

## Lecture 10

- Binary tree
  - ... (see lecture 6)
  - ADT
  - Traversals: recursive, non-recursive
  - Iterator
- Trees
  - K-ary trees

### Lecture06 – treeIntro

An introduction containing:

- Definition
- Terminology
- Binary tree: types, properties, representation

Lect. PhD. Lupsa Dana  
Babes - Bolyai University  
Computer Science and Mathematics Faculty  
2022 - 2023

## Previously, in Lectures 8 & 9

- Hash Function , Hash Table  
& Collision Resolution Methods
- Other types of hashing

# Trees

Lecture06

A tree is a finite set  $T$  of 0 or more nodes, with the following properties:

- If  $T$  is empty, then the tree is empty
- If  $T$  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,  $T_1, T_2, \dots, T_k$ . The trees  $T_1, T_2, \dots, T_k$  are called the subtrees (children) of  $R$ , and  $R$  is called the parent of the subtrees.

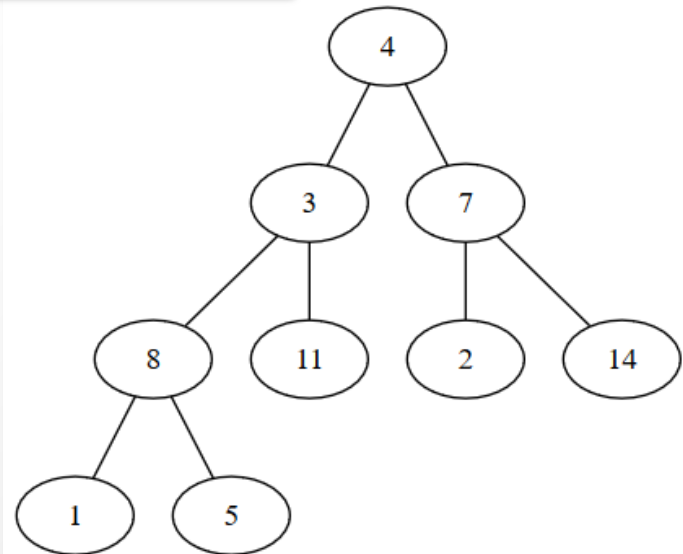
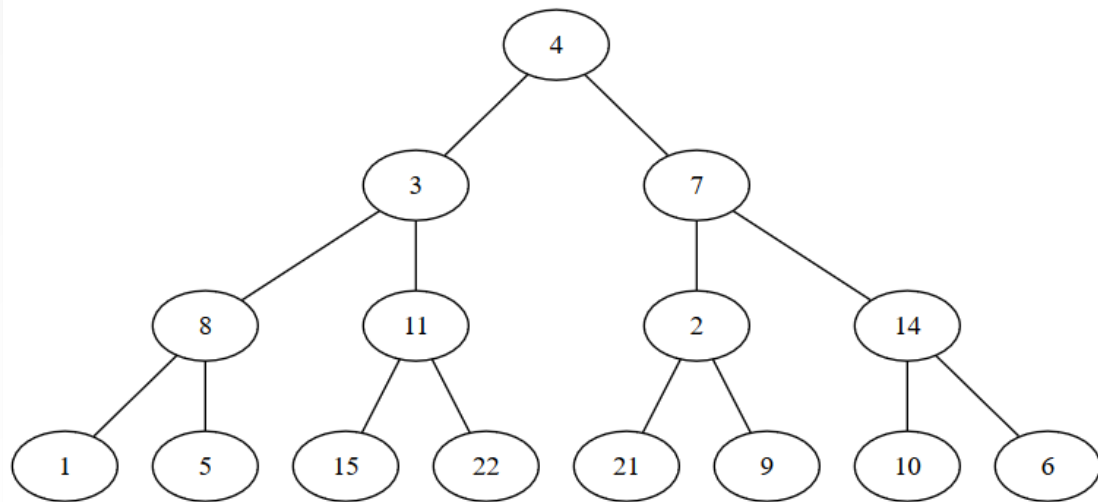
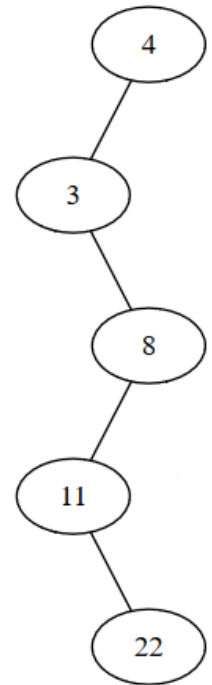
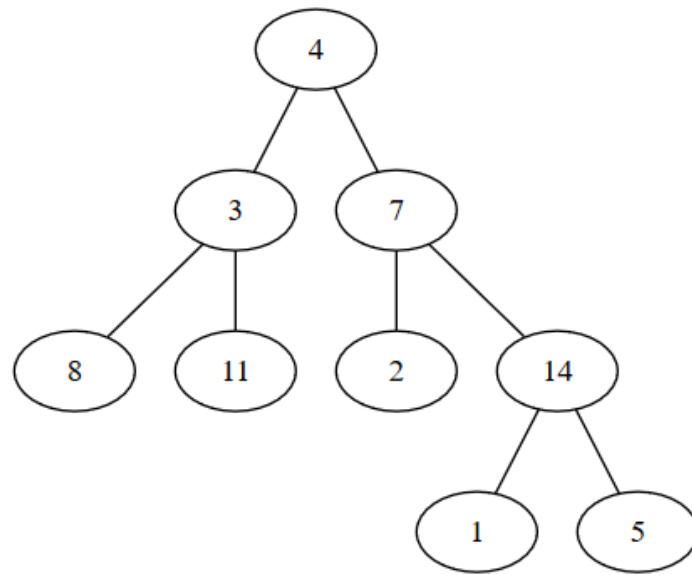
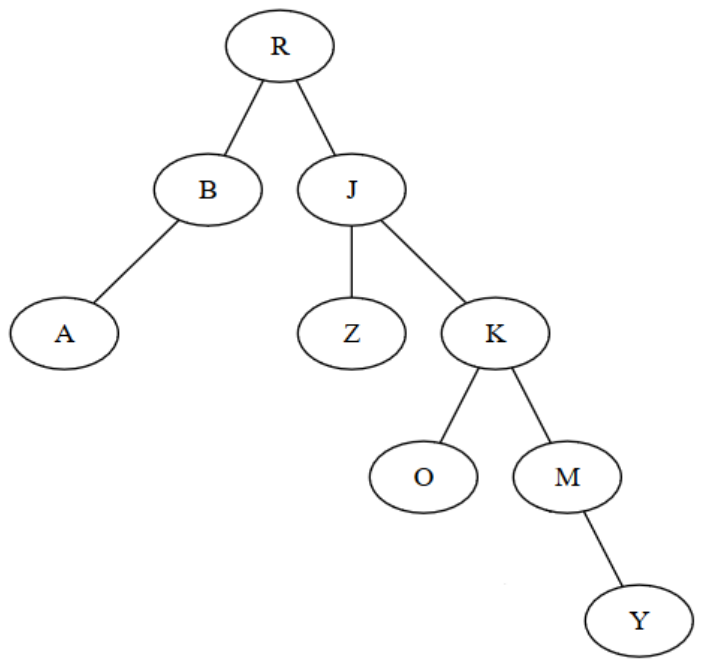
# Binary trees

Lecture06

Binary tree: a tree in which each node has at most two children.

- 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.
- A binary tree is called **full** if every internal node has exactly two children.
- A binary tree is called **complete** if all leaves are on 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 **balanced** if the difference between the height of the left and right subtrees of a node is at most 1.

# Binary trees



# ADT Binary Tree

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

- `init(bt)`

- `destroy(bt)`

- `isEmpty(bt)`

- `iterator (bt, traversal, i)`

- `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  $e$

- `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)`

- `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  $right$

- `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

- `left(bt)`

- `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

# Binary tree - representation

Lecture06

- Representation using an array, similar to a binary heap

Disadvantage:

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

- Linked representation
  - with dynamic allocation
  - on an array

BTNode:

info: TElem

left: ↑ BTNode

right: ↑ BTNode

BinaryTree:

root: ↑ BTNode

# Binary tree - traversals

Preorder traversal:

- Visit the root of the tree
- Traverse the left subtree - if exists
- Traverse the right subtree - if exists

Inorder traversal:

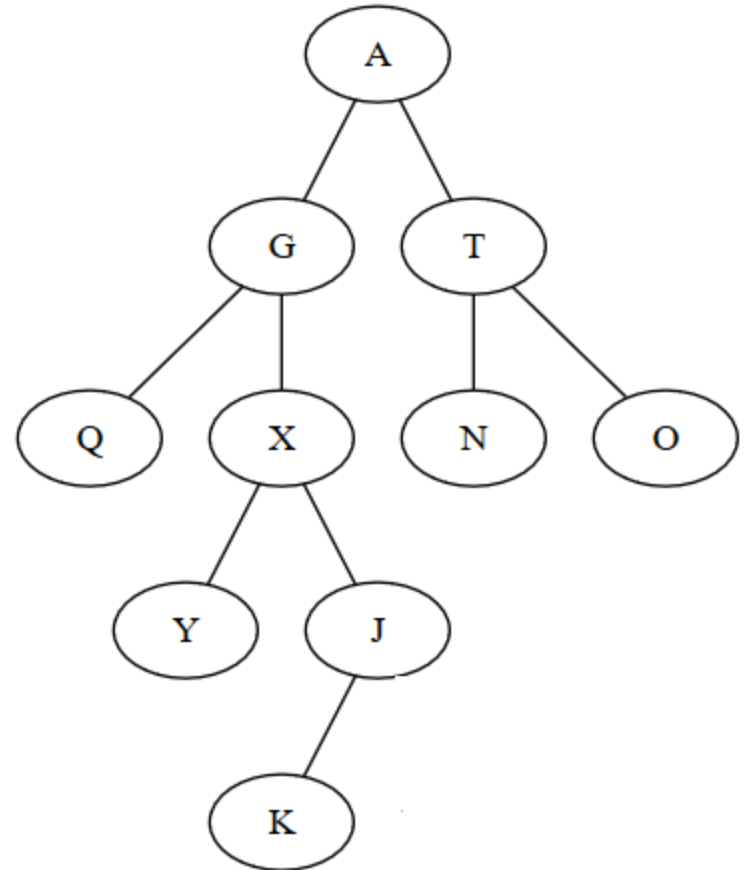
- Traverse the left subtree - if exists
- Visit the root of the tree
- Traverse the right subtree - if exists

Postorder traversal:

- Traverse the left subtree - if exists
- Traverse the right subtree - if exists
- Visit the root of the tree

Traversal on levels

5/12/2023



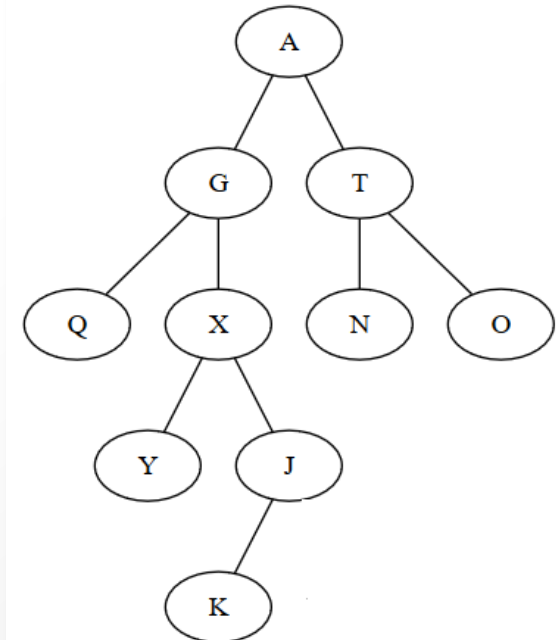


# Preorder traversal - recursive implementation

```
subalgorithm preorderRec(tree) is:  
  preorder_recursive(tree.root)  
end-subalgorithm
```

```
subalgorithm preorder_recursive(node) is:  
  if node  $\neq$  NIL then  
    @ visit [node].info  
    preorder_recursive([node].left)  
    preorder_recursive([node].right)  
  end-if  
end-subalgorithm
```

The traversal takes  $\Theta(n)$  time for a tree with  $n$  nodes.



