ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

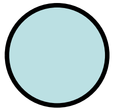
Search trees

АПС

Algorithms and data structures

- lectures -

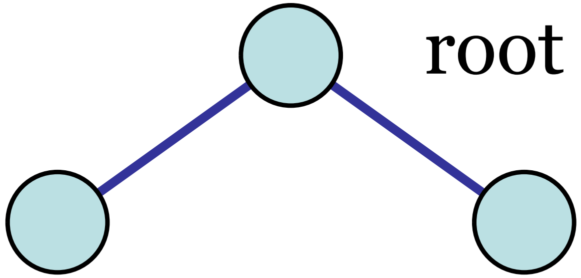
Number of nodes and height of a binary tree



root - level 0

1

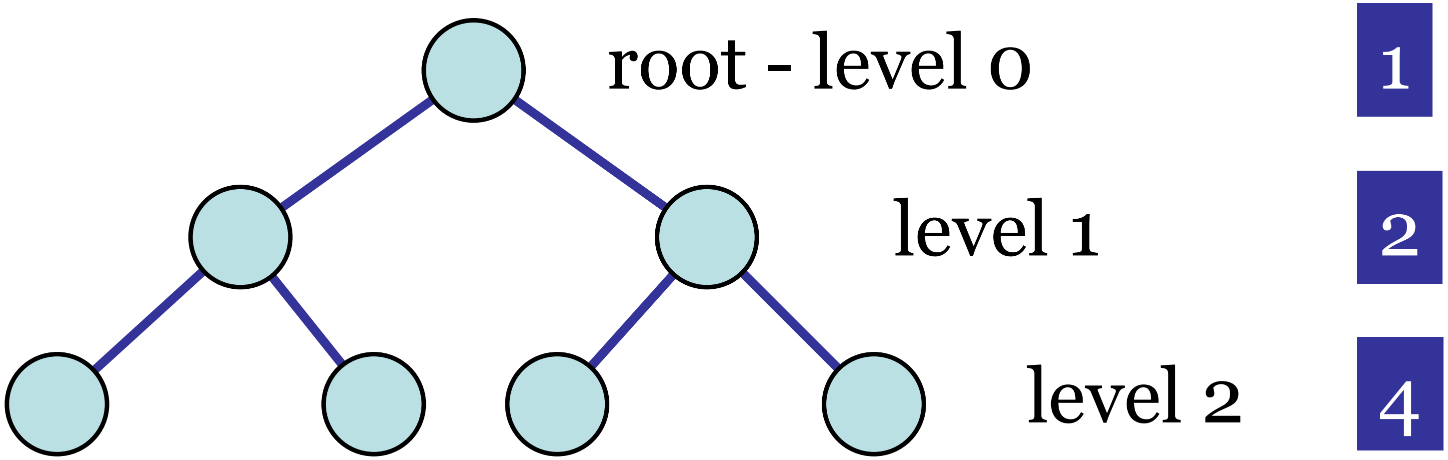
Number of nodes and height of a binary tree



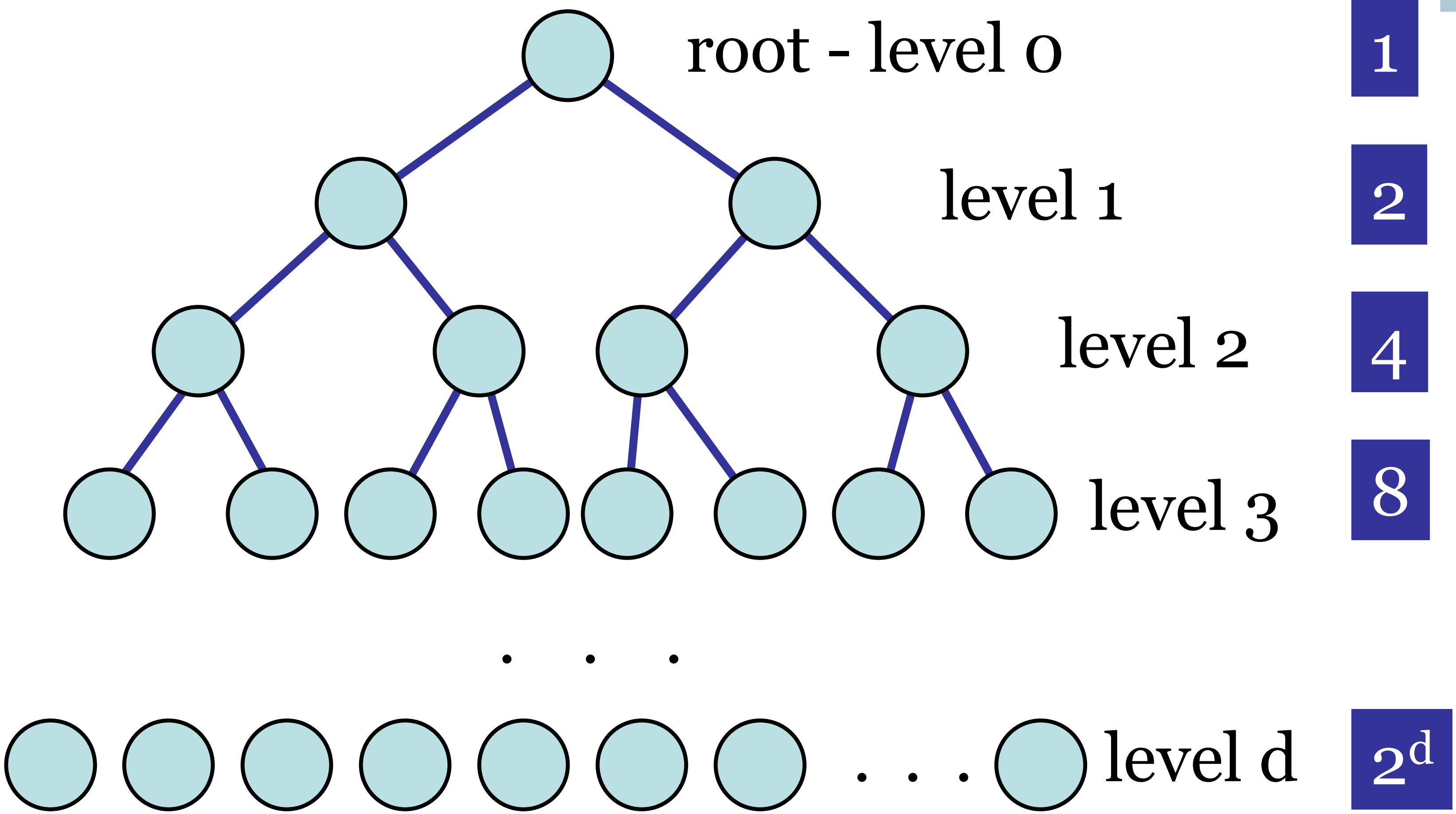
root - level 0 — 1

level 1 — 2

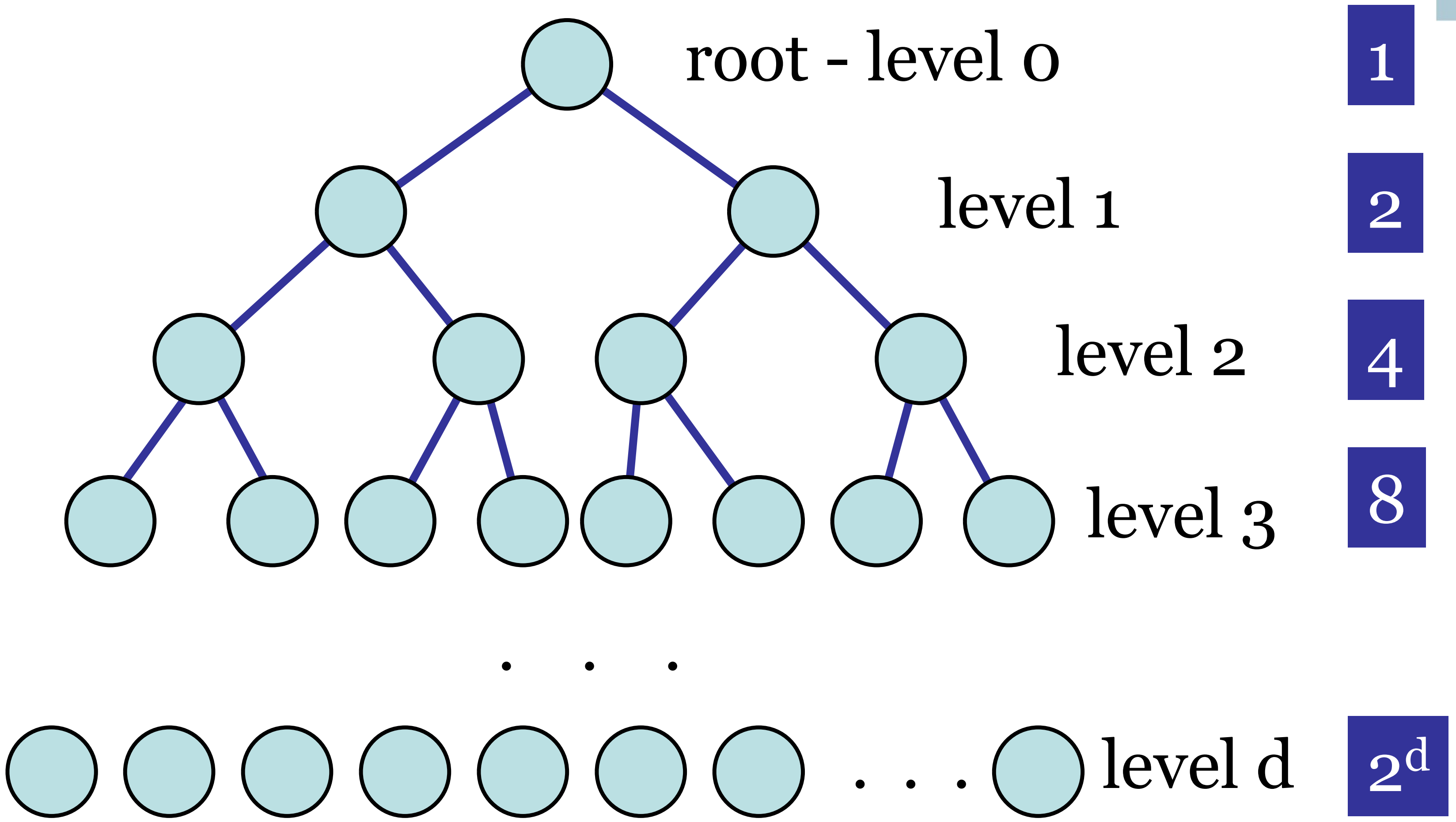
Number of nodes and height of a binary tree



Number of nodes and height of a binary tree



Number of nodes and height of a binary tree



$$1 + 2 + 4 + \ldots + 2^d = 2^{d+1} - 1$$

Number of nodes and height of a binary tree

- if there is a binary tree with *n* nodes and tree depth *d*, then

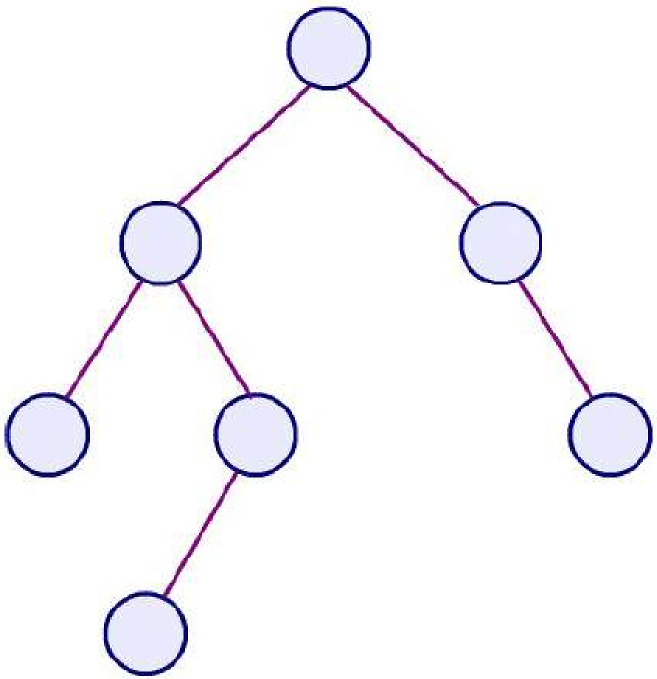
$$n \leq 2^d - 1$$

- the minimal depth of the binary tree with n nodes can be calculated with

$$d_{min} = \lceil \log_2(n + 1) \rceil$$

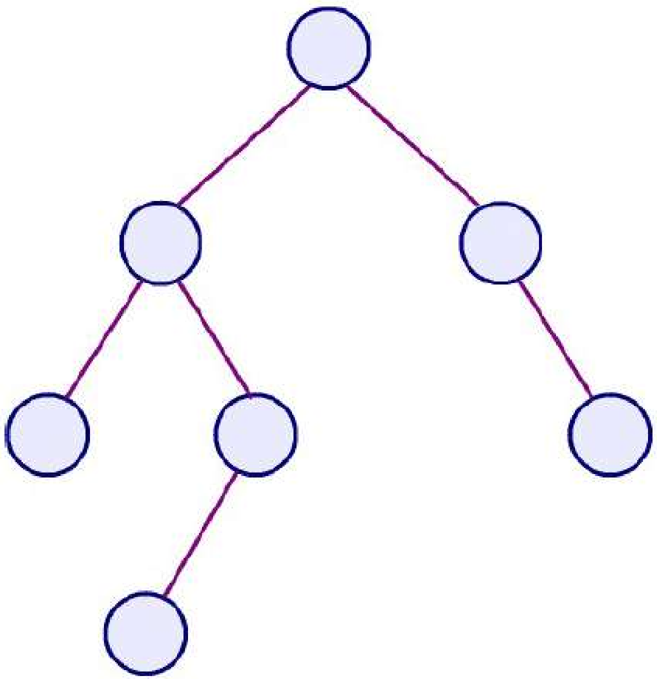
Balanced binary tree

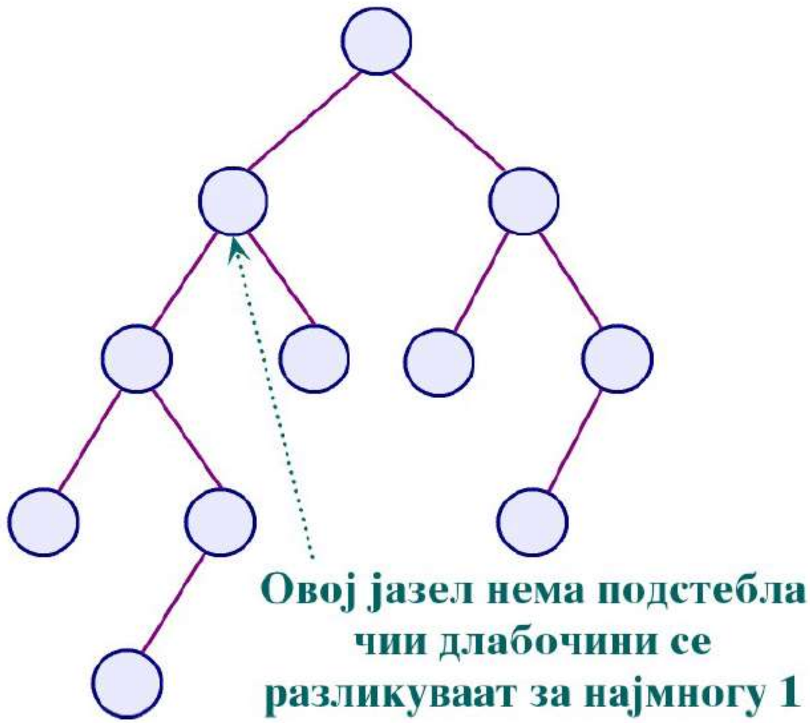
- A balanced binary tree is a binary tree where for each node in the tree, the heights of its left and right subtrees **do not differ by more than one**.



Balanced binary tree

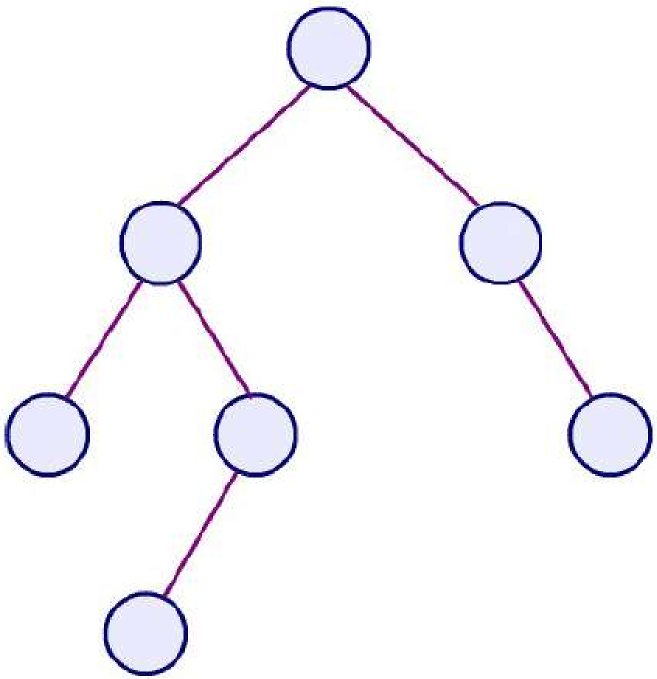
- A balanced binary tree is a binary tree where for each node in the tree, the heights of its left and right subtrees **do not differ by more than one**.



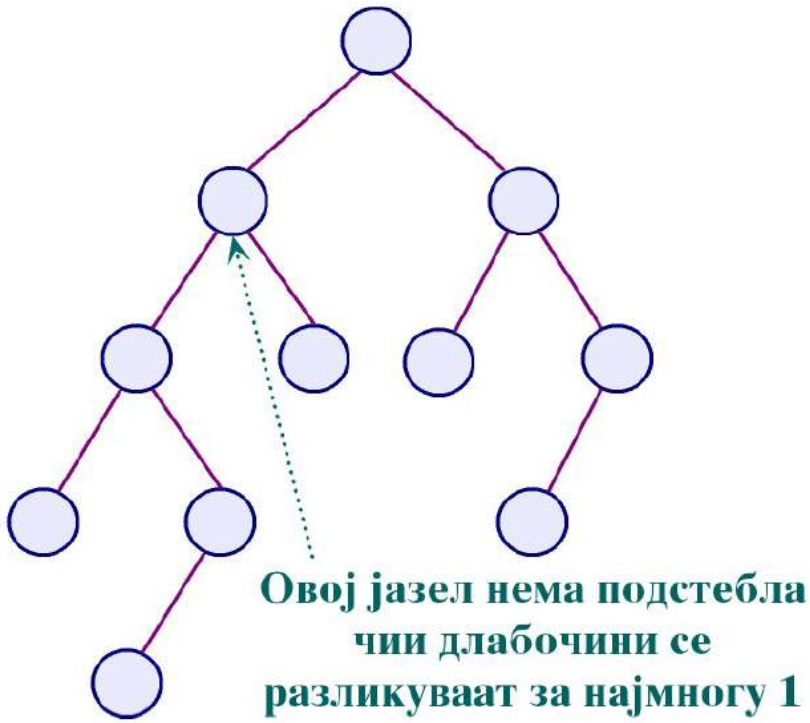


Balanced binary tree

- A balanced binary tree is a binary tree where for each node in the tree, the heights of its left and right subtrees **do not differ by more than one**.



