Binary Search Trees

Trees can be efficient structures for searching

The idea is to organize the tree so we always know which branch to follow

Invariants:

All data in the left subtree is less than the data in the parent node

All data in the right subtree is greater than the data in the parent node
Searching in a Binary Search Tree

To find a node with given target data:

Set the current node to the root of the tree
If the current node is null then return failure
If the data is equal to the target then return the current node
If the data in the current node is less than the target then set the current node to the right child of the current node and repeat
If the data in the current node is greater than the target then set the current node to the left child of the current node and repeat
Creating a Binary Search Tree

Look up the target in the tree

If not found then replace the last null node encountered with a new node containing the target data and null left and right children
Example - Inserting 5, 3, 7 in a tree

3
5
7
Example - Then inserting 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 in the tree

```
  0
   1
    2
  3
   3
    4
  5
   5
    6
  7
   7
    8
     9
```
Example - Then inserting 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 in the tree
Binary Search Tree Code

```java
public class IntBST {
    private IntBT root = null;

    public IntBST() {
        root = null;
    }

    public IntBT find( int target ) {
        IntBT node = root;
        while( node != null ) {
            if( node.getData() == target ) {
                return node;
            } else if( node.getData() < target ) {
                node = node.getRight();
            } else {
                node = node.getLeft();
            }
        }
        return null;
    }
```
public void add( int element ) {
    if( root == null ) {
        root = new IntBT( null, null, element );
    } else {
        add( element, root );
    }
}

public void add( int element, IntBT node ) {
    if( element < node.getData() ) {
        if( node.getLeft() == null ) {
            node.setLeft(new IntBT(null, null, element));
        } else {
            add( element, node.getLeft() );
        }
    } else {
        if( node.getRight() == null ) {
            node.setRight(new IntBT(null,null,element));
        } else {
            add( element, node.getRight() );
        }
    }
}
public int size( IntBT node ) {
    if( node == null ) {
        return 0;
    } else {
        return 1 + size( node.getLeft() )
            + size( node.getRight() );
    }
}

public int count( int target ) {
    return count( target, root );
}

public int count( int target, IntBT node ) {
    int sum = 0;
    if( node != null ) {
        if( node.getData() == target ) {
            return 1 + count( target, node.getLeft() )
                + count( target, node.getRight() );
        } else if( node.getData() < target ) {
            return count( target, node.getRight() );
        } else {
            return count( target, node.getLeft() );
        }
    } else {
        return 0;
    }
}
public void print() {
    print( root, 0 );
}

public void print( IntBT node, int depth ) {
    if( node == null ) {
        return;
    } else {
        print( node.getLeft(), depth + 1 );
        for( int i = 0; i < depth; i++ ) {
            System.out.print(" ");
        }
        System.out.println( node.getData() );
        print( node.getRight(), depth + 1 );
    }
}
} // IntBST
Test Code

```java
public class TestIntBST {
    public static void main( String args[] ) {
        IntBST bst = new IntBST();
        bst.add(5);
        bst.add(3);
        bst.add(7);

        bst.print();

        for( int i = 0; i < 10; i++ ) {
            bst.add(i);
        }

        bst.print();

        for( int i = 10; i < 20; i++ ) {
            bst.add(i);
        }

        bst.print();
    }
} // TestIntBST
```
Huffman Coding (Julie Adams)

Huffman Coding is widely used to effectively compress data.

As opposed to using ASCII or UNICODE.

Every piece of data entered into the computer is converted to a binary representation.

With ASCII this representation requires 8 bits per character.

With Huffman Coding we can save between 20% to 90% of the bits.
Huffman Coding (Julie Adams)

The idea behind Huffman Coding is to represent the most frequently used characters with fewer bits.
Each character has a weight associated with it which is the frequency that the character appears in the string to encode.
Each individual character is considered to be a tree and to have an associated weight.
Then the trees are combined by choosing the two trees with the smallest weight and creating a new tree.
This continues until all the trees are combined into one tree.
The result should be a fewer number of bits for the most frequently used characters.
Huffman Coding

In English, the letter "e" appears much more frequently than the letters "q", "x", or "z"

However, most coding systems use the same number of bits for each letter.

We can save space if we choose a coding that uses less bits for the common letters and more bits for the less common letters.
Encoding strategy

Given a binary tree with the letters in the leaf nodes we can encode a letter as follows:

Start at the root node of the coding tree
Find the path from the root to the letter we want to encode
Starting from the root, output a "0" for each left branch and a "1" for each right branch

Note that different letters might be encoded with different length bit strings
Decoding strategy

If we use a binary tree, we can code left branches with a "0" and right branches with a "1"

Then any path to a leaf node can be described by a string of "0"s and "1"s (which are to be interpreted as commands "go left" or "go right"

In the leaf nodes we put all of the letters

We can now code any letter in a leaf node as a sequence of "0"s and "1"s

If the tree has leaves at different depths then the codes for these letters will have different lengths
How do we know when one letter stops and the next starts?

When we are decoding a series of "0"s and "1"s we know when we reach a leaf node

We then know we are at the end of a letter

We then know the decoded letter and can restart decoding the series of "0"s and "1"s at the current place in the bit stream
How do we build a good tree for encoding and decoding

Suppose we have "frequency counts" that give the number of occurrences of the various letters

Construct, for each letter, a depth-0 tree containing the letter

    Each tree contains only one node which is a leaf node

Associate with each tree the frequency count for the letter

While we have more than one tree

    Find the two trees with the smallest frequency count
    make a new tree with these trees as the left and right branches
    of a new root node (in either order)
    assign a frequency count to this new tree which is the sum of
    the frequency counts of the two component trees
    replace the two trees in the collection of trees with this new tree

When we have a single tree, this is the tree we want