DATA STRUCTURES AND ALGORITHMS

LECTURE 12

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2018 - 2019

In Lecture 11...

- Trees
- Binary Trees

Today

1. Binary trees
2. Huffman coding
3. Binary Search Trees

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 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).

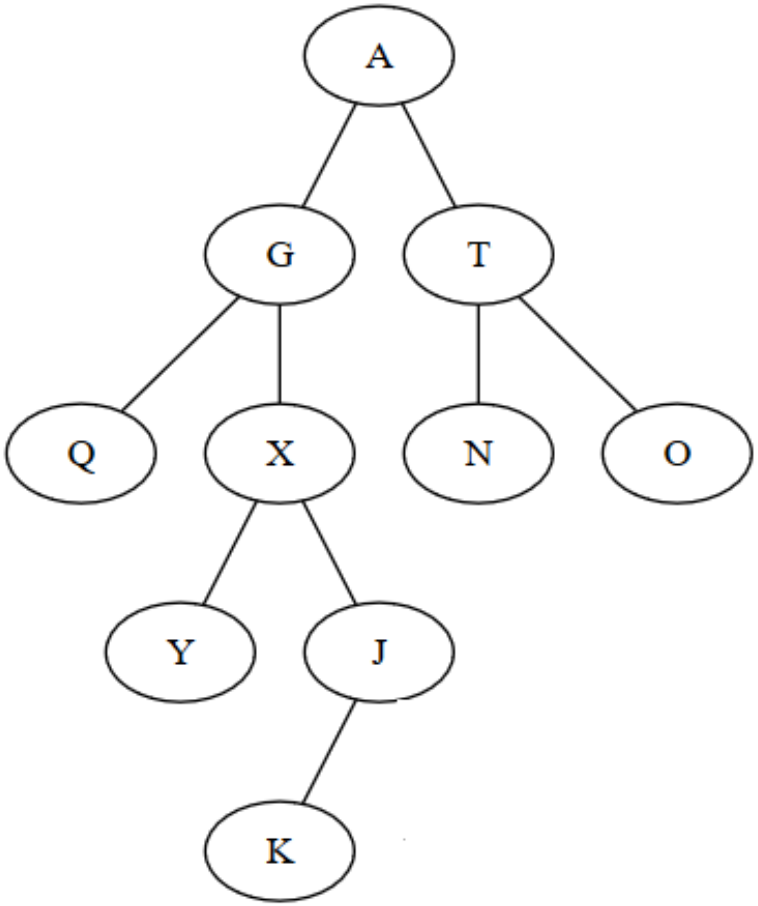
Inorder traversal

- 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).

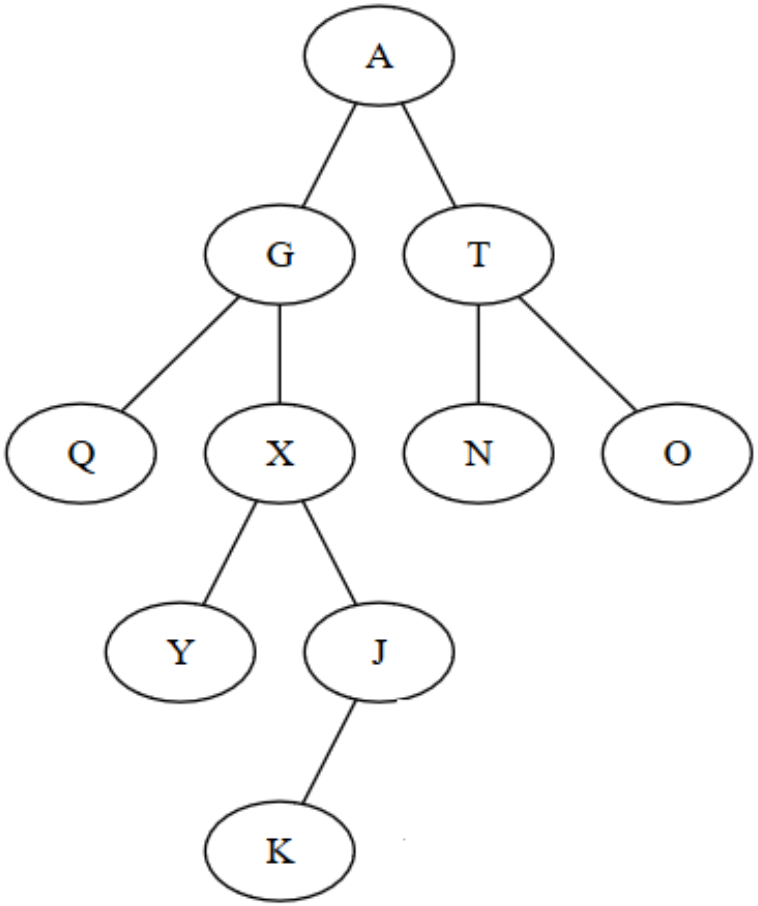
Postorder traversal

- 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 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.

