

Binary Trees

You have covered these in CS1, but we'll look at them in greater detail in this class, also noting implementation differences between C and Java. We will declare a Binary Tree Node class, `BinTreeNode` to store a single node. Here are the instance variables of the class:

```
public int data;  
public BinTreeNode left;  
public BinTreeNode right;
```

Also, just as we did with linked lists, we will have a second class that maintains a Binary Tree, which will be called `BinTree`. It will only have a single `BinTreeNode` instance variable:

```
private BinTreeNode root;
```

We will actually do most of our work through the `BinTreeNode` class. The only cases that we will pay attention to in the `BinTree` class are the ones that deal with the empty binary tree. An empty binary tree will be represented by the instance variable `root` being null. But, for the most part, all the methods in the `BinTree` class will call on their counterparts in the `BinTreeNode` class.

Review: Tree Traversals

Visiting each node in a binary tree is not as straight forward as in a linked list. In particular, you can't necessarily go in a "straight line." But, recursion helps solve the problem. Notice that each BinTreeNode is made of three components:

- 1) A value
- 2) A left subtree
- 3) A right subtree

One can visit each node in the structure by simply visiting these three components in any order. Notice that components 2 and 3 are trees as well, thus necessitating recursion. Our convention is that we will visit the left subtree before the right. This leaves us three different orders to visit the three components above. Each of these orders corresponds to a tree traversal as follows:

Preorder Tree Traversal

1. Visit the root
2. Visit the left subtree
3. Visit the right subtree

Inorder Tree Traversal

1. Visit the root
2. Visit the left subtree
3. Visit the right subtree

Postorder Tree Traversal

1. Visit the root
2. Visit the left subtree
3. Visit the right subtree

Code for traversals

The code for each of the three tree traversals is quite similar. Here is the code for the Inorder traversal:

In the BinTree class:

```
public void Inorder() {  
    if (root != null)  
        root.Inorder();  
}
```

In the BinTreeNode class:

```
public void Inorder() {  
    if (left != null)  
        left.Inorder();  
    System.out.print(data+" ");  
  
    if (right != null)  
        right.Inorder();  
}
```

Before, we move on, let's trace through an example of each of the traversals.

Inserting a Node Into a Binary Tree

If we are doing an insert into a BinTreeNode object (which must have at least one node), here's a basic outline of the steps necessary:

- 1) Compare the value to insert to the root value.**
- 2) If the value to insert is less, insert the node into the left subtree. This can be done using a temporary pointer. Also, if this subtree is null, make room for the node and insert it.**
- 3) If the value is greater than or equal to the root value, do exactly as listed above in step 2, except to the right side.**

Now, the only case not considered is inserting into an empty Binary Tree object. Here, we can just create a single BinTreeNode storing the value to insert. The code corresponding to this is in the BinTree class.

Consider the following illustrations:

Code to do an Insert BinTreeNode class

```
// Inserts a new node storing the value x into the binary search  
// tree with the root as the current object.
```

```
public void insert(int x) {  
  
    // Create new node to insert  
    BinTreeNode temp = new BinTreeNode(x);  
    BinTreeNode cur = this; // temp node to traverse tree.  
    boolean done = false;  
  
    while (!done) {  
  
        // Decide if node should be inserted in R or L subtree.  
        if (cur.data >= x) {  
  
            // Only traverse down tree if left pointer isn't null.  
            if (cur.left != null)  
                cur = cur.left;  
            // Or, insert temp node in the correct location.  
            else {  
                cur.left = temp;  
                done = true;  
            }  
        }  
  
        else {  
            // Only traverse down tree if right pointer isn't null.  
            if (cur.right != null)  
                cur = cur.right;  
            // Or, insert temp node in the correct location.  
            else {  
                cur.right = temp;  
            }  
        }  
    }  
}
```

```
        done = true;
    }
}
}
```

The necessary changes to the BinTree class are minimal:

```
// Inserts a node containing x into the binary tree.
public void insert(int x) {
    if (root == null)
        root = new BinTreeNode(x);
    else
        root.insert(x);
}
```

Deleting from a Binary Tree

When deleting a node from a binary tree, here are the three cases to consider:

- 1) Node to delete is a leaf node**
- 2) Node to delete has one child**
- 3) Node to delete has two children.**

The first case is the easiest: Just find the parent and set either it's left or right (which ever is appropriate) to null.

For the second case, we see that we can just "patch" the appropriate parent link to the child of the node to be deleted.

Finally, for the third case, we can find a value in the tree to replace the value at the node to be deleted. In particular, the minimum value in the right subtree of the node can be stored in the node to be deleted. Then, we can delete the old node that value was stored in because it had at most 1 child.

Consider each of these examples: