CSC220 Lab09
Trees

The goal of this week's assignment is:

1. Practice using trees
2. Practice recursive programming
3. Learn about the importance of debugging

Part 0

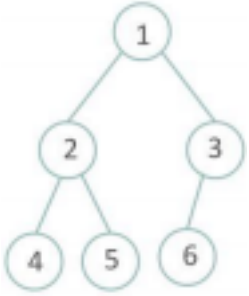
- You are going to create a new project for this (and each of the remaining labs). Create a new Java project and call it **Lab09.**
- Create a package inside your project and call it **lab09**.
- Download the Lab09-Assignment09 ZIP folder from Blackboard (google drive link). Copy the following files into your new lab09 package:
    - **IntNode.java**. This class provides the implementation of a single node of a binary tree that holds integer values.
    - **IntTree.java**. This class has been partially implemented for you. The current implementation provides a simple binary tree class that includes methods to construct a tree of integer values (saw during the lecture), to print the data using an in-order traversal.
    - **IntTreeTester.java.** This class will include the main function that will test your code.

Part 1 – Problem description

For this lab, you are asked to write methods to be added to **IntTree.java.** As explained above, this class is started for you and the basic functionality is already implemented for you. Do not modify the methods provided. The current implementation provides a simple binary tree class that includes methods to construct a tree of integer values, to print the data using an in-order traversal. Two constructors have been provided for you:

1. **public IntTree(int max)**

   The trees built using this constructor have nodes numbered starting with 1 and numbered sequentially level by level with no gaps in the trees till the max value (saw during the lecture). For instance IntTree(6) would create the following tree.

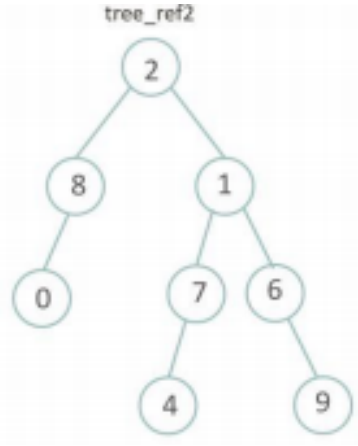


2. **public IntTree(int[] arr)**

   The trees built using this constructor will use the integer values in the input array sequentially to fill the tree as before level by level. If the input integer value is -1, the corresponding node will be skipped. For instance, the following lines will create the tree illustrated below:

   Int[] arr2 = {2, 8, 1, 0, -1, 7, 6, -1, -1, -1, -1, 4, -1, -1, 9};

   IntTree tree_ref2 = new IntTree(arr2);



Remember that one of the purposes of this week is to practice recursive programming. Almost all of the following methods are easier to write recursively. Remember that you ARE NOT allowed to change the signature of the methods. However, you can use helper functions if you need to. For debugging purposes, you might want to check the structure of the tree. We provided a simple variation of in-order traversal of a tree (suggested in the book – Chapter 17.2 – page 1031): “instead of printing values all on a line, we will print them one per line and use indentation to indicate the level of the tree”. For example, the previous example tree can be printed using the following line: **tree_ref2.printSideways();**

The output looks like the following:

```
                9
            6
        1
            7
                4
    2
        8
            0
```

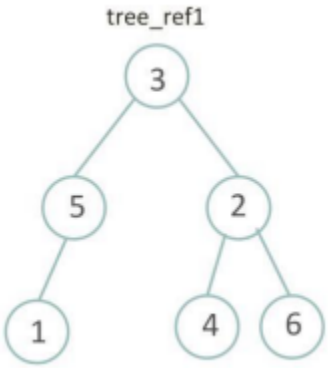
You need to imagine rotating this output 90 degrees in a clockwise fashion (or tilting your head to the left) to see the tree structure. This function can come handy… Before you move on to the next part, take a couple of minutes and look at the definition of the **IntNode** and **IntTree** class. Consult the lab TAs if you have questions. You should know what fields are available, why we need them, and how to call various methods.

Part 2 – numEmpty method

The signature of this method should be (only modify where it says “FILL IN” – DO NOT change the signature):

public int numEmpty()

This method returns the number of empty branches in a tree. An empty tree is considered to have one empty branch (the tree itself). For nonempty trees, your methods should count the total number of empty branches among the nodes of the tree. A leaf node has two empty branches, a node with one nonempty child has one empty branch, and a node with two nonempty children has no empty branches. For example, reference tree #1 (called tree_ref1 in the code) has 7 empty branches (two under the value 1, one under 5, and two under each of 4 and 6). The tree is demonstrated below.



Part 3 – printLevel method

The signature of this method should be:

public String printLevel(int level)

This method accepts an integer parameter n and returns the values at level n as a string, from left to right, **one per line**. We will use the convention that the top root (of the entire tree) is at level 1, its children are at level 2, and so on. If there are no values at the level, your method should return an empty string. Your method should return an empty string if it is passed a value for a level that is less than 1. For example, getting level 3 of reference tree 2 (called tree_ref2 in the code), should look like the following:

0
7
6

Part 4 – getDepth method

The signature of this method should be:

public int getDepth()

This method returns the depth of the tree as an integer. We will use the same convention as in **printLevel** where the top root is at depth 1, its children are at depth 2, and so on. For example, the depth of tree_ref2 is 4. If the tree is empty, the method should return 0.

Part 5 – Testing

As usual you need to test the functionality of the methods you have implemented. A set of comprehensive tests has been provided for you as part of **IntTreeTester.java**. Uncomment the lab 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 in eclipse. If the “stop” square button is red and continues to stay red, that means your code has an infinite loop!

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.

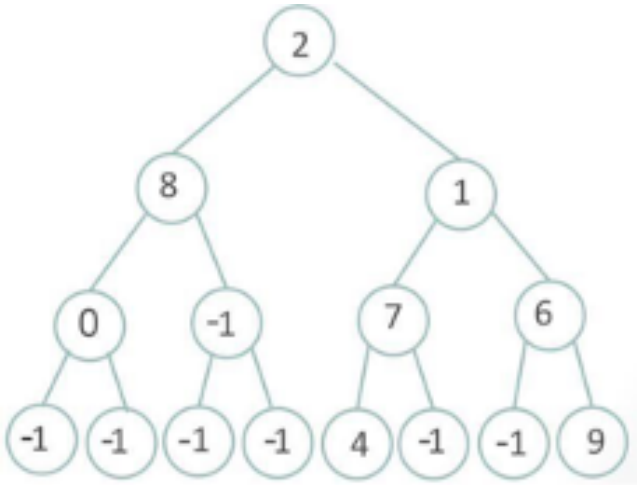
Part 3 – perfectify method

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.

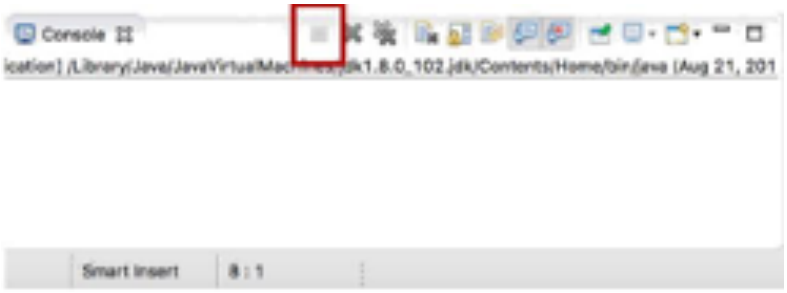
Part 4 – Test your code

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!