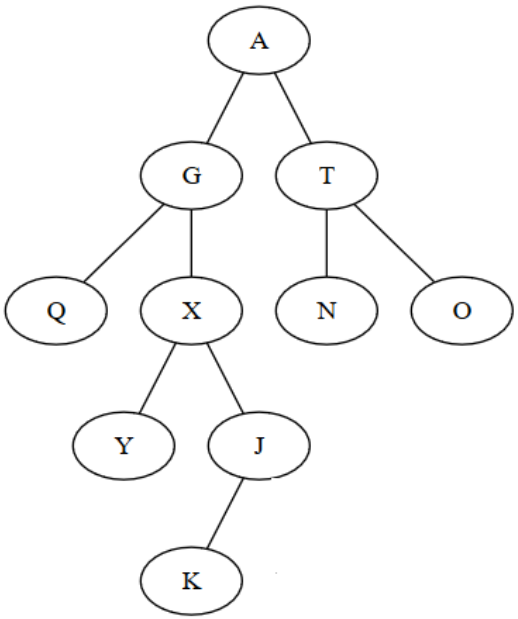
Postorder traversal with two stacks

- 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.

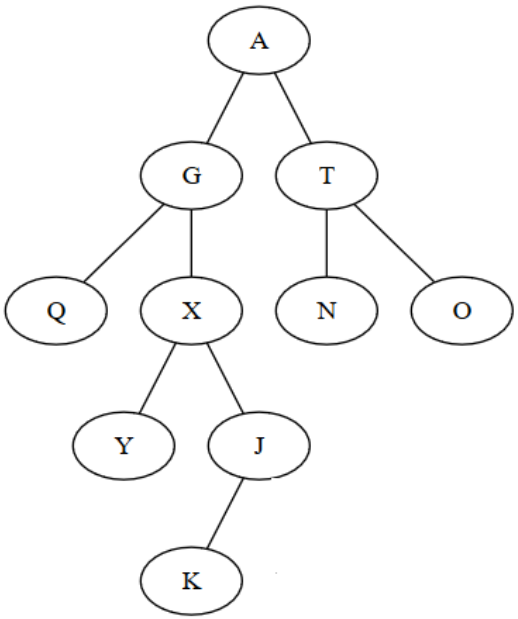
Postorder traversal with one stack

- 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



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 ≠ NIL execute
        if [node].right ≠ 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 ≠ 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 ← NIL
        end-if
        while node ≠ NIL execute
            if [node].right ≠ NIL then
                push(s, [node].right)
            end-if
            push(s, node)
            node ← [node].left
        end-while
    end-while
end-subalgorithm
```

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

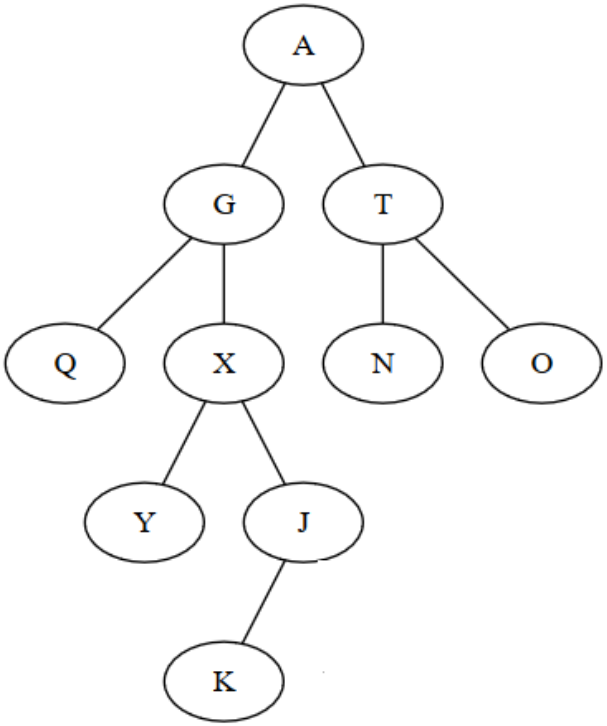
Traversal without a stack

- Preorder, postorder and inorder traversals can be implemented without an auxiliary stack if we use a representation for a node, where we keep a pointer to the parent node and a boolean flag to show whether the current node was visited or not.
- When we start the traversal we assume that all nodes have the *visited* flag set to false.
- During the traversal we set the flags to true, but when traversal is over, we have to make sure that they are set to false again (otherwise a second traversal is not possible).

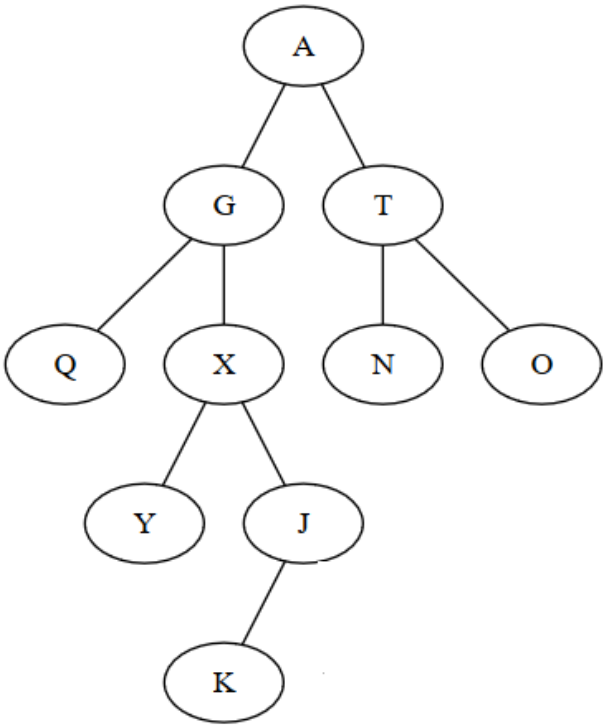
Inorder traversal without a stack

- We take a current node which is set to the root of the tree
- Repeat the following until the node becomes NIL
  - If the node has a left child and it was not visited, the node becomes the left child
  - Otherwise, if the node is not visited, we will visit it
