

# DATA STRUCTURES (AND ALGORITHMS)

Trees. Binary trees.

---

*Lect. PhD. Diana-Lucia Miholca*

2023 - 2024



Babeş - Bolyai University

Faculty of Mathematics and Computer Science

- Hash tables
  - Coalesced chaining
  - Open addressing

- Trees
  - Terminology
  - Binary Trees



Trees represent one of the most commonly used data structures.



In graph theory, a **tree** is a connected, acyclic and usually undirected graph.




When talking about trees as data structures, we actually mean **rooted trees** in which one node is designated to be the *root* of the tree.

Trees are among the most commonly used data structures.

In the graph theory, a tree is defined as a connected, acyclic **acyclic** graph **graaf**, which is usually undirected **undirected**.

When talking about trees as data structure, we actually mean rooted trees, which are the trees having one node designated to be the root of the tree.

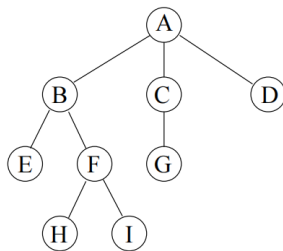
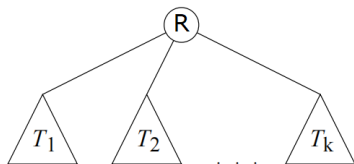
 A **tree** is a finite set  $\mathcal{T}$  of 0 or more elements, called *nodes*, with the following properties:

- 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*,  $T_1, T_2, \dots, T_k$ , the root node  $R$  being linked by an edge to the root of each of these trees. The trees  $T_1, T_2, \dots, T_k$  are called the *subtrees* of  $R$ .


Formally, a tree (actually a rooted tree) is defined recursively as follows. It is a finite set  $T$  of 0 or more elements called nodes.

If the set  $T$  is empty then the tree is empty. Otherwise, there is a distinguished node  $R$  called the root of the tree, the rest of the nodes being divided into  $k$  disjunct **disjunct** subsets of nodes, denoted  $T_1, T_2, \dots, T_k$ , where each is itself a tree. The root  $R$  of the tree is linked by an edge to the root of each of these trees. These trees are called subtrees of the root.

## Tree - Visualization



Trees are typically drawn like graphs, where there is an edge (usually undirected) from a node to each of its children. The two figures on the current slide exemplify this. The left side image / The one on the left is generic **djăneruic**. There we have the root  $R$  **ar** and the  $k$  subtrees of the root. The drawing on the right, however, offers a concrete example.

 An **ordered tree** is a tree in which the order of the children is well defined and relevant.

Let's discuss some terminology associated with tree data structure.

If there is an order among the  $T_i$ 's, so among the children of a node, then we say that the tree is an ordered tree. This means that instead of having a set of children for each node, we have a list of children.

The degree of a node in a tree is the number of children it has.

The leaf nodes in the tree are the nodes with no children, the nodes having the degree 0.

The others, having at least one child are called internal nodes.

**D** An **ordered tree** is a tree in which the order of the children is well defined and relevant.

**D** The **degree** of a node is defined as the number of its children.

Let's discuss some terminology associated with tree data structure.

If there is an order among the  $T_i$ 's, so among the children of a node, then we say that the tree is an ordered tree. This means that instead of having a set of children for each node, we have a list of children.

The degree of a node in a tree is the number of children it has.

The leaf nodes in the tree are the nodes with no children, the nodes having the degree 0.

The others, having at least one child are called internal nodes.



**D** An **ordered tree** is a tree in which the order of the children is well defined and relevant.

**D** The **degree** of a node is defined as the number of its children.

**D** The nodes with the degree 0 are called **leaf nodes**.

Let's discuss some terminology associated with tree data structure.

If there is an order among the  $T_i$ 's, so among the children of a node, then we say that the tree is an ordered tree. This means that instead of having a set of children for each node, we have a list of children.

The degree of a node in a tree is the number of children it has.

The leaf nodes in the tree are the nodes with no children, the nodes having the degree 0.

The others, having at least one child are called internal nodes.

**D** An **ordered tree** is a tree in which the order of the children is well defined and relevant.

**D** The **degree** of a node is defined as the number of its children.

**D** The nodes with the degree 0 are called **leaf nodes**.

**D** The nodes that are not leaf nodes are called **internal nodes**.


Let's discuss some terminology associated with tree data structure.

If there is an order among the  $T_i$ 's, so among the children of a node, then we say that the tree is an ordered tree. This means that instead of having a set of children for each node, we have a list of children.

The degree of a node in a tree is the number of children it has.

The leaf nodes in the tree are the nodes with no children, the nodes having the degree 0.

The others, having at least one child are called internal nodes.

 The **depth** or **level** of a node is the length of the unique path (measured as the number of edges on the path) from the root to the node.




The root of the tree is at level

The depth or level of a node in the tree is the length of the unique path from the root to that node. The length of a path is the number of edges on the path. Note that the root of the tree is at level 0, which also means that it is at depth zero.

The height of a node is the length of the longest path from the node to a leaf. Thus all leaves are at height 0.

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.

 The **depth** or **level** of a node is the length of the unique path (measured as the number of edges on the path) from the root to the node.



The root of the tree is at level 0 (and has depth 0).

The depth or level of a node in the tree is the length of the unique path from the root to that node. The length of a path is the number of edges on the path. Note that the root of the tree is at level 0, which also means that it is at depth zero.

The height of a node is the length of the longest path from the node to a leaf. Thus all leaves are at height 0.

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.

**D** The **depth** or **level** of a node is the length of the unique path (measured as the number of edges on the path) from the root to the node.



The root of the tree is at level 0 (and has depth 0).

**D** The **height** of a node is the length of the longest path from the node to a leaf.



All leaves are at height

The depth or level of a node in the tree is the length of the unique path from the root to that node. The length of a path is the number of edges on the path. Note that the root of the tree is at level 0, which also means that it is at depth zero.

The height of a node is the length of the longest path from the node to a leaf. Thus all leaves are at height 0.

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.

**D** The **depth** or **level** of a node is the length of the unique path (measured as the number of edges on the path) from the root to the node.



The root of the tree is at level 0 (and has depth 0).

**D** The **height** of a node is the length of the longest path from the node to a leaf.



All leaves are at height 0.

The depth or level of a node in the tree is the length of the unique path from the root to that node. The length of a path is the number of edges on the path. Note that the root of the tree is at level 0, which also means that it is at depth zero.

The height of a node is the length of the longest path from the node to a leaf. Thus all leaves are at height 0.

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.

**D** The **depth** or **level** of a node is the length of the unique path (measured as the number of edges on the path) from the root to the node.



The root of the tree is at level 0 (and has depth 0).

**D** The **height** of a node is the length of the longest path from the node to a leaf.



All leaves are at height 0.

**D** 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.

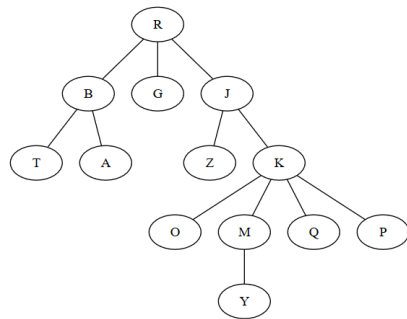
The depth or level of a node in the tree is the length of the unique path from the root to that node. The length of a path is the number of edges on the path. Note that the root of the tree is at level 0, which also means that it is at depth zero.

