DATA STRUCTURES AND ALGORITHMS

LECTURE 13

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

In Lecture 12

- Binary Tree
- Binary Search Tree

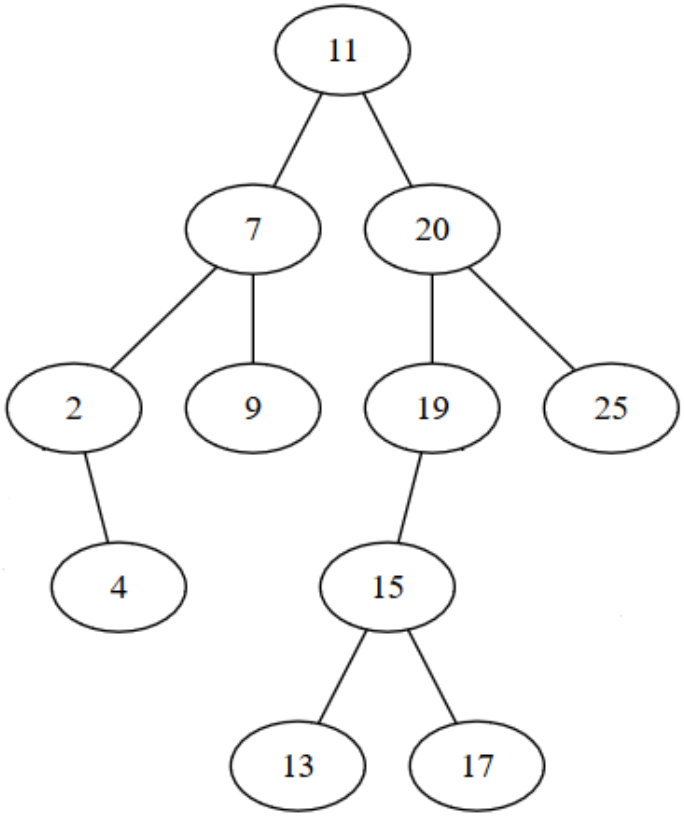
Today

- Binary Search Tree
- Conclusions

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

- 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

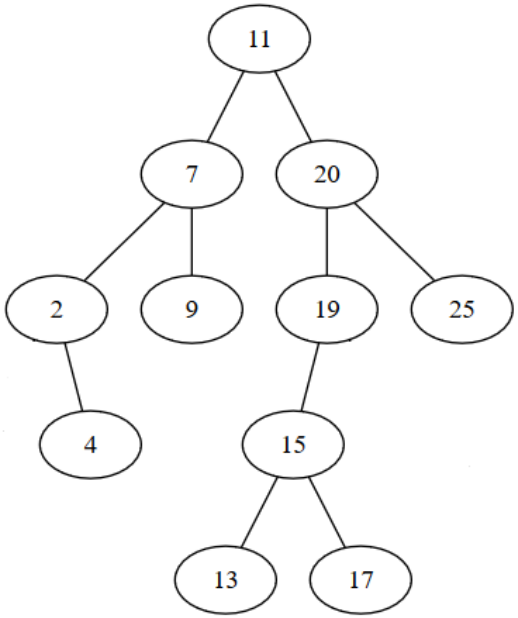
- We will use a linked representation with dynamic allocation (similar to what we used for binary trees)

