DATA STRUCTURES AND ALGORITHMS

LECTURE 12-13

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 11...

- Hash Tables
    - Separate chaining
    - Coalesced chaining
    - Open addressing

Today

Trees

- Trees are one of the most commonly used data structures because they offer an efficient way of storing data and working with the data.
- In graph theory a *tree* is a connected, acyclic graph (usually undirected).
- When talking about trees as a data structure, we actually mean *rooted trees*, trees in which one node is designated to be the *root* of the tree.

Tree - Definition

- A tree is a finite set $\mathcal{T}$ of 0 or more elements, called *nodes*, with the following properties:
  - If $\mathcal{T}$ is empty, then the tree is empty
  - If $\mathcal{T}$ is not empty then:
    - There is a special node, R, called the *root* of the tree
    - The rest of the nodes are divided into k ($k \geq 0$) disjunct *trees*, T_1, T_2, ..., T_k, the root node R being linked by an edge to the root of each of these trees. The trees T_1, T_2, ..., T_k are called the *subtrees* (*children*) of R, and R is called the *parent* of the subtrees.

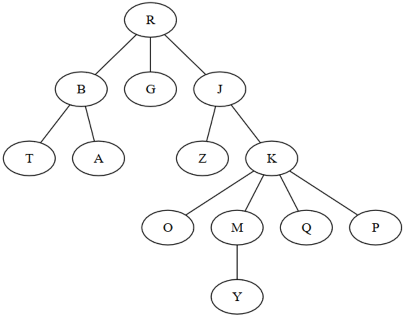
Tree - Terminology I

- An *ordered tree* is a tree in which the order of the children is well defined and relevant (instead of having a set of children, each node has a list of children).
- The *degree* of a node is defined as the number of children of the node.
- The nodes with the degree 0 (nodes without children) are called *leaf nodes*.
- The nodes that are not leaf nodes are called *internal nodes*.

Tree - Terminology II

- The *depth* or *level* of a node is the length of the path (measured as the number of edges traversed) from the root to the node. This path is unique. The root of the tree is at level 0 (and has depth 0).
- The *height* of a node is the length of the longest path from the node to a leaf node.
- The *height of the tree* is defined as the height of the root node, i.e., the length of the longest path from the root to a leaf.

Tree - Terminology Example



- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node *K*: 2 (path R-J-K)
- Height of node *K*: 2 (path K-M-Y)
- Height of the tree (height of node *R*): 4
- Nodes on level 2: T, A, Z, K

k-ary trees

- How can we represent a tree in which every node has at most *k* children?
- One option is to have a structure for a *node* that contains the following:
    - information from the node
    - address of the parent node (not mandatory)
    - *k* fields, one for each child

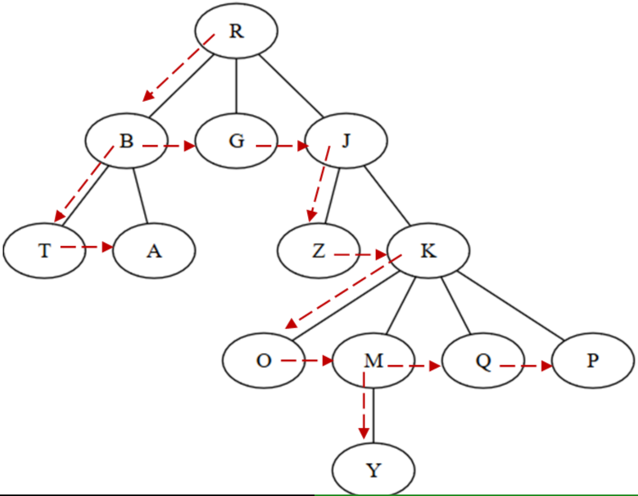
k-ary trees

- Another option is to have a structure for a *node* that contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - an array of dimension k, in which each element is the address of a child
  - number of children (number of occupied positions from the above array)
- Disadvantage of these approaches is that we occupy space for k children even if most nodes have less children.

k-ary trees

- A third option is the so-called *left-child right-sibling* representation in which we have a structure for a node which contains the following:
    - information from the node
    - address of the parent node (not mandatory)
    - address of the leftmost child of the node
    - address of the right sibling of the node (next node on the same level from the same parent).

Left-child right sibling representation example



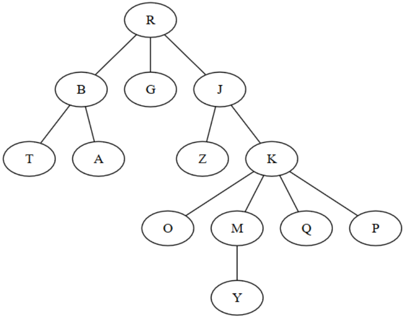
Tree traversals

- A node of a tree is said to be *visited* when the program control arrives at the node, usually with the purpose of performing some operation on the node (printing it, checking the value from the node, etc.).
- *Traversing* a tree means visiting all of its nodes.
- For a k-ary tree there are 2 possible traversals:
  - Depth-first traversal
  - Level order (breadth first) traversal

Depth first traversal

- Traversal starts from root
- From root we visit one of the children, then one child of that child, and so on. We go down (in depth) as much as possible, and continue with other children of a node only after all descendants of the "first" child were visited.
- For depth first traversal we use a stack to remember the nodes that have to be visited.

Depth first traversal example

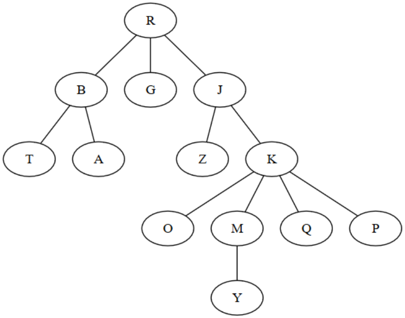


- Stack *s* with the root: *R*
- Visit *R* (pop from stack) and push its children: *s* = [B G J]
- Visit *B* and push its children: *s* = [T A G J]
- Visit *T* and push nothing: *s* = [A G J]
- Visit *A* and push nothing: *s* = [G J]
- Visit *G* and push nothing: *s* = [J]
- Visit *J* and push its children: *s* = [Z K]

Level order traversal

- Traversal starts from root
- We visit all children of the root (one by one) and once all of them were visited we go to their children and so on. We go down one level, only when all nodes from a level were visited.
- For level order traversal we use a queue to remember the nodes that have to be visited.

Level order traversal example

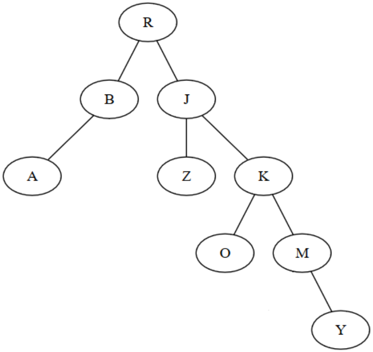


- Queue q with the root: R
- Visit R (pop from queue) and push its children: q = [B G J]
- Visit B and push its children: q = [G J T A]
- Visit G and push nothing: q = [J T A]
- Visit J and push its children: q = [T A Z K]
- Visit T and push nothing: q = [A Z K]
- Visit A and push nothing: q = [Z K]

Binary trees

- An ordered tree in which each node has at most two children is called *binary tree*.
- In a binary tree we call the children of a node the *left child* and *right child*.
- Even if a node has only one child, we still have to know whether that is the left or the right one.

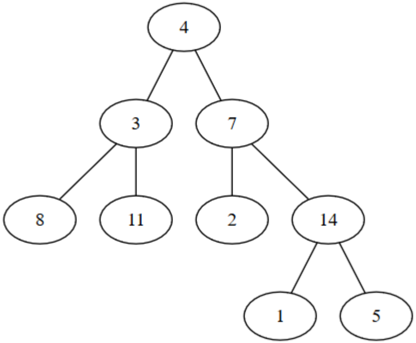
Binary tree - example



- A is the left child of B
- Y is the right child of M

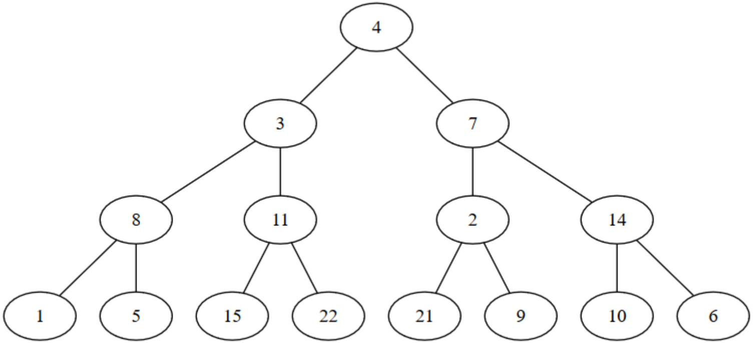
Binary tree - Terminology I

- A binary tree is called *full* if every internal node has exactly two children.



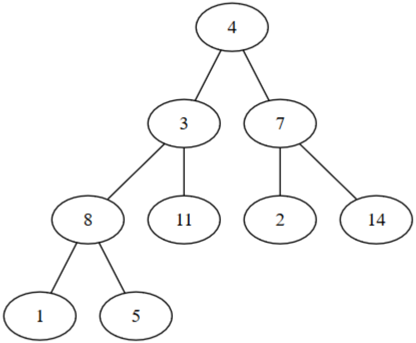
Binary tree - Terminology II

- A binary tree is called *complete* if all leaves are one the same level and all internal nodes have exactly 2 children.



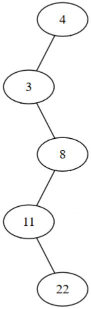
Binary tree - Terminology III

- A binary tree is called *almost complete* if it is a *complete* binary tree except for the last level, where nodes are completed from left to right (binary heap - structure).



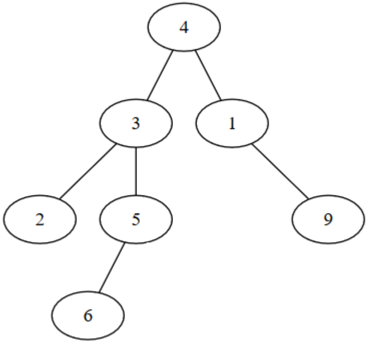
Binary tree - Terminology IV

- A binary tree is called *degenerate* if every internal node has exactly one child (it is actually a chain of nodes).



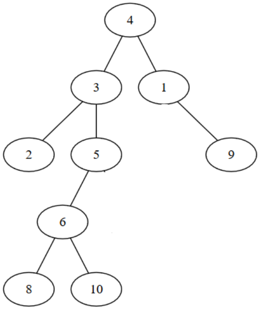
Binary tree - Terminology V

- A binary tree is called *balanced* if the difference between the height of the left and right subtrees is at most 1 (for every node from the tree).



Binary tree - Terminology VI

- Obviously, there are many binary trees that are none of the above categories, for example:



Binary tree - properties

- A binary tree with n nodes has exactly $n - 1$ edges (this is true for every tree, not just binary trees)
- The number of nodes in a complete binary tree of height N is $2^{N+1} - 1$ (it is $1 + 2 + 4 + 8 + ... + 2^N$)
- The maximum number of nodes in a binary tree of height N is $2^{N+1} - 1$ - if the tree is complete.
- The minimum number of nodes in a binary tree of height N is N - if the tree is degenerate.
- A binary tree with N nodes has a height between $log_2 N$ and N.