  - Otherwise, if the node has a right child and it was not visited, the node becomes the right child
  - Otherwise, set the children (left and right) of this node back to unvisited and change the node to its parent)

Inorder traversal without a stack - example



Inorder traversal without a stack - example



- Start with node A and go left when possible (Node: Q)
- Visit Q and go right if possible, if not, go up (Node: G)
- Visit G and go right if possible (Node: X)
- Go left as much as possible (Node: Y)
- Visit Y and go right if possible, if not, go up (Node: X)
- Visit X and go right if possible (Node: J)
- Go left as much as possible (Node: K)
- Visit K and go right if possible, if not go up (Node: J)
- Visit J and go right if possible, it not go up and set children of J to non-visited (Node: X)
- Go up and set children of X to non-visited (Node: G)
- ...

Inorder traversal without a stack - implementation

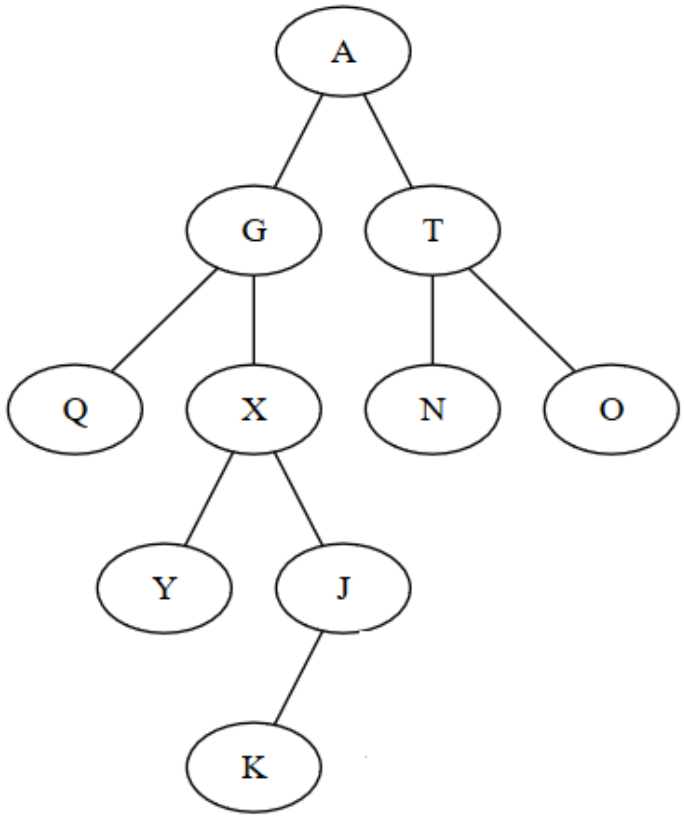
```
subalgorithm inorderNoStack(tree) is:
//pre: tree is a BinaryTree, with nodes containing pointer to parent and visited
    current ← tree.root
    while current ≠ NIL execute
        if [current].left ≠ NIL and [[current].left].visited = false then
            current ← [current].left
        else if [current].visited = false then
            @visit current
            [current].visited ← true
        else if [current].right ≠ NIL and [[current].right].visited = false then
            current ← [current].right
        else
        //we are going up, but before that reset the children to not-visited
            if [current].left ≠ NIL then
                [[current].left].visited ← false
            end-if
//continued on the next slide...
```

Inorder traversal without a stack - implementation

