Project 4: Trees
Due: Tuesday, Nov 11 at 11:50 am

Project partners. For this project you will again work in pairs. The teams are:

Pong & Greg, Elissa & Murat, Nicolas & Izamar, Steven & Roberto, Noah & Ruichen, Yifan & Lauren, Katie & Grace, Asaf & Will, Juan & Moustafa, Jessica & Carson, Albert & Albert!

Working in pairs. Here are some guidelines:

1. If at all possible, you should always code together, i.e., both people at the same workstation at the same time.

2. Set up a schedule of meeting times right away. Two or three meetings a week of 2-3 hours each is probably a good start. (I would schedule meetings with each other that total at least 5 hours a week, and keep another 3 on reserve.)

3. Alternate who does the typing so that each person gets to code for half the time. Note that following along as someone else codes is also an important and valuable skill!

4. It behooves you both not to divide up the work. But if you must, make sure both parties are heavily involved in the design and problem-solving of all aspects of the project. Also make sure both parties have carefully read, checked, and understand all code. I.e., if you code up a method without your partner, it is important that your partner can still explain in detail how the code works.

Comments and style. Javadoc-style comments are required throughout your code along with automatically created Javadocs for this project and all future projects. As an example, see the commenting style in the code that has been provided to you in previous projects. Previous projects have also explained how to use the command to automatically generate the Javadoc webpages of documentation for your classes and methods. For a quick javadoc style guide see:
or
http://agile.csc.ncsu.edu/SEMaterials/tutorials/javadoc/.

10% of your grade will be allotted to coding style, documentation/comments, and correctly following submission instructions.

Submission. Your completed project should be submitted twice: First, via moodle, then also in hard copy at the start of class on the day it is due. All files that you submit via moodle should be collected into a folder called <username>_proj4. For example, if you are Joe Smith your folder should be called jsmith_proj4. You should include all .java files, as well as their corresponding compiled .class files. Be sure that the .class files you submit are exactly what I would get if I compiled the .java files you submit. You should also include any other supporting files (.txt files, etc, if you have any). The folder should then be compressed into a .zip file (or some other standard type of compression file). The compressed file should then be uploaded using the Project 4 link on the COM212 moodle page. See course website for late project submission policies.
Part A (45%). Binary Search Tree. Create a class (called BST) that can be instantiated to create a Binary Search Tree of integer values. These integer values will be the keys of the BST. Your class should have the following standard BST methods:

- `find(int k, BSTNode v)`
  - Searches for the key k in the sub-tree rooted at v. Returns the node where k is found, or returns null if k is not found. (Pass in root for v to search the entire tree.)
- `insert(int k)`
  - Inserts a new node with key k into the BST.
- `delete(int k)`
  - Removes and returns a node in the tree with key k. Returns null if k is not found.

As we saw in class, `insert` and `delete` should operate in such a way as to maintain the BST property of each node’s left child having a lower key, and each node’s right child having a higher key.

Your class should also have the following basic traversal functionality:

- `preOrder(BSTNode v)`
  - Prints out the keys of the tree in a preorder traversal
- `inOrder(BSTNode v)`
  - Prints out the keys of the tree in an inorder traversal
- `postOrder(BSTNode v)`
  - Prints out the keys of the tree in a postorder traversal

These BST methods must be implemented recursively, and the implementation should be based on the code we discussed in class.

To solve this problem, in addition to the BST class, you will also need a BSTNode class, for creating the nodes in the BST. In this class, create a `displayNode` method that simply prints out the node’s key.

Your BST class should also have a method that displays the tree, to allow for testing and debugging. Code for this display method is provided for you here. It will draw the contents of the tree in the shape of an actual tree so it is easier to check that your code is behaving properly.

Note that this display method requires a Stack data structure to be available. (Boy, those stacks sure come in handy!) You can either include a Stack class that you’ve already written for a previous project, write a new one, or simply import the built-in Java Stack class by adding this line:

```java
import java.util.*; // for built-in Stack class
```

to the very top of your code.

To test your class, a main method in a class called TreeApp has been provided here. You should be able to read and understand it. This method creates an instance of your BST class, inserts several initial values into the tree, then runs an interactive program for the user at the console prompt to allow
further insertions, deletions, etc. In the course of this, it calls an intermediary traverse method on the BST, the code for which is provided for you here. Note that this method calls the three recursive tree traversal methods that you will have written. Further note that it uses something called a switch statement (which you can Google), and this is basically the only time that break statements should be used in your code. The figure below shows a sample run of the provided test program.

![Command Prompt - java TreeApp](image)

Note that in the output, the tree does not have its edges drawn, but you can imagine them, as long as you keep in mind that each line of output represents a different level of the tree. The dashes represent nulls, or empty spots in what would otherwise be a complete BST (one where every internal node has exactly two children).

