DATA STRUCTURES

Trees. Binary trees.

Lect. PhD. Diana-Lucia Miholca

2022 - 2023

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In previous lecture

- Hash tables
  - Coalesced chaining
  - Open addressing

Today

- Trees
  - Terminology
  - Binary Trees

Trees

Trees represent one of the most commonly used data structures.

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

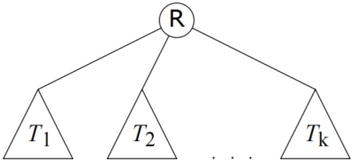
When talking about trees as data structures, we actually mean **rooted trees** in which one node is designated to be the *root* of the tree.

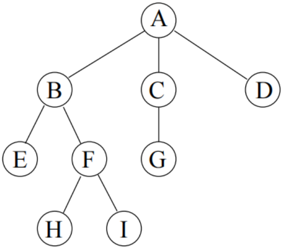
Tree - Definition

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

- If $\mathcal{T}$ is empty, then the tree is empty
- If $\mathcal{T}$ is not empty then:
  - There is a special node, R, called the *root* of the tree
  - The rest of the nodes are divided into k ($k \geq 0$) disjunct *trees*, T_1, T_2, ..., T_k, the root node R being linked by an edge to the root of each of these trees. The trees T_1, T_2, ..., T_k are called the *subtrees* of R.

Tree - visualization





Tree - Terminology

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

Tree - Terminology

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

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

Tree - Terminology

- An **ordered tree** is a tree in which the order of the children is well defined and relevant.
- The **degree** of a node is defined as the number of its children.
- The nodes with the degree 0 are called **leaf nodes**.

Tree - Terminology

- An **ordered tree** is a tree in which the order of the children is well defined and relevant.
- The **degree** of a node is defined as the number of its children.
- The nodes with the degree 0 are called **leaf nodes**.
- The nodes that are not leaf nodes are called **internal nodes**.

Tree - Terminology

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).

Tree - Terminology

- 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 **height** of a node is the length of the longest path from the node to a leaf.
  - All leaves are at height 0.

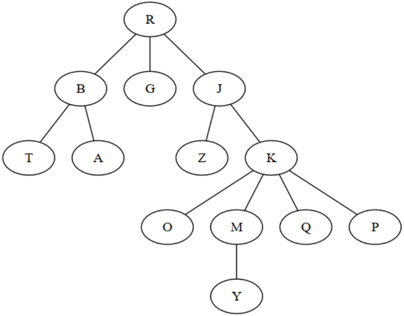
Tree - Terminology

- 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 **height** of a node is the length of the longest path from the node to a leaf.
  - All leaves are at height 0.

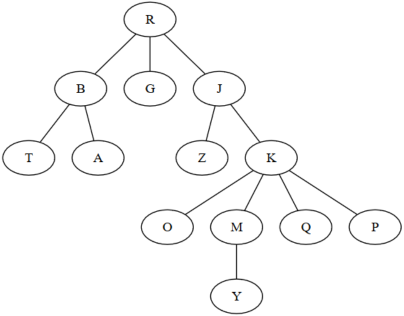
- The **height of the tree** is defined as the height of the root node, i.e. the length of the longest path from the root to a leaf.

Tree - Terminology Example



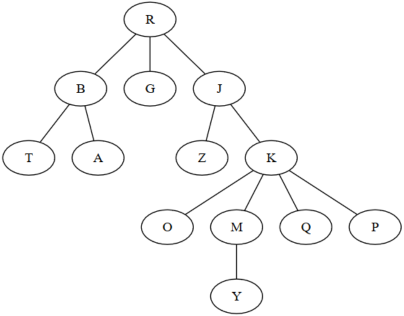
- Root of the tree:

Tree - Terminology Example



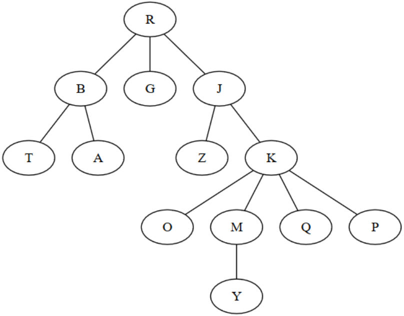
- Root of the tree: R
- Children of R:

Tree - Terminology Example



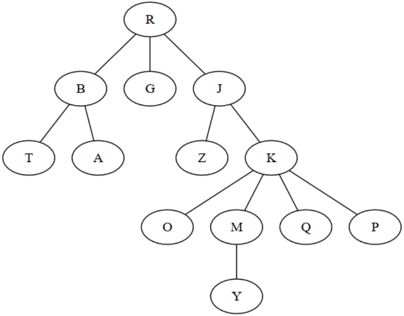
- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*:

Tree - Terminology Example



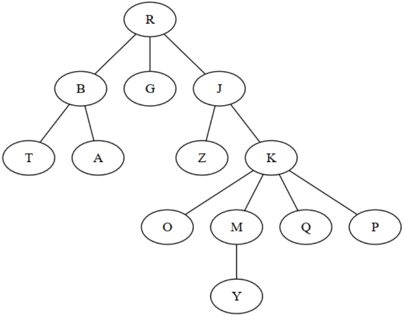
- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes:

Tree - Terminology Example



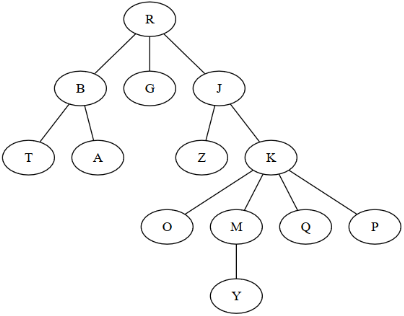
- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes:

Tree - Terminology Example



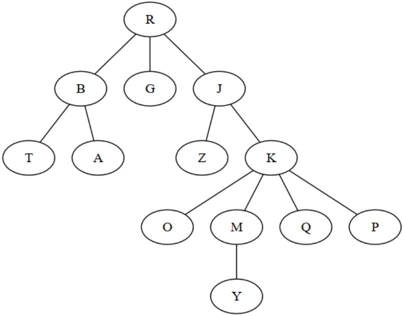
- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node *K*:

Tree - Terminology Example



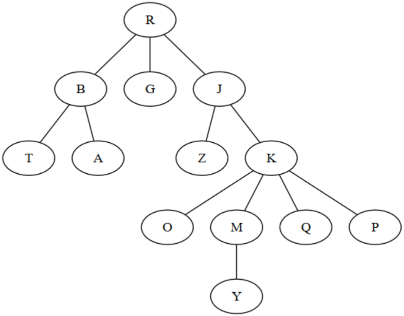
- Root of the tree: *R*
- Children of *R*: B, G, J
- Parent of *M*: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node *K*: 2 (path R-J-K)
- Height of node *K*:

Tree - Terminology Example



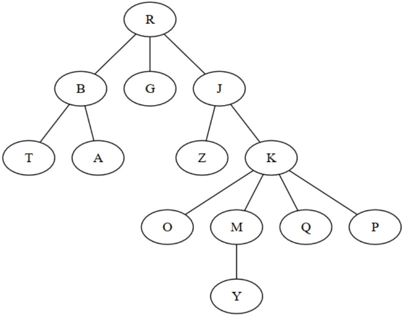
- Root of the tree: R
- Children of R: B, G, J
- Parent of M: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node K: 2 (path R-J-K)
- Height of node K: 2 (path K-M-Y)
- Height of the tree (height of node R):

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:

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

Representing 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 *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

Representing k-ary trees

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)

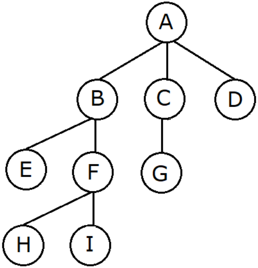
The disadvantage of these approaches is that we occupy space for *k* children even if most nodes have less children.

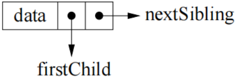
Representing k-ary trees

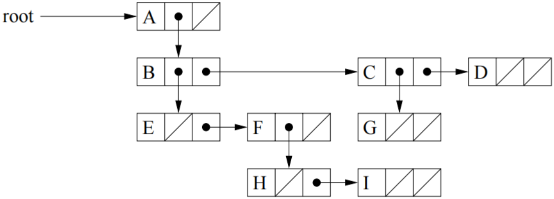
A third option is the so-called *left-child right-sibling* representation in which we have a structure for a node which contains the following:

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

Left-child right sibling representation - Example







Tree traversals

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)

