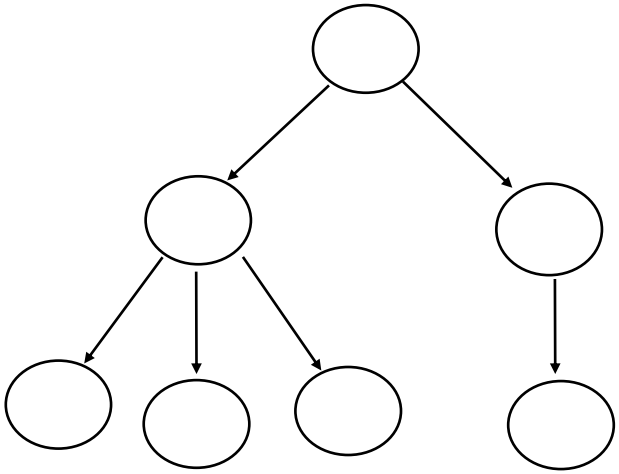
Нелинейни динамични структури от данни

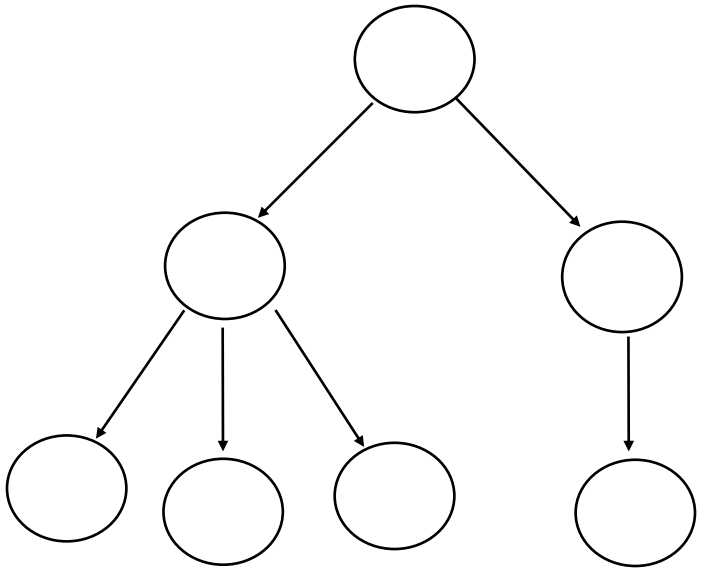
ДЪРВОВИДНИ СТРУКТУРИ

Съществуват две дефиниции на дървовидна структура или дърво.

Нерекурсивна: Дърво е свързан граф без цикли.

Рекурсивна: Дърво от тип T е структура, образувана от данна от тип T, наречена "корен", и крайно множество (възможно и празно) елементи - дървета от тип T, наречени "поддървета".



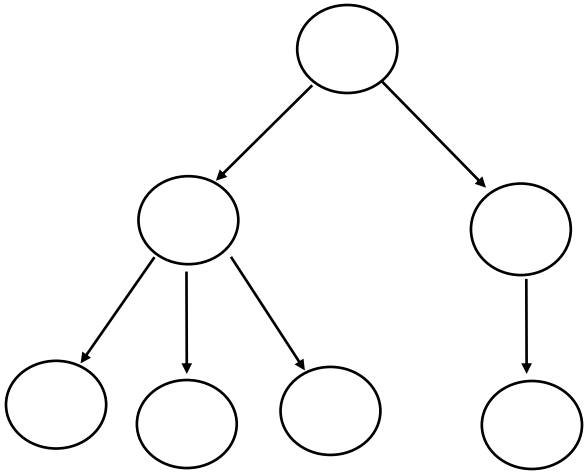


Всяко дърво се образува от възли и дъги, които ги свързват. Върховете могат да бъдат от два вида:

- родител
- наследник

Върхът без родител се нарича “корен” и всяко дърво има само един такъв връх.

От всеки връх могат да излизат няколко дъги (техния брой се нарича "степен" на елемента), но влиза само една дъга, т. е. връзката "родител-наследник" е еднопосочна и не позволява цикли:



Оттук се вижда, че структурата едносвързан списък представлява частен случай на дърво или т.н. "изродено" дърво.

Степен на дърво се нарича максималната степен на елемент от тази структура. За горния пример тя е 3.

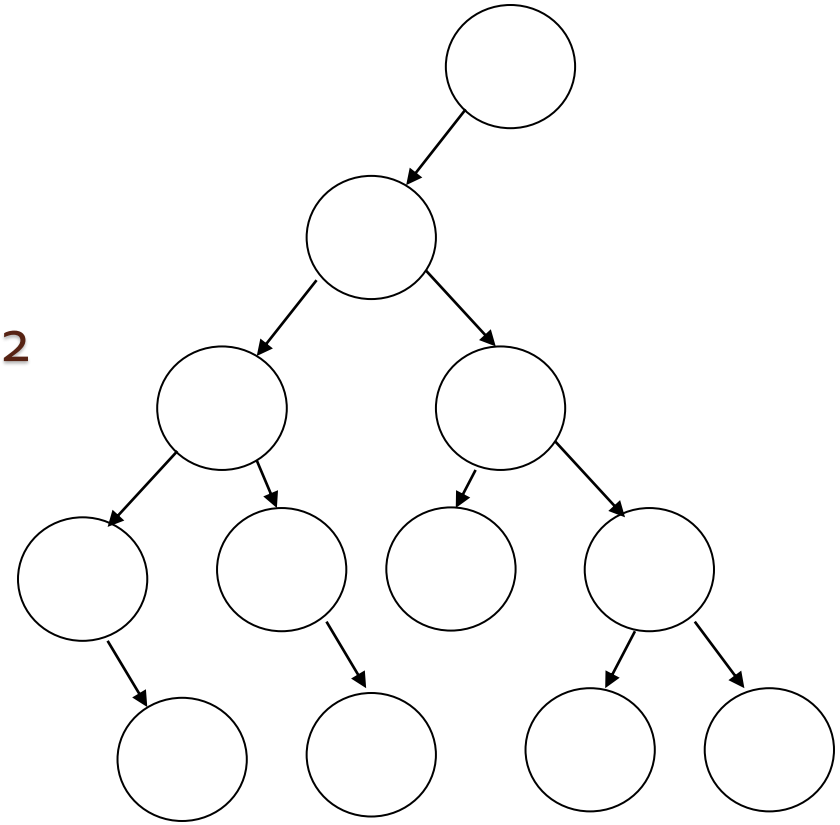
Върховете без наследници се наричат "листа".

Височина на дърво е дължината на най-дългия път от корена до листо.

От дефиницията следва, че между два произволни върха в едно дърво съществува единствен път.

Например:

Дървото има степен 2



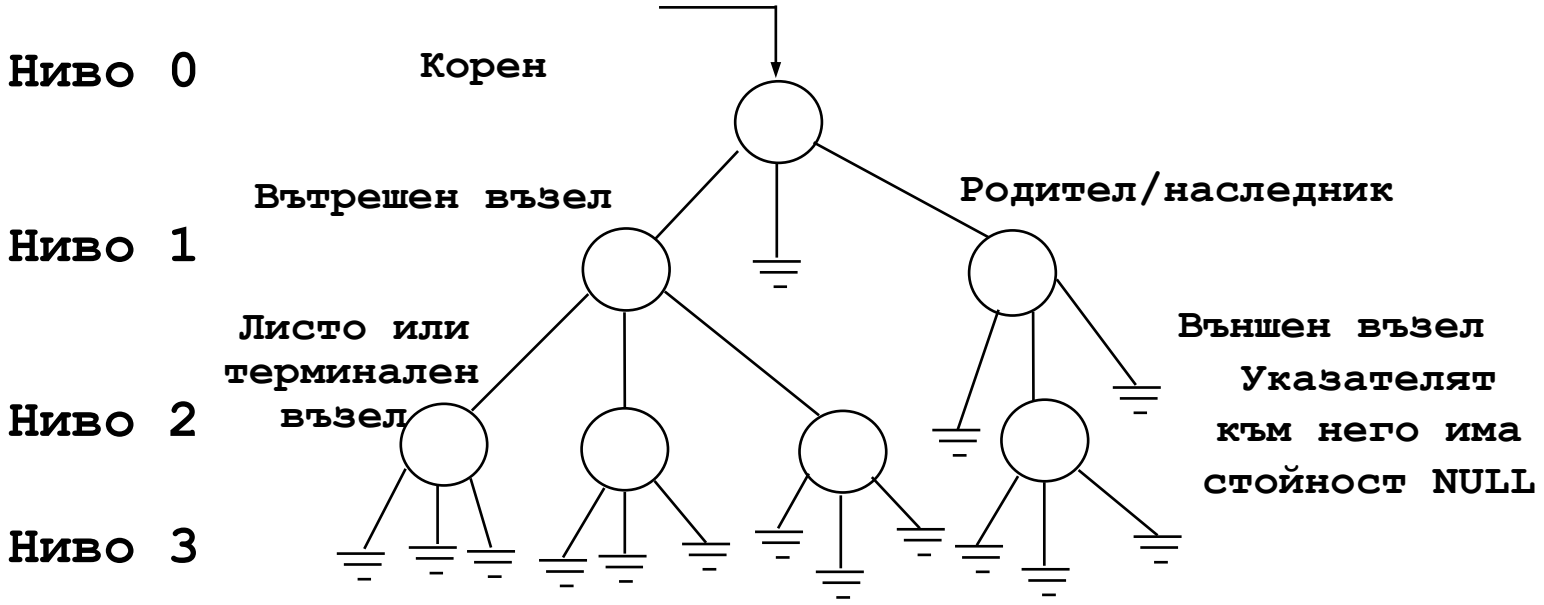
Височината на дървото е 5

Динамично представяне на структурата дърво (със степен 3):

```
struct elem
   {char key; elem *p1, *p2, *p3;} *root;
```

В така описаното дърво се съхраняват символни данни.

Коренът на дървото е свързан с указателя **root**. В контекста посочената в примера дефиниция, представеното по-горе дърво може да се изобрази като:

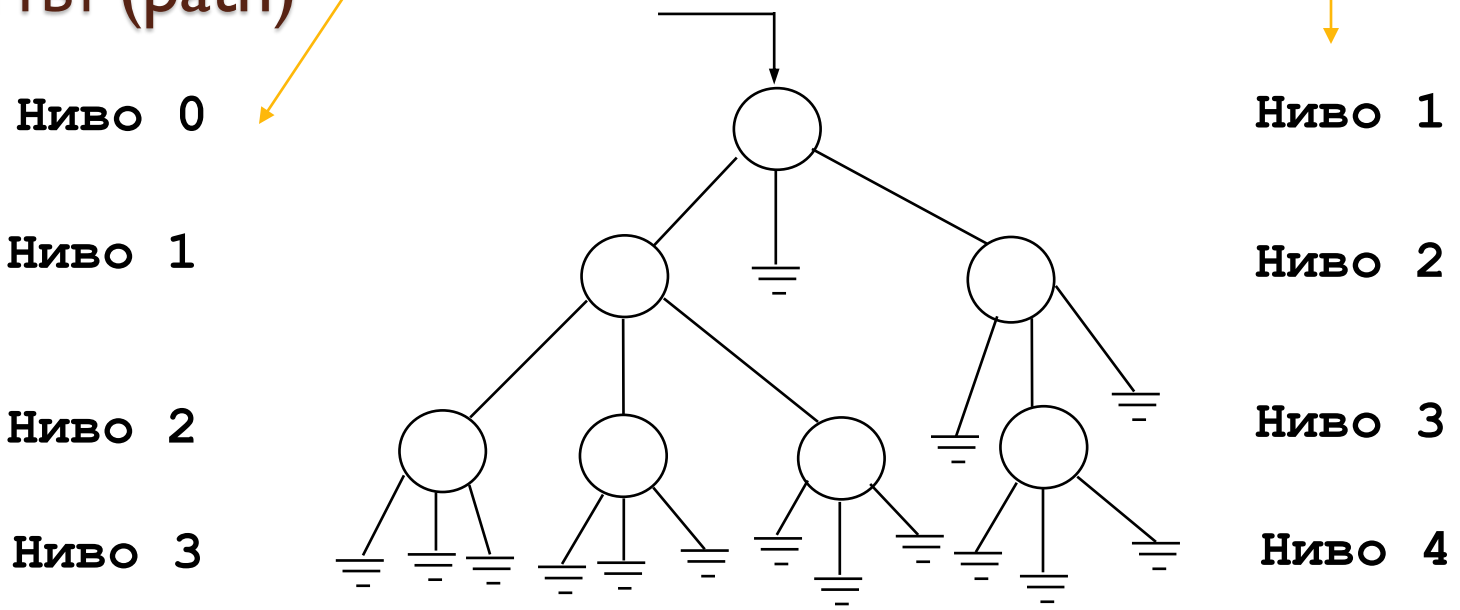


Терминология:

- Връх (vertex)
- Възел (node)
- Ребро (edge)
- Дъга (arc)
- Път (path)

Робърт Седжуик, *Алгоритми на C, том 1,2*, Софтпрес, С., 2002

Лендерт Амерал, *Алгоритми и структури от данни в C++, ИК СОФТЕХ*, С., 2001



Най-голямо приложение намира частния случай на дървовидна структура - двоично или бинарно дърво

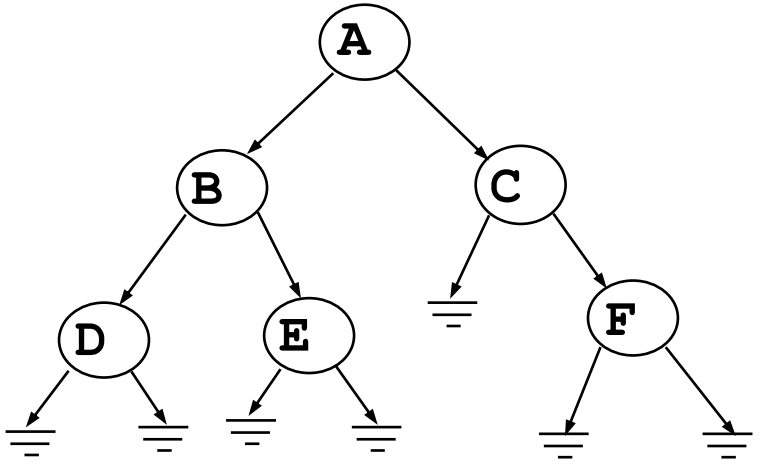
Двоично дърво е дървовидна структура от степен 2, т.е. всеки възел има не повече от два наследника - ляв и десен:

Ниво 0

Ниво 1

Ниво 2

Ниво 3



Динамично представяне на двоично дърво.

```
struct elem
   {char key; elem *left, *right;} *root;
```

Няколко допълнителни дефиниции:

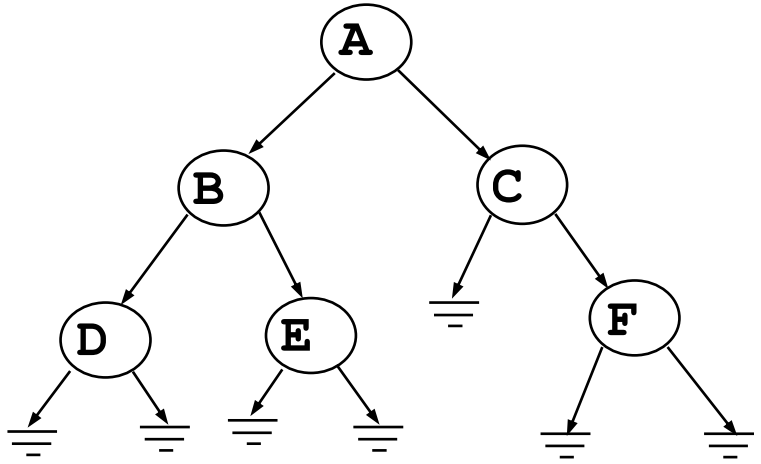
- Дължина на пътищата в дърво е сумата на нивата на неговите възли (за даденото дърво - 28).
- Дължина на вътрешните пътища в бинарно дърво е сумата от нивата на всички негови вътрешни възли (за даденото дърво - 8).
- Дължина на външните пътища в бинарно дърво е сумата от нивата на всички негови външни възли (за даденото дърво - 20).

Ниво 0

Ниво 1

Ниво 2

Ниво 3



Основни математически свойства на двоичните дървета:

- Двоични дървета с **N** вътрешни възли има **N+1** външни възли.
- Двоично дърво с **N** вътрешни възли има **2N** връзки: **N-1** връзки към вътрешни възли и **N+1** връзки към външни възли.
- Дължината на външните пътища във всяко двоично дърво с **N** вътрешни възли е с **2N** по-голямо от дължината на вътрешните пътища.
- Височината на бинарно дърво с **N** вътрешни възли е не по-малка от $\mathbf{Log_2 N}$ и не по-голяма от **N - 1**. Ако височината е **h** (за даденото дърво - 3), тогава трябва да имаме:

$$2^{h-1} < N + 1 <= 2^h$$

Неравенството означава, че височината в най-добрия случай е равна на $\mathbf{Log_2 N}$, закръглено до най-близкото по-голямо цяло число.

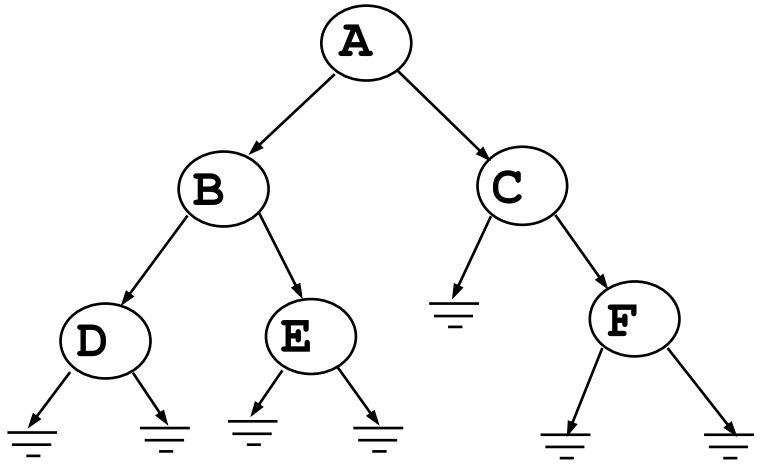
- Дължината на вътрешните пътища на двоично дърво с **N** вътрешни възли е не по-малка от $\mathbf{N*Log_2 (N/4)}$ и не по-голяма от **N*(N-1)/2**.

