

CS231 Fall 2013 Lecture 26

Stephanie R Taylor

November 4, 2013

1 Administrative Topics

- I return the quizzes.

2 Trees

There are lots of weird terms for trees and their parts, depending on your analogy of choice:

- parent, child, sibling, ancestor, descendants
- left, right, up, down
- root, internal node, leaf, branch

Mixed metaphors:

- root is at the top, leaves at the bottom
- a leaf is a node with no children

Levels: each node belongs to one level, or generation, determined by the distance from the root.

There is exactly one path from the root to each node in the tree (if not, then by definition you don't have a tree).

3 Binary trees

Trees that have a maximum of 2 children (but may have 0 or 1) are called binary trees.

Recursive definition: a binary tree is either the null or empty tree or a node that has two children that are binary trees (called subtrees).

If the definition is recursive, guess what most of the code is like? [recursive]

Who cares about binary trees?

- Family ancestry trees
- arithmetic expressions
 - Consider the following tree: $-b + \sqrt{b^2 - 4ac}/2a$. Does it look familiar?
 - Lets look at a simpler example: $2 * 3 + 4$. [do it 2 ways]. Which way correctly represents the expression?

3.1 Properties of binary trees

A fully-complete tree (Dale's term) is a binary tree in which every non-empty level has the maximum possible number of nodes.

How many nodes are there on level i in a fully-complete tree? $[2^i]$ How many nodes are there in a fully-complete tree with levels $0:n$? $[2^{n+1} - 1]$ What is the maximum level in a fully-complete tree with a total of P nodes? $[\log(P + 1) - 1]$ What fraction of the nodes of a fully-complete tree are on the bottom level? [half]

A complete tree is one that is fully-complete except possibly on the bottom level, in which case, the leaves on that level are as far left as possible.

3.2 Implementations of Binary Trees

What comes to mind when you think of implementing binary trees? [have them build the TreeNode class] Consider the following definition of a Tree

that stores an integer at each node:

```
public class TreeNode {  
    public int data;  
    TreeNode left, right;  
  
    public TreeNode(int d, TreeNode l, TreeNode r) {  
        this.data = d;  
        this.left = l;  
        this.right = r;  
    }  
}
```

User:

```
TreeNode tree1 = new TreeNode(5, null, null);  
TreeNode tree2 = new TreeNode(7, null, null);  
TreeNode root = new TreeNode(3, tree1, tree2);
```

This is a linked representation of a binary tree.

Now let's use this implementation to find the total sum of all the values in a tree. That is, let's implement the following method. Each of you try it yourself first:

```
int totalSum(TreeNode root) {  
    if( root == null )  
        return 0;  
    else  
        return root.data + totalSum(root.left) + totalSum(root.right);  
}
```

The totalSum() method also shows the beauty of using recursion when doing some computation involving all of the nodes of the tree.

Problem: User shouldn't directly deal with the TreeNodes, just as the user avoided dealing with nodes in a linked list. Furthermore, we want some way of representing an empty binary tree (with no TreeNodes). We need to create a BinaryTree wrapper class that the user interacts with.

```
public class BinaryTree {  
    private TreeNode root;  
  
    ... its methods ...  
}
```

Clearly, the `totalSum` method belongs in the `BinaryTree` class. Here is what the user should see. User

```
BinaryTree tree = new BinaryTree();
...add nodes & data to the tree...
int result = tree.totalSum();
```

We shouldn't have to compute this sum ourselves. It makes sense to ask a tree to compute its own total sum. Good OOP design guideline: "Ask not what you can do for an object, ask what an object can do for you." Another one is: "Whoever has the data should be responsible for manipulating that data." This is called the "Expert Pattern".

It is easy to fix the code using the existing `totalSum` as an auxiliary recursive method

```
//in BinaryTree class version 1
public int totalSum() {
    return totalSum(root);
}

private int totalSum(TreeNode root) {
    if( root == null )
        return 0;
    else
        return root.data + totalSum(root.left) + totalSum(root.right);
}
```

This is better, but awkward. All the work is being done in the `BinaryTree` class. Instead, each node should be able to compute the sum of its subtree for you! Once again, the OOP Expert Pattern design guideline.

```
//in BinaryTree class version 2
int totalSum() {
    return root.totalSum();
}

//in the TreeNode class
public int totalSum() {
    return data + left.totalSum() + right.totalSum();
}
```

This is so much more elegant, but there is a serious problem with both `totalSum` methods. Can you see it? [no base case so crashes with `NullPointerException`]

erException] We can easily fix this in the BinaryTree's method using a test for root==null. [add it]

But what about the totalSum method in the TreeNode class? Will this solution work?

```
//in the TreeNode class
public int totalSum() {
    if( this == null )
        return 0;
    else
        return value + left.totalSum() + right.totalSum();
}
```

[No. You are checking for null too late. It will crash when the user calls this method on a null object before the body of the method gets executed.]

Let's try another version of totalSum in the TreeNode class.

```
public int totalSum() {
    if(left == null && right == null )
        return data;
    else if( left == null )
        return data + right.totalSum();
    else if( right == null )
        return data + left.totalSum();
    else
        return data + left.totalSum() + right.totalSum();
}
```

This now works, but boy is it ugly! Can we fix it? Yes!

4 Special Null nodes

The code can be made even cleaner than above if you do it even more the OO way! New OOP design guideline: "Replace conditionals with polymorphism." Heres what I mean: (version 3)

```
public class Node {
    int data;
    Node left , right;

    public Node(int d, Node l, Node r) {
```

```

        this.data = d; this.left = l; this.right = r;
    }

    public int totalSum() {
        return value + left.totalSum() + right.totalSum();
    }
}

public class NullNode extends Node {
    public NullNode() { super(0, null, null); }

    public int totalSum() { return 0; }
}

```

The key is never to allow a Node to have null children. That is, make sure that every Node object has exactly two children objects. Then there is always an object to invoke your method on so the totalSum method for Nodes doesn't need to test for null.

So we create a special kind of Tree node called "NullNode" that has its own version of the totalSum method. Note that the constructor for NullNode just calls the super's constructor using d and null and null as arguments, since those values are not important. What's important is that it has its own implementation of totalSum. Note that it has null for left and right pointer values, but it doesn't need to test for null in totalSum since its version of totalSum is not recursive.

[Walk through the execution of totalSum using a simple example (draw the NullNodes as triangles to distinguish them) to see how polymorphism works.] Polymorphism appears in the call "left.totalSum()" in that the version of the method that is executed depends on the actual class of the object referred to by "left". The JRE figures it out while the program is executing.

Here's a summary of where we are:

```

public class BinaryTree {
    private Node root;

    public Tree() {
        root = new NullNode(0);
    }

    public int totalSum() {
        return root.totalSum();
    }
}

```

```

    }
}

public class Node {
    int data;
    Node left , right;

    public Node (int data, Node left , Node right) {
        this.data = data; this.left = left; this.right = right;
    }

    public int totalSum() {
        return data + left.totalSum() + right.totalSum();
    }
}

public class NullNode extends Node {
    public NullNode() { super(0, null, null); }

    public int totalSum() { return 0; }
}

```

Now what do we do to create such a tree? We don't have any methods yet for building a non-empty tree, so we need to add some. For now, assume that the BinaryTree class has some such methods and so the user can create arbitrary binary trees, but we'll add them later.