Lab 8 - Using a Stack to Create a Calculator
Due (via Bitbucket and hard copy) Wednesday, 30 March 2015
50 points

Lab Goals

• Practice working with the Java Stack class
• Implement a calculator with postfix notation
• Answer a few questions to test your knowledge of course material

Assignment Details

In this lab we will build off of last week’s lab, turning our infix-to-postfix converter into the first stage of a simple calculator. This calculator will give us the ability to add, subtract, multiply, and divide one-digit numbers through the use of the stack. When combined with Lab 7, you will have a full program that allows you to enter a string such as “4+6/(5−2)*7” and have the program print “18.”

Because it is possible that you will not have Lab 7 completed, you have two options for this lab:

1. Request a solution to Lab 7 to insert into your Lab 8 code. You may only select this option after you submit Lab 7, whether it is in working condition or not.

2. Have your program take a postfix string as input, rather than convert an infix string into postfix first. You will still get the same number of points, but you won’t have a fully-functioning single-digit calculator in the end.

The choice is yours which route you decide to take. You can change your mind from #2 to #1 at any time, but once you switch to option #1, you are stuck with that decision.

Part 1: Practice (15 points)

Once we have a string in postfix notation, we can apply it to another Stack procedure to solve the equation. In the previous lab, we added the operators to the stack. In this lab, the stack will handle operands instead.

Consider the infix string “4 + 5 * 6,” which evaluates in postfix notation to “456 * +.” In postfix notation, each operator is computed on the two operands that immediate precede it. In this case, the “*” operator multiplies 5 and 6, while the “+” operator adds 4 to that result. We can get
a solution from the postfix string by reading through the string character by character as before, adding each operand to a stack, and evaluating the top two operands from the stack each time we read an operator.

Let’s take this example character by character. First we read a “4” and add it to the stack, then we read a “5” and add it to the stack, then we read a “6” and add it to the stack. Our stack contents are \{4, 5, 6\}, with the 6 accessible first. Now, we read a “*” character. We pop the top two characters in the stack into temporary variables, say \(a = 6\) and \(b = 5\). Because we read a “*” character, we multiply these two variables together, yielding 30. Then, we push the 30 back onto the stack. Our stack contents are \{4, 30\}, with the 30 accessible first. Finally, we read a “+” character. We pop the top two characters in the stack into temporary variables, \(a = 30\) and \(b = 4\). Because we read a “+” character, we add these two variables together, yielding 34. Then, we push the 34 back onto the stack. We have now reached the end of the input string, and the only remaining value in the stack should be the answer to “4 + 5 * 6,” which is 34.

Let’s try it with the more complicated example at the beginning of the lab, evaluating “4 + 6/5−2)∗7.” Once converted into postfix notation, the string should read “4652−/7*+.” We read across the first four characters, pushing 4, 6, 5, and 2 into the stack. Our stack contents are \{4, 6, 5, 2\}, with the 2 accessible first. Now we read a “−” character. We’ll pop the top two values from the stack, and set \(a = 2\) and \(b = 5\). Unlike in our previous example, the order is now important because subtraction is not a commutative operation (neither is division), so we want to be sure that we’re subtracting \(5 − 2\), or \(b − a\). The result is 3, so we push 3 back onto the stack. The stack contents are \{4, 6, 3\}, with 3 accessible first. Now we read a “/” character, pull out the 3 and 6 values from the stack, divide 6/3 to get 2, and push the 2 onto the stack. The stack contents are \{4, 2\}, with the 2 accessible first. The next character is a 7, so we push it onto the stack, giving us a stack of \{4, 2, 7\}. The next character is “*,” so we pop the 7 and 2 from the stack, multiply them to get 14, and push the 14 onto the stack, giving us a stack of \{4, 14\}. Finally, our last character is “+,” so we pop the 14 and 4 from the stack, add them to get 18, and push the 18 onto the stack. We have reached the end of the input string, and the only value in the stack is 18, our answer.

With this knowledge of the Stack mechanics, find solutions to the following postfix expressions:

1. 814 * +
2. 814 + *
3. 7642/ − *
4. 12 − 3 + 4 − 5 + 6 − 7 +
5. 63 − 7 * 3 /

Part 2: Implementing a Simple Calculator (25 points)

Now that we have seen how to solve a postfix expression by hand, we can turn to the code. The goal in this section of the lab is to write a Java method that takes a postfix string as input and returns or prints a solution to that expression. You may assume that the postfix string that the method receives is valid – you don’t need to worry about error checking for not enough items on the
stack to evaluate an operator, or extra items remaining on the stack at the end of the expression. You may still run into these conditions if your stack code or infix-to-postfix converter is not correct, but proper or improper handling of these error cases will not increase or reduce your grade.

Partial credit will be given for achieving each of the following (in order of implementation complexity):

- Solve a postfix expression with two operands and a commutative operator (e.g. $ab +$ or $ab*$)
- Solve a postfix expression with all commutative operators (e.g. $ab + c * d+$)
- Solve a postfix expression with all the same, non-commutative operator (e.g. $ab - c -$)

Part 3: Additional Questions (10 points)

Please answer the following questions thoroughly:

1. Take an array of input items $\{1, 2, 3, 4, 5\}$ and push them from left-to-right into a stack. Then, pop each of the items off of the stack one-by-one and add them to a queue. Then, remove each of the items from the queue and push them back onto the stack. Then, pop each of the items off of the stack and back into the array. Are the items still in the same order? Why/why not?

2. In class, we implemented our Queue using an array as the underlying data storage. What if instead we used a SinglyLinkedList called data, as we did when we were implementing our Stack? Give methods for add(), remove(), and peek() for this list-based Queue implementation.

3. Is it possible to implement a Stack with a Queue as the underlying data storage, or to implement a Queue with a Stack as the underlying data storage? Either explain how we can, or explain why we cannot.

Part 4: Extra Credit (5 points)

Up to 10 points of extra credit will be awarded for supporting multiple-digit numerical values in the infix string. This presents two complications.

First, in our postfix notation, the infix statement “42 + 65” in will be converted to “4265+” in postfix notation, which is not differentiable from “4 + 265” and “426 + 5.” To solve this issue, I would recommend surrounding each number with a special character not used elsewhere in the mathematical expressions – say a “$” character. Then, the infix statement “42 + 65” will be converted into “$42$$65$+,” and the 42 and 65 values are clearly delimited by the dollar signs.

The second complication is reading the infix input character-by-character, and determining the value of the number represented – reading “1” then “2” then “3” and knowing that it represents the number 123. There are a number of ways to handle this issue, so I will leave it to you to puzzle over.
Submission Details

For this assignment, your submission (to both your BitBucket repository and by hardcopy) should include the following:

1. (Print and Upload) Your solutions to the postfix expressions in Part One.
2. (Upload) Your source code for your calculator in Part Two.
3. (Print and Upload) Sample output showing how your calculator works on several different input strings.
4. (Print and Upload) The answers to the questions from Part Three
5. (Print and Upload) An Assignment Information Sheet for your code submission

Before you turn in this assignment, you also must ensure that the course instructor has read access to your BitBucket repository that is named according to the convention cs112s2016-<your user name>.