```
        if [current].right ≠ NIL then
            [[current].right].visited ← false
        end-if
        current ← [current].parent
    end-if
  end-while
  if tree.root ≠ NIL then
      [tree.root].visited ← false
  end-if
end-subalgorithm
```

Level order traversal

- In case of level order traversal we first visit the root, then the children of the root, then the children of the children, etc.



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

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 ← [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 ≠ NIL then
        node ← [node].right
        while node ≠ 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

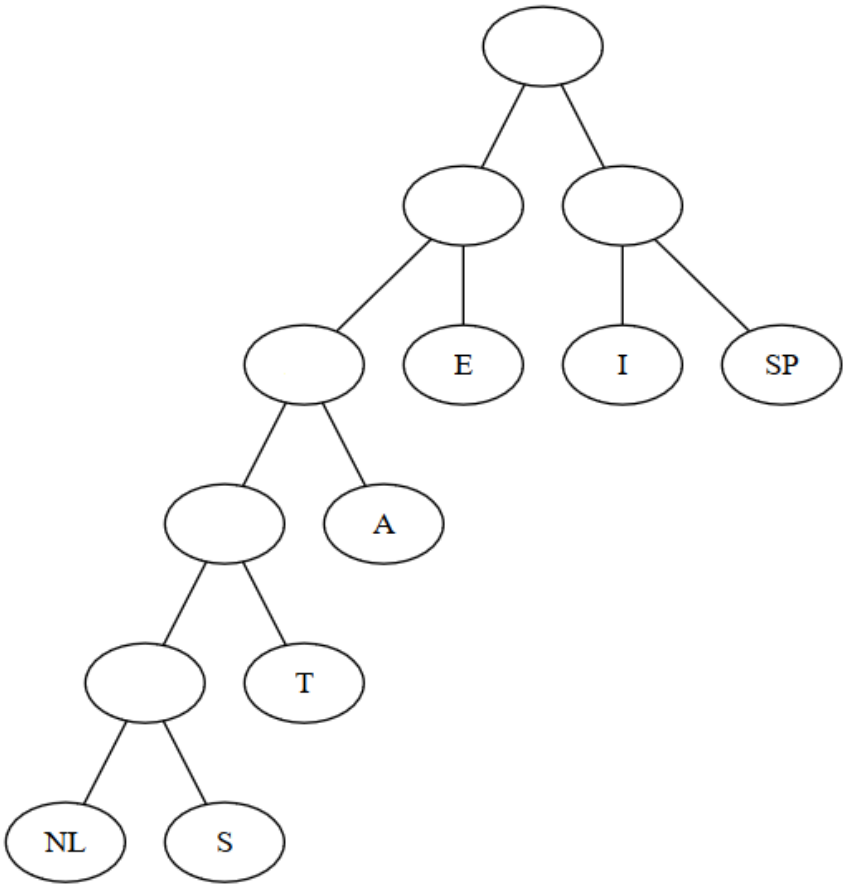
- When building the Huffman encoding for a message, we first have to compute the frequency of every character from the message, because we are going to define the codes based on the frequencies.
- Assume that we have a message with the following letters and frequencies

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 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

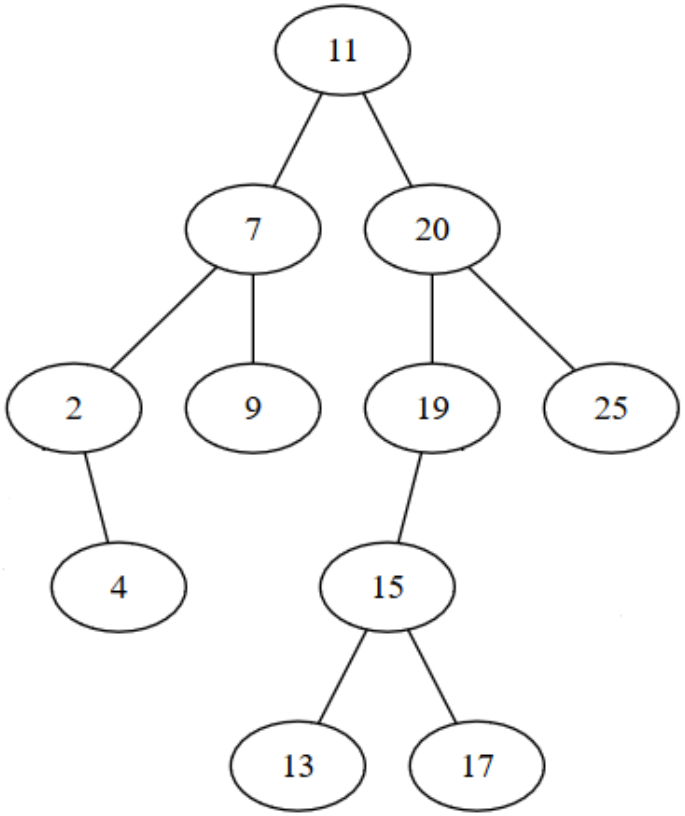
Huffman coding

- Assume we have the following code and we want to decode it: 01101100010001001110010000 0
- 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 or equal to the information from x
- 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

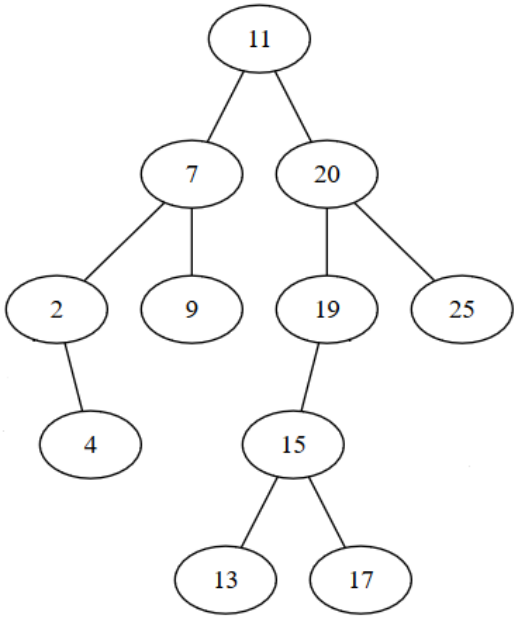
- We will use a linked representation with dynamic allocation (similar to what we used for binary trees)

