Binary Search Trees

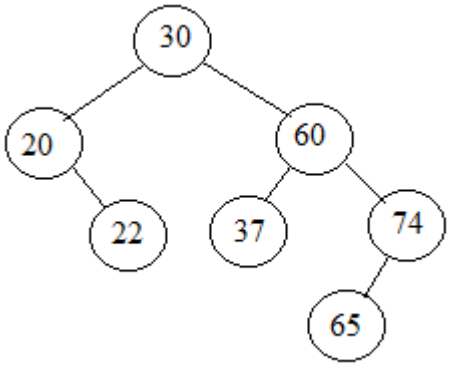
A binary tree is a data structure that allows us to store values in a structured order so that we can insert, delete and search for data in an efficient manner. If we wanted to maintain data that got frequently modified by inserts and deletes, an array poses a number of problems, in terms of efficiency:

1. If we store the array in arbitrary order, a search requires us to look all the way through the whole array for a value, which is undesireable.

2. If we store the array in a sorted order, each insert could require us to shift a majority of the array values, which for a large amount of data is also undesirable.

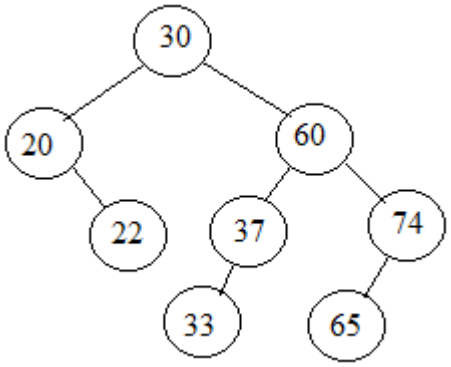
Our basic idea for storing data is as follows:

In a single object store a value (your data), but also store a reference to another object of the same class to the left, and to the right. To determine which values go to the left and right of any object, use the simple following rule: if an object stores the value x, then all values less than x must appear to the left of that object, and the rest of the values appear to the right of the object. Here is an example several objects strategically linked in this manner:



Given this structure, searching for a value does NOT require us to look at all values in the structure. For example, if we are searching for 12, we compare it to the value stored at the top node (called the root node), and see that 12 is less than 30. This means that if 12 is in the tree, it's to the left of 30. Then we compare 12 to 20 and see that if 12 is in the three, it has to be to the left of 20. Since nothing is to the left of 20, 12 isn't in the tree.

Similarly, when inserting a value into the tree, we don't have to look at each value in the tree. Consider inserting 33. It must go to the right of 30, the left of 60 and the left of 37, where we find its final spot:



In both algorithms, we must simply walk down one path of the tree, from top to one of the bottom nodes (called leaf nodes). For large trees, it's typically the case that they store many more nodes than can be found down a single path. In this manner, the efficiency of both inserting and searching for a value is much better than just using a regular array.

Since this is meant to be a very topical lecture on binary search trees, these notes won't cover some of the typical vocabulary or the mechanics of the delete function, though it is possible to write a delete function that is equally efficient to the insert and search functions shown in this lecture.

Class to Store a Binary Tree Node

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;
```

In an actual binary tree, it's possible to store no values, whereas if we declare and create a bintreenode object, we already have one node. So what we will do is have to write a bit of extra code when we use a bintreenode to store a binary tree. Another method of implementation would be to write a second wrapper class which contains a single instance variable, one BinTreeNode. While this is probably better stylistically, it seems to add unnecessary code and make it a bit more complicated to understand, so due to the topical nature of this lecture, I'll skip writing a wrapper class. (A wrapper class, used in this context is simply a new class that has only one instance variable that belongs to a different class, and no other instance variables.)

To store an empty binary tree, we will simply set our bintreenode reference where we will store the root of the tree to null. Then, when inserting an item into this empty tree, we'll have to write some extra code.

Recursive Search Method in bintreenode

If a tree is empty, then it can't contain anything we are searching for. Thus, for an initial call to a search in a binary search tree, we'll assume it already has at least one node and write our search method inside the bintreenode class. Our value must lie in one of three places:

1. The root (top node)
2. Somewhere in the left sub tree
3. Somewhere in the right sub tree.

Note, in our example tree on the second page, the nodes 20 and 22 are in the left subtree of the root, 30 and 33, 37, 60, 65 and 74 are all in the right subtree of the root, 30.

Due to the structure of a binary search tree, we never have to look in both #2 and #3. At most, we will look in one of those two places.

Since we can think about #2 and #3 as trees themselves, these are easily modeled as recursive searches. Thus, our base case is when the value we are searching for is in the root. If it's not, we'll either recursively search in the left or right subtree. Note: if the subtree we want to search in is null, we can automatically return false, since our value isn't in the tree.

Here is what we have in code:

```
public boolean search(int val) {

    if (val == data) return true;

    if (left != null && val < data) return left.search(val);

    if (right != null && val > data) return right.search(val);

    return false;
}
```

The first line is the base case. In the second line, we only want to recurse to the left if two things are true - a left subtree exists and the value we are searching for is less than the data stored at our root. Notice our recursive call in this object oriented style. We are calling the search method, but instead, we are calling it on the instance variable left, which is, in effect, another (smaller) tree. We mirror this code on the right side as well. After those two if statements, if something hasn't been returned, it's because the side where we wanted to look for the value was null. This menas we should return false, as we do in the last line.

Recursive Insert Method in bintreenode

Inserting is extremely similar to searching. We will still recursively march down a single path in the binary search tree. The difference will be that we will ALWAYS go until we get to the "end" of a path. Here, instead of returning false as we do for a search, we attach the appropriate reference, either left or right, to a newly created node.

First, let's take a quick look at the bintreenode constructor, which we'll use:

```
public bintreenode(int val) {
    data = val;
    left = null;
    right = null;
}
```

This sets up a single node with no left or right subtrees.

When we call insert, at the end of the whole set of recursive calls, we only want a total of one new node to be created.

For this code in the bintreenode class, our base case will be when the place we want to insert the node (either left or right), is null. Then we'll directly set left or right to reference a single newly created node. Alternatively, we'll recursively insert to the left or right.

Here is the insert code:

```
public void insert(int val) {

    if (val < data) {
        if (left == null)     left = new bintreenode(val);
        else                  left.insert(val);
    }

    else {
        if (right == null)    right = new bintreenode(val);
        else                  right.insert(val);
    }
}
```

Our main code branches in two ways, depending on whether val is less than the data at our root node (or greater than or equal to it.) In both branches, we have two cases, either the direction we want to go is null, in which case we set left or right to our newly created node, or if not, we recursively insert val into the appropriate subtree.

Preorder Binary Search Tree Traversal

Visiting each node in a binary tree is not as straight forward as an array. 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. The preorder traversal visits the three components as they are listed above. For the purposes of this lecture, the traversal will simply print out the values stored in each node in the tree, in this order. Given the preorder traversal of a binary search tree, its structure can be ascertained. (This is left as an exercise for the reader!!!)

Here is the code for the preorder traversal:

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

In the sample code posted online, in the main class, bintree, a single instance variable, root of type bintreenode is declared. Special checks to see if root is null are used to avoid calling any bintreenode method on a null object (which results in a NullPointerException and crashes the program.)

Here is one example of how that is handled in main, for testing the search method:

```
if (root != null && root.search(value))
    System.out.println("Great,"+value+" is stored in the tree.");
else
    System.out.println("Sorry, "+value+" is NOT in the tree.");
```

The extra null check is necessary first (with the use of short-circuiting), before the search method is called on root to avoid the NullPointerException.