

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 12

Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2017

# In Lecture 11...

- Hash tables
- Trees
- Binary Trees

# Today

- 1 Binary Tree Traversal
- 2 Huffman coding
- 3 Binary Search Trees

# Binary Tree Traversal

- A node of a (binary) tree is 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 (binary) tree means visiting all of its nodes.
- For a binary tree there are 4 possible traversals:
  - Preorder
  - Inorder
  - Postorder
  - Level order (breadth first)

# 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

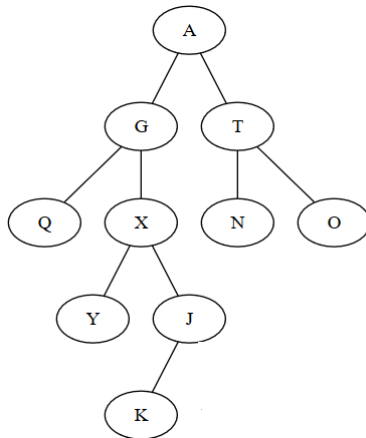
## BinaryTree:

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  $\uparrow$  BTreeNode*

**if** node  $\neq$  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)
```

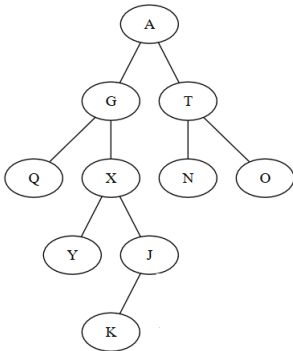
**end-subalgorithm**

- 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
```

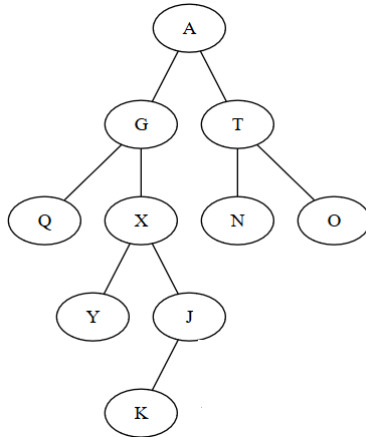
## Preorder traversal - non - recursive implementation

- 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  $\uparrow$  BTreeNode*

**if** node  $\neq$  NIL **then**

*inorder\_recursive*([node].left)

    @visit [node].info

*inorder\_recursive*([node].right)

**end-if**

**end-subalgorithm**

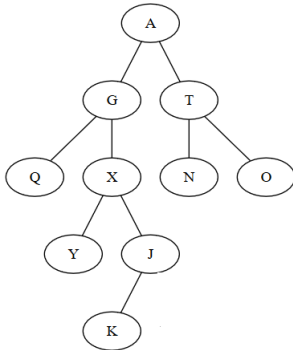
- We need again a wrapper subalgorithm to perform the first call to *inorder\_recursive* with the root of the tree as parameter.
- The traversal takes  $\Theta(n)$  time for a tree with  $n$  nodes.



# 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 pushed 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
  
```

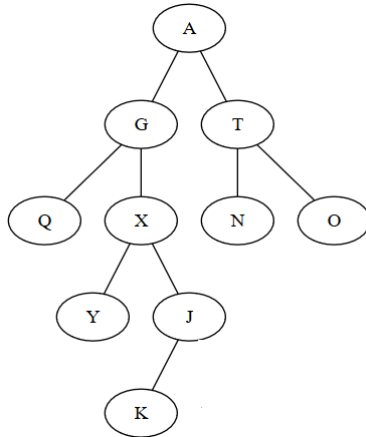
# Inorder traversal - non-recursive implementation

- 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  $\uparrow$  `BTNode`*

**if** `node  $\neq$  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.



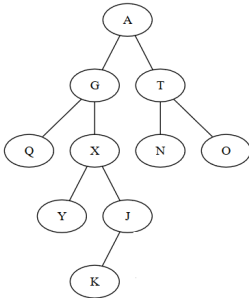
## Postorder traversal with two stacks

- 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  $\leftarrow$  tree.root

**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**

**while** not isEmpty(s) **execute**

    node  $\leftarrow$  pop(s)

**if** [node].right  $\neq$  NIL and (not isEmpty(s)) and [node].right = top(s) **th**

        pop(s)

        push(s, node)

        node  $\leftarrow$  [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  $O(n)$

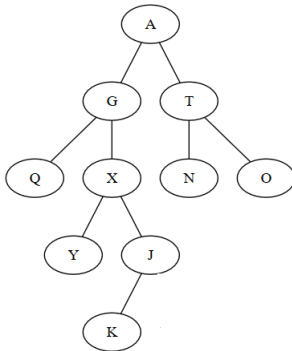
# Traversal without a stack

- 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



- 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, if 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  $\leftarrow$  tree.root

**while** current  $\neq$  NIL **execute**

**if** [current].left  $\neq$  NIL and [[current].left].visited = false **then**

current  $\leftarrow$  [current].left

**else if** [current].visited = false **then**

@visit current

[current].visited  $\leftarrow$  true

**else if** [current].right  $\neq$  NIL and [[current].right].visited = false **then**

current  $\leftarrow$  [current].right

**else**

*//we are going up, but before that reset the children to not-visited*

**if** [current].left  $\neq$  NIL **then**

[[current].left].visited  $\leftarrow$  false

**end-if**

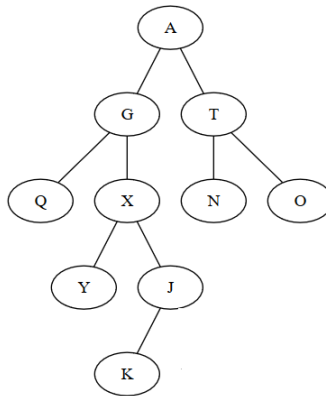
*//continued on the next slide...*

# Inorder traversal without a stack - implementation

