DATA STRUCTURES AND ALGORITHMS
LECTURE 11

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

In Lecture 10...

- Hash tables
    - Open Addressing
    - Perfect hashing
    - Cuckoo hashing
    - Linked hash tables

Today

- Linked hash tables
- Trees
- Binary trees

Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.

- Since it is still a hash table, we want to have, on average, $\Theta(1)$ for insert, remove and search, these are done in the same way as before, the *extra* linked list is used only for iteration.

Linked Hash Table - Implementation

- What structures do we need to implement a Linked Hash Table?

<u>Node:</u>
info: TKey
nextH: ↑ Node *//pointer to next node from the collision*
nextL: ↑ Node *//pointer to next node from the insertion-order list*
prevL: ↑ Node *//pointer to prev node from the insertion-order list*

<u>LinkedHT:</u>
m:Integer
T:(↑ Node)[]
h:TFunction
head: ↑ Node
tail: ↑ Node

- How can we implement the *insert* operation?

```
subalgorithm insert(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k is added into lht
   allocate(newNode)
   [newNode].info ← k
   @set all pointers of newNode to NIL
   pos ← lht.h(k)
   //first insert newNode into the hash table
   if lht.T[pos] = NIL then
      lht.T[pos] ← newNode
   else
      [newNode].nextH ← lht.T[pos]
      lht.T[pos] ← newNode
   end-if
//continued on the next slide...
```

Linked Hash Table - Insert

```
//now insert newNode to the end of the insertion-order list
if lht.head = NIL then
    lht.head ← newNode
    lht.tail ← newNode
else
    [newNode].prevL ← lht.tail
    [lht.tail].nextL ← newNode
    lht.tail ← newNode
end-if
end-subalgorithm
```

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.

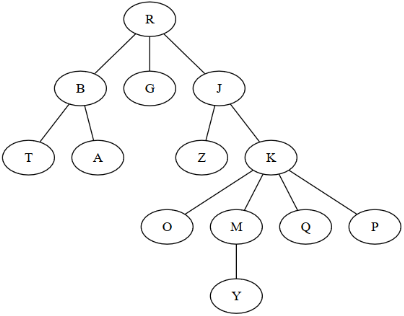
Tree - Terminology I

- 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

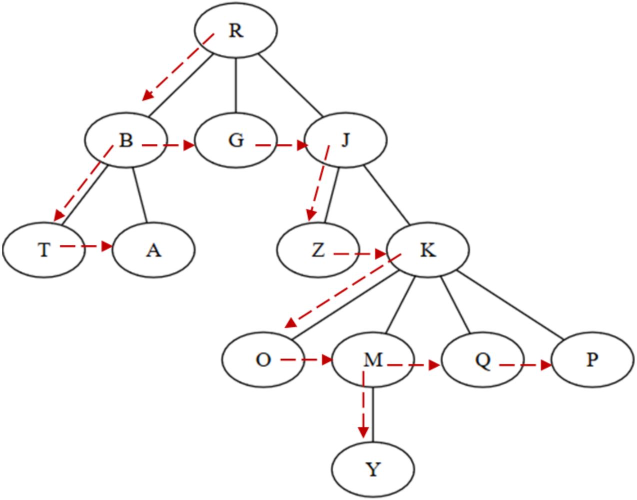
- How can we represent a tree in which every node has at most k children?
- One option is to have a structure for a *node* that contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - k fields, one for each child
- Obs: this is doable if k is not too large

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.

- 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



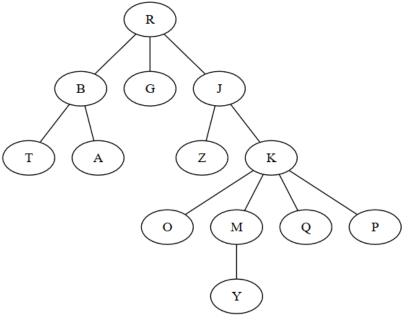
Tree traversals

- 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

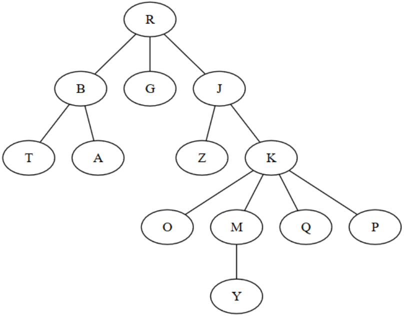
Depth first traversal

- Traversal starts from root
- From root we visit one of the children, than 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