BSTNode:
info: TElem
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 TElem 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:

BST - search operation - recursive implementation

- Complexity of the search algorithm: $O(h)$ (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 $\leftarrow$ search_rec(tree.root, e)
end-function

BST - search operation - non-recursive implementation

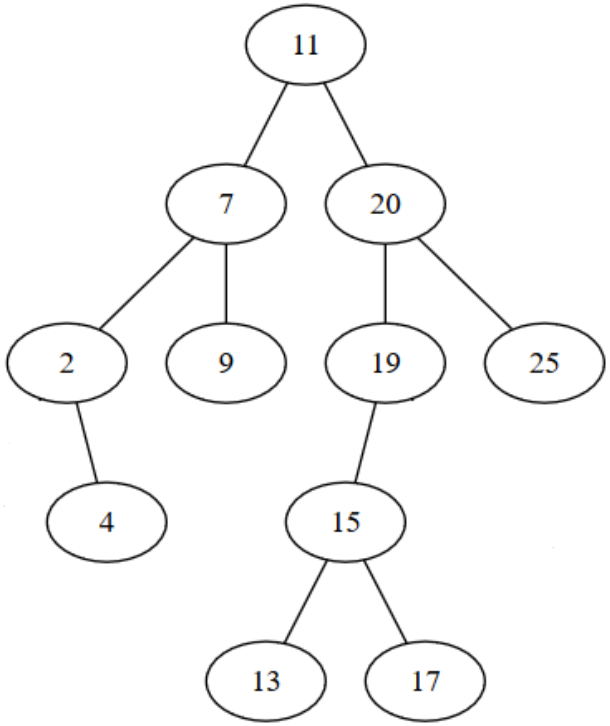
- How can we define the search operation non-recursively?

BST - search operation - non-recursive implementation

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

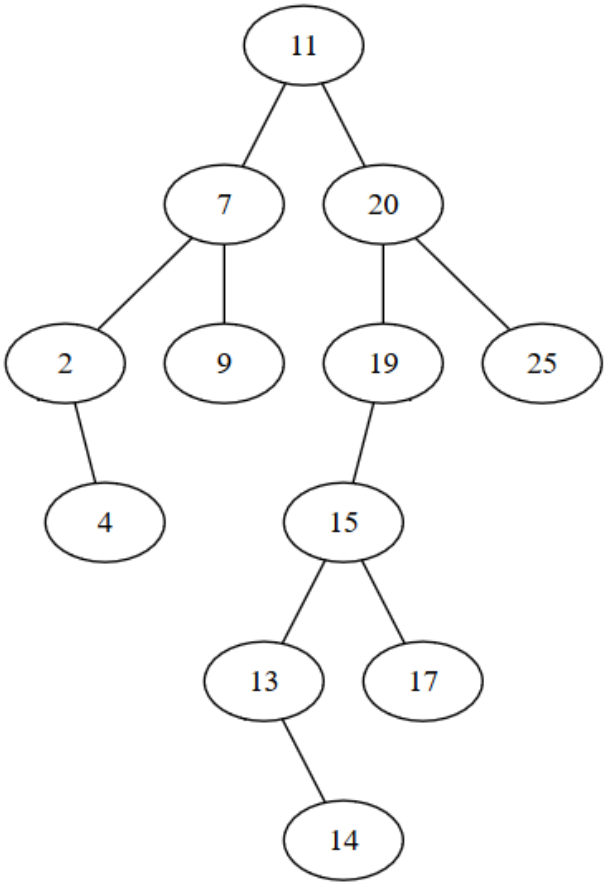
```
function search (tree, elem) is:
//pre: tree is a BinarySearchTree and elem is the TElem we are searching for
    currentNode ← tree.root
    found ← false
    while currentNode ≠ NIL and not found execute
        if [currentNode].info = elem then
            found ← true
        else if [currentNode].info < 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
```

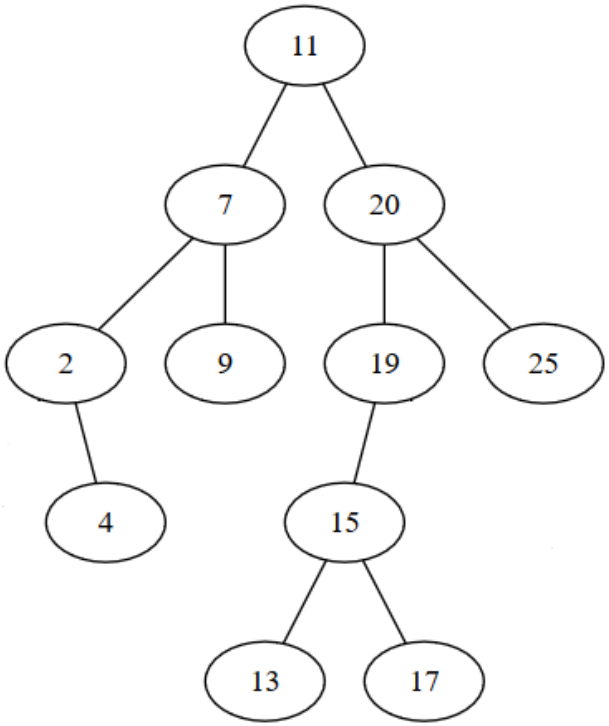
- Complexity:

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 - 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 ← the minimum value from the tree
    currentNode ← tree.root
    if currentNode = NIL then
        @empty tree, no minimum
    else
        while [currentNode].left ≠ NIL execute
            currentNode ← [currentNode].left
        end-while
        minimum ← [currentNode].info
    end-if
end-function
```