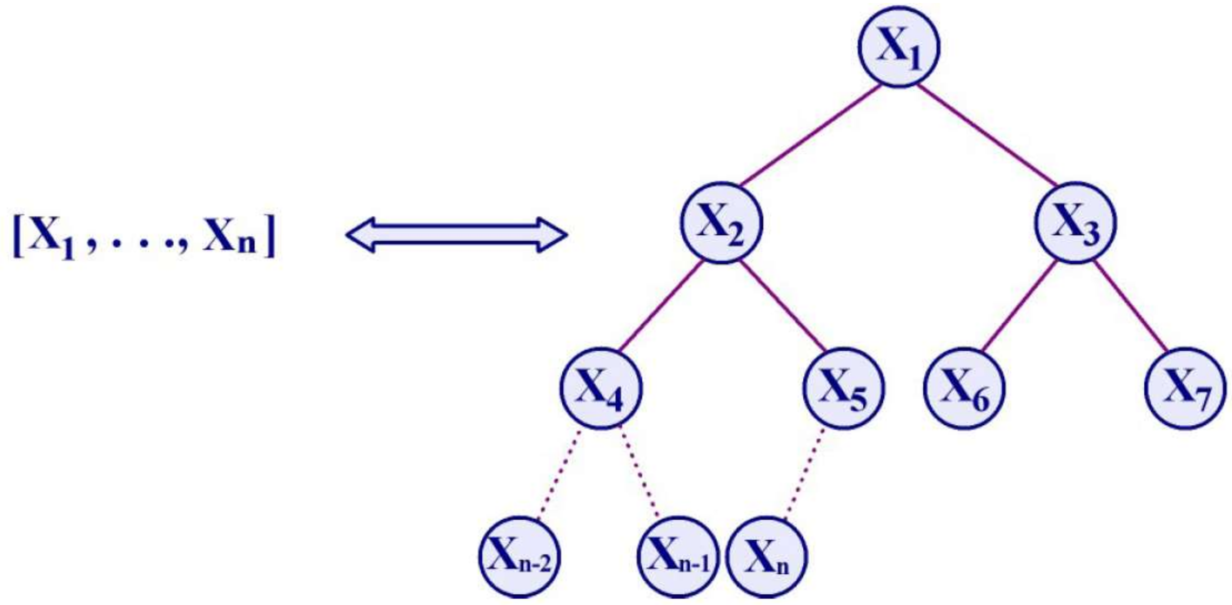
balanced



not balanced

Balanced binary tree application - Heap tree

- A given sequence (x_1, ...,x_n) can be represented in a form of a binary tree
- In doing so, the tree is filled from the root to the leaves in a way that it will be maximally filled



Balanced binary tree application - Heap tree

A heap tree is a complete binary tree for which the key value of the parent node is greater than or equal to the key value of its children, for each node in the tree

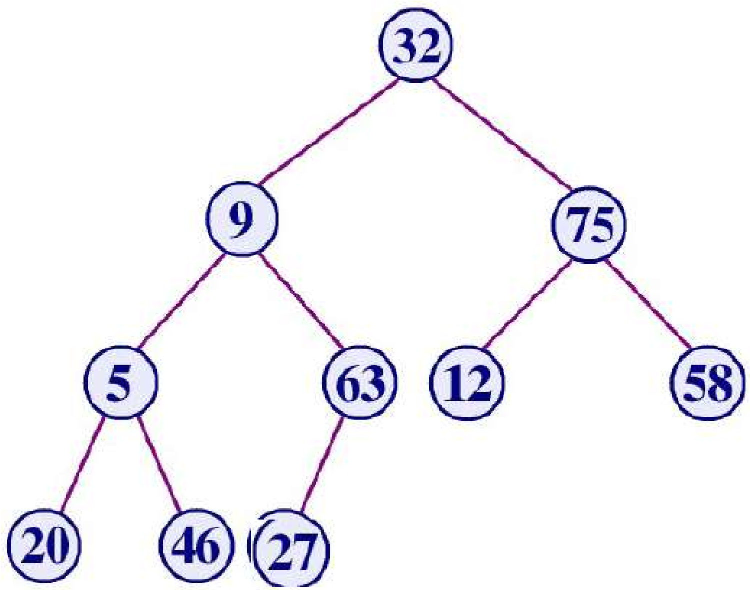
- We will show that inserting and deleting an element in such a tree has complexity O(logN)
- Heap trees are used in the implementation of efficient algorithms for sorting and implementation of **priority lists**

Binary tree application - Heap sort

- Steps in heap sort algorithm:
  - Create a heap tree
  - Until the heap tree is empty
    - place the entry (key) from the root of the heap tree into the resulting sorted array
    - remove that element from the heap tree
    - form a heap tree again
    - **Each node R_j has children R_{2j} and R_{2j+1}.**
- Heap trees / Heap sort visualization
  - https://www.cs.usfca.edu/~galles/visualization/Heap.html