Depth first traversal example

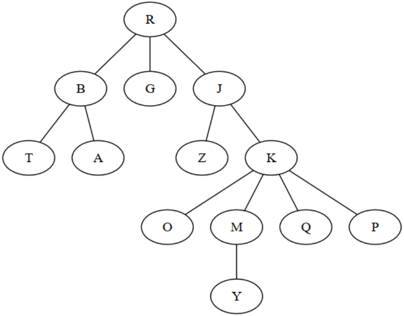


- Stack s with the root: R
- Visit R (pop from stack) and push its children: s = [B G J]
- Visit B and push its children: s = [T A G J]
- Visit T and push nothing: s = [A G J]
- Visit A and push nothing: s = [G J]
- Visit G and push nothing: s = [J]
- Visit J and push its children: s = [Z K]
- etc...

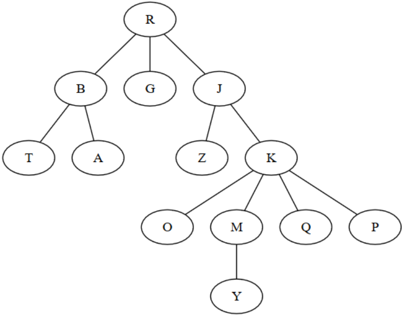
Level order traversal

- 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



Level order traversal example

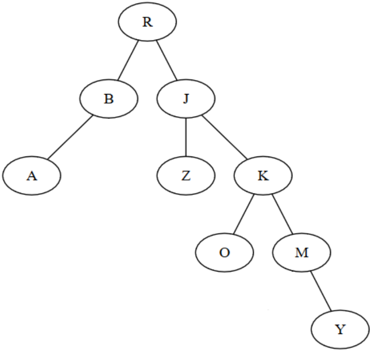


- Queue q with the root: R
- Visit R (pop from queue) and push its children: q = [B G J]
- Visit B and push its children: q = [G J T A]
- Visit G and push nothing: q = [J T A]
- Visit J and push its children: q = [T A Z K]
- Visit T and push nothing: q = [A Z K]
- Visit A and push nothing: q = [Z K]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

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.

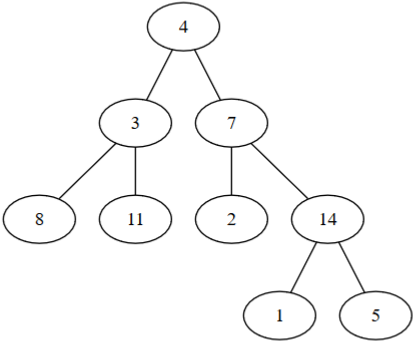
Binary tree - example



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

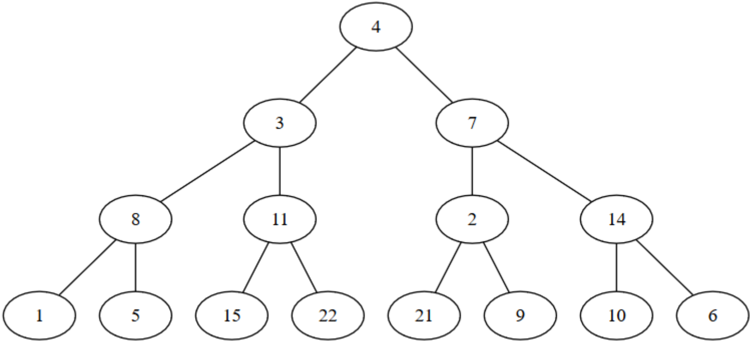
Binary tree - Terminology I

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

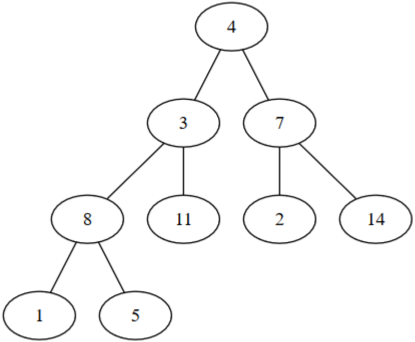


Binary tree - Terminology II

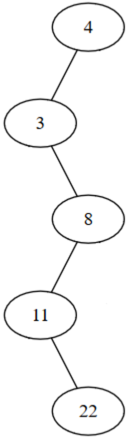
- A binary tree is called *complete* if all leaves are one the same level and all internal nodes have exactly 2 children.