The height of a node is the length of the longest path from the node to a leaf. Thus all leaves are at height 0.

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:



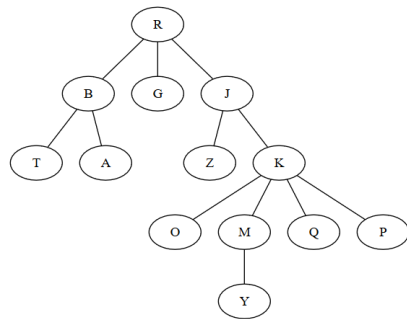
Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.



## Tree - Terminology - Example

- Root of the tree: *R*
- Children of *R*:

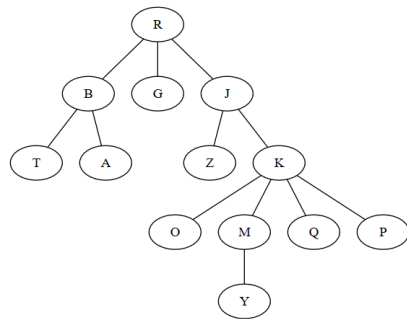


Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.

## Tree - Terminology - Example

- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*:

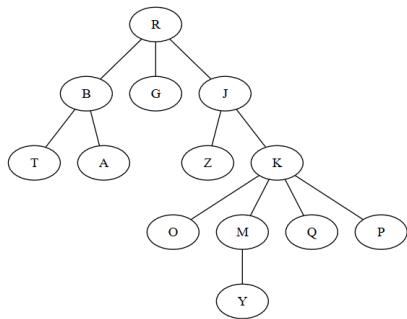


Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.

## Tree - Terminology - Example

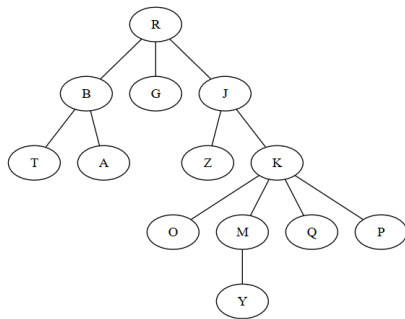
- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes:



Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.

## Tree - Terminology - Example

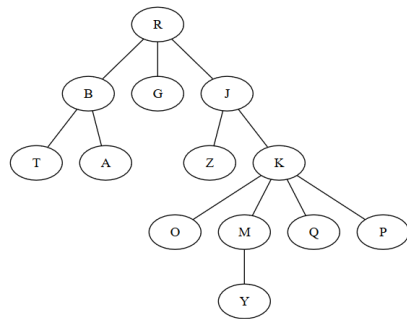


- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes:

Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.

## Tree - Terminology - Example

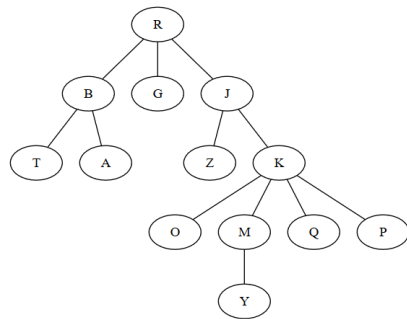


- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node *K*:

Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.

## Tree - Terminology - Example

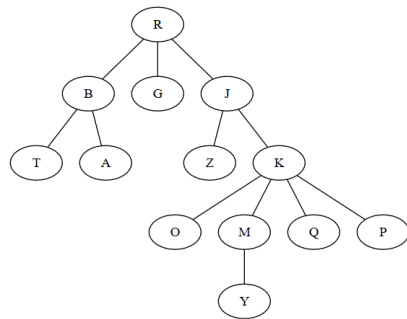


- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node *K*: 2 (path R-J-K)
- Height of node *K*:

Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.

## Tree - Terminology - Example

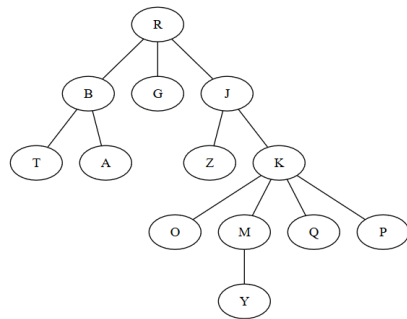


- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node *K*: 2 (path R-J-K)
- Height of node *K*: 2 (path K-M-Y)
- Height of the tree (height of node *R*):

Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.

## Tree - Terminology - Example



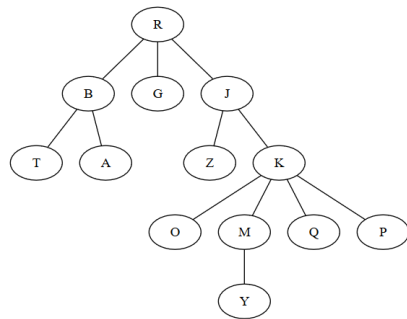
- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node *K*: 2 (path R-J-K)
- Height of node *K*: 2 (path K-M-Y)
- Height of the tree (height of node *R*): 4
- Nodes on level 2:

Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.




## 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

Here we have an example for practicing the terminology.

The root of the tree is the node containing R. Its children are the nodes on level 1, containing B, G **gi** and J **gei**. The parent of the node containing M is the node containing K **key**. The leaves are: T, A, Z **zi**, O **o**, Y **wai**, Q **chiu** and P **pi**. The internal nodes are all the others, namely R, B, J **gei**, K **key** and M. The node containing K is at depth 2 (the unique path from the root to K contains two edges) and weight 2, the length of the longest path from K to a root being 2. The height of the exemplified tree is the height of the root R, which is four. When it comes to the nodes on a given level, for level two, for instance, we have the nodes T, A, Z and K.

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

 One option is to have a structure *node* with the following:

- information from the node
- address of the parent node (not mandatory)
- $k$  fields, one for each (possible) child

 Obs: this is doable if  $k$  is not too large


One difficulty with representing general trees is that since there is no bound on the number of children a node can have, there is no obvious bound on the size of a given node. However, if we know that each node in the tree can have at most  $k$  children, then the tree is called  $k$ -ary (cheiri) and we could think about possible representations of such a tree.

One option is to have a structure for a node which contains the following: the information from the node, optionally the address of the parent node and  $k$  fields, one for each possible child.

As an observation, this is doable if  $k$  is not too large.

 Another option is to have a structure *node* with the following:

- information from the node
- address of the parent node (not mandatory)
- an array of length  $k$ , in which each element is the address of a child
- number of children (number of occupied positions from the above array)

 The disadvantage of these approaches is that we occupy space for  $k$  children even if most nodes have less children.

A second way of representing a  $k$ -ary tree is to have, again, the structure node, but with an array of length  $k$ , whose elements are the addresses of the  $k$  children and, in another field, the number of children, which is also the number of occupied positions from the array. Of course, these are additional to the actual information from the node and the optional field storing the address of the parent node.

The disadvantage of these two representation options is that we occupy space for  $k$  children even if most nodes are probably to have less children.

