Lab 4: Stumped

Pre-Lab Notes

Lab submissions will **not be accepted late** – they must be submitted by **midnight the day after they're assigned** (to account for time zone issues). Some latitude might be allowed (I won't disqualify a submission that's mere minutes late!) but that's purely discretionary.

Labs are treated differently from assignments – you're encouraged to work with fellow students, ask questions of the TA during your lab section, and generally collaborate to complete your work. You should still submit your own work, **including citing with whom you collaborated in your submission**, but using answers developed by others will not be treated as plagiarism.

Nevertheless, you are still encouraged to make a good faith effort to complete your labwork, in order to exercise your understanding of the topics it covers. Lab submissions are marked purely 1 or 0 by whether the TA believes you at least made an honest attempt, even if you didn't succeed, so the value is in what the attempt teaches you rather than simply having the right answer.

1. Trees and Traversals

We've recently discussed the non-linear Tree data structure, as well as the Tree traversal algorithms that can make them a useful and efficient option compared to their linear alternatives. However, Trees have fewer standard classes and tools available under Java, meaning you will often have to implement them yourself. This lab exercise aims to start you practicing both writing your own Trees and your own Tree-traversing algorithms.

1.1 Binary Trees and Arithmetic Expressions

A variant of Tree we discussed in class is the Binary Tree, which limits each node of the Tree to just zero, one, or two children. These children are also in order, either as the “left” child or the “right” child.

Binary Trees have a variety of uses, especially if paired with other properties (like the “complete” Binary Trees, or the Binary Search Trees). Even on their own, though, one use for Binary Trees is representing an arithmetic expression. This is something we looked at briefly during lecture to give us some examples of different types of Tree-traversing algorithms.

Now, it's your turn to implement one of these algorithms we discussed.

2. Outline

You need to implement a function that can turn a Binary Tree representation of an arithmetic expression back into a plain line of text. For a clearer picture, check the slides from Unit 14 on Binary Trees, which showed an example arithmetic expression in both written and binary tree forms.

To simplify the question, we'll limit the available operands to just +, -, *, and /. These four are enough to set up some expressions where the order of operations isn't simply read from left to right. They along with the numbers that make up the expression can be stored as Strings, returned as a completed String, or simply printed to the terminal, so long as your function can accurately parse the contents of a Binary Tree.

2.1 Setting Up The Tree

The first step is building classes for your Binary Tree and Binary Tree Nodes. Each node will need one variable for the element (a string or character should do in this case, since we're just looking to print them), as well as three other Binary Tree Node fields for the left, right, and parent nodes.

The Binary Tree class will need to store a root Binary Tree Node and track the size of a tree with an int. You'll want a way to set a node as the root (maybe a constructor option that hands a node to the tree to be the root, maybe a dedicated function for setting a root). You'll also need functions for setting and returning the left, right, and parent nodes of a node.

Be sure to implement functions that return boolean answers about whether a node has a left child, a right child, is internal, or is external – these might help you in the next subsection! There's plenty more functions you could implement, of course, like one for removing nodes, or calculating the height of the tree, but this minimum set of functions will do for now.

For testing purposes, you'll need to build some nodes manually in your main method and make sure the way you arrange them in your tree accurately represents an arithmetic expression – maybe try doodling out the tree you want to build on paper first before trying to build it, to make sure you know what your function will be getting and what the result is supposed to be!

2.2 Traversing

Now that you've built a tree of nodes that accurately represent an arithmetic function, it's time to develop the function that prints it!

Some advice to get you started, remember that many tree traversal algorithms can be implemented as recursive functions. When designing a recursive function for a tree, it often makes sense to have the function accept a node, and give it the root node to begin with. Also consider that every internal node in an arithmetic expression tree is an operator, whose left child will appear before it and right child will appear after it. Alternatively, if the node is external, it has no children to worry about and can be printed/returned directly. Printing parentheses to bracket every pair of terms may also help you when debugging your output!

Feeling ambitious? Maybe want to build a function which can handle the reverse, converting a written equation into a Binary Tree? That's possible too, but be warned – you may want to tweak how you write out the equations a little to make your job easier. For example, if the written expression was fully parenthesized (meaning parenthesis brackets are correctly used to group up each pair of terms according to their order of operation), it would be much easier to build the tree by keeping track of the parentheses as you read in the expression!

3. Deliverables

All lab submissions will be done through CourSys at https://courses.cs.sfu.ca/. For this lab, that should include:

1. A .zip archive of your code, organized into a single Java project. To ensure compatibility, it's recommended to use IntelliJ while developing your code, then using the export function under File -> Export -> Project To .zip File. **Don't forget to include a test file**, a Main.java with a main method that prints the results of a few runs of your functions to the terminal would be best.

Be sure you tidy up your code before submitting! Check the Java code style guide posted on the course website, and do your best to provide helpful comments.