1 Trees

The next major set of data structures belongs to what's called Trees. They are called that, because if you try to visualize the structure, it kind of looks like a tree (root, branches, and leafs). Trees are node based data structures, meaning that they're made out of small parts called nodes. You already know what a node is, and used one to build a linked list. Tree Nodes have two or more child nodes; unlike our list node, which only had one child.

Trees are named by the number of children their nodes have. For example, if a tree node has two children, it is called a binary tree. If it has three children, it is called tertiary tree. If it has four children, it is called a quad tree, and so on. Fortunately, to simplify things, we only need binary trees. With binary trees, we can simulate any tree; so the need for other types of trees only becomes a matter of simplicity for visualization.

Since we'll be working with binary trees, lets write a binary tree node. It's not going to be much different from our pOneChildNode class; actually, it's going to be quite the same, only added support for one more pointer. The source for the follows:

Listing 1: Node with two child reference.

```
public class pTwoChildNode {

    protected Object data;
    protected pTwoChildNode left,right;

    public pTwoChildNode(){
        data = null;
        left = right = null;
    }
    public pTwoChildNode(Object d){
        data = d;
        left = right = null;
    }
    public void setLeft(pTwoChildNode l){
        left = l;
    }
    public void setRight(pTwoChildNode r){
        right = r;
    }
    public void setData(Object d){
        data = d;
    }
    public pTwoChildNode getLeft(){
        return left;
    }
    public pTwoChildNode getRight(){
        return right;
    }
    public Object getData(){
        return data;
```

```
    }
    public String toString(){
        return ""+data;
    }
}
```

This node is so much similar to the previous node we did, that I'm not even going to cover this one. (I assume you'll figure it out by simply looking at the source) The children of the node are named left and right; these will be the left branch of the node and a right branch. If a node has no children, it is called a leaf node. If a node has no parent (it's the father of every node), it is called the root of the tree. This weird terminology comes from the tree analogy, and from the family tree analogy.

Some implementations of the tree node, might also have a back pointer to the parent node, but for what we'll be doing with the nodes, it's not necessary. The next section will talk about a generic binary tree which will be later used to create something cool.

2 Generic Tree

Binary Trees are quite complex, and most of the time, we'd be writing a unique implementation for every specific program. One thing that almost never changes though is the general layout of a binary tree. We will first implement that general layout as an abstract class (a class that can't be used directly), and later write another class which extends our layout class.

Trees have many different algorithms associated with them. The most basic ones are the traversal algorithms. Traversals algorithms are different ways of going through the tree (the order in which you look at it's values). For example, an in-order traversal first looks at the left child, then the data, and then the right child. A pre-order traversal, first looks at the data, then the left child, and then the right; and lastly, the post-order traversal looks at the left child, then right child, and only then data. Different traversal types mean different things for different algorithms and different trees. For example, if it's binary search tree (I'll show how to do one later), then the in-order traversal will print elements in a sorted order.

Well, lets not just talk about the beauties of trees, and start writing one! The code that follows creates an abstract Generic Binary Tree.

Listing 2: An abstract binary tree.

```
public abstract class pGenericBinaryTree {
    private pTwoChildNode root;

    protected pTwoChildNode getRoot(){
        return root;
    }
    protected void setRoot(pTwoChildNode r){
        root = r;
    }
    public pGenericBinaryTree(){
        setRoot(null);
```

```
}
public pGenericBinaryTree(Object o){
    setRoot(new pTwoChildNode(o));
}
public boolean isEmpty(){
    return getRoot() == null;
}
public Object getData(){
    if(!isEmpty())
        return getRoot().getData();
    return null;
}
public pTwoChildNode getLeft(){
    if(!isEmpty())
        return getRoot().getLeft();
    return null;
}
public pTwoChildNode getRight(){
    if(!isEmpty())
        return getRoot().getRight();
    return null;
}
public void setData(Object o){
    if(!isEmpty())
        getRoot().setData(o);
}
public void insertLeft(pTwoChildNode p,Object o){
    if((p != null) && (p.getLeft() == null))
        p.setLeft(new pTwoChildNode(o));
}
public void insertRight(pTwoChildNode p,Object o){
    if((p != null) && (p.getRight() == null))
        p.setRight(new pTwoChildNode(o));
}
public void print(int mode){
    if(mode == 1) pretrav();
    else if(mode == 2) intrav();
    else if(mode == 3) postrav();
}
public void pretrav(){
    pretrav(getRoot());
}
protected void pretrav(pTwoChildNode p){
    if(p == null)
        return;
    System.out.print(p.getData()+"␣");
    pretrav(p.getLeft());
```

```
        pretrav(p.getRight());
    }
    public void intrav(){
        intrav(getRoot());
    }
    protected void intrav(pTwoChildNode p){
        if(p == null)
            return;
        intrav(p.getLeft());
        System.out.print(p.getData()+" ");
        intrav(p.getRight());
    }
    public void postrav(){
        postrav(getRoot());
    }
    protected void postrav(pTwoChildNode p){
        if(p == null)
            return;
        postrav(p.getLeft());
        postrav(p.getRight());
        System.out.print(p.getData()+" ");
    }
}
```