ADT Binary Tree I

- Domain of ADT Binary Tree:

$$\mathcal{BT} = \{bt \mid bt \text{ binary tree with nodes containing information of type TElem}\}$$

ADT Binary Tree II

- init(bt)
  - **descr:** creates a new, empty binary tree
  - **pre:** true
  - **post:** $bt \in \mathcal{BT}$, bt is an empty binary tree

ADT Binary Tree III

- initLeaf(bt, e)
  - **descr:** creates a new binary tree, having only the root with a given value
  - **pre:** $e \in TElem$
  - **post:** $bt \in \mathcal{BT}$, bt is a binary tree with only one node (its root) which contains the value e

ADT Binary Tree IV

- initTree(bt, left, e, right)
  - **descr:** creates a new binary tree, having a given information in the root and two given binary trees as children
  - **pre:** $left, right \in \mathcal{BT}$, $e \in TElem$
  - **post:** $bt \in \mathcal{BT}$, bt is a binary tree with left child equal to *left*, right child equal to *right* and the information from the root is e

ADT Binary Tree V

- insertLeftSubtree(bt, left)
    - **descr:** sets the left subtree of a binary tree to a given value (if the tree had a left subtree, it will be changed)
    - **pre:** $bt, left \in \mathcal{BT}$
    - **post:** $bt' \in \mathcal{BT}$, the left subtree of bt' is equal to *left*

ADT Binary Tree VI

- insertRightSubtree(bt, right)
  - **descr:** sets the right subtree of a binary tree to a given value (if the tree had a right subtree, it will be changed)
  - **pre:** $bt, right \in \mathcal{BT}$
  - **post:** $bt' \in \mathcal{BT}$, the right subtree of bt' is equal to $right$

ADT Binary Tree VII

- root(bt)
    - **descr:** returns the information from the root of a binary tree
    - **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
    - **post:** $root \leftarrow e$, $e \in TElem$, e is the information from the root of bt
    - **throws:** an exception if bt is empty

ADT Binary Tree VIII

- left(bt)
  - **descr:** returns the left subtree of a binary tree
  - **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
  - **post:** $left \leftarrow, l,\ l \in \mathcal{BT},\ l$ is the left subtree of bt
  - **throws:** an exception if bt is empty

ADT Binary Tree IX

- right(bt)
    - **descr:** returns the right subtree of a binary tree
    - **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
    - **post:** $right \leftarrow r$, $r \in \mathcal{BT}$, r is the right subtree of bt
    - **throws:** an exception if bt is empty

ADT Binary Tree X

- isEmpty(bt)
  - **descr:** checks if a binary tree is empty
  - **pre:** $bt \in \mathcal{BT}$
  - **post:**

$$empty \leftarrow \begin{cases} True, & \text{if } bt = \Phi \\ False, & \text{otherwise} \end{cases}$$

ADT Binary Tree XI

- iterator (bt, traversal, i)
    - **descr:** returns an iterator for a binary tree
    - **pre:** $bt \in \mathcal{BT}$, *traversal* represents the order in which the tree has to be traversed
    - **post:** $i \in \mathcal{I}$, *i* is an iterator over *bt* that iterates in the order given by *traversal*

ADT Binary Tree XII

- destroy(bt)
    - **descr:** destorys a binary tree
    - **pre:** $bt \in \mathcal{BT}$
    - **post:** bt was destroyed

ADT Binary Tree XIII

- Other possible operations:
    - change the information from the root of a binary tree
    - remove a subtree (left or right) of a binary tree
    - search for an element in a binary tree
    - return the number of elements from a binary tree

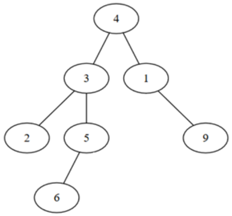
Possible representations

- If we want to implement a binary tree, what representation can we use?
- We have several options:
  - Representation using an array (similar to a binary heap)
  - Linked representation
    - with dynamic allocation
    - on an array

Possible representations I

- Representation using an array
  - Store the elements in an array
  - First position from the array is the root of the tree
  - Left child of node from position i is at position $2 * i$, right child is at position $2 * i + 1$.
  - Some special value is needed to denote the place where no element is.

Possible representations II



Pos	Elem
1	4
2	3
3	1
4	2
5	5
6	-1
7	9
8	-1
9	-1
10	6
11	-1
12	-1
13	-1
...	...

- Disadvantage: depending on the form of the tree, we might waste a lot of space.

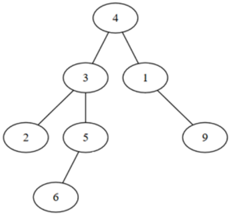
Possible representations I

- Linked representation with dynamic allocation
  - We have a structure to represent a node, containing the information, the address of the left child and the address of the right child (possibly the address of the parent as well).
  - An empty tree is denoted by the value NIL for the root.
  - We have one node for every element of the tree.

Possible representations II

- Linked representation on an array
  - Information from the nodes is placed in an array. The *address* of the left and right child is the *index* where the corresponding elements can be found in the array.
  - We can have a separate array for the parent as well.

Possible representations III



Pos	1	2	3	4	5	6	7	8
Info	4	3	2	5	6	1	9	
Left	2	3	-1	5	-1	-1	-1	
Right	6	4	-1	-1	-1	7	-1	
Parent	-1	1	2	2	4	1	6	

- We need to know that the root is at position 1 (could be any position).
- If the array is full, we can allocate a larger one.
- We can keep a linked list of empty positions to make adding a new node easier.

Binary Tree Traversal

- For a binary tree there are 4 possible traversals:
  - Preorder
  - Inorder
  - Postorder
  - Level order (breadth first) - the same as in case of a general tree

Binary tree representation

- In the following, for the implementation of the traversal algorithms, we are going to use the following representation for a binary tree:

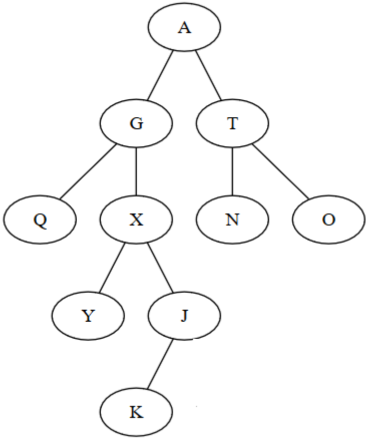
```
BTNode:
    info: TElem
    left: ↑ BTNode
    right: ↑ BTNode
```

```
BinaryTree:
    root: ↑ BTNode
```

Preorder traversal

- In case of a preorder traversal:
  - Visit the *root* of the tree
  - Traverse the left subtree - if exists
  - Traverse the right subtree - if exists
- When traversing the subtrees (left or right) the same preorder traversal is applied (so, from the left subtree we visit the root first and then traverse the left subtree and then the right subtree).

Preorder traversal example



- Preorder traversal: A, G, Q, X, Y, J, K, T, N, O

Preorder traversal - recursive implementation

- The simplest implementation for preorder traversal is with a recursive algorithm.

```
subalgorithm preorder_recursive(node) is:
//pre: node is a ↑ BTNode
  if node ≠ NIL then
    @visit [node].info
    preorder_recursive([node].left)
    preorder_recursive([node].right)
  end-if
end-subalgorithm
```

Preorder traversal - recursive implementation

- The *preorder_recursive* subalgorithm receives as parameter a pointer to a node, so we need a wrapper subalgorithm, one that receives a *BinaryTree* and calls the function for the root of the tree.

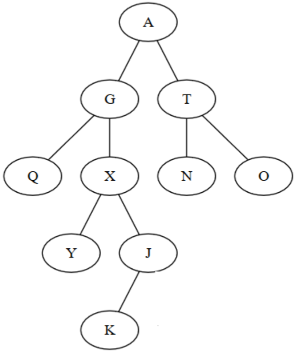
```
subalgorithm preorderRec(tree) is:
//pre: tree is a BinaryTree
  preorder_recursive(tree.root)
end-subalgorithm
```

- Assuming that visiting a node takes constant time (print the info from the node, for example), the whole traversal takes $\Theta(n)$ time for a tree with n nodes.

Preorder traversal - non-recursive implementation

- We can implement the preorder traversal algorithm without recursion, using an auxiliary *stack* to store the nodes.
    - We start with an empty stack
    - Push the root of the tree to the stack
    - While the stack is not empty:
        - Pop a node and visit it
        - Push the node's right child to the stack
        - Push the node's left child to the stack

Preorder traversal - non-recursive implementation example



- Stack: A
- Visit A, push children (Stack: T G)
- Visit G, push children (Stack: T X Q)
- Visit Q, push nothing (Stack: T X)
- Visit X, push children (Stack: T J Y)
- Visit Y, push nothing (Stack: T J)
- Visit J, push child (Stack: T K)
- Visit K, push nothing (Stack: T)
- Visit T, push children (Stack: O N)
- Visit N, push nothing (Stack: O)
- Visit O, push nothing (Stack:)
- Stack is empty, traversal is complete

Preorder traversal - non-recursive implementation

```
subalgorithm preorder(tree) is:
//pre: tree is a binary tree
    s: Stack //s is an auxiliary stack
    if tree.root $\neq$ NIL then
        push(s, tree.root)
    end-if
    while not isEmpty(s) execute
        currentNode $\leftarrow$ pop(s)
        @visit currentNode
        if [currentNode].right $\neq$ NIL then
            push(s, [currentNode].right)
        end-if
        if [currentNode].left $\neq$ NIL then
            push(s, [currentNode].left)
        end-if
    end-while
end-subalgorithm
```

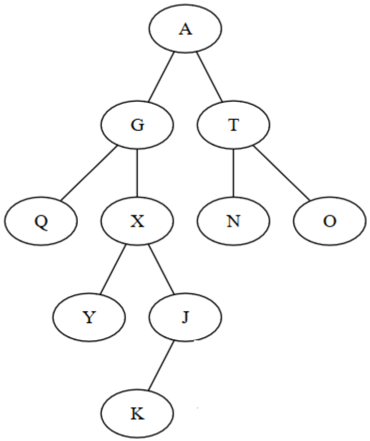
Preorder traversal - non - recursive implementation

- Time complexity of the non-recursive traversal is $\Theta(n)$, and we also need $O(n)$ extra space (the stack)

Inorder traversal

- In case of *inorder* traversal:
  - Traverse the left subtree - if exists
  - Visit the *root* of the tree
  - Traverse the right subtree - if exists
- When traversing the subtrees (left or right) the same inorder traversal is applied (so, from the left subtree we traverse the left subtree, then we visit the root and then traverse the right subtree).

Inorder traversal example



- Inorder traversal: Q, G, Y, X, K, J, A, N, T, O

Inorder traversal - recursive implementation

- The simplest implementation for inorder traversal is with a recursive algorithm.

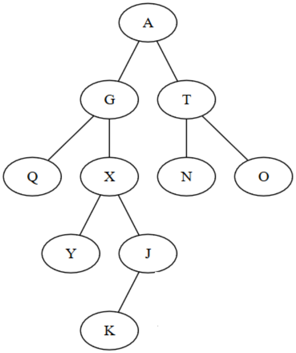
```
subalgorithm inorder_recursive(node) is:
//pre: node is a ↑ BTNode
  if node $\neq$ NIL then
    inorder_recursive([node].left)
    @visit [node].info
    inorder_recursive([node].right)
  end-if
end-subalgorithm
```

- We need again a wrapper subalgorithm to perform the first call to *inorder_recursive* with the root of the tree as parameter.
- The traversal takes $\Theta(n)$ time for a tree with n nodes.

Inorder traversal - non-recursive implementation

- We can implement the inorder traversal algorithm without recursion, using an auxiliary stack to store the nodes.
    - We start with an empty stack and a current node set to the root
    - While current node is not NIL, push it to the stack and set it to its left child
    - While stack not empty
        - Pop a node and visit it
        - Set current node to the right child of the popped node
        - While current node is not NIL, push it to the stack and set it to its left child

Inorder traversal - non-recursive implementation example



- CurrentNode: A (Stack:)
- CurrentNode: NIL (Stack: A G Q)
- Visit Q, currentNode NIL (Stack: A G)
- Visit G, currentNode X (Stack: A)
- CurrentNode: NIL (Stack: A X Y)
- Visit Y, currentNode NIL (Stack: A X)
- Visit X, currentNode J (Stack: A)
- CurrentNode: NIL (Stack: A J K)
- Visit K, currentNode NIL (Stack: A J)
- Visit J, currentNode NIL (Stack: A)
- Visit A, currentNode T (Stack:)
- CurrentNode: NIL (Stack: T N)
- ...