***How can we implement  
inorder and postorder  
traversals?***

# On-level traversal

subalgorithm onLevel(tree) is:

    init(q)           // 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

# Preorder traversal - non-recursive implementation

subalgorithm preorder(tree) is:

  init(s)       //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

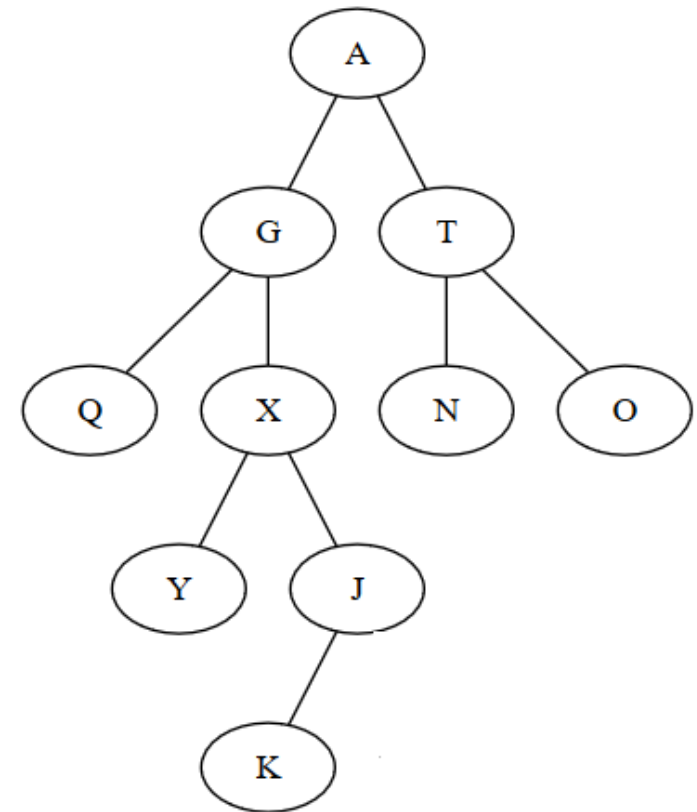
    if [currentNode].left  $\neq$  NIL then

      push(s, [currentNode].left)

    end-if

  end-while

end-subalgorithm



# Preorder traversal - properties

- Time complexity of the non-recursive traversal is  $\Theta(n)$ , and we also need  $O(n)$  extra space (the stack)
- Preorder traversal is exactly the same as depth first traversal (you can see it especially in the implementation), with the observation that here we need to be careful to first push the right child to the stack and then the left one (in case of depth-first traversal the order in which we pushed the children was not that important).

# Inorder traversal - non-recursive implementation

subalgorithm inorder(tree) is:

init(s)        //s is an auxiliary stack

currentNode  $\leftarrow$  tree.root

while currentNode  $\neq$  NIL execute

    push(s, currentNode)

    currentNode  $\leftarrow$  [currentNode].left

end-while

while not isEmpty(s) execute

    currentNode  $\leftarrow$  pop(s)

    @visit currentNode

    currentNode  $\leftarrow$  [currentNode].right

    while currentNode  $\neq$  NIL execute

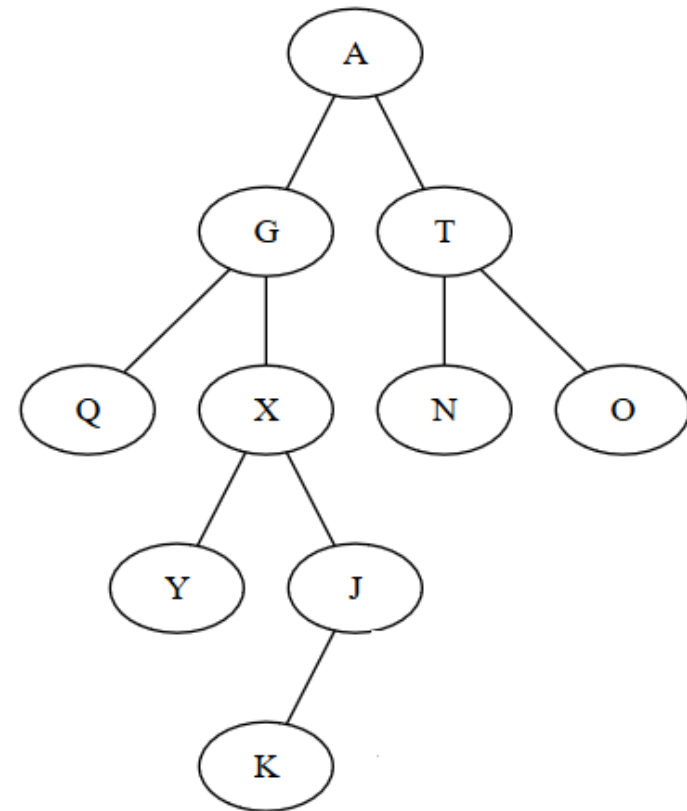
        push(s, currentNode)

        currentNode  $\leftarrow$  [currentNode].left

    end-while

end-while

end-subalgorithm



# Postorder traversal - non-recursive , 2 stacks

subalgorithm preorder(tree) is:

init(s1); init(s2)

if tree.root  $\neq$  NIL then

push(s1, tree.root)

end-if

while not isEmpty(s1) execute

currentNode  $\leftarrow$  pop(s1)

push(s2, currentNode)

if [currentNode].left  $\neq$  NIL then

push(s1, [currentNode].left)

end-if

if [currentNode].right  $\neq$  NIL then

push(s1, [currentNode].right)

end-if

end-while

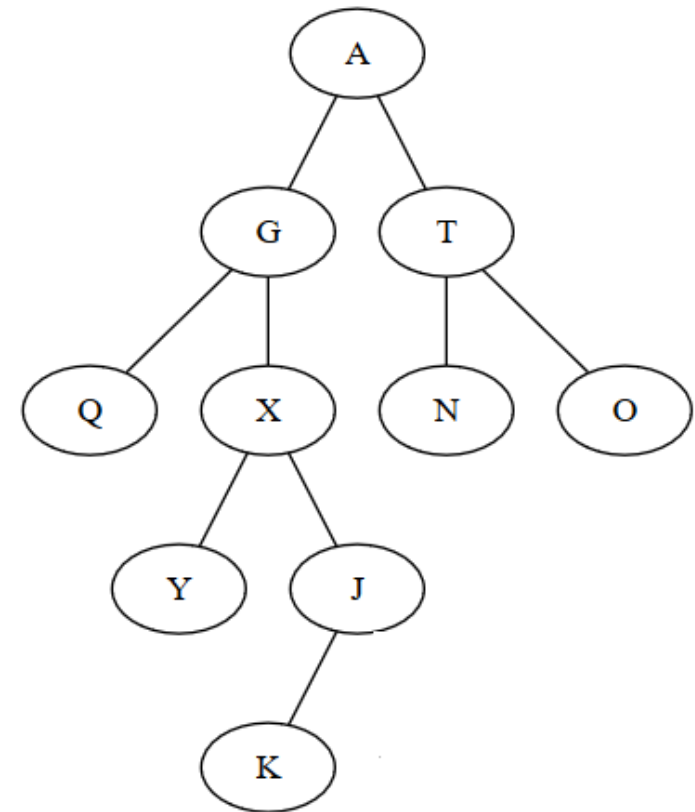
while not isEmpty(s2) execute

currentNode  $\leftarrow$  pop(s2)