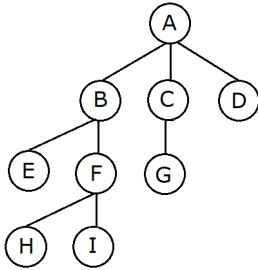


A third option is the so-called *left-child right-sibling* representation in which we have a structure for a node which contains the following:

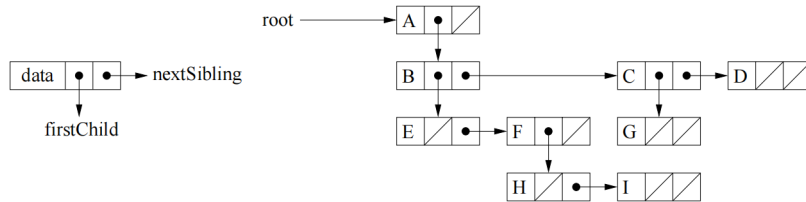
- information from the node
- address of the parent node (not mandatory)
- address of the leftmost child of the node
- address of the right sibling of the node (next node on the same level from the same parent).

A third option is called left -child right-sibling and suppose to store to store in the node structure the following: the information, optionally the address of the parent node, the address of the leftmost child and the address of the right sibling of the node, which is the next node on the same level and for the same parent.


## Left-child right sibling representation - Example



Here's an example for left-child right-sibling representation.



 *Traversing a tree* means visiting all of its nodes.

 A node is said to be *visited* when the program control arrives at the node, usually with the purpose of performing some operation on it.

 For a k-ary tree there are 2 possible traversals:

- Depth-first traversal
- Breadth-first traversal (level order traversal)

Traversing a tree means visiting or traversing all the nodes in the tree.

A node of a tree is said to be *visited* when the program control **proguam căncîrol** arrives at the node, usually with the purpose of performing some operation on the node, like printing it, checking the value from the node, etc.

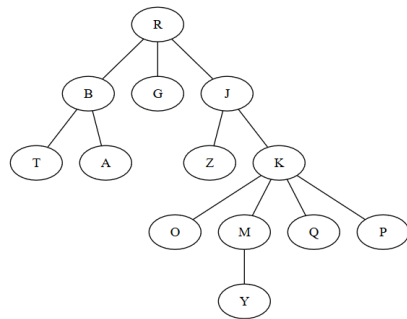
For a k-ary tree, there are 2 possible traversals: depth-first traversal and breadth-first traversal, also called level order traversal.

- ▶ Traversal starts with visiting the root of the tree
- ▶ 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 in the traversal.

In the depth first traversal, we start from the root and then we visit one of the children of the root, then one child of that visited child and so on. We go down, or in depth, as much as possible, and continue with other children of a node only after all descendants of the first child were visited.

As an observation, for the depth-first traversal, we use a stack to remember the nodes in the traversal.

## Depth-first traversal - Example

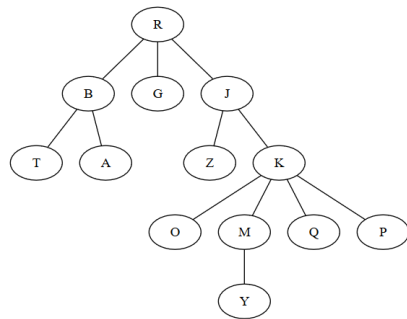


Let's see which are the first node visited during a depth-first traversal for the tree on the current slide.

Obviously, we start from the root. So, we push the root on the stack. Then we visit the root, the node containing R, by pop-ing the only element currently in the stack. Next, we push its children, which are B, G and J. Again, we pop from the stack and visit that node, so we visit B and push its children. Now, the stack contains T, A, G and J. So, we visit T and push nothing. The stack remains with A, G and J. We visit A and push nothing, then G and, again, we have nothing to push since G is a leaf. Currently, the stack contains only J. So we visit it and push its children, which are Z and K. We will continue in the same way until all nodes of the tree have been visited.



## Depth-first traversal - Example



- ▶ Stack  $s$  with the root:  $[R]$
- ▶ Visit (pop from stack)  $R$  and push its children (in the reverse order):  $s = [J\ G\ B]$
- ▶ Pop & visit  $B$  and push its children:  $s = [J\ G\ A\ T]$
- ▶ Pop & visit  $T$  and push nothing:  $s = [J\ G\ A]$
- ▶ Visit  $A$  and push nothing:  $s = [J\ G]$
- ▶ Visit  $G$  and push nothing:  $s = [J]$
- ▶ Visit  $J$  and push its children:  $s = [K\ Z]$
- ▶ etc...

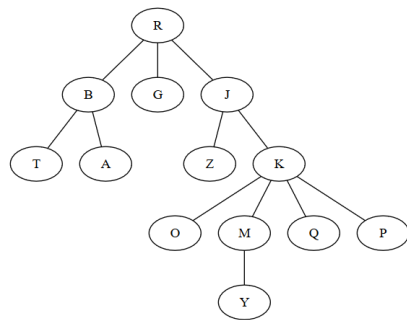
Let's see which are the first node visited during a depth-first traversal for the tree on the current slide.

Obviously, we start from the root. So, we push the root on the stack. Then we visit the root, the node containing  $R$ , by pop-ing the only element currently in the stack. Next, we push its children, which are  $B$ ,  $G$  and  $J$ . Again, we pop from pop from the stack and visit that node, so we visit  $B$  and push its children. Now, the stack contains  $T$ ,  $A$ ,  $G$  and  $J$ . So, we visit  $T$  and push nothing. The stack remains with  $A$ ,  $G$  and  $J$ . We visit  $A$  and push nothing, then  $G$  and, again, we have nothing to push since  $G$  is a leaf. Currently, the stack contains only  $J$ . So we visit it and push its children, which are  $Z$  and  $K$ . We will continue in the same way until all nodes of the tree have been visited.

- ▶ Traversal starts with visiting the root of the tree
- ▶ We visit all children of the root (one by one) and once all of them have been visited we go to their children and so on.
  - ✎ We go down one level, only when all nodes from a level have been visited.
- ✎ For level order traversal we use a queue to remember the nodes that have to be visited.

In the breath-first or level order search, the traversal also starts from the root of the tree. Then, we visit all children of the root, one by one, and once all of them have been visited, we continue by visiting their children and so on. We go down one level, only when all nodes from a level were visited. So, we go down to the next level, only when all nodes from the current level have been visited. Here, for level order traversal, we use a queue (instead of a stack) in order to remember the nodes that have to be visited.

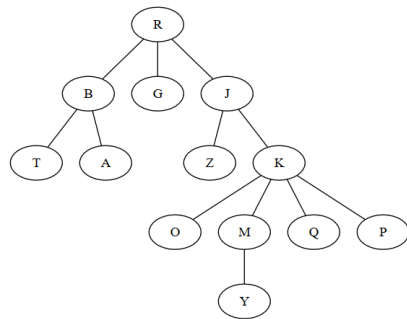
## Level-order traversal - Example



Let's reconsider our example and see which are the first nodes visited during a breath-first traversal, this time.

Again, we start by pushing root in the auxiliary queue. Then we perform a pop from the queue and visit the node containing R. Next, we push its children, which are B, G and J. Again, we pop from the queue and visit B pushing its children. Now, the queue contains G, J, T and A. So, we visit G and push nothing. The queue remains with J, T and A. We visit J and push its children, then J and, again, we push its children. Currently, the queue contains T, A, Z and K. So we pop and visit T and push nothing, since T has no children, being a leaf. We will continue in the same manner until the entire tree has been traversed.


## 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...

Let's reconsider our example and see which are the first nodes visited during a breath-first traversal, this time.