Inorder traversal - non-recursive implementation

```
subalgorithm inorder(tree) is:
//pre: tree is a BinaryTree
    s: Stack //s is an auxiliary stack
    currentNode ← tree.root
    while currentNode ≠ NIL execute
        push(s, currentNode)
        currentNode ← [currentNode].left
    end-while
    while not isEmpty(s) execute
        currentNode ← pop(s)
        @visit currentNode
        currentNode ← [currentNode].right
        while currentNode ≠ NIL execute
            push(s, currentNode)
            currentNode ← [currentNode].left
        end-while
    end-while
end-subalgorithm
```

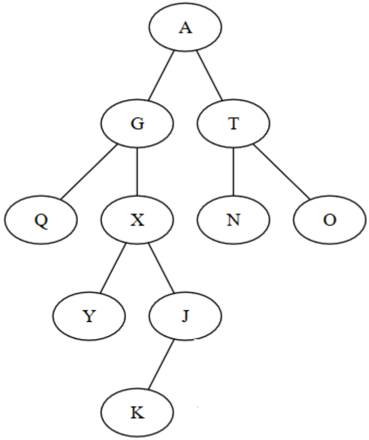
Inorder traversal - non-recursive implementation

- Time complexity $\Theta(n)$, extra space complexity $O(n)$

Postorder traversal

- In case of *postorder* traversal:
    - Traverse the left subtree - if exists
    - Traverse the right subtree - if exists
    - Visit the *root* of the tree
- When traversing the subtrees (left or right) the same postorder traversal is applied (so, from the left subtree we traverse the left subtree, then traverse the right subtree and then visit the root).

Postorder traversal example



- Postorder traversal: Q, Y, K, J, X, G, N, O, T, A

Postorder traversal - recursive implementation

- The simplest implementation for postorder traversal is with a recursive algorithm.

```
subalgorithm postorder_recursive(node) is:
//pre: node is a ↑ BTNode
  if node ≠ NIL then
    postorder_recursive([node].left)
    postorder_recursive([node].right)
    @visit [node].info
  end-if
end-subalgorithm
```

- We need again a wrapper subalgorithm to perform the first call to *postorder_recursive* with the root of the tree as parameter.
- The traversal takes $\Theta(n)$ time for a tree with n nodes.

Postorder traversal - non-recursive implementation

- We can implement the postorder traversal without recursion, but it is slightly more complicated than preorder and inorder traversals.
- We can have an implementation that uses two stacks and there is also an implementation that uses one stack.

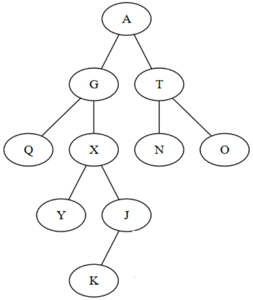
Postorder traversal with two stacks

- The main idea of postorder traversal with two stacks is to build the reverse of postorder traversal in one stack. If we have this, popping the elements from the stack until it becomes empty will give us postorder traversal.
- Building the reverse of postorder traversal is similar to building preorder traversal, except that we need to traverse the right subtree first (not the left one). The other stack will be used for this.
- The algorithm is similar to *preorder* traversal, with two modifications:
    - When a node is removed from the stack, it is added to the second stack (instead of being visited)
    - For a node taken from the stack we first push the left child and then the right child to the stack.

Postorder traversal with one stack

- We start with an empty stack and a current node set to the root of the tree
- While the current node is not NIL, push to the stack the right child of the current node (if exists) and the current node and then set the current node to its left child.
- While the stack is not empty
    - Pop a node from the stack (call it current node)
    - If the current node has a right child, the stack is not empty and contains the right child on top of it, pop the right child, push the current node, and set current node to the right child.
    - Otherwise, visit the current node and set it to NIL
    - While the current node is not NIL, push to the stack the right child of the current node (if exists) and the current node and then set the current node to its left child.

Postorder traversal - non-recursive implementation example



- Node: A (Stack:)
- Node: NIL (Stack: T A X G Q)
- Visit Q, Node NIL (Stack: T A X G)
- Node: X (Stack: T A G)
- Node: NIL (Stack: T A G J X Y)
- Visit Y, Node: NIL (Stack: T A G J X)
- Node: J (Stack: T A G X)
- Node: NIL (Stack: T A G X J K)
- Visit K, Node: NIL (Stack: T A G X J)
- Visit J, Node: NIL (Stack: T A G X)
- Visit X, Node: NIL (Stack: T A G)
- Visit G, Node: NIL (Stack: T A)
- Node: T (Stack: A)
- Node: NIL (Stack: A O T N)
- ...

Postorder traversal - non-recursive implementation

```
**subalgorithm** postorder(tree) **is:**
//pre: tree is a BinaryTree
    s: Stack //s is an auxiliary stack
    node ← tree.root
    **while** node $\neq$ NIL **execute**
        **if** [node].right $\neq$ NIL **then**
            push(s, [node].right)
        **end-if**
        push(s, node)
        node ← [node].left
    **end-while**
    **while** not isEmpty(s) **execute**
        node ← pop(s)
        **if** [node].right $\neq$ NIL and (not isEmpty(s)) and [node].right = top(s) **th**
            pop(s)
            push(s, node)
            node ← [node].right
//continued on the next slide
```

Postorder traversal - non-recursive implementation

```
        **else**
            @visit node
            node $\leftarrow$ NIL
        **end-if**
        **while** node $\neq$ NIL **execute**
            **if** [node].right $\neq$ NIL **then**
                push(s, [node].right)
            **end-if**
            push(s, node)
            node $\leftarrow$ [node].left
        **end-while**
    **end-while**
**end-subalgorithm**
```

- Time complexity $\Theta(n)$, extra space complexity $O(n)$

Binary tree iterator

- The interface of the binary tree contains the *iterator* operation, which should return an iterator.
- This operation receives a parameter that specifies what kind of traversal we want to do with the iterator (preorder, inorder, postorder, level order)
- The traversal algorithms discussed so far, traverse all the elements of the binary tree at once, but an iterator has to do element-by-element traversal.
- For defining an iterator, we have to divide the code into the functions of an iterator: *init*, *getCurrent*, *next*, *valid*

Inorder binary tree iterator

- Assume an implementation without a parent node.
- What fields do we need to keep in the iterator structure?

```
InorderIterator:
    bt: BinaryTree
    s: Stack
    currentNode: ↑ BTNode
```

Inorder binary tree iterator - init

- What should the *init* operation do?

```
subalgorithm init (it, bt) is:
//pre: it - is an InorderIterator, bt is a BinaryTree
    it.bt ← bt
    init(it.s)
    node ← bt.root
    while node ≠ NIL execute
        push(it.s, node)
        node ← [node].left
    end-while
    if not isEmpty(it.s) then
        it.currentNode ← top(it.s)
    else
        it.currentNode ← NIL
    end-if
end-subalgorithm
```

Inorder binary tree iterator - getCurrent

- What should the *getCurrent* operation do?

```
**function** getCurrent(it) **is:**
  getCurrent $\leftarrow$ [it.currentNode].info
**end-function**
```

Inorder binary tree iterator - valid

- What should the *valid* operation do?

```
function valid(it) is:
    if it.currentNode = NIL then
        valid ← false
    else
        valid ← true
    end-if
end-function
```

Inorder binary tree iterator - next

- What should the *next* operation do?

```
subalgorithm next(it) is:
    node ← pop(it.s)
    if [node].right $\neq$ NIL then
        node ← [node].right
        while node $\neq$ NIL execute
            push(it.s, node)
            node ← [node].left
        end-while
    end-if
    if not isEmpty(it.s) then
        it.currentNode ← top(it.s)
    else
        it.currentNode ← NIL
    end-if
end-subalgorithm
```

Preorder, Inorder, Postorder

- How to remember the difference between traversals?
  - Left subtree is always traversed before the right subtree.
  - The visiting of the root is what changes:
    - PREorder - visit the root before the left and right
    - INorder - visit the root between the left and right
    - POSTorder - visit the root after the left and right

Think about it

- Assume you have a binary tree, but you do not know how it looks like, but you have the *preorder* and *inorder* traversal of the tree. Give an algorithm for building the tree based on these two traversals.
- For example:
  - Preorder: A B F G H E L M
  - Inorder: B G F H A L E M

Think about it

- Can you rebuild the tree if you have the *postorder* and the *inorder* traversal?
- Can you rebuild the tree if you have the *preorder* and the *postorder* traversal?

Huffman coding

- The *Huffman coding* can be used to encode characters (from an alphabet) using variable length codes.
- In order to reduce the total number of bits needed to encode a message, characters that appear more frequently have shorter codes.
- Since we use variable length code for each character, *no code can be the prefix of any other code* (if we encode letter E with 01 and letter X with 010011, during deconding, when we find a 01, we will not know whether it is E or the beginning of X).

Huffman coding

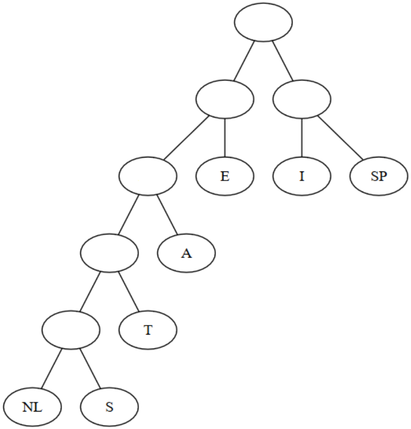
- When building the Huffman encoding for a message, we first have to compute the frequency of every character from the message, because we are going to define the codes based on the frequencies.
- Assume that we have a message with the following letters and frequencies

Character	a	e	i	s	t	space	newline
Frequency	10	15	12	3	4	13	1

Huffman coding

- For defining the Huffman code a binary tree is build in the following way:
    - Start with trees containing only a root node, one for every character. Each tree has a weight, which is the frequency of the character.
    - Get the two trees with the least weight (if there is a tie, choose randomly), combine them into one tree which has as weight the sum of the two weights.
    - Repeat until we get have only one tree.

Huffman coding



Huffman coding

- Code for each character can be read from the tree in the following way: start from the root and go towards the corresponding leaf node. Every time we go left add the bit 0 to encoding and when we go right add bit 1.
- Code for the characters:
  - NL - 00000
  - S - 00001
  - T - 0001
  - A - 001
  - E - 01
  - I - 10
  - SP - 11
- In order to encode a message, just replace each character with the corresponding code

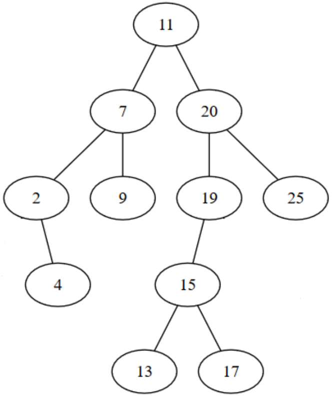
Huffman coding

- Assume we have the following code and we want to decode it: 0110110001000100111001000000
- We do not know where the code of each character ends, but we can use the previously built tree to decode it.
- Start parsing the code and iterate through the tree in the following way:
  - Start from the root
  - If the current bit from the code is 0 go to the left child, otherwise go to the right child
  - If we are at a leaf node we have decoded a character and have to start over from the root
- The decoded message: E I SP T T A SP I E NL

Binary search trees

- A *Binary Search Tree* is a binary tree that satisfies the following property:
    - if x is a node of the binary search tree then:
        - For every node y from the left subtree of x, the information from y is less than or equal to the information from x
        - For every node y from the right subtree of x, the information from y is greater than the information from x
- In order to have a binary search tree, we need to store information in the tree that is of type *TComp*.
- Obviously, the relation used to order the nodes can be considered in an abstract way (instead of having "$\leq$" as in the definition).