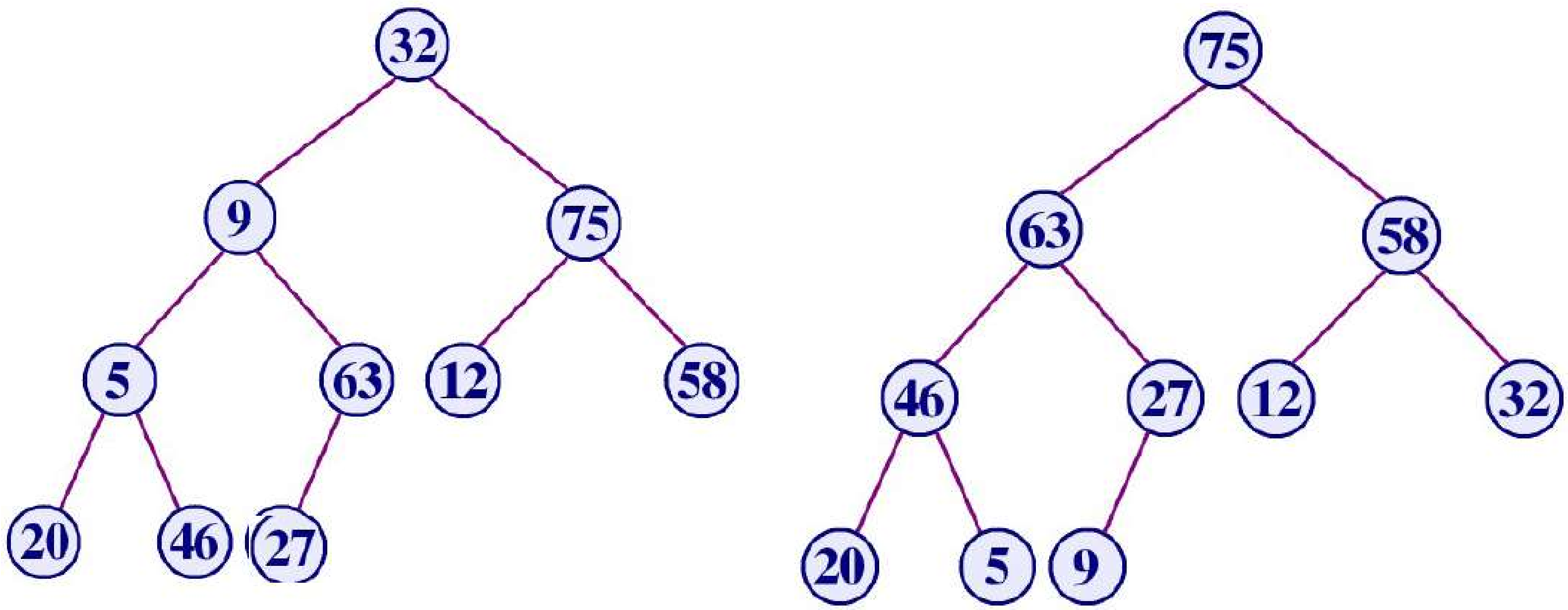
Binary tree application - Heap sort

Input array: 32, 9, 75, 5, 63, 12, 58, 20, 46, 27



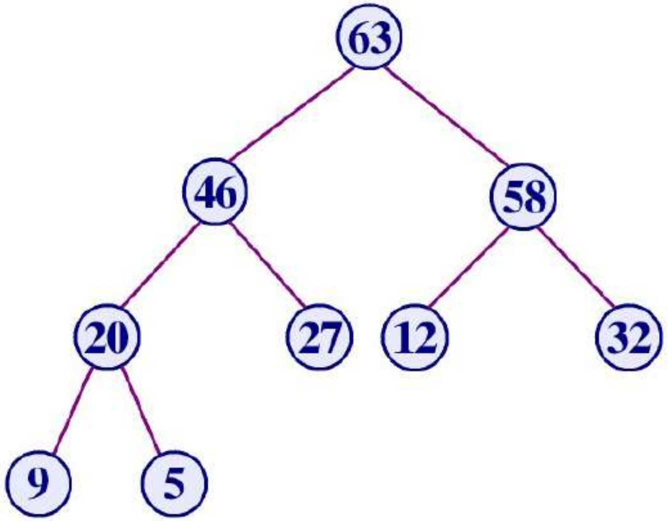
Binary tree application - Heap sort

Input array : 32, 9, 75, 5, 63, 12, 58, 20, 46, 27



Initial heap tree

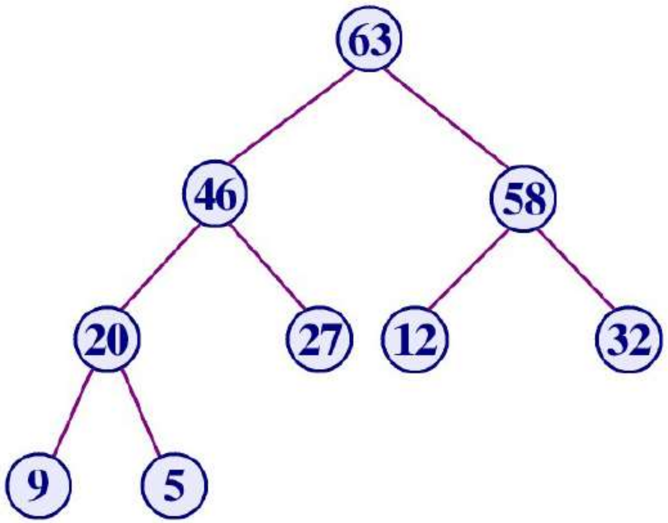
Binary tree application - Heap sort

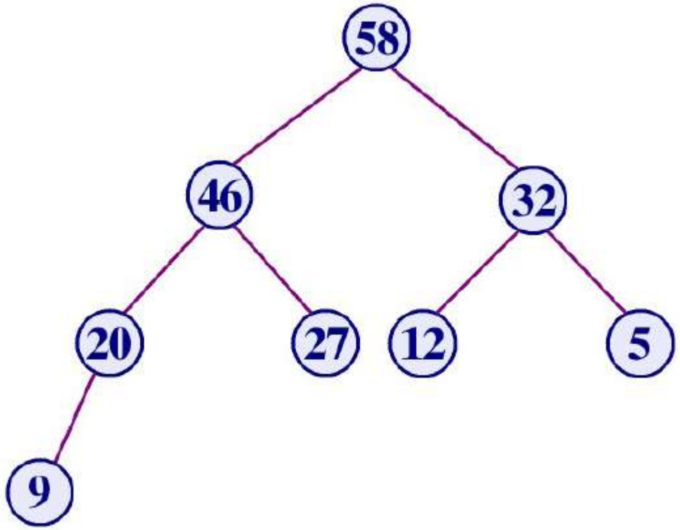


Сортирано: **75**

Heap големина: **i = 9**

Binary tree application - Heap sort

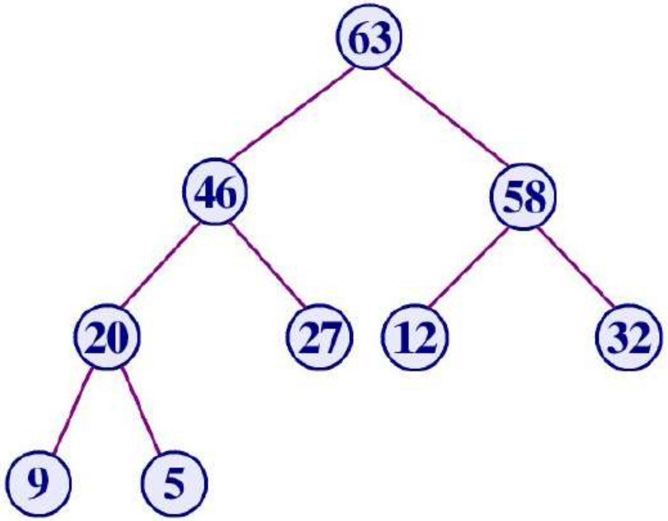




Сортирано: 75 — 63, 75

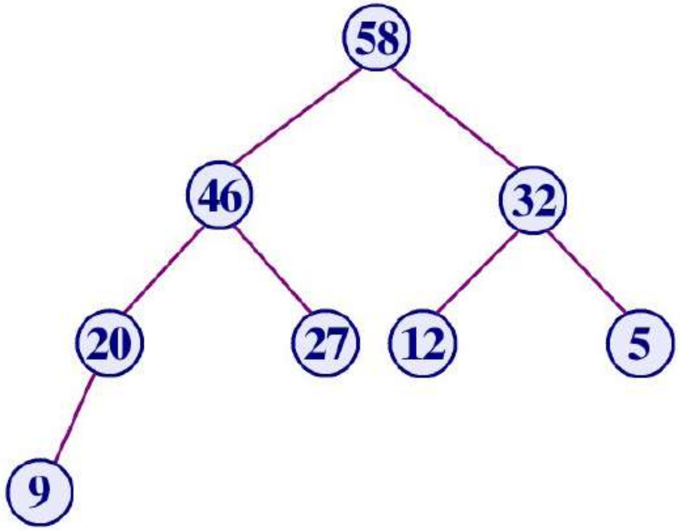
Heap големина: i = 9 — i = 8

Binary tree application - Heap sort



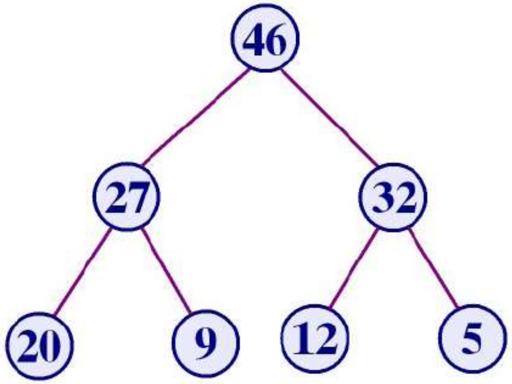
Сортирано: **75**

Heap големина: **i = 9**



63, 75

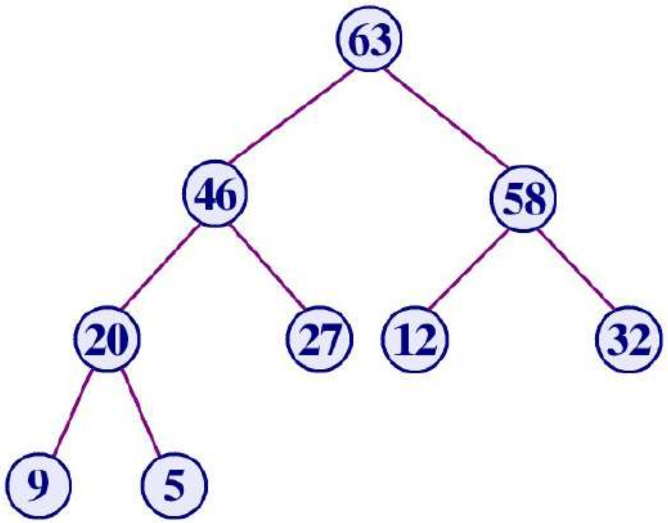
i = 8



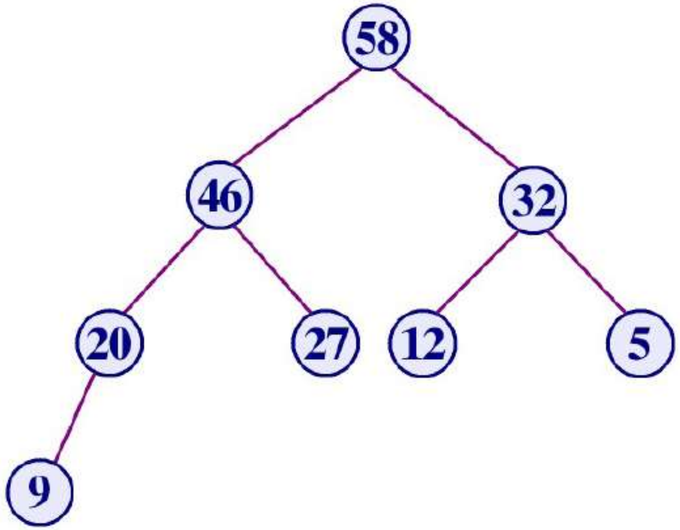
Сортирано: **58 , 63, 75**

Heap големина: **i = 7**

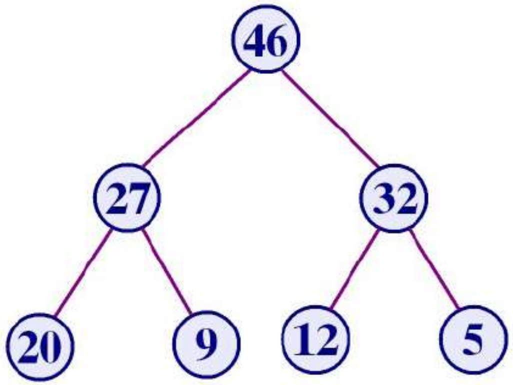
Binary tree application – Heap sort



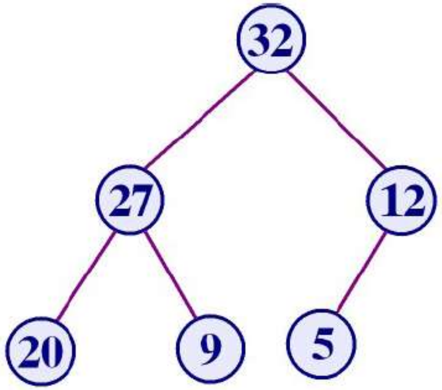
Сортирано: 75
Heap големина: i = 9



63, 75
i = 8

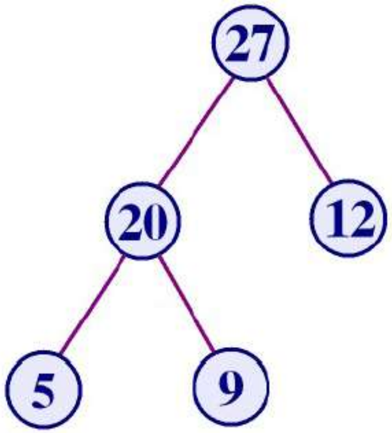


Сортирано: 58 , 63, 75
Heap големина: i = 7



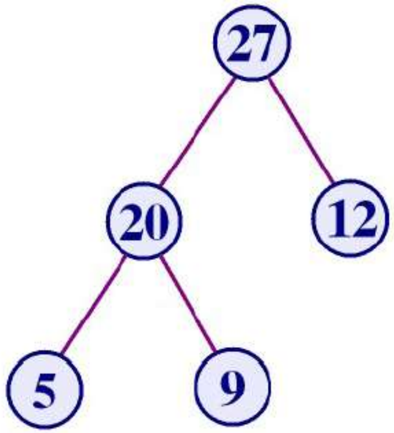
46, 58, 63, 75
i = 6

Binary tree application - Heap sort

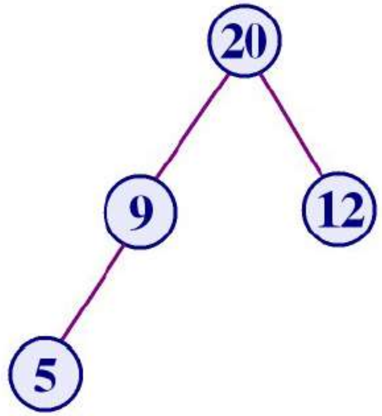


Сортирано: 32, 46, 58 , 63, 75
Heap големина: i = 5

Binary tree application - Heap sort

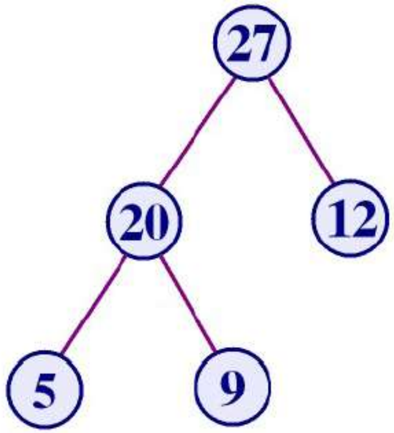


Сортирано: 32, 46, 58 , 63, 75
Heap големина: i = 5

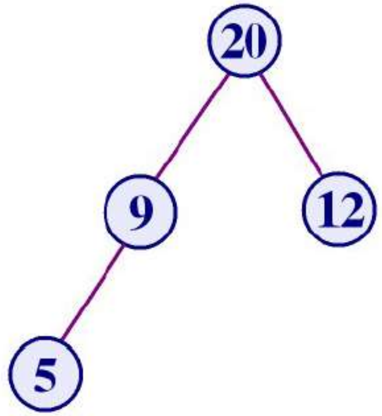


27, 32, 46, 58, 63, 75
i = 4

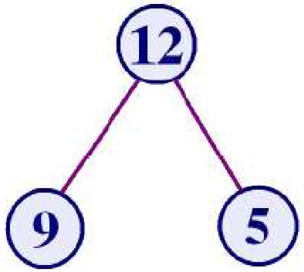
Binary tree application - Heap sort



Сортирано: 32, 46, 58 , 63, 75
Heap големина: i = 5

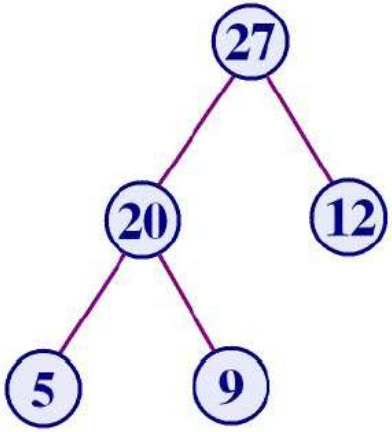


27, 32, 46, 58, 63, 75
i = 4

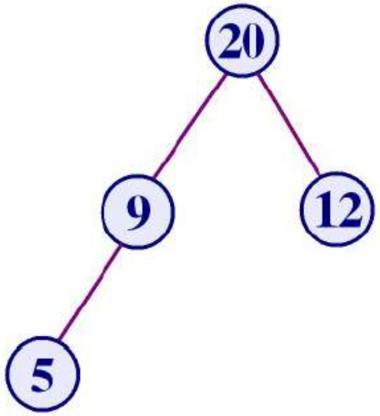


Сортирано: 20, 27, 32, 46, 58, 63, 75
Heap големина: i = 3

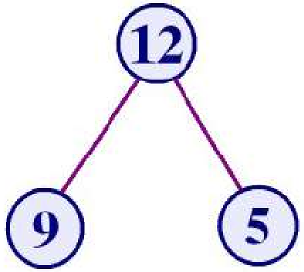
Binary tree application - Heap sort



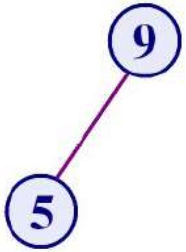
Сортирано: 32, 46, 58 , 63, 75
Heap големина: i = 5



27, 32, 46, 58, 63, 75
i = 4

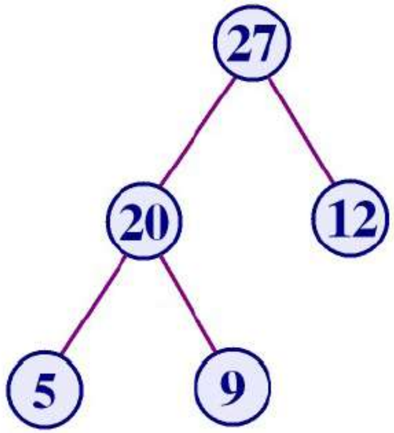


Сортирано: 20, 27, 32, 46, 58, 63, 75
Heap големина: i = 3

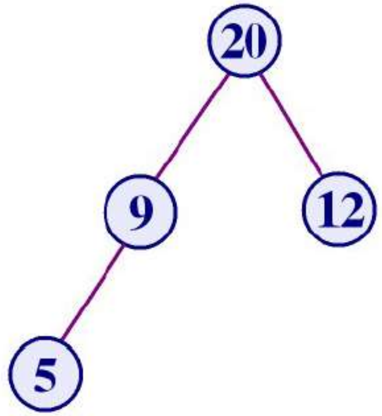


12, 20, 27, 32, 46, 58, 63, 75
i = 2

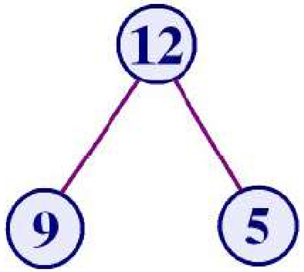
Binary tree application - Heap sort



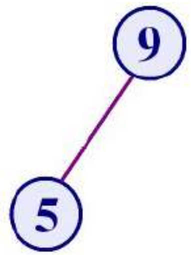
Сортирано: 32, 46, 58 , 63, 75
Heap големина: i = 5



27, 32, 46, 58, 63, 75
i = 4



Сортирано: 20, 27, 32, 46, 58, 63, 75
Heap големина: i = 3



12, 20, 27, 32, 46, 58, 63, 75
i = 2

Сортирано: 5, 9, 12, 15, 27, 32, 46, 58, 63, 77
резултат

Binary tree application - Heap sort

```
procedure ADJUST (i,n)
R ← Ri; K ← Ki; j ← 2i
while j ≤ n do
    if j < n and Kj < Kj+1 then j ← j + 1
    if K ≥ Kj then exit
    R⌊j/2⌋ ← Rj; j ← 2j
end
R⌊j/2⌋ ← R
end ADJUST
```

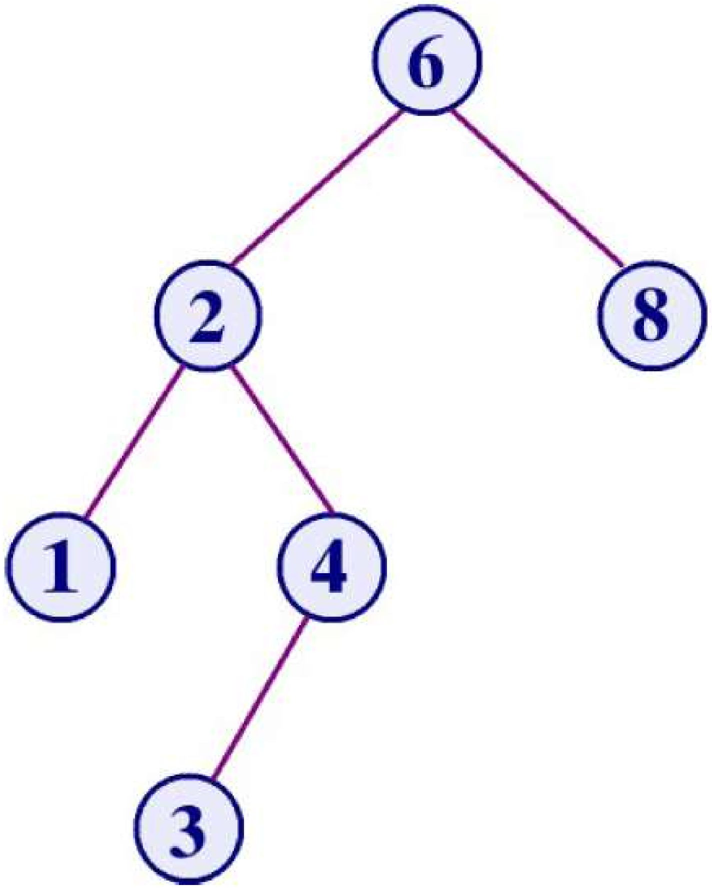
Algorithms complexity
O(nlogn)

```
procedure HSORT (R,n)
for i ← ⌊n/2⌋ to 1 by -1 do call ADJUST (i,n)
for i ← n-1 to 1 by -1 do
    T ← Ri+1; Ri+1 ← R1; R1 ← T;
    call ADJUST (1,i)
end HSORT
```

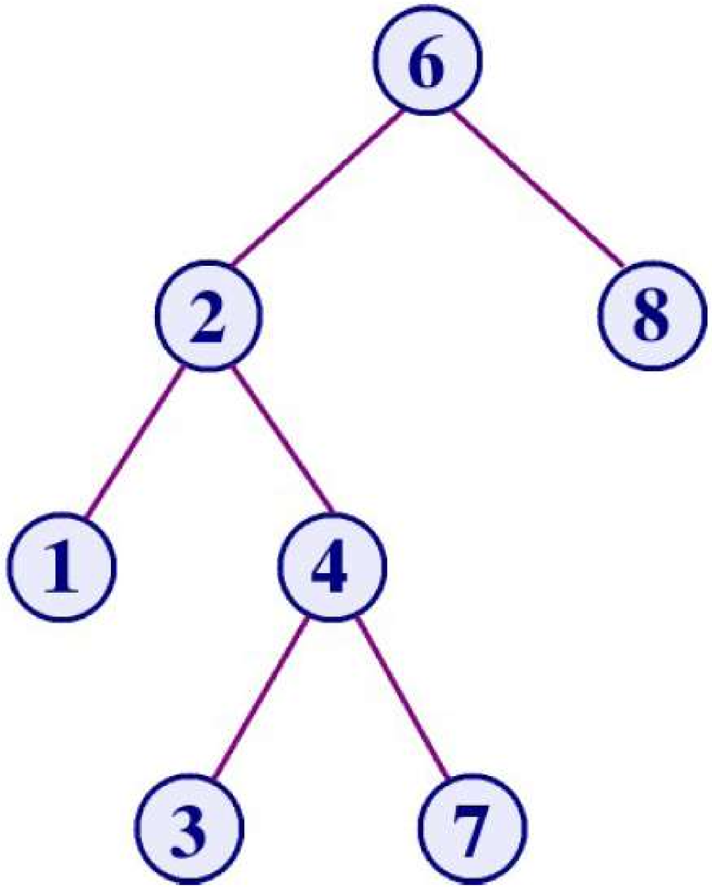
Binary search trees

- Each node in the tree contains information called **key**
- For each node that has a value X for the key, the following rule applies:
  - all nodes of its left subtree have key values **less** than the X value
  - all nodes of its right subtree have key values **greater** than the X value
  - no duplicate keys

Binary search trees



Binary search tree



NOT a binary search tree

Binary search trees

- Interesting properties of binary search trees:
  - How to find the node with the smallest key value?
  - How to find the node with the highest key value?
  - What do you get when the tree is traversed in inorder?

Binary search trees

- Definition of a node for a binary search tree:

Binary search trees

- Definition of a node for a binary search tree:

The definition of a node is the same as the definition of a node of any binary tree!

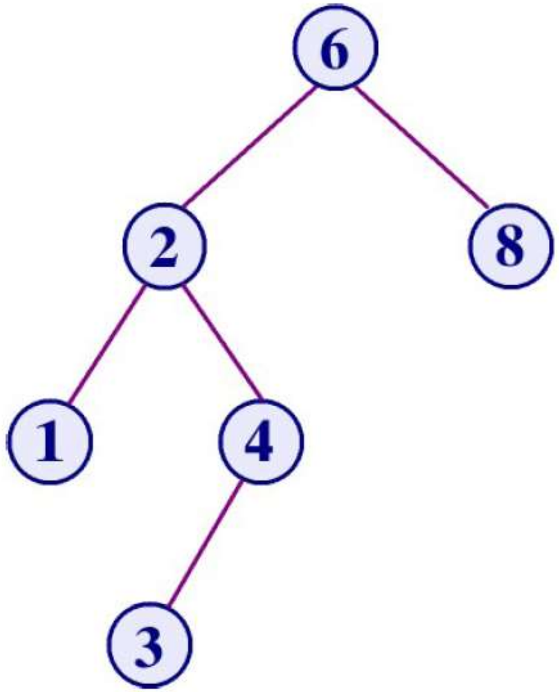
Binary search trees

- Operations with binary search trees:
  - inserting a node into the tree
  - deleting a node from the tree
  - search through the tree

- Visualization : https://visualgo.net/bn/bst

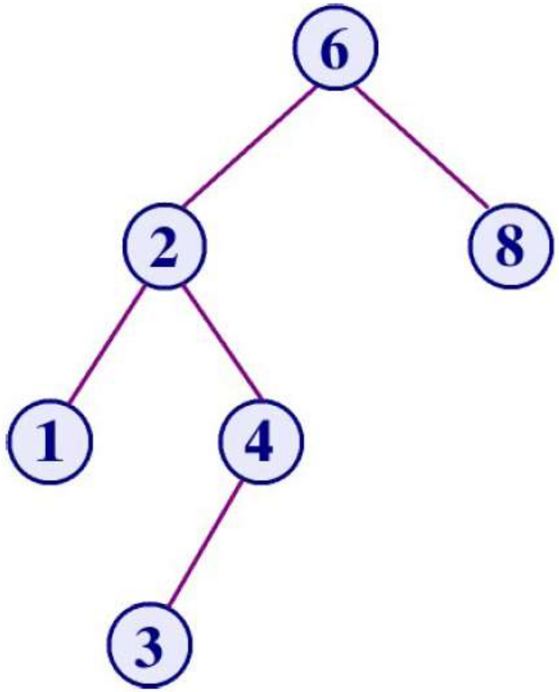
Binary search trees

- Inserting a node:

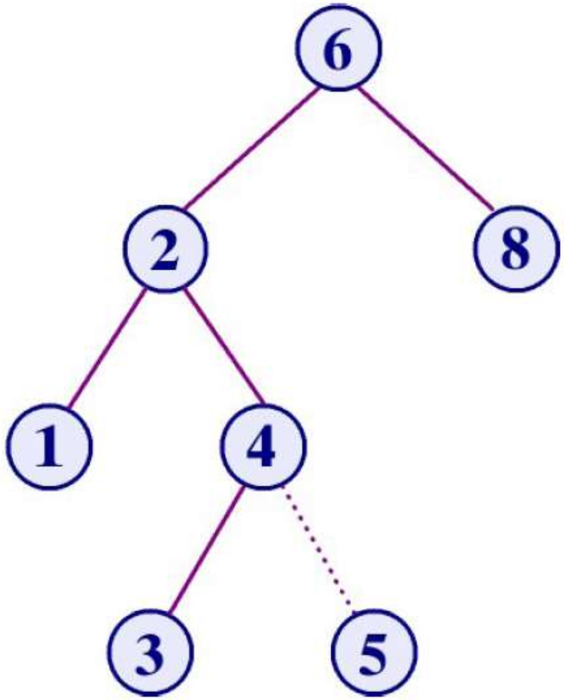


Binary search trees

- Inserting a node:



+ 5 =>



Binary search trees

- Inserting a node:

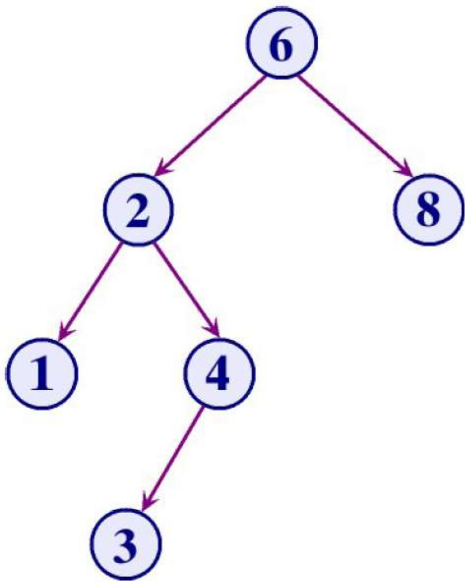
```
TREE-INSERT(T, z)
  y ← NULL
  x ← root[T]
  while x ≠ NULL
  do begin
        y ← x
        if key[z] < key[x] then x ← left[x]
        else x ← right[x]
  end do
  if y = NULL then root[T] ← z
  else if key[z] < key[y] then left[y] ← z
       else right[y] ← z
```

Binary search trees

- Deleting a node:
  - this operation is a bit more complicated
  - deleting a node that is a leaf
  - deleting a node that has one child
  - deleting a node that has two children

