European Funds
Knowledge Education Development
Wrocław University of Science and Technology
European Union
European Social Fund
ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Data Structures and Algorithms – W07

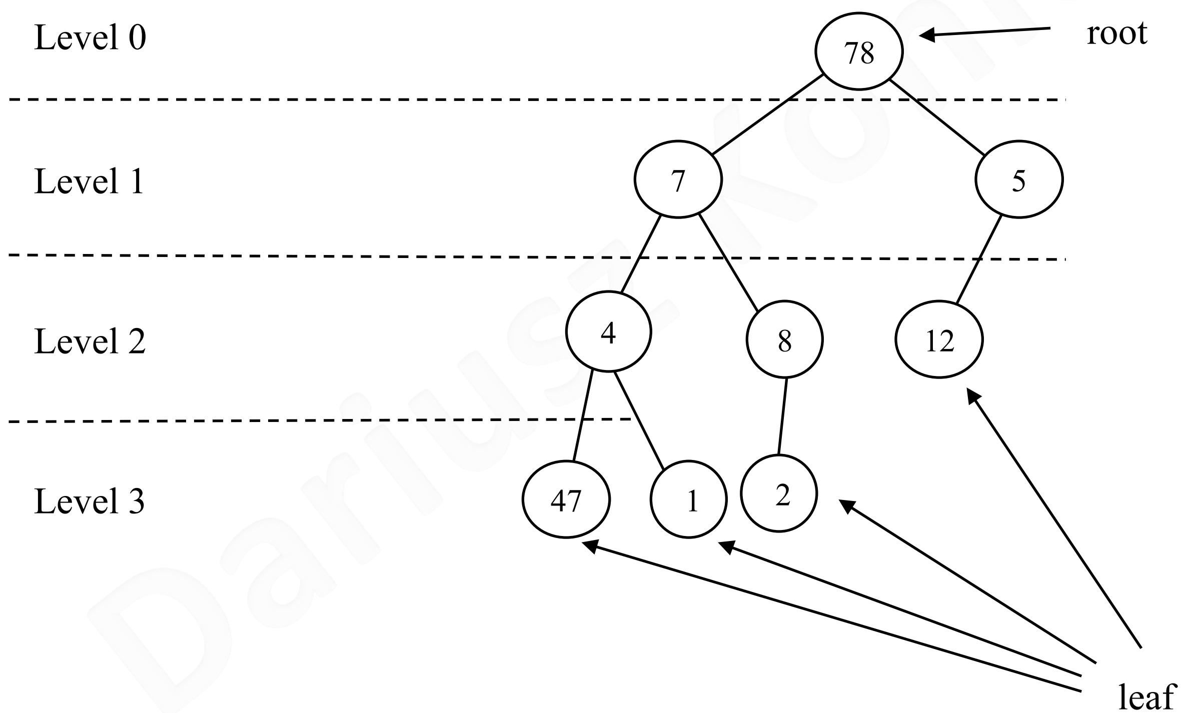
Binary Search Tree

Contents

- Binary tree – definition
- BST – definition
- BST: implementation with reference to the parent– pseudocode
  - Basic operations on BST: search, overview, min, max, successor, insertion, deletion
- BST: realizations without references to a parent – Java
  - Basic operations on BST: search, overview, min, max, successor, insertion, deletion
- The general scheme of methods on the binary tree
- Unbalanced tree
- Summary

Binary tree

- A **binary tree** is a coherent node structure in which each node can **have at most two** direct sub-nodes (called **children** / descendants).
- The node above the node is called the **parent** / ancestor.
- There is one node without a parent - it is called the **root**.
- Nodes without descendants are called **leaves**.
- The **level** / **depth** of the node is the number of edges from the root down to the node.
- The **height of the tree** is the maximum from the depth of the leaves.
- For an **empty tree**, the height is set to -1.

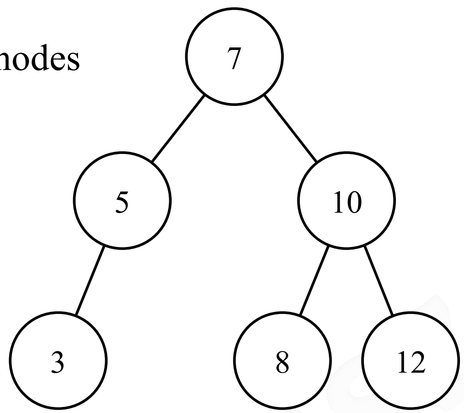


BST

- A **BST** tree (**binary search tree**), is a binary tree with an additional property:

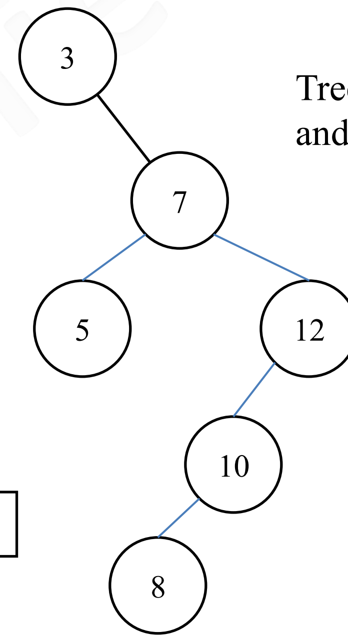
Let *x* be a node in a binary search tree. If *y* is a node in the left subtree of *x*, then *key*[*y*] ≤ *key*[*x*]. If *y* is a node in the right subtree of *x*, then *key*[*y*] ≥ *key*[x].

Tree with n = 6 nodes and height h = 3



log n <= h <= n

average height h=O(log n)

