ETEC2101	Lab4: **Binary Search Trees**	Points: **100** (135 possible)
Assigned: **10/3/2016**	Due: **Monday 10/17/2016 at noon (no late work)**	

Tasks:

1. (**40 points**) Create a **BSTNode** class in the **etec2101** package. For this lab, don't nest it under the BinarySearchTree class[1]. Make this class complete invisible to the user class (who we'll assume exists in another package)
    a. Should contain a generic Comparable-derived (E) object as a "payload" and have left / right child pointers (and perhaps a parent?)
    b. void **add**(E val) : Recursively adds this node as a descendent of this node.
    c. int **getHeight**() : Returns the height of the (sub-)tree rooted at this node. Count this node in your height calulation.
    d. int **count**(E val) : Return the number of occurrences of val in this sub-tree. Only go down those child paths that you need to go down.
    e. [You'll likely need a recursive "helper" function so the BinarySearchTree.toString method can work as described – I'll leave the design of this function up to you]
    f. (**15 points**) Support a remove method (most of the work will be here, but the BST class will contain a wrapper). Make sure you're handling all cases and "promoting" a new node if you have two children.
2. (**35 points**) Create a **BinarySearchTree** class in the **etec2101** package.
    a. Most of these methods will be "wrappers" that pass off the bulk of the work to recursive functions in the BSTNode class.
    b. Create an enum called **TraversalType** for the type of traversal we want to do in our iterator method (see item #3 for more details)
    c. void **add**(E val)
    d. String **toString**() : See the sample test program for a description of the output I want.
    e. int **getHeight**()
    f. int **count**(E val)
    g. ArrayList<E> **toArray**() : Create an ArrayList which contains the ordered values in this tree. This is easy if you do an in-order traversal (see item #3), but is still do-able without the iterator.
    h. void **rebalance**(): Should re-balance the tree. I used the as Array method and a recursive helper method (in this class) to accomplish this.
3. (**25 points**) Create a **BinarySearchTreeIterator** iterator class in the **etec2101** package. For this lab, don't nest it under the BinarySearchTree class.
    a. Support in-order, post-order, pre-order traversals (pass it as a constructor argument)
    b. Also add an iterator method to BinarySearchTree to create the iterator and return it.
    c. It's a bit inefficient, but I would suggest filling an ArrayList (or LinkedList) of values upon creation, using a recursive solution (probably in the Node class), then reference that as the user asks for values. It's easier than trying to do the iterator non-recursively.
4. (**10 points, but you'll need one for testing…**) Create a tester **MainClass** in a package called **lab04_xy** (xy = your initials).
    a. You can use my test code if you like (posted on the web-page). I've included the body of this at the end of this document.

[1] If you're curious, I'm doing this for readability (we'll have many similarly-named methods) and to illustrate the difference in access modifiers when splitting into multiple files.

b. If you choose not to use my test code, I still expect the same "interface" (class / method names).

5. (**10 points**) Style. A small part of this is good naming and structure. But a larger part is creating well-written JavaDoc-style documentation on all public classes and methods. I'll let you decide if you document the private stuff and instance variables. Ask what I mean by "well-written JavaDocs documentation"…☺

My Tester Program

```
package lab04_soln_jw;

import etec2101.BinarySearchTree;
import etec2101.BinarySearchTreeIterator;

public class Lab04_soln_jw
{
    public static void main(String[] args)
    {
        BinarySearchTree<String> T = new BinarySearchTree();
        T.add("monkey");
        T.add("gorilla");
        T.add("cougar");
        T.add("jaguar");
        T.add("dog");
        T.add("aardvark");
        T.add("zebra");
        T.add("tortoise");
        T.add("unicorn");
        System.out.println("T =\n" + T);

        System.out.println("Height(T) = " + T.getHeight());

        T.remove("unicorn");
        System.out.println("T (after removing 'unicorn') = \n" + T);

        T.remove("cougar");
        System.out.println("T (after removing 'cougar') = \n" + T);
        T.remove("monkey");
        System.out.println("T (after removing 'monkey') = \n" + T);
        T.add("dog");
        System.out.println("There are " + T.count("dog") + " \"dog's\" in T");
        System.out.println("There are " + T.count("tortoise") +
                           " \"tortoises's\" in T");
        System.out.println("There are " + T.count("wolf") + " \"wolf's\" in T");

        System.out.println("T = \n" + T);

        System.out.println("In-order traversal of T\n=======================");
        BinarySearchTreeIterator I = T.iterator(BinarySearchTree.TraversalType.in_order);
        while (I.hasNext())
        {
            Object s = I.next();
            System.out.println("\t" + s);
        }

        BinarySearchTree<Double> U = new BinarySearchTree();
        U.add(13.2);
        U.add(19.7);
        U.add(23.4);
        U.add(16.4);
        U.add(29.8);
        U.add(32.6);
        U.add(42.0);
        U.add(41.0);
        U.add(53.0);
        U.add(68.3);
        U.add(72.5);
        System.out.println("U = \n" + U);
```

```
        U.rebalance();
        System.out.println("U (after rebalance) = \n" + U);

        System.out.println("pre-order traversal of U\n========================");
        I = U.iterator(BinarySearchTree.TraversalType.pre_order);
        while (I.hasNext())
                System.out.println("\t" + I.next().toString());

        System.out.println("post-order traversal of U\n=========================");
        I = U.iterator(BinarySearchTree.TraversalType.post_order);
        while (I.hasNext())
                System.out.println("\t" + I.next().toString());
    }
}
```

Output of my Tester Program

```
run:
T =
[monkey]
   [gorilla]
      [cougar]
         [aardvark]
         [dog]
      [jaguar]
   [zebra]
      [tortoise]
         [unicorn]

Height(T) = 4
T (after removing 'unicorn') =
[monkey]
   [gorilla]
      [cougar]
         [aardvark]
         [dog]
      [jaguar]
   [zebra]
      [tortoise]

T (after removing 'cougar') =
[monkey]
   [gorilla]
      [dog]
         [aardvark]
      [jaguar]
   [zebra]
      [tortoise]

T (after removing 'monkey') =
[jaguar]
   [gorilla]
      [dog]
         [aardvark]
   [zebra]
      [tortoise]

There are 2 "dog's" in T
There are 1 "tortoises's" in T
There are 0 "wolf's" in T
T =
[jaguar]
   [gorilla]
      [dog]
         [aardvark]
         [dog]
   [zebra]
      [tortoise]

In-order traversal of T
=======================
       aardvark
```

```
        dog
        dog
        gorilla
        jaguar
        tortoise
        zebra
U =
[13.2]
   [19.7]
      [16.4]
      [23.4]
         [29.8]
            [32.6]
               [42.0]
                  [41.0]
                  [53.0]
                     [68.3]
                        [72.5]

U (after rebalance) =
[32.6]
   [19.7]
      [13.2]
         [16.4]
      [23.4]
         [29.8]
   [53.0]
      [41.0]
         [42.0]
      [68.3]
         [72.5]

pre-order traversal of U
========================
        32.6
        19.7
        13.2
        16.4
        23.4
        29.8
        53.0
        41.0
        42.0
        68.3
        72.5
post-order traversal of U
=========================
        16.4
        13.2
        29.8
        23.4
        19.7
        42.0
        41.0
        72.5
        68.3
        53.0
        32.6
BUILD SUCCESSFUL (total time: 0 seconds)
```