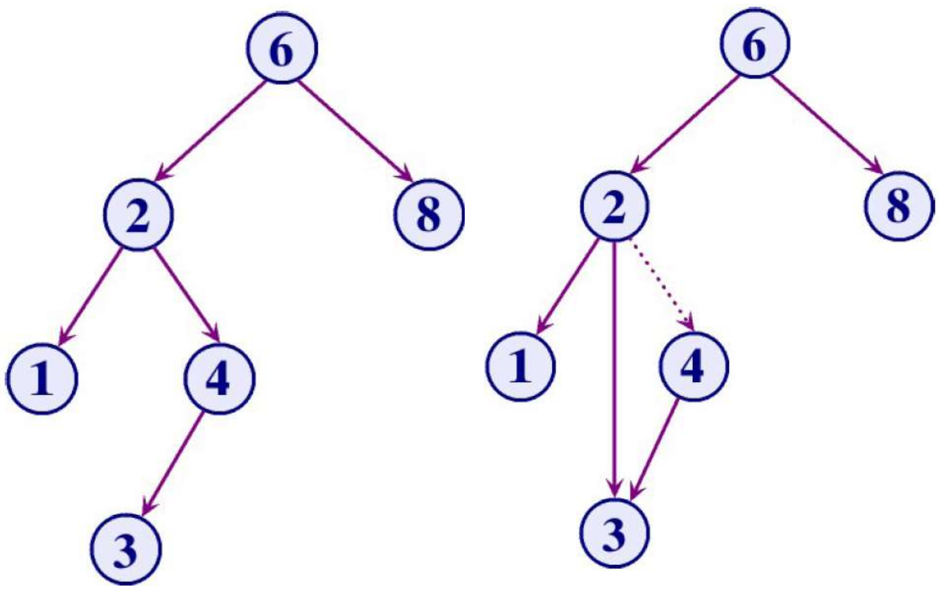
Binary search trees

- Deleting a node:



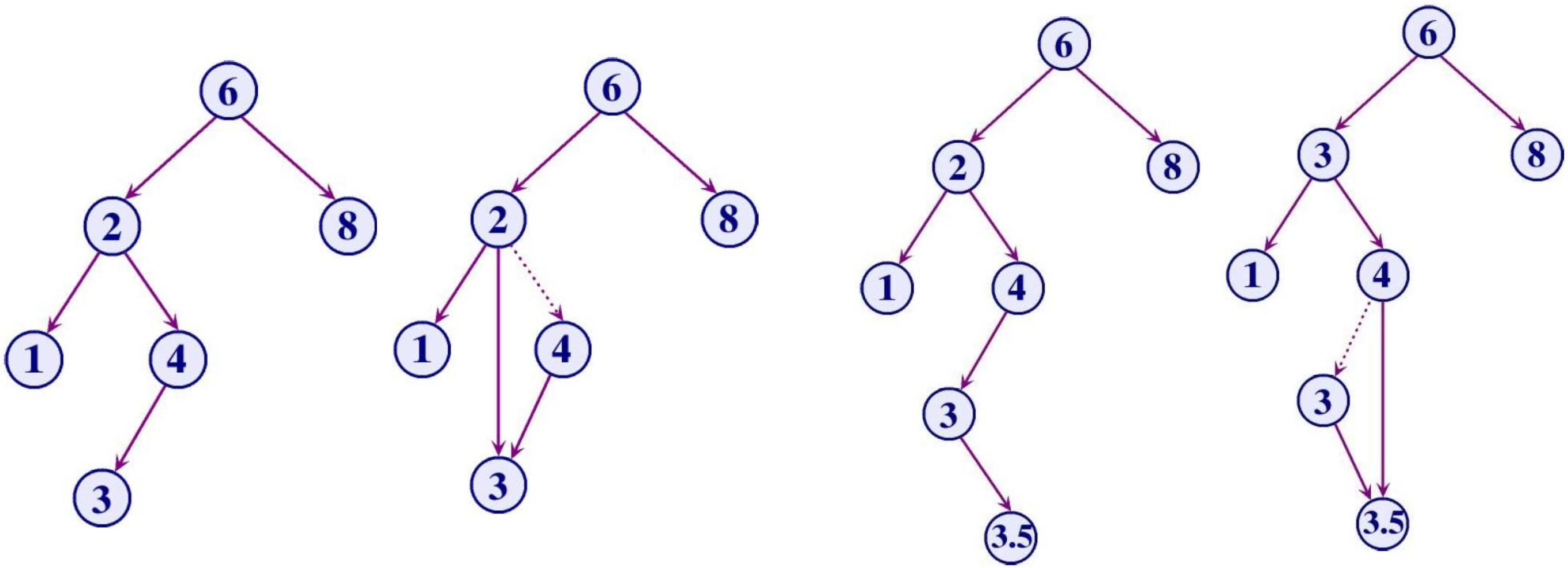
Binary search trees

- Deleting a node:



Binary search trees

- Deleting a node:



Binary search trees

```
delete( element_type x, SEARCH_TREE T )
{
   tree_ptr tmp_cell, child;
   if( T == NULL )
      error("Elementot ne e pronajden");
   else
   if( x < T->element ) /* Odi levo */
      T->left = delete( x, T->left );
   else
      if( x > T->element ) /* Odi desno */
         T->right = delete( x, T->right );
      else /* Najdeniot element da se izbrise */
      if( T->left && T->right ) /* Dve deca */
      { /* Zameni so najmaliot od desnoto podsteblo */
         tmp_cell = find_min( T->right );
         T->element = tmp_cell->element;
         T->right = delete( T->element, T->right );
      }
      else /* Edno dete */
```

Binary search trees

```
    else /* Edno dete - prodolzenie*/
    {
        tmp_cell = T;
        if( T->left == NULL )      /* Samo desno dete */
            child = T->right;
        if( T->right == NULL )     /* Samo levo dete */
            child = T->left;
        free( tmp_cell );
        return child;
    }
return T;
}
```

Binary search trees

- Search in the tree:

recursive

```
TREE-SEARCH (x, k)
if x = NULL or k = key[x]
then return x
if k < key[x] then return TREE-SEARCH (left[x], k)
else return TREE-SEARCH (right[x], k)
```

Binary search trees

- Search in the tree :

recursive

```
TREE-SEARCH (x, k)
if x = NULL or k = key[x]
then return x
if k < key[x] then return TREE-SEARCH (left[x], k)
else return TREE-SEARCH (right[x], k)
```

nonrecursive

```
ITERATIVE-TREE-SEARCH (x,k)
while x ≠ NULL and k ≠ key[x] do
    if k < key[x] then x ← left[x]
    else x ← right[x]
return x
```

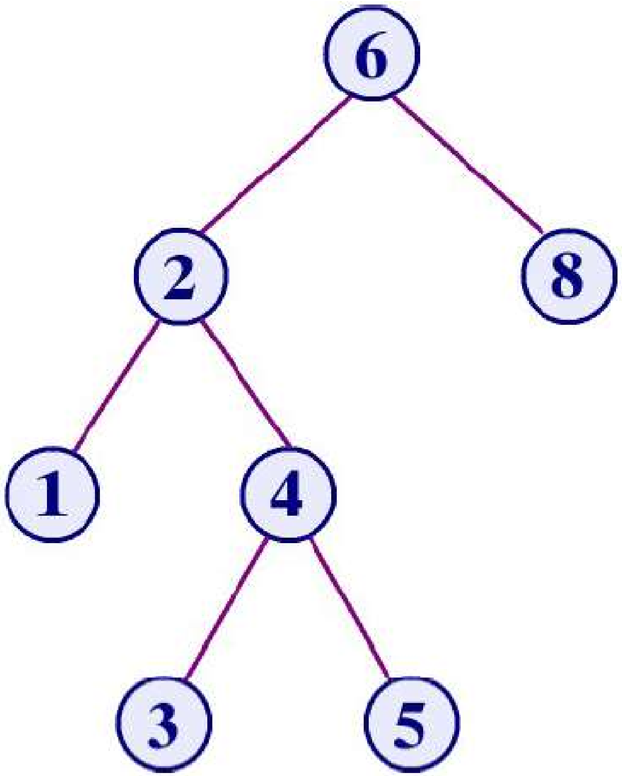
Binary search trees

- The duration of all considered operations in binary search trees depends on the height of the node to be processed
- The level of nodes in an n-element tree can vary significantly in the interval from $\log_2 n$ to n
- If we want better performance, the tree should be **balanced**

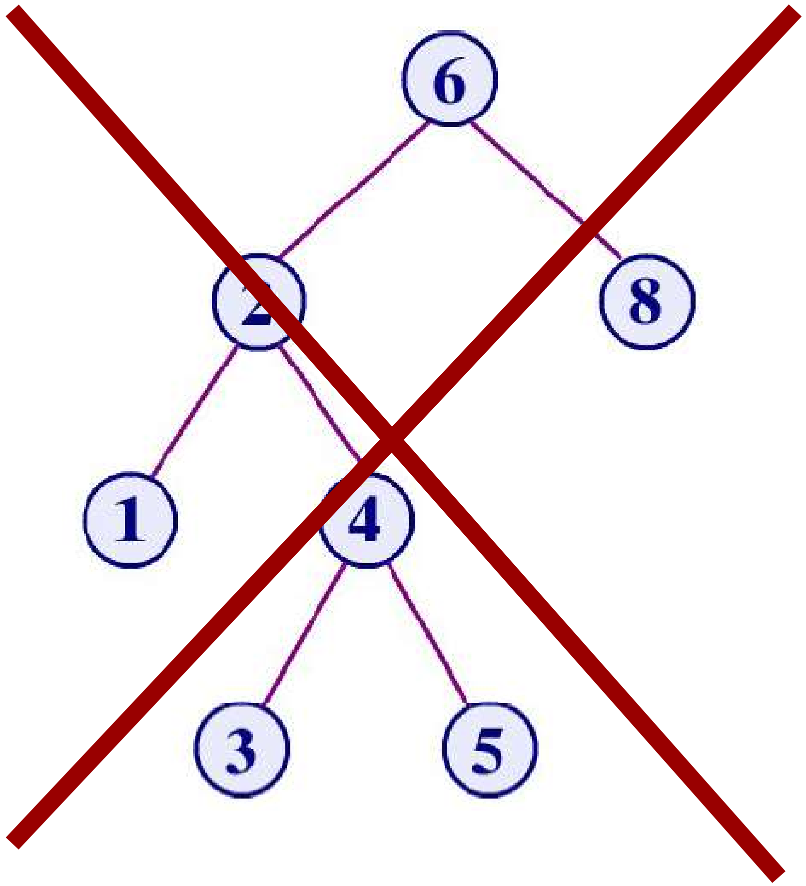
AVL tree

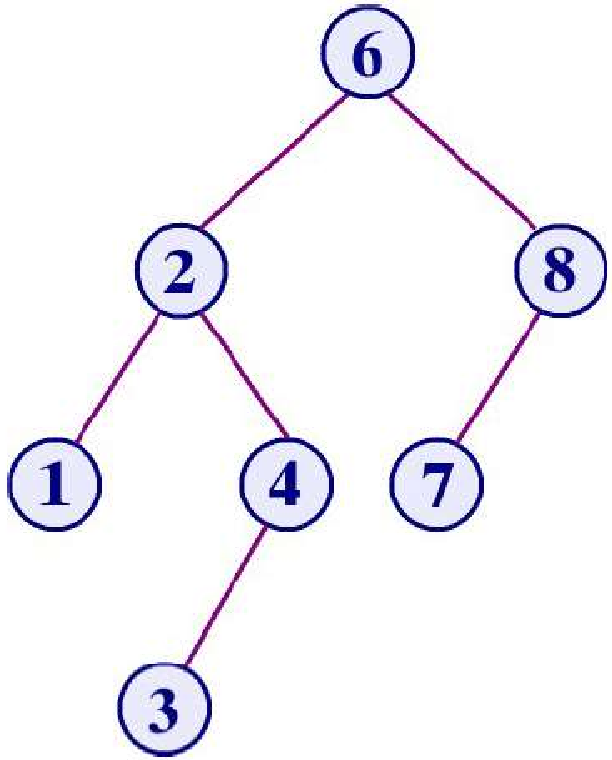
- AVL (Adelson-Velskii & Landis) tree is a binary search tree that is also a balanced tree
- This ensures that the depth of the tree (and thus the complexity of the most common operations) is of order ***O*(log *n*)**

AVL tree



AVL tree





AVL tree

- AVL trees perform better
- The realization of these trees must be done programmatically
- Node insertion and deletion operations will be more complicated than the same operations in binary search trees

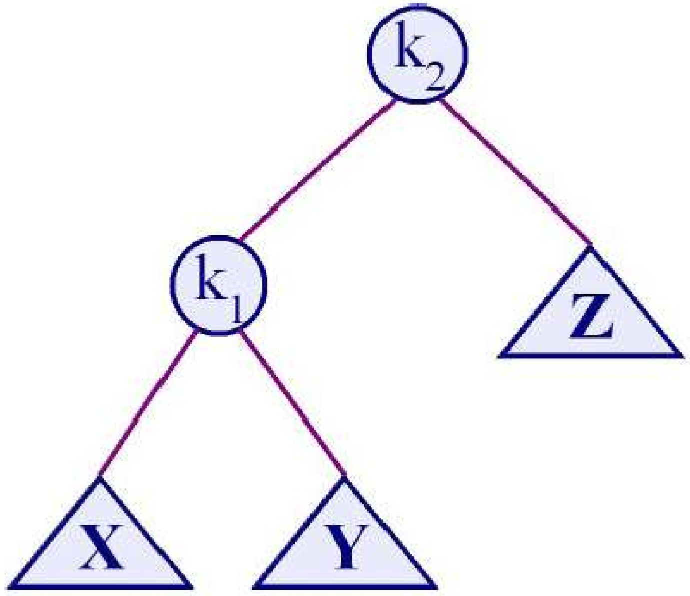
AVL tree

- AVL trees perform better
- The realization of these trees must be done programmatically
- Node insertion and deletion operations will be more complicated than the same operations in binary search trees

Problem: A violation of the balance of the tree!

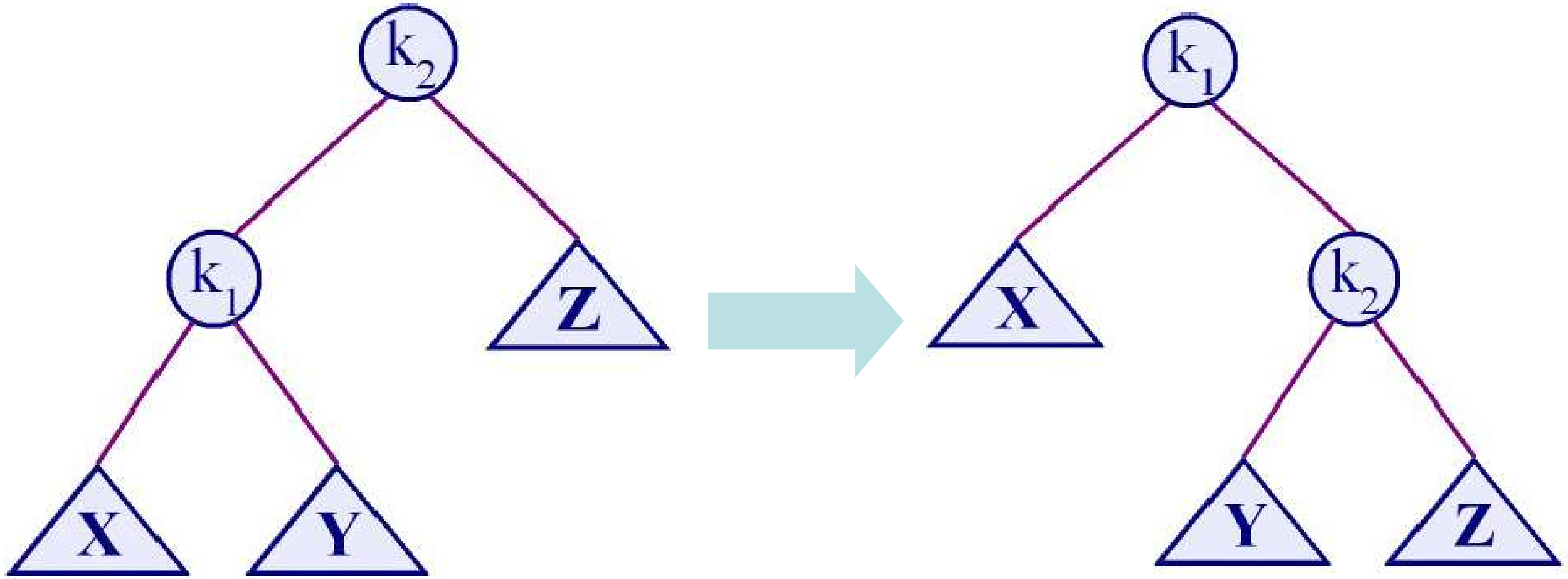
AVL tree

- The balance of an AVL tree is maintained by a single node **rotation** operation