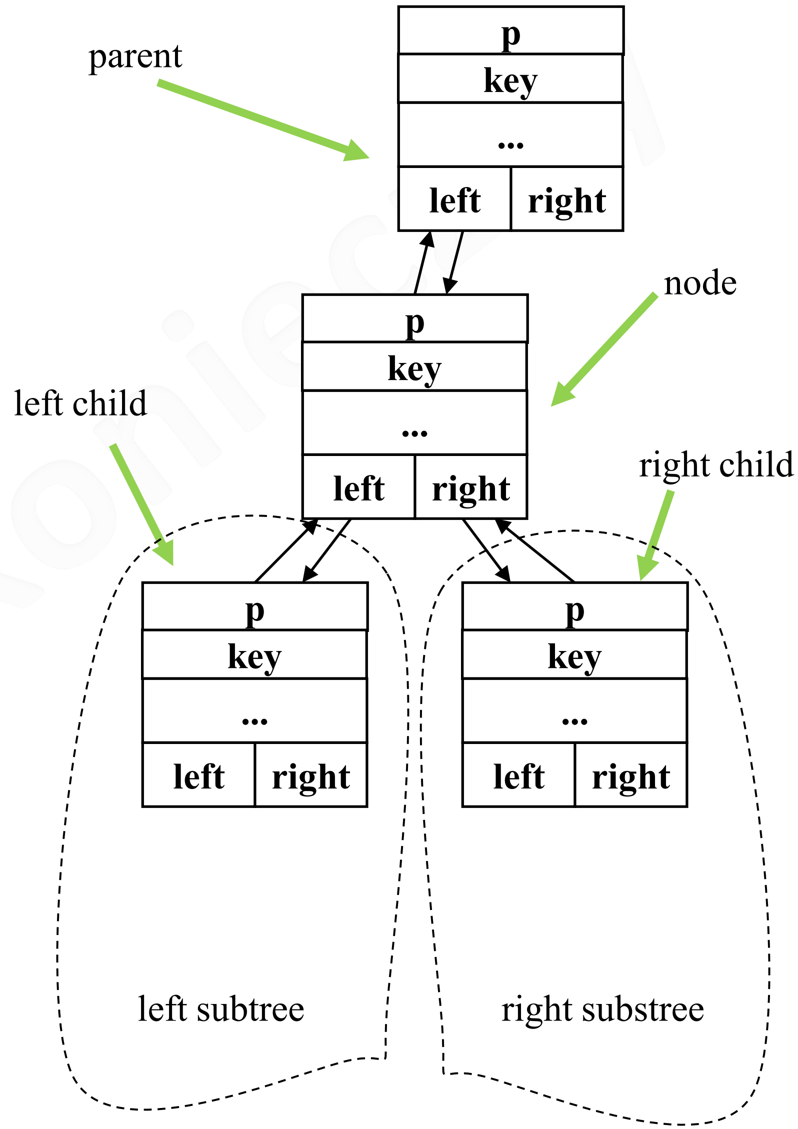


Tree with n = 6 nodes and height h = 5

- Some definitions assume that the keys in the tree **can not be repeated**, hence **sharp inequalities** appear in the definition. This condition **will be added** in the tree projects **in this lecture**.

BST – realization 1

- First realization:
  - Each node has a **key**, **left**, **right** and **p** fields to remember: key, reference to left and right child, reference to parent. It may also have some other additional fields.
  - The root in the parent field has **null**, in the same way the leaves in the left and right fields have **null**.
- The right descendant along with all subnodes up to the leaves is called the **right subtree**, the left subtree is an analogous structure.



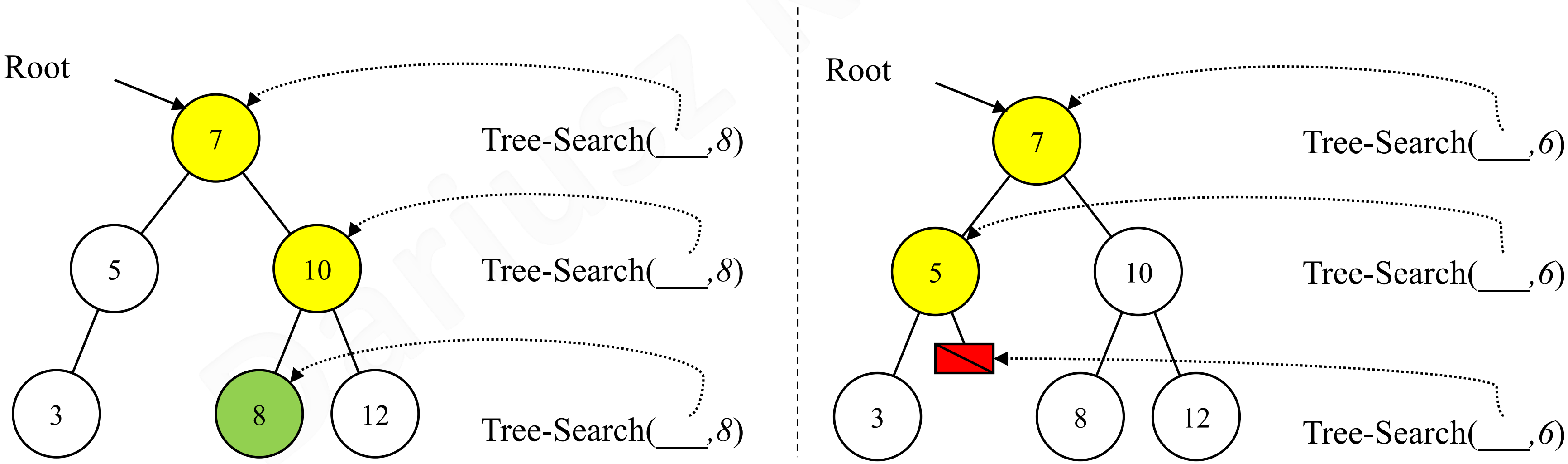
Searching in BST

- The parameter is the key of the element you are looking for. The result - a node with a found key or a **`null`** value.
- We use the BST property.
- Idea: Starting from the root, we compare the wanted key with the key in a given node. Equal - we found a node. The desired key is smaller - we are looking in the left sub-tree, the larger one - in the right sub-tree.

```
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)
```

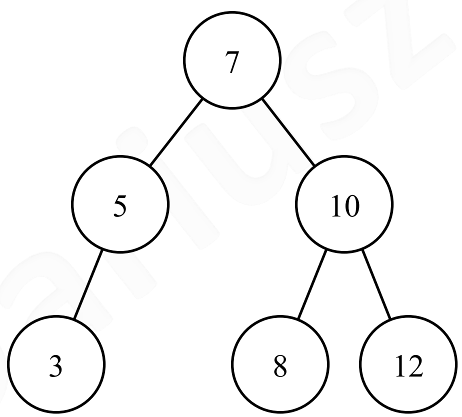
Tree-Search(root,*k*)

Complexity: O(*h*)



Tree walk in BST

- Browsing a tree (called **tree-walk**) involves visiting all vertices **exactly once** (in a systematic way).
- Visiting the vertex (eg writing to the stream) can be done in the following ways:
  - Before we visit his subtrees (preorder-walk)
  - After visiting the subtrees (postorder-walk)
  - Between visiting the subtrees (inorder-walk).
- The above methods have two versions:
  - first the left subtree, then the right one (this lecture)
  - first the right subtree, then the left one



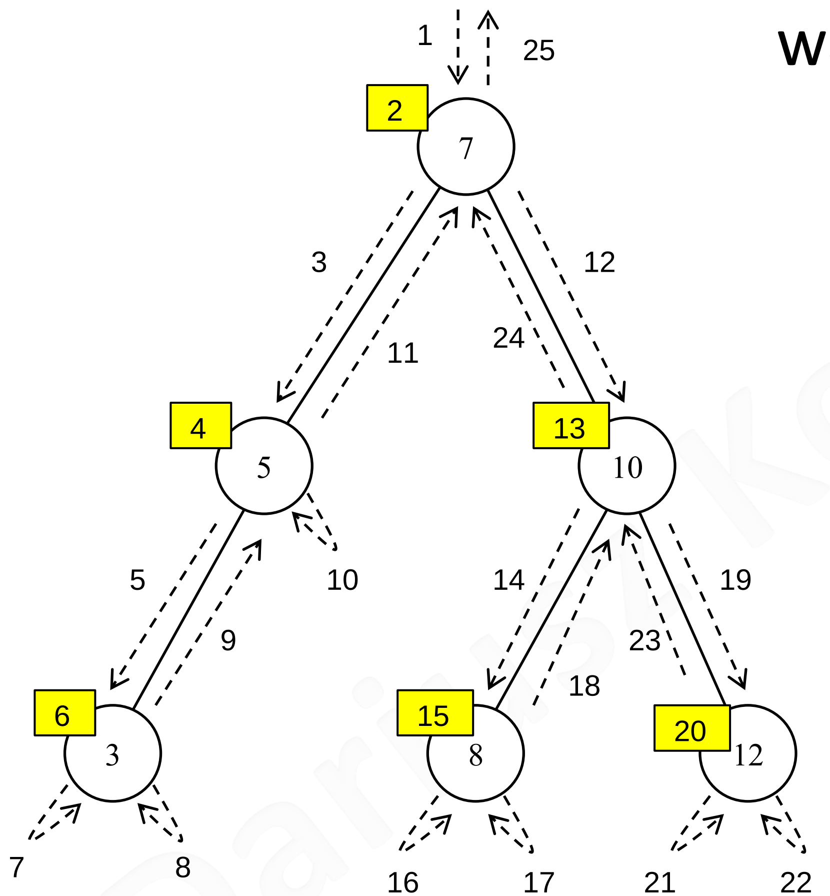
Preorder-Walk: 7,5,3,10,8,12

Postorder-Walk: 3,5,8,12,10,7

Inorder-Walk: 3,5,7,8,10,12

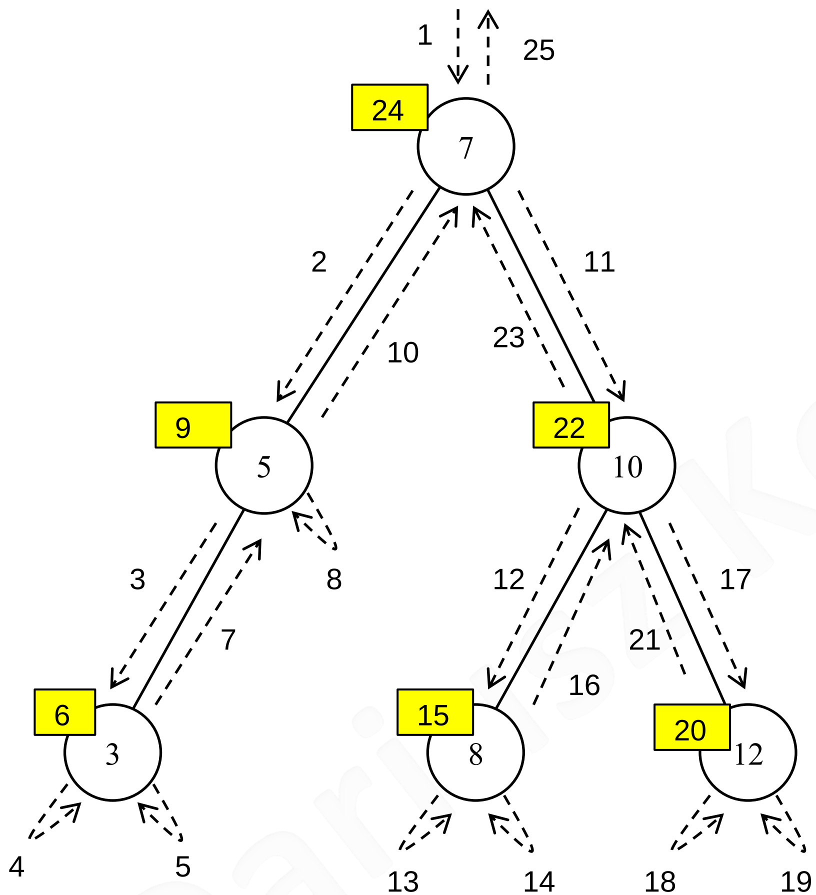
Pre-order walk

- The visit can be done **before** walking into subtrees



Preorder-Walk: 7,5,3,10,8,12

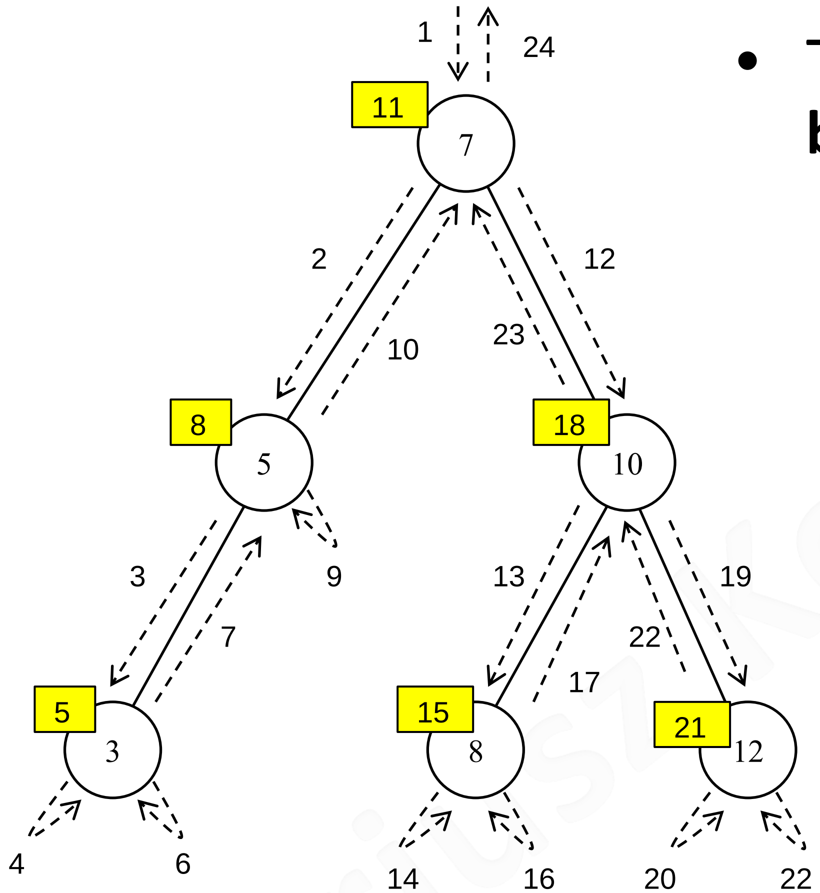
Post-order walk



- The visit can be done **after** walking into subtrees

Postorder-Walk: 3,5,8,12,10,7

In-order walk



- The visit can be done **between** walking into subtrees

Inorder-Walk: 3,5,7,8,10,12

In-order walk – code and analysis

```
    Tree-Inorder-Walk(x)
{1} if x <> null then
{2}   Tree-Inorder-Walk(left[x])
{3}   show key[x]
{4}   Tree-Inorder-Walk(right[x])
```

