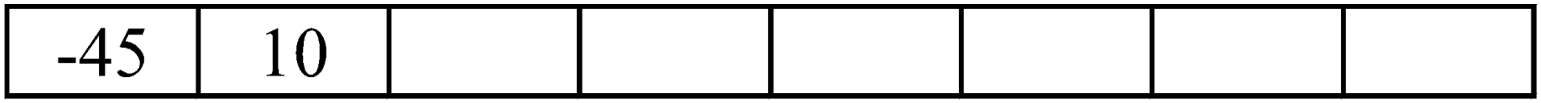
k=3

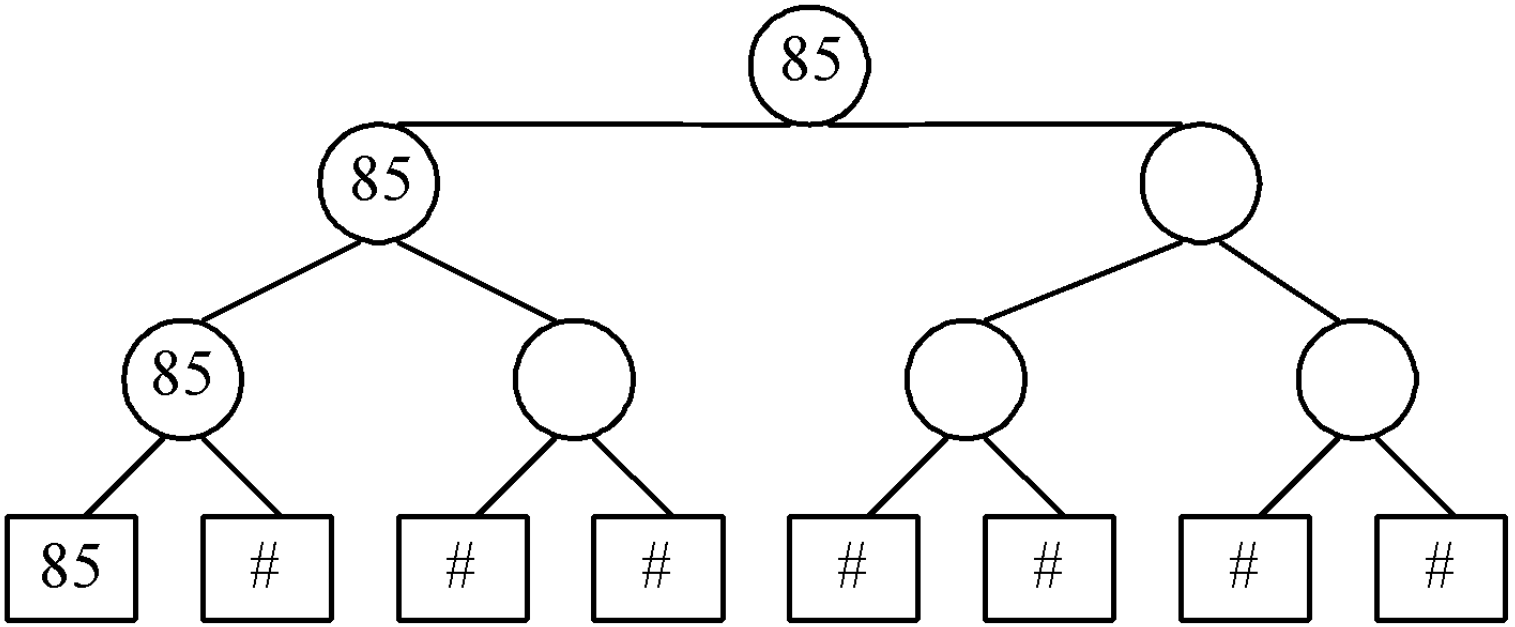



-45
-45
10
20
-45
10
10
85
20
15
-45
10
41
10
36


-45


10
15
10
20
15
10
10
85
20
15
#
10
41
10
36


-45
10

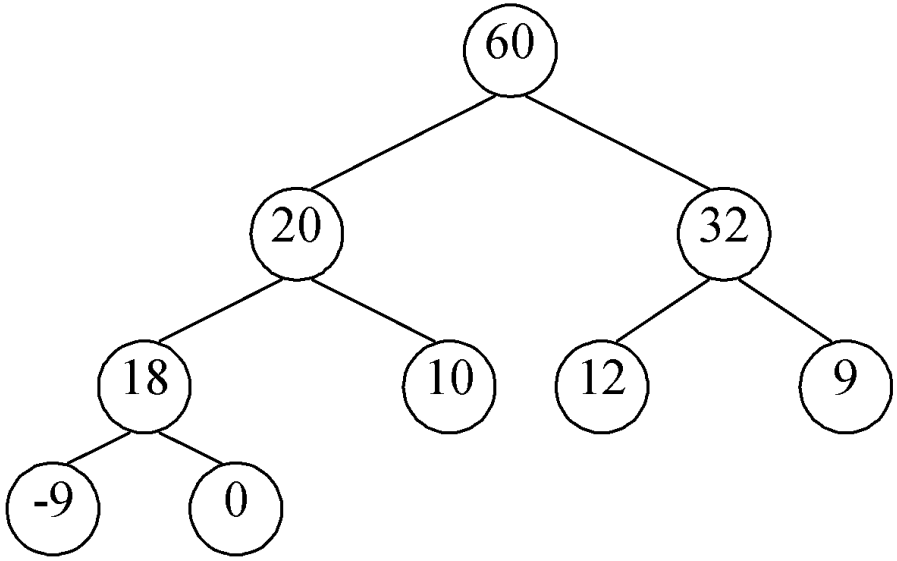



| -45 | 10 | 10 | 15 | 20 | 36 | 41 | |
|---|---|---|---|---|---|---|---|