Binary Search Tree Example



- If we do an inorder traversal of a binary search tree, we will get the elements in increasing order (according to the relation used)

Binary Search Tree

- The terminology and many properties discussed for binary tree is valid for binary search trees as well:
  - We can have a binary search tree that is full, complete, almost complete, degenerate or balanced
  - The maximum number of nodes in a binary search tree of height N is $2^{N+1} - 1$ - if the tree is complete.
  - The minimum number of nodes in a binary search tree of height N is N - if the tree is degenerate.
  - A binary search tree with N nodes has a height between $log_2 N$ and N (we will denote the height of the tree by h).

Binary Search Tree

- Binary search trees can be used as representation for sorted containers: sorted maps, sorted multimaps, priority queues, sorted sets, etc.
- In order to implement these containers on a binary search tree, we need to define the following basic operations:
    - search for an element
    - insert an element
    - remove an element
- Other operations that can be implemented for binary search trees (and can be used by the containers): get minimum/maximum element, find the successor/predecessor of an element.

Binary Search Tree - Representation

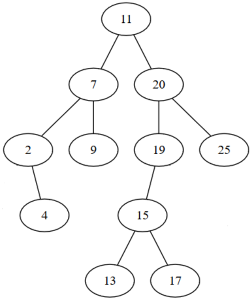
- We will use a linked representation with dynamic allocation (similar to what we used for binary trees)
- We will assume that the info is of type TComp and use the relation "$\leq$"

```
BSTNode:
    info: TComp
    left: ↑ BSTNode
    right: ↑ BSTNode
```

```
BinarySearchTree:
    root: ↑ BSTNode
```

Binary Search Tree - search operation

- How can we search for an element in a binary search tree?



- How can we search for element 15? And for element 14?

BST - search operation - recursive implementation

- How can we implement the *search algorit*hm recursively?

BST - search operation - recursive implementation

- How can we implement the *search algorithm* recursively?

```
function search_rec (node, elem) is:
//pre: node is a BSTNode and elem is the TComp we are searching for
   if node = NIL then
      search_rec ← false
   else
      if [node].info = elem then
         search_rec ← true
      else if [node].info < elem then
         search_rec ← search_rec([node].right, elem)
      else
         search_rec ← search_rec([node].left, elem)
   end-if
end-function
```

BST - search operation - recursive implementation

- Complexity of the search algorithm: $O(h)$ (h being the height of the tree) (which is $O(n)$)
- Since the *search* algorithm takes as parameter a node, we need a wrapper function to call it with the root of the tree.

```
function search (tree, e) is:
//pre: tree is a BinarySearchTree, e is the elem we are looking for
  search ← search_rec(tree.root, e)
end-function
```

BST - search operation - non-recursive implementation

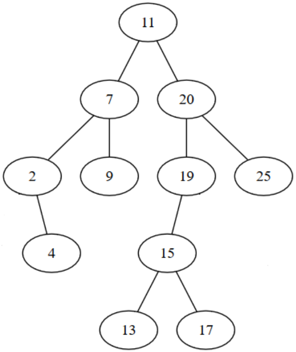
- How can we define the search operation non-recursively?

BST - search operation - non-recursive implementation

- How can we define the search operation non-recursively?

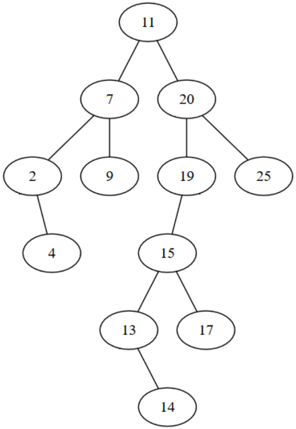
```
**function** search (tree, elem) **is:**
*//pre: tree is a BinarySearchTree and elem is the TComp we are searching for*
    currentNode ← tree.root
    found ← false
    **while** currentNode $\neq$ NIL and not found **execute**
        **if** [currentNode].info = elem **then**
            found ← true
        **else if** [currentNode].info < elem **then**
            currentNode ← [currentNode].right
        **else**
            currentNode ← [currentNode].left
        **end-if**
    **end-while**
    search ← found
**end-function**
```

BST - insert operation



- How/Where can we insert element 14?

BST - insert operation



BST - insert operation - recursive implementation

- How can we implement the *insert* operation?

BST - insert operation - recursive implementation

- How can we implement the *insert* operation?
- We will start with a function that creates a new node given the information to be stored in it.

```
function initNode(e) is:
//pre: e is a TComp
//post: initNode ← a node with e as information
    allocate(node)
    [node].info ← e
    [node].left ← NIL
    [node].right ← NIL
    initNode ← node
end-function
```

BST - insert operation - recursive implementation

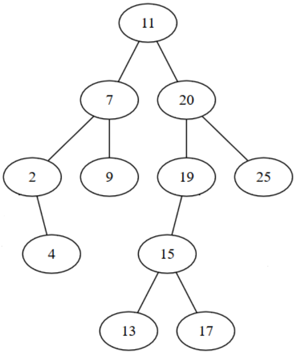
```
function insert_rec(node, e) is:
//pre: node is a BSTNode, e is TComp
//post: a node containing e was added in the tree starting from node
  if node = NIL then
    node ← initNode(e)
  else if [node].info ≥ e then
    [node].left ← insert_rec([node].left, e)
  else
    [node].right ← insert_rec([node].right, e)
  end-if
  insert_rec ← node
end-function
```

- Complexity: O(n)
- Like in case of the *search* operation, we need a wrapper function to call *insert_rec* with the root of the tree.

BST - insert operation - recursive implementation

```
**subalgorithm** insert(tree, e) **is:**
*//pre: tree is a BinarySearchTree, e is TComp*
*//post: a node containing e was added in the tree*
   tree.root ← insert_rec(tree.root, e)
**end-function**
```

BST - Finding the minimum element



- How can we find the minimum element of the binary search tree?

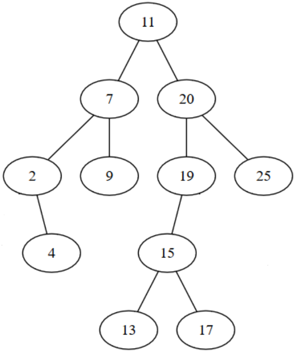
BST - Finding the minimum element

```
function minimum(tree) is:
//pre: tree is a BinarySearchTree
//post: minimum $\leftarrow$ the minimum value from the tree
   currentNode $\leftarrow$ tree.root
   if currentNode = NIL then
      @empty tree, no minimum
   else
      while [currentNode].left $\neq$ NIL execute
         currentNode $\leftarrow$ [currentNode].left
      end-while
      minimum $\leftarrow$ [currentNode].info
   end-if
end-function
```

BST - Finding the minimum element

- Complexity of the minimum operation: $O(n)$
- We can have an implementation for the minimum, when we want to find the minimum element of a subtree, in this case the parameter to the function would be a node, not a tree.
- We can have an implementation where we return the node containing the minimum element, instead of just the value (depends on what we want to do with the operation)
- Maximum element of the tree can be found similarly.

Finding the parent of a node

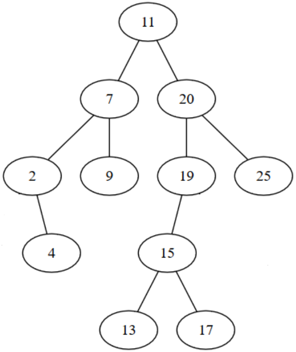


- Given a node, how can we find the parent of the node? (assume a representation where the node has no parent field).

Finding the parent of a node

```
function parent(tree, node) is:
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node $\neq$ NIL
//post: returns the parent of node, or NIL if node is the root
    c $\leftarrow$ tree.root
    if c = node then //node is the root
        parent $\leftarrow$ NIL
    else
        while c $\neq$ NIL and [c].left $\neq$ node and [c].right $\neq$ node execute
            if [c].info $\geq$ [node].info then
                c $\leftarrow$ [c].left
            else
                c $\leftarrow$ [c].right
            end-if
        end-while
        parent $\leftarrow$ c
    end-if
end-function
```

BST - Finding the successor of a node



- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?

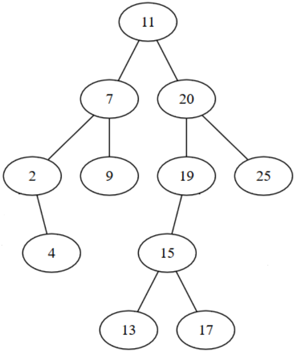
BST - Finding the successor of a node

```
**function** successor(tree, node) **is:**
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node $\neq$ NIL
//post: returns the node with the next value after the value from node
//or NIL if node is the maximum
    **if** [node].right $\neq$ NIL **then**
        c $\leftarrow$ [node].right
        **while** [c].left $\neq$ NIL **execute**
            c $\leftarrow$ [c].left
        **end-while**
        successor $\leftarrow$ c
    **else**
        p $\leftarrow$ parent(tree, c)
        **while** p $\neq$ NIL **and** [p].left $\neq$ c **execute**
            c $\leftarrow$ p
            p $\leftarrow$ parent(tree, p)
        **end-while**
        successor $\leftarrow$ p
    **end-if**
```

BST - Finding the successor of a node

- Complexity of successor: depends on parent function:
    - If *parent* is $\Theta(1)$, complexity of successor is $O(n)$
    - If *parent* is $O(n)$, complexity of successor is $O(n^2)$
- What if, instead of receiving a node, successor algorithm receives as parameter a value from a node (assume unique values in the nodes)? How can we find the successor then?
- Similar to successor, we can define a predecessor function as well.

BST - Remove a node



- How can we remove the value 25? And value 2? And value 11?

BST - Remove a node

- When we want to remove a value (a node containing the value) from a binary search tree we have three cases:
  - The node to be removed has no descendant
    - Set the corresponding child of the parent to NIL
  - The node to be removed has one descendant
    - Set the corresponding child of the parent to the descendant
  - The node to be removed has two descendants
    - Find the maximum of the left subtree, move it to the node to be deleted, and delete the maximum
      OR
    - Find the minimum of the right subtree, move it to the node to be deleted, and delete the minimum

BST - Remove a node

```
function removeRec(node, elem) is
//pre: node is a pointer to a BSTreeNode and elem is the value we remove
//post: the node with value elem was removed from the (sub)tree that starts
//with node
    if node = NIL then
        removeRec ← NIL
    else if [node].info > elem then
        [node].left ← removeRec([node].left, elem)
        removeRec ← node
    else if [node].info < elem then
        [node].right ← removeRec([node].right, elem)
        removeRec ← node
     else //[node].info = elem, we want to remove node
//continued on the next slide...
```

```
      **if** [node].left = NIL **and** [node].right = NIL **then**
         removeRec ← NIL
      **else if** [node].left = NIL **then**
         removeRec ← [node].right
      **else if** [node].right = NIL **then**
         removeRec ← [node].left
      **else**
         min ← minimum([node].right)
         [node].info ← [min].info
         [node].right ← removeRec([node].right, [min].info)
         removeRec ← node
      **end-if**
   **end-if**
**end-function**
```

- We assume that *minimum* returns a node with the minimum, not just the value.
- Complexity: $O(n)$

Binary Search Tree

- Think about it:
    - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?