BSTNode:
  info: TElem
  left: ↑ BSTNode
  right: ↑ BSTNode

BinarySearchTree:
  root: ↑ BSTNode

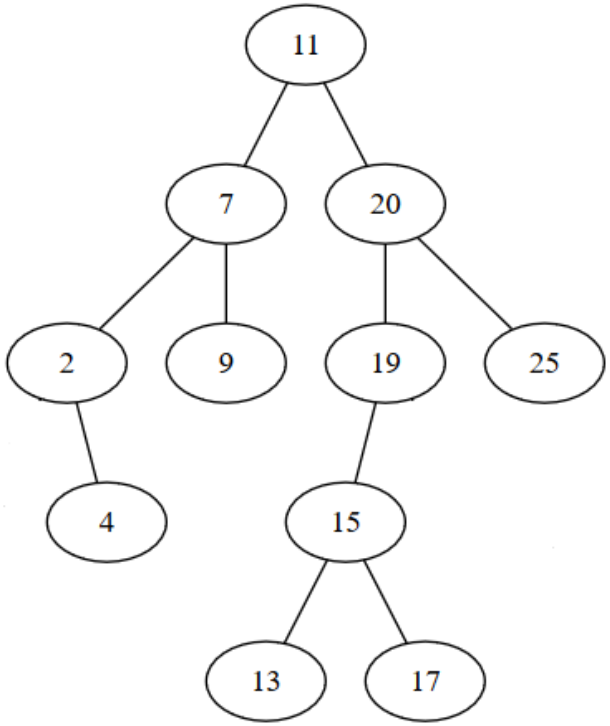
Binary Search Tree - search operation

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



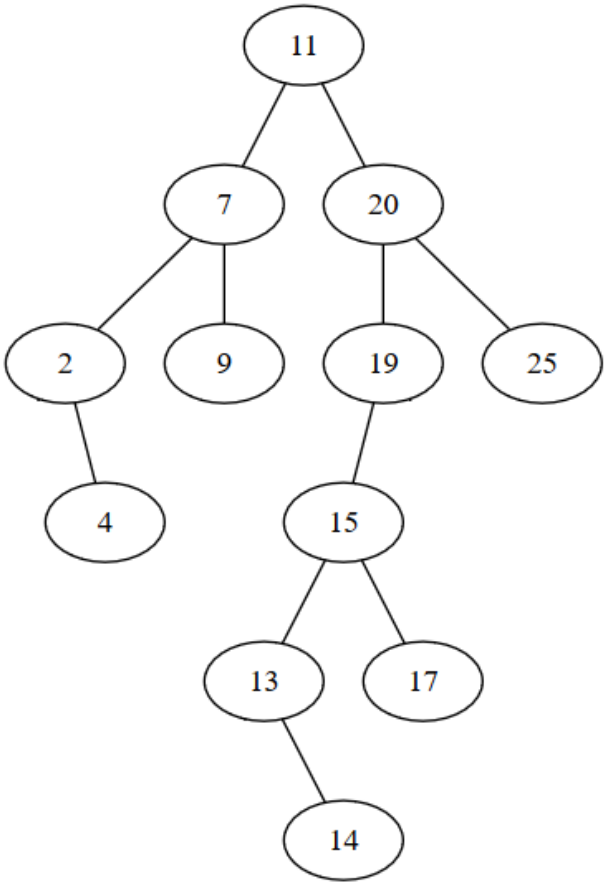
- How can we search for element 15? And for element 14?

BST - 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

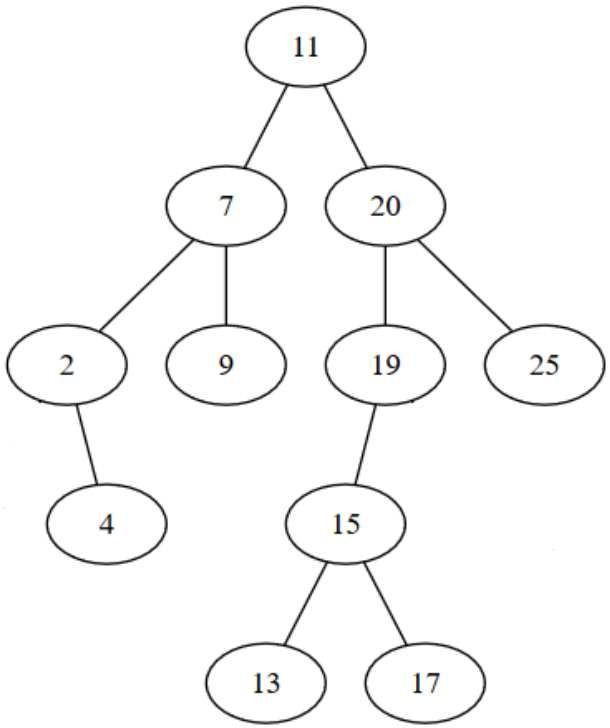
```
function insert_rec(node, e) is:
//pre: node is a BSTNode, e is TComp
//post: a node containing e was added in the tree starting from node
    if node = NIL then
        node ← initNode(e)
    else if [node].info ≥ e then
        [node].left ← insert_rec([node].left, e)
    else
        [node].right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