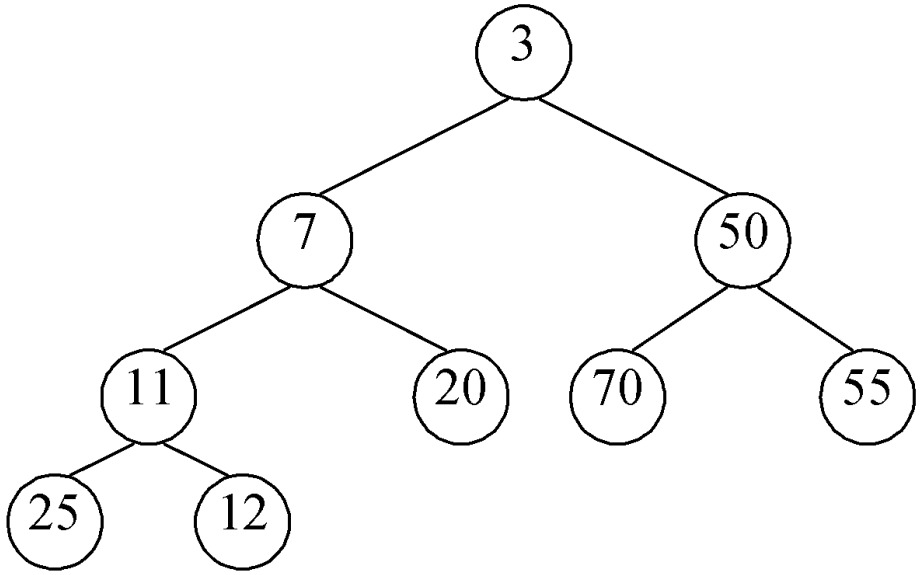
$O(n \log_2 n)$

# *Пирамиды (heap-tree)*

Максимальные




Минимальные



# *Преобразование массива в пирамиду*

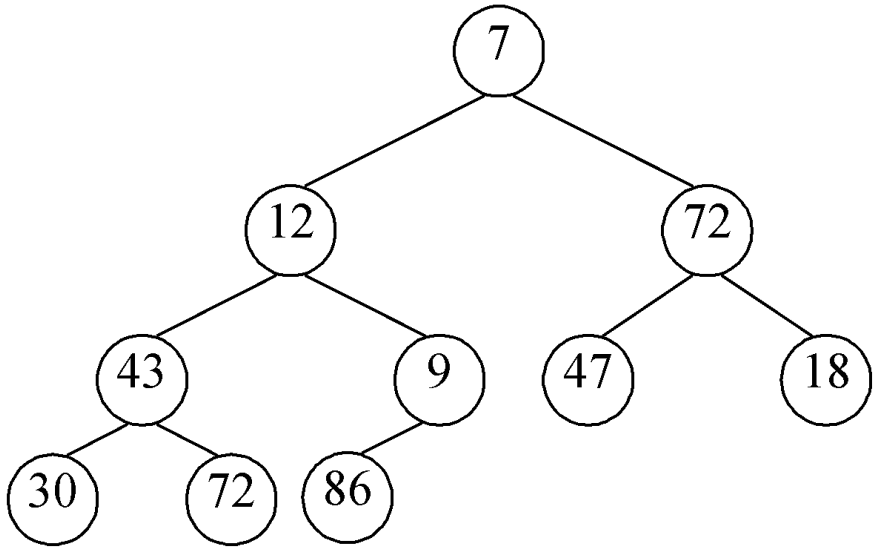
n n-1 (n-2)/2

Пример: Построим максимальную пирамиду

int H[10]={7, 12, 72, 43, 9, 47, 18, 30, 72, 86}

5, 6, …, 9