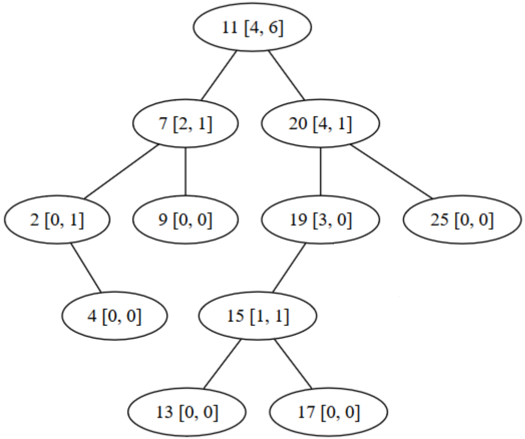
Binary Search Tree

- Think about it:
    - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
    - Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?

Binary Search Tree

- Think about it:
    - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
    - Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?
    - Hint: Keep in each node the number of nodes in the left subtree (and maybe the number of nodes in the right subtree).

Binary Search Tree



- Obviously, these values have to be modified when we

Binary Search Tree

- Think about it:
  - Having the postorder traversal for a binary search tree, could you build the tree?
  - What if you had the preorder, or the inorder traversal?

Balanced Binary Search Trees

- Specific operations for binary trees run in $O(h)$ time, which can be $O(n)$ in worst case
- Best case for height is a balanced tree, where height of the tree is $O(log_2 n)$

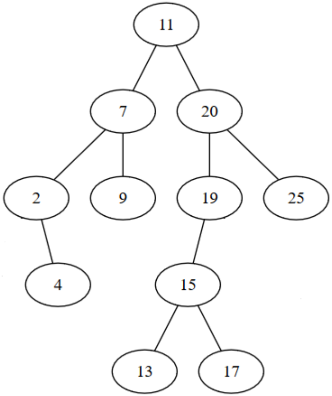
Balanced Binary Search Trees

- Specific operations for binary trees run in $O(h)$ time, which can be $O(n)$ in worst case
- Best case for height is a balanced tree, where height of the tree is $O(log_2 n)$
- To reduce the complexity of algorithms, we want to keep the tree balanced. In order to do this, we want every node to be balanced.
- When a node loses its balance, we will perform some operations (called rotations) to make it balanced again.

AVL Trees

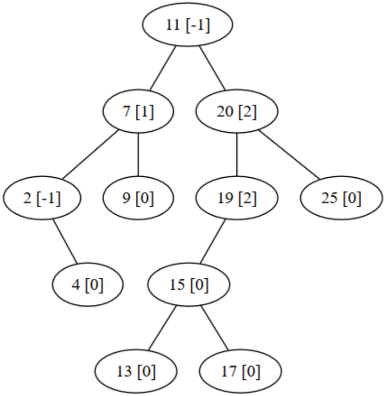
- Definition: An AVL (Adelson-Velskii Landis) tree is a binary tree which satisfies the following property (AVL tree property):
  - If x is a node of the AVL tree:
    - the difference between the height of the left and right subtree of x is 0, 1 or -1 (balancing information)
- Observations:
- Height of an empty tree is -1
- Height of a single node is 0

AVL Trees



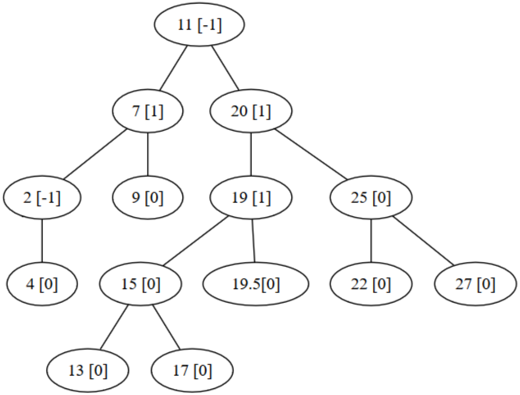
- Is this an AVL tree?

AVL Trees



- Values in square brackets show the balancing information of a node. The tree is not an AVL tree, because the balancing information for nodes 19 and 20 is 2.

AVL Trees



- This is an AVL tree.

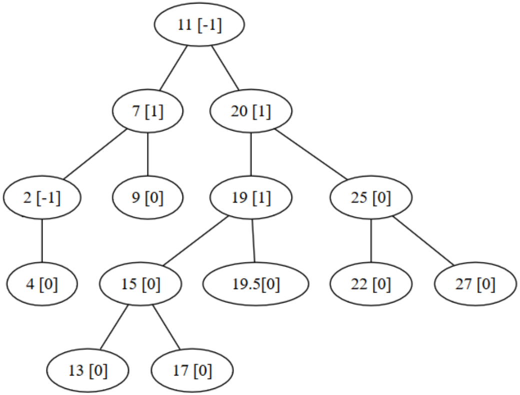
AVL Trees - rotations

- Adding or removing a node might result in a binary tree that violates the AVL tree property.
- In such cases, the property has to be restored and only after the property holds again is the operation (add or remove) considered finished.
- The AVL tree property can be restored with operations called **rotations**.

AVL Trees - rotations

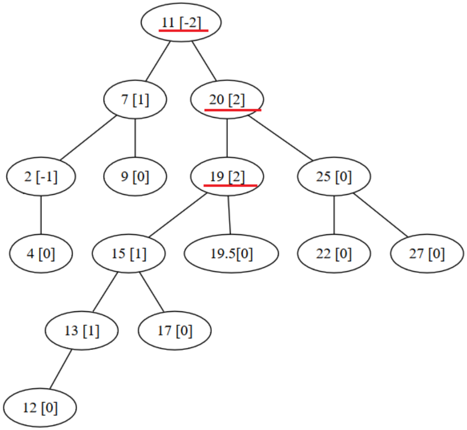
- After an insertion, only the nodes on the path to the modified node can change their height.
- We check the balancing information on the path from the modified node to the root. When we find a node that does not respect the AVL tree property, we perform a suitable *rotation* to rebalance the (sub)tree.

AVL Tress - rotations



- What if we insert element 12?

AVL Trees - rotations

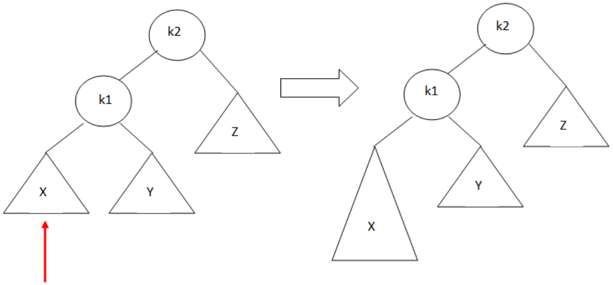


- Red lines show the unbalanced nodes. We will rebalance node 19.

AVL Trees - rotations

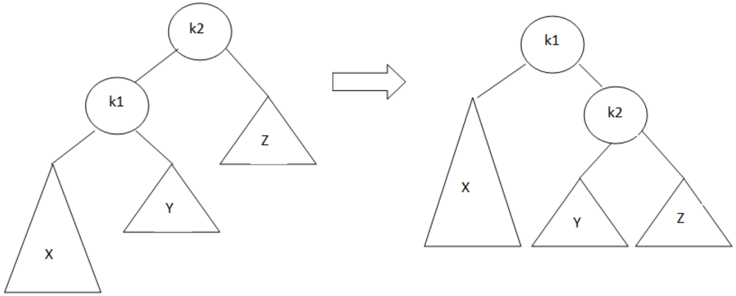
- Assume that at a given point α is the node that needs to be rebalanced.
- Since α was balanced before the insertion, and is not after the insertion, we can identify four cases in which a violation might occur:
  - Insertion into the left subtree of the left child of α
  - Insertion into the right subtree of the left child of α
  - Insertion into the left subtree of the right child of α
  - Insertion into the right subtree of the right child of α

AVL Trees - rotations - case 1

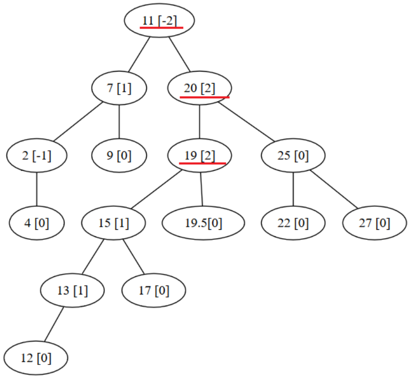


- Solution: single rotation to right

AVL Trees - rotation - Single Rotation to Right

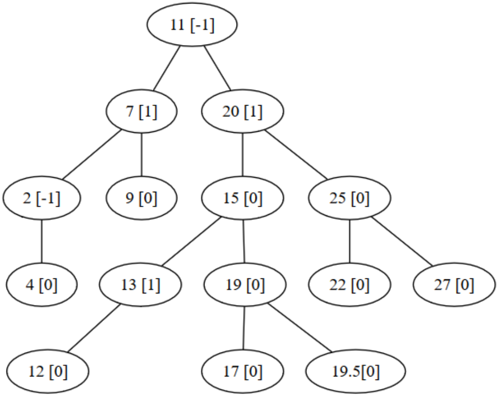


AVL Trees - rotations - case 1 example

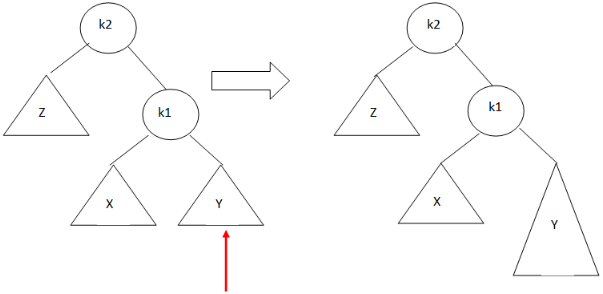


- Node 19 is imbalanced, because we inserted a new node (12) in the left subtree of the left child.
- Solution: **single rotation to right**

AVL Trees - rotation - case 1 example

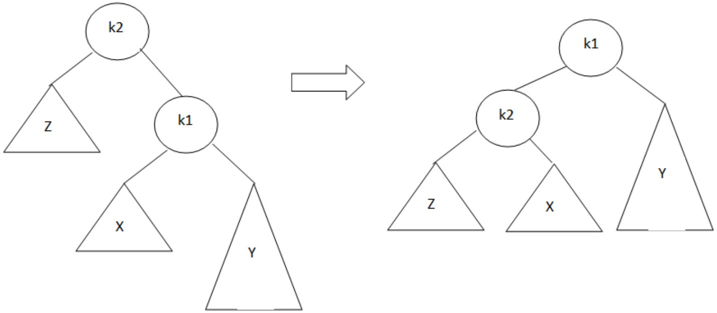


AVL Trees - rotations - case 4

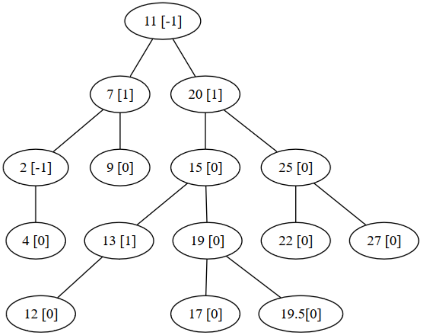


- Solution: **single rotation to left**

AVL Trees - rotation - Single Rotation to Left

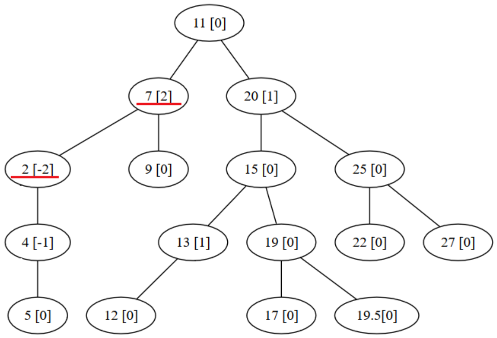


AVL Trees - rotations - case 4 example



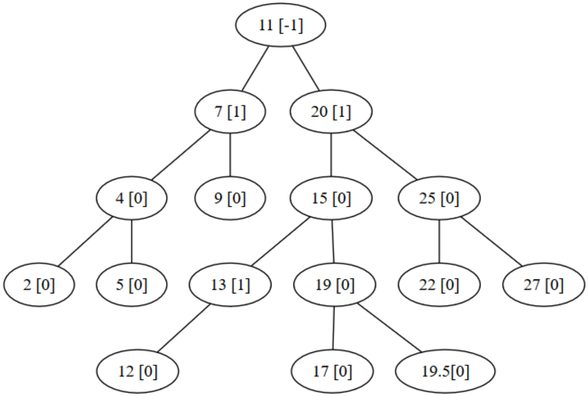
- Insert value 5

AVL Trees - rotations - case 4 example



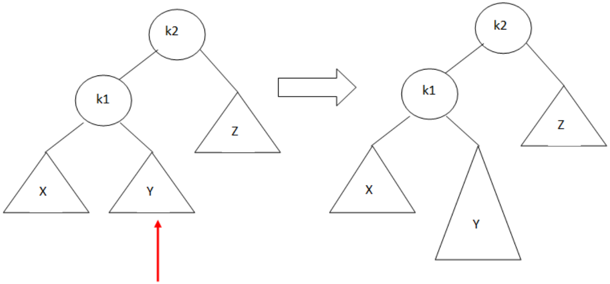
- Node 2 is imbalanced, because we inserted a new node (5) to the right subtree of the right child
- Solution: **single rotation to left**