Again, we start by pushing root in the auxiliary queue. Then we perform a pop from the queue and visit the node containing  $R$ . Next, we push its children, which are  $B$ ,  $G$  and  $J$ . Again, we pop from the queue and visit  $B$  pushing its children. Now, the queue contains  $G$ ,  $J$ ,  $T$  and  $A$ . So, we visit  $G$  and push nothing. The queue remains with  $J$ ,  $T$  and  $A$ . We visit  $J$  and push its children, then  $J$  and, again, we push its children. Currently, the queue contains  $T$ ,  $A$ ,  $Z$  and  $K$ . So we pop and visit  $T$  and push nothing, since  $T$  has no children, being a leaf. We will continue in the same manner until the entire tree has been traversed.

 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**.

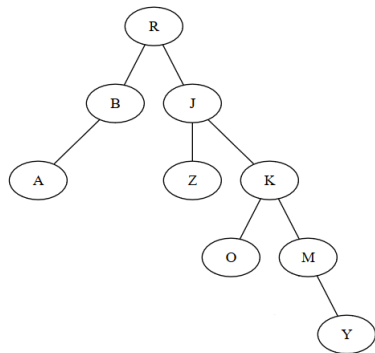
 Even if a node has only one child, we still have to know whether that is the left or the right one.

What's a binary tree? A binary tree is an ordered tree in which each node has at most two children / in which each node has no more than two children. Remember that an ordered tree is a tree in which the order of the children is important.

In case of a binary tree, therefore, the two children are called left child and right child and it is important which is which.

This means that even if a node has only one child, we still have to know whether that only child is the left or the right one.

## Binary tree - Example



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

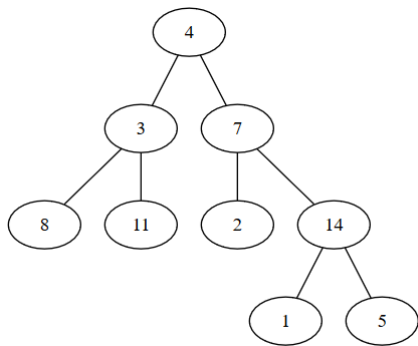
Here's an example of binary tree. For instance, *A* is the left child of *B*, while *Y uai* is the right child of *M*.

The left child of a node is drawn beneath the node and to the left, while the right child is drawn beneath and to the right.

## Binary tree - Terminology



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

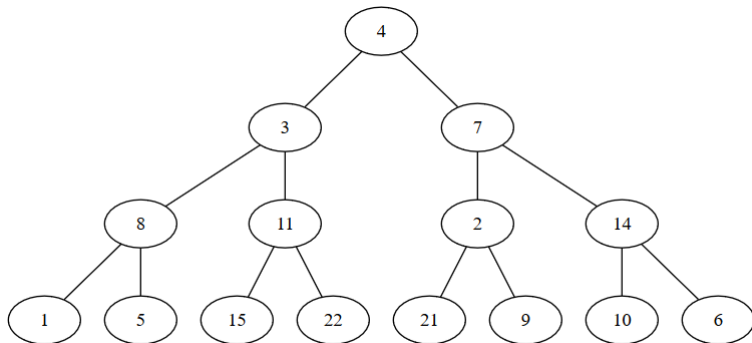


In the following, we will define and exemplify some special types of binary trees.

A binary tree in which every internal node (that is every non-leaf node) has exactly two children is called a full binary tree. Note that for the provided example, nodes have either two children or no children.

## Binary tree - Terminology

**D** A binary tree is called **complete** if every level of the tree is completely filled.

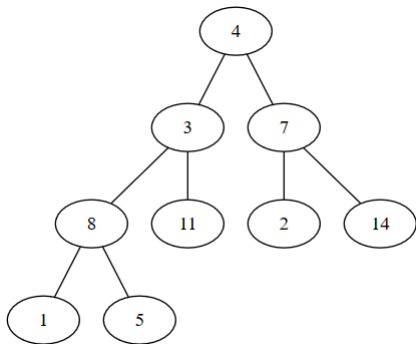


A complete is a binary tree in which every level of the tree is completely filled, which entails that all the leaves are on the same level and all internal nodes have exactly 2 children.



## Binary tree - Terminology

**D** A binary tree is called **almost complete** if every level of the tree is completely filled, except possibly the bottom level, which is filled from left to right.

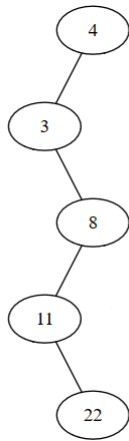


We also have almost complete trees. An almost complete tree is defined as a tree in which every level is completely filled, except possibly the bottom level, which is filled from left to right.

We can observe in the exemplified tree that the first tree levels are completely filled, while the last level, has only the first two possible nodes.

## Binary tree - Terminology

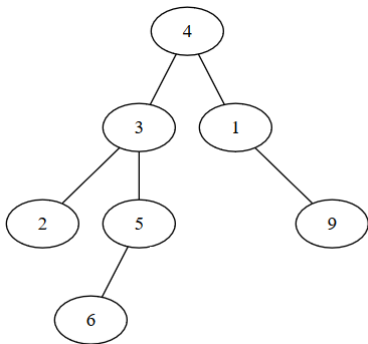
**D** A binary tree is called **degenerated** if every internal node has exactly one child (it is actually a chain of nodes).



A binary tree is called degenerated **digenăreitid** if every internal node has exactly one child. Therefore, such a binary tree is actually a chain of nodes, like the one on the current slide.

## Binary tree - Terminology

**D** 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.



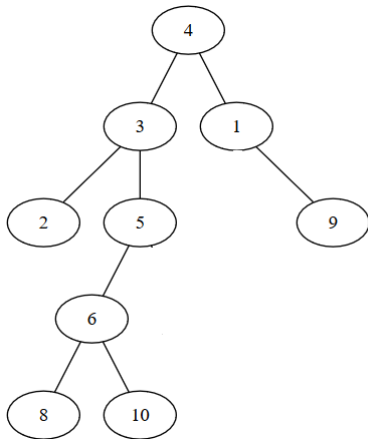
At the opposite pole lies the balanced binary tree in which the difference between the height of the left and right subtrees is at most 1 for every node of the tree.

We can see in the current example that the heights for the left and right children of every node differ by at most one.

## Binary tree - Terminology



There are many binary trees that are none of the above categories, for example:



Obviously, there are many binary trees that are none of the above categories, like this one.

🔍 How many edges are there in a binary tree with  $n$  nodes?

Here are some numerical properties of binary trees.

The first one is that a binary tree with  $n$  nodes has exactly  $n-1$  edges. This is because, excepting the root, every node is linked by an edge to its parent.

When it comes to a complete binary tree, which is a tree where every level is completely filled, the total number of nodes is  $2^n - 1$ , minus one, since we have 1 node on the first level, two nodes on the second one, four nodes on the third one and so on. On the last level we have  $2^n$  nodes, so, by summing up these powers of two, we obtain the total number of nodes.

❓ How many edges are there in a binary tree with  $n$  nodes?


📖 A binary tree with  $n$  nodes has exactly  $n - 1$  edges.

- This is true for every tree, not just binary trees

Here are some numerical properties of binary trees.


The first one is that a binary tree with  $n$  nodes has exactly  $n-1$  edges. This is because, excepting the root, every node is linked by an edge to its parent.

When it comes to a complete binary tree, which is a tree where every level is completely filled, the total number of nodes is  $2^n - 1$ , minus one, since we have 1 node on the first level, two nodes on the second one, four nodes on the third one and so on. On the last level we have  $2^n$  nodes, so, by summing up these powers of two, we obtain the total number of nodes.