```
    if [current].right  $\neq$  NIL then
        [[current].right].visited  $\leftarrow$  false
    end-if
    current  $\leftarrow$  [current].parent
end-if
end-while
if tree.root  $\neq$  NIL then
    [tree.root].visited  $\leftarrow$  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

# Level order traversal implementation

- For level order traversal we do not have a recursive implementation, only a non-recursive one.
- The algorithm is very similar to preorder traversal, except that we use a queue instead of a stack.
  - We start with an empty queue
  - Push the root of the tree to the queue
  - While the queue is not empty:
    - Pop a node and visit it
    - Push the node's left child to the queue
    - Push the node's right child to the queue

## Level order traversal - non-recursive implementation

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

## Level order traversal - non-recursive implementation

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

# 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: ↑ BTreeNode



# 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  $\leftarrow$  bt

init(it.s)

node  $\leftarrow$  bt.root

**while** node  $\neq$  NIL **execute**

    push(it.s, node)

    node  $\leftarrow$  [node].left

**end-while**

**if** not isEmpty(it.s) **then**

    it.currentNode  $\leftarrow$  top(it.s)

**else**

    it.currentNode  $\leftarrow$  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  $\leftarrow$  false  
  else  
    valid  $\leftarrow$  true  
  end-if  
end-function
```

# Inorder binary tree iterator - next

- What should the *next* operation do?

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

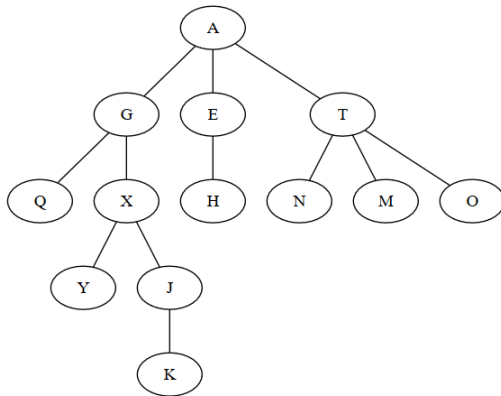
- Can you rebuild the tree if you have the *postorder* and the *inorder* traversal?
- Can you rebuild the tree if you have the *preorder* and the *postorder* traversal?

# Tree traversal

- When we have a tree that is not binary, there are only two possible traversals:
  - Level order (breadth first) - looks exactly the same as in case of a binary tree, we use an auxiliary queue to store the nodes.
  - Depth first - similar to breadth first, but we use an auxiliary stack to store the nodes (it is the generalizations of preorder traversal). By using a stack, the algorithms will always go *as deep as possible* in the tree and only once a whole subtree of a node was visited we pass to the next child.



## Tree traversal - Example



- Breadth-first traversal: A, G, E, T, Q, X, H, N, M, O, Y, J, K
- Depth-first traversal: A, G, Q, X, Y, J, K, E, H, T, N, M, O

# 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 decoding, when we find a 01, we will not know whether it is E or the beginning of X).

# Huffman coding

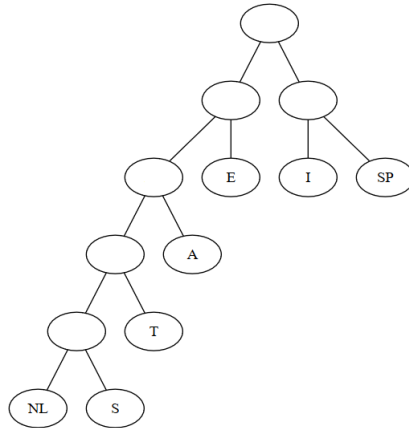
- When building the Huffman encoding for a message, we first have to compute the frequency of every character from the message, because we are going to define the codes based on the frequencies.
- Assume that we have a message with the following letters and frequencies

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

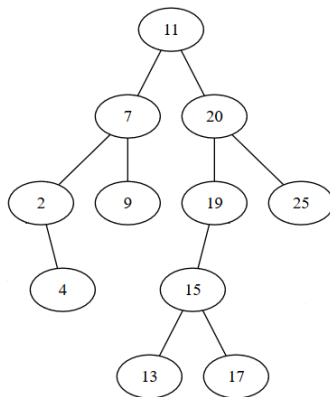
- Assume we have the following code and we want to decode it:  
011011000100010011100100000
- We do not know where the code of each character ends, but we can use the previously built tree to decode it.
- Start parsing the code and iterate through the tree in the following way:
  - Start from the root
  - 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  $x$
- 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:  $\uparrow$  BSTNode

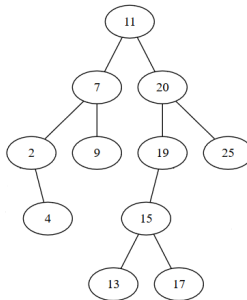
right:  $\uparrow$  BSTNode

### BinarySearchTree:

root:  $\uparrow$  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 algorithm* 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  $\leftarrow$  false  
  else  
    if [node].info = elem then  
      search_rec  $\leftarrow$  true  
    else if [node].info < elem then  
      search_rec  $\leftarrow$  search_rec([node].right, elem)  
    else  
      search_rec  $\leftarrow$  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**