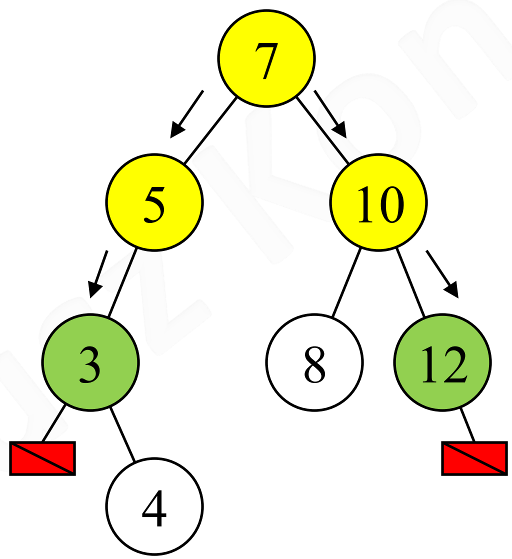
- Calling: Tree-Inorder-Walk (root)
- Common scheme in operation on a tree:
  - two methods:
    - one recursive works for the node specified in the call argument
    - the second calls the first with the root as an argument
- Other walks differ only in the place where the line {3} of code are called.
- In-order walk - walk in the order of sorted keys, hence the most used

- An interesting problem: creating iterators for each of the walk methods

Searching for min and max in BST

- Searching for minimum in BST consist in proceeding throw left children, till the node, which has no left child.
- Operation of searching maximum proceeds analogous using right children

```
Tree-Minimum(x)
while left[x] <> null do
    x := left[x]
return x
```

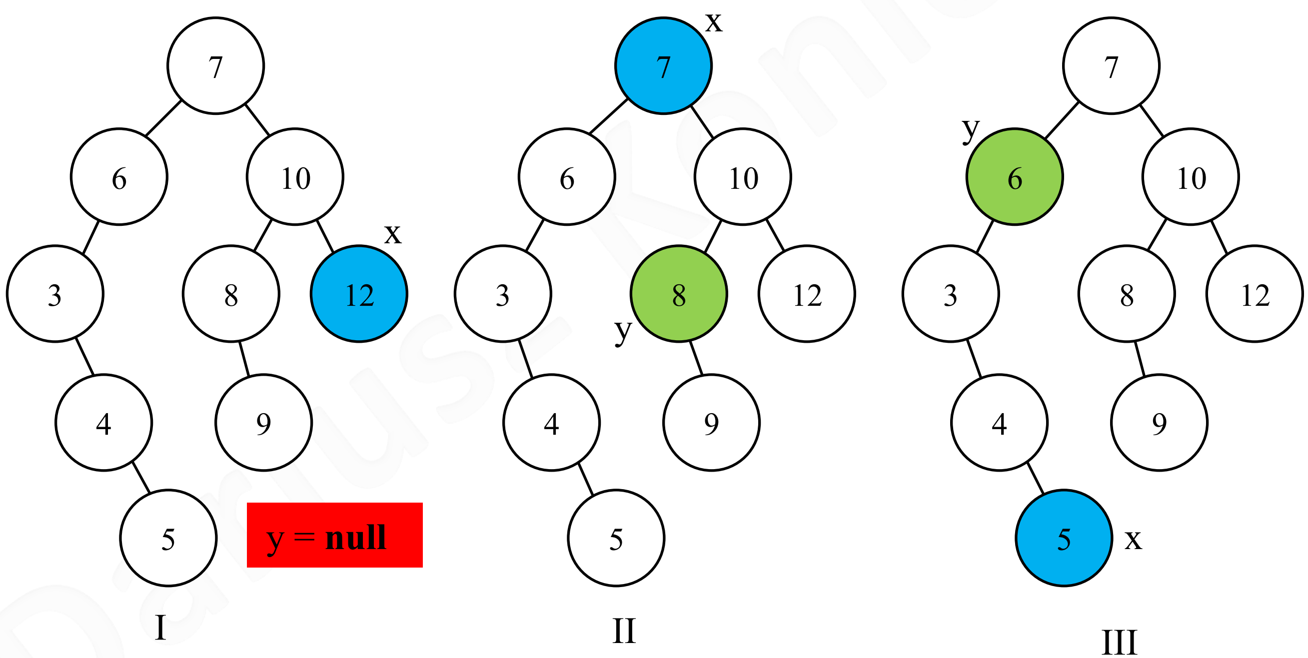



```
Tree-Maksimum(x)
while right[x] <> null do
  x := right[x]
return x
```

Complexity: O(h)

Successor and predecessor in BST

- Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk.
- There are 3 cases:
- I) node has no successor
- II) successor of node x is in its right subtree
- III) successor of node x is placed higher



Searching successor in BST

- In case II) it is to search minimum in its right subtree.
- In I) and III) cases it is to search a parent, which has to be on path from left child. If such node does not exist – successor also does not exist.

```
Tree-Successor(x)
if right[x] <> null then
  return Tree-Minimum(right[x])
y := p[x]
while (y <> null) and (x = right[y]) do
  x := y
  y := p[y]
return y
```

Complexity: O(*h*)

Insertion in BST 1/2

- Insertion new node with key v in BST consist in finding new position for this node as a leaf. It is similar to normal search.
- We assume that for new node `z` we have `key[z]=v`, `left[z]=`**`null`**, `right[z]=`**`null`**

```
Tree-Insert(root, z)
y := null
x := root
while x <> null do
  y := x
  if key[z] < key[x]
    then x := left[x]
    else x := right[x]
p[z] := y
if y = null
  then root := z
  else if key[z] < key [y]
    then left[y] := z
    else right[y] := z
```

Tree after insertion values in sequence:

5, 3, 7, 11, 4, 2, 12, 10

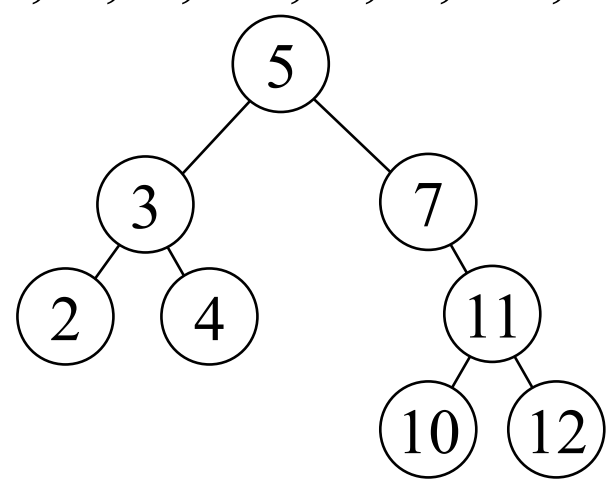
Insertion in BST 1/2

- Insertion new node with key v in BST consist in finding new position for this node as a leaf. It is similar to normal search.
- We assume that for new node `z` we have `key[z]=v`, `left[z]=`**`null`**, `right[z]=`**`null`**

```
Tree-Insert(root, z)
y := null
x := root
while x <> null do
  y := x
  if key[z] < key[x]
    then x := left[x]
    else x := right[x]
p[z] := y
if y = null
  then root := z
  else if key[z] < key [y]
    then left[y] := z
    else right[y] := z
```