AVL Trees - rotation - case 4 example



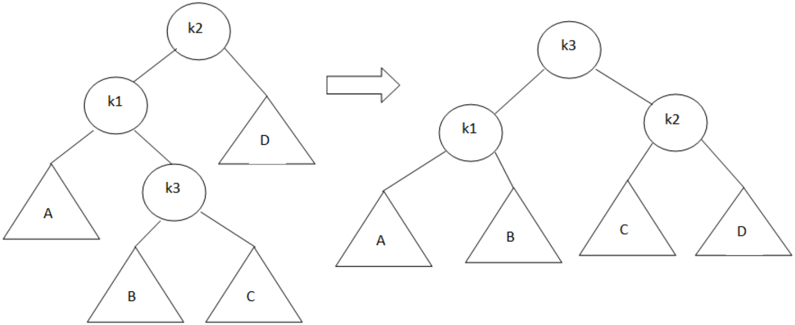
- After the rotation

AVL Trees - rotations - case 2

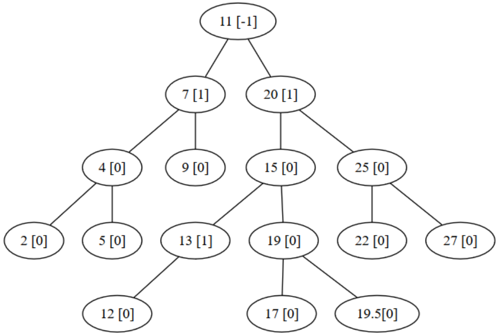


- Solution: **Double rotation to right**

AVL Trees - rotation - Double Rotation to Right

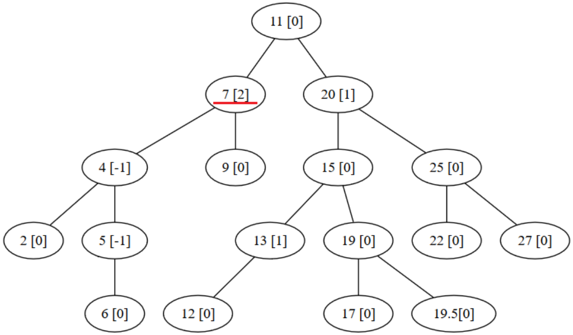


AVL Trees - rotations - case 2 example



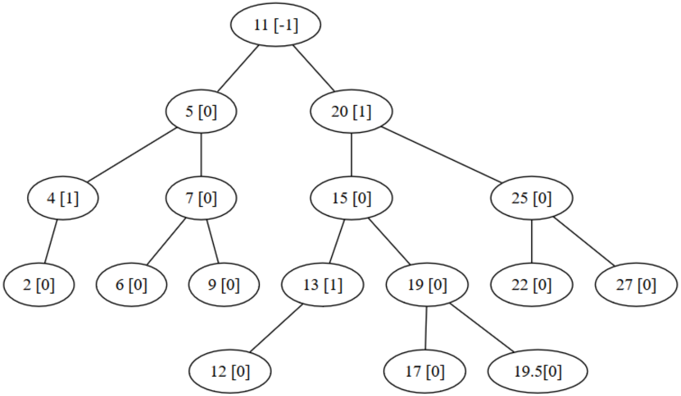
- Insert value 6

AVL Trees - rotations - case 2 example



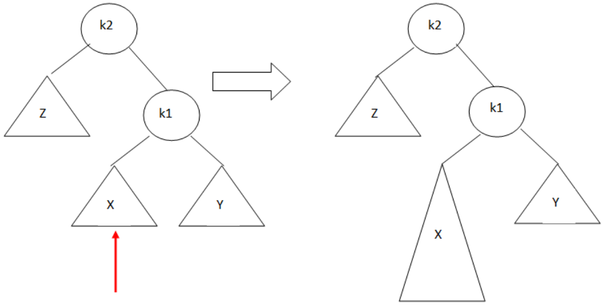
- Node 7 is imbalanced, because we inserted a new node (6) to the right subtree of the left child
- Solution: **double rotation to right**

AVL Trees - rotation - case 2 example



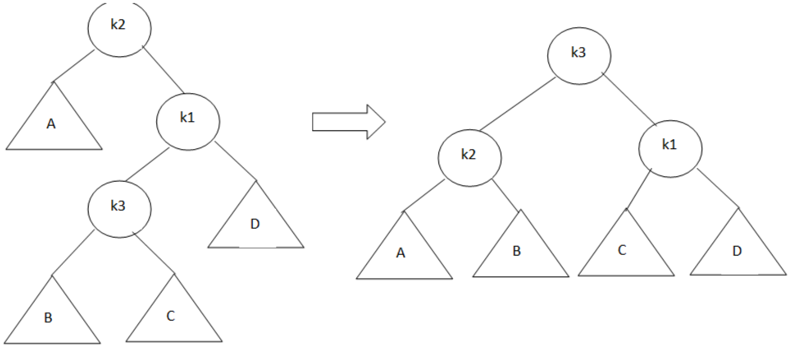
- After the rotation

AVL Trees - rotations - case 3

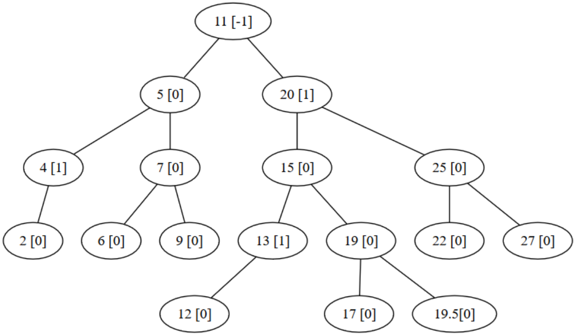


- Solution: **Double rotation to left**

AVL Trees - rotation - Double Rotation to Left

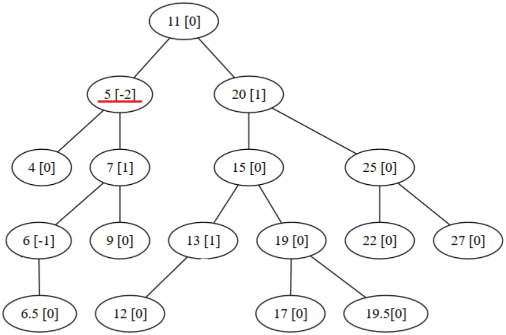


AVL Trees - rotations - case 3 example



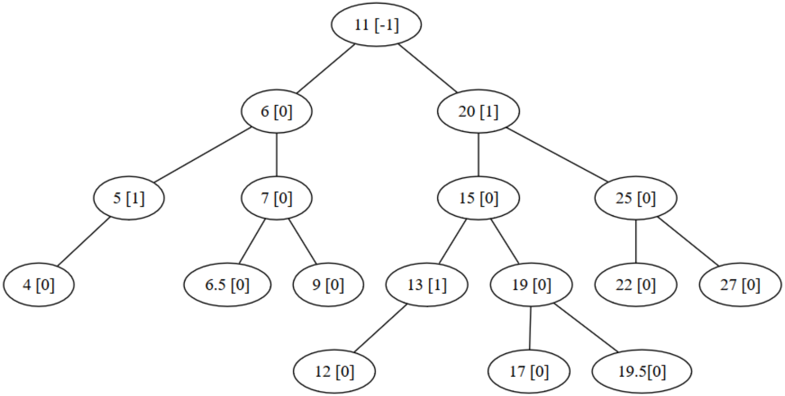
- Remove node with value 2 and insert value 6.5

AVL Trees - rotations - case 3 example



- Node 5 is imbalanced, because we inserted a new node (6.5) to the left subtree of the right child
- Solution: **double rotation to left**

AVL Trees - rotation - case 3 example

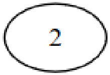


- After the rotation

AVL rotations example I

- Start with an empty AVL tree
- Insert 2

AVL rotations example II

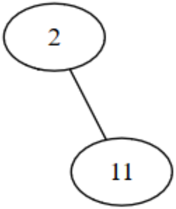


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example III

- No rotation is needed
- Insert 11

AVL rotations example IV

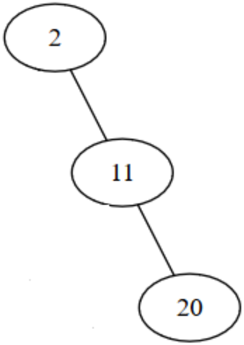


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example V

- No rotation is needed
- Insert 20

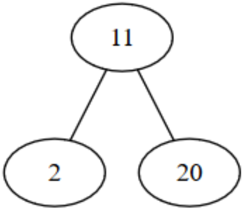
AVL rotations example VI



- Do we need a rotation?
- If yes, on which node and what type of rotation?

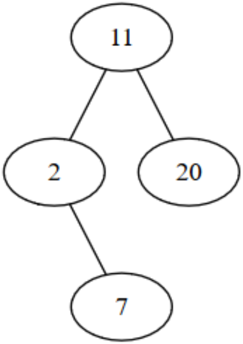
AVL rotations example VII

- Yes, we need a single left rotation on node 2
- After the rotation:



- Insert 7

AVL rotations example VIII

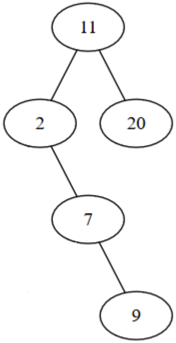


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example IX

- No rotation is needed
- Insert 9

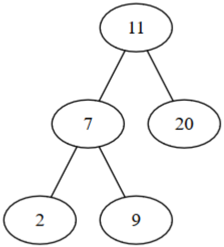
AVL rotations example X



- Do we need a rotation?
- If yes, on which node and what type of rotation?