Ниво 0

Ниво 1

Ниво 2

Ниво 3



Основни операции със структурата дърво

. Инициализиране на дърво
. Обхождане на дърво
. Добавяне на елемент в дърво
. Премахване на елемент от дърво
. Търсене на елемент.

Инициализация на двоично дърво

На указателя към корена на дървото се присвоява стойност **NULL** при дефинирането на структурата:

```
struct elem
   {char key; elem *left, *right;} *root=NULL;
```

Обхождане на двоично дърво

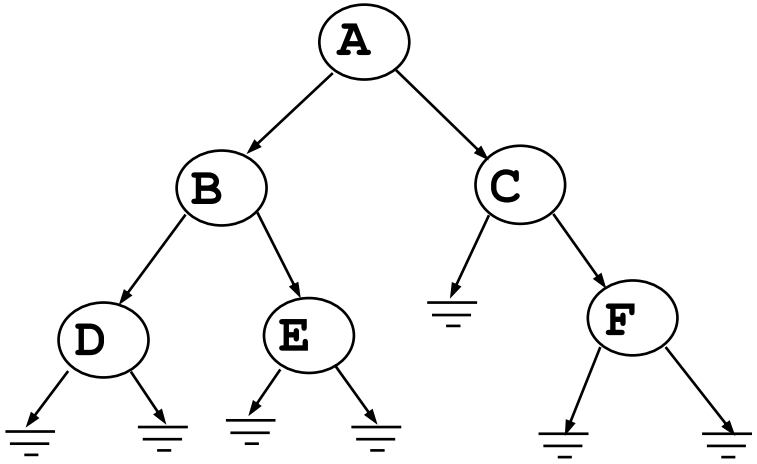
Обхождането означава посещаване на всеки възел по веднъж. Т.к. дървото е нелинейна структура, обхождането му може да става по различни начини. Различават методи за обхождане в ширина и в дълбочина, като последния се извършва по три стандартни начина:

- в прав ред - **PREORDER**
- във вътрешен ред (или симетрично обхождане) - **INORDER**
- в обратен ред - **POSTORDER**

При обхождането в прав ред се възлите се посещават в следната последователност: корен - ляво поддърво - дясно поддърво, което за горния пример на двоично дърво ще даде такава последователност: **ABDECF.**

Функция за рекурсивно обхождане на двоично дърво в прав ред:

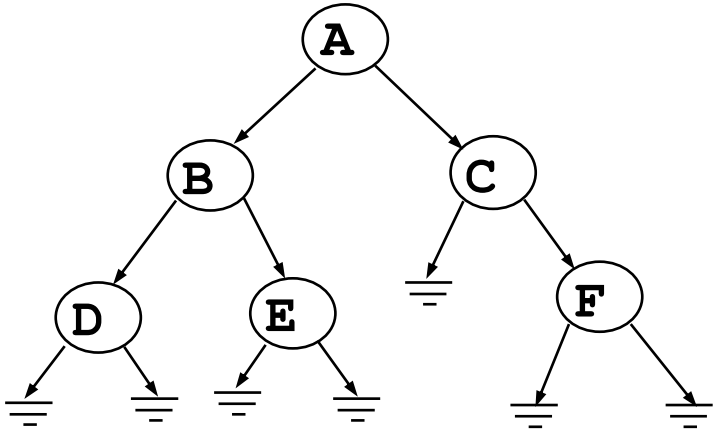
```
void preorder(elem *t)
{
  if (t)
    {
     cout<<t->key<<" ";
     preorder(t->left);
     preorder(t->right);
    }
}
```



Обхождането във вътрешен ред се извършва така: ляво поддърво - корен - дясно поддърво, т.е. за горния пример - **DBEACF**.

Функция за рекурсивно обхождане на двоично дърво във вътрешен ред

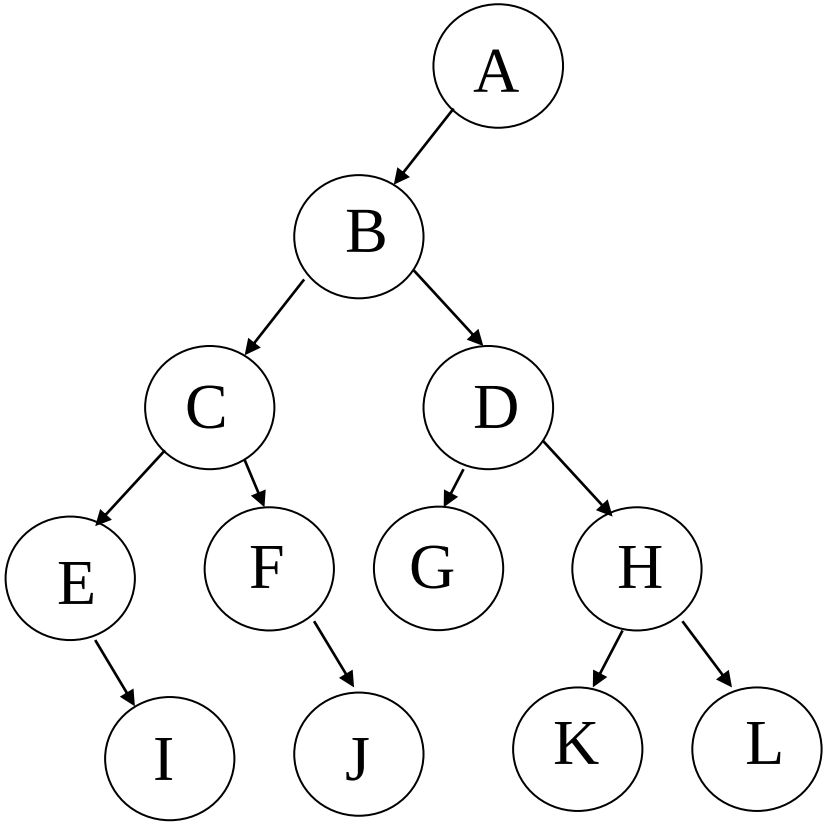
```
void inorder(elem *t)
{
  if (t)
    {
     inorder(t->left);
     cout<<t->key<<" ";
     inorder(t->right);
    }
}
```



Обхождането в обратен ред означава следната последователност на посещаваните възли: ляво поддърво - дясно поддърво - корен, за вече споменатия пример тя е **DEBFCA**.

```
void postorder(elem *t)
{
  if (t)
    {
     postorder(t->left);
     postorder(t->right);
     cout<<t->key<<" ";
    }
}
```

Още един пример за обхождане на двоично дърво:



Preorder (КЛД): A B C E I F J D G H K L

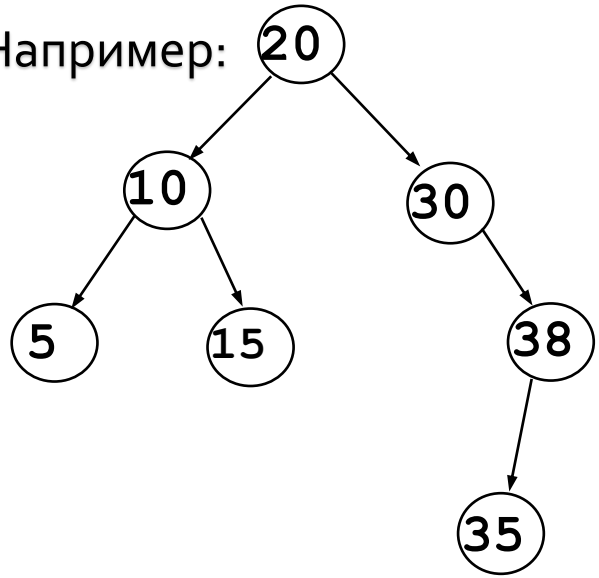
Inorder (ЛКД): E I C F J B G D K H L A

Postorder (ЛДК): I E J F C G K L H D B A

Останалите три основни операции с дървовидни структури - добавяне, изключване и търсене на елемент - ще разгледаме върху една често използвана разновидност на структурата - **двоични дървета за търсене** или **подредени двоични дървета (или BST - Binary Search Tree)**.

Едно двоично дърво се нарича **двоично дърво за търсене**, ако за всеки връх с ключова стойност **K** е вярно:

- всички елементи в лявото му поддърво имат ключови стойности по-малки от **K**;
- всички елементи в дясното му поддърво имат ключови стойности по-големи от **K**;
- ключовите стойности са уникални. Например:



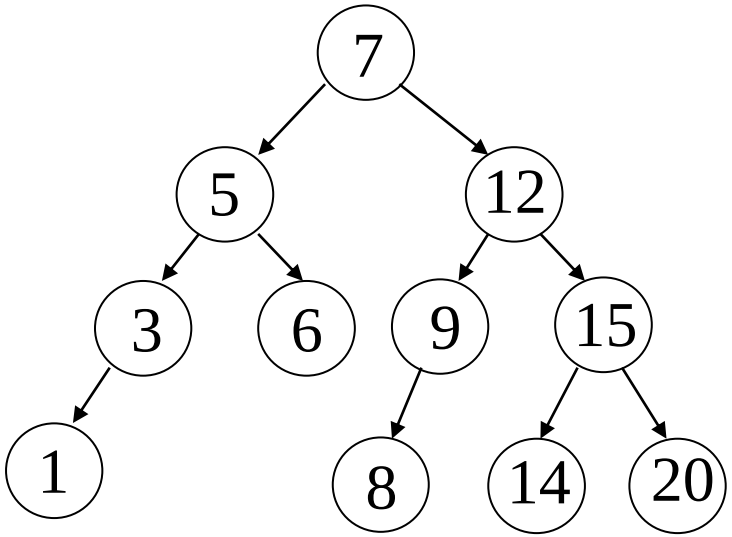
Симетричното обхождане на двоично дърво за търсене извежда ключовите стойности на структурата, подредени в нарастващ ред. Така за горния пример функция INORDER ще даде следния резултат: 5 10 15 20 30 35 38

Добавяне на елемент в подредено двоично дърво

Нови елементи се добавят в структурата във вид на листа.

Разгледаната долу операция за добавяне на нов възел в двоично дърво за търсене е реализирана като рекурсивна функция.

```
void add(int n, elem* &t)
{
  if (t==NULL)
    {
      t=new elem;
      t->key=n;
      t->left=t->right=NULL;
    }
      else
    {
      if (t->key<n)
         add(n,t->right);
       else
         add(n,t->left);
    }
}
```

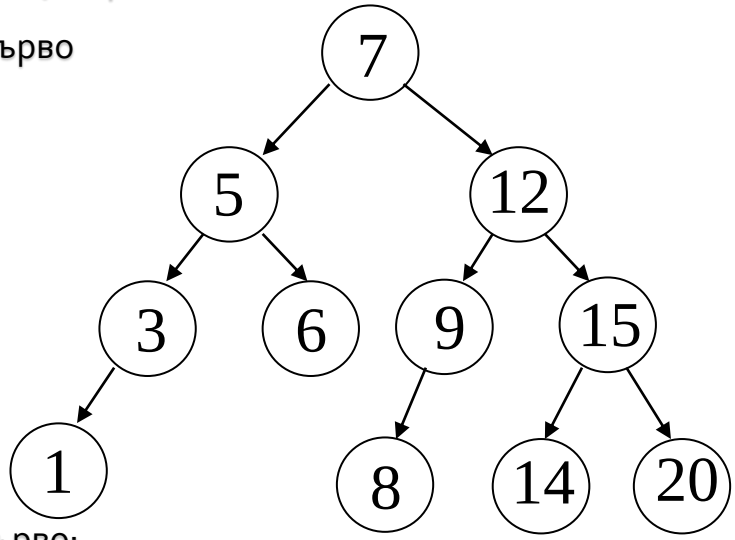


Търсене на елемент в двоично дърво

Двете примерни функции връщат като резултат указател към търсения елемент. Ако елементът не принадлежи на дървото, върнатата стойност е NULL

Итеративно търсене на елемент в подредено двоично дърво

```
elem *search_iter(elem *t, int k)
{
  while ((t!=NULL)&&(t->key!=k))
   if (t->key<k)
       t=t->right;
     else
       t=t->left;
  return t;
}
```



Рекурсивно търсене на елемент в подредено двоично дърво:

```
elem *search_rec(elem *t, int k)
{
  if (t!=NULL)
     if (t->key<k)
         t=search_rec(t->right, k);
           else
         if(t->key>k)
           t=search_rec(t->left, k);
  return t;
}
```

Още един вариант на итеративна функция за търсене в двоично дърво:

```
elem * &search(int k) // функцията връща адреса на указател
{                                  // към търсения елемент
  elem **p=&root;
  for (;;)
  {
    if (*p==NULL) return *p;
    if (k<(*p)->key) p=&(*p)->left;
       else
    if (k>(*p)->key) p=&(*p)->right;
       else
             return *p;
  }
}
```

ПРЕМАХВАНЕ НА ВЪЗЕЛ ОТ ДВОИЧНО ДЪРВО

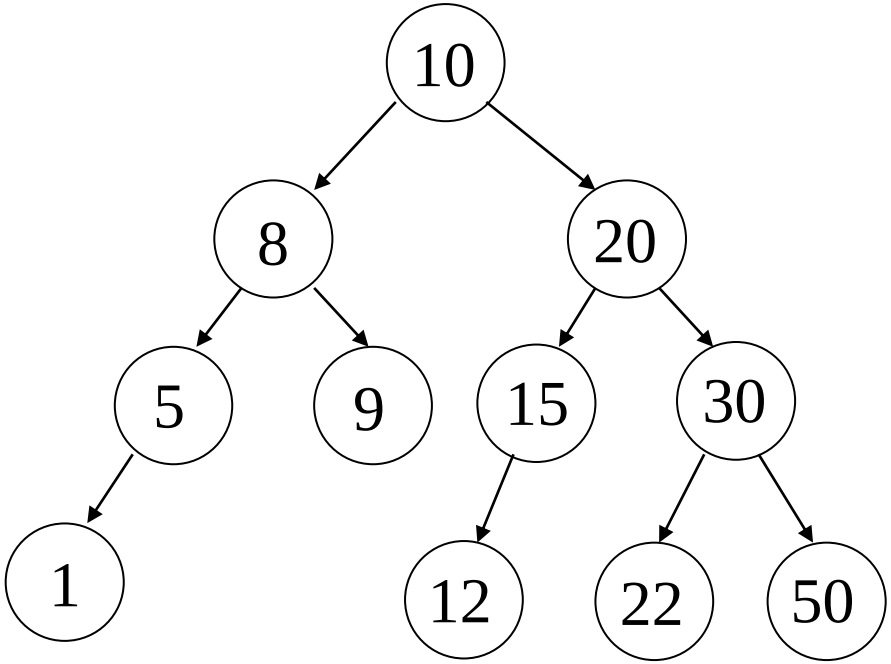
Операцията премахване на възел от двоично дърво за търсене не трябва да нарушава подредбата на структурата. Възможни са следните варианти:

- Ако премахваният елемент е листо, неговото отстраняване не нарушава структурата;

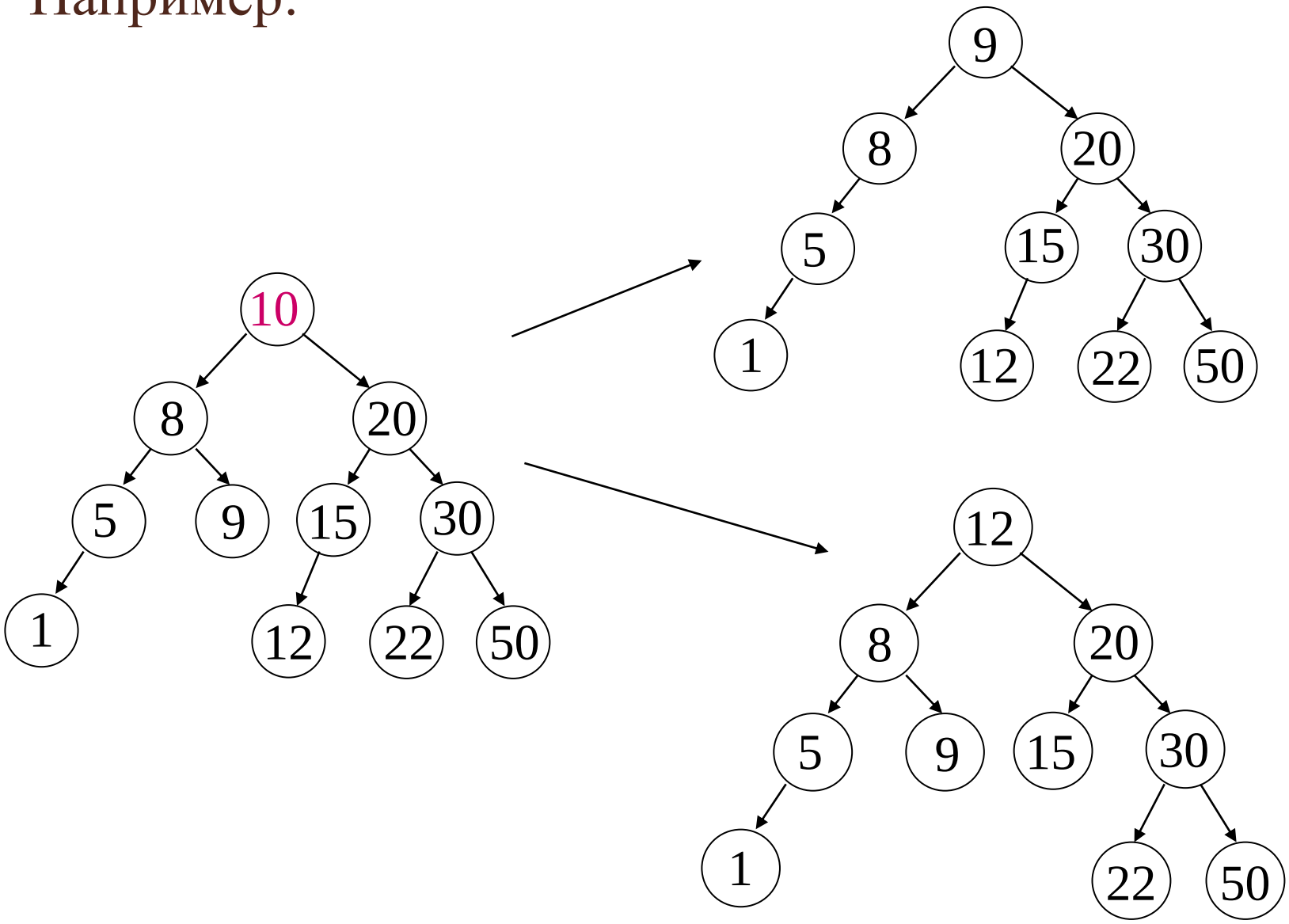
- Ако премахваният елемент има само един наследник, той замества изтривания възел;

- Ако премахваният елемент има два наследника, то изтриваният възел се замества или с най-десния (най-големия) елемент от лявото му поддърво или с най-левия (най-малкия) елемент от дясното му поддърво.