@ visit currentNode

end-while

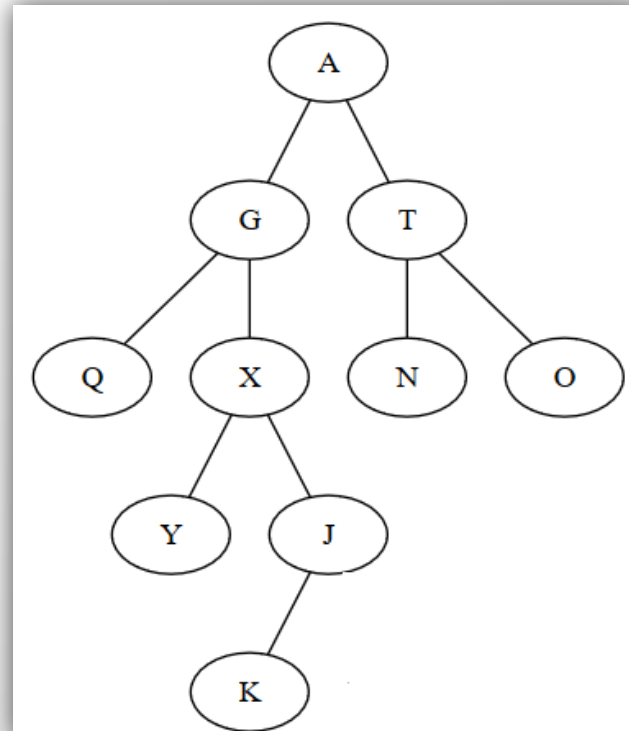
end-subalgorithm



subalgorithm postorder(tree) is:

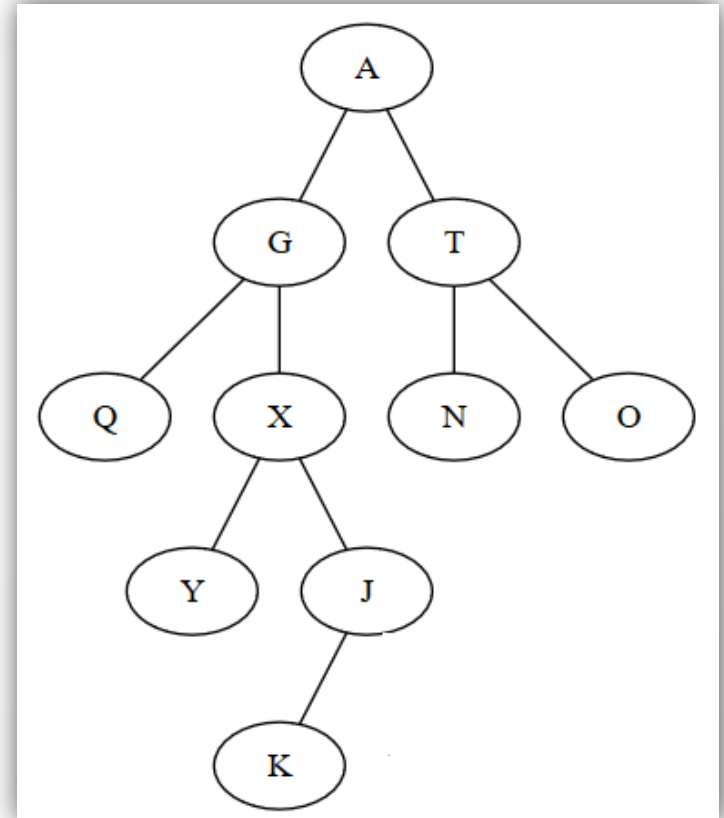
```
init(s)                                //s is an auxiliary stack
node ← tree.root
while node ≠ NIL execute
    if [node].right ≠ NIL then
        push(s, [node].right)
    end-if
    push(s, node)
    node ← [node].left
end-while
while not isEmpty(s) execute
    node ← pop(s)
    if [node].right ≠ NIL and (not isEmpty(s)) and
        [node].right = top(s) then
        pop(s)
        push(s, node)
        node ← [node].right
    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
```

## Postorder traversal with one stack



# Postorder traversal

- 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)





# 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 an information to show “the state” of a node

Use flag *visited*:

- 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

subalgorithm inorderNoStack(tree) is:

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

current  $\leftarrow$  [current].parent

end-if ... end-if

end-while

end-subalgorithm

# Binary tree iterator

- For defining an iterator, we have to divide the traversal code into the functions of an iterator: init, getCurrent, next, valid
- We are going to work with nodes without parent node.
  - The iterator will use a stack.