```
subalgorithm insert(bst, e) is:
    current ← bst.root
    parent ← NIL
    while current ≠ NIL execute
        parent ← current
        if e ≤ [current].info then
            current ← [current].left
        else
            current ← [current].right
        end-if
    end-while
    if parent = NIL then
        bst.root ← initNode(e)
    else
        if e ≤ [parent].info then
            [parent].left ← initNode(e)
        else
            [parent].right ← initNode(e)
        end-if
    end-if
end-subalgorithm
```

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
```

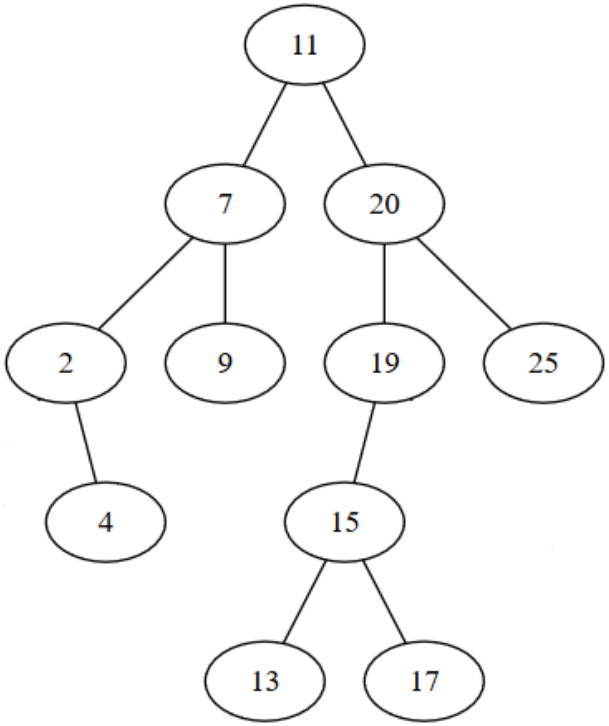
BST - Finding the minimmum element

- 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 ≠ 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
```

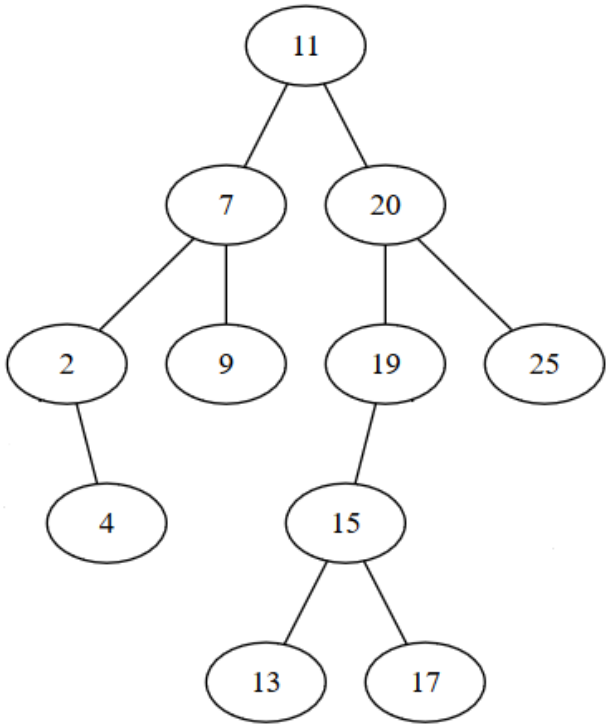
- Complexity:

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
```

