Table of contents:

[Definition of a binary tree 1](#_Toc104733674)

[A few applications on binary trees 1](#_Toc104733675)

[Facts about binary trees 1](#_Toc104733676)

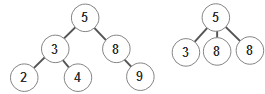
[A real binary tree 2](#_Toc104733677)

[Java implementation of a binary tree 3](#_Toc104733678)

A tree in a field

Description automatically generated with medium confidence

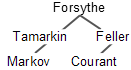
# Definition of a binary tree

****A binary tree is a tree in which each node has at most two children. The first tree on the right is a binary tree. It has nodes with two children, one child, and 0 children. The second tree is not a binary tree because its root has three children.

In a binary tree, the children are called the left child and the right child.

# A few applications on binary trees

Binary trees have lots of applications. Here’s an example of a binary tree that is on the internet: The website [genealogy.math.ndsu.nodak.edu](https://www.genealogy.math.ndsu.nodak.edu/) maintains the PhD genealogy of over 237,500 PhDs in math and CS. On this site, a PhD can have up to two advisors, so the tree of advisors of a PhD is a binary tree.

****To the right, we show the first three levels of the advisor tree for George Forsythe, the first chair of CS at Stanford, beginning in 1965. George worked in the relatively new field of *numerical analysis*. At the time of his move from the Math Dept. to the new CS Dept., he quipped that, “Many numerical analysts have progressed from being queer people in math departments to queer people in CS departments." If you stay in CS or Math, you will quite likely hear of him and his intellectual grandparents, Markov and Courant, again.

# Facts about binary trees

****Here are some facts about binary trees.

1. **Minimum number of nodes in a binary tree of height h:** h+1. A tree with the minimum number of nodes will have one node on each level. Example: the tree to the right has height 2 and 3 nodes.
2. **Maximum number of nodes at depth d**: 2d. Check out the tree to the right. You can see that:
3. The number of nodes at depth 0 is 20 = 1, the root.
4. At each level, the number of nodes is twice that on the previous level because each node on the previous level has two children.
5. **Maximum number of nodes in a binary tree of height h**: 20 + 21 + … + 2h = 2h+1 – 1.

The formula 20 + 21 + … + 2h for the maximum number of nodes is a direct result of the previous point 2. As an example, for the perfect binary tree above, the number of nodes is 20 + 21 + 22 = 7. For a proof of 20 + 21 + … + 2h = 2h+1 – 1, see JavaHyperText entry *binary tree* .

1. **Height of a balanced binary tree:** O(log n). A binary tree is balanced if for each node, the heights of its left and right subtrees differ by at most 1. The height of a balanced binary tree with n nodes is O(log n). (For a proof see JavaHyperText entry *binary tree*.

# Java implementation of a binary tree

To the right is the start of class TreeNode, which implements a node of a binary tree and contains a value of generic type T. It needs only three fields: the field that contains a value, the left subtree, and the right subtree.

**public class** TreeNode<T> {

**private** T data;

**private** TreeNode<T> left; // left subtree (null if empty)

**private** TreeNode<T**>** right; // right subtree (null if empty)

/\*\* Constructor: one-node tree with data d \*/

**public** TreeNode (T d) { data= d; }

/\*\* Constr: Tree with root data d, left tree l, right tree r \*/  
 **public** TreeNode (T d, TreeNode<T> le, TreeNode<T> r) {

data= d; left= le; right= r;

}

}

Two constructors are provided for flexibility. The first creates a one-node binary tree; the second creates a root with two given subtrees.

Naturally, this class has more methods —perhaps observers, a toString method, and so on. There is no need to describe them here.

# An OO implementation of a binary tree

**public interface** BTree<T> {

/\*\* Number of nodes in the tree \*/

**int** size();

/\*\* = "this tree contains v" \*/

**boolean** contains(T v);

}

The implementation above is not OO-oriented. We now give the idea behind an OO implementation of a binary tree, with different objects for an empty tree and a nonempty tree. We start off to the right with an interface that contains two abstract methods, the size of a tree and a method that returns true iff a node of the tree contains the value v.

To the right is class Empty, which implements interface BTree. Since this class is for an empty tree, its size is 0 and it certainly does not contain a value! We don't have to write a constructor; Java will insert the constructor public Empty(){};

**public class** Empty<T>  
 **implements** Btree<T> {

**public** **int** size() { **return** 0; }

**public** **boolean** contains(T v)  
 { **return** **false**; }

}

Below, we give class Node, whose instances represent a node of a non-empty binary tree. The OO approach eliminates the need for if-statements to determine whether a node is empty or not. All methods become simpler.

**public** Tree<T> left() { **return** left; }

....**public** Tree<T> right() { **return** right; }

@Override  
 **public** **int** size() {  
 return 1 + left.size() + right.size();  
 }

@Override  
 **public** **boolean** contains(T v) {  
 **return** value.equals(v) ||   
 left.contains(v) || right.contains(v);  
} }

/\*\* A tree with a root and left and right subtrees \*/

**public class** Node<T> implements Btree<T> {

**private** T value; // not null

**private** Tree<T> left, right; // not null  
 /\*\* Constructor: Tree with root value v, left tree le,  
 and right tree ri.  
 Precondition: le and ri are not null. \*/  
 Node(T v, Tree<T> le, Tree<T> ri)  
 { value= v; left= le; right= ri; }

**public** T value() { **return** value; }