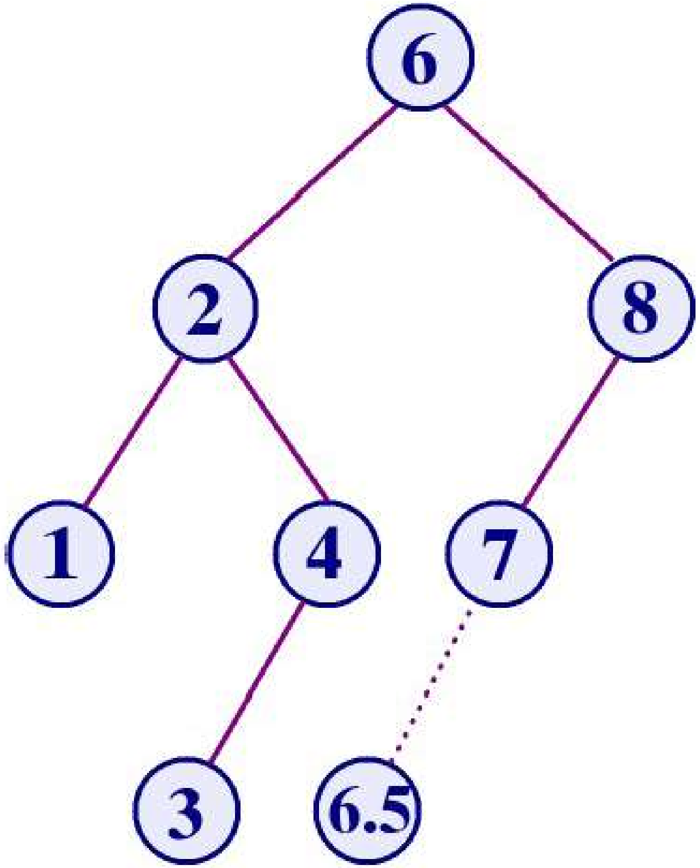


AVL tree

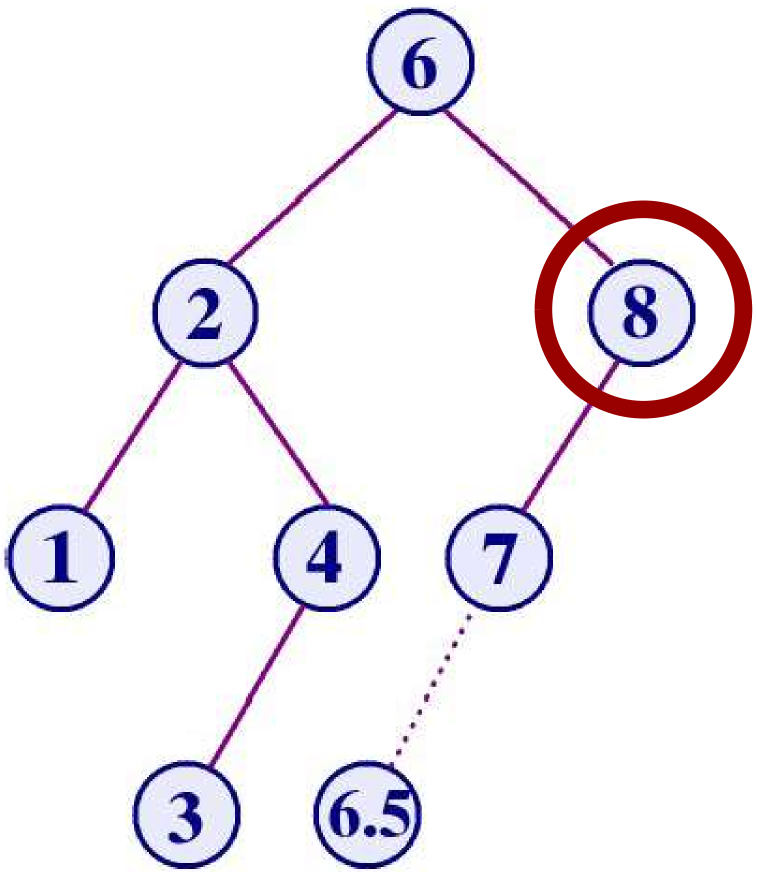
- The balance of an AVL tree is maintained by a single node **rotation** operation

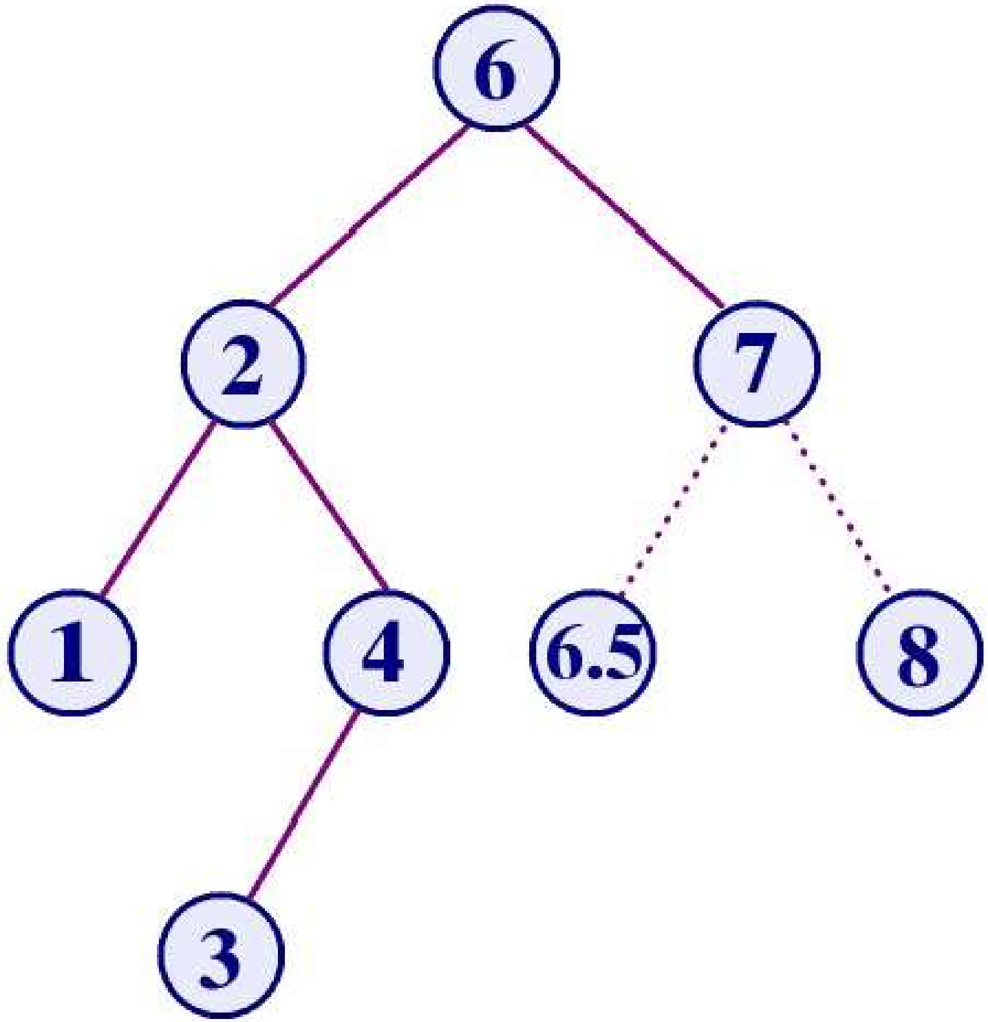


AVL tree



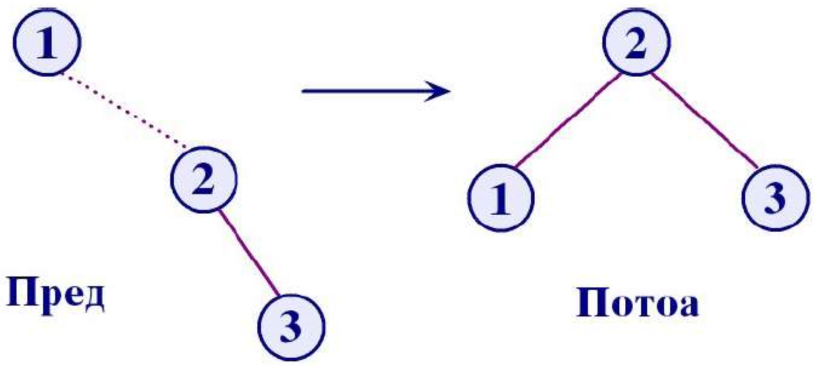
AVL tree





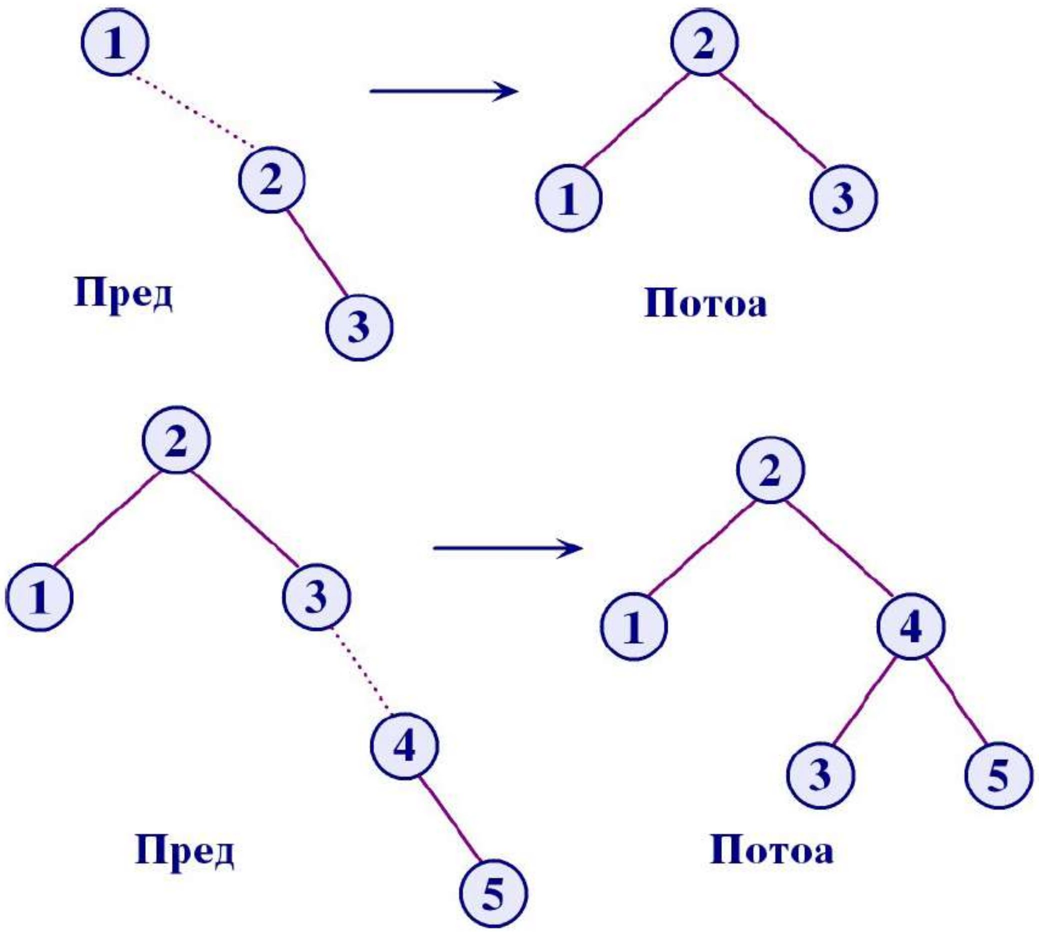
AVL tree

- Example: Building an AVL tree from the values 1, 2, 3, 4, 5, 6 and 7

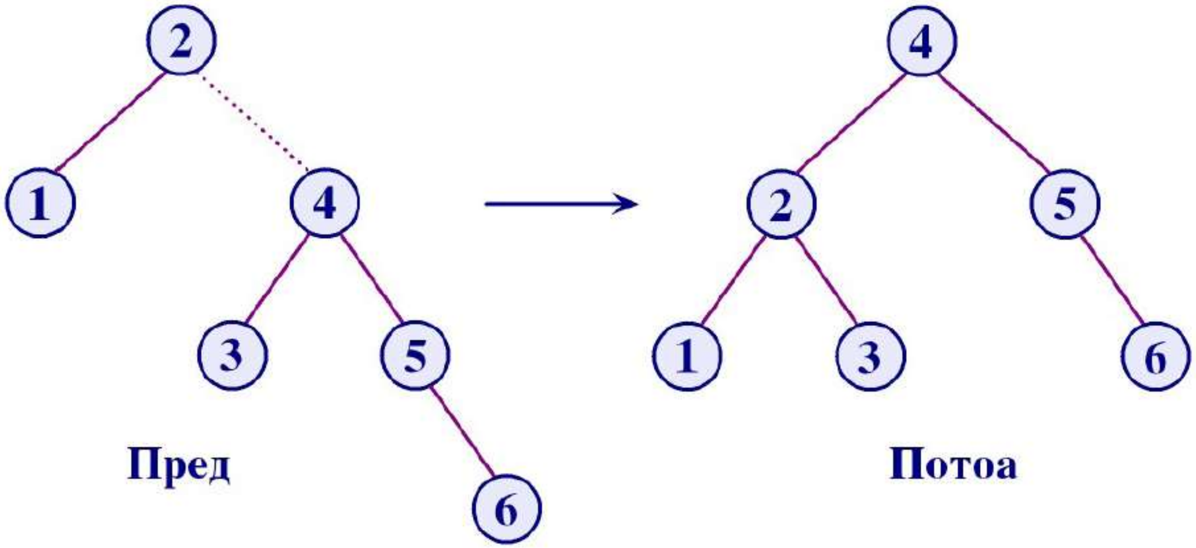


AVL tree

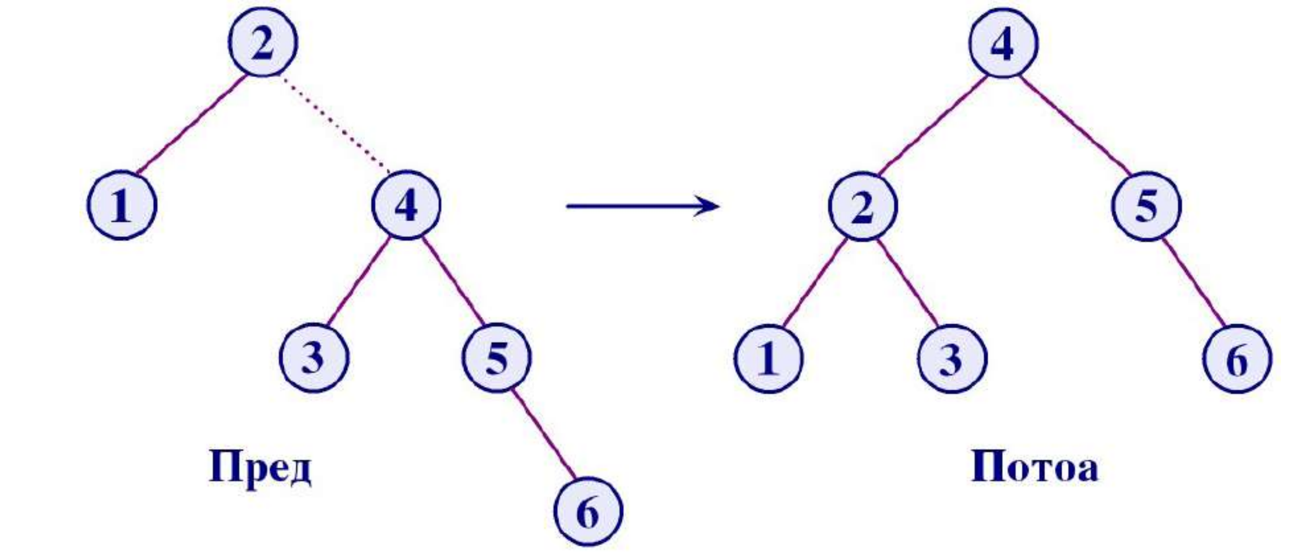
- Example: Building an AVL tree from the values 1, 2, 3, 4, 5, 6 and 7