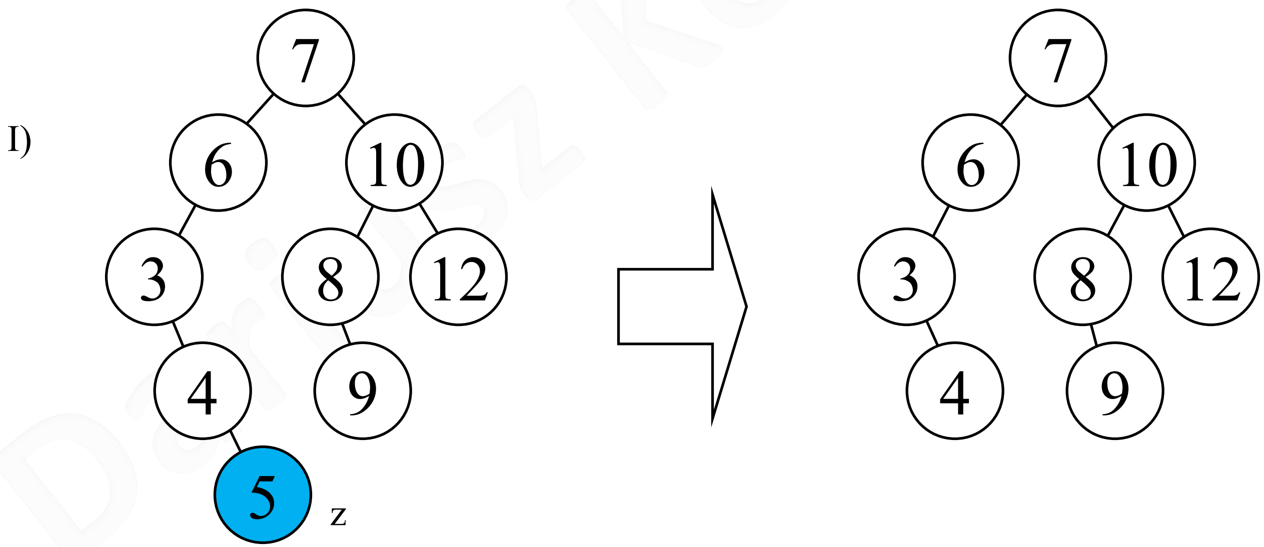
Tree after insertion values in sequence:

5, 3, 7, 11, 4, 2, 12, 10



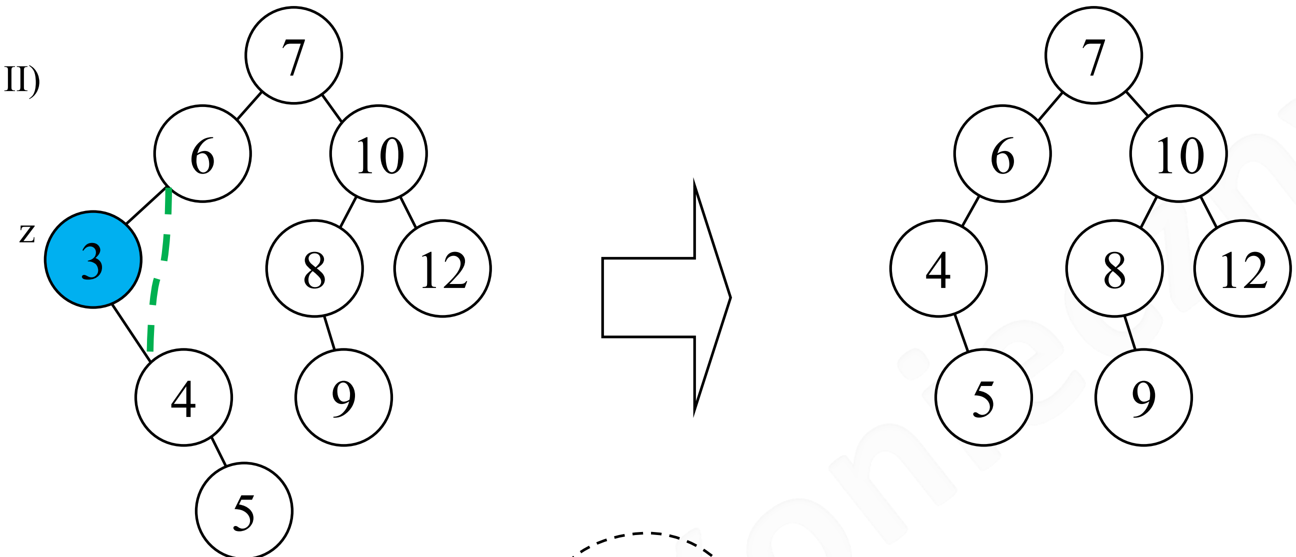
Deletion in BST 1/2

- Deletion in BST consist in repairing tree in minimal step to make proper BST.
- There are 3 cases during deletion a node ***z***:
  - I) node ***z*** has no children.
  - II) node ***z*** has exactly one child.
  - III) node ***z*** has two children.
- It has to be done:
  - I) in parent of node ***z*** modify proper child field to **null**.
  - II) in parent of node ***z*** modify proper child field, pointing to ***z,*** to value equal child of node ***z****.*
  - III) Find the succesor of ***z*** (let's mark it ***y***). This successor has no two children for sure. Swap data fields and key field in ***y*** and ***z***, and then delete node ***y*** (now it is case I or II).