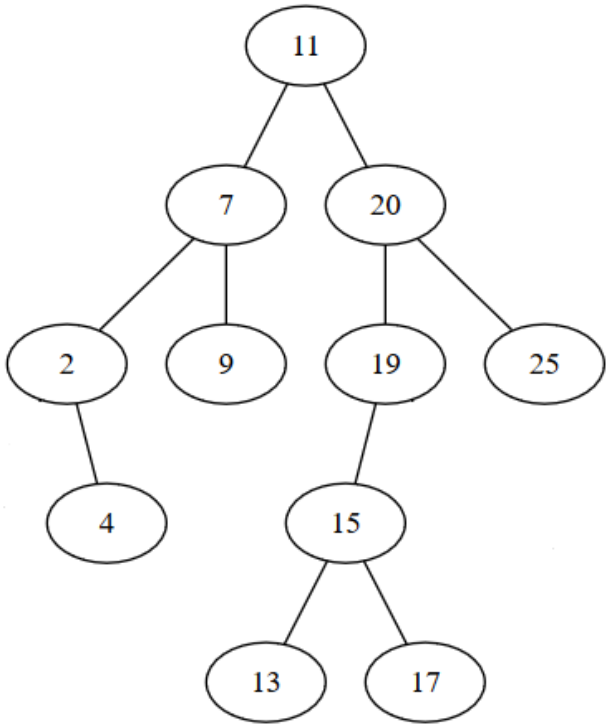
- Complexity: $O(n)$

BST - Finding the successor of a node



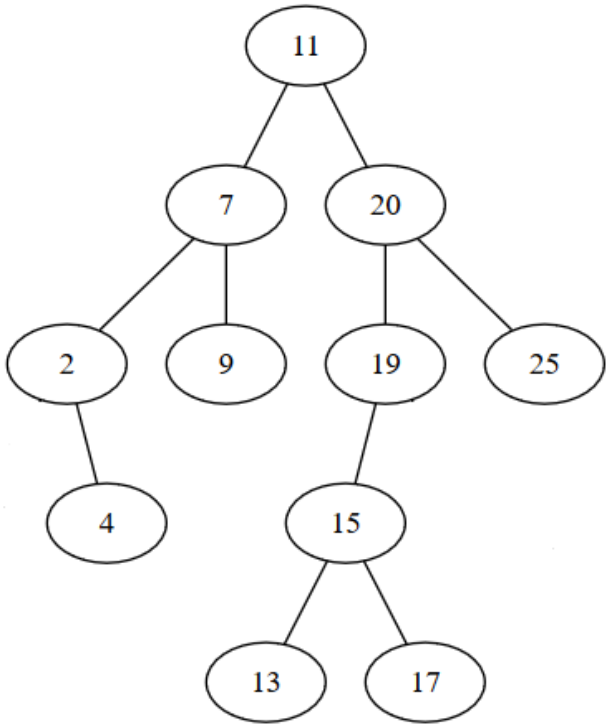
- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11?

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.