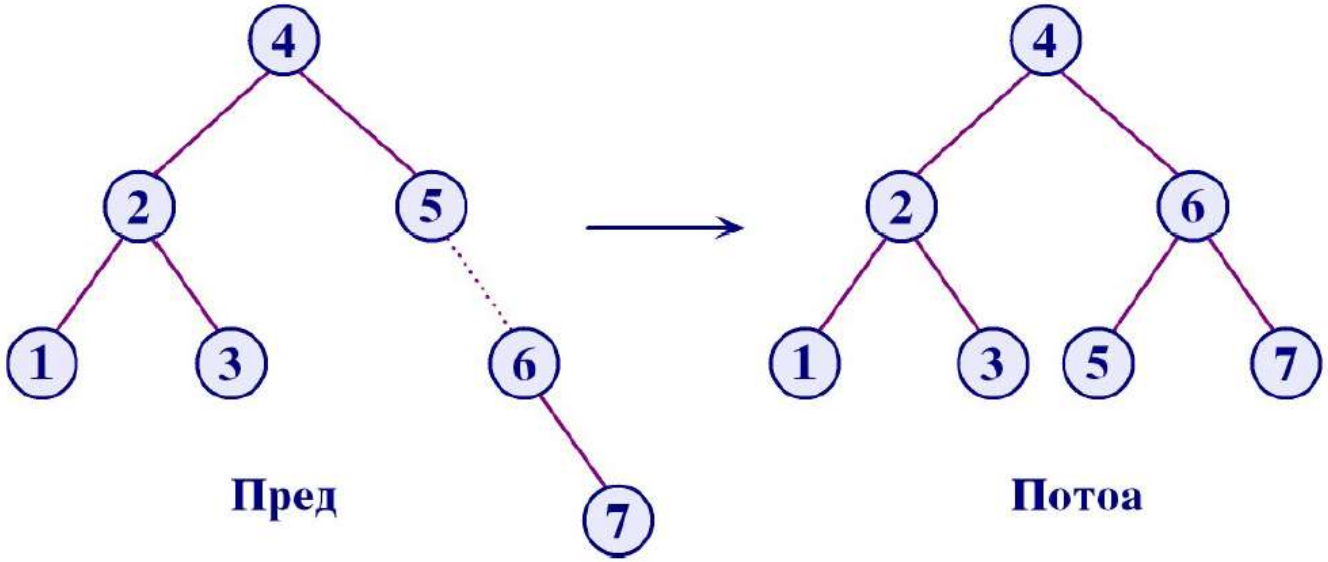


AVL tree

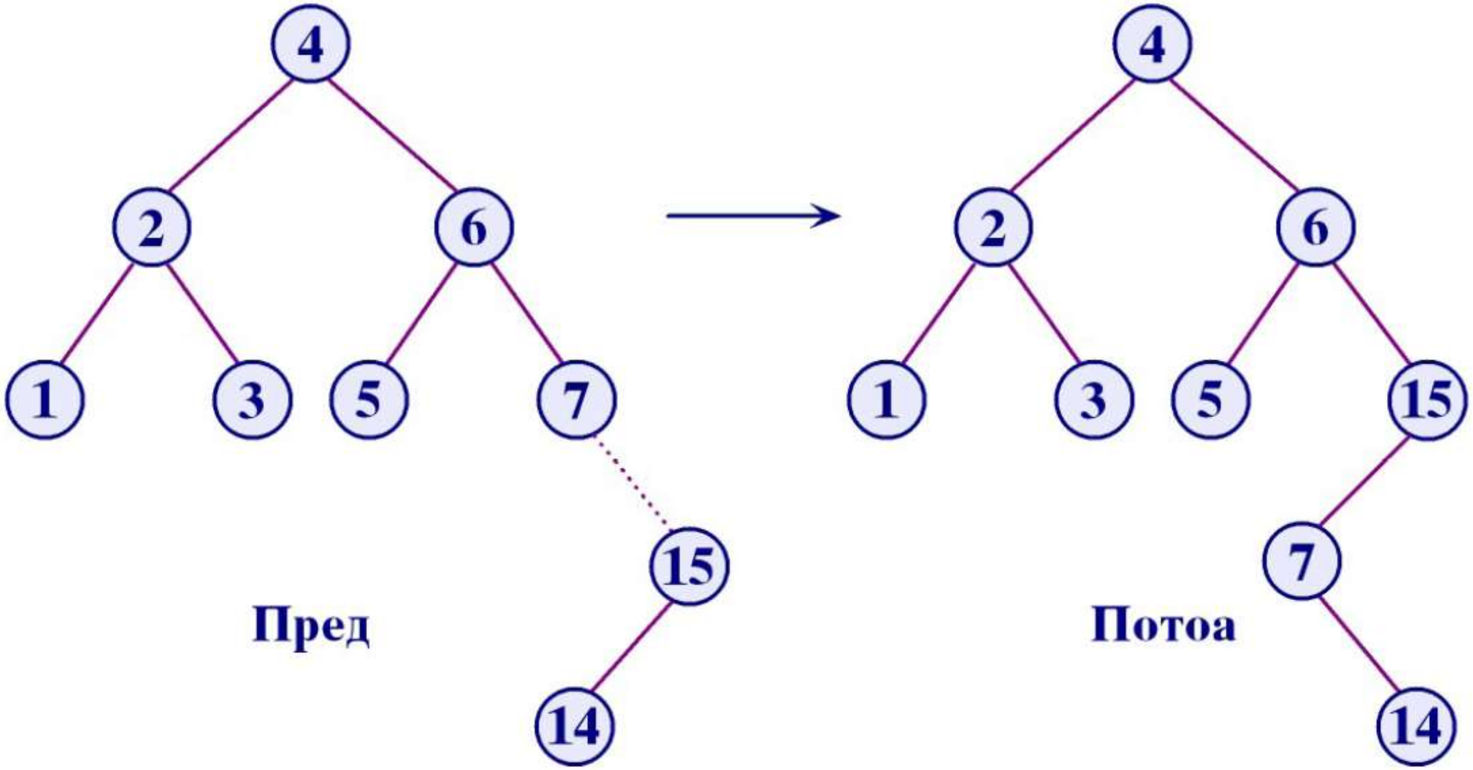


AVL tree

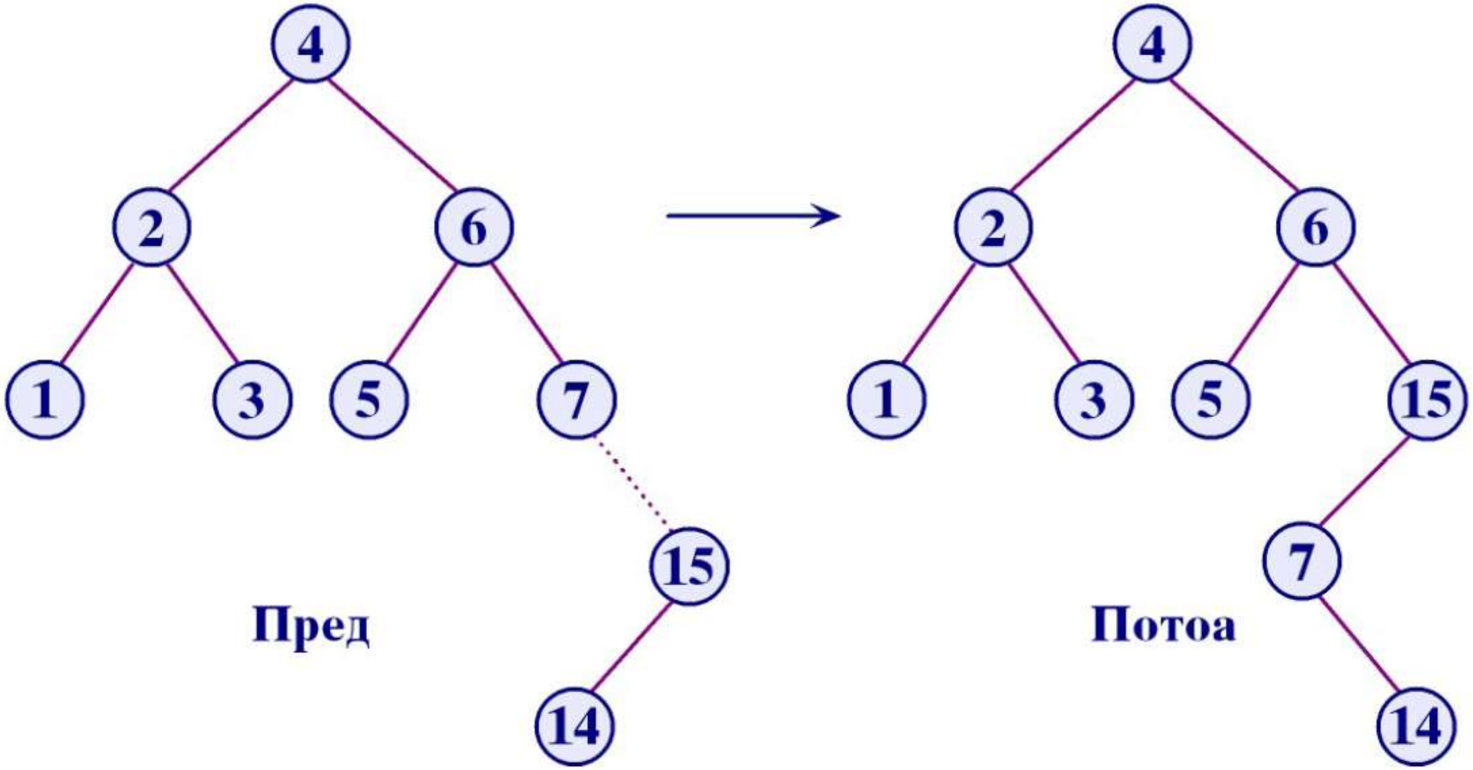




AVL tree



AVL tree



Problem: A single rotation does not help balance the tree in this case!

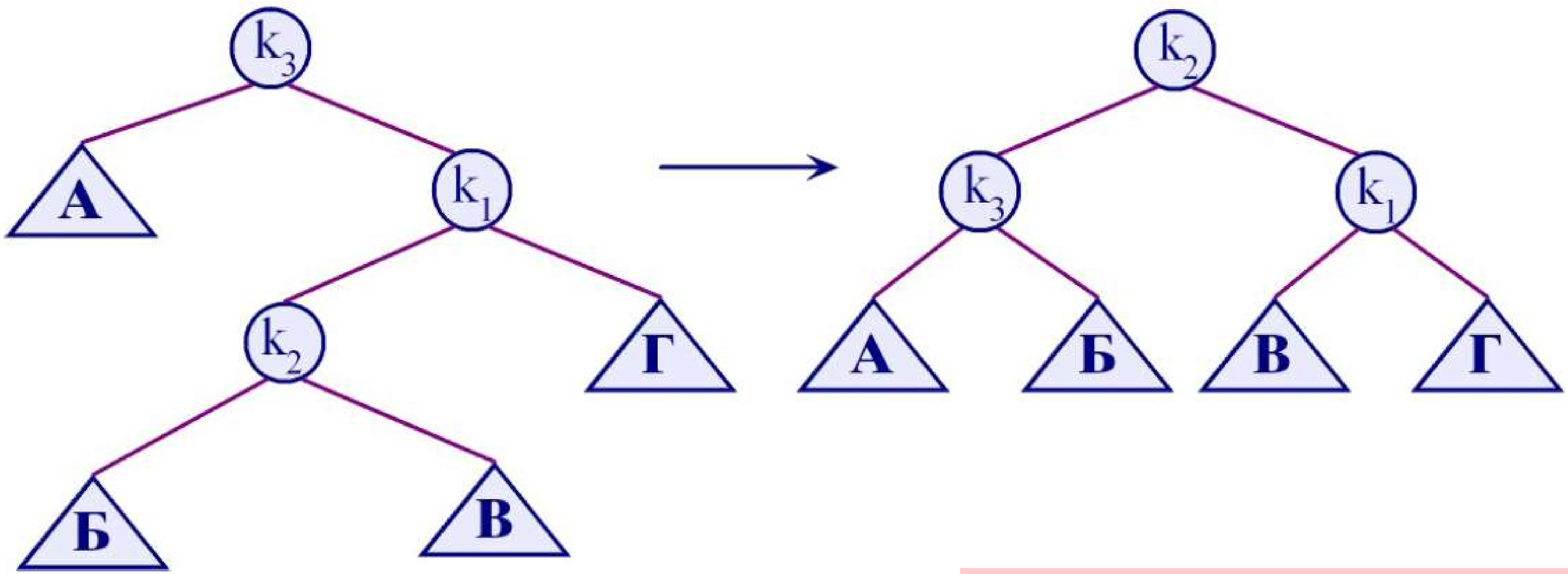
AVL tree

- Entering key 14 causes an imbalance, but applying a single rotation to the left does not resolve it

- This is because the newly entered key is an element (the element that visually goes towards the middle of the subtree) that is:
  - larger than some left subtree
  - smaller than some right subtree