 How many edges are there in a binary tree with  $n$  nodes?

 A binary tree with  $n$  nodes has exactly  $n - 1$  edges.

- This is true for every tree, not just binary trees

 How many nodes are there in a complete binary tree of height  $N$ ?

Here are some numerical properties of binary trees.

The first one is that a binary tree with  $n$  nodes has exactly  $n-1$  edges. This is because, excepting the root, every node is linked by an edge to its parent.

When it comes to a complete binary tree, which is a tree where every level is completely filled, the total number of nodes is  $2^n - 1$ , minus one, since we have 1 node on the first level, two nodes on the second one, four nodes on the third one and so on. On the last level we have  $2^n$  nodes, so, by summing up these powers of two, we obtain the total number of nodes.

? How many edges are there in a binary tree with  $n$  nodes?

§ A binary tree with  $n$  nodes has exactly  $n - 1$  edges.

- This is true for every tree, not just binary trees

? How many nodes are there in a complete binary tree of height  $N$ ?

§ The number of nodes in a complete binary tree of height  $N$  is  $2^{N+1} - 1$  ( $1 + 2 + 4 + 8 + \dots + 2^N$ )

Here are some numerical properties of binary trees.

The first one is that a binary tree with  $n$  nodes has exactly  $n-1$  edges. This is because, excepting the root, every node is linked by an edge to its parent.

When it comes to a complete binary tree, which is a tree where every level is completely filled, the total number of nodes is 2 at power  $n$  minus 1, minus one, since we have 1 node on the first level, two nodes on the second one, four nodes on the third one and so on. On the last level we have 2 at power  $n$  nodes, so, by summing up these powers of two, we obtain the total number of nodes.



❓ What is the maximum number of nodes in a binary tree of height  $N$ ?

Therefore,  $2^{n-1}$  is also the maximum number of nodes in a binary tree of height  $n$ .

The minimum number of nodes in such a binary tree is  $N+1$ , when the tree is degenerated.

A random binary tree with  $n$  nodes has the height between integer part of logarithm of  $N+1$  to base 2 and  $N-1$ . Obviously, the minimum height is achieved when the tree is complete, while the maximum height is for a degenerated tree.

? What is the maximum number of nodes in a binary tree of height  $N$ ?

§ The maximum number of nodes in a binary tree of height  $N$  is  $2^{N+1} - 1$  - if the tree is complete.

Therefore,  $2$  at power  $n-1$  one  $-1$  is also the maximum number of nodes in a binary tree of height  $n$ .

The minimum number of nodes in such a binary tree is  $N+1$ , when the tree is degenerated.

A random binary tree with  $n$  nodes has the height between integer part of logarithm of  $N+1$  to base  $2$  and  $N-1$ . Obviously, the minimum height is achieved when the tree is complete, while the maximum height is for a degenerated tree.

? What is the maximum number of nodes in a binary tree of height  $N$ ?

§ The maximum number of nodes in a binary tree of height  $N$  is  $2^{N+1} - 1$  - if the tree is complete.

? What is the minimum number of nodes in a binary tree of height  $N$ ?

Therefore,  $2$  at power  $n-1$  one  $-1$  is also the maximum number of nodes in a binary tree of height  $n$ .

The minimum number of nodes in such a binary tree is  $N+1$ , when the tree is degenerated.

A random binary tree with  $n$  nodes has the height between integer part of logarithm of  $N+1$  to base  $2$  and  $N-1$ . Obviously, the minimum height is achieved when the tree is complete, while the maximum height is for a degenerated tree.

? What is the maximum number of nodes in a binary tree of height  $N$ ?

§ The maximum number of nodes in a binary tree of height  $N$  is  $2^{N+1} - 1$  - if the tree is complete.

? What is the minimum number of nodes in a binary tree of height  $N$ ?

§ The minimum number of nodes in a binary tree of height  $N$  is  $N + 1$  - if the tree is degenerated.

Therefore,  $2$  at power  $n-1$  one  $-1$  is also the maximum number of nodes in a binary tree of height  $n$ .

The minimum number of nodes in such a binary tree is  $N+1$ , when the tree is degenerated.

A random binary tree with  $n$  nodes has the height between integer part of logarithm of  $N+1$  to base  $2$  and  $N-1$ . Obviously, the minimum height is achieved when the tree is complete, while the maximum height is for a degenerated tree.

? What is the maximum number of nodes in a binary tree of height  $N$ ?

§ The maximum number of nodes in a binary tree of height  $N$  is  $2^{N+1} - 1$  - if the tree is complete.

? What is the minimum number of nodes in a binary tree of height  $N$ ?

§ The minimum number of nodes in a binary tree of height  $N$  is  $N + 1$  - if the tree is degenerated.

§ A binary tree with  $N$  nodes has a height between  $\lceil \log_2 N \rceil$  and  $N - 1$ .

Therefore,  $2^{N+1} - 1$  is also the maximum number of nodes in a binary tree of height  $n$ .

The minimum number of nodes in such a binary tree is  $N+1$ , when the tree is degenerated.

A random binary tree with  $n$  nodes has the height between integer part of logarithm of  $N+1$  to base 2 and  $N-1$ . Obviously, the minimum height is achieved when the tree is complete, while the maximum height is for a degenerated tree.

- Domain of ADT Binary Tree:

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

When it comes to the abstract data type binary tree, its domain can be defined as the set of all the element bi-ti, where bi-ti is a binary tree consisting of nodes that contain information of type TElem.

- `init(bt)`
  - **descr:** creates a new, empty binary tree
  - **pre:** true
  - **post:**  $bt \in \mathcal{BT}$ , *bt* is an empty binary tree

The first operation we expect to have in the interface is the constructor, which creates a new, empty binary tree and has no pre-conditions. The postconditions ensues that the result returned via the only parameter is of type binary tree and, 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*

Another possible constructor is the one which creates an one-node binary tree. So, this constructor, named `initLeaf` takes an element as an input and return via its first parameter the created binary tree. The precondition impose that the given element is of type `TElem`, while the postconditions ensures that *bt* is of type binary tree, *bt* having only one node, which is its root and contains the element *e* which was given as input.



- `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  $e$

A third constructor is one that receives an element but also two trees and created a new binary tree having the given element in the root and the two given trees as children.

The preconditions are that  $e$  is of type  $TElem$ , while  $left$  and  $right$  are of type binary tree.

As postconditions, we have that the output  $bt$  is of type binary tree, being a tree with the root conining the element  $e$  and the two subtrees being  $left$  and  $right$ .

- `insertLeftSubtree(bt, left)`
  - **descr:** sets the left subtree of a binary tree (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  $left$

We also have setters for the left and right subtrees of a given tree.

The operation `insertLeftSubtree` sets the left tree of a given binary tree to another given tree. So, both parameters have to be of type Binary Tree, the left subtree of  $bt$  prime, which is  $bt$  in its new state, after applying the operation, being equal to  $left$ .

- `insertRightSubtree(bt, right)`
  - **descr:** sets the right subtree of a binary tree (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  $right$

We have a similar operation, called `insertLeftSubtree` for setting the left subtree of a given tree. The only difference is that the second parameter is a tree which become the left, and not the right, subtree of the given tree.