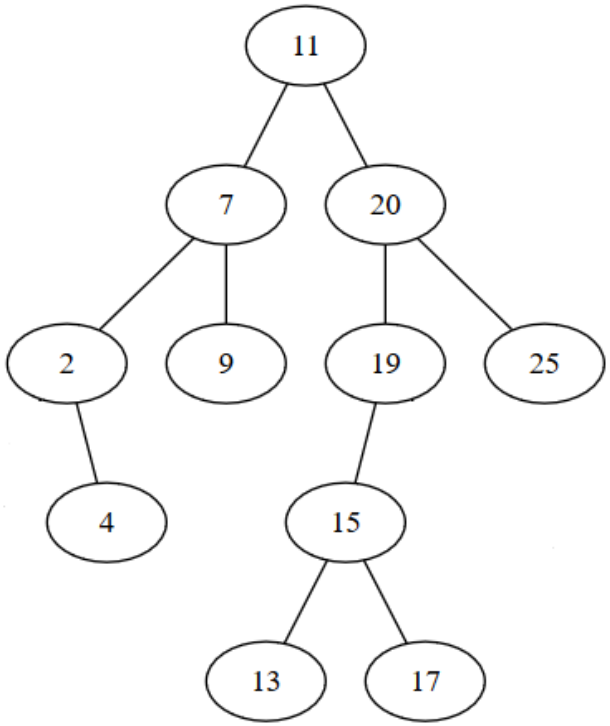
BST - Finding the minimmum element

- Complexity of the minimum operation:

BST - Finding the minimmum 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

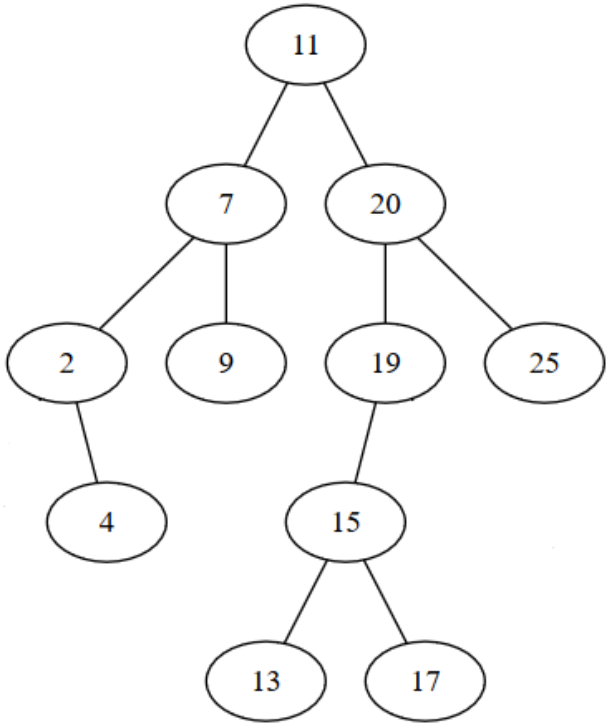
- Complexity:

Finding the parent of a node

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

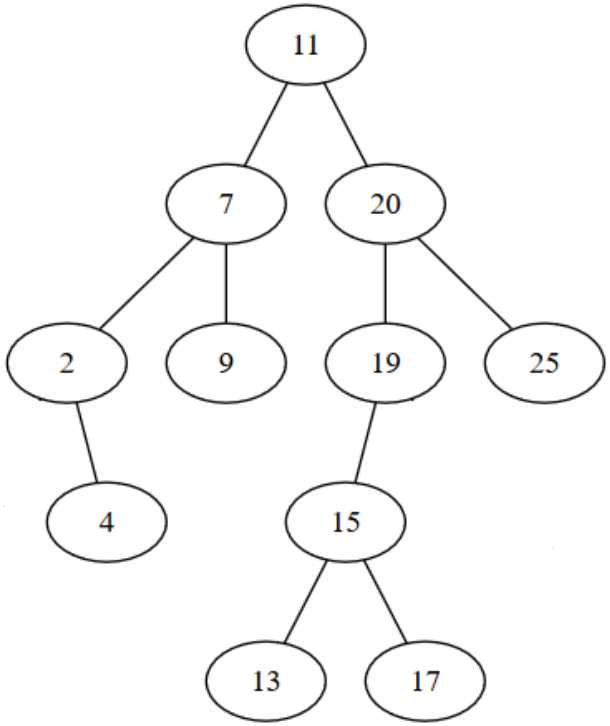
- Complexity: $O(n)$

BST - Finding the successor of a node



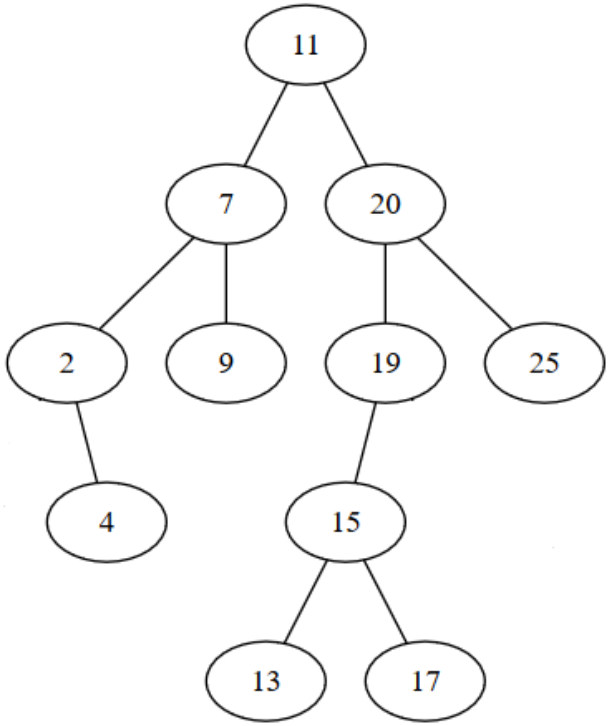
- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11?

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)?
- How can we find the next after 11? After 13?

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)?
- How can we find the next after 11? After 13? After 17?

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 ≠ NIL
//post: returns the node with the next value after the value from node
//or NIL if node is the maximum
    if [node].right ≠ NIL then
        c ← [node].right
        while [c].left ≠ NIL execute
            c ← [c].left
        end-while
        successor ← c
    else
        p ← parent(tree, c)
        while p ≠ NIL and [p].left ≠ c execute
            c ← p
            p ← parent(tree, p)
        end-while
        successor ← p
    end-if
end-function
```

BST - Finding the successor of a node

- Complexity of successor:

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.

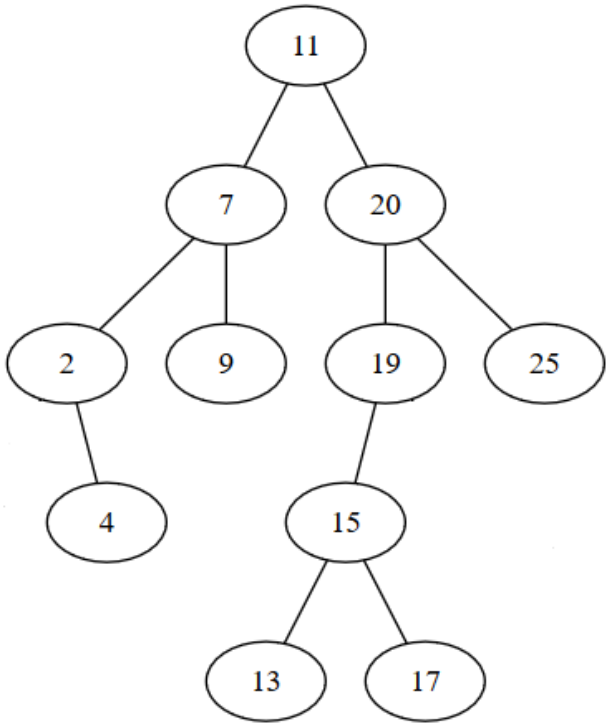
BST - Finding successor of a node

- If we do not have direct access to the parent, finding the successor of a node is $O(n^2)$.
- Can we reduce this complexity?

BST - Finding successor of a node

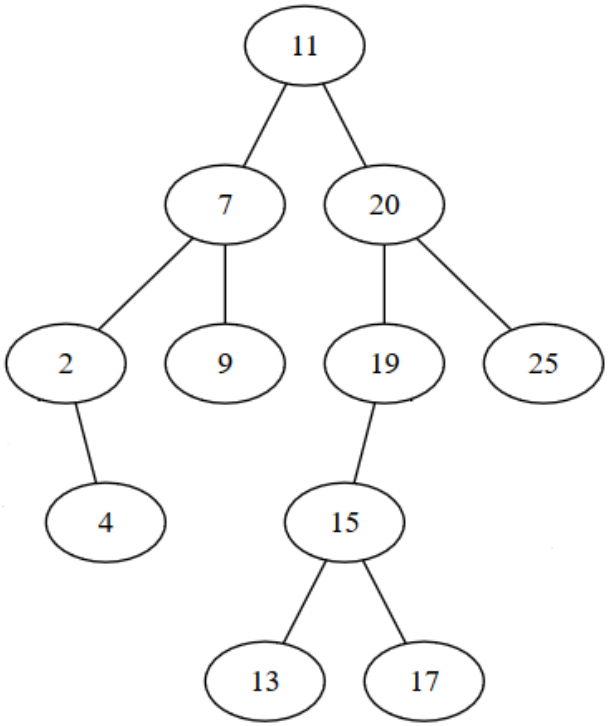
- If we do not have direct access to the parent, finding the successor of a node is $O(n^2)$.
- Can we reduce this complexity?
- $O(n^2)$ is given by the potentially repeated calls for the *parent* function. But we only need the *last* parent, where we went left. We can do one single traversal from the root to our node, and every time we continue left (i.e. current node is greater than the one we are looking for) we memorize that node in a variable (and change the variable when we find a new such node). When the current node is at the one we are looking for, this variable contains its successor.

BST - Remove a node



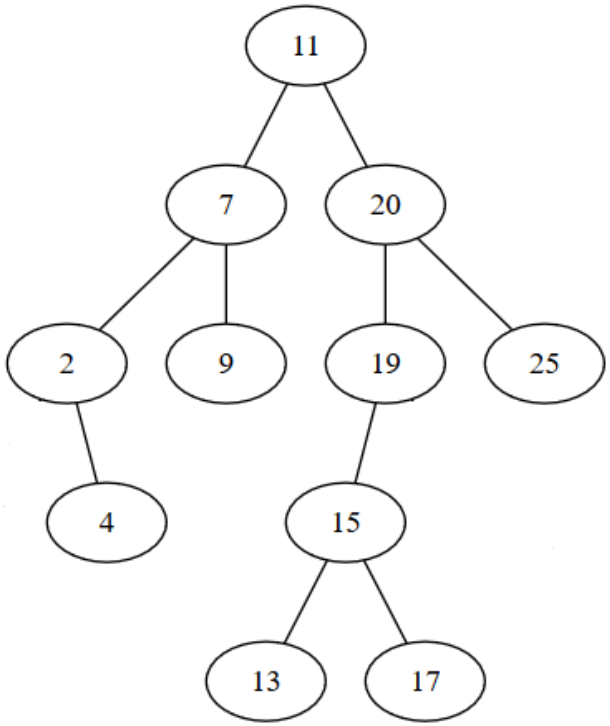
- How can we remove the value 25?

BST - Remove a node



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

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

Think about it

- For a Binary Tree we have discussed that in some situations, if we have two tree traversals, we can rebuild the tree based on them.
- Can we do the same for a Binary Search Tree from one single traversal? Which one(s)?

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?
  - We can keep in every node, besides the information, the number of nodes in the left subtree. This gives us automatically the "position" of the root in the SortedList. When we have operations that are based on positions, we use these values to decide if we go left or right.

Binary Search Tree