4, 3, …, 0



H[4]=9

H[9]=86

H[3]=43

H[8]=72  H[7]=30

H[2]=72

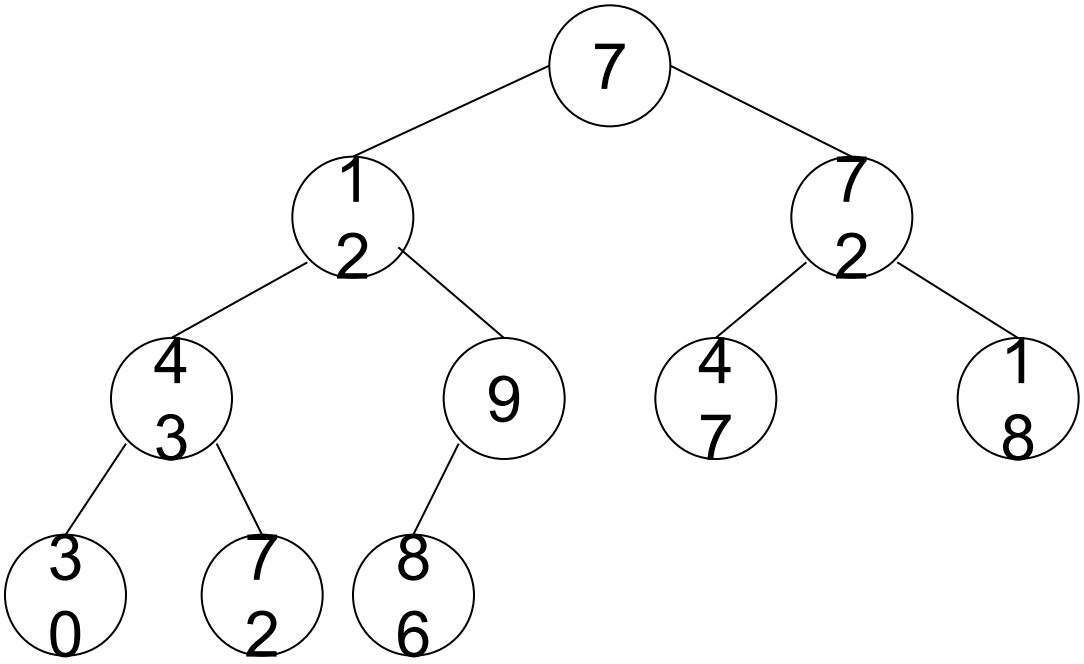
H[5]=47  H[6]=18

H[1]=12

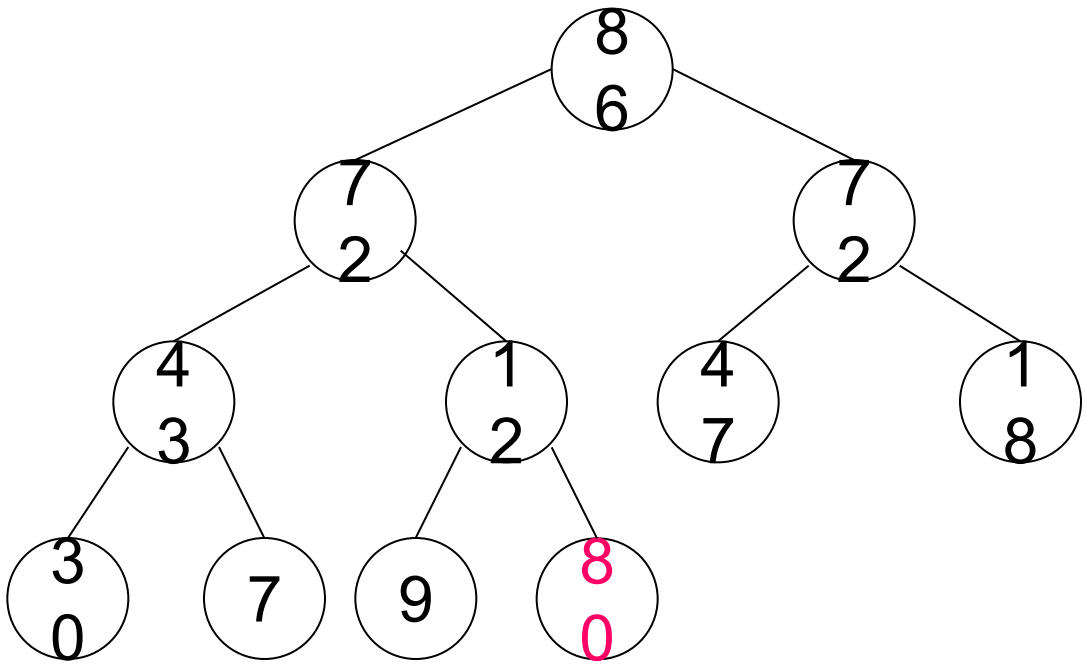
H[3]=72  H[4]=86

H[0]=7

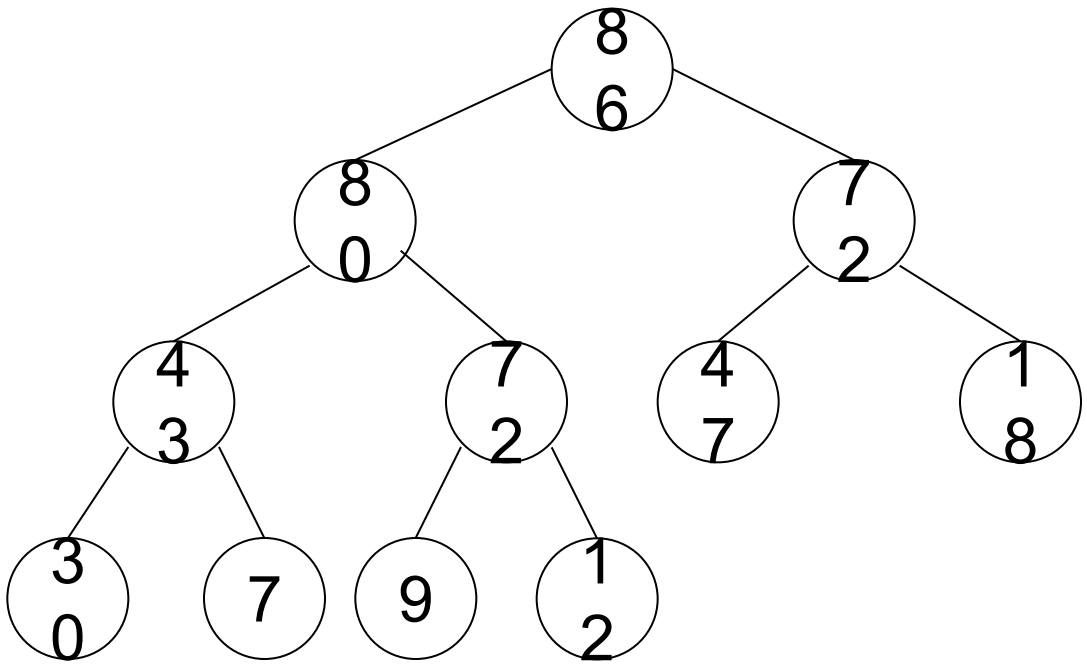
H[1]=86  H[2]=72



# *Добавление элемента в пирамиду*

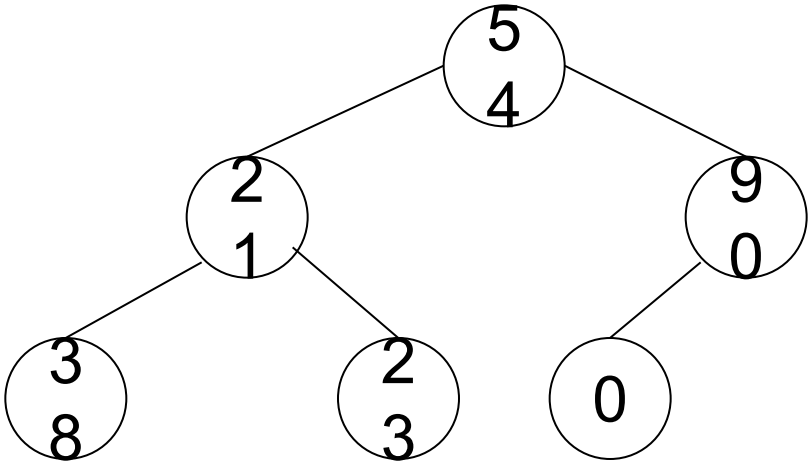


# *Удаление из пирамиды*



# *Пирамидальная сортировка*

Пример: int A[]={54, 21, 90, 38, 23, 0};




38 23 21 0

$n \log_2 n$

# *Сбалансированные деревья*

AVL

```
struct AVLTREE{
   int dann;
   int balance;
   AVLTREE *pleft;
   AVLTREE *pright;
   };
```

balance<0

balance=0

balance>0