Iterator:

bt: BinaryTree

s: Stack

currentNode: ↑ BTNode

# Remember: Inorder traversal - non-recursive

subalgorithm inorder(tree) is:

init(s) *// s is an auxiliary stack*

currentNode  $\leftarrow$  tree.root

while currentNode  $\neq$  NIL execute

    push(s, currentNode)

    currentNode  $\leftarrow$  [currentNode].left

end-while

while not isEmpty(s) execute

    currentNode  $\leftarrow$  pop(s)

    @visit currentNode

    currentNode  $\leftarrow$  [currentNode].right

    while currentNode  $\neq$  NIL execute

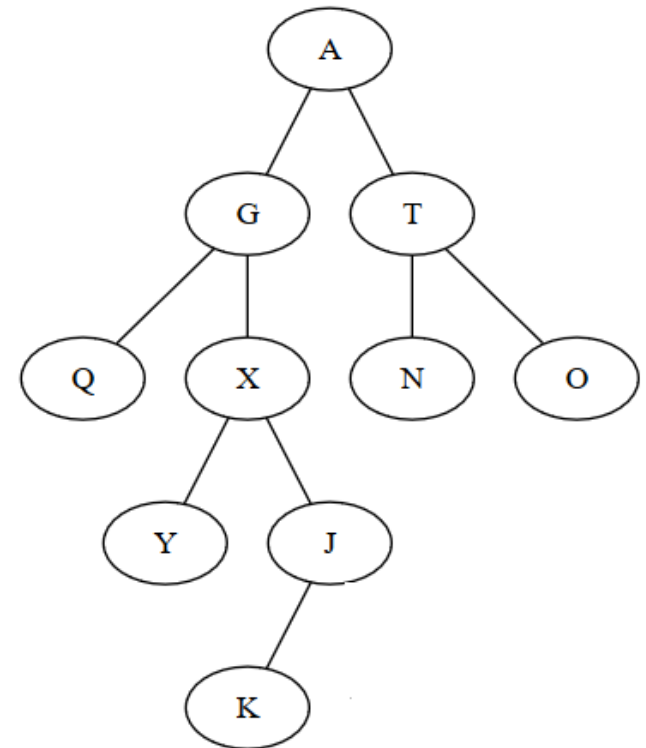
        push(s, currentNode)

        currentNode  $\leftarrow$  [currentNode].left

    end-while

end-while

end-subalgorithm



# Inorder binary tree iterator

subalgorithm init (it, bt) is:

```
    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 ← pop(it.s)
    else
        it.currentNode ← NIL
    end-if
end-subalgorithm
```

subalgorithm next(it) is:

```
    node ← [node].right
    while node ≠ NIL execute
        push(it.s, node)
        node ← [node].left
    end-while
    if not isEmpty(it.s) then
        it.currentNode ← pop(it.s)
    else
        it.currentNode ← NIL
    end-if
end-subalgorithm
```

function valid(it) is:

```
    valid ← ( it.currentNode ≠ NIL )
end-function
```

function getCurrent(it) is:

```
    getCurrent ← [it.currentNode].info
end-function
```

# Tree traversals: problems

## Think about:

- Assume you have a binary tree, 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.

e.g.:

Preorder: A B F G H E L M

Inorder: B G F H A L E M

- 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?

e.g.:

Pre: 1 2 3 4 5

Post: 5 4 3 2 1

or: 3 4 2 5 1

What if the tree is full?

# Tree traversals: problems

Rebuild the tree if you have the preorder and the postorder traversal of a tree with distinct elements. We also know that the tree is full.

- The first element in the preorder traversal is the ***root*** of the tree.
- We will find the ***index*** of ***left\_child*** in the postorder traversal.
- All the elements to the left of this ***index*** and element at this index will be in the left subtree of ***root***. And all the elements to the right of this index will be in the right subtree of the ***root***.

Now we have the preorder and postorder traversals for the 2 subtrees of the ***root***!

**e.g.:**

Preorder :            1, 2, 4, 5, 3, 6, 8, 9, 7

Postorder :           4, 5, 2, 8, 9, 6, 7, 3, 1

# Tree: other problems

## **Think about:**

Give the iterative and recursive algorithms for the following problems:

- Search for a given element in a binary tree.
- Determine the height of a binary tree.
- Determine the level at which a given element appears in a binary tree.
- Determine the parent of a node containing a given element .



