Homework 3 Programming

Please remember that to submit the assignment you must click Mark as Complete under the Education menu in the toolbar.

Problem 1 - Implementing Expression Trees - 35 points

Implement a class called *ExpressionTree* in the provided ExpressionTree.java file. This class implements the *ExpressionTreeInterface* file. The constructor to ExpressionTree will take in only one String that contains a postfix expression. The operands will be integers and the operators will be restricted to +, -, *, and /. Individual tokens, that is, the operands and operators, will be delimited by only one space. So for example:

34 2 - 5 *

would mean (34-2)*5.

Your constructor will run the stack based algorithm we discussed in class to build an expression tree. In order to implement the *ExpressionTree* class, you will have to implement a static nested class called *ExpressionNode*, which will contain the implementation of the individual nodes that form an expression tree. You should use these nodes to represent the individual operators and operands. You may use any code posted on Canvas or from the Weiss textbook as a starting point for this assignment. For a stack data structure, you can use java.util.LinkedList.

Once you have the ExpressionTree constructed you should provide the following four methods as required by the interface as well as the constructor as specified below:

- `public int eval()` - this method, when invoked on an expression tree object, will return the integer result of evaluating the expression tree. Your algorithm will need to call a private recursive method that takes in the root. Assume integer arithmetic at all times, including during division. Assume that no invalid arithmetic operations occur in the expression (e.g. no division by zero).
- `public String postfix()` - this method, when invoked on an expression tree object, will return a String that contains the corresponding postfix expression. It will need to call a private recursive method that takes in the root. Each operator and operand must be delimited by only one space and there should be no extraneous white space on either ends of the String. Any deviations from these instructions, will result in a loss of points.
- `public String prefix()` - this method, when invoked on an expression tree object, will return a String that contains the corresponding prefix expression. It will need to call a private recursive method that takes in the root. Each operator and operand must be delimited by only one space and there should be no extraneous white space on either ends of the String. Any deviations from these instructions, will result in a loss of points.
- `public String infix()` - this method, when invoked on an expression tree object, will return a String that contains the corresponding correct infix expression. Keep in mind that parentheses will be needed (excessive parenthesis will be tolerated as long as they are correctly placed). It will need to call a private recursive method that takes in the root. Each operator and operand must be delimited by only one space and there should be no extraneous white space on either ends of the String. Any deviations from these instructions, will result in a loss of points.
- `public ExpressionTree(String expression)` - this is the constructor of the expression tree. It will take in a String that stores a postfix expression (as indicated above). Build the expression tree from that postfix expression using the stack based algorithm here.

We will test this program with our own tester class in a separate file. You should also create a tester class for your own testing purposes - make sure to consider edge cases such as invalid postfix expression inputs to your constructor. Your tester class will not be graded.

Problem 2 - Binary Search Tree (BST) Algorithms - 30 points

In this problem, you will implement various algorithms operating on binary search trees. We have provided with you a standard implementation of a generic BST in *BinarySearchTree.java*. Note that this class is an **abstract class**, which means that some of its methods are not implemented. In previous assignments, you have implemented interfaces which specified methods that you needed to write. Very similarly, an abstract class is a class with some unimplemented methods (it can be thought of somewhat like an interface but with some methods actually implemented). You will need to write a *BetterBST* class which **extends BinarySearchTree**. Your *BetterBST* class can then be treated just like a regular *BinarySearchTree*, just with some additional functionality.

The methods that you will need to implement in *BetterBST* perform various algorithms on BST instances. For some of these methods, you may find it convenient to implement a private helper method as you did in previous assignments.

- `public int height()` - return the height of the BST
- `public T imbalance()` - check whether the tree is balanced. A balanced tree is one where every node's left and right subtrees differ in height by *no more than 1*. Return the value at first node you find which has a height imbalance *greater than 1* between its subtrees, or `null` if no such node exists (i.e. the tree is balanced). In class, we discussed AVL trees, which enforce this balance condition.
- `public T smallestGreaterThan(T t)` - given some generic comparable value *t*, find the smallest value in the BST that is larger than *t*. For example, if a binary search tree contains the values 1, 3, and 5, and the function is called with *t = 2*, it should return 3.
- `public BinarySearchTree<T> mirror()` - return a mirrored version of the BST instance to satisfy a reversed BST condition. In a reversed BST condition, for every node, *X*, in the tree, the values of all the items in its right subtree are smaller than the item in *X*, and the values of all the items in its left subtree are larger than the item in *X*. In the mirrored tree, any node that is a left child becomes a right child and vice versa. You should not modify the BST instance itself. Instead, you should create a new BST instance and build it.
- `public LinkedList<BinaryNode<T>> levelOrderTraversal` - return a LinkedList<BinaryNode<T>> of a level order traversal of the binary tree. For example, given the tree below, the method should return: {3, 1, 5, 4} (Hint: think about how you might use a queue to solve this problem. Take a look at an algorithm for breadth first search)

```
     3
 1       5
       4
```

Make sure you read the BST code in depth before you start implementing *BetterBST*. In particular, take note of the various internal methods, nested classes, and instance variables that you can access from *BetterBST*.

We will test this program with our own tester class in a separate file. You should also create a tester class for your own testing purposes. Your tester class will not be graded.