- `root(bt)`
  - **descr:** returns the information from the root of a binary tree
  - **pre:**  $bt \in \mathcal{BT}$ ,  $bt \neq \Phi$
  - **post:**  $root = e$ ,  $e \in TElem$ ,  $e$  is the information from the root of  $bt$
  - **throws:** an exception if  $bt$  is empty

We also have getter operations. For instance the function `root` returns, given a binary tree, the information in the root of the tree. As preconditions, we have that  $bt$  is of type binary tree and should not be empty, in order to have a root. As postconditions, we have that the result, returned via the function's name is an  $e$  of type  $TElem$ ,  $e$  being the information in the root node of the given tree. If the given tree is empty, then an exception is thrown.

- `left(bt)`
  - **descr:** returns the left subtree of a binary tree
  - **pre:**  $bt \in \mathcal{BT}$ ,  $bt \neq \Phi$
  - **post:**  $left = l$ ,  $l \in \mathcal{BT}$ ,  $l$  is the left subtree of  $bt$
  - **throws:** an exception if  $bt$  is empty

For obtaining the left subtree of a tree, we have the `left` function. Given a binary tree `bt`, which, according to the preconditions, has to be non-empty, returns its left subtree. Therefore, the postconditions ensure that, if the preconditions are met, then the result is a binary tree `l` that is the left subtree of `bt`. Otherwise, if the preconditions are violated, then an exception is thrown.

- `right(bt)`
  - **descr:** returns the right subtree of a binary tree
  - **pre:**  $bt \in \mathcal{BT}$ ,  $bt \neq \Phi$
  - **post:**  $right = r$ ,  $r \in \mathcal{BT}$ ,  $r$  is the right subtree of  $bt$
  - **throws:** an exception if  $bt$  is empty

Here are the specifications for a similar function, but which returns the right subtree instead of the left subtree.

- `isEmpty(bt)`

- **descr:** checks if a binary tree is empty
- **pre:**  $bt \in \mathcal{BT}$
- **post:**

$$empty = \begin{cases} True, & \text{if } bt = \Phi \\ False, & \text{otherwise} \end{cases}$$

The functions `isEmpty`, `check`, as the name suggests, if a given binary tree is empty or not. So, the function receives as its only parameter a binary tree `bt` and return via its name a boolean value, which is true, if `bt` is empty and false, otherwise.

- **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  $traversal$

We also have an operation which provides a traversal mechanism, for visiting all the nodes in a given binary tree.

This operation takes as input parameters a binary tree and another parameter named "traversal" which indicates the order in which we want the nodes to be visited. As output parameter we have  $i$ , which will be the iterator.



- `destroy(bt)`
  - **descr:** destorys a binary tree
  - **pre:**  $bt \in \mathcal{BT}$
  - **post:**  $bt$  was destroyed

The last operation that is necessary in a minimal interface for a binary tree is the destructor. It simply destroys a binary tree by freeing the memory occupied by it.



## Other possible operations:

- changing the information from the root of a binary tree
- removing a subtree (left or right) of a binary tree
- searching for an element in a binary tree
- returning the number of elements from a binary tree

There are also other possible operations that can extend this interface, such as:

- changing the information from the root node of a binary tree, removing the left or right subtree, searching for a given element or returning the total number of elements in a binary tree,



We have several options for representing binary trees:

- Representation using an array
- Linked representation
  - with dynamic allocation
  - on array

When it comes to representing a tree in a computer's memory, we have several options:

A first one is to represent it using an array - we'll see in a moment how.

Alternatively, we have the linked representation, which can be dynamically allocated or on array.



### Representation using an array:

- The root of the tree is at the first index
- The left child of the node at index  $i$  is at index  $2 * i$ , while the right child is at index  $2 * i + 1$ .
- The parent of the node at index  $i > 1$  is at index  $[i/2]$



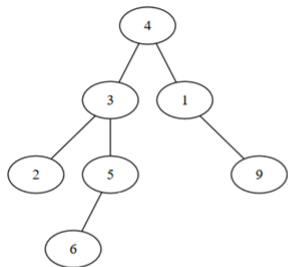
Some special value is needed to denote that a slot is empty.

When representing a binary tree using an array, the elements are, obviously, stored in that array and:

- the first element is the element in the root of the tree
- for the node at index  $i$ , the left child is at index  $2$  times  $i$ , while the right child is at index  $2$  times  $i$  plus  $1$  (if they exist)
- for the element at index  $i$ , where  $i$  is greater than  $1$ , - to except the root - its parent is at index integer part of  $i$  over  $2$

Please note that, unlike in the case of a heap, which is an almost complete binary tree, for a random binary, we will have empty slots in the array, corresponding to missing nodes. Therefore, we need some special value to denote that a slot is empty.

## Possible representations - Array-based



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
...	...

Here we have an example. Note that we marked with -1 the empty slots. For instance, 1, being the right child of the root is on index 3, which is 2 times 1 + one. Its left child should be on index 6, but at index 6 we have -1 one, since the left child is missing. The right child of 1, instead, is nine and we can find it at index seven.

The main disadvantage of this representation is that we might have numerous empty slots and thus waste a lot of memory space. This actually depends on the form of the tree.



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

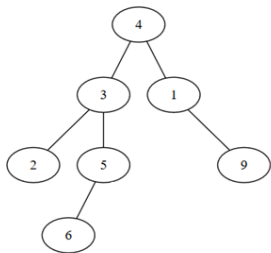


### Linked representation on an array:

- There are arrays storing:
  - the information from the nodes
  - the index of the left child
  - the index of the right child
  - optionally, the index of the parent

When opting for a linked representation, but on an array, the information from the nodes and the addresses of the left child, right child and, optionally, the address of the parent are all stored in arrays. The addresses are actually indexes.

## Possible representations - Linked on array



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 index of the root



Here is 1, but it could be any other



If the array is full, we have to resize it

Here is an example.

For this example, the root is at index 1. It could be at any other index, but we need to know its index.

We can observe that the left child of the root is at index 2, where we find the element tree, while the right child is at index six, where we find the element 1.

Since the array can become full, it will be necessary to resize it as needed, in order not to limit the number of element the tree can contain.

## Possible representations - Linked on array



We can keep a linked list of empty indexes



Has to be created when creating the tree.



Even if the tree is non-linear, we can still use *left* (and/or *right*) to link it



Reallocating  $\Rightarrow$  linking again the empty positions

info							
left	2	3	4	5	6	7	-1
right							

firstEmpty = 1

root = -1

cap = 8

When adding a new node to the tree, a new slot in the arrays have to be allocated, so it would be useful to keep a linked list of empty indexes.

The linked list of empty indexes has to be created when the empty binary tree is created. Please note that even if the 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 the empty positions.

Obviously, when we perform a resize, the newly created empty positions have to be linked again in the list of empty indexes.





### Linked representation with dynamic allocation:

- There is one node for every element of the tree
- The structure representing a node contains:
  - the information
  - a pointer to the left child
  - a pointer to the right child
  - optionally, a pointer to the parent
- NIL denotes the absence of a node
  - $\Rightarrow$  the root of an empty tree is NIL

In the case of the linked representation with dynamic allocation, there is one node for every element of the tree.

The node is represented as a structure that contains: the information, a pointer to the left child, a pointer to the right child and optionally, a pointer to the parent node as well.

The absence of a node is denoted by NIL. Therefore, an empty tree has NIL as its root.