AVL tree

- Double node rotation

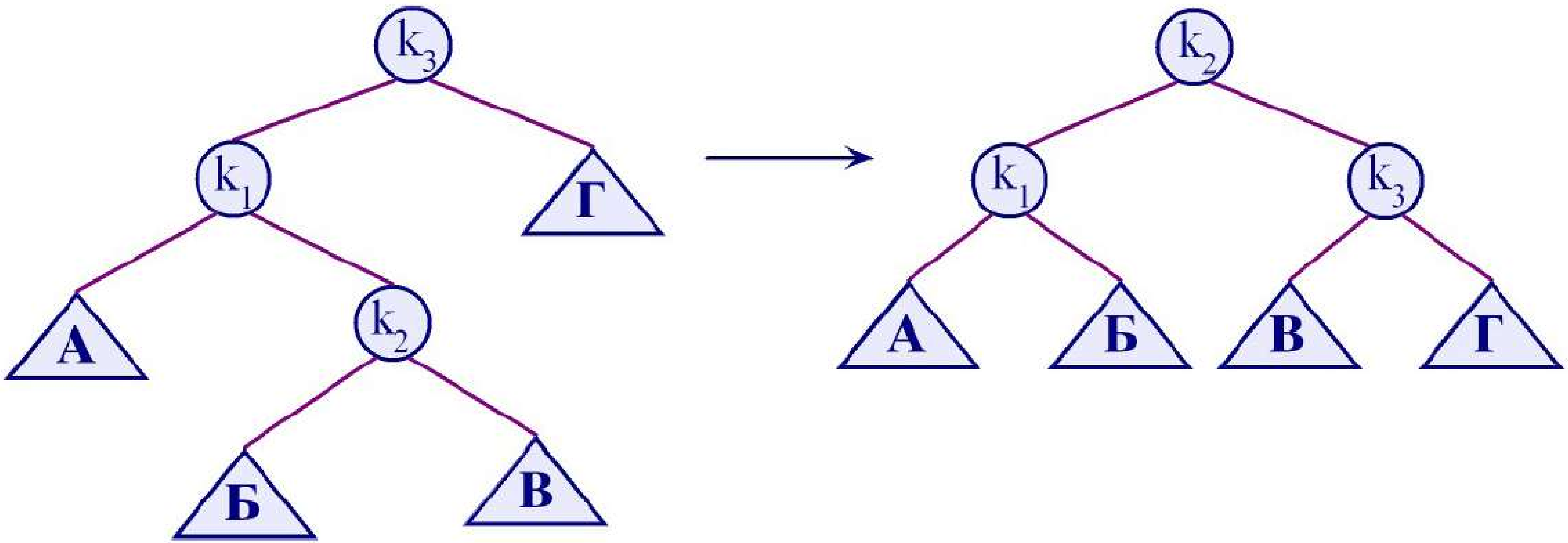


double rotation

right - left

AVL tree

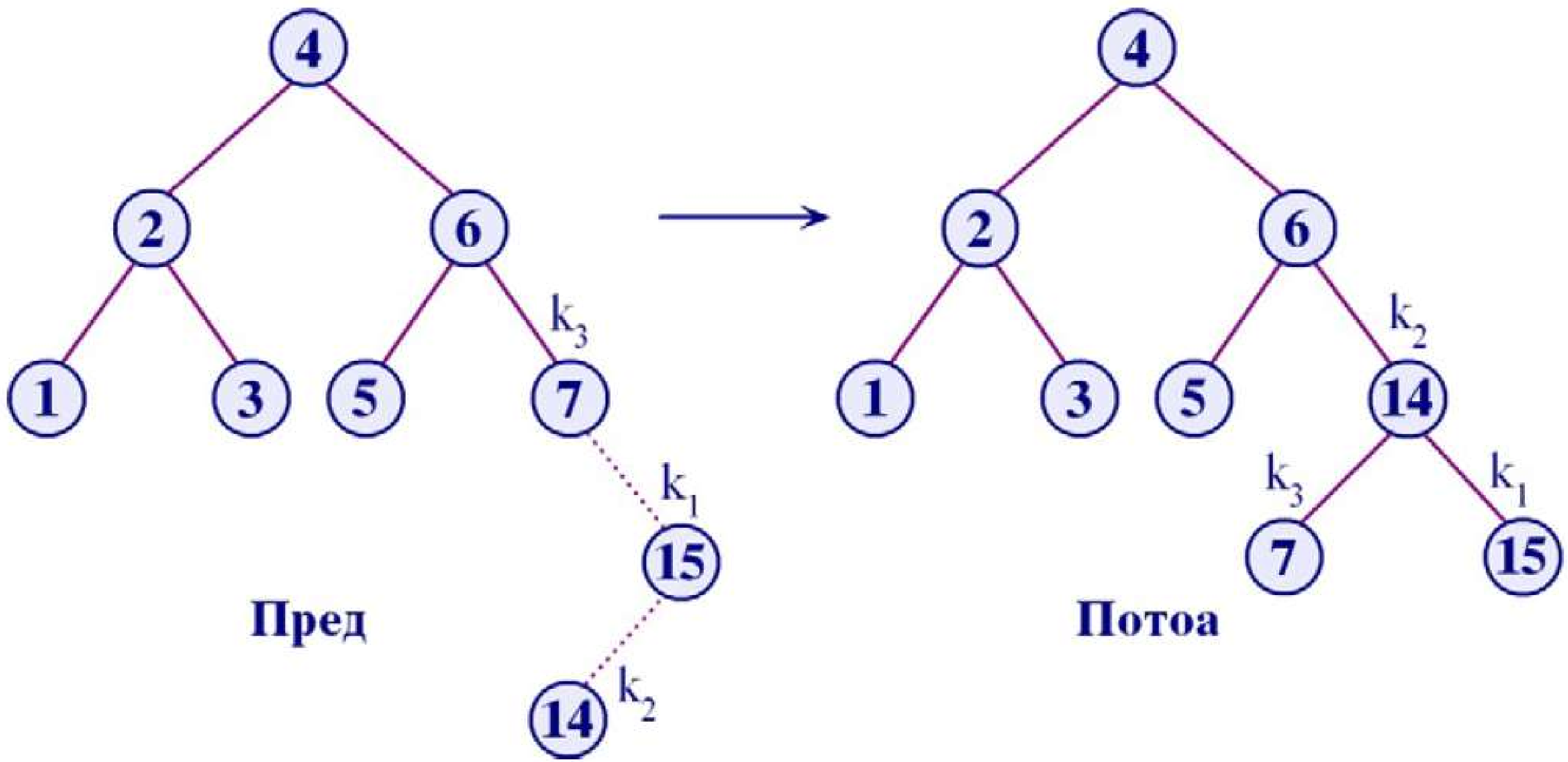
- Double node rotation



double rotation

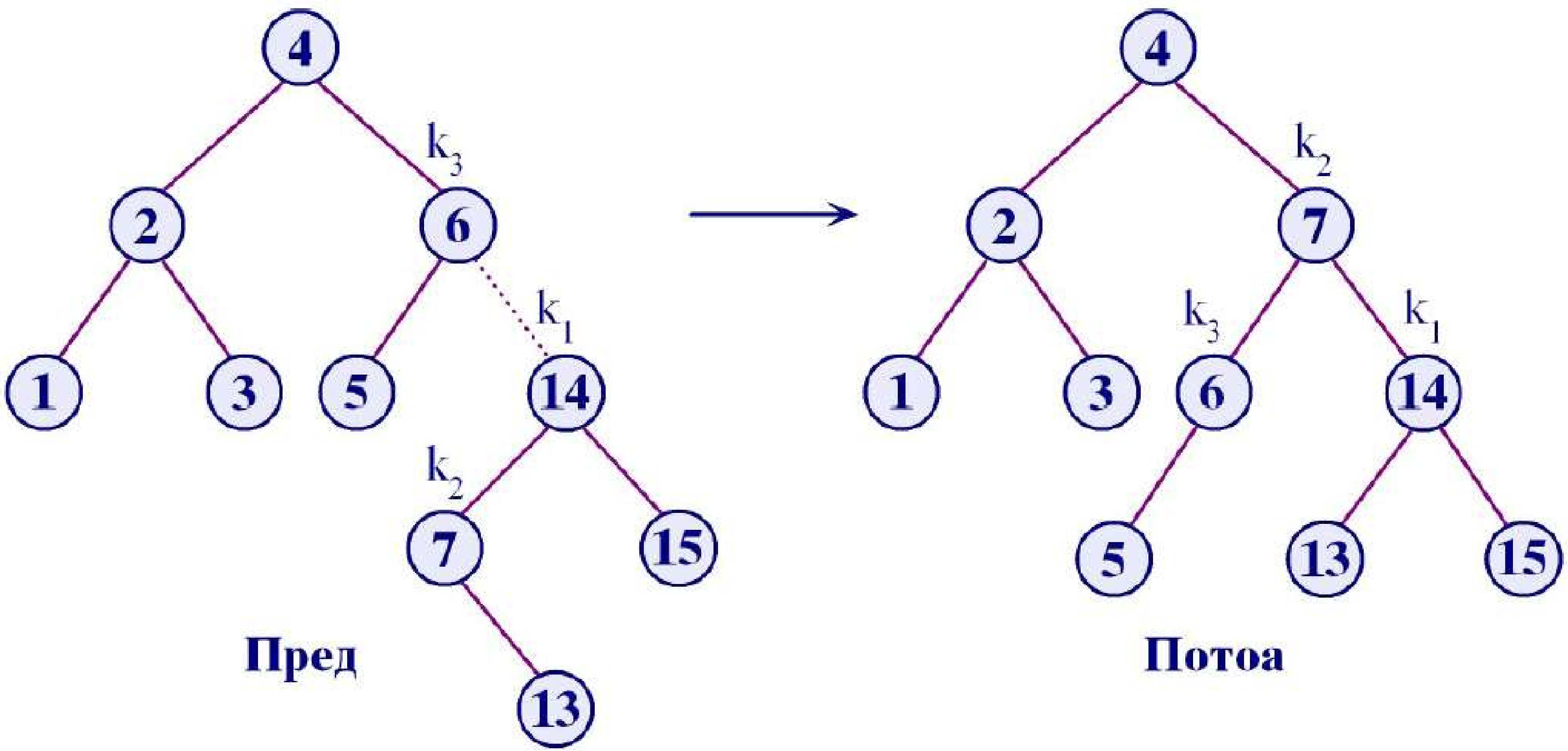
left - right

AVL tree

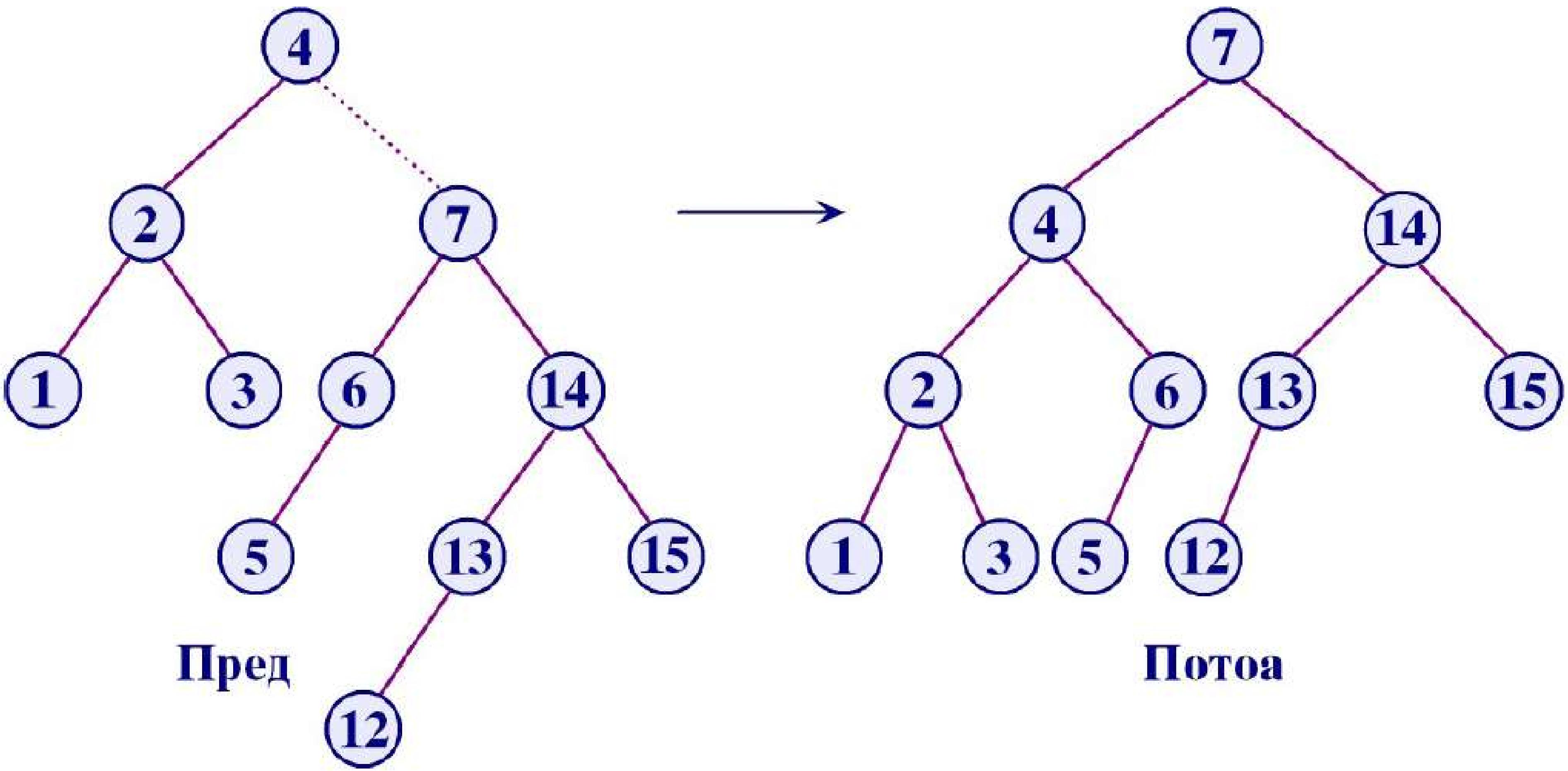


AVL tree

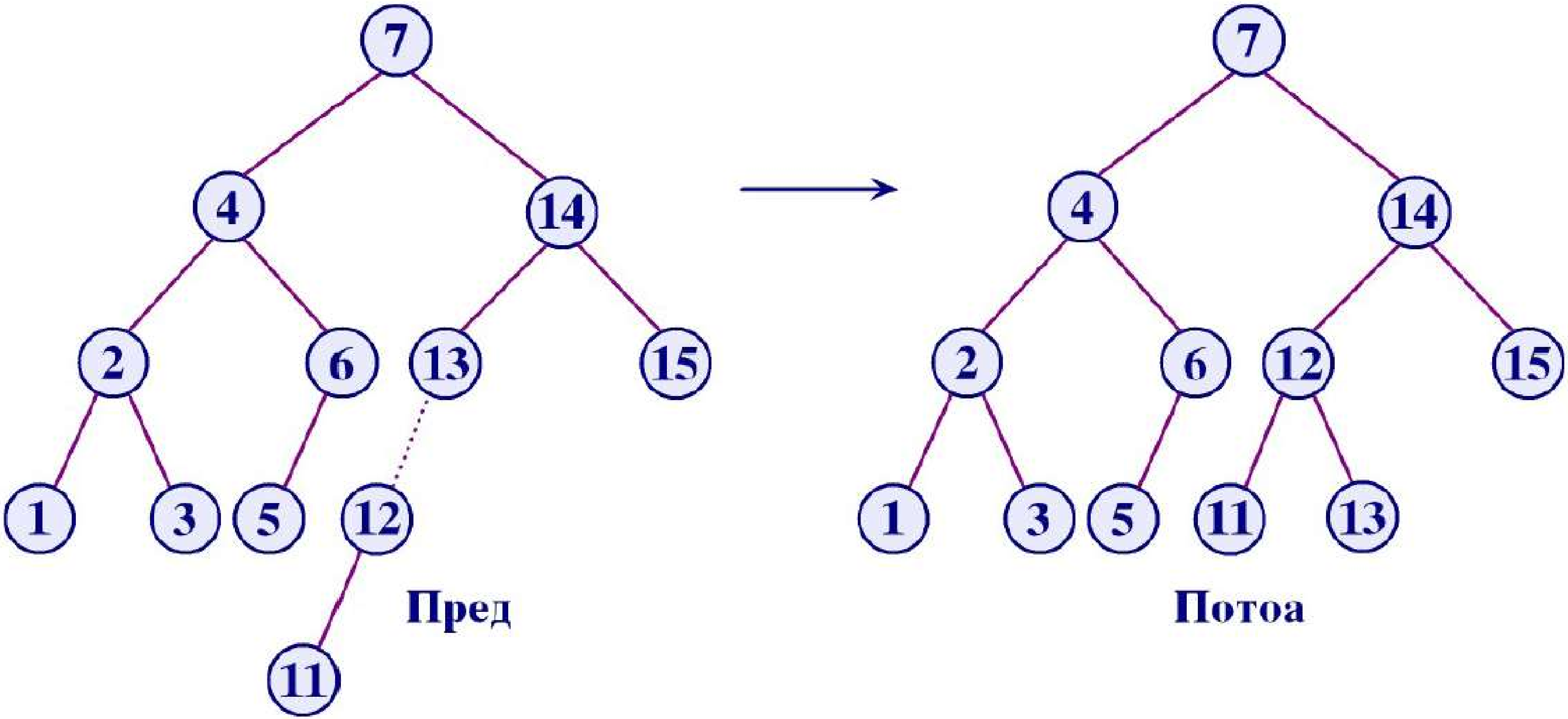
- Let's successively proceed by entering information nodes 13, 12, 11, 10, 9 and 8



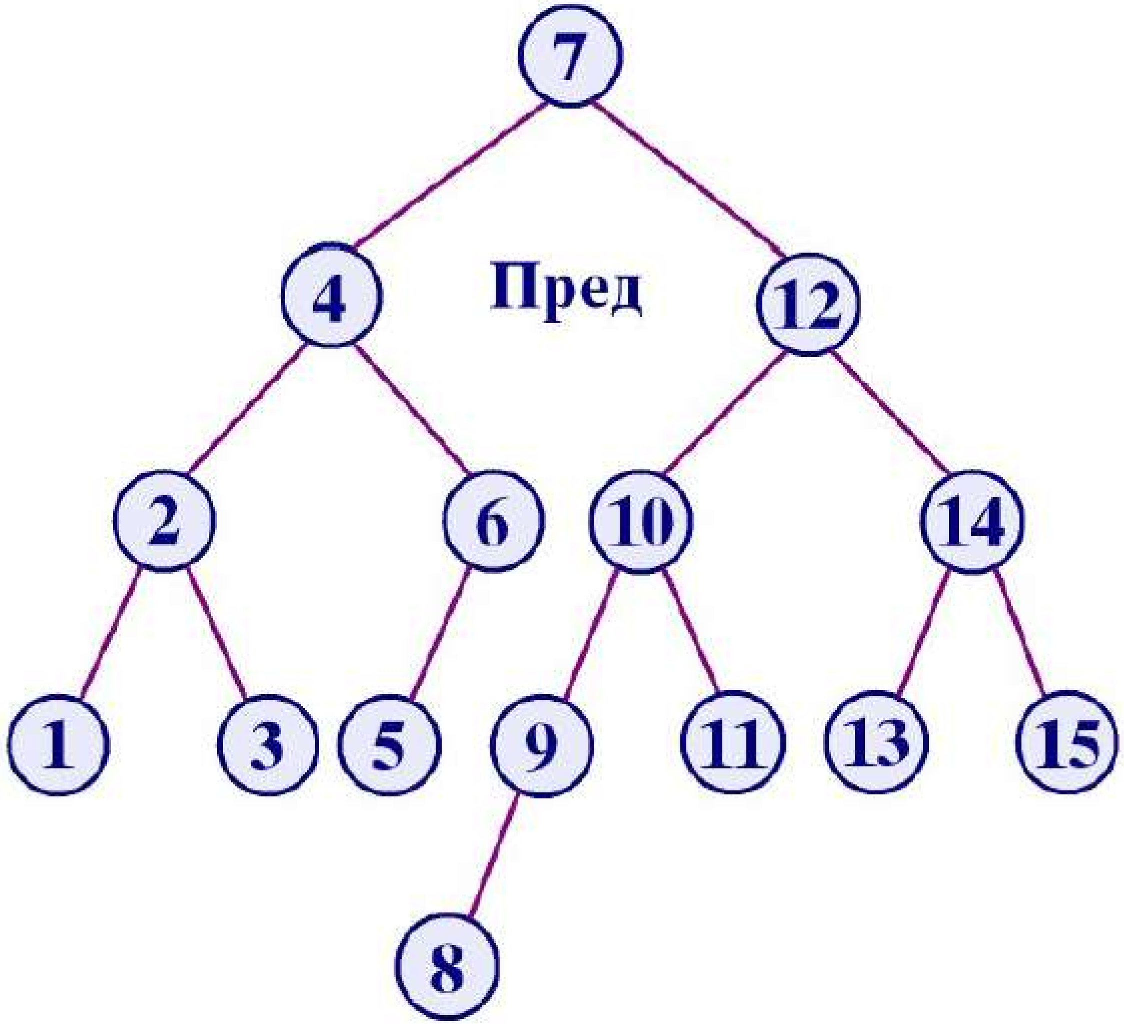
AVL tree



AVL tree



AVL tree



AVL tree

- The node deletion operation is much more complicated than the node insertion operation.
- The most commonly used is the so-called "lazy" deletion, which marks the nodes that are not used, but does not physically remove them.

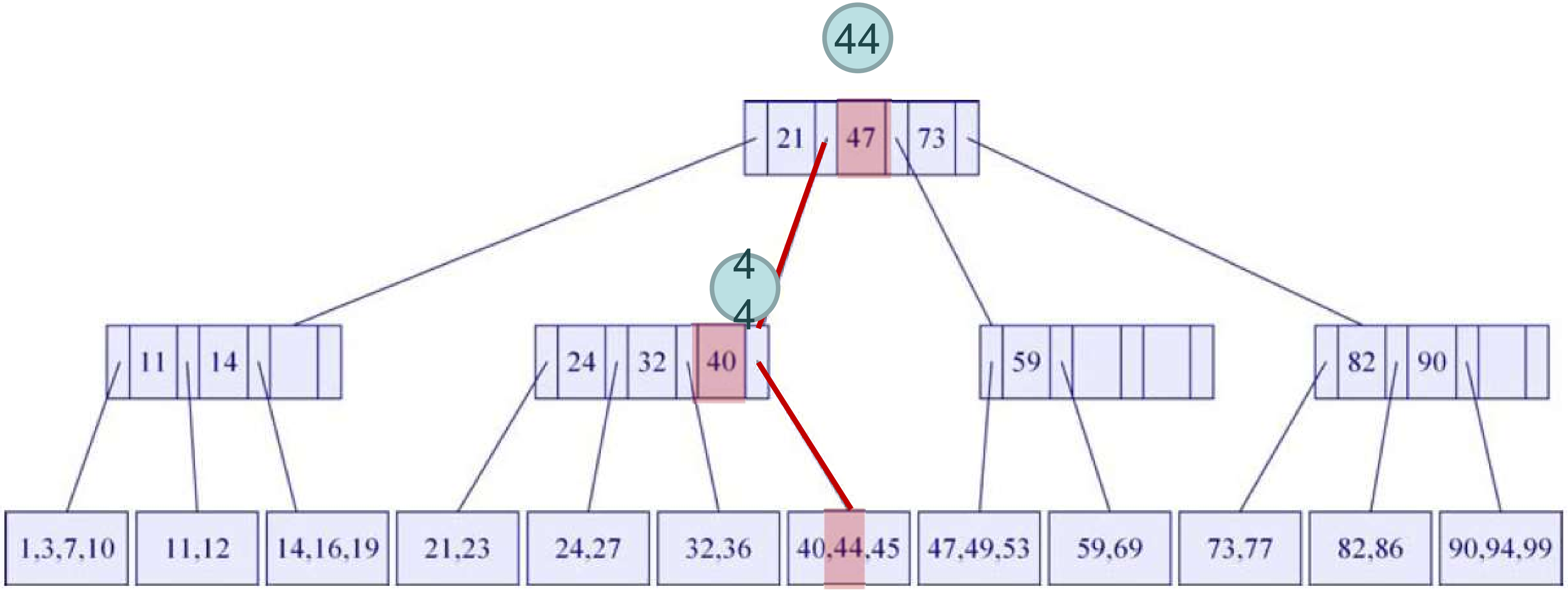
B - trees

- Nonbinary search trees (B, B+, B*, R…)
- B-tree of order *m* is a tree with the following structural properties:
  - The root is either a leaf or has between 2 and *m* children
  - All internal nodes (except the root) have between $\lceil m/2 \rceil$ and *m* children
  - All leaves are at the same level

B - trees

- Characteristic operations in B-trees:
  - search
  - inserting a value
  - deleting a value

Example: B – tree of order 4



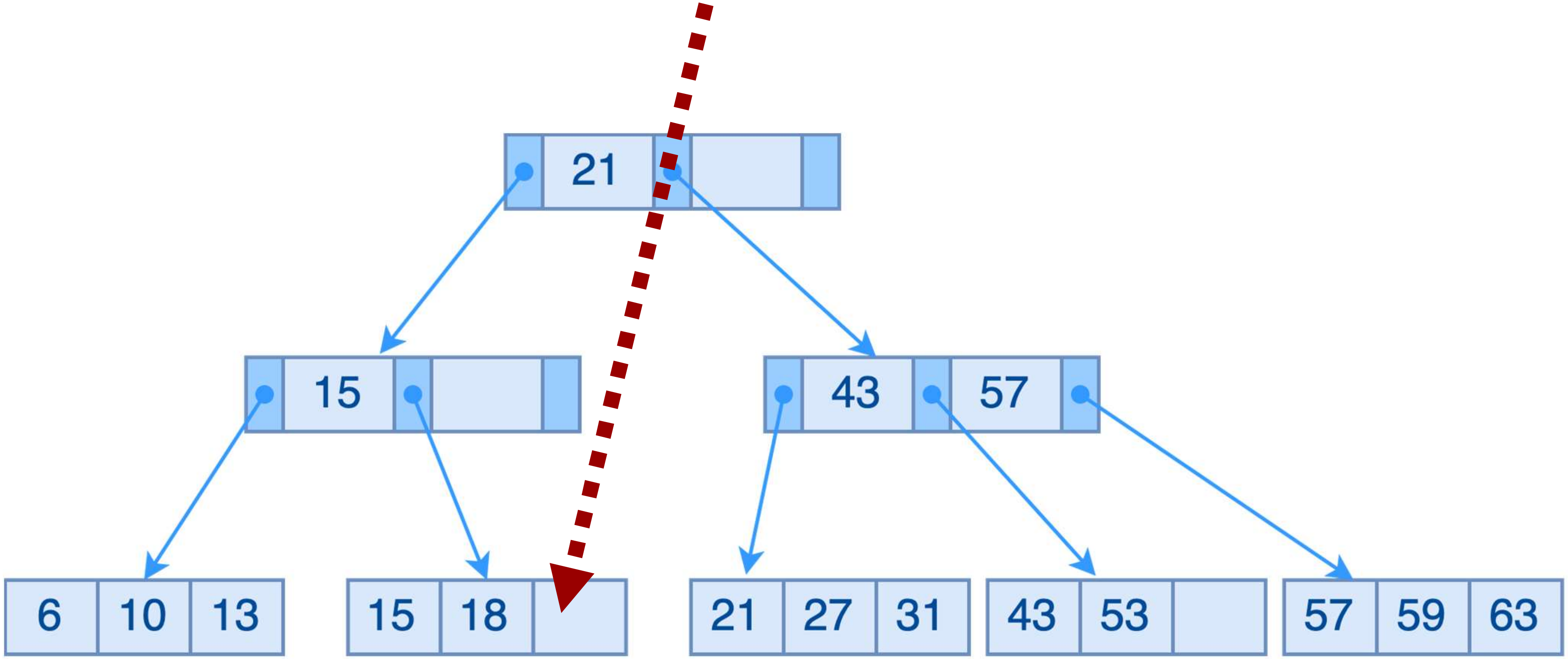
Search: 44

B - trees

- Inserting a value in a B-tree:
  - Finding the node where the value should be inserted
  - Splitting the node
    - if the principles of the B-tree are not preserved
    - the node has more values than allowed - in this case the node **splitting** process is applied
      - it can also cause splitting of nodes in higher levels as well