Например:



Например:


10
8
20
5
9
15
30
1
12
22
50
9
8
20
5
15
30
1
12
22
50
12
8
20
5
9
15
30
1
22
50

Изтриване на възел:

```
struct tree  {
                int key;
                tree *left,*right;
              };
tree *root=NULL,*z;
...
int del(int k)
{
  tree * &p=search(k), *p0=p, **qq, *q;
// search(k)- функция, която търси в дървото възел
// с ключова стойност k и връща адреса на указател към него
  if (p==NULL) return 0;  //Възелът не е намерен
  if ((p->right==NULL))
    {
     p=p->left; delete p0;
    }
      else
    if(p->left==NULL)
      {
      p=p->right; delete p0;
      }
        else                          //Възелът е с два наследника
      {
           qq=&p->left;
           while ((*qq)->right)
           qq=&(*qq)->right;
           p->key=(*qq)->key;
           q=*qq;
           *qq=q->left;
           delete q;
      }
    return 1;
}
```

Задача: създаване на двоично дърво, пресмятане на неговата височина и брой възли с последващо изобразяване на дървото. (Елементите на дървото са от символен тип.)

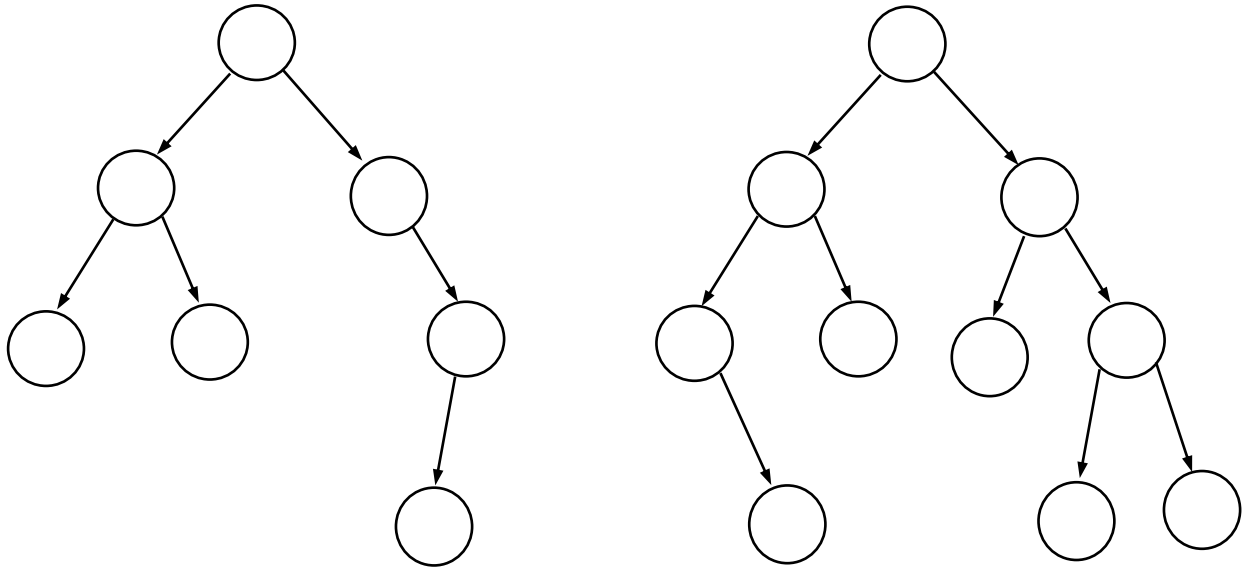
```
//TREE'S PARAMETERS
#include <iostream>
using namespace std;
struct elem
   { char key; elem *left, *right;} *root=NULL;

void add(char n, elem* &t);    //prototypes
int count(elem *t);
int height(elem *t);
void printnode(char n, int h);
void show (elem *t, int h);
void main()
{
  char c;
  cout<<"Enter some characters to be placed in a binary tree (followed by /): \n";
  while ((cin>>c)&&(c!='/'))
       add(c, root);
  cout<<"\nNode's count: "<<count(root);
  int h=height(root)+1;       //+1 - for NULL-nodes
  cout<<"\nTree's height: "<<h<<'\n';
  show(root, h);
}
```

```
void add(char n, elem* &t)
{
  if (t==NULL)
    {
      t=new elem;
      t->key=n;
      t->left=t->right=NULL;
    }
      else
    {
      if (t->key<n)
                add(n,t->right);
              else
                add(n,t->left);
    }
}
int count (elem *t)
{
  if (t==NULL) return 0;
  return count(t->left)+count(t->right)+1;
}
```

```
int height (elem *t)
{
  if (t==NULL) return -1;
  int u=height(t->left), v=height(t->right);
  if (u>v) return u+1;
     else
                  return v+1;
}
void printnode (char n, int h)
{
  for (int i=0; i<h; i++)
     cout<<"   ";
  cout<<n<<'\n';
}
void show (elem *t, int h)
{
  if (t==NULL)
    {
      printnode('*', h);
      return;
    }
  show(t->right, h+1);
  printnode(t->key, h);
  show(t->left, h+1);
}
```

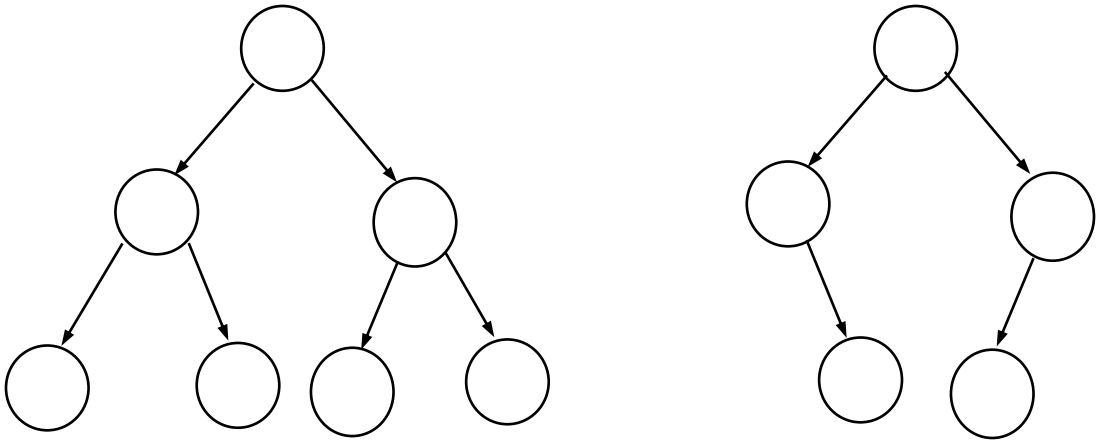
Балансирани двоични дървета

Всяко двоично дърво може да бъде **балансирано**. Двоично дърво се нарича **балансирано** или **балансирано по височина**, ако разликата във височините на поддърветата на всеки негов възел е най-много 1:



Лявото дърво не е балансирано. Дясното е балансирано.

Едно двоично дърво се нарича **идеално балансирано** или с **балансирано тегло**, ако всеки негов възел има ляво и дясно поддърво, в които броят на възлите се различава най-много с 1:



Очевидно е, че всяко идеално балансирано дърво (с балансирано тегло) също така е дърво с балансирана височина. Обратното твърдение не е вярно.

Идеално балансираното дърво може да бъде **пълно**, ако за всеки възел разликата в броя на възлите в неговите поддървета е нула. Лявото дърво на горната фигура е пълно.

За да бъде двоичното дървото пълно, броят на възлите му трябва да е нечетен. Ако **K** е броя на нивата в дървото (коренът се брои за 0-во ниво), то броят на вътрешните възлите в пълното двоично дърво **N** е равен на $2^{K+1} - 1$.

Балансът обикновено е важен фактор при работа с двоични дървета за търсене, където е от значение времето за търсене и не трябва да има отделни дълги клони. Балансираните дървета често наричат AVL-дървета в чест на Adelson-Velski и Landis - двамата учени, създали най-много алгоритми за този клас дървовидни структури.

Основните операции с балансираните дървета - добавяне, премахване и търсене на елемент с зададена ключова стойност - имат сложност $O(Log_2 N)$.

Ако **h(N)** е височината на балансирано двоично дърво с **N** вътрешни възела, то

$$Log_2(N+1) <= h(N) <= 1.4404 Log_2(N+2) - 0.328$$

Динамична променлива, описваща балансирано двоично дърво

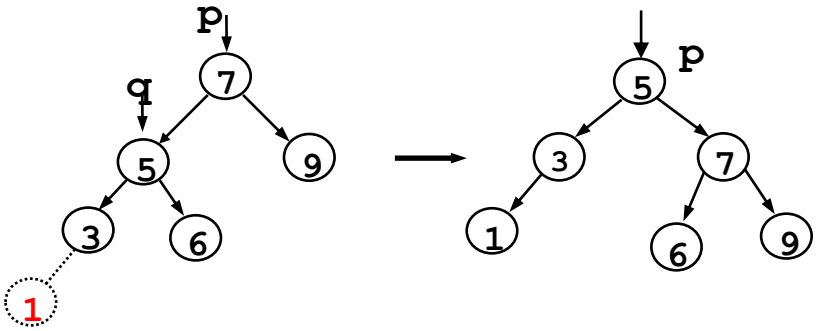
Във всеки възел на такава структура необходимо е да се съхранява информация за баланса на съответните поддървета:

```
struct tree{
              int key;
              int bal;      // 0 - двата клона на възела
                            // с еднаква дължина
                            // -1 - левия клон е по-дълъг;
                            // +1 - десния е по-дълъг
              tree *left,*right;
            };
tree *root=NULL;
```

Всяко двоично дърво може да бъде балансирано.

Има 4 основните операции за балансиране на двоично дърво, наречени ротации.

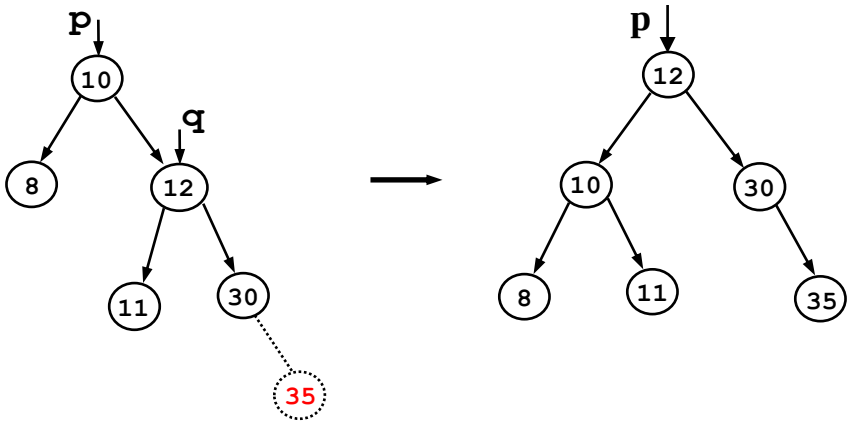
- **Дясна ротация**. Тази операция е необходима тогава, когато във вече балансираното двоично дърво се добавя елемент към най-левия клон:



Функция за **дясна ротация** в двоично дърво за търсене:

```
void R_rot(tree * &p)
{
 tree *q;
 if(p->bal==-1){
                q=p->left;
                if(q->bal==-1) {
                        p->left=q->right;
                        q->right=p;
                        p->bal=0;
                        p=q;
                        }
                }
}
```

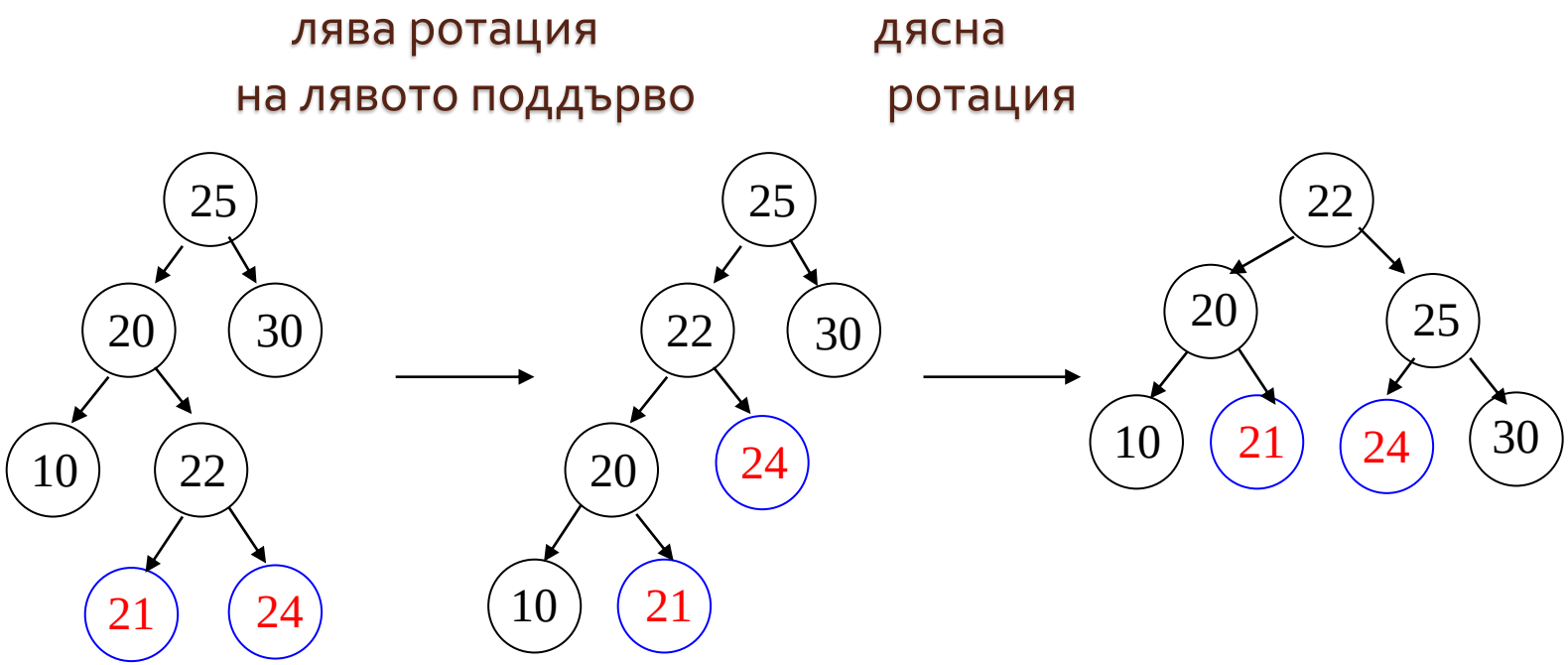
- Лява ротация. Тази операция е необходима тогава, когато във вече балансираното двоично дърво се добавя елемент към най-десния клон:



Функция, реализираща операцията лява ротация в двоично дърво за търсене:

```
void L_rot(tree * &p)
{
 tree *q;
 if(p->bal==1){
                q=p->right;
                if(q->bal==1) {
                        p->right=q->left;
                        q->left=p;
                        p->bal=0;
                        p=q;
                        }
                }
}
```

Двойни ротации:

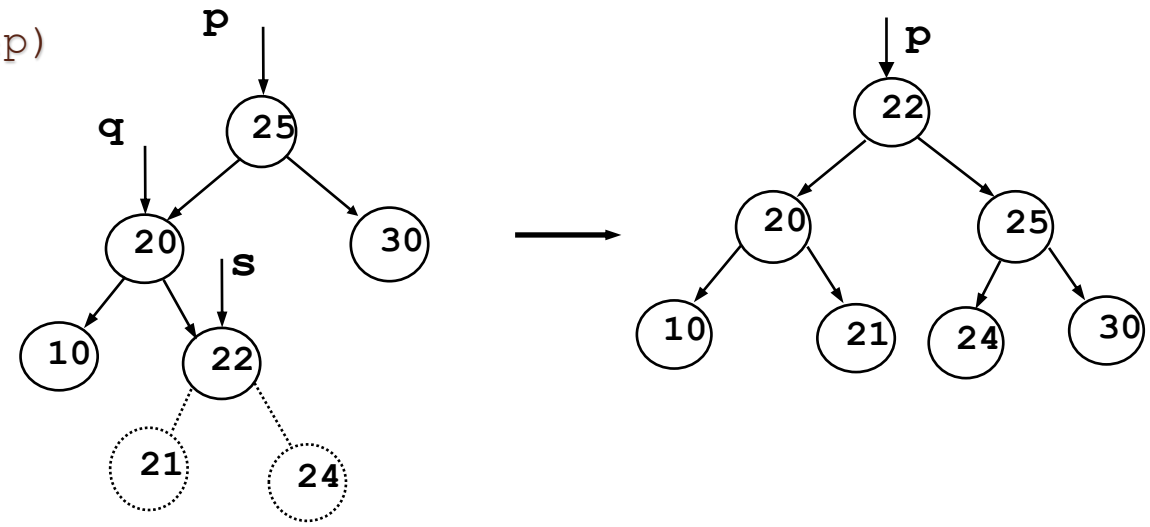


- ляво-дясна ротация

- Ляво-дясна ротация. Тази операция е необходима тогава, когато във вече балансираното двоично дърво се добавя елемент към дясното листо на по-високото ляво поддърво, в случая 21 или 24:

Функция за ляво-дясна ротация двоично дърво за търсене:

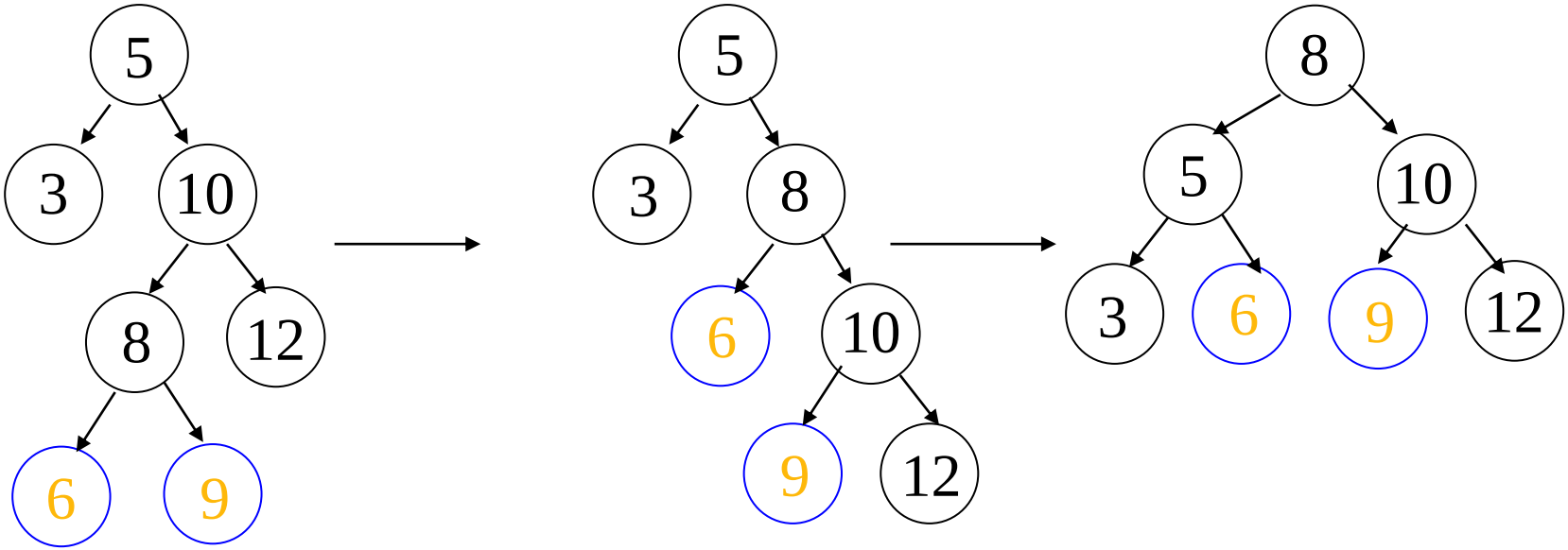
```
void L_R_rot(tree * &p)
{
tree *q,*s;
if(p->bal==-1)
 {
  q=p->left;
  if(q->bal==1)
   {s=q->right;
    q->right=s->left;
    s->left=q;
    p->left=s->right;
    s->right=p;
    if(s->bal==-1) p->bal=1;
       else p->bal=0;
    if(s->bal==1) q->bal=-1;
       else q->bal=0;
    p=s;
   }
 }
}
```



Още един пример:

дясна ротация
на дясното поддърво

лява
ротация

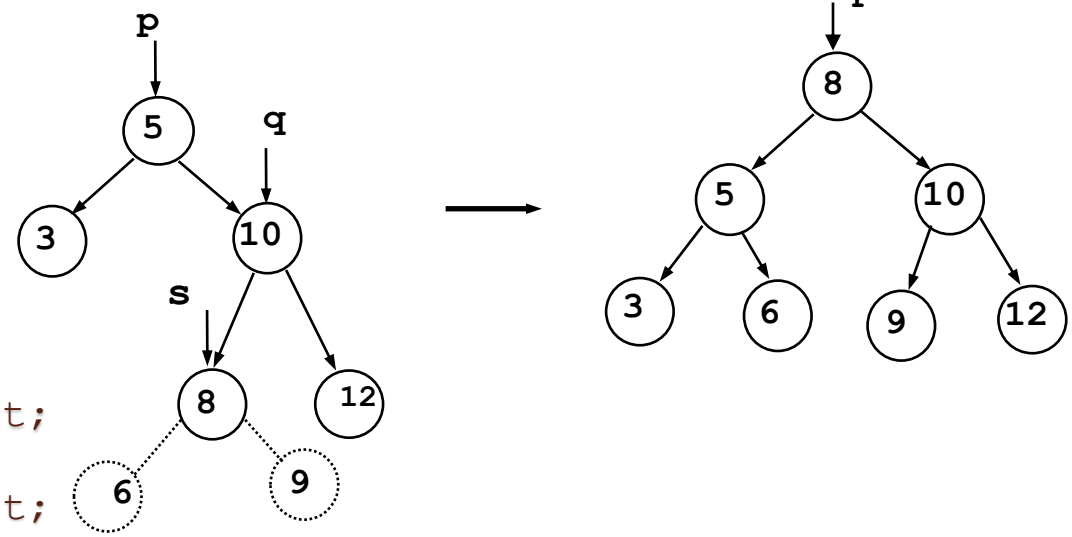


- дясно-лява ротация

- Дясно-лява ротация. Тази операция е необходима тогава, когато във вече балансираното двоично дърво се добавя елемент към лявото листо на по-високото дясно поддърво, в случая 6 или 9:

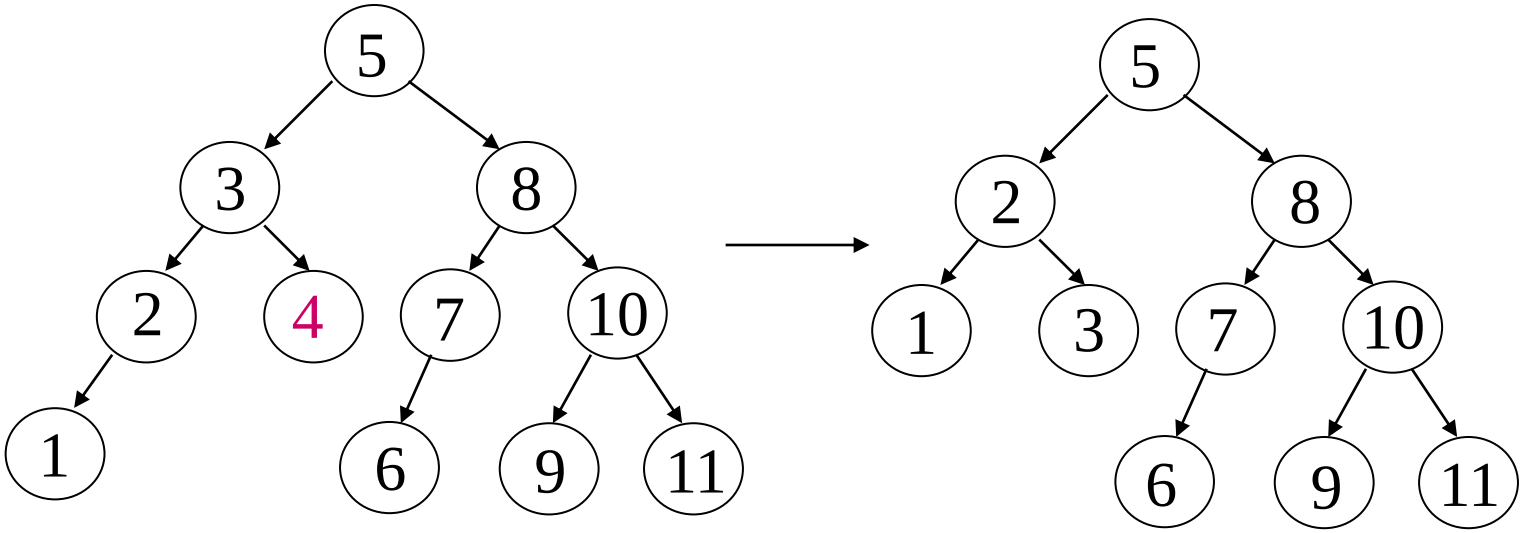
Функция за дясно-лява ротация двоично дърво за търсене

```
void R_L_rot(tree * &p)
{
tree *q,*s;
if(p->bal==1)
  { q=p->right;
    if(q->bal==-1)
      {  s=q->left;
         q->left=s->right;
         s->right=q;
         p->right=s->left;
         s->left=p;
         if(s->bal==1) p->bal=1;
                else p->bal=0;
         if(s->bal==-1) q->bal=1;
                else q->bal=0;
         p=s;
      }
  }
}
```



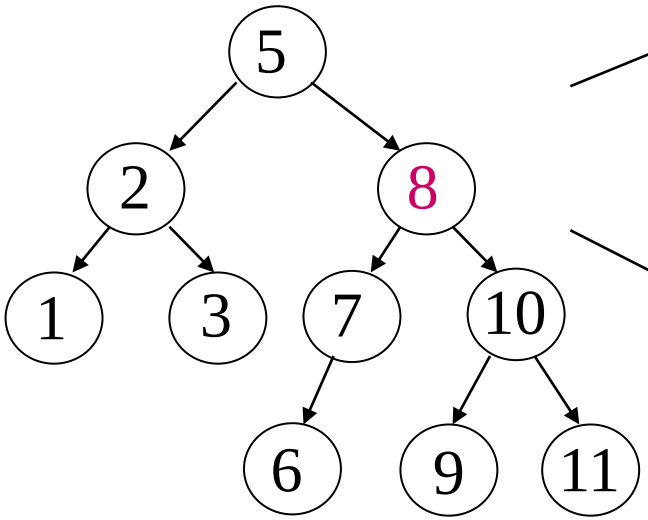
Изтриване на възел с последващо балансиране на двоично дърво:

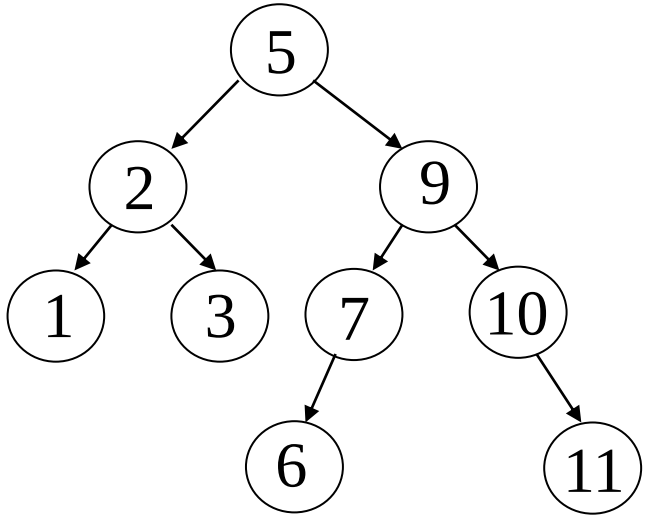
например, възел със стойност 4

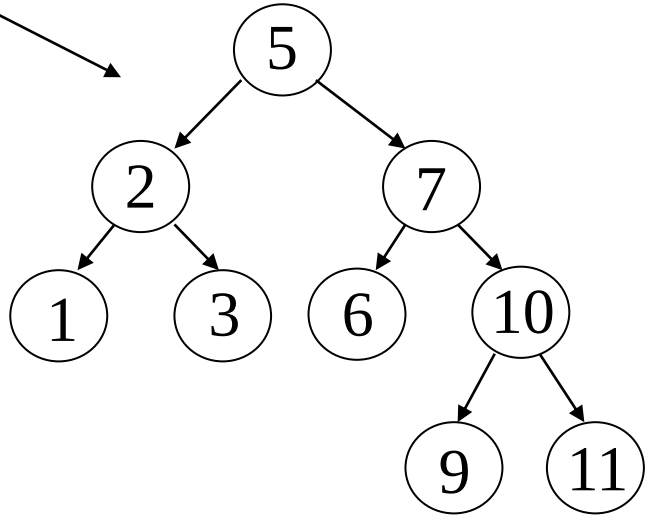


Структурата на дървото не се променя, но е необходима дясна ротация на лявото поддърво, за да се възстанови баланса

Например, 8:

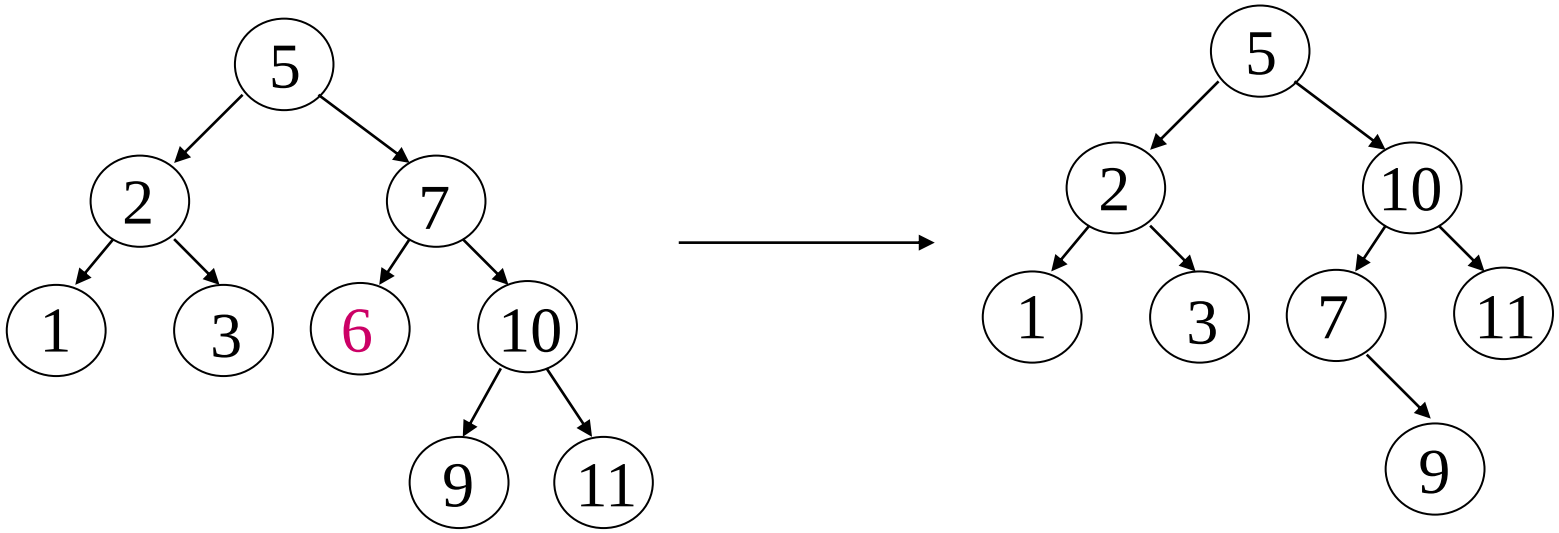






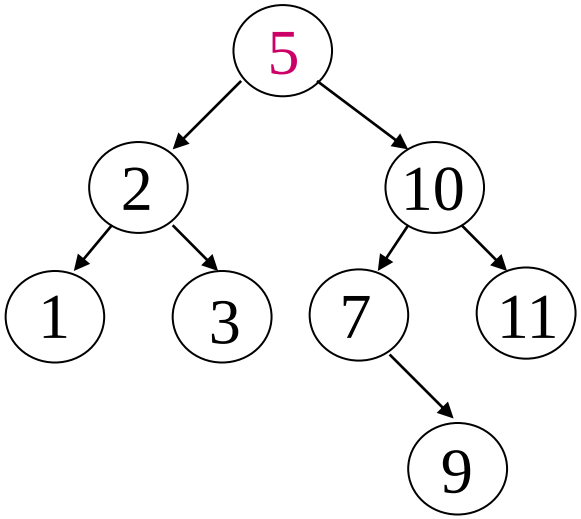
И двата варианта не изискват допълнително балансиране

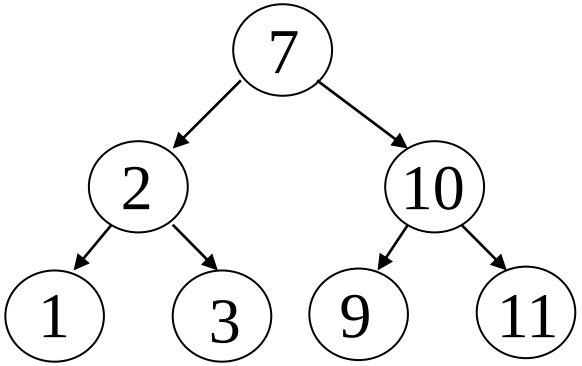
Например, 6:

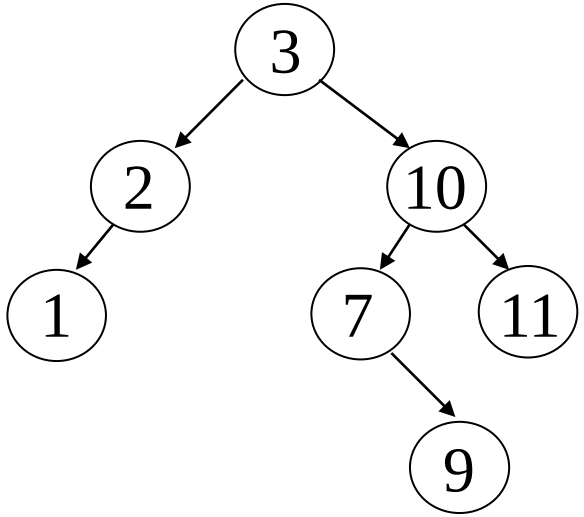


Извършена е допълнителна лява ротация на дясното поддърво

Примерно, изтриваме 5:

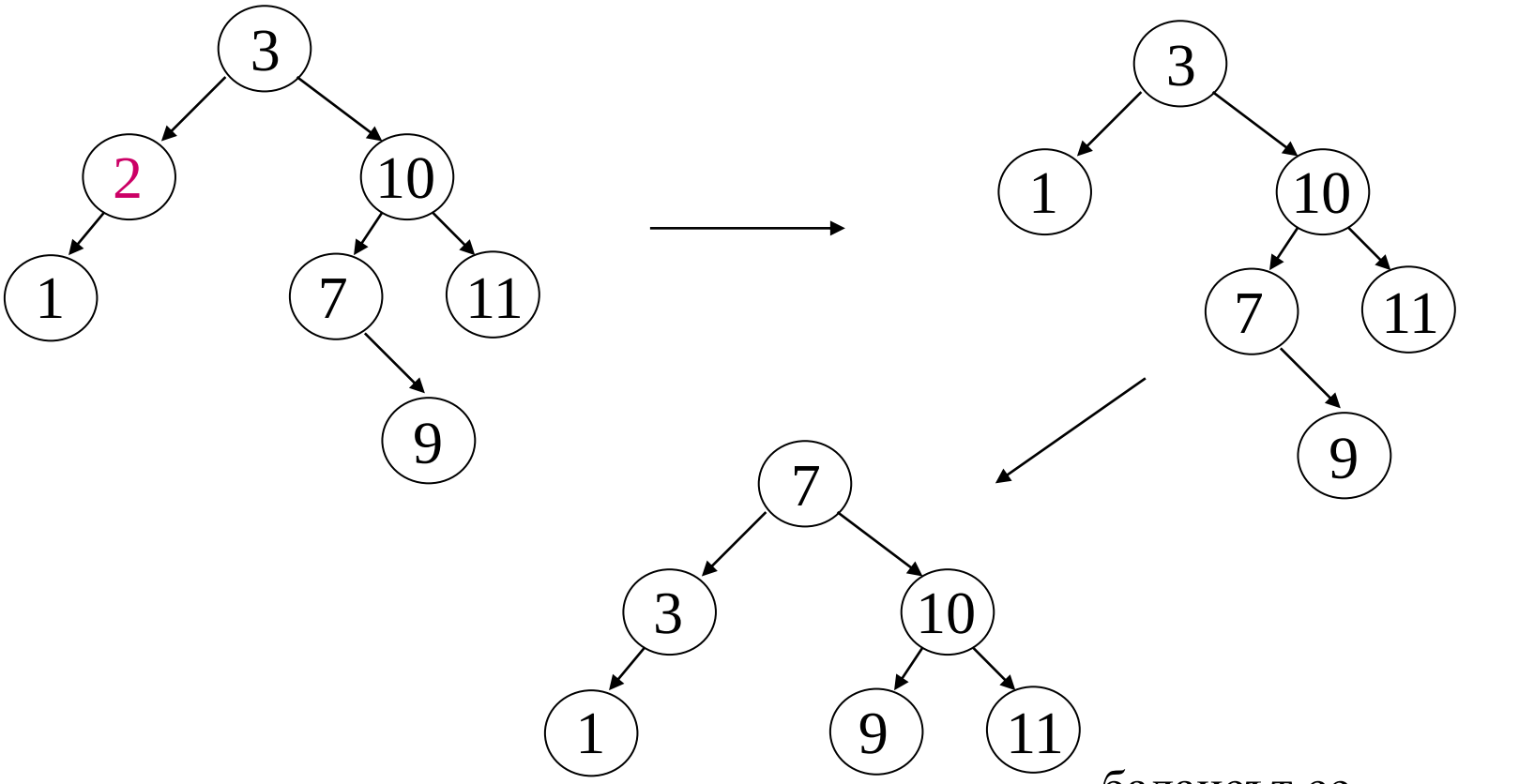






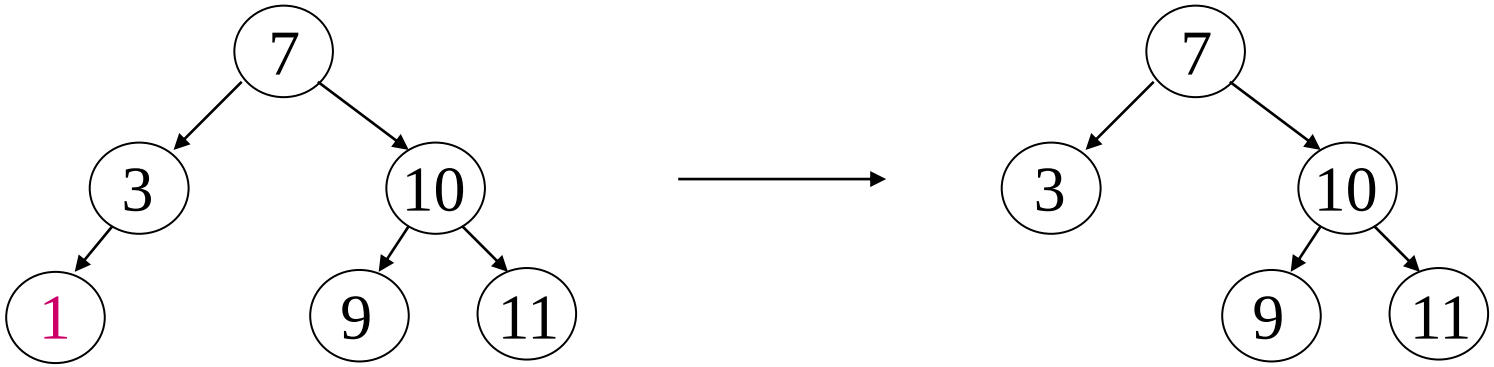
(и в двата случая няма допълнително балансиране)

Например, 2:



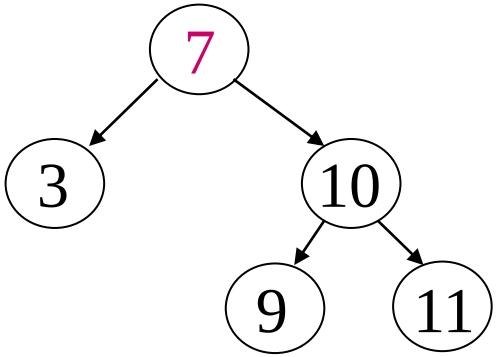
балансът се възстановява чрез дясно-лява ротация

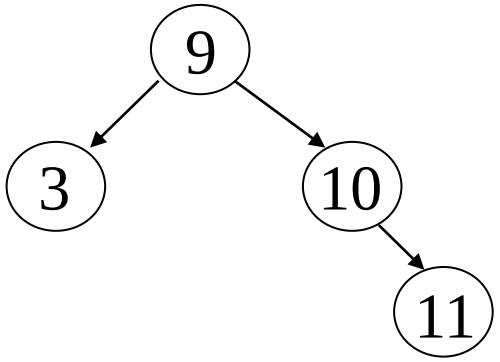
Изтриваме 1:

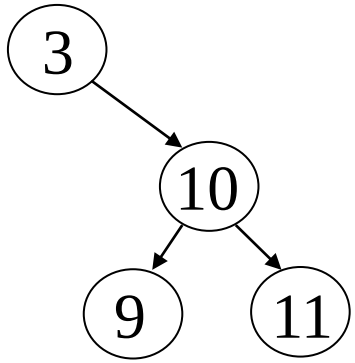


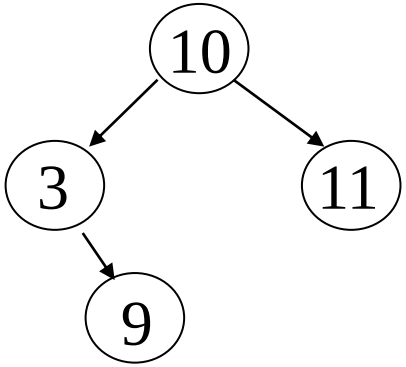
(не е нужно допълнително балансиране)

Изключваме 7:







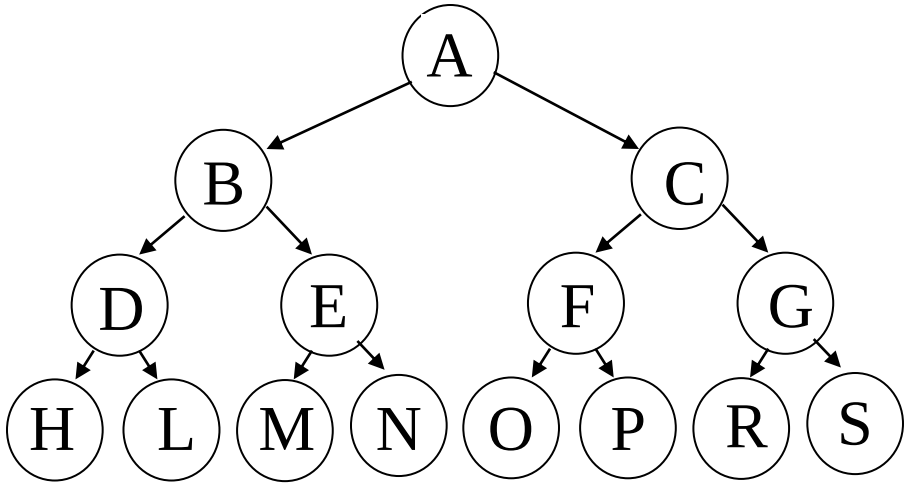


балансиране (от какъв тип?)

и т. н.

Статично представяне на дървовидни структури (двоично дърво)

Ако дървото е пълно (съдържа ($2^{h+1} - 1$) елементи, 2^i на i-то ниво, h е височина на дървото, започва от о ниво), елементите му могат да се запишат в масив:



Arr[i]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A	B	C	D	E	F	G	H	L	M	N	O	P	R	S

ляв син на Arr[i] -> Arr[2i+1]

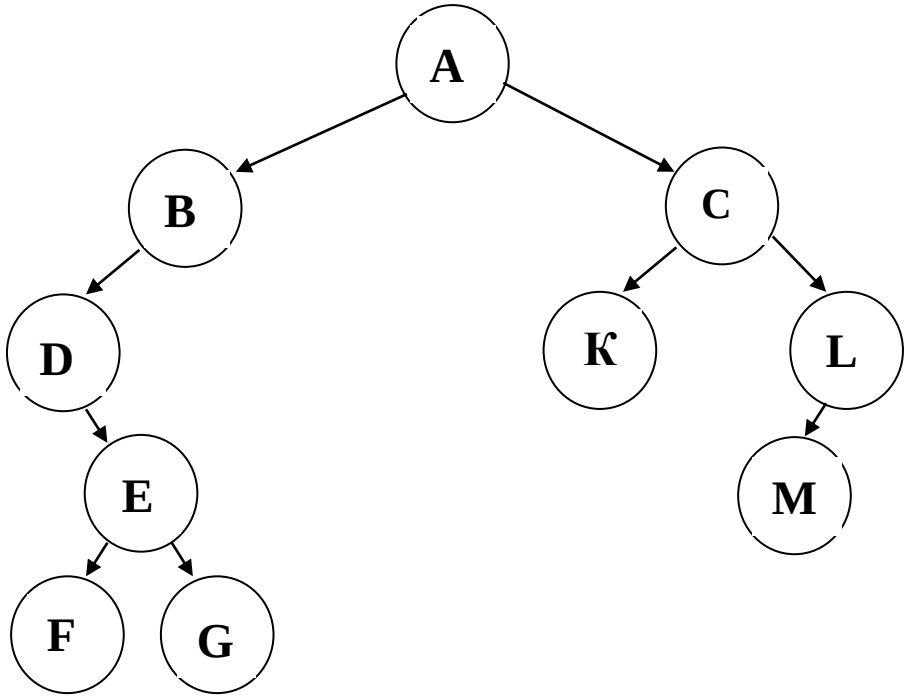
десен син на Arr[i] -> Arr[2*(i+1)]

родител на Arr[i]-> Arr[(i-1) /2]

За непълни (разредени) дървета:

```
struct tree
       { char v; int left, right; } tab[10];
struct b_tree
       { int root; struct tree tab[10];} T;
```

	v	left	right
1	D	0	4
2	A	3	7
3	B	1	0
4	E	5	6
5	F	0	0
6	G	0	0
7	C	8	9
8	K	0	0
9	L	10	0
10	M	0	0



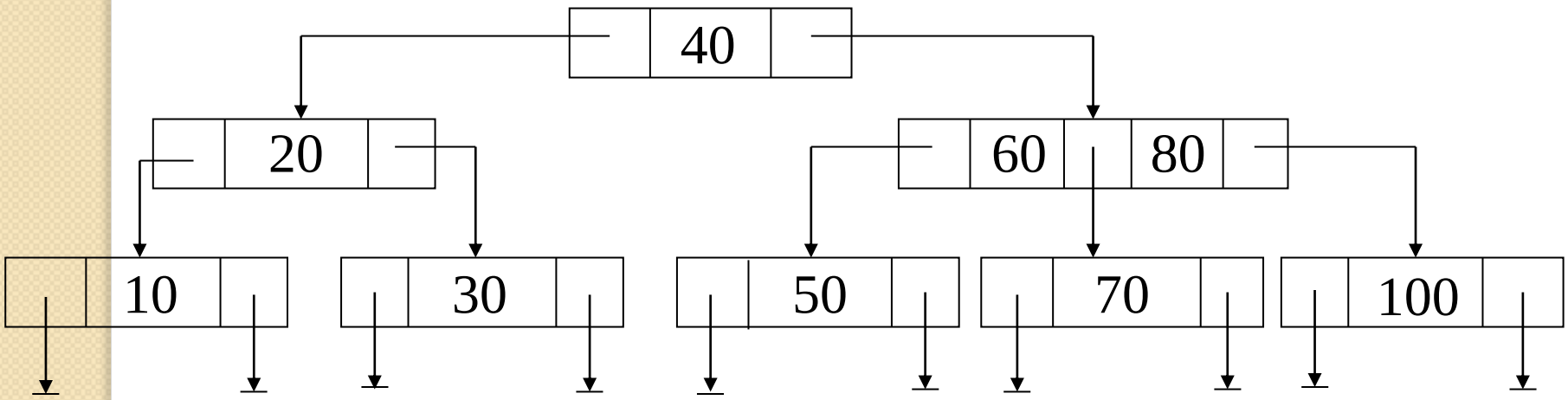
T.root=2

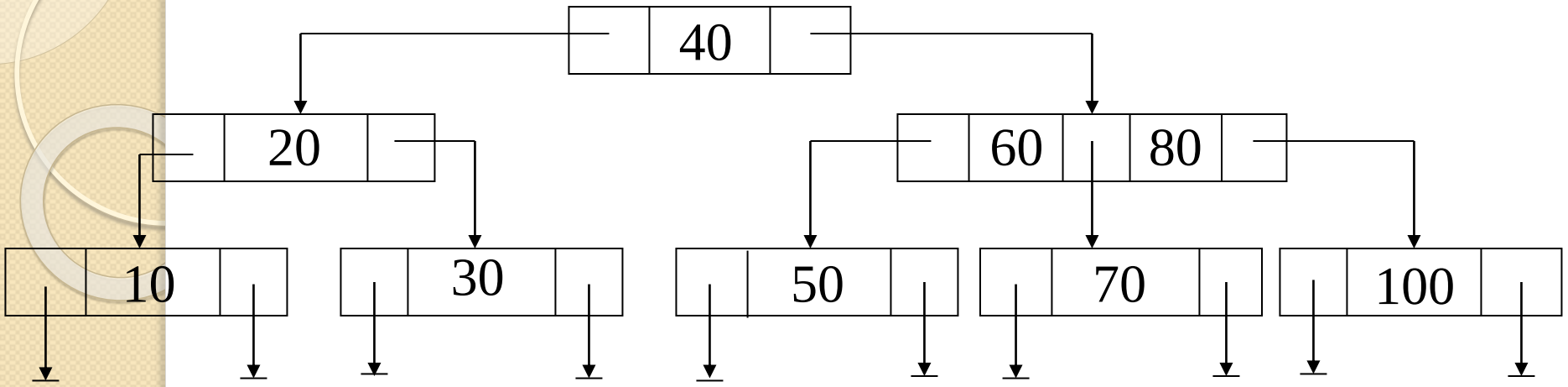
T.root=0 - инициализация

2-3 - дърво

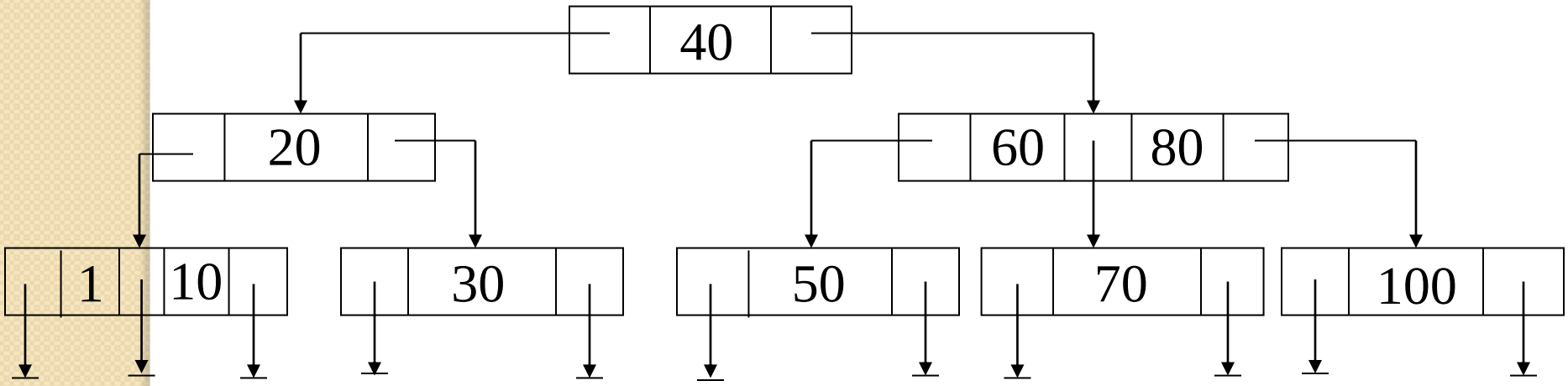
Това са идеално балансирано дърво за претърсване, в което:

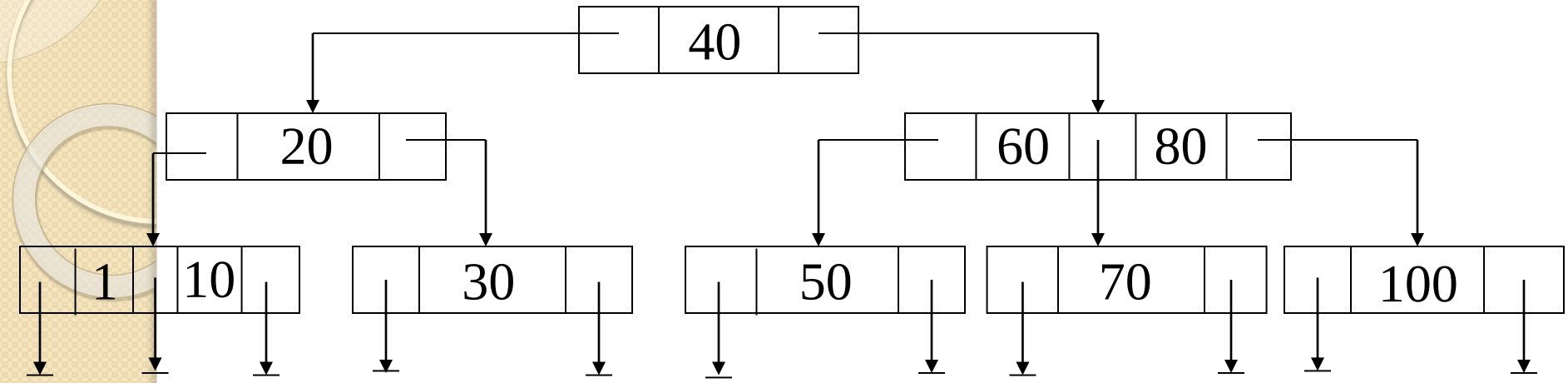
- всеки вътрешен възел съдържа една ключова стойност и два указателя или две ключови стойности и три указателя (наследника);
- всяко листо съдържа една или две ключови стойности;
- всички пътища от корена до листата имат еднакви дължини.