## Binary tree - Dynamically allocated linked representation

- In the following, we are going to use the dynamically allocated linked representation for a binary tree:

### Representation of a node in a Binary Tree:

BTNode:

info: TElem

left: ↑ BTNode

right: ↑ BTNode

### Representation of a Binary Tree:

BinaryTree:

root: ↑ BTNode

So, this is the dynamically allocated linked representation for a binary tree. First, we have a structure for the node, which contains the information in the node and pointers to other two nodes that represent the left and the right children of the node. In the representation of the binary tree itself, we only have the pointer to the root of the tree.



In case of preorder traversal:



Visit the *root* of the tree



Traverse the left subtree - if exists



Traverse the right subtree - if exists



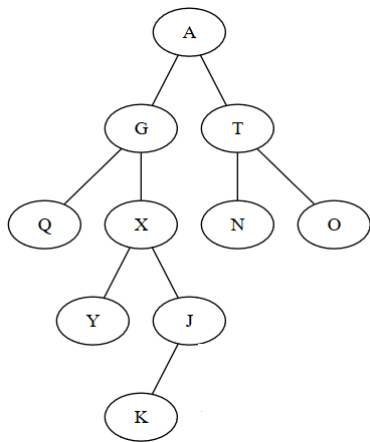
When traversing the subtrees (left or right) the same preorder traversal is applied.

The preorder traversal follows the Root Left Right policy.

The name preorder itself suggests that the root node would be traversed first. So, it involves visiting the root of the tree first. Then, the left subtree is traversed, if it exists and, afterward, the right subtree is traversed, again only if there is a right subtree.

Note that the subtrees are also traversed using the preorder technique. So, from the left subtree, for instance, we visit the root first and then we traverse its left subtree and then its right subtree.

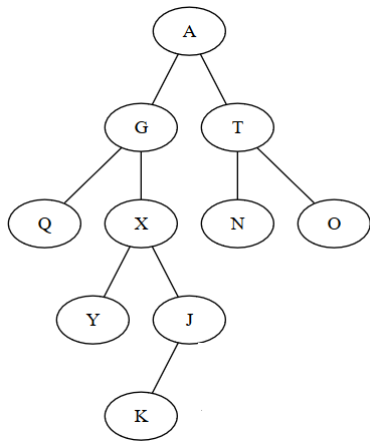
## Preorder traversal example



Here is an example. When traversing this binary tree in preorder, we start by visiting the root node, which contains A. Then, we continue by traversing the left subtree in preorder. So, first we visit the root of the left subtree which contains G and then continue by traversing its left subtree which contains only Q. Then we have X, Y and so on...

- Preorder traversal:

## Preorder traversal example



Here is an example. When traversing this binary tree in preorder, we start by visiting the root node, which contains A. Then, we continue by traversing the left subtree in preorder. So, first we visit the root of the left subtree which contains G and then continue by traversing its left subtree which contains only Q. Then we have X, Y and so on...

- 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.



### Recursive preorder traversal:

**subalgorithm** preorder\_recursive(node) **is:**

*//pre: node is a  $\uparrow$  BTreeNode*

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

  @visit node

  preorder\_recursive([node].left)

  preorder\_recursive([node].right)

**end-if**

**end-subalgorithm**

The simplest way of implementing **implementing** the preorder traversal is using a recursive **recursive** algorithm.

The recursive sub-algorithm starts the traversal from a node, pointed by its unique parameter.

We start by checking the **condition for continuation**, which is that the current sub-tree is not empty. So, if node is not NIL, which means that the subtree to be traversed is non-empty, we visit the node pointed by the parameter *node*, by processing the information there, then we traverse in preorder the left subtree by calling the same subalgorithm for the node the field *left* in the current node points to. Next, the subalgorithm is called for the right subtree.



We need a wrapper subalgorithm that receives a *BinaryTree* and calls the function for the root of the tree.



### Recursive preorder traversal - call

**subalgorithm** preorderRec(tree) **is:**

//pre: tree is a *BinaryTree*

    preorder\_recursive(tree.root)

**end-subalgorithm**

Since the *preorder\_recursive* subalgorithm receives as parameter a pointer to a node, we need a wrapper subalgorithm that receives a *BinaryTree* and calls the function for the root of the tree.

We can see its implementation of the current slide.

When it comes to the times complexity of the preorder traversal, assuming that visiting a node takes constant time (for instance, if we simply print the information from the node), the whole traversal takes  $\Theta(n)$  time for a tree with  $n$  nodes.

## Preorder traversal - Recursive implementation



We need a wrapper subalgorithm that receives a *BinaryTree* and calls the function for the root of the tree.



### Recursive preorder traversal - call

**subalgorithm** preorderRec(tree) **is:**

//pre: tree is a *BinaryTree*

    preorder\_recursive(tree.root)

**end-subalgorithm**



Assuming that visiting a node takes constant time, the traversal takes  $\Theta(n)$  time for a tree with  $n$  nodes.

Since the *preorder\_recursive* subalgorithm receives as parameter a pointer to a node, we need a wrapper subalgorithm that receives a *BinaryTree* and calls the function for the root of the tree.

We can see its implementation of the current slide.

When it comes to the times complexity of the preorder traversal, assuming that visiting a node takes constant time (for instance, if we simply print the information from the node), the whole traversal takes  $\Theta(n)$  time for a tree with  $n$  nodes.



## Preorder traversal - non-recursive implementation

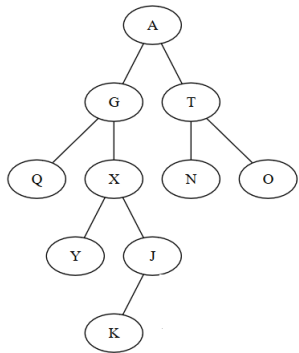


We can implement the preorder traversal algorithm without recursion, using an auxiliary *stack* to store the nodes.

- ▶ 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

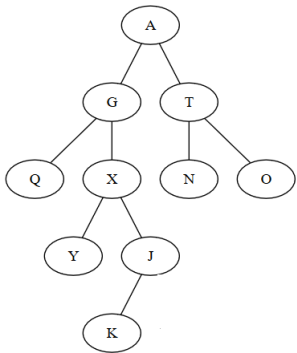
The preorder traversal can also be implemented iteratively, using an auxiliary stack. The idea of the algorithm is the following: We start with an empty stack. Then we push the root of the tree to the empty stack and while the stack is non-empty we repeatedly **uripitidli** perform the following operations: we pop a node from the stack and visit it, then we push its right child on the stack and, afterwards, we also push its left child on the stack.

## Preorder traversal - Non-recursive implementation example



Here we have an example for the non-recursive preorder traversal. So, we start by pushing A on an empty stack. Then we pop from the stack, visit A and push its children: first the right one and then the left one. Consequently, the stack contains T and G. Again, we pop from the stack, we visit G and push its children on the stack, first X then Q. We visit Q next and so on, until the stack becomes empty, which means that the entire tree has been traversed.

## 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

Here we have an example for the non-recursive preorder traversal. So, we start by pushing A on an empty stack. Then we pop from the stack, visit A and push its children: first the right one and then the left one. Consequently, the stack contains T and G. Again, we pop from the stack, we visit G and push its children on the stack, first X then Q. We visit Q next and so on, until the stack becomes empty, which means that the entire tree has been traversed.



### Iterative preorder traversal:

**subalgorithm** preorder(tree) **is:**

*//pre: tree is a binary tree*

The subalgorithm receives the tree as its unique parameter.

It starts by creating an empty stack using the constructor, which is the init operation in the interface of the Stack ADT.

Then, if the tree is non empty, the root of the tree is pushed on the stack.

While the stack is not empty, we repeat the following: we pop from the stack, we visit the node pointed by the popped pointer to node and then we try to push its right and left children, in this order, on the stack. So, if the current node as a right child, the pointer to the right child is pushed on the stack. The same happens for the left child. Please note that the reason why we add the right child first and only then the left one is that they will be popped from the stack in the reverse order of their pushing on the stack.

# Preorder traversal - non-recursive implementation



## Iterative preorder traversal:

**subalgorithm** preorder(tree) **is:**

*//pre: tree is a binary tree*

*init(s) //s:Stack 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**

The subalgorithm receives the tree as its unique parameter.

It starts by creating an empty stack using the constructor, which is the init operation in the interface of the Stack ADT.

Then, if the tree is non empty, the root of the tree is pushed on the stack.

While the stack is not empty, we repeat the following: we pop from the stack, we visit the node pointed by the popped pointer to node and then we try to push its right and left children, in this order, on the stack. So, if the current node as a right child, the pointer to the right child is pushed on the stack. The same happens for the left child. Please note that the reason why we add the right child first and only then the left one is that they will be popped from the stack in the reverse order of their pushing on the stack.



The time complexity of the non-recursive traversal is:

The time complexity of the non-recursive traversal is  $\Theta(n)$ , as well as in the case of the recursive version, but we also need here  $O(n)$  extra space for the auxiliary stack.

Please note that preorder traversal is the same as *depth first traversal*, with the observation that here we need to be careful to first push the right child to the stack and only then the left one, while in case of *depth-first traversal* the order in which we pushed the children was not that important because we discussed the traversal for an unordered tree, for which the order of the children was irrelevant irelăvănt.



The time complexity of the non-recursive traversal is:  $\Theta(n)$ .



Extra-space complexity:

The time complexity of the non-recursive traversal is  $\Theta(n)$ , as well as in the case of the recursive version, but we also need here  $O(n)$  extra space for the auxiliary stack.

Please note that preorder traversal is the same as *depth first traversal*, with the observation that here we need to be careful to first push the right child to the stack and only then the left one, while in case of *depth-first traversal* the order in which we pushed the children was not that important because we discussed the traversal for an unordered tree, for which the order of the children was irrelevant **irelăvănt**.

## Preorder traversal - non-recursive implementation



The time complexity of the non-recursive traversal is:  $\Theta(n)$ .



Extra-space complexity:  $O(n)$  (for the stack).



Obs: Preorder traversal is the same as *depth first traversal*.

The time complexity of the non-recursive traversal is Theta of  $n$ , as well as in the case of the recursive version, but we also need here  $O$  of  $n$  extra space for the auxiliary stack.

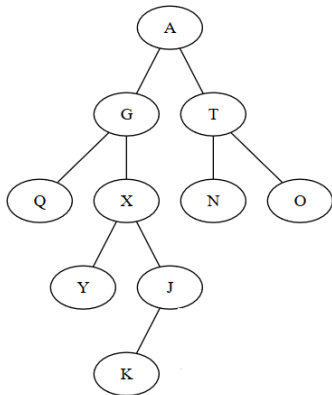
Please note that preorder traversal is the same as *depth first traversal*, with the observation that here we need to be careful to first push the right child to the stack and only then the left one, while in case of *depth-first traversal* the order in which we pushed the children was not that important because we discussed the traversal for an unordered tree, for which the order of the children was irrelevant irelăvănt.



## 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:

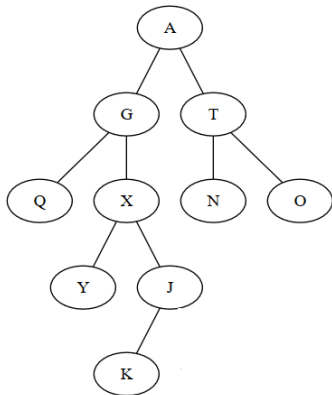
The last traversal, namely the level-order or breath first traversal, is the same as for the non-binary tree and we discussed it in the previous lecture. Please remember it requires to first visit the root, then all the children of the root, then the children of the children and so on. For instance, for this tree, the level order traversal will visit the nodes in the following order: A, g, t, q, x, n, o, y, j and k.

Also remember that for level order traversal, we use an auxiliary queue (instead of a stack).

## 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

The last traversal, namely the level-order or breath first traversal, is the same as for the non-binary tree and we discussed it in the previous lecture. Please remember it requires to first visit the root, then all the children of the root, then the children of the children and so on. For instance, for this tree, the level order traversal will visit the nodes in the following order: A, g, t, q, x, n, o, y, j and k.

Also remember that for level order traversal, we use an auxiliary queue (instead of a stack).



### Iterative level order traversal:

**subalgorithm** levelOrder(tree) **is:**

*//pre: tree is a binary tree*

The subalgorithm receives the tree as its unique parameter.

It starts by creating an empty stack using the constructor, which is the init operation in the interface of the Stack ADT.

Then, if the tree is non empty, the root of the tree is pushed on the stack.

While the stack is not empty, we repeat the following: we pop from the stack, we visit the node pointed by the popped pointer to node and then we try to push its right and left children, in this order, on the stack. So, if the current node as a right child, the pointer to the right child is pushed on the stack. The same happens for the left child. Please note that the reason why we add the right child first and only then the left one is that they will be popped from the stack in the reverse order of their pushing on the stack.

# Level-order traversal - non-recursive implementation



## Iterative level order traversal:

**subalgorithm** levelOrder(tree) **is:**

*//pre: tree is a binary tree*

*init(q) //q:Queue 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**

The subalgorithm receives the tree as its unique parameter.

It starts by creating an empty stack using the constructor, which is the init operation in the interface of the Stack ADT.

Then, if the tree is non empty, the root of the tree is pushed on the stack.

While the stack is not empty, we repeat the following: we pop from the stack, we visit the node pointed by the popped pointer to node and then we try to push its right and left children, in this order, on the stack. So, if the current node as a right child, the pointer to the right child is pushed on the stack. The same happens for the left child. Please note that the reason why we add the right child first and only then the left one is that they will be popped from the stack in the reverse order of their pushing on the stack.

# Trees - Applications



## Real-world applications of tree data structure:



### Hierarchical data storage

- Storing folder structure, XML/HTML data



### Layout of a webpage

- The homepage is the root, the main sections are its children. the subsections are the children's children...



### Machine Learning

- Representing Decision Trees



### Working with Morse Code

- The organization of Morse code is done in the form of a binary tree



### Binary Expression Trees

- For evaluating arithmetic expressions: operators are stored in the internal nodes, while the operands are stored in the leaves

There are many real-world applications of trees. Let's see some of them. Tree data structure are used to naturally store hierarchical **hairarchicăl** data, like folder structure, XML **EX-EM-EL** or HTML **eicî- ti - em - el** data.

Since the layout of a webpage is designed as a tree structure, the homepage or index page being the root node, main sections and site index being their child nodes, which again are parents to multiple other child nodes (subsections), the tree data structure is also used to organize Web pages.

The binary trees are also used in Machine Learning, for representing the Decision Tree automatic classification model.

When working with Morse Code, the organization of the Morse Code is done in the form of a binary tree.

Binary tree are also used to evaluate arithmetic expressions. The operators are stored at the internal nodes and the operands are stored at the leaf node. The arithmetic expression is evaluated bottom up.



## Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

This is the bibliography used for this lecture and recommended for optional extra reading.



We have time left for questions, so if you have any questions, i would be glad to answer them.