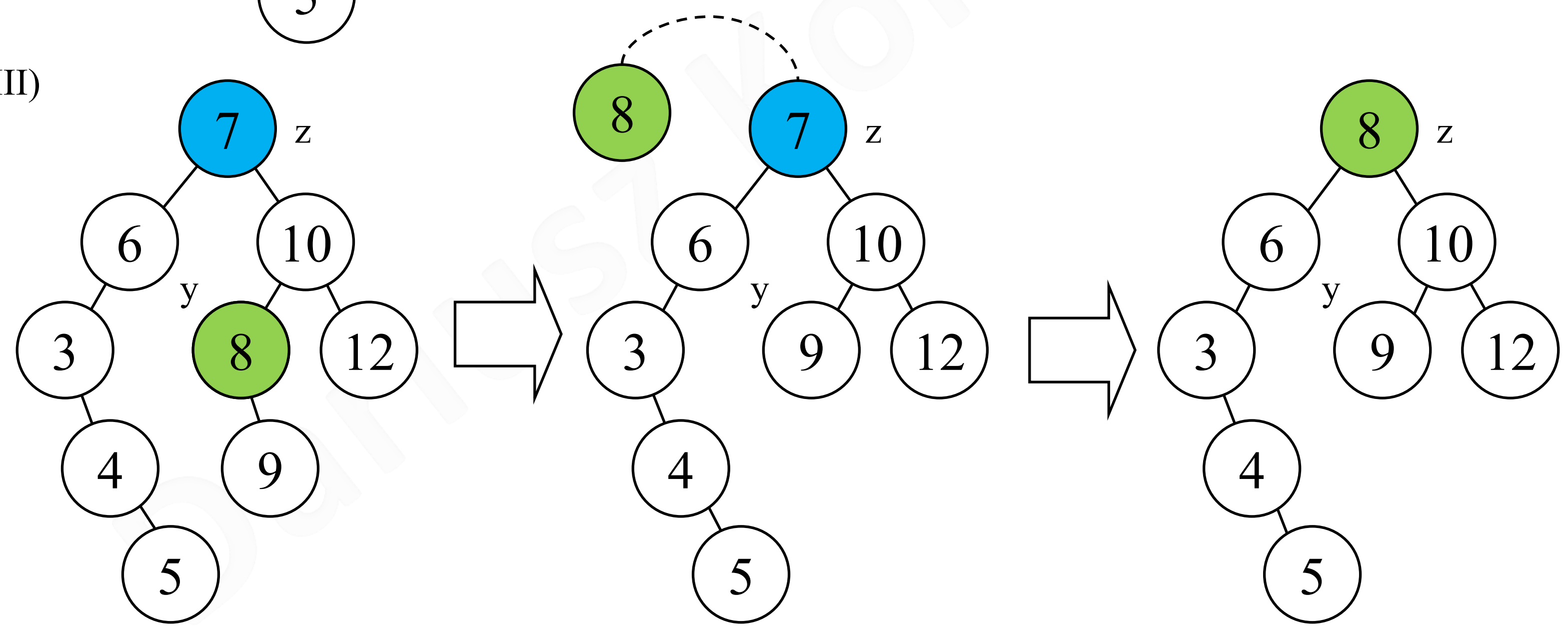


Deletion in BST 2/2

II)



III)



Deletion in BST – code and analysis

- It is easy also return the just removed node

```
Tree-Delete(root, z)
if (left[z] = null) or (right[z] = null)
  then y: = z
  else y := Tree-Successor(z)
if left[y] <> null
  then x := left[y]
  else x := right[y]
if x <> null
  then p[x] = p[y]
if p[y] = null
  then root := x
  else if y = left[p[y]]
    then left[p[y]] := x
    else right[p[y]] := x
if y <> z
  then swap(key[z],key[y])
 { if node y has another fields, copy them here}
return y
```

complexity: O(h)

BST – realization 2

- Most operations for BST do not require an up-link (to the parent)
- You can implement a node in the BST tree **without** binding to the parent.
- Using OOP instead of a key, it is better to use a **comparator.**
- The internal value (`value`) will be called the **element**.

```
public class BST<T> {
  class Node{
    T value; // element
    Node left;
    Node right;
    Node(T obj){
      value=obj;}
    Node(T obj, Node leftNode,Node rightNode){
      value=obj;
      left=leftNode;
      right=rightNode;}
  }
private final Comparator<T> _comparator;
private Node _root;
public BST(Comparator<T> comp){
  _comparator=comp;
  _root=null;}
```

Searching for an element

- It is practically no different from the first implementation.
- This time iterative implementation.
- A private function that returns a reference to a found node (and not just an element).

```
public T find(T elem){
  Node node=search(elem);
  return node==null?null:node.value;
}

private Node search(T elem) {
  Node node=_root;
  int cmp=0;
  while(node!=null && (cmp=_comparator.compare(elem, node.value))!=0)
    node=cmp<0? node.left:node.right;
  return node;
}
```

In-order walk with an executor 1/2

- Using the executor to better use the tree walk - instead of the show can be any method.
- The presented executor will be one-use object.

```
package aisd.executor;
public interface IExecutor<T,R> {
  void execute(T elem);
  R getResult();
}
```

```
public <R> void inOrderWalk(IExecutor<T,R> exec){
  inOrderWalk(_root,exec);}
private <R> void inOrderWalk(Node node, IExecutor<T,R> exec) {
  if(node!=null){
    inOrderWalk(node.left, exec);
    exec.execute(node.value);
    inOrderWalk(node.right, exec);
  }
}
```

- A more difficult problem - creating iterators for walks in this implementation - for the optimal computational complexity, use a stack inside the iterator that will remember the nodes on the path from the root.

In-order walk with an executor 2/2

- Executor with a buffer to remember the string

```
class IntegerToStringExec implements IExecutor<Integer, String>{
  StringBuffer line=new StringBuffer();
  @Override
  public void execute(Integer elem) {
    line.append(elem+"; ");}
  @Override
  public String getResult(){
    line.delete(line.length()-2, line.length());
    return line.toString();}}
```

```
public static void main(String[] args) {
  BST<Integer> tree=new BST<Integer>(new Comparator<Integer>(){
    public int compare(Integer o1, Integer o2) {
      return o1-o2;}});
tree.insert(7);
tree.insert(5);
tree.insert(2);
tree.insert(10);
tree.insert(12);
IntegerToStringExec exec=new IntegerToStringExec();
tree.inOrderWalk(exec);
System.out.println(exec.getResult());
```

```
2; 5; 7; 10; 12
```

Searching for min and max

- In OOP practice, it is worth splitting into a private method looking for a node and a public method returning item stored in the node.

```
public T getMin(){
  if(_root==null) throw new NoSuchElementException();
  Node node=getMin(_root);
  return node.value;
}
public T getMax(){
  if(_root==null) throw new NoSuchElementException();
  Node node=getMax(_root);
  return node.value;
}
private Node getMin(Node node) {
  assert(node!=null);
  while(node.left!=null)
    node=node.left;
  return node;
}
private Node getMax(Node node) {
  assert(node!=null);
  while(node.right!=null)
    node=node.right;
  return node;
}
```

Searching for a successor

- Searching for a successor, having a different node as an argument, makes it impossible to use the algorithm from the first implementation
  - We can not move from the node towards the root
- We will **look for** the successor of the element. From here, first look for a node with an element, and then returning you can use the return path for cases I and III (slide 13)

```
public T successor(T elem){
  Node succNode=successorNode(_root, elem);
  return succNode==null?null:succNode.value;}
private Node successorNode(Node node, T elem) {
  if(node==null) throw new NoSuchElementException();
  int cmp=_comparator.compare(elem,node.value);
  if(cmp==0){
    if(node.right!=null)
      return getMin(node.right);
    else return null;
  } else if(cmp<0){
    Node retNode=successorNode(node.left, elem);
    return retNode==null?node:retNode;
  } else { // cmp>0
    return successorNode(node.right, elem);
  }
}
```

Insertion in BST 1/3

- The iterative version as an exercise - analogically to the code of the first implementation, without the lines {8} (slide 15), with the creation of the element just before insertion.
- The recursive version - Java passes arguments to the methods by value, so in order to insert a new value, you must re-assign the subtrees in which the new element will be added.

