CSCI241 Fall 2020: Lab 3
Due Sunday 10/18 at 9:59pm

Unlike the homework assignments for CSCI241, you are **encouraged** to work with your peers in completing the labs. However, each student must write and submit their own code—no files may be exchanged. If any of this is unclear, please ask for further clarification.

1 Overview

Thinking recursively is an important skill, and it does not come easily to most students. This lab is intended to give you practice implementing various recursive methods. First, you will work with some string and integer processing methods, filling in missing pieces at first and then writing some in their entirety. Finally, you'll write some basic methods for a Binary Search Tree class, which will be further developed in A2.

2 Git and submission for Lab 3

The Github Classroom link for Lab 3 is available in the Lab 3 assignment on Canvas. Clone your repository as you did for Lab 2; be sure to clone it somewhere outside your other lab and assignment repositories to avoid having nested local working copies of different repos. As usual, you will submit your code for this lab by committing and pushing the completed files to your remote Lab 3 respository on GitHub before the deadline.

It is recommended that you `git add` *and* `git commit` *regularly while developing your code; e.g., one or more times per method you implement. When you have something working,* `git push` *your changes to GitHub.*

3 Writing Recursive Methods

As discussed in lecture, the process of writing recursive methods is often mind-bendingly tricky, because the method you are writing calls itself. The best way to think about recursive methods as you're writing them does *not* involve tracing the code through the recursive call—because you haven't finished implementing it yet! Instead, you must rely on the **specification** of the method to tell you what the recursive call **will** do—namely, correctly solve a smaller subproblem—when the method is implemented. Then, assuming that the method implements the spec for the subproblem call, you can write the method to implement the spec to solve the entire problem. It's easiest to break it down into a four-step procedure:

1. Write a precise method specification, including any applicable preconditions and details of how the method behaves on all possible inputs.

2. Write code to handle any base cases—any inputs for which the problem can be solved directly without using recursion.

3. For all other cases, define the solution to the larger problem in terms solutions to subproblems. This is usually the hardest step, as it involves coming up with a *recursive definition* of your problem.

4. Implement the specification for the larger problem, using recursive calls to solve the subproblems. When analyzing your code, **do not** trace your code into the recursive call, but simply replace the recursive call with what the spec says it should do.

4 Recursion.java

The Lab 3 repository contains two files; the first one you'll be working in is Recursion.java under the directory src/main/java/lab3/. You will implement the following methods. In many of these methods, some of the four steps have been done for you.

Testing The `Recursion` class's methods will be tested using JUnit and Gradle just like Assignment 1 and Lab 2.

1. `int len(String s)`: Compute the length of a string, without using the String's `length` method. Steps 1 (spec), 3 (recursive definition), and 4 (implementation of the spec in the recursive case) have been done for you: you simply need to fill in the condition for the base case. Replace the `true` in the `if` statement's condition with the correct base case conditional expression.

2. `countE(String s)`: Count the number of occurrences of the character `'e'` in the String `s`. Steps 1 (spec) and 3 (recursive definition) have been done for you. Complete steps 2 and 4 to finish implementing the method.

3. `int sumDigs(int n)`: Sum the digits in the decimal representation of an integer n. Steps 1 (spec), and 2 (base case) have been done for you. Complete steps 3 and 4 to finish implementing the method.

4. `String reverse(String s)`: Return the reverse of a string. Step 1 (spec) has been done for you. Complete steps 2, 3, and 4 to implement the method recursively.

5. The remaining methods in Recursion.java (below the main method) are **not** required for Lab 3. They are provided for additional practice/challenge to complete on your own time if you wish.

5 BST.java

In A2 you'll be implementing a Binary Search Tree. As a warmup, you'll write some basic tree processing methods.
Remember that when thinking about recursion in trees, it's important to think of the left and right children not as nodes, but as **subtrees**. A typical recursive tree method will do some processing on the root, one or more subtrees, and possibly some work to combine the results of that processing into a solution to the larger problem.
`BST.java`, under the directory src/main/java/lab3/, contains class BST, which has two fields `root` and `traversal`, of type `Node` and `String` respectively. `Node` is an *inner class* of `BST`, which means it's a class whose definition is nested within the `BST` class. This simply means it's a helper class that is not intended for use outside the BST. The tree processing methods you'll write will be nonstatic members of the `BST` class.
Because we are writing our methods as public methods of BST but we'll be operating recursively on Nodes, many of the public methods will have associated private helper methods that take a `Node` as an argument. Complete the following methods in BST.java.

Testing Test code for this part is written for you (you're welcome!). BST.java has a special constructor that takes an int and generates various test trees, and the main method tests the methods below on each tree. Test your methods frequently and make sure your code passes all the tests.

1. First, implement `boolean isLeaf(Node n)` method. This does not require recursion. Notice that there is *not* a precondition specifying that `n` can't be null.

2. The public `int size()` method of BST takes no arguments, but simply calls a private helper method `int size(Node n)` that calculates the size of a tree rooted at the given node. Getting the size of the whole tree is as simple as calling `size(root)`. Implement the `private size(Node n)` helper method. Steps 1 (spec) and 3 (recursive definition) have been done for you.

3. Implement the three canonical recursive tree traversals: `inOrder`, `preOrder`, and `postOrder`. For these traversals, we will not be printing each node value but instead accumulating the nodes values in the string *traversal*. As before, the implementation for each is done in the helper method that takes a `Node n` as its argument. Step 1 (spec) has been done for you.

4. Implement the `int height(Node n)` helper method to calculate the height of the tree rooted at `n`. Notice that the specification defines a special case in the definition of height: an empty tree (that is, a tree with no nodes) is defined to have a height of -1. Step 1 (spec) has been done for you.

Rubric

Make sure you have committed your completed Recursion.java and BST.java files to git and pushed to Github. As before, **do NOT push .class files**.

Each correctly implemented method is worth 1 point:

1. len
2. countE
3. sumDigs
4. reverse
5. isleaf
6. size
7. inOrder
8. preOrder
9. postOrder
10. height

Additional deductions may be taken for:

- Code that does not compile
- Code that generates run-time exceptions
- Changed method headers (name, parameters, return values)
- Other assorted failures to follow instructions
- Poor coding style (e.g. inconsistent indentation, poor variable names)