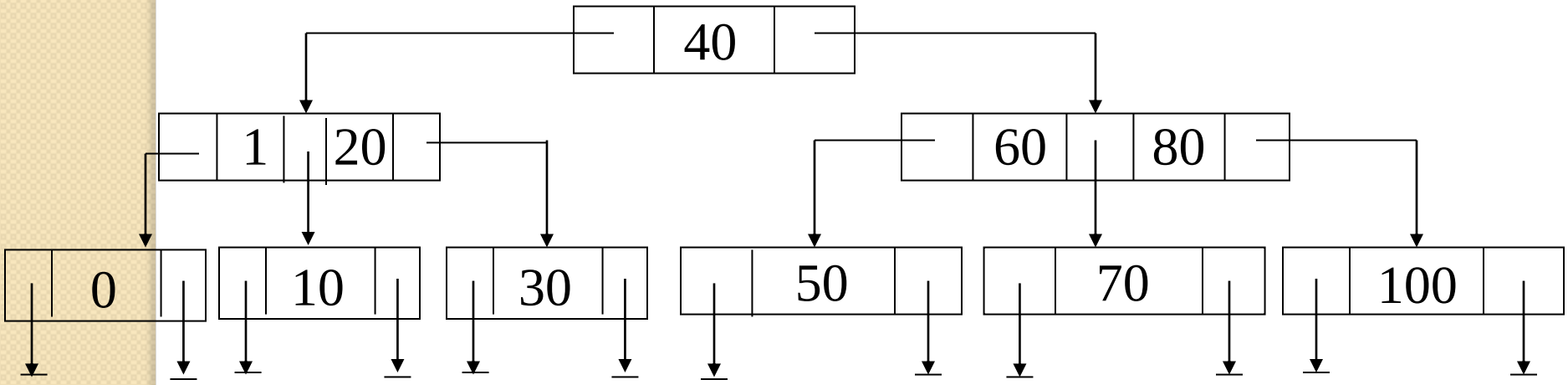


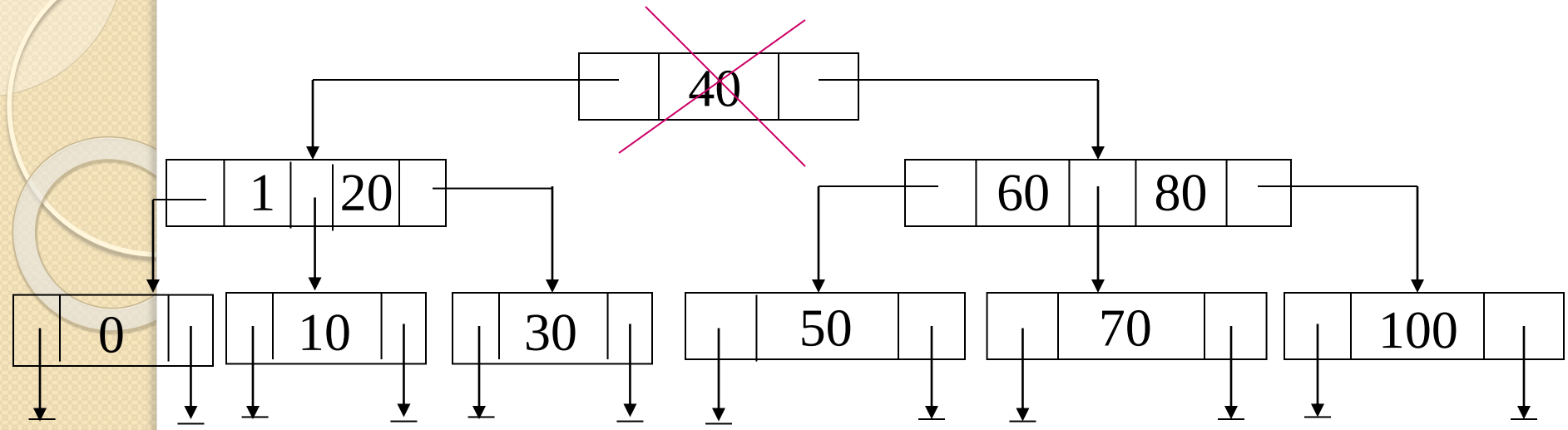
Примерно, добавяме елемент със стойност 1:



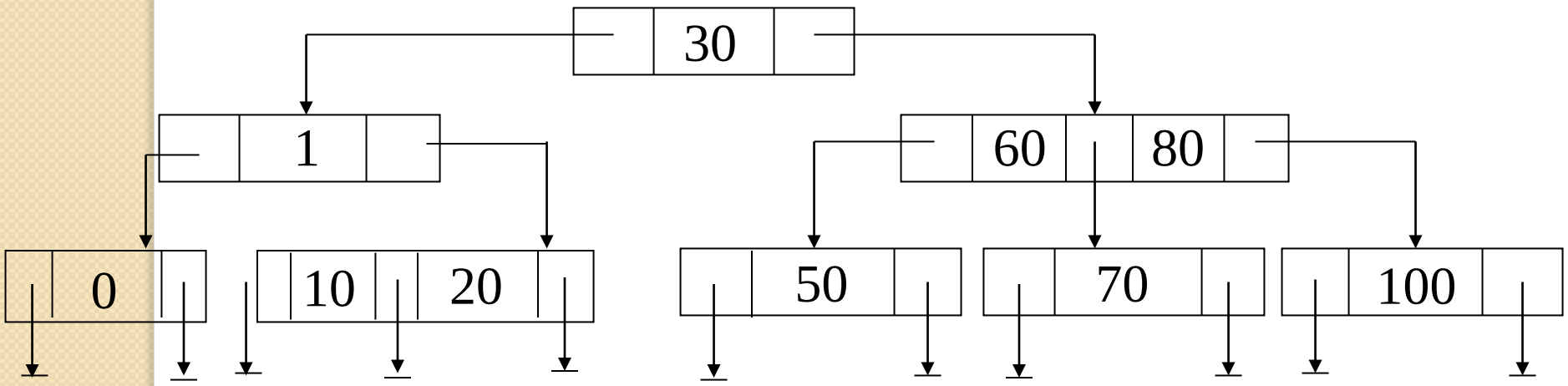


Примерно, добавяме **0**:





Изтриваме **40**: (замества се или с **30**, или с **50**)

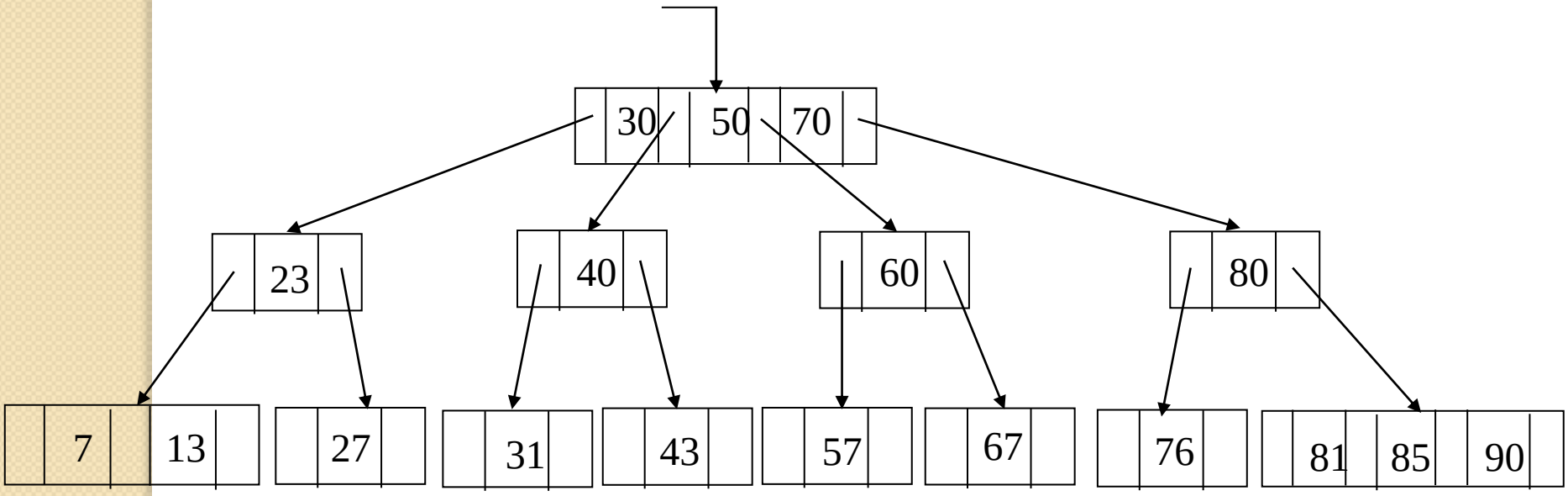


2-3-4 - дървета

2-3-4 - дърво е дърво T със следните свойства:

- T е идеално балансирано дърво;
- всеки вътрешен връх има 2, 3 или 4 наследника;
- всички пътища от корена до произволно листо имат еднаква дължина:

Например:



Съществуват ефективни алгоритми за работа с 2-3-4 -дървета, които гарантират сложност на основните операции (добавяне, търсене, изтриване) от порядъка на $O(\log_2 n)$. Съществуват начини за преобразуване на 2-3-4 -дървета в еквивалентни на тях двоични дървета или R-B -дървета, алгоритмите за които са по-прости.

(Преслав Наков, Панайот Добриков. *Програмираме = ++Алгоритми.* Top Team Co, София, 2002,
Flaming B., *Practical Data Structures in C++.* Azarona Software, 1993)

B-дървета или дървета на Байер и МакКрейт (Bayer&McCreight)

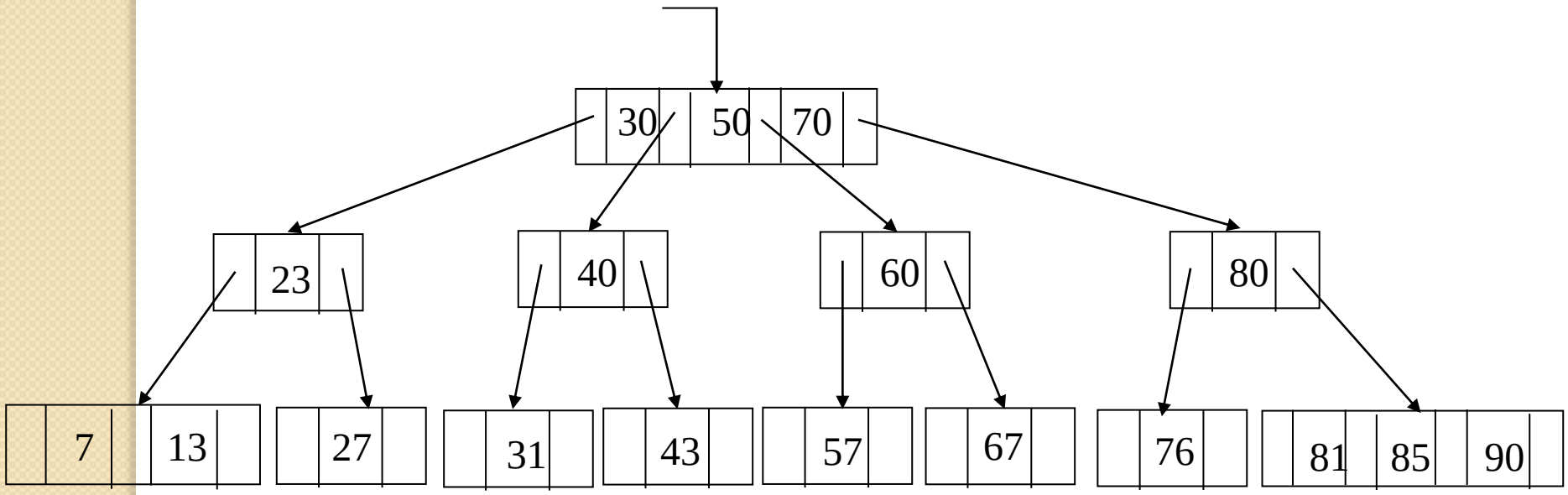
Това са дървета с много разклонения. B-дърво може да се разглежда като обобщение на 2-3 и 2-3-4-дървета.

B-дърво от ред ***m*** се нарича дърво за претърсване със следните свойства:

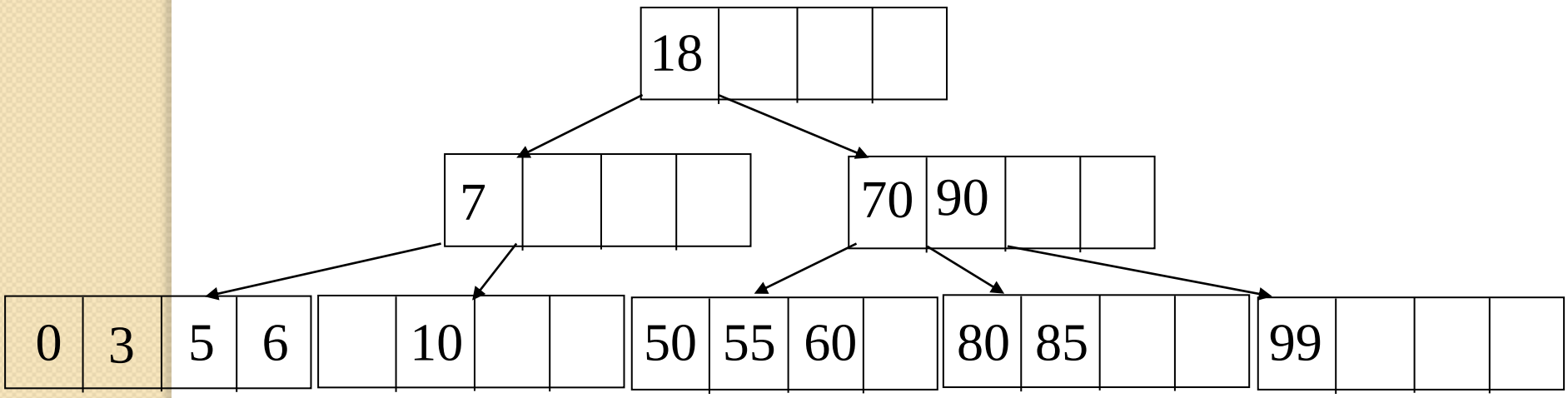
- За всички възли, освен корена, брой наследници ***i*** е
  m/2<= ***i*** <= ***m***
- За корена брой наследници ***i*** е
  2 <= ***i*** <= ***m***;
- Всички пътища от корена до произволно листо имат еднаква дължина.

Ако възел съдържа ***n*** полета с данни, той има ***i***=***n***+1 наследника. Максималния брой наследници е ***m***. Всеки вътрешен възел, който не е корен, има поне ***m***/2 връзки (наследника), ако ***m*** е четно, и поне (***m***+1)/2 връзки (наследника, ако ***m*** е нечетно, където ***m*** - е порядъка на дървото (или максималният брой връзки за всеки възел).

Разгледаното 2-3-4-дърво е B-дърво от 4 ред:



Примерно B-дърво от 5 ред:



Освен полета с данни, всеки елемент от тази структура пази също така и ***n*** - броя на полетата с данни в него. Например, за корена на дървото ***n***=1, за най-лявото листо ***n***=4.

Основните правила са:

- ключовете $k_0, k_1, \ldots k_{n-1}$, съхранявани във възела са в нарастващ ред:

$$k_0 < k_1, < \ldots < k_{n-1}$$

- съдържанието на всеки възел може да се представи като:

$$p_0, k_0, p_1\ k_1, \ldots p_{n-1}, k_{n-1}, p_n$$

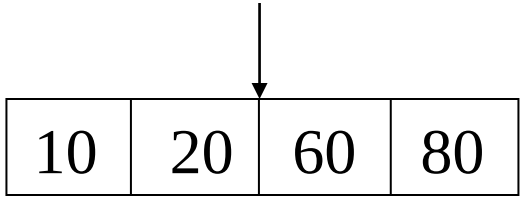
- ако възелът е листо, всички негови указатели $p_0, p_1, \ldots p_{n-1}, p_n$ са NULL.
- ако възелът не е листо, всеки от неговите $n+1$ указатели p_i сочи към наследник. За $i = 0, 1, 2, \ldots, n$ всички ключове в наследника, сочени от p_i, са по-големи от k_{i-1}. Също така, за $i = 0, 1, 2, \ldots, n-1$, всички ключове в наследника, сочен от p_i, са по-малки от k_i.
- ако вътрешен възел съдържа n ключа (стойности), той съдържа точни $n+1$ връзки (указатели), което означава, че $m-n-1$ указатели в този възел не се използват.

B-дървета са удобни и ефективни в много приложения. За тях са разработени голямо количество алгоритми.

При тях, за разлика от идеално-балансирани дървета, всички листа се намират на едно и също ниво.