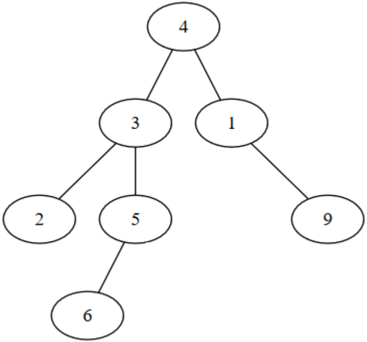
- A binary tree is called *almost complete* if it is a *complete* binary tree except for the last level, where nodes are completed from left to right (binary heap - structure).



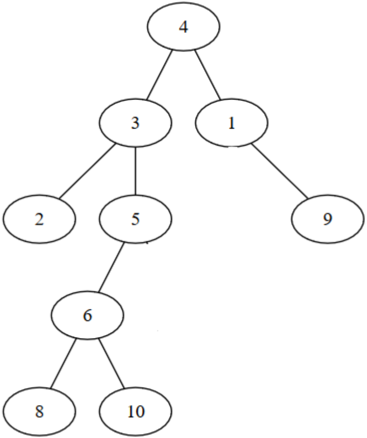
- A binary tree is called *degenerate* if every internal node has exactly one child (it is actually a chain of nodes).



- 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).



- 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 $N+1$ - if the tree is degenerate.
- A binary tree with N nodes has a height between $log_2 N$ and $N-1$.

- 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

- 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

- 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

- 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

- 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

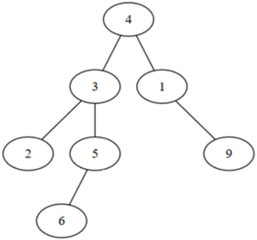
Possible representations

- 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.

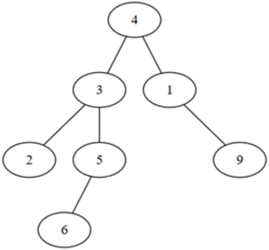
Possible representations I

- 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 have to keep a linked list of empty positions to make adding a new node easier.

Possible representations IV

- The linked list of empty positions has to be created when the empty binary tree is created. While a tree is a non-linear data structure, we can still use the left (and/or right) array to create a singly (or doubly) linked list of empty positions.
- Obviously, when we do a resize, the newly created empty positions have to be linked again.

info								
left	2	3	4	5	6	7	8	-1
right								

firstEmpty = 1

root = -1

cap = 8

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) - the same as in case of a (non-binary) tree

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

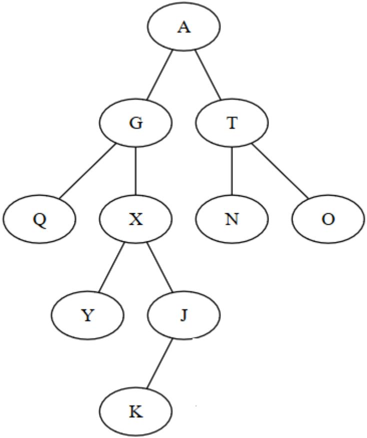
<u>BTNode:</u>
info: TElem
left: ↑ BTNode
right: ↑ BTNode

<u>BinaryTree:</u>
root: ↑ BTNode

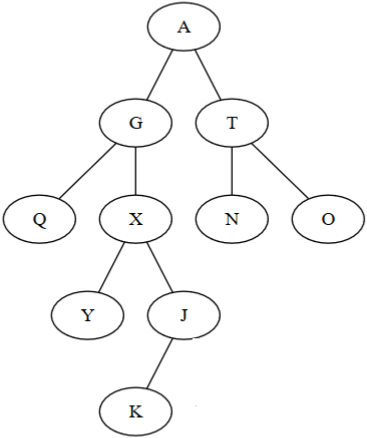
Preorder traversal

- 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 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 $\uparrow$ *BTNode*
  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.