Now, lets go over it. The pGenericBinaryTree is a fairly large class, with a fair amount of methods. Lets start with the one and only data member, the root! In this abstract class, root is a private head of the entire tree. Actually, all we need is root to access anything (and that's how you'd implement it in other languages). Since we'd like to have access to root from other places though (from derived classes, but not from the "outside," we've also added two methods, named getRoot(), and setRoot() which get and set the value of root respectively.

We have two constructors, one with no arguments (which only sets root to null), and another with one argument (the first element to be inserted on to the tree). Then we have a standard isEmpty() method to find out if the tree is empty. You'll also notice that implementing a counter for the number of elements inside the tree is not a hard thing to do (very similar to the way we did it in a linked list).

The getData() method returns the data of the root node. This may not be particularly useful to us right now, but may be needed in some derived class (so, we stick it in there just for convenience). Throughout data structures, and mostly entire programming world, you'll find that certain things are done solely for convenience. Other "convenient" methods are getLeft(), getRight() and setData().

The two methods we'll be using later (for something useful), are: insertLeft(pTwoChildNode,Object), and insertRight(pTwoChildNode,Object). These provide a nice way to quickly insert something into the left child (sub-tree) of the given node.

The rest are just print methods. The trick about trees are that they can be traversed in many different ways, and these print methods print out the whole tree, in different traversals. All of these are useful, depending on what you're doing, and what type of tree you have.

Sometimes, some of them make absolutely no sense, depending on what you're doing.

Printing methods are recursive; a lot of the tree manipulation functions are recursive, since they're described so naturally in recursive structures. A recursive function is a function that calls itself (kind of like pretrav(), intrav(), and postrav() does).

Go over the source, make sure you understand what each function is doing (not a hard task). It's not important for you to understand why we need all these functions at this point (for now, we "just" need them); you'll understand why we need some of them in the next few sections, when we extend this class to create a really cool sorting engine.

3 Comparing Objects

Comparing Objects in Java can be a daunting task, especially if you have no idea how it's done. In Java, we can only compare variables of native type. These include all but the objects (ex: int, float, double, etc.). To compare Objects, we have to make objects with certain properties; properties that will allow us to compare.

We usually create an interface, and implement it inside the objects we'd like to compare. In our case, we'll call the interface pComparable. Interfaces are easy to write, since they're kind of like abstract classes.

Listing 3: A custom comparable interface.

```
public interface pComparable {
    public int compareTo(Object o);
}
```

As you can see, there is nothing special to simple interfaces. Now, the trick is to implement it. You might be saying, why am I covering comparing of objects right in the middle of a binary tree discussion... well, we can't have a binary search tree without being able to compare objects. For example, if we'd like to use integers in our binary search tree, we'll have to design our own integer, and let it have a pComparable interface.

Next follows our implementation of pInteger, a number with a pComparable interface. I couldn't just extend the java.lang.Integer, since it's final (cannot be extended) (those geniuses!).

Listing 4: A custom integer class.

```
public class pInteger implements pComparable{
    int i;

    public pInteger(){
    }
    public pInteger(int j){
        set(j);
    }
    public int get(){
        return i;
    }
    public void set(int j){
```

```
        i = j;
    }
    public String toString(){
        return ""+get();
    }
    public int compareTo(Object o){
        if(o instanceof pInteger)
            if(get() > ((pInteger)o).get())
                return 1;
            else if(get() < ((pInteger)o).get())
                return -1;
        return 0;
    }
}
```

I believe most of the interface is self explanatory, except maybe for the compareTo(Object) method. In the method, we first make sure that the parameter is of type pInteger, and later using casting, and calling methods, we compare the underlying native members of pInteger and return an appropriate result.

A note on JDK 1.2: In the new versions of the JDK, you won't need to implement your own pComparable, or your own pInteger; since it's built in! There is a Comparable interface, and it's already implemented by the built in java.lang.Integer, java.lang.String, and other classes where you might need comparisons. I'm doing it this way only for compatibility with the older versions of JDK. I'll talk about JDK 1.2 features later in this document (hopefully).

4 Binary Search Trees

And now, back to the show, or shall I say Binary Trees! A binary tree we'll be designing in this section will be what's known as binary search tree. The reason it's called this is that it can be used to sort numbers (or objects) in a way, that makes it very easy to search them; traverse them. Remember how I've said that traversals only make sense in some specific context, well, in binary search tree, only the in-traversal makes sense; in which numbers (objects) are printed in a sorted fashion. Although I'll show all traversals just for the fun of it.

