

CS231 Fall 2013 Lecture 27

Stephanie R Taylor

November 6, 2013

1 Administrative Topics

- Nothing special to note today.
- If I am correct, this is the second time I have had no administrative topics. But that is worth noting.
- So the administrative topic today is that today is special because this is only the second time we have had no administrative topics.

2 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 0 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.

3 Traversals

A traversal of a tree is a process that visits each node in the tree once, in some order. By visit I mean perform some computation. In the example above, the totalSum() method did a traversal of the tree, where visit meant adding its value to the total. Another traversal might just count the number of nodes in the tree. Another traversal might print the value at the node.

As we visit the nodes, what should we print first? The data in the node or the data in the left or right subtree? [can do any]

There are three recursive orderings:

1. pre-order: visit the root first, then left tree, then right
2. in-order: visit the left tree, then root, then right tree
3. post-order: visit the left tree, then right tree, then root

They are called recursive orderings because they can be implemented naturally using recursion.

If the computation at each node is printing, then we have these three methods:

```
public void printInOrder() {
    left.printInOrder();
    print(this.data);
    right.printInOrder();
}
public void printPreOrder() {
    print(this.data);
    left.printPreOrder();
    right.printPreOrder();
}
public void printPostOrder() {
    left.printPostOrder();
    right.printPostOrder();
    print(this.data);
}
```

What does an in-order traversal of our example yield? What does a post-order traversal of our example yield? What does a pre-order traversal of our example yield?

So $3+4*5$, $+ 3 * 4 5$, and $3 4 5 * +$ are all different ways of denoting the same expression. These versions are called infix, prefix, and postfix.

If we also apply these print methods to the trees corresponding to $(3+4)*5$, what do we get? They have different prefix and postfix forms, but both trees have the same inorder traversal printout! Therefore from just the inorder printout, you can't reconstruct the tree. That's why parentheses are needed in infix expressions. However, you can from the preorder and postorder printouts (assuming you know that the tree corresponds to a binary expression tree with operators for nodes and values for leaves). Parentheses are never needed for prefix and postfix expressions.

Practice converting. For each expression, convert it into the other two forms:

- infix: $2-(3+4)*5$
- prefix: $+ - 3 4 5$

- postfix 8 4 3 - *