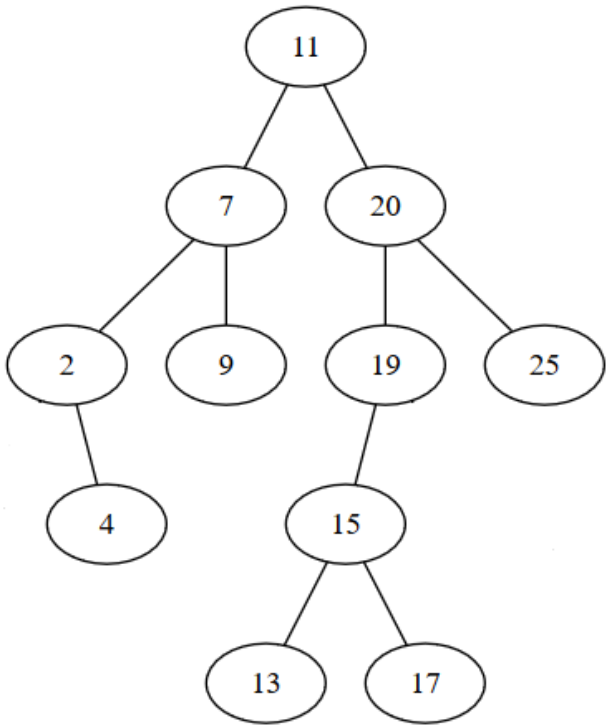
BST - Finding successor of a node

- 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

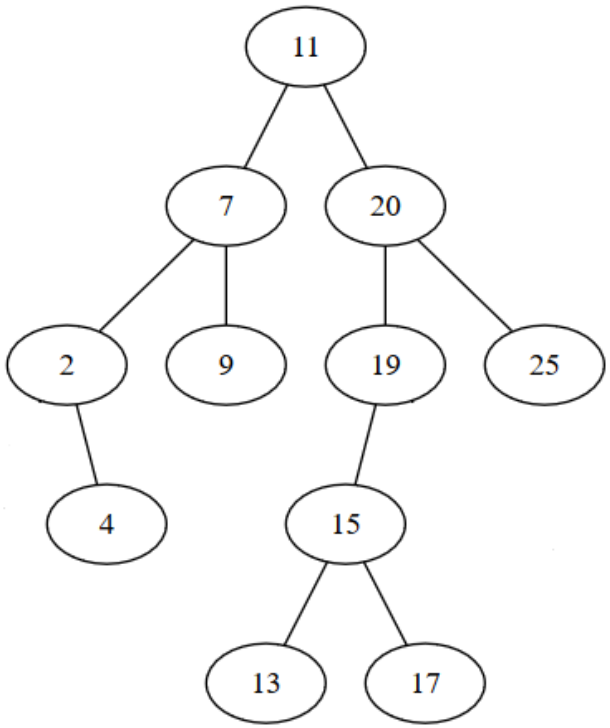
- If we do not have direct access to the parent, finding the successor of a node is $O(n^2)$.
- Can we reduce this complexity?
- $O(n^2)$ is given by the potentially repeated calls for the *parent* function. But we only need the *last* parent, where we went left. We can do one single traversal from the 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



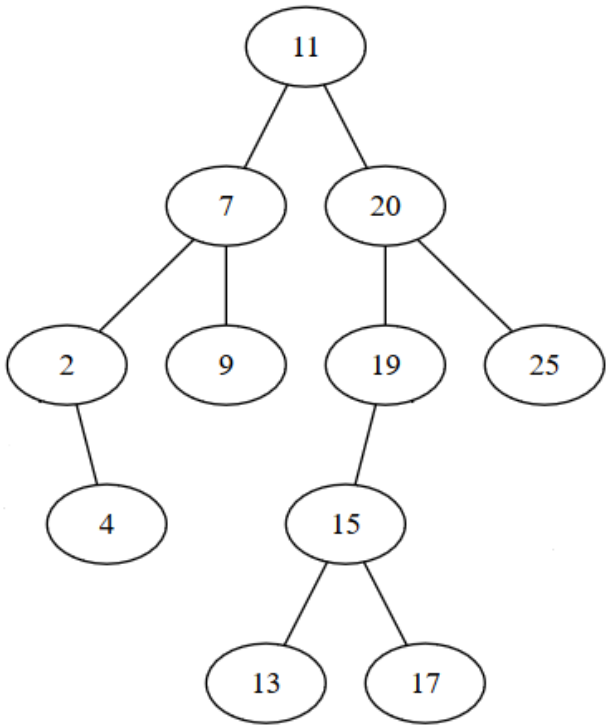
- How can we remove the value 25?

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

Iterator for a BST

- For an iterator on a binary search tree we need to traverse the nodes of the tree. What type of traversal do we need?
  - preorder
  - postorder
  - inorder
  - level order

Iterator for a BST

- For an iterator on a binary search tree we need to traverse the nodes of the tree. What type of traversal do we need?
  - preorder
  - postorder
  - inorder
  - level order
- a BST is used to implement a sorted container, so we need a traversal which visits the elements in a sorted order. Which traversal gives us the elements in sorted order?

Iterator for a BST

- For an iterator on a binary search tree we need to traverse the nodes of the tree. What type of traversal do we need?
  - preorder
  - postorder
  - inorder
  - level order
- a BST is used to implement a sorted container, so we need a traversal which visits the elements in a sorted order. Which traversal gives us the elements in sorted order?
- We need an inorder traversal