Нека разгледаме добавянето на нови елементи в B-дърво с порядък 5 (***m***=5):

Да започнем с празно B-дърво, в което ще вмъкваме четири цели числа - например 60, 20, 80 и 10:

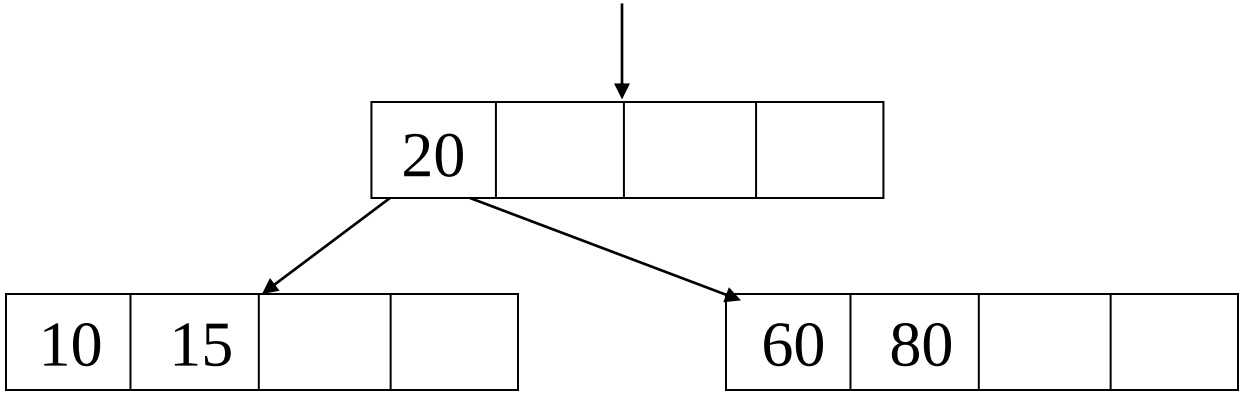


n=4

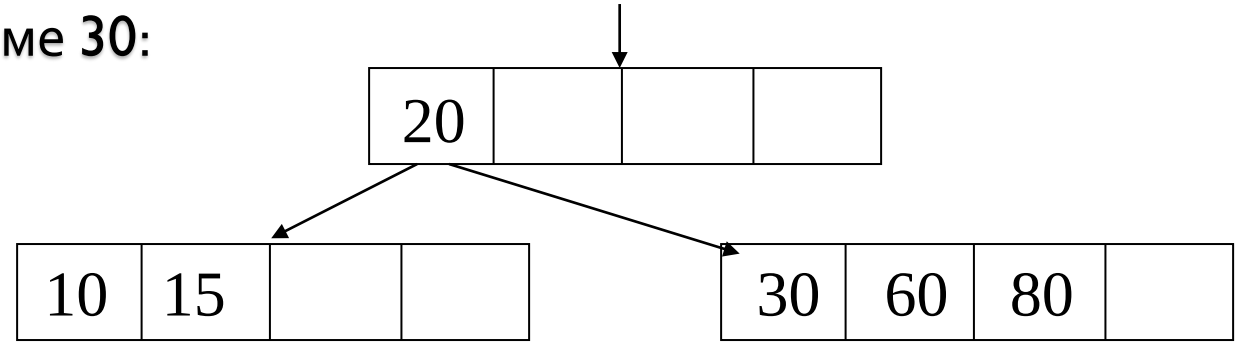
P_0= P_1= P_2= P_3= P_4= NULL

k_0=10, k_1=20, k_2=60, k_3=80

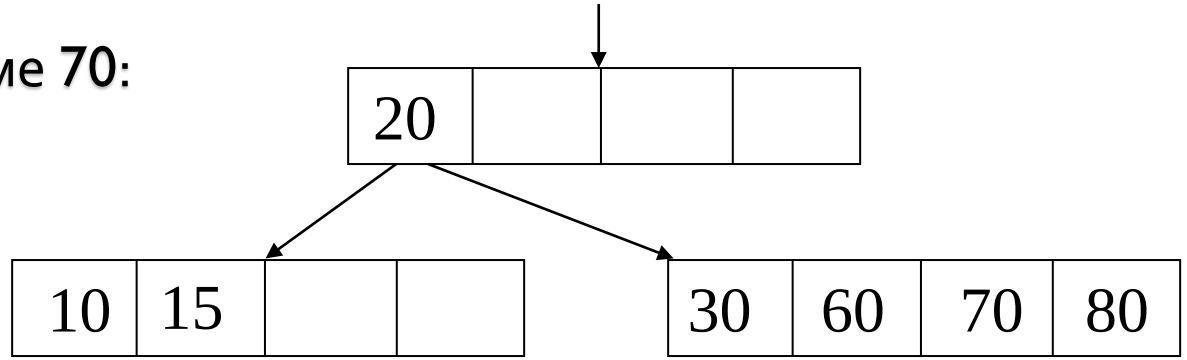
Добавяме стойност 15:



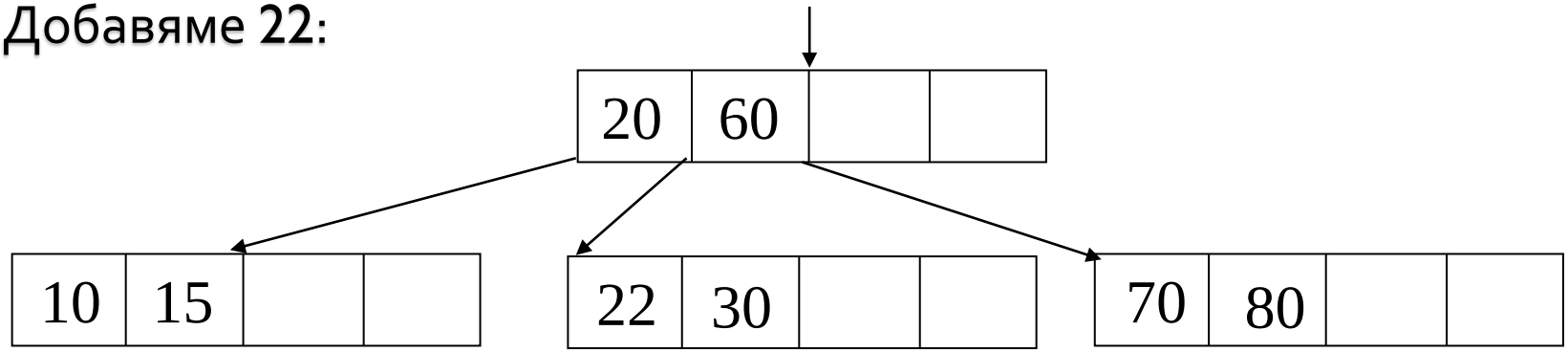
Добавяме **30**:



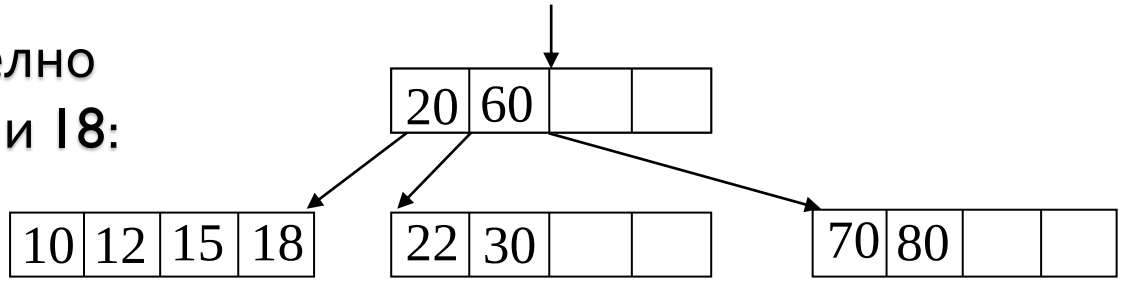
Добавяме **70**:



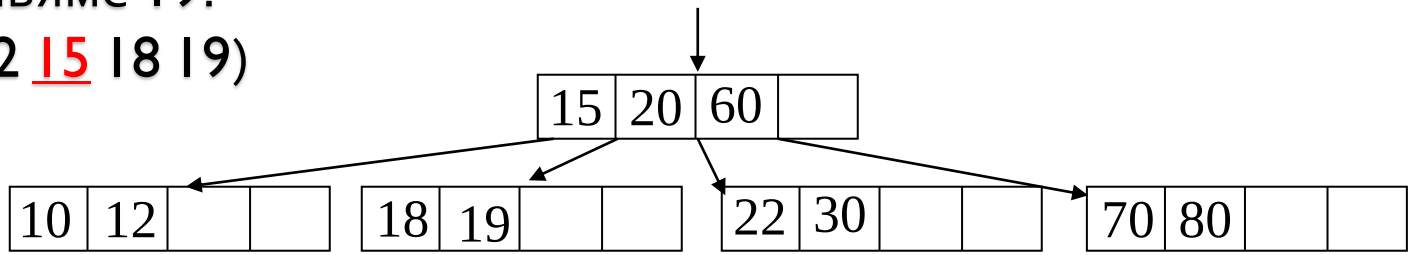
Добавяме **22**:



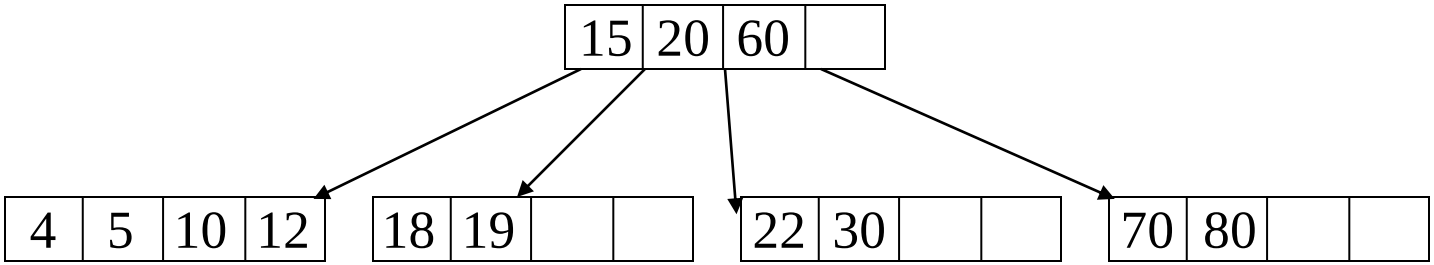
Последователно добавяме 12 и 18:



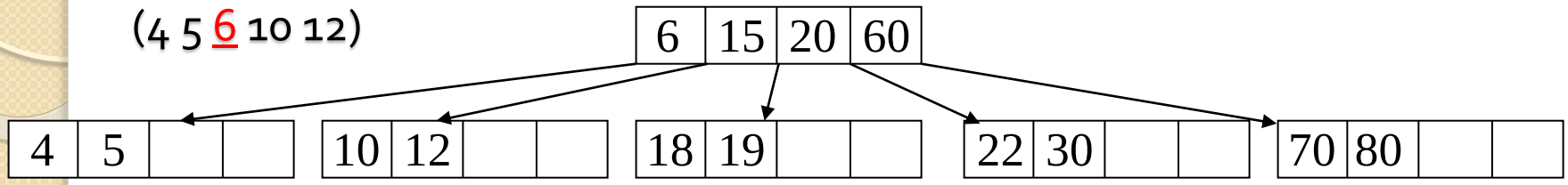
Добавяме 19:
(10 12 15 18 19)



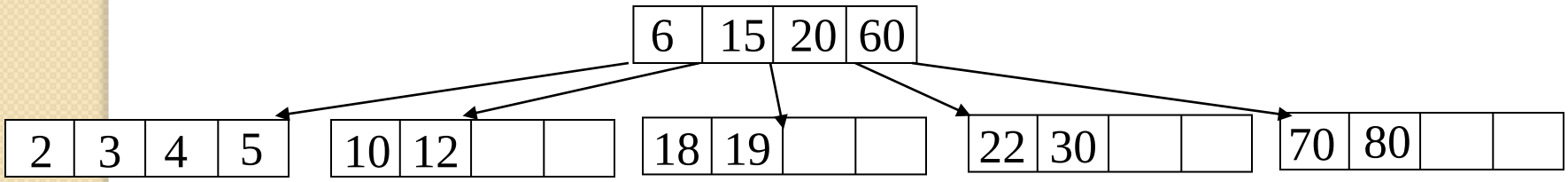
Добавяме 4 и 5:



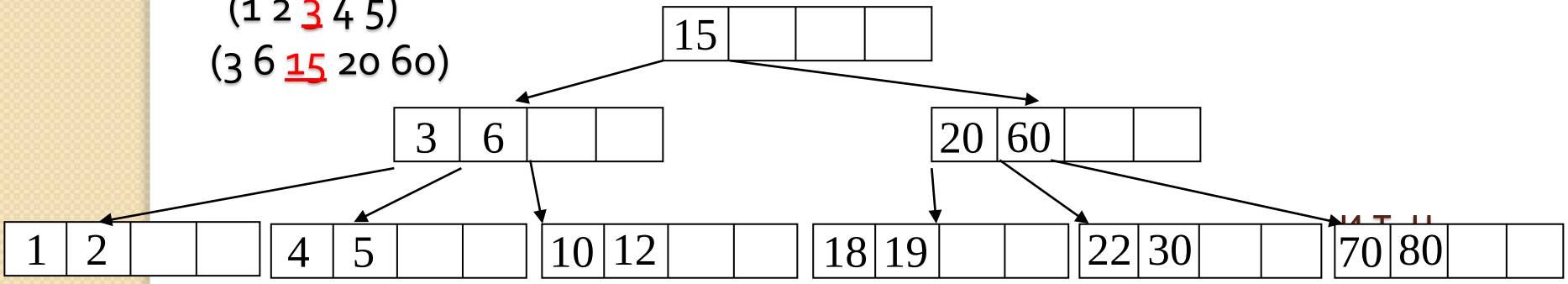
Продължаваме да добавяме стойности в най-левия клон.
Например, 6:
(4 5 6 10 12)



Добавяме последователно 2 и 3:

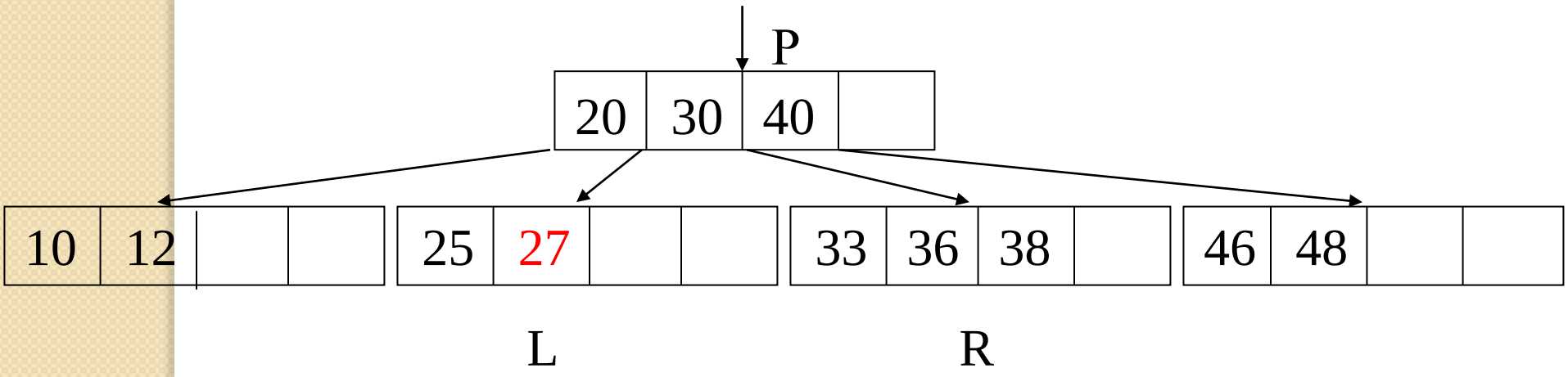


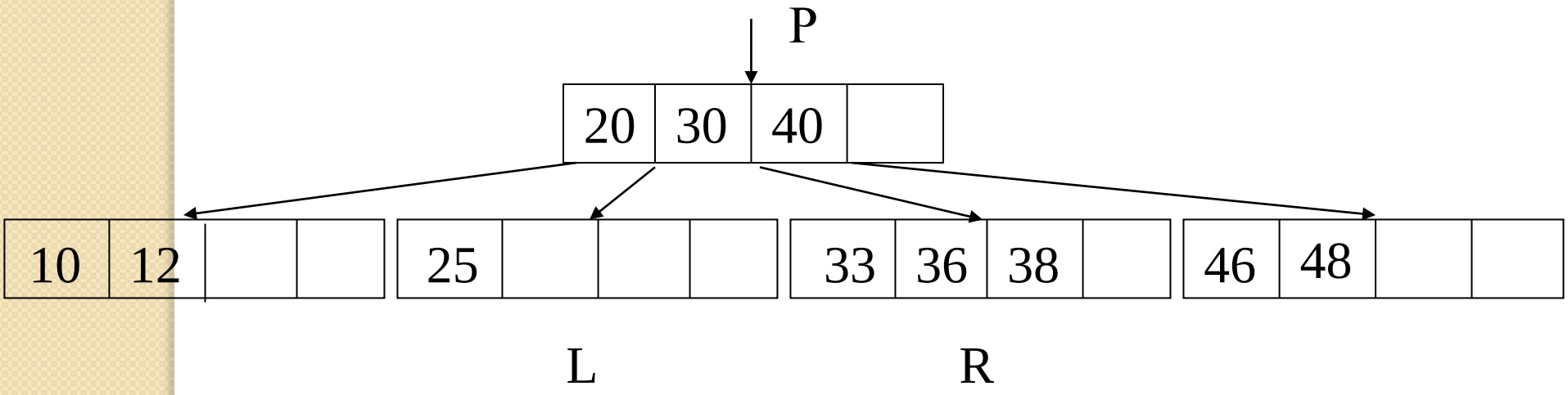
Добавяме 1:
(1 2 3 4 5)
(3 6 15 20 60)