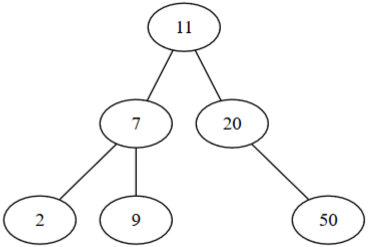
AVL rotations example XI

- Yes, we need a single left rotation on node 2
- After the rotation:



- Insert 50

AVL rotations example XII

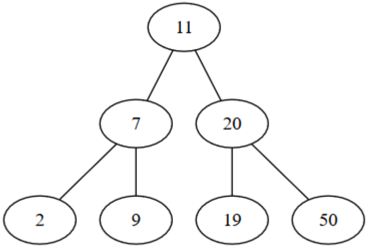


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XIII

- No rotation is needed
- Insert 19

AVL rotations example XIV

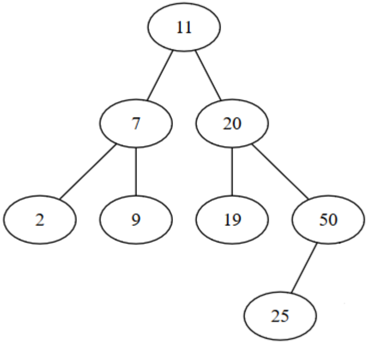


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XV

- No rotation is needed
- Insert 25

AVL rotations example XVI

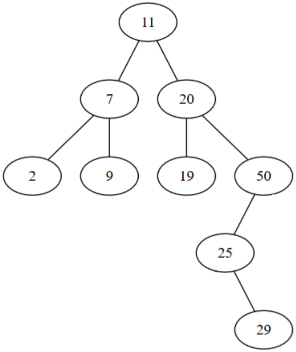


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XVII

- No rotation is needed
- Insert 29

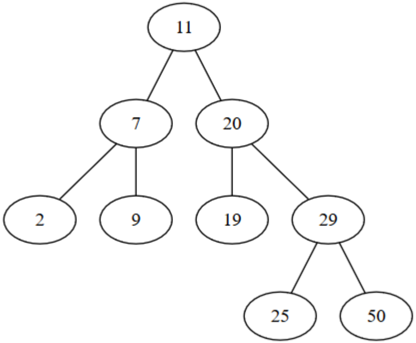
AVL rotations example XVIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

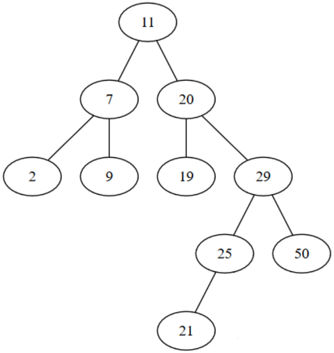
AVL rotations example XIX

- Yes, we need a double right rotation on node 50
- After the rotation



- Insert 21

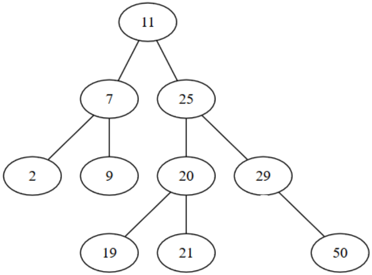
AVL rotations example XX



- Do we need a rotation?
- If yes, on which node and what type of rotation?

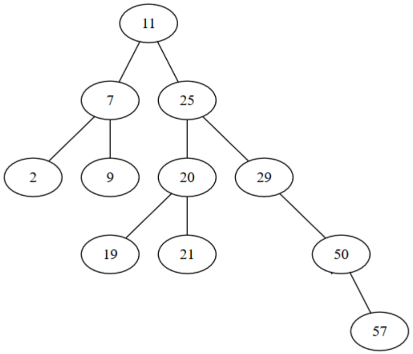
AVL rotations example XXI

- Yes, we need a double left rotation on node 20
- After the rotation



- Insert 57

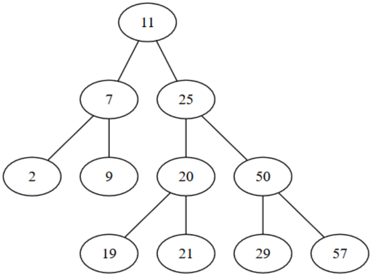
AVL rotations example XXII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

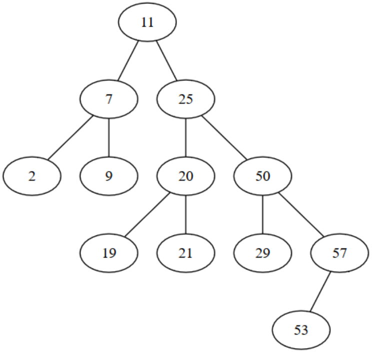
AVL rotations example XXIII

- Yes, we need a single left rotation on node 50
- After the rotation



- Insert 53

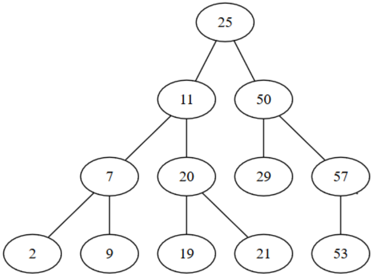
AVL rotations example XXIV



- Do we need a rotation?
- If yes, on which node and what type of rotation?

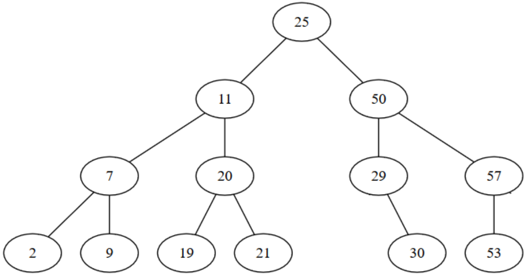
AVL rotations example XXV

- Yes, we need a single left rotation on node 11
- After the rotation



- Insert 30

AVL rotations example XXVI

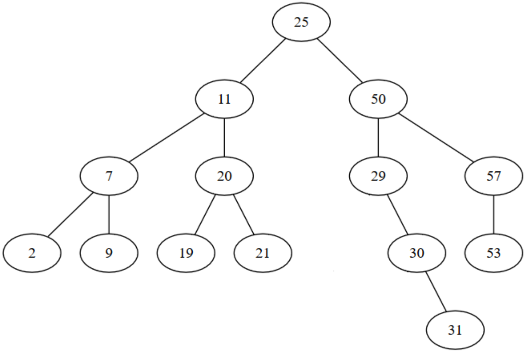


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXVII

- No rotation is needed
- Insert 31

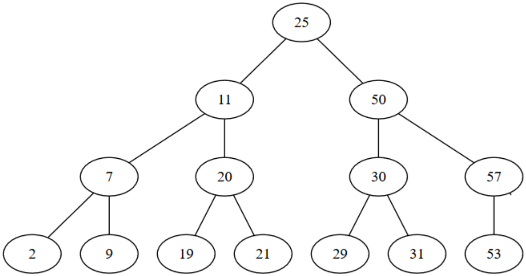
AVL rotations example XXVIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

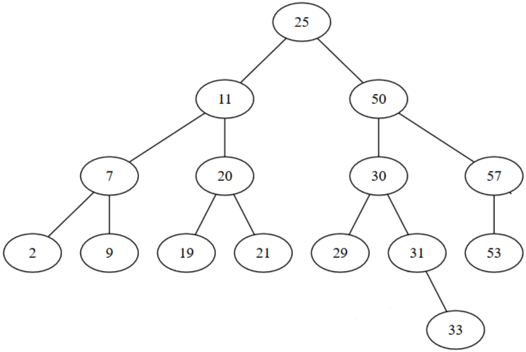
AVL rotations example XXIX

- Yes, we need a single left rotation on node 29
- After the rotation



- Insert 33

AVL rotations example XXX

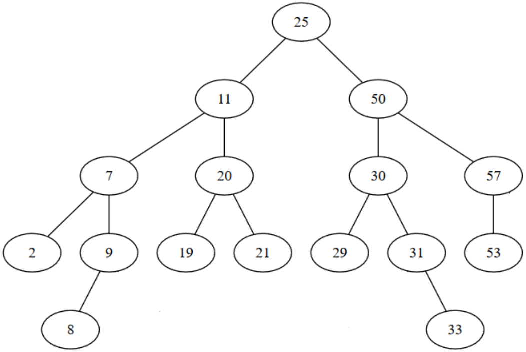


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXI

- No rotation is needed
- Insert 8

AVL rotations example XXXII

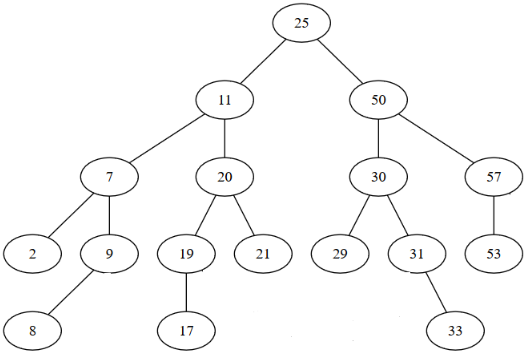


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXIII

- No rotation is needed
- Insert 17

AVL rotations example XXXIV

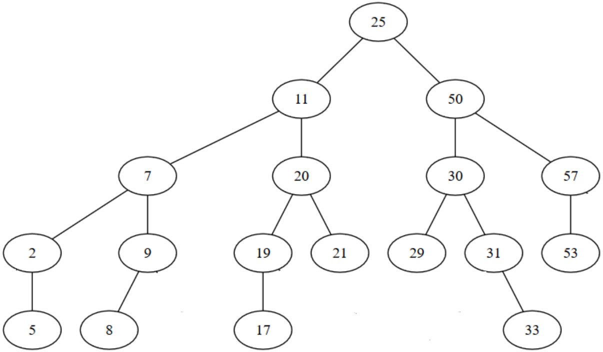


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXV

- No rotation is needed
- Insert 5

AVL rotations example XXXVI

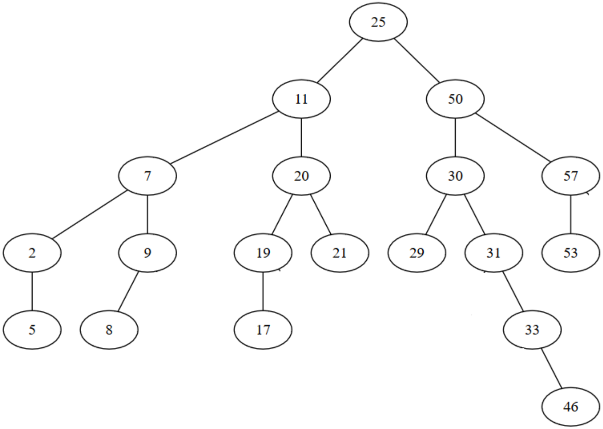


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XXXVII

- No rotation is needed
- Insert 46

AVL rotations example XXXVIII



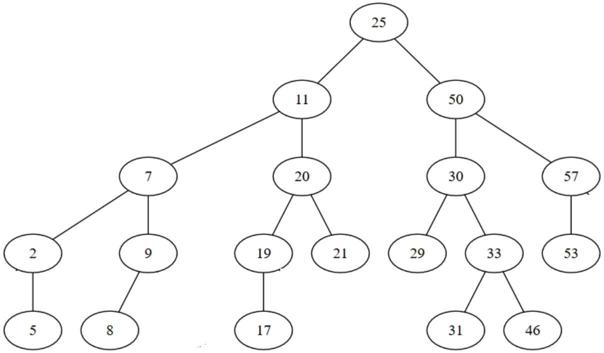
- Do we need a rotation?

AVL rotations example XXXIX

- If yes, on which node and what type of rotation?