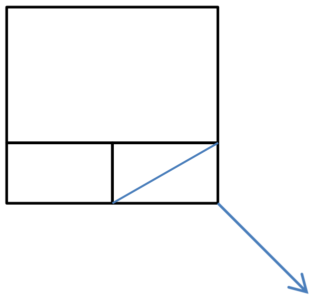
```
public void insert(T elem){
  _root=insert(_root,elem);}

private Node insert(Node node, T elem) {
  if(node==null)
    node=new Node(elem);
  else{
    int cmp=_comparator.compare(elem, node.value);
    if(cmp<0)
      node.left=insert(node.left,elem);
    else if(cmp>0)
      node.right=insert(node.right,elem);
    else
      throw new DuplicateElementException(elem.toString());
  }
  return node;
}
```

Insertion in BST 2/3

- Comming to a place for a new element

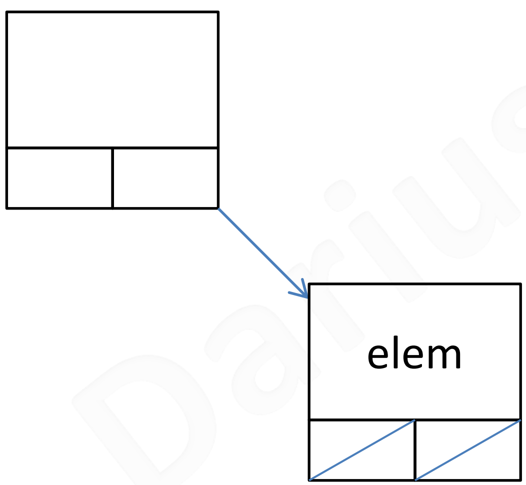
node



```
node.right=insert(node.right,elem)
```

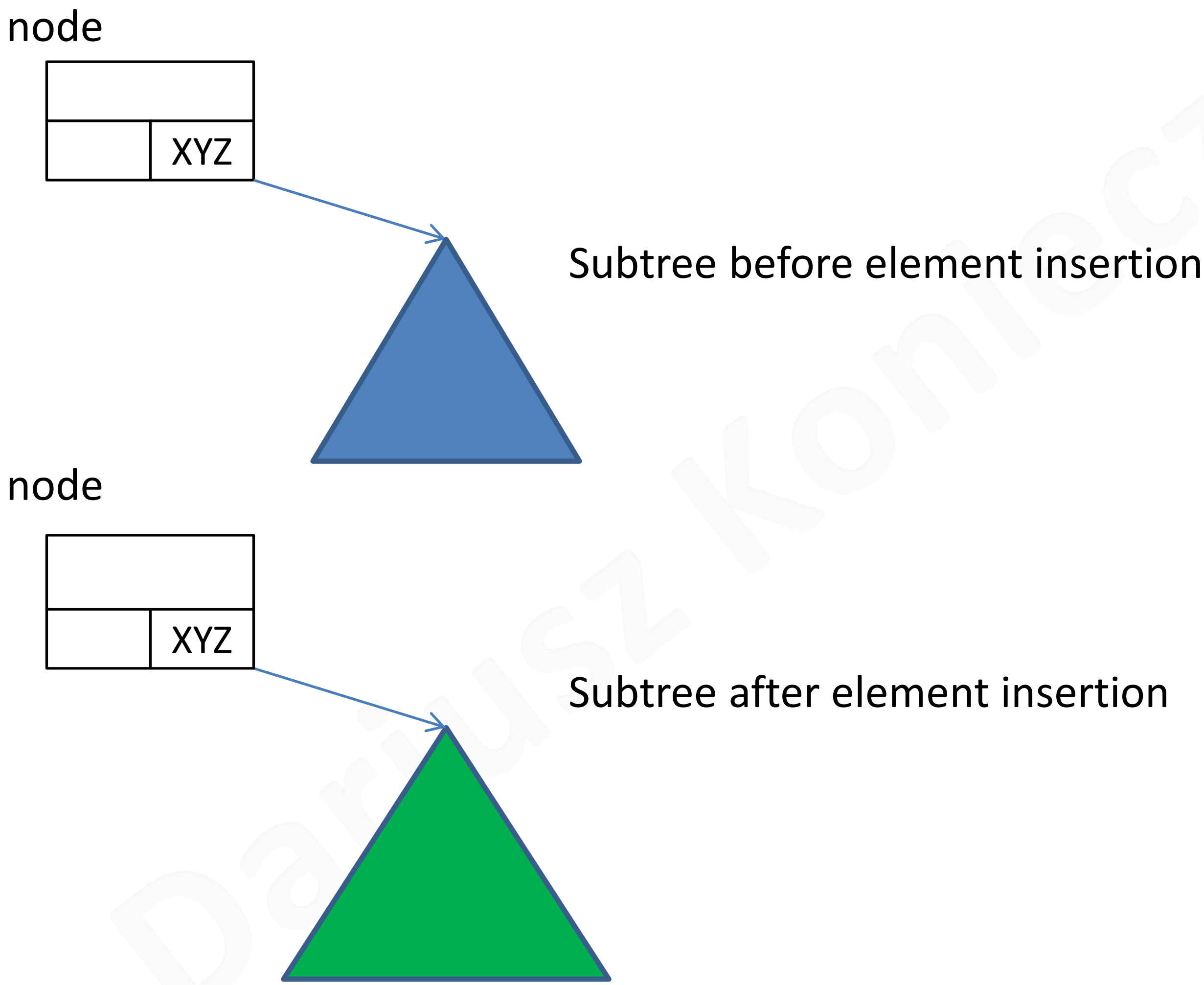
```
if(node==null)
  node=new Node(elem);
else{ …
}
return node;
```

node



Insertion in BST 3/3

- Any other situation



Element deletion in BST 1/2

- You can not use the version from the first implementation directly
- The version of the node's search should be completed so that it also returns the parent node.
- The second concept is a recursive implementation that returns back to the parent (returning a similar idea to adding)
- For simplicity, the public method will be `void`.
- If we go down to case I or II, we will solve it without a problem. In the case of III - two children - a separate method will remove the minimum from the right subtree, earlier the value of this node inserting into the node with two children.

```
public void delete(T elem){
  _root=delete(elem,_root);
}

//… kontynuacja na następnym slajdzie
```