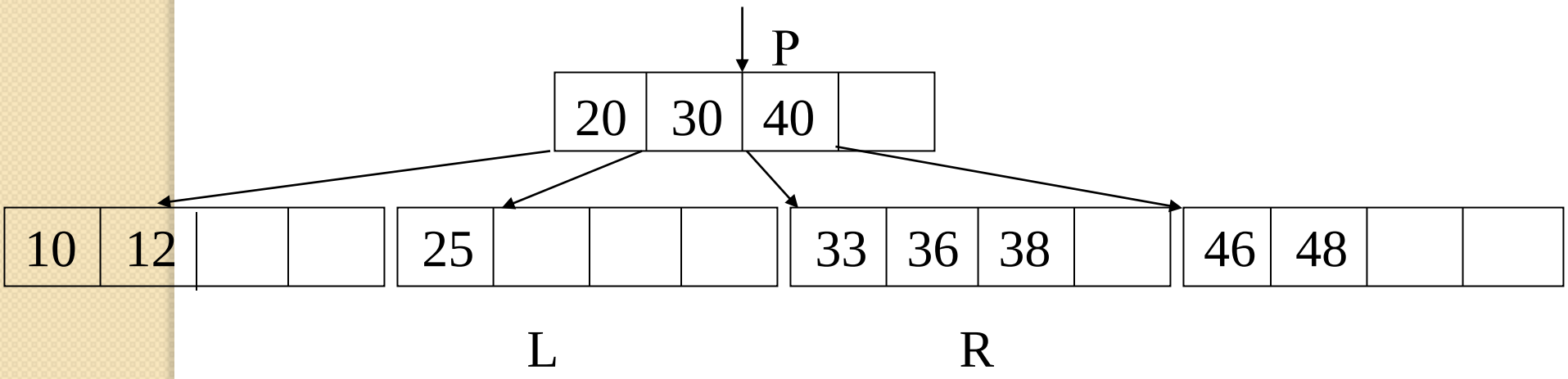
Изтриване на възли в B-дърво

Пак се разглежда B-дърво от 5 ред, така че всеки възел съдържа най-много 4 полета с данни и 5 указателя:

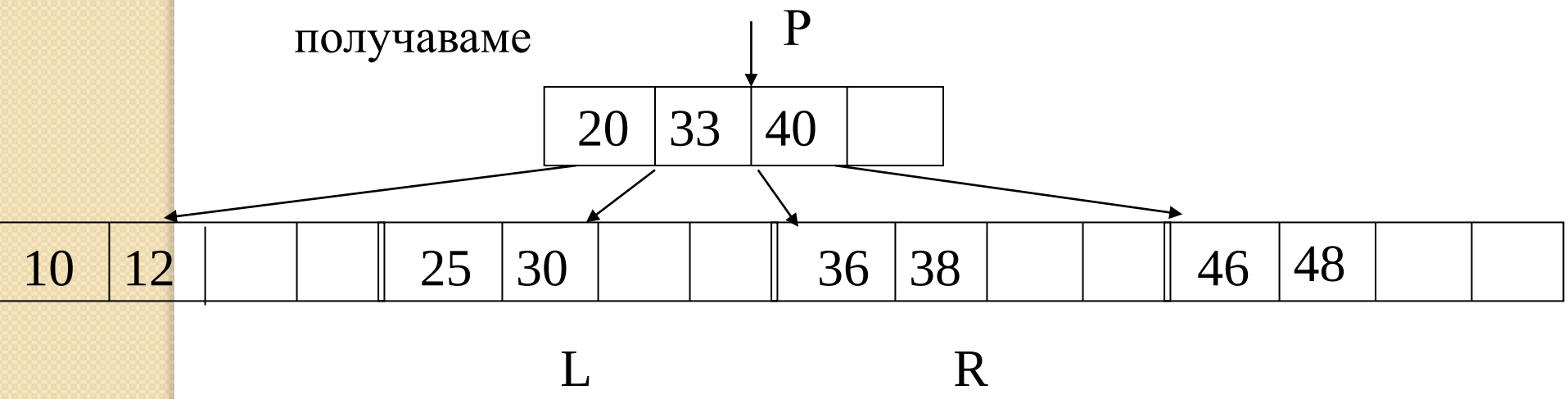




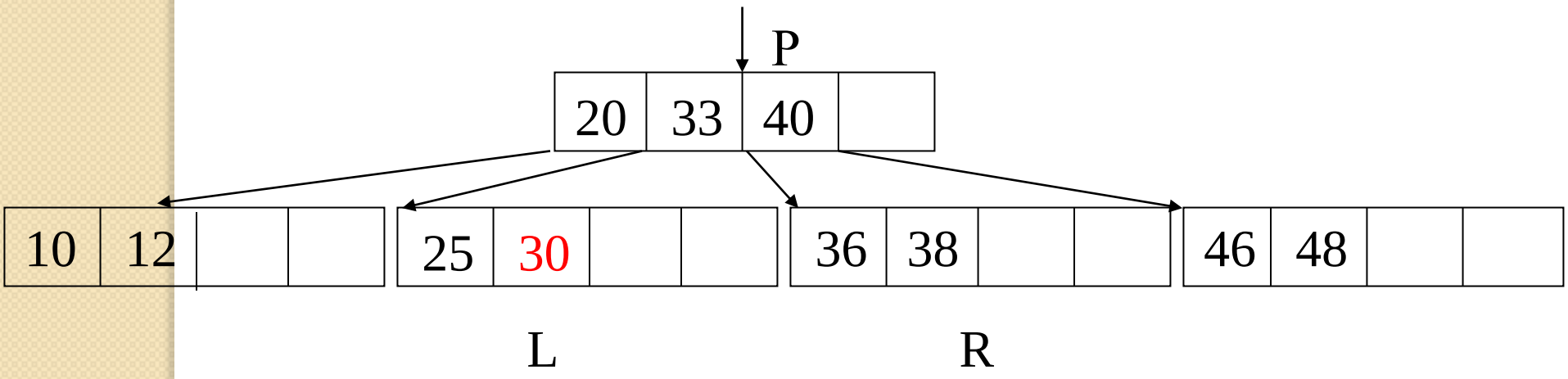
Т. е. от дървото



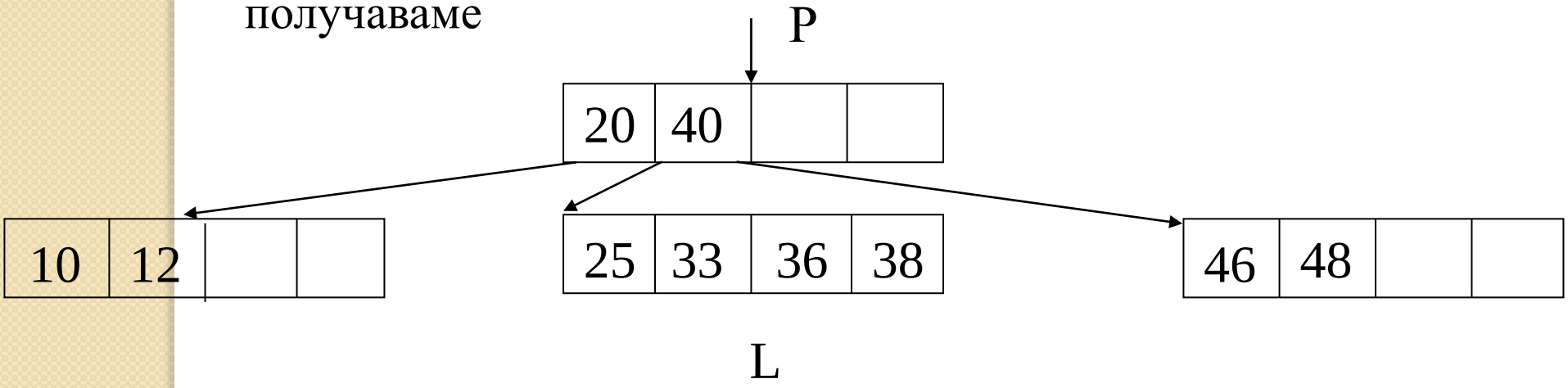
получаваме



Например



получаваме



Подробно - Лендерт Амерал, *Алгоритми и структури от данни в C++,* ИК СОФТЕХ, София, 2001

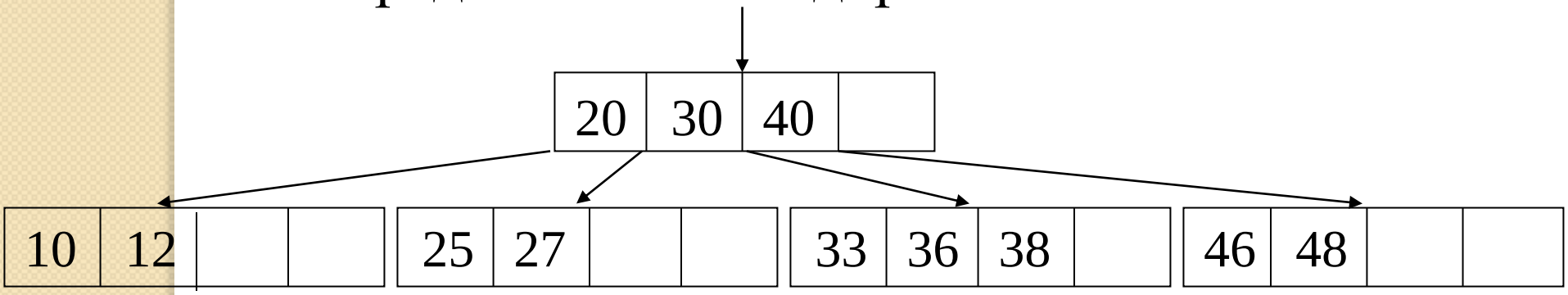
B-дървета са идеално балансирани, височината им h е най-много:

$$h=\log_{[m/2]}((V+1)/2)$$

което при голямо m гарантира добра височина даже при огромно V.

(m - порядъка на B-дървото
V - брой възли в него)

Статично представяне на B-дърво



получаваме

индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
стойност	3	20	30	40		2	10	12			2	25	27			3	33	36	38		2	46	48		

брой ключови полета във възела

За B-дърво от ***m***-ти ред е необходима памет за ***V*m*** елемента.
V- брой възли в дървото.

Представяне на B-дървета върху диск - Лендерт Амерал, *Алгоритми и структури от данни в C++,* ИК СОФТЕХ, София, 2001

Приложение на B-дървета:

- в системи за управление на голям обем информация
  - релационни БД
  - обектно-релационни БД (Oracle)
- в операционни системи (например, Windows)
  - странична организация на паметта
  - организация на файловата система

Допълнителна литература:

1. Н.Уирт, *Алгоритми + структури от данни = програми*, “Техника, София, 1980

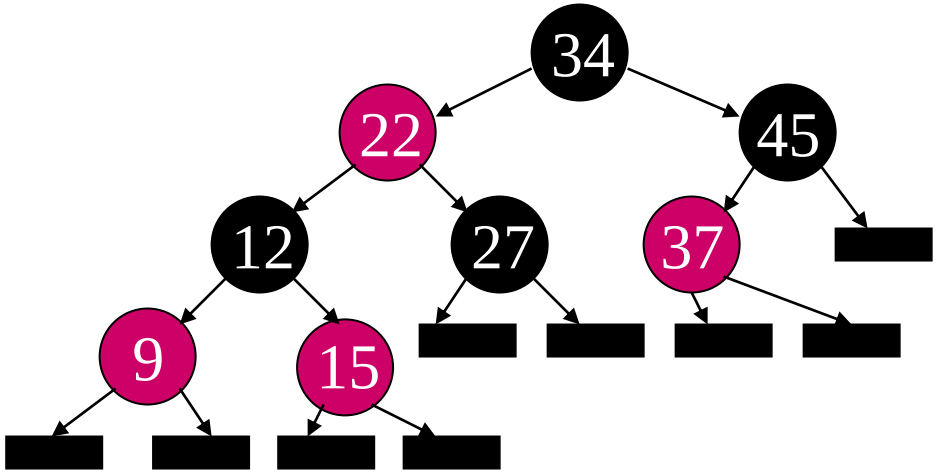
2. Д.Шишков и др., *Структури от данни и алгоритми*, “Интеграл”, Добрич, 1995

3. D. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1998.

Червено-черни дървета (Red-Black Trees)

Червено-черно дърво се нарича балансирано (но не идеално балансирано!) двоично дърво за претърсване (BST), в което всеки връх е означен или като червен, или като черен и допълнително са изпълнени следните свойства:

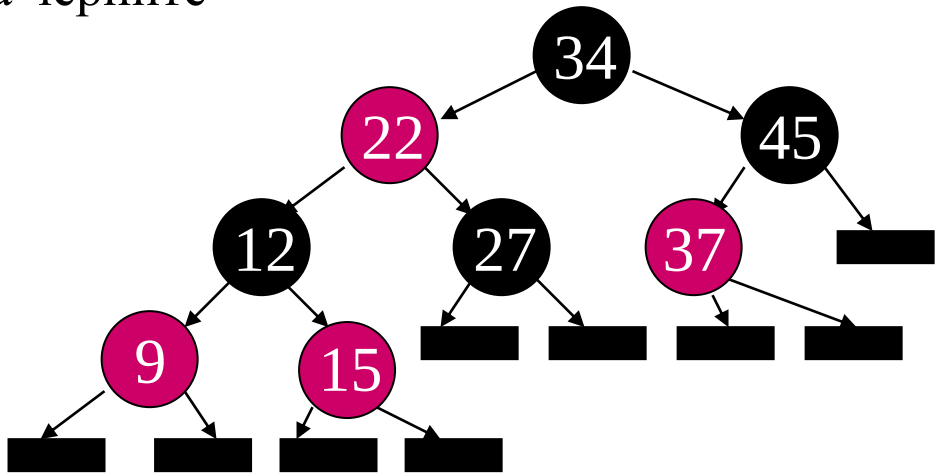
1. Всеки връх или е червен, или е черен;
2. Коренът е черен;
3. Всички външни възли (върхове със стойност NULL) са черни;
4. Ако даден връх е червен, то двата му наследника са черни, освен това всеки червен възел има черен родител;
5. Всички пътища от произволен връх T до произволно листо от поддървото с корен T съдържат еднакъв брой черни върхове.



В нито един от пътищата на R-B-дърво няма два последователни червени възела!

Нека е даден път от възел T до листо, който съдържа ***n*** черни възела.

От правило 4 следва, че броят на червените възли в този път не може да превишава броя на черните



(т. к. за всеки червен трябва да има по един черен родителски възел). Така дължината на пътя е между ***n*** и ***2n***. От правило 5 обаче следва, че всички останали пътища от този възел до листата (до които може да се стигне от него) съдържат също **n** черни възела, т.е. и тяхната дължината е между ***n*** и ***2n***. Поради това височината на цялото дърво (с корен T) е най-много ***2n***. Затова числото ***n*** се нарича полувисочина.

От (5) също така следва, че полувисочините на лявото и дясното поддърво на дадено дърво са равни.

С елементарна индукция по ***n*** може да се докаже, че ако полувисочината на дадено дърво е n, то броят на елементите ***N*** в това дърво е поне 2^n-1 .

Следователно

$$n <= log_2(N+1)$$

или

височината на дървото $<=2\ log_2(N+1)$, т. е. правилното оцветяване гарантира балансираност на дървото.

Подробни описания на алгоритми за работа с R-B-дървета:

1. Робърт Седжуик, *Алгоритми на C, том 2*, Софтпрес, С., 2002

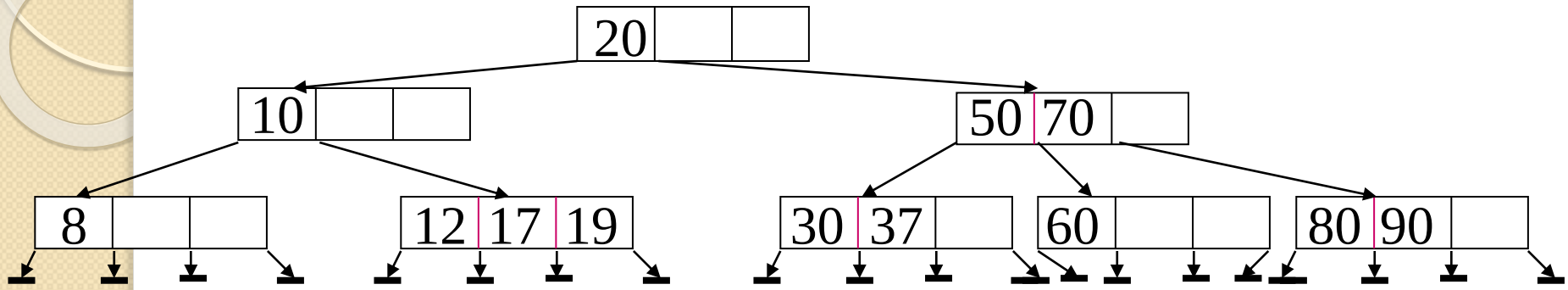
2. Д. Шишков и др., *Структури от данни и алгоритми*, "Интеграл", Добрич, 1995

R-B-дървета имат и още две основни свойства:

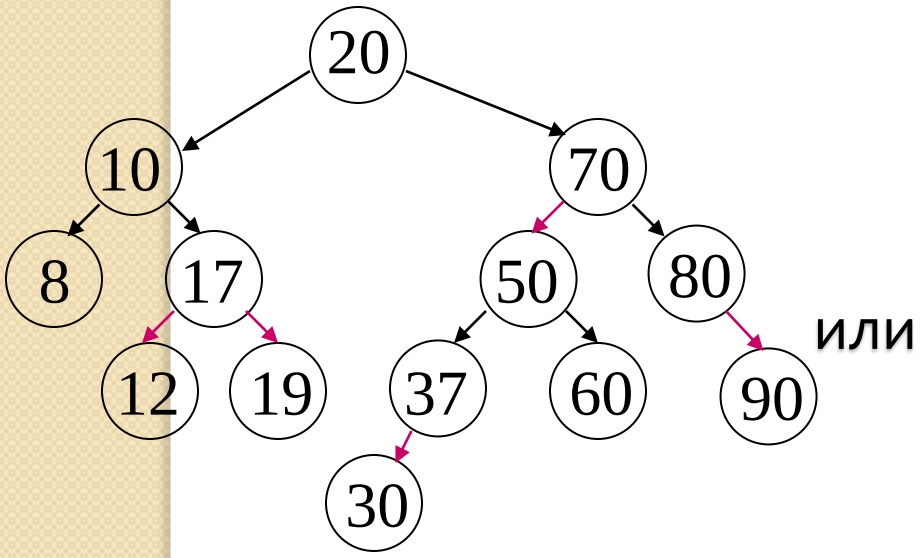
1. Вече разгледаната функция за търсене в BST за тях работи без промени;
2. Те съответстват директно на 2-3-4 дървета, така че всички алгоритми за последните са приложими и за тях.

Защо?

Например имаме следното **2-3-4** - дърво:



Дървото може да бъде преобразувано така:



или