Course: CSC 115 (J)

Lab 07: Binary Trees

Objectives

You will be constructing a *BinaryTree* class, and writing recursive methods to traverse it and calculate its height.

During lab 7, you will:

- Complete a *TreeNode* class so that it uses generics to store data.
- Complete a *BinaryTree* class so that it includes basic functionality for a binary tree.
- Come to a better understanding of basic tree operations such as add, traversals, and calculating the height of a tree.

Part I – Generics

1. Download and complete *TreeNode.java* so that it uses generics to store data instead of an Object.
    - Notice that the *TreeNode* has references to a left and a right child, unlike the Nodes we used for *LinkedLists* that just have a reference to the next element.
    - Change the class declaration and any references to *TreeNode* so that they include the generic syntax, except the constructor.
    - Change all references to the type of data being stored from Object to the generic identifier.
2. Make sure to **compile** and **test** your generic *TreeNode* before moving on to Part 2 of the lab.

Part II – Basic Binary Tree

1. Download *DrawableBTree.java* from connex. This is a helpful class that will visualize trees you create, making them easier to debug.
2. Download and complete BinaryTree.java. It is recommended that you follow the steps listed below.
    1. Start by defining the three constructors for the *BinaryTree* class. These methods are used to build a tree. Follow the comments provided in the code as you implement them, and make sure to keep track of the size of the tree being created.
    2. Once you have the constructors written, you're ready to test them. Try compiling and running the program. You should see a window pop up that visualizes the tree being created in the main method. The program should also print out the size of the tree. Check to make sure all the output is correct before moving on to the next step.

3. Fill in the *isEmpty()* method so that it returns true if the *BinaryTree* is empty, and false otherwise.
4. Write code that **recursively** calculates the height of the *BinaryTree*.
    - First note that *height()* is already written, and just contains a call to the *recursiveHeight* method passing in the root node of the tree. You will need to complete the *recursiveHeight* method.
    - If the node given to *recursiveHeight* is not *null*, it should determine the height by finding the maximum height of its two subtrees and adding 1.
    - It might help to use the *Math.max(number1, number2)* method from the Java API, which returns the greater of the two parameters passed into it.
5. Test the height method by un-commenting the second section of the tester code, and make sure it works before moving on to Part 3 of the lab.

Part III – Tree Traversals

- A traversal is a systematic method of iterating through a data structure and processing each element once and only once.
- For this Part, you will implement a method that returns the data from your tree as a String according to an *In-Order* traversal.
    - First note that the *traversalInOrder()* method is already written, and just calls the *recursiveInOrder* method passing in the root of the tree.
    - You will need to complete the *recursiveInOrder* method so that it **recursively** adds the data in the tree into a String according to an *InOrder* traversal.
    - An *InOrder* traversal follows the procedure given below:
        1. Traverse the current node's left subtree.
        2. Process the current node.
        3. Traverse the current node's right subtree.
    - Make sure to think about the **base case** for your recursive algorithm. When do we not want to follow the procedure listed above?
    - Test the traversal by un-commenting the third section of the tester code, and make sure it works.
- If you have time, you can go ahead and implement *PreOrder* and *PostOrder* traversals. They will be very similar to the *InOrder* traversal, except they will process the current node before doing any recursion (for *PreOrder*) or after doing both recursive calls (for *PostOrder*).

Have fun