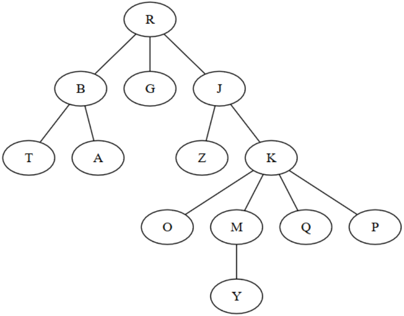
Depth-first traversal

- Traversal starts with visiting the root of the tree

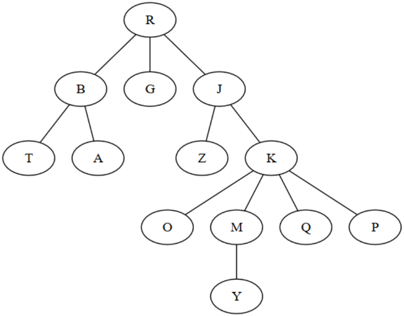
- We visit one of the children, than one child of that child, and so on. We go down (in depth) as much as possible, and continue with other children of a node only after all descendants of the "first" child were visited.

- For depth first traversal we use a stack to remember the nodes in the traversal.

Depth-first traversal - Example



Depth-first traversal - Example

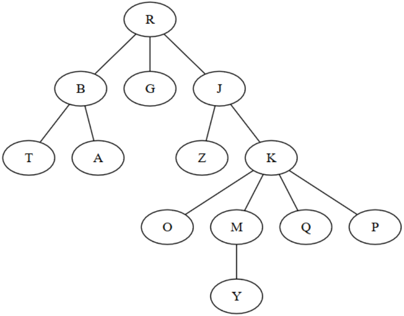


- Stack s with the root: R
- Visit R (pop from stack) and push its children: s = [B G J]
- Visit B and push its children: s = [T A G J]
- Visit T and push nothing: s = [A G J]
- Visit A and push nothing: s = [G J]
- Visit G and push nothing: s = [J]
- Visit J and push its children: s = [Z K]
- etc...

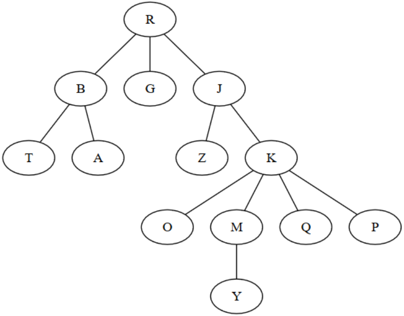
Level-order traversal

- Traversal starts 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.

Level-order traversal - Example



Level-order traversal - Example



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

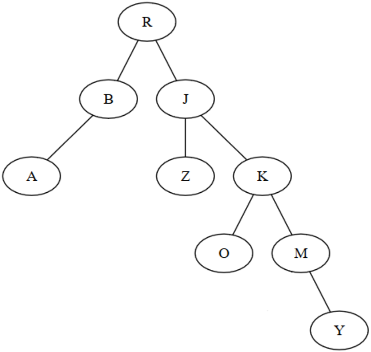
Binary trees

An ordered tree in which each node has at most two children is called **binary tree**.

In a binary tree we call the children of a node the **left child** and **right child**.

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

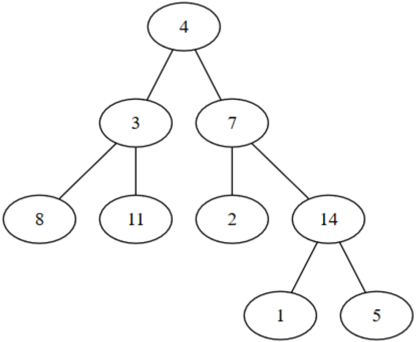
Binary tree - Example



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

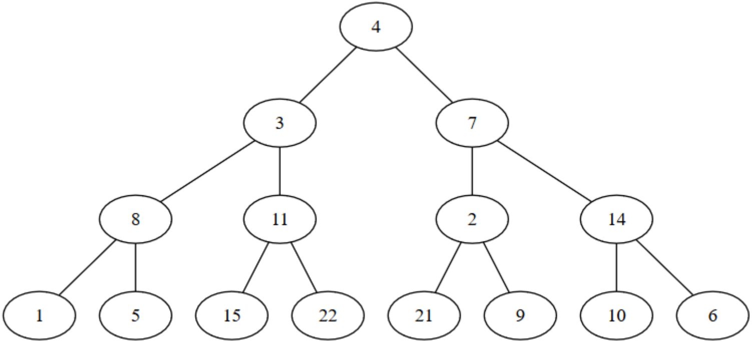
Binary tree - Terminology

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



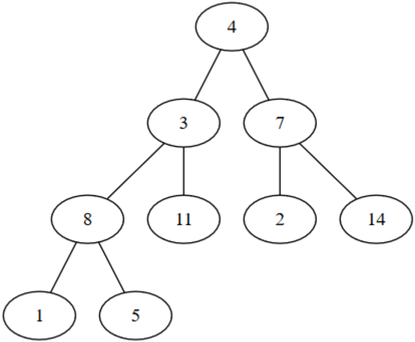
Binary tree - Terminology

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



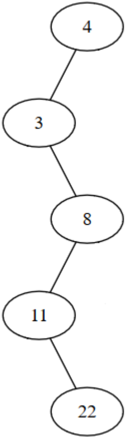
Binary tree - Terminology

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.