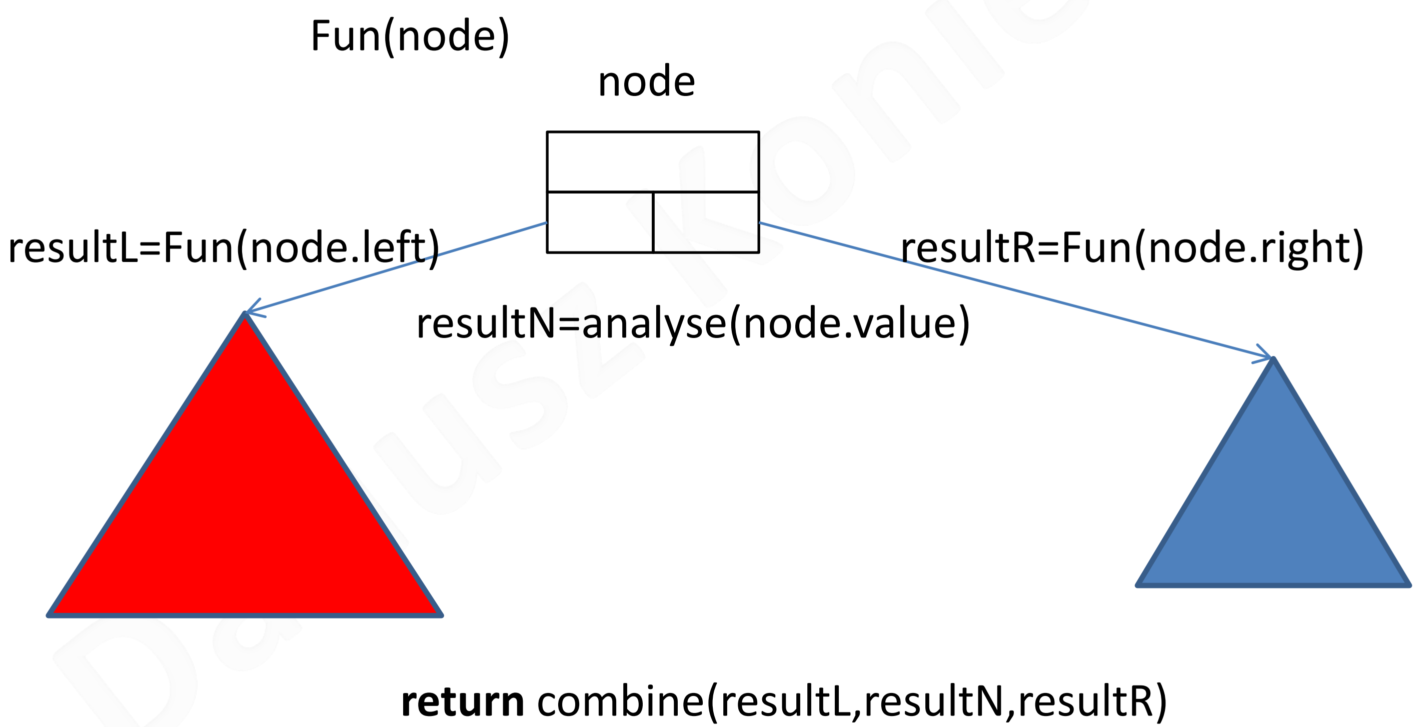
Element deletion in BST 2/2

```
protected Node delete(T elem, Node node) {
  if(node==null) throw new NoSuchElementException();
  else {
    int cmp=_comparator.compare(elem,node.value);
    if(cmp<0)
      node.left=delete(elem,node.left);
    else if(cmp>0)
      node.right=delete(elem,node.right);
    else if(node.left!=null &&node.right!=null)
      node.right=detachMin(node,node.right);
    else node = (node.left != null) ? node.left : node.right;
  }
  return node;
}

private Node detachMin(Node del, Node node) {
  if(node.left!=null) node.left=detachMin(del, node.left);
  else {
    del.value=node.value;
    node=node.right;
  }
  return node;
}
```

Diagram of methods for a binary tree

- A large part of methods that do not modify binary tree, implemented recursively, work according to the following scheme ("divide and conquere"):
  - Follow the method for the left child (and get the resultL)
  - Follow the method for the right child (and get the resultR)
  - Analyze the element of the input node (and get the resultN)
  - Combine all results to get a final result for the subtree rooted in the node

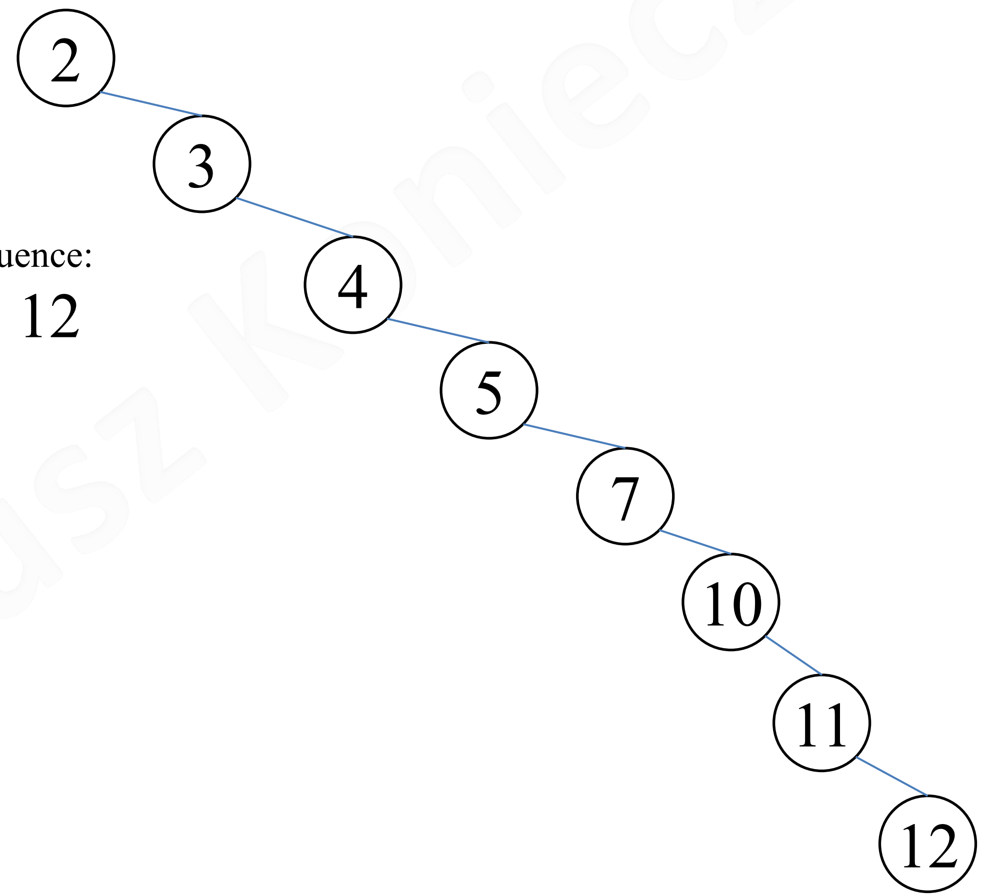


Unbalanced tree

- A simple BST tree during creation / removal can become very unbalanced.
- Paradoxically, the worst sequence of values is e.g. a sorted sequence - it creates a degenerate tree. Degenerate to the linked list with linear complexity of most operations

Tree after insertion values in sequence:

2, 3, 4, 5, 7, 10, 11, 12



BST - summary

- The complexity of most operations on BST depends on the height of the tree: O(h)
- A simple BST tree can be very unbalanced: h = O(n)
- You can implement a BST tree with and without a parent reference:
  - Advantages and disadvantages
- Many operations can be implemented recursively or iteratively:
  - Some of the operations implemented iteratively require an internal stack, and some do not.

- There is no implementation of a simple BST tree in the Java standard library.
  - Not always the optimal structure