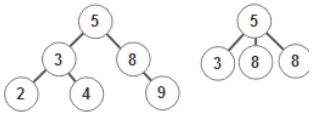
Binary Trees

Table of contents:

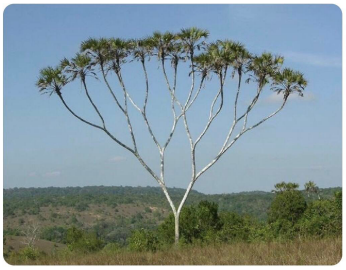
Definition of a binary tree



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.

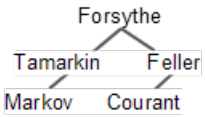
The binary tree actually exists!!



An application of binary trees

Binary trees have lots of applications. Here's an example of a binary tree that is on the internet: The website https://www.mathgenealogy.org 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. Here is David Gries's genealogy tree, with indentation used to show childhood: griesGenealogy.pdf

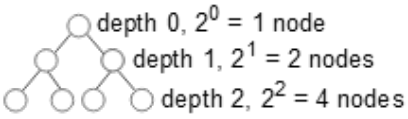


Facts about binary trees

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`:** 2^{d}. Check out the tree to the right. You can see that:



   1. The number of nodes at depth 0 is $2^0 = 1$, the root.
   2. At each level, the number of nodes is twice that on the previous level because each node on the previous level has two children.

3. **Maximum number of nodes in a binary tree of height `h`:** $2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$.

   The formula $2^0 + 2^1 + \ldots + 2^h$ 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 $2^0 + 2^1 + 2^2 = 7$. For a proof of $2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$, see JavaHyperText entry *binary tree* .

4. **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*.

Binary Trees

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.

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.

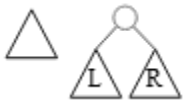
```
public class TreeNode<T> {
   private T data;
   private TreeNode<T> left;   // left subtree (null if empty)
   private TreeNode<T> right; // right subtree (null if empty)

   /** Constructor: one-node tree with data d */
   public TreeNode (T d) { data= d; }

   /** Constr: Tree with root data d, left tree le, right tree ri */
   public TreeNode (T d, TreeNode<T> le, TreeNode<T> re) {
      data= d; left= le; right= r;
   }
}
```

Recursion on binary trees

Binary trees can be defined recursively like this: A binary tree is either

- empty (represented by **null**) or
- an object consisting of a value, a left binary tree, and a right binary tree.

Therefore, binary trees lend themselves naturally to processing using recursion. We will give two examples. First, if we represent a binary tree as a triangle, as to the right, then a binary tree with at least one node can be represented by the second diagram (we won't always write the L and R for left and right subtree). Drawing this diagram helps us focus on how to write a recursive method: deal with the root, the left subtree recursively, and the right subtree recursively, in some order.



Counting nodes satisfying some property

We write a recursive function to count the number of nodes of a tree `t` that contain the value `v`, assuming that the function resides in class `TreeNode`. Its specification and heading are in the box below.

1. What is the base case? Tree `t` may be empty (null), in which case 0 should be returned.

2. What next? Look at the diagram above of a tree with at least one node! We have to deal with the root and two subtrees. We count 1 if the root's value is `v`. And we have to recursively count how many nodes in each subtree contain `v`. This leads to the method that appears to the right.

```
/** = number of nodes of t that contain v.
  * Note: t can be null. */
public int ct(TreeNode t, T v) {
   if (t == null) return 0;
   int c=  v.equals(t.data) ? 1 : 0;
   return c + ct(t.left, v) + ct(t.right, v);
}
```

Another method for the same problem

In method `ct` above, the tree to be processed is given as a parameter. Since the method appears in each object of class `TreeNode`, we write it differently, as shown to the right. Important here is that "this tree" cannot be empty, since it refers to the tree whose root is the object in which the method occurs!

```
/** = number of nodes of this tree that
   *    contain v. */
public int ct(T v) {
   return (v.equals(data) ? 1 : 0) +
          (left == null ? 0 : left.ct(v)) +
          (right == null ? 0 : right.ct(v));
}
```

The method shown to the right doesn't separate the base case (which is when both subtrees are empty) from the recursive case. Instead, looking at the picture of a nonempty tree, it has three conditional expressions, for the root, the left subtree, and the right subtree, each part giving the number of nodes with value `v` in that part. This is the simplest way to write the method.

Searching for a node that satisfies some property

Method `ct` is an example of a recursive method that counts the nodes having a certain property. Another kind of method searches a tree for a node having some property *and terminates as soon as such a node is found*. We illustrate by writing a method that returns a node of tree `t` that contains `v`, but returns **null** if none contain `v`. Its spec and heading are in the box on the next page.

Binary Trees

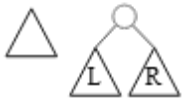
1. What is the base case? Check `t` for **null** first, especially since it is mentioned in the spec! If `t` is null, return **null**.

2. Is there another base case? Look at the tree below and to the right. There's a root! If the root contains `v`, return the root.

3. What is the recursive case? There are two subtrees. Either of them might contain `v`. Call each in turn, and if a call returns a node, return that node.

Here's an important point. In the method body, we wrote the call `getV(t.left, v)` only once and stored its value in a local variable. If we didn't do that, there would be several such duplicate calls, unnecessarily traversing the tree several times. We don't want that.

```
/** = a node of t whose value is v (null if none).
  * Note: t can be null. */
public TreeNode getV(TreeNode t, T v) {
   if (t == null) return null;
   if (v.equals(t.data)) return t;
   TreeNode t1= getV(t.left, v);
   if (t1 != null) return t1;
   return getV(t.right, v);
}
```



Structure of method bodies

Look carefully at the structure of the body of method `getV`. There are no if-else structures. If-else structures, especially when nested, can be hard to understand, and we stay away from them when possible. Instead, in each method body, one case is handled at a time and a value is returned, starting with base cases and moving on to recursive cases. Try to structure your own recursive methods in this fashion.

Keep it simple

At times, there is a tendency to look at the root of a subtree to determine how to process the subtree recursively. Avoid this tendency, for it usually complicates the code unnecessarily. In a way, it is processing the tree according to the diagram that appears to the right, instead of a tree that is viewed as a root and two subtrees. Keep things simple.