Remember to test all your code incrementally as you go. This helps to fix any small mistakes that compound into buggy behavior later on, but then become very hard to track because they were made so early in the programming process. And remember to vary the tests you run, not only running basic tests, but also special cases (or “boundary conditions”), with the goal of “breaking” your program. This is how you really unearth any bugs you may have in your code. Post to the moodle forum any ideas your have for testing and debugging... or any good tests you’ve devised that you’d like to share with classmates to test their code with!

**Part B (35%). Infix to Postfix.** Note: Be sure to save your files from Part A in a separate folder from your files for Part B before working on Part B.

Create a program that converts a standard infix expression (that we will assume is fully parenthesized) into a postfix one (which doesn’t need or have any parentheses). A nice way to do this is by using a binary tree!
For example, if the input string is the following fully-parenthesized infix expression:

\[(3 * 4) + 5\]

then your program should output:

- 3 4 * 5 +

To see how an arithmetic expression can be very neatly represented as a binary tree, see Example 8.6 on page 318 of the GTS text. The diagram in Figure 8.6 gives the binary tree representation of an example expression. As an initial exercise, write down the postorder traversal of this sample tree. (Put this at the top of your code in comments, drawing a box around it to draw my attention to it, as I will be grading it.) Verify that what you’ve written is in fact the postfix version of the original infix expression!

For the purposes of this project, we will make the simplifying assumption that the infix expression (aside from parentheses) uses only single digit integers and the four basic operators +, -, *, and / and is fully-parenthesized.

The pseudocode for building a binary expression tree from a fully-parenthesized infix expression is as follows:

```plaintext
/* Input: a fully-parenthesized arithmetic expression E comprised of * characters that are either single-digit integers, arithmetic * operators, or parentheses. * Output: a binary tree representing arithmetic expression E. */

Algorithm buildExpression(E):

Initialize a new empty stack for holding binary expression trees.

For each character of the input string E:

- If the character is a number or operator x,
  a. Create a single-node binary tree T whose root stores x
  b. Push T onto the stack

- On the other hand, if the character is a close-paren “)"
  a. Pop the top three trees from the stack (these represent a subexpression: E1 ○ E2)
  b. Attach the trees for E1 and E2 as the children for ○
  c. Push the resulting tree back onto the stack

After the loop, return the top element of the stack, which is the final expression tree.
```
Notes:

- As you can see, you will need a binary tree data structure for this algorithm. So you will need to create a binary tree class, called BTree. To create this class you will first need another class for the nodes of the tree. Call it TNode. Each element of the tree will be a character (either an open paren, close paren, arithmetic operator, or single-digit value). So make your TNode’s element field of type char. Your TNodes won’t need a parent reference for this task.

- For your BTree class, all you need at first is a private field (for the root), a constructor, and a method called getRoot() that simply returns the root of the tree. Later you can add other methods you’ll need, like a postorder traversal method.

- Notice that the buildExpression algorithm uses a Stack data structure, whose elements are trees. You can again use the built-in Java Stack class if you’d like (see Part A for the one line of code you need), or you can build your own from scratch. If you build your own, keep in mind the elements of the stack we need are BTree objects.

- The specs for two methods you’ll need to code, addRoot and attach, can be found on page 324 of the GTS text. Take note of where/how these methods are used/invoked in the above buildExpression algorithm. You will have to implement them in your BTree class.

Put your buildExpression method in a class called ArithmeticApp, which needn’t have any fields or constructors. Add a main method to this class in which you create an ArithmeticApp object, from which to test the buildExpression method. In main, you can add code to output the expression in postfix.

**Part C (5%): Infix evaluator.** Take Part B and combine it with Part C of project 3 (your Postfix Evaluator application) to make an infix evaluator: the user inputs a fully-parenthesized infix expression and your program outputs what it evaluates to. As an intermediary step, have your program also output the postfix version of the input expression.

**Part D (5%): Skew avoidance.** As we have discussed (or will discuss) in class, when deleting nodes from a BST by always using the successor of the node being deleted, the tree becomes skewed over time. Modify your code so that it alternates (or randomizes) whether it uses the successor or predecessor for the delete function, thereby avoiding the skewing problem.

**Bonus (+10%): fast (O(log n)-time) insert, find, and delete.** Completing this bonus is highly recommended for those of you intending to major in CS. As we learned in class, and AVL tree is one way of ensuring a BST remains balanced enough to guarantee fast BST operations. Once you have your BST from Part A completely working, save out a copy (which will be submitted for grading). Then try to upgrade your BST to an AVL tree! Important note: submit this in addition to your basic Part A, not instead of Part A.