A binary search tree will extend our pGenericBinaryTree, and will add on a few methods. One that we definitely need is the insert() method; to insert objects into a tree with binary search in mind. Well, instead of just talking about it, lets write the source!

Listing 5: Binary Search Tree

```
public class pBinarySearchTree extends pGenericBinaryTree{

    public pBinarySearchTree(){
        super();
    }

    public pBinarySearchTree(Object o){
```

```
        super(o);
    }

    public void print(){
        print(2);
    }

    public void insert(pComparable o){
        pTwoChildNode t,q;
        for(q = null, t = getRoot();
            t != null && o.compareTo(t.getData()) != 0;
            q = t,t = o.compareTo(t.getData()) < 0
                ? t.getLeft():t.getRight());
        if(t != null)
            return;
        else if(q == null)
            setRoot(new pTwoChildNode(o));
        else if(o.compareTo(q.getData()) < 0)
            insertLeft(q,o);
        else
            insertRight(q,o);
    }
}
```

As you can obviously see, the insert(pComparable) method is definitely the key to the whole thing. The method starts out by declaring two variables, 't', and 'q'. It then falls into a for loop. The condition inside the for loop is that 't' does not equal to null (since it was initially set to getRoot(), which effectively returns the value of root), and while the object we're trying to insert does not equal to the object already inside the tree.

Usually, a binary search tree does not allow duplicate insertions, since they're kind of useless; that's why we're attempting to catch the case where we're trying to insert a duplicate. Inside the for loop, we set 'q' to the value of the next node to be examined. We do this by first comparing the data we're inserting with the data in the current node, if it's greater, we set 't' to the right node, if less, we set it to the left node (all this is cleverly disguised inside that for statement).

We later check the value of 't' to make sure we've gotten to the end (or leaf) of the tree. If 't' is not null, that means we've encountered a duplicate, and we simply return. We then check to see if the tree is empty (didn't have a root), if it didn't, we create a new root by calling setRoot() with a newly created node holding the inserted data.

If all else fails, simply insert the object into the left or the right child of the leaf node depending on the value of the data. And that's that!

Understanding binary search trees is not easy, but it is the key to some very interesting algorithms. So, if you miss out on the main point here, I suggest you read it again, or get a more formal reference (where I doubt you'll learn more).

Anyway, as it was with our stacks and queues, we always had to test everything, so, lets test it! Below, I give you the test module for the tree.

Listing 6: Binary Search Tree Test

```
class pBinarySearchTreeTest{
    public static void main(String[] args){
        pBinarySearchTree tree = new pBinarySearchTree();
        pInteger n;
        int i;
        System.out.println("Numbers inserted:");
        for(i=0;i<10;i++){
            tree.insert(n=new pInteger((int)(Math.random()*1000)));
            System.out.print(n+" ");
        }
        System.out.println("\nPre-order:");
        tree.print(1);
        System.out.println("\nIn-order:");
        tree.print();
        System.out.println("\nPost-order:");
        tree.print(3);
    }
}
```

As you can see, it's pretty simple (and similar to our previous tests). It first inserts ten pInteger (pComparable) objects in to the tree, and then traverses the tree in different orders. These different orders print out the whole tree. Since we know it's a binary search tree, the in-order traversal should produce an ordered output. So, lets take a look at the output!

```
Numbers inserted:
500 315 219 359 259 816 304 902 681 334
Pre-order:
500 315 219 259 304 359 334 816 681 902
In-order:
219 259 304 315 334 359 500 681 816 902
Post-order:
304 259 219 334 359 315 681 902 816 500
```

Well, our prediction is confirmed! The in-order traversal did produce sorted results. There is really nothing more I can say about this particular binary search tree, except that it's worth knowing. This is definitely not the fastest (nor was speed an issue), and not necessarily the most useful class, but it sure may proof useful in teaching you how to use trees.

And now, onto something completely different! NOT! We're going to be doing trees for a while... I want to make sure you really understand what they are, and how to use them.

5 Tree Traversals

I've talked about tree traversals before in this document, but lets review what I've said. Tree's are created for the sole purpose of traversing them. There are two major traversal algorithms, the depth-first, and breadth-first.

So far, we've only looked at depth-first. Pre-traversal, in-traversal, and post-traversal are subsets of depth-first traversals. The reason it's named depth-first, is because we eventually end up going to the deepest node inside the tree, while still having unseen nodes closer to the root (it's hard to explain, and even harder to understand). Tracing a traversal surely helps; and you can trace that traversal from the previous section (it's only ten numbers!).

The other type of traversal is more intuitive; more "human like." Breadth-first traversal goes through the tree top to bottom, left to right. Lets say you were given a tree to read (sorry, don't have a non-copyrighted picture I can include), you'd surely read it top to bottom, left to right (just like a page of text, or something).