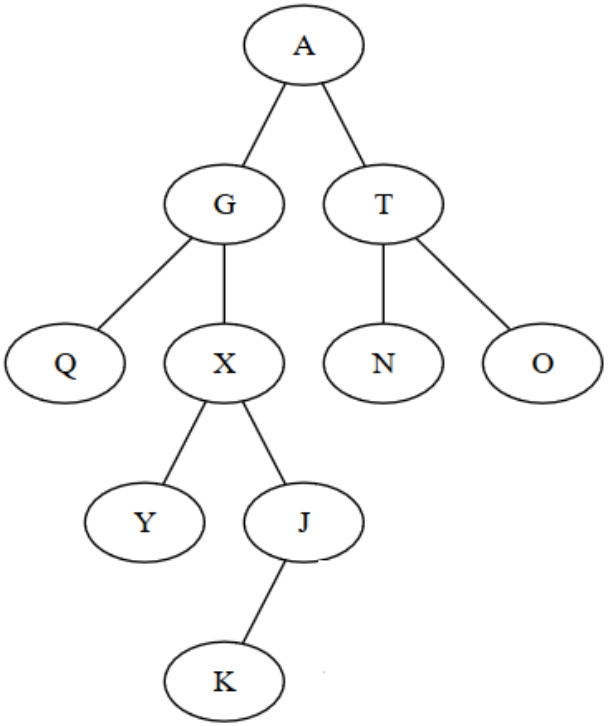
Iterator for BST

- Inorder traversal means that we first visit the left subtree of a node, then the node and after that the right subtree.
- We have seen at the binary trees, how an inorder traversal can be implemented recursively. Let's see now how can we implement it non-recursively.

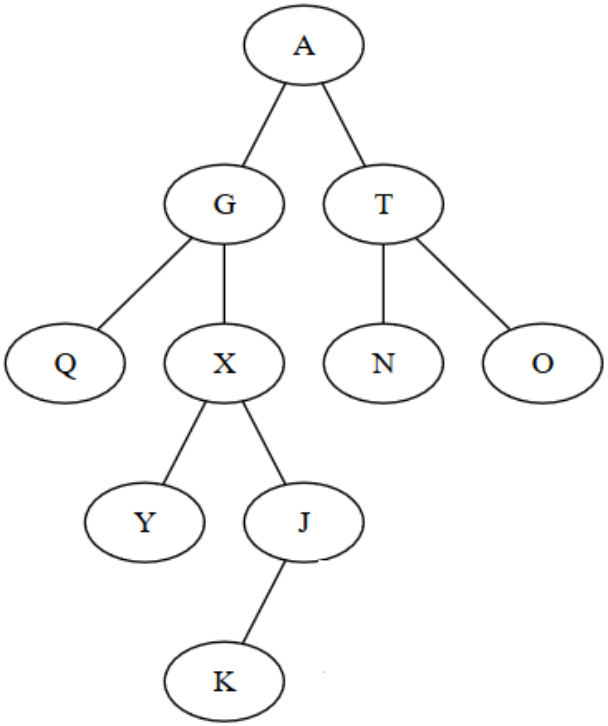
Nonrecursive inorder traversal

- 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



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 binary search tree iterator

- Let's see how we can split this traversal into a separate functions for an iterator: *init*, *getCurrent*, *valid*, *next*.
- Assume an implementation without a parent node.
- What fields do we need to keep in the iterator structure?

Inorder binary search tree iterator

- Let's see how we can split this traversal into a separate functions for an iterator: *init*, *getCurrent*, *valid*, *next*.
- Assume an implementation without a parent node.
- What fields do we need to keep in the iterator structure?

InorderIterator:
bst: BinarySearchTree
s: Stack
currentNode: ↑ BSTNode

Inorder binary search tree iterator - init

- What should the *init* operation do?

Inorder binary search tree iterator - init

- What should the *init* operation do?

```
subalgorithm init (it, bst) is:
//pre: it - is an InorderIterator, bst is a BinarySearchTree
    it.bst ← bst
    init(it.s)
    node ← bst.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?

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?

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?

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
```

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?