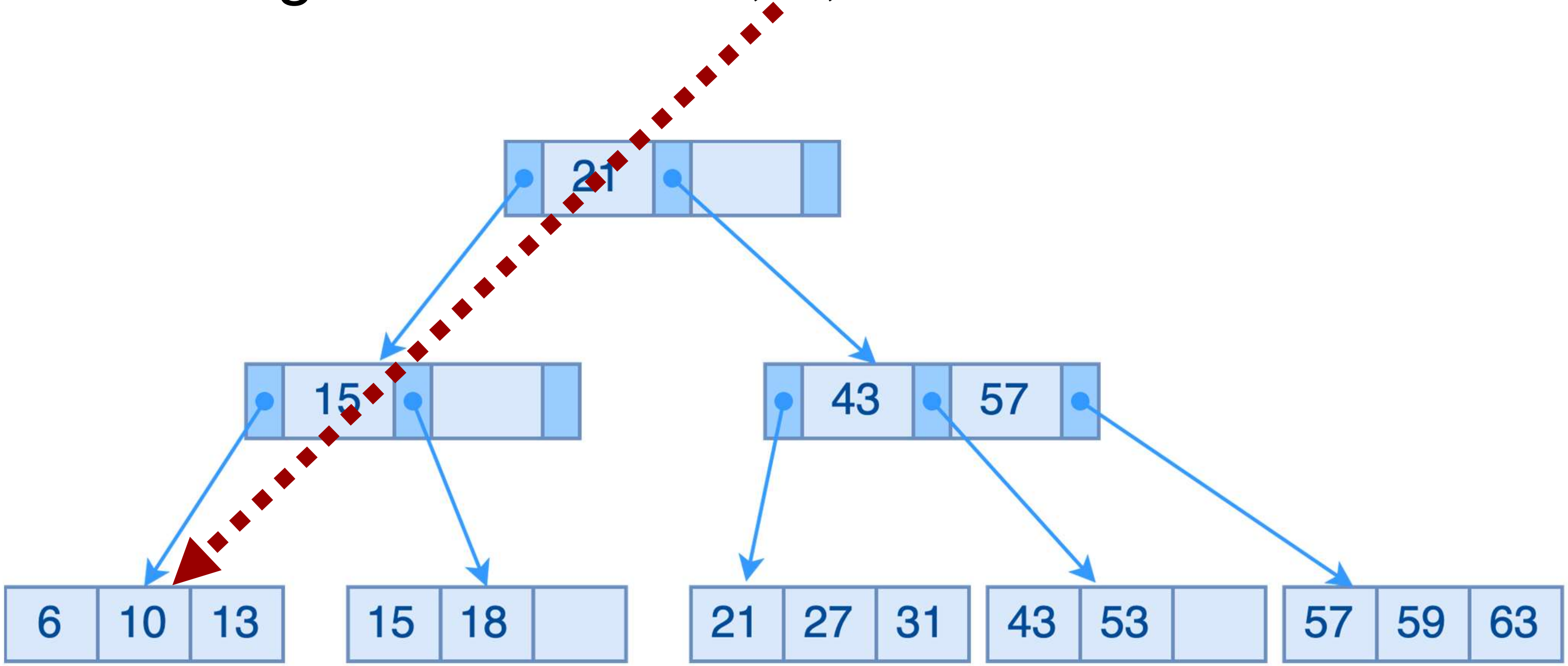
Example: B - trees

- Inserting the values: 19, 2, 20 and 29



Example: B - trees

- Inserting the values: 19, 2, 20 and 29

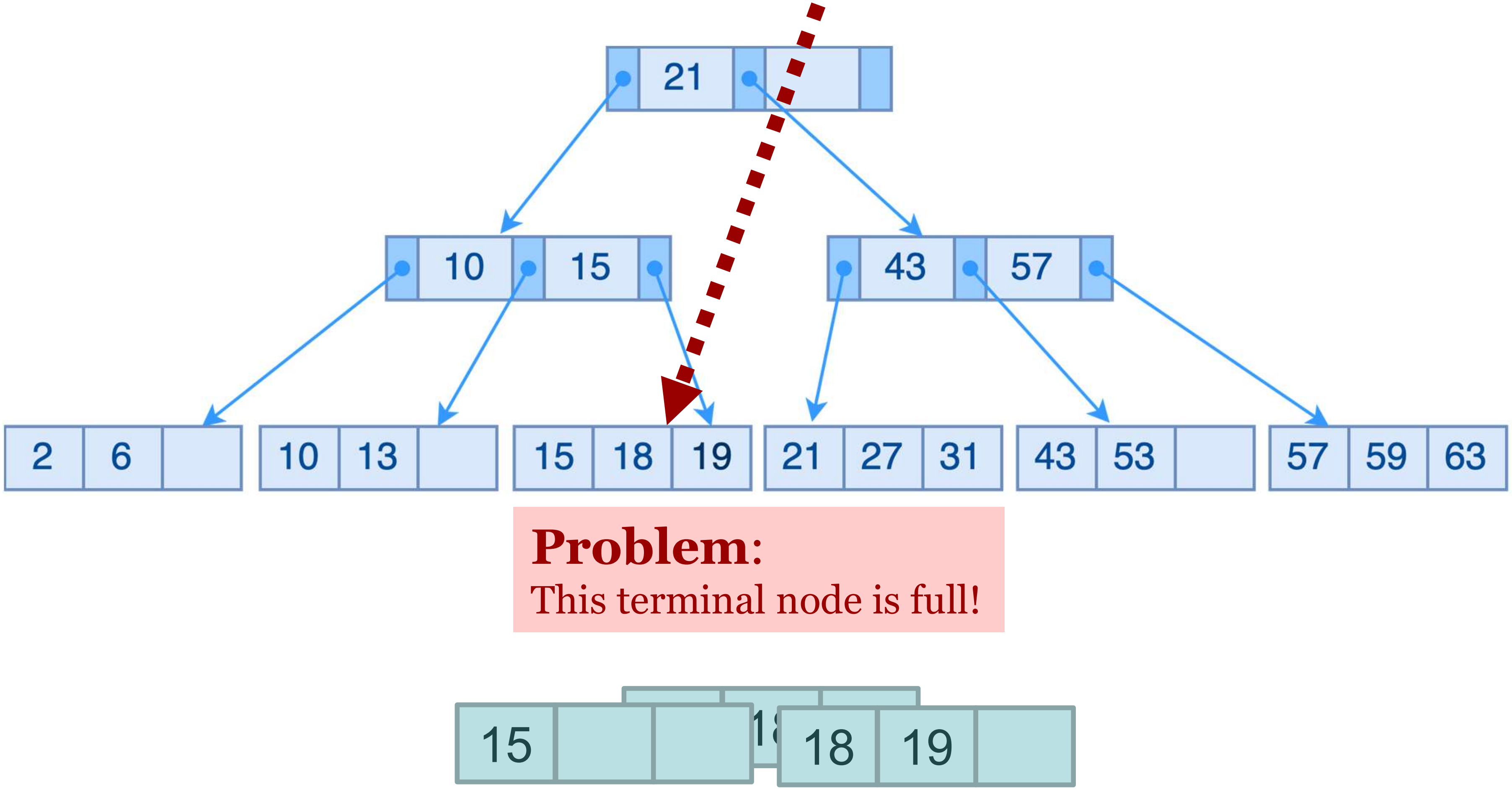


Problem:
This terminal node is full!

Solution:
Node splitting!

Example: B - trees

- Inserting the values: 19, 2, 20 and 29

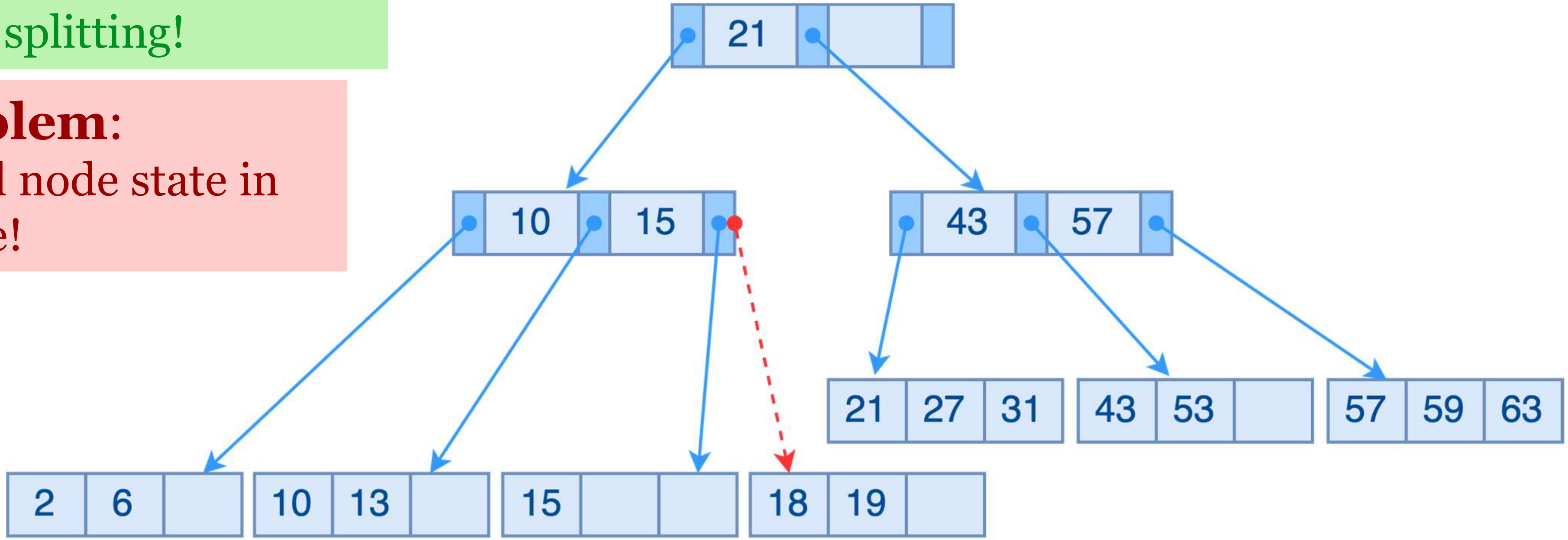


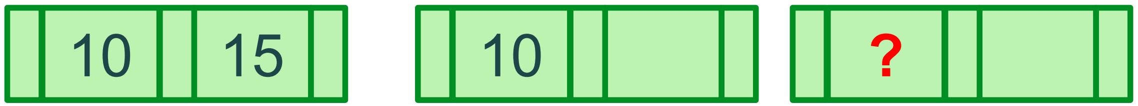
Example: B - trees

- Inserting the values: 19, 2, 20 and 29

Solution:
Node splitting!

Problem:
Illegal node state in B-tree!

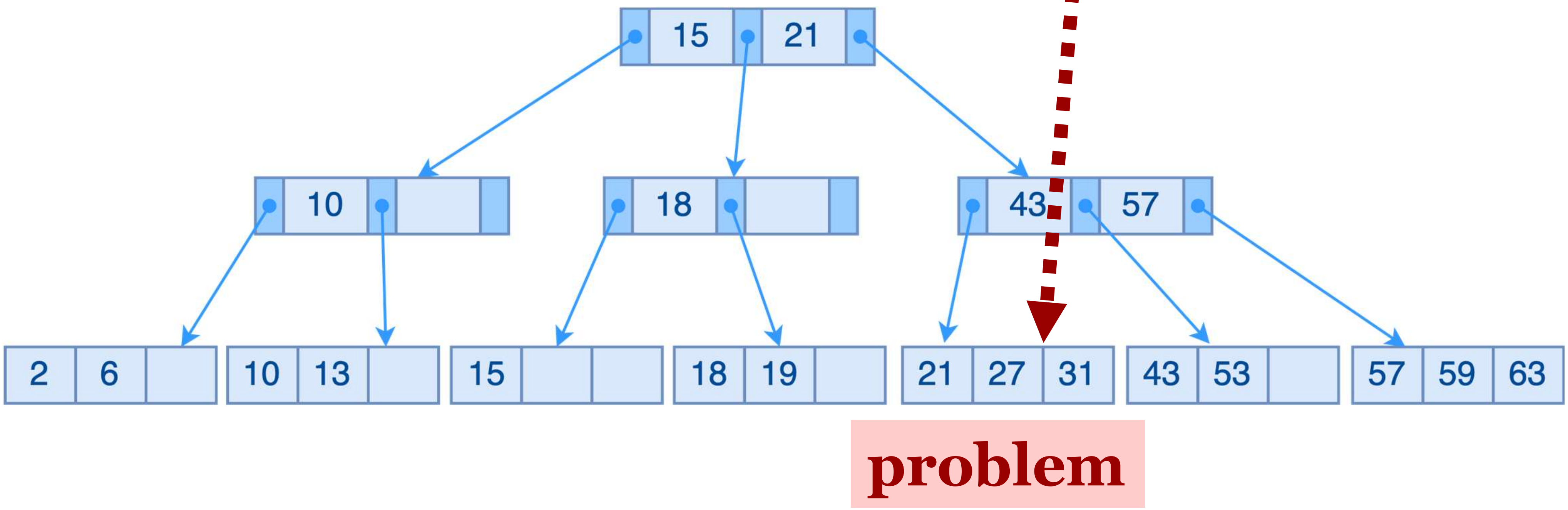




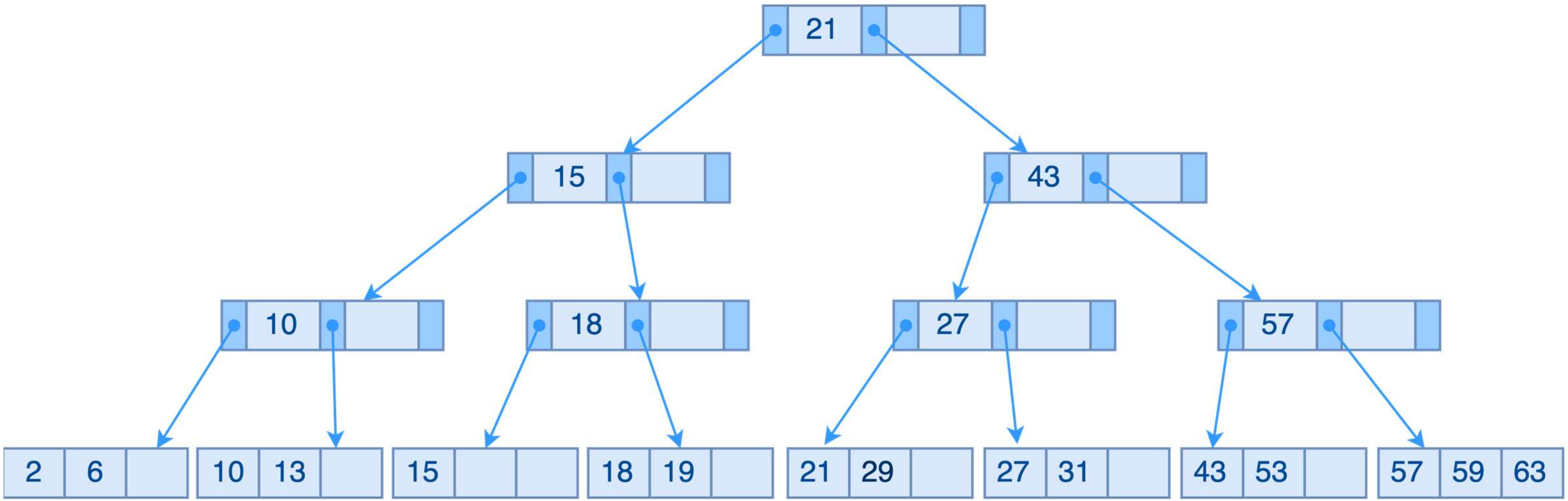
The value of ? is taken to be the leftmost value of the right child, and 15 goes up one level

Example: B - trees

- Inserting the values: 19, 2, 20 and 29



Example: B - trees

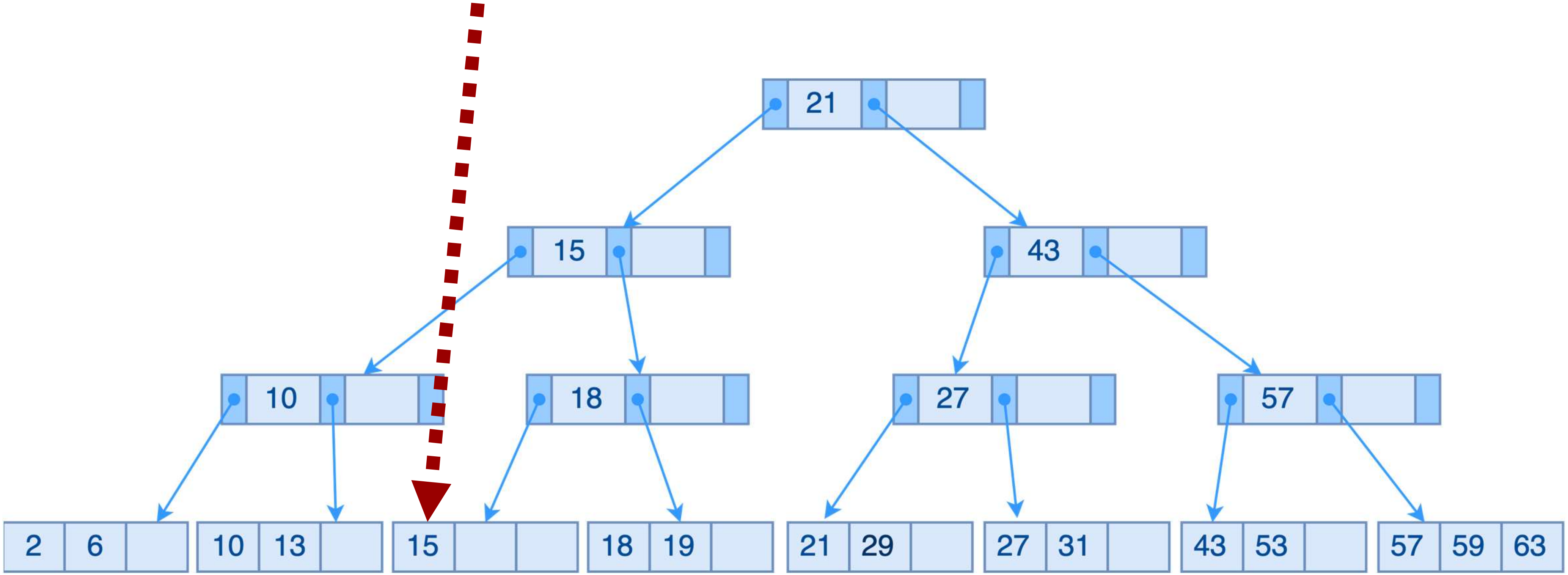


B - trees

- Deleting a value from a B-tree:
  - Finding the value
  - Removing the value from the node
    - the principles of the B-tree are preserved
    - if the node has fewer values than allowed - in this case the node **merging** process is applied

Example: B - trees

- Deleting of 15



Problem:
The node violates the B-tree structure

Solution:
Merging neighbouring nodes

Example: B - trees

- Deleting of 15