**CSC220 Assignment09**

**Trees**

The goal of this week’s assignment is:

1. Practice using trees

2. Practice recursive programming

3. Learn about the importance of debugging

**Things you must do:**

1. There are many details in this assignment. Make sure you read the whole thing carefully before writing any code and closely follow this instruction.

2. You must complete your assignment individually.

3. Always remember Java is case sensitive.

4. Your file names, class names, and package name must match exactly as they are specified here.

5. Your project must include the methods you implemented in the lab.

**Things you must not do:**

1. You must not change the file names, class names, package names.

2. You must not change the signature of any of these methods (name, parameters, …). Just fill in the missing code inside them.

3. You must not create any different class.

**DO NOT** start your assignment unless you have all the features in the lab working!

In this assignment we continue adding functionality to our class. As we saw in lab, each method is two-parted: (1) a **public** version accessible by the tester class that handles special cases (e.g. if the tree is empty), and (2) a **private** helper method for dealing with the recursion work (where the method signature is typically overloaded). This public method makes use of the private helper method by invoking it. We will follow this design pattern in the assignment. As always, please follow the instructions closely.

**Part 1 –** toString **method**

The signature of this method should be:

public String toString()

This method should return an empty string for an empty tree. For a leaf node, it should return the data in the node as a string. For a branch node, it should return a parenthesized String that has three elements separately by commas: the data at the root, a string representation of the left subtree, and then a string representation of the right subtree. For example, calling the toString method on the reference tree 2 (tree\_ref2 in the code) should return the following string (without the surrounding quotes):

“(2, (8, 0, empty), (1, (7, 4, empty), (6, empty, 9)))”

**Part 2 –** luckyTree **method**

The signature of this method should be:

public boolean luckyTree(int value)

We say that a binary tree is “lucky” if the tree contains at least three occurrences of some value. For instance, tree\_ref2.luckyTree(7) is not lucky because it contains only one 7. The starting code gives some examples that are lucky, e.g. tree\_ref3 and tree\_ref4.

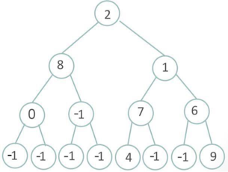
This method checks if the tree is lucky with respect to the parameter value. If it is, return true. Otherwise, return false.

**Part 3 –** perfectify **method**

The signature of this method should be:

public void perfectify()

This method adds nodes to the binary tree until the binary tree is a perfect tree. A perfect binary tree is one where all leaves are at the same level. Another way of thinking of it is that you are adding dummy nodes to the tree until every path from the root to a leaf is the same length. A perfect tree's shape is triangular and every branch node has exactly two children, and all of the leaves are at the same level. An empty tree stays empty. Each new node you add to the tree should store the value **-1**. For example, calling perfectify on tree\_ref2 should change the tree’s state to the following:



It will be helpful to know the current and maximum depth of the tree. Think about which methods from lab you implemented may be helpful here.

**Part 4 – Test your code**

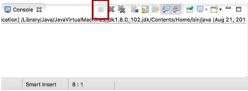
As usual you need to test the functionality of the methods you have implemented. A set of comprehensive test has been provided for you as part of **IntTreeTester.java**. Uncomment the assignment portion of the tests and run the main function. If you see any red text that says “TEST FAILED”, you need to debug your code.

How to debug your code?

1. Use the Eclipse debugger you learned about during the first lab

2. If you see JavaStackOverflow, that means that you have an infinite recursive call and your recursive call is filling up the “call stack” (we talked about this concept in class). Go back and debug the method that is causing the problem.

3. Infinite loops! How would you know you have an infinite loop? As you should know from CSC120, if you have an infinite loop in your code, your code will not stop running. An easy way to inspect that in Eclipse is to look at your console window, if your code is done running the console should look like the



If the little square marked above is red and continues to stay red, that means your code has an infinite loop!

**Remarks**

* Make sure to submit your assignment by (re-)uploading your **Lab09** folder into your **csc220-cXXXX** folder by the deadline.
* **For all your assignments, please start early and seek help early (either from the instructor or the TAs).**