Think of a way you visualize a tree... With the root node on top, and all the rest extending downward. What Breadth-First allows us to do is to trace the tree from top to bottom as you see it. It will visit each node at a given tree depth, before moving onto the the next depth.

A lot of the algorithms are centered around Breadth-First method. Like the search tree for a Chess game. In chess, the tree can be very deep, so, doing a Depth-First traversal (search) would be costly, if not impossible. With Breadth-First as applied in Chess, the program only looks at several moves ahead, without looking too many moves ahead.

The Breadth-First traversal is usually from left to right, but that's usually personal preference. Because the standard consul does not allow graphics, the output may be hard to correlate to the actual tree, but I will show how it's done.

As with previous examples, I will provide some modified source that will show you how it's done. An extended pBinarySearchTree is shown below:

Listing 7: Breadth first traversal

```
public class pBreadthFirstTraversal extends pBinarySearchTree{

    public void breadth_first(){
        pEasyQueue q = new pEasyQueue();
        pTwoChildNode tmp;
        q.insert(getRoot());
        while(!q.isEmpty()){
            tmp = (pTwoChildNode)q.remove();
            if(tmp.getLeft() != null)
                q.insert(tmp.getLeft());
            if(tmp.getRight() != null)
                q.insert(tmp.getRight());
            System.out.print(tmp.getData()+" ");
        }
    }
}
```

As you can see, the class is pretty simple (only one function). In this demo, we're also using pEasyQueue, developed earlier in this document. Since breadth first traversal is not like depth first, we can't use recursion, or stack based methods, we need a queue. Any recursive method can be easily simulated using a stack, not so with breadth first, here, we definitely need a queue.

As you can see, we start by first inserting the root node on to the queue, and loop while the queue is not isEmpty(). If we have a left node in the node being examined, we insert it in to the queue, etc. (same goes for the right node). Eventually, the nodes inserted in to the queue, get removed, and subsequently, have their left children examined. The process continues until we've traversed the entire tree, from top to bottom, left to right order.

Now, lets test it. The code below is pretty much the same code used to test the tree, with one minor addition; the one to test the breadth-first traversal!

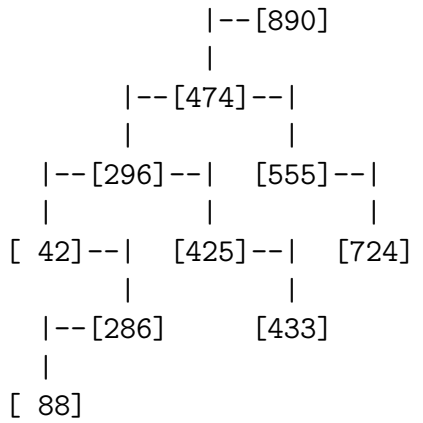
Listing 8: Breadth first traversal test

```
class pBreadthFirstTraversalTest {
    public static void main(String[] args){
        pBreadthFirstTraversal tree = new pBreadthFirstTraversal();
        pInteger n;
        int i;
        System.out.println("Numbers inserted:");
        for(i=0;i<10;i++){
            tree.insert(n=new pInteger((int)(Math.random()*1000)));
            System.out.print(n+" ");
        }
        System.out.println("\nPre-order:");
        tree.print(1);
        System.out.println("\nIn-order:");
        tree.print();
        System.out.println("\nPost-order:");
        tree.print(3);
        System.out.println("\nBreadth-First:");
        tree.breadth_first();
    }
}
```

As you can see, nothing too hard. Next, goes the output of the above program, and you and I will have to spend some time on the output.

```
Numbers inserted:
890 474 296 425 433 555 42 286 724 88
Pre-order:
890 474 296 42 286 88 425 433 555 724
In-order:
42 88 286 296 425 433 474 555 724 890
Post-order:
88 286 42 433 425 296 724 555 474 890
Breadth-First:
890 474 296 555 42 425 724 286 433 88
```

Looking at the output in this format is very abstract and is not very intuitive. Lets just say we have some sort of a tree, containing these numbers above. We were looking at the root node. Now, looking at the output of this program, can you guess what the root node is? Well, it's the first number in breadth-first: 890. The left child of the root is: 474. Basically, the tree looks like:



As is evident from this picture, I'm a terrible ASCII artist! (If it's screwed up, sorry).

What you can also see is that if you read the tree form left to right, top to bottom, you end up with the breadth first traversal. Actually, this is a good opportunity for you to go over all traversals, and see how they do it, and if they make sense.

There are many variations to these traversals (and I'll show a few in subsequent sections), for now, however, try to understand these four basic ones.

Well, see you again in the next section, where we examine some ways you can speed up your code and take a look at JDK 1.2 features, which will simplify our life a LOT!