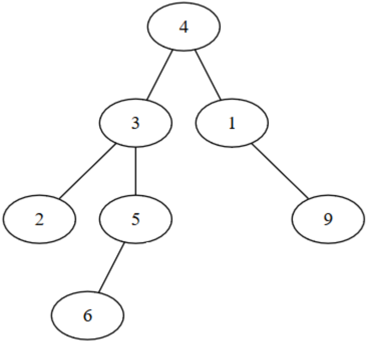
Binary tree - Terminology

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



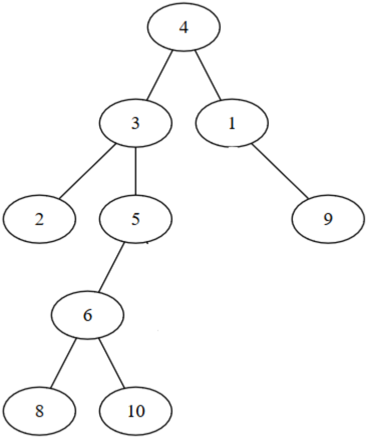
Binary tree - Terminology

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.



Binary tree - Terminology

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



Binary tree - Properties

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

Binary tree - Properties

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

Binary tree - Properties

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?

Binary tree - Properties

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 $2^{N+1} - 1$ $(1 + 2 + 4 + 8 + ... + 2^N)$

Binary tree - Properties

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

Binary tree - Properties

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.

Binary tree - Properties

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?

Binary tree - Properties

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.

Binary tree - Properties

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 $[log_2(N + 1)]$ and $N - 1$.

ADT Binary Tree

- Domain of ADT Binary Tree:

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

ADT Binary Tree

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

ADT Binary Tree

- initLeaf(*bt*, *e*)
  - **descr:** creates a new binary tree, having only the root with a given value
  - **pre:** $e \in TElem$
  - **post:** $bt \in \mathcal{BT}$, *bt* is a binary tree with only one node (its root) which contains the value *e*

ADT Binary Tree

- 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*

ADT Binary Tree

- 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 $left$

ADT Binary Tree

- 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*

ADT Binary 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

ADT Binary Tree

- 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

ADT Binary Tree

- 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

ADT Binary Tree

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

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

ADT Binary Tree

- iterator (bt, traversal, i)
  - **descr:** returns an iterator for a binary tree
  - **pre:** *bt* $\in \mathcal{BT}$, *traversal* represents the order in which the tree has to be traversed
  - **post:** $i \in \mathcal{I}$, *i* is an iterator over *bt* that iterates in the order given by *traversal*

ADT Binary Tree

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

ADT Binary Tree

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

Possible representations

We have several options for representing binary trees:

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

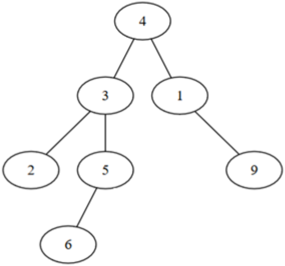
Possible representations

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.

Possible representations



Pos	Elem
1	4
2	3
3	1
4	2
5	5
6	-1
7	9
8	-1
9	-1
10	6
11	-1
12	-1
13	-1
...	...

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

Possible representations

Linked representation with dynamic allocation:

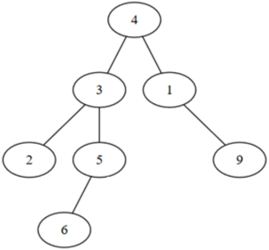
- There is one node for every element of the tree
- The structure representing a node contains:
  - the information
  - a pointer to the left child
  - a pointer to the right child
  - optionally, a pointer to the parent
- NIL denotes the absence of a node
  - $\Rightarrow$ the root of an empty tree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

Possible representations

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

Possible representations



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

Possible representations

- 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	8	-1
right								

firstEmpty = 1

root = -1

cap = 8

Trees- Applications

Real-word 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

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

Thank you