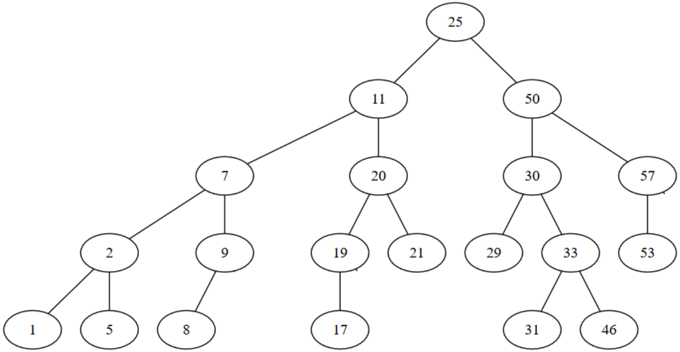
AVL rotations example XL

- Yes, we need a single left rotation on node 31
- After the rotation



- Insert 1

AVL rotations example XLI

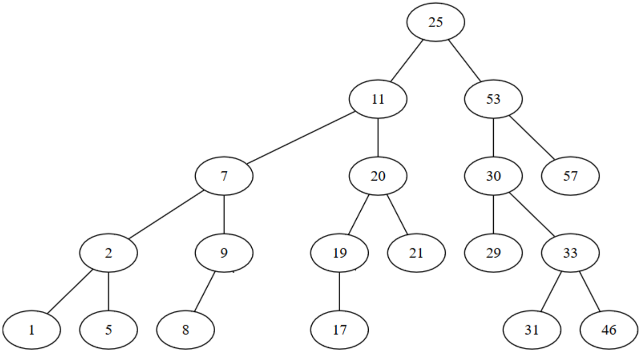


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XLII

- No rotation is needed
- Remove 50

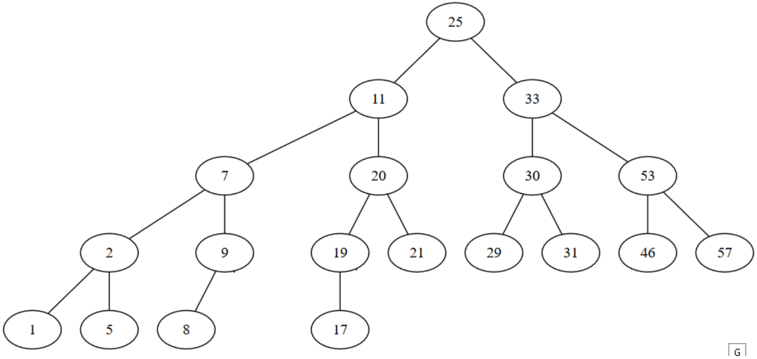
AVL rotations example XLIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

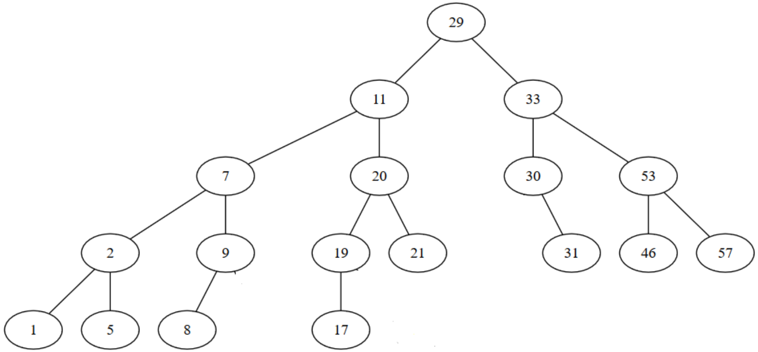
AVL rotations example XLIV

- Yes we need double right rotation on node 53
- After the rotation



- Remove 25

AVL rotations example XLV

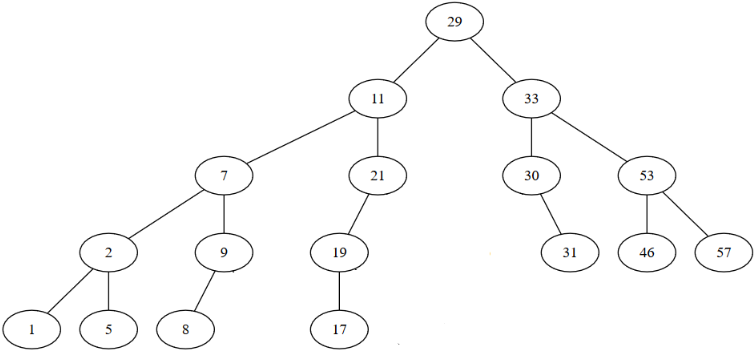


- Do we need a rotation?
- If yes, on which node and what type of rotation?

AVL rotations example XLVI

- No rotation is needed
- Remove 20

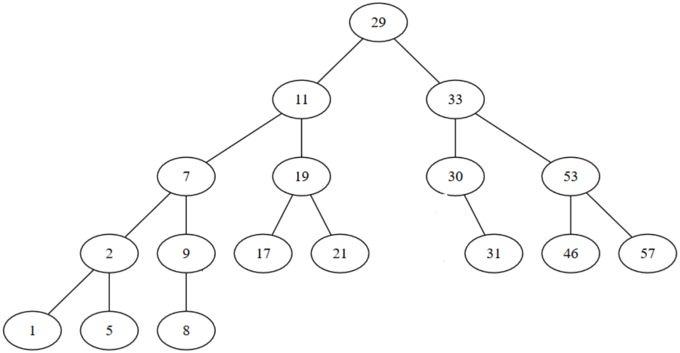
AVL rotations example XLVII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

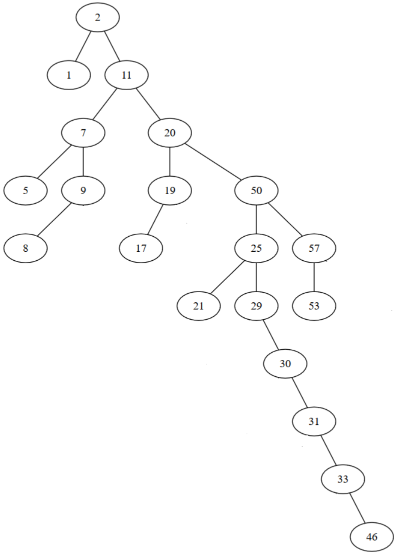
AVL rotations example XLVIII

- Yes, we need a single right rotation on node 21
- After the rotation



Comparison to BST

- If, instead of using an AVL tree, we used a binary search tree, after the insertions the tree would have been:


2
1
11
7
20
5
9
19
50
8
17
25
57
21
29
53
30
31
33
46

AVL Trees - representation

- What structures do we need for an AVL Tree?

AVL Trees - representation

- What structures do we need for an AVL Tree?

```
AVLNode:
    info: TComp //information from the node
    left: ↑ AVLNode //address of left child
    right: ↑ AVLNode //address of right child
    h: Integer //height of the node
```

```
AVLTree:
    root: ↑ AVLNode //root of the tree
```

AVL Tree - implementation

- We will implement the *insert* operation for the AVL Tree.
- We need to implement some operations to make the implementation of *insert* simpler:
    - A subalgorithm that (re)computes the height of a node
    - A subalgorithm that computes the balance factor of a node
    - Four subalgorithms for the four rotation types (we will implement only one)
- And we will assume that we have a function, *createNode* that creates and returns a node containing a given information (left and right are NIL, height is 0).

AVL Tree - height of a node

```
subalgorithm recomputeHeight(node) is:
//pre: node is an ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: if node ≠ NIL, h of node is set
  if node ≠ NIL then
    if [node].left = NIL and [node].right = NIL then
      [node].h ← 0
    else if [node].left = NIL then
      [node].h ← [[node].right].h + 1
    else if [node].right = NIL then
      [node].h ← [[node].left].h + 1
    else
      [node].h ← max ([[node].left].h, [[node].right].h) + 1
    end-if
  end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

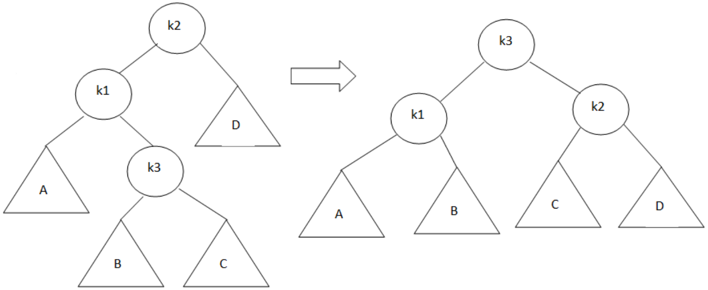
AVL Tree - balance factor of a node

```
function balanceFactor(node) is:
//pre: node is an ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: returns the balance factor of the node
    if [node].left = NIL and [node].right = NIL then
        balanceFactor ← 0
    else if [node].left = NIL then
        balanceFactor ← -1 - [[node].right].h //height of empty tree is -1
    else if [node].right = NIL then
        balanceFactor ← [[node].left].h + 1
    else
        balanceFactor ← [[node].left].h - [[node].right].h
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

AVL Tree - rotations

- Out of the four rotations, we will only implement one, double right rotation (DRR).
- The other three rotations can be implemented similarly (RLR, SRR, SLR).



AVL Tree - DRR

```
function DRR(node) is: //pre: node is an ↑ AVLNode on which we perform
the double right rotation
//post: DRR returns the new root after the rotation
    k2 ← node
    k1 ← [node].left
    k3 ← [k1].right
    k3left ← [k3].left
    k3right ← [k3].right
    //reset the links
    newRoot ← k3
    [newRoot].left ← k1
    [newRoot].right ← k2
    [k1].right ← k3left
    [k2].left ← k3right
//continued on the next slide
```

AVL Tree - DRR

```
//recompute the heights of the modified nodes
    recomputeHeight(k1)
    recomputeHeight(k2)
    recomputeHeight(newRoot)
    DRR ← newRoot
end-function
```

- Complexity: $\Theta(1)$

AVL Tree - insert

```
**function** insertRec(node, elem) **is**
//pre: node is a ↑ AVLNode, elem is the value we insert in the (sub)tree that
//has node as root
//post: insertRec returns the new root of the (sub)tree after the insertion
  **if** node = NIL **then**
    insertRec ← createNode(elem)
  **else if** elem ≤ [node].info **then**
    [node].left ← insertRec([node].left, elem)
  **else**
    [node].right ← insertRec([node].right, elem)
  **end-if**
//continued on the next slide...
```

AVL Tree - insert

```
    recomputeHeight(node)
    balance ← getBalanceFactor(node)
    if balance = -2 then
    //right subtree has larger height, we will need a rotation to the LEFT
        rightBalance ← getBalanceFactor([node].right)
        if rightBalance < 0 then
        //the right subtree of the right subtree has larger height, SRL
            node ← SRL(node)
        else
            node ← DRL(node)
        end-if
//continued on the next slide...
```

AVL Tree - insert

```
    else if balance = 2 then
    //left subtree has larger height, we will need a RIGHT rotation
        leftBalance ← getBalanceFactor([node].left)
        if leftBalance > 0 then
        //the left subtree of the left subtree has larger height, SRR
            node ← SRR(node)
        else
            node ← DRR(node)
        end-if
    end-if
    insertRec ← node
end-function
```

AVL Tree - insert

- Complexity of the *insertRec* algorithm: $O(log_2 n)$
- Since *insertRec* receives as parameter a pointer to a node, we need a wrapper function to do the first call on the root

```
**subalgorithm** insert(tree, elem) **is**
//pre: tree is an AVL Tree, elem is the element to be inserted
//post: elem was inserted to tree
  tree.root ← insertRec(tree.root, elem)
**end-subalgorithm**
```

- remove subalgorithm can be implemented similarly (start from the remove from BST and add the rotation part).

Project presentation

- Project presentations will be held on the 5th of May.
- Project presentation schedule is available online
- Every student has to come to the presentation with his/her own group.
- Do not forget to bring the documentation on paper.
- Be prepared to make modifications to your project (small ones). Failure to perform the modifications will result in a failing grade for the project.
- If you fail your project, the will have to redo it for the retake session.

Written exam

- Will be on the 4th of May, from 16:00 in room 5/I.
- Exam will take 3 hours
- You will need a grade of at least 5 for the written exam to be able to pass this course.