# Tree traversals

When we have a tree that is not binary:

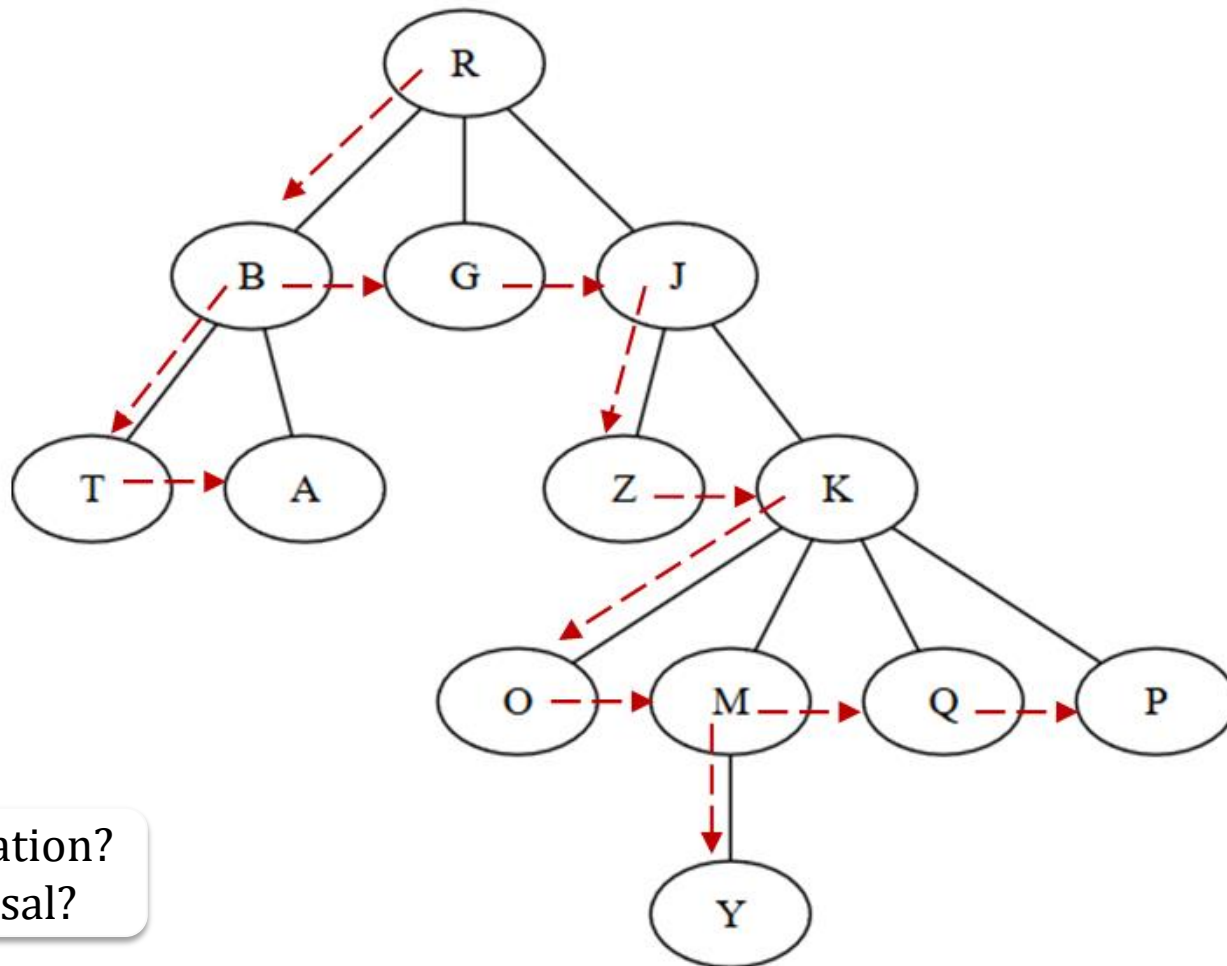
Level order (breadth first)

- use an auxiliary queue to store the nodes
  - as in case of a binary tree

Depth first

- use an auxiliary stack to store the nodes
  - it is the generalizations of preorder traversal

# K-ary trees



Representation?  
Tree traversal?

# K-ary trees

How can we represent a tree in which every node has at most  $k$  children?

- One option is to have a structure for a node that contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - 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: we occupy space for  $k$  children even if most nodes have less children.

- Another 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).

# Tree traversals

Traversing a tree means visiting all of its nodes.

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.).

Traversals for a k-ary tree:

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