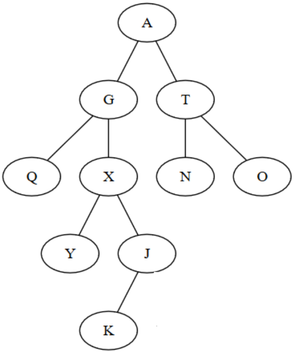
subalgorithm preorderRec(tree) **is:**
//pre: tree is a BinaryTree
  preorder_recursive(tree.root)
end-subalgorithm

- 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

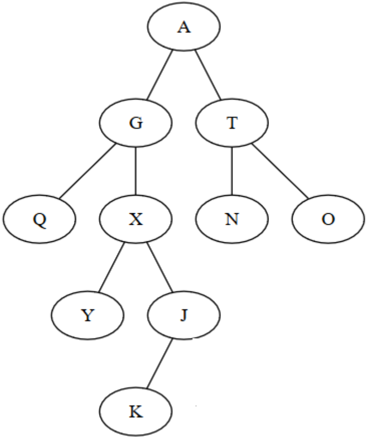
```
subalgorithm preorder(tree) is:
//pre: tree is a binary tree
   s: Stack //s is an auxiliary stack
   if tree.root ≠ NIL then
      push(s, tree.root)
   end-if
   while not isEmpty(s) execute
      currentNode ← pop(s)
      @visit currentNode
      if [currentNode].right ≠ NIL then
         push(s,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

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

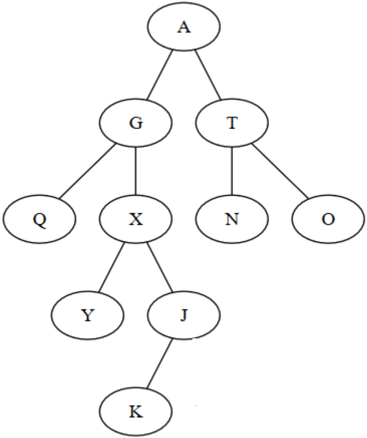
Inorder traversal

- 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 example



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

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

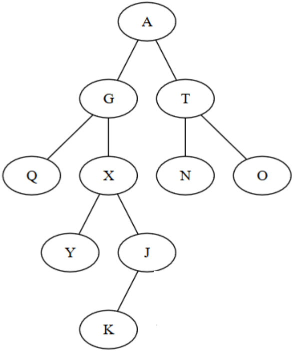
subalgorithm inorder_recursive(node) **is:**
//pre: node is a $\uparrow$ BTNode
  if node $\neq$ NIL **then**
    inorder_recursive([node].left)
    @visit [node].info
    inorder_recursive([node].right)
  end-if
end-subalgorithm

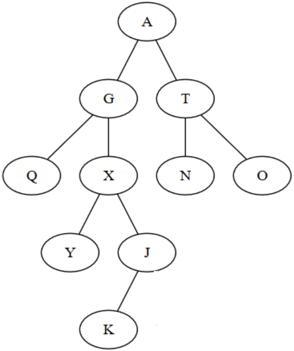
- We need again a wrapper subalgorithm to perform the first call to *inorder_recursive* with the root of the tree as parameter.
- The traversal takes $\Theta(n)$ time for a tree with n nodes.

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
```

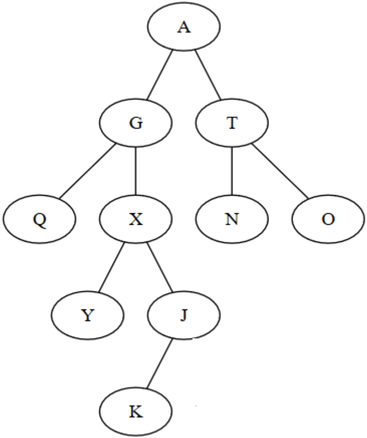
Inorder traversal - non-recursive implementation

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

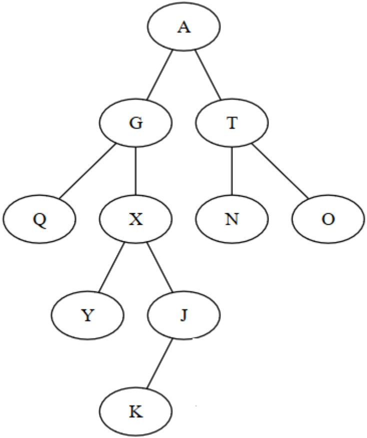
Postorder traversal

- 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 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.