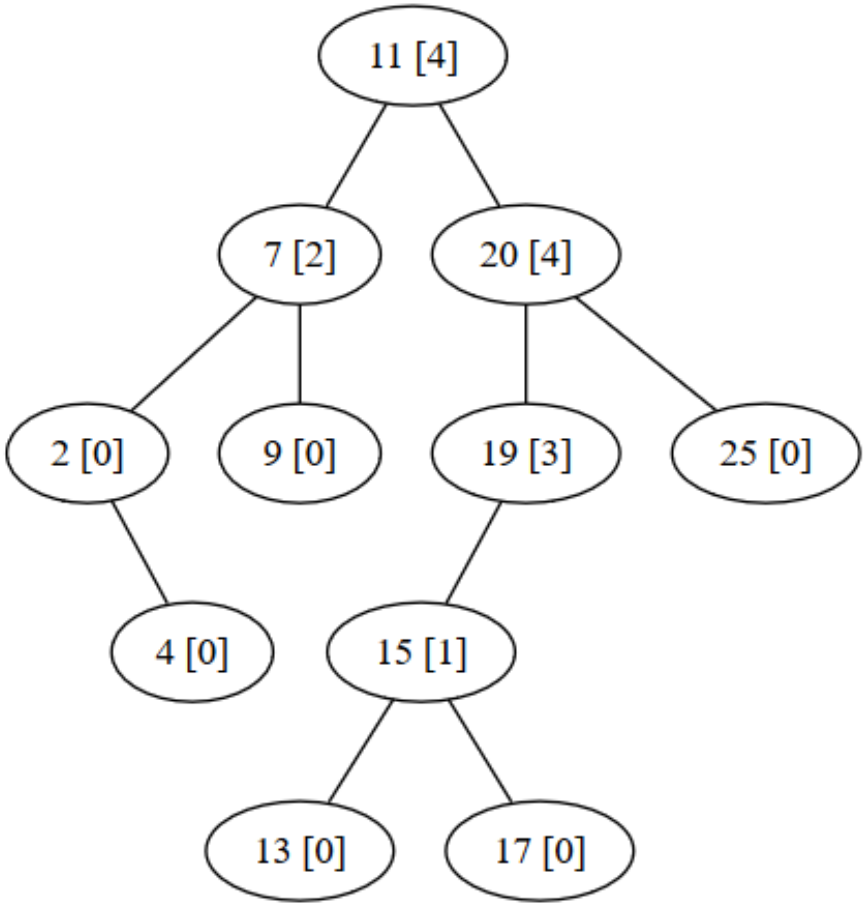
Binary Search Tree

- 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



Conclusions

- During the semester we have talked about the most important containers (ADT) and their main properties and operations
  - Bag, Set, Map, Multimap, List, Stack, Queue and their sorted versions
- We have also talked about the most important data structures that can be used to implement these containers
  - Dynamic array, Linked lists, Binary heap, Hash table, Binary Search Tree

Conclusions

- You should be able to identify the most suitable container for solving a given problem:

Conclusions

- You should be able to identify the most suitable container for solving a given problem:
- Example: *You have a type Student which has a name and a city. Write a function which takes as input a list of students and prints for each city all the students that are from that city. Each city should be printed only once and in any order.*
- How would you solve the problem? What container would you use?

Conclusions

- When you use containers existing in different programming languages, you should have an idea of how they are implemented and what is the complexity of their operations:

Conclusions

- When you use containers existing in different programming languages, you should have an idea of how they are implemented and what is the complexity of their operations:
- Consider the following algorithm (written in Python):

```
def testContainer(container, l):
 '''
container is a container with integer numbers
l is a list with integer numbers
'''
    count = 0
    for elem in l:
        if elem in container:
            count += 1
    return count
```

- The above function counts how many elements from the list *l* can be found in the container. What is the complexity of *testContainer*?

Conclusions

- Consider the following problem: *We want to model the content of a wallet, by using a list of integer numbers, in which every value denotes a bill. For example, a list with values [5, 1, 50, 1, 5] means that we have 62 RON in our wallet.*
  Obviously, we are not allowed to have any numbers in our list, only numbers corresponding to actual bills (we cannot have a value of 8 in the list, because there is no 8 RON bill).
  We need to implement a functionality to pay a given amount of sum and to receive rest of necessary.
  There are many optimal algorithms for this, but we go for a very simple (and non-optimal): keep removing bills of the wallet until the sum of removed bills is greater than or equal to the sum you want to pay.
  If we need to receive a rest, we will receive it in 1 RON bills.

Conclusions

- For example, if the wallet contains the values [5, 1, 50, 1, 5] and we need to pay 43 RON, we might remove the first 3 bills (a total of 56) and receive the 13 RON rest in 13 bills of 1.

Conclusions

- For example, if the wallet contains the values [5, 1, 50, 1, 5] and we need to pay 43 RON, we might remove the first 3 bills (a total of 56) and receive the 13 RON rest in 13 bills of 1.
- This is an implementation provided by a student. What is wrong with it?

```
public void spendMoney(ArrayList<Integer> wallet, Integer amount) {
    Integer spent = 0;
    while (spent < amount) {
        Integer bill = wallet.remove(0); //removes element from position 0
        spent += bill;
    }
    Integer rest = spent - amount;
    while (rest > 0) {
        wallet.add(0, 1);
        rest--;
    }
}
```