Lab 7 - Using Stacks to Create a Calculator
Due (via Bitbucket and hard copy) Thursday, 6 April 2017
100 points

Lab Goals

• Practice working with the Java Stack class

• Learn the difference between infix and postfix notation

• Implement a calculator with postfix notation

• Answer a few questions to test your knowledge of course material

Assignment Details

Stacks are a data structure which can store any number of data types, with two restrictions: (1) only the top item in the stack is accessible, and (2) the first data item entered into the stack must be the last data item removed from the stack (also thought of as the last data item entered into the stack must be the first data item removed from the stack).

In this lab, we will implement a method to turn an input string representing a calculation from infix notation into postfix notation. From there, we will extend this postfix notation, 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. By the end of this lab, 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.”

Part 1: Definitions and Practice (15 points)

Infix notation is the representation of mathematical formulas that you are used to seeing. In infix notation, operators are placed between the operands. For example, the formula “a + b” is written in infix notation, with the “plus” operator located between the “a” and “b” operands.

In contrast, operators are placed after their associated operands in postfix notation. The equivalent formula to the above is “ab+” in postfix notation, with the “plus” operator located after the “a” and “b” operands.

Postfix notation is typically implemented in newer compilers, because parentheses can make the direct conversion from infix to machine language difficult. Adding postfix notation as a step midway simplifies compilation. Parentheses are not needed, and hence not used, in postfix notation.
Due Thu., 6 Apr. 2017 2 Lab 7

This is because postfix notation structures strings so that the operator computes two operands that precede it.

For example, consider the string “a * b + c” in infix notation. The “star” operator is applied to the “a” and “b” operands, so it should follow those operands. Hence, we can start our postfix notation string with “ab*.” After the system computes this string, it is effectively a single operand (think “d = a * b”), so that “d” operand added to operand “c” is handled next. Hence, the final postfix string will be “ab*c+.”

This gets more complicated when we start to consider order of operations. The infix string “a + b * c” is mathematically evaluated in reverse, with “b * c” evaluated before “a” is added to get the final result. To handle this case, we can introduce the idea of stacks to the infix-to-postfix conversion. As we read the infix string from left to right, we start by leaving the “a” alone, moving it directly to the postfix string. When reading the “+,” we will push it to a stack. We will read the “b” character and add it to the postfix string. Now we read the “*” character. Because the “*” operation supercedes the “+” operation in the order of operations, we will push the “*” to the stack. Finally, we read the “c” character, and add it to the postfix string. Since we have reached the end of the infix string, we must now pop the operators off the the stack. First the “*” is popped, followed by the “+” operator. This yields a final postfix string of “abc*+,” in which the “*” operator is applied to the “b” and “c” operands, then the “+” operator is applied to that result and the “a” operand.

We can add an additional layer of complexity when adding parentheses to the infix string input, the whole reason for making this conversion. Given the input string “(a + b) * c,” we want the “a + b” component to be evaluated before the “*c” is applied. Hence, our final postfix string should be “ab + c*.” We can handle this by pushing the opening parenthesis character to the same stack as the operators. Reading across the infix string, we read “(” and push it onto the stack. We read the “a” and add it to the postfix string, read the “+” and push it to the stack, and then read the “b” and add it to the postfix string. Now, when we encounter the closing parenthesis character, we will pop everything from the stack until the first “)” is found on the stack, which adds the “+” character to the postfix string. The closing parenthesis character is discarded, and the rest of the postfix string generation continues as before.

The final case that we should consider is if our input string includes two or more consecutive operators on the same level of order of operations precedence. For example, consider the infix string “a + b - c - d.” Naturally, these operators are evaluated in order – the “+” is applied to the “a” and “b,” the first “-” is applied to the “a + b” result and “c,” and the second “-” is applied to the “a + b - c” result and “d.” Hence, our postfix string should be “ab+c-d-.” We can generate this string with a comparison step similar to the comparison between operator levels in the “a + b * c” example. When we reach the first “-” operator, we already have “ab” in the postfix string and “+” on the top of the stack. We compare the “-” in the infix string to the “+” at the top of the stack. Because they are on the same level, we will pop the “+” from the stack and add it to the postfix string, and then push the “-” onto the stack. Similarly, when we reach the second “-” character, we already have “ab + c” in the postfix string and “-” on the top of the stack. We compare the “-” in the infix string to the “-” at the top of the stack. Because they are on the same level (and the same operator), we will pop the “-” from the stack and add it to the postfix string, then push the new “-” onto the stack.
We can summarize the actions that we should take in converting a string from infix to postfix notation with the table shown here:

<table>
<thead>
<tr>
<th>Next Infix Character</th>
<th>Top Character on the Stack</th>
<th>Empty Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Append to Postfix</td>
<td>Append to Postfix</td>
</tr>
<tr>
<td>)</td>
<td>Pop to Stack</td>
<td>ERROR – Infix string is invalid</td>
</tr>
<tr>
<td>+ or −</td>
<td>Push to Stack</td>
<td>Push to Stack</td>
</tr>
<tr>
<td>* or /</td>
<td>Push to Stack</td>
<td>Push to Stack</td>
</tr>
<tr>
<td>End of string</td>
<td>ERROR – Infix string is invalid</td>
<td>Pop Everything to Postfix</td>
</tr>
<tr>
<td></td>
<td>Pop Everything to Postfix</td>
<td>Done</td>
</tr>
</tbody>
</table>

Figure 1: In this table, the column represents the top character on the stack, the row represents the next character seen in the infix string, and the table cells represent the operation that should be computed on the stack and/or on the postfix string.

Using these rules, please convert the following infix strings into postfix notation:

1. \( a * b \)
2. \( a/b + c \)
3. \( (a + b)/c + d \)
4. \( (a + b)/(c + d) \)
5. \( ((a - b) * c)/d \)

**Part 2: Implementing an Infix-to-Postfix Converter (25 points)**

Given the rules from the table at the top of the page and the expertise that you have gained from practicing in the previous section, you will now write a Java program to convert an input string from infix notation into postfix notation.
First, you should check the string for matching parentheses using the Parentheses application that we wrote in lecture last week. This will prevent both of the error conditions noted in the table above. If that test is passed, you can then pass the infix string into the converter function.

The converter function should take the infix string as an input parameter, and should return the postfix string at the end of the computation. You should use a Stack to store the operators as described in the previous section. You can assume that each of the operands will be a single character in length.

Partial credit will be given for (in order of implementation complexity):

- Converting strings with only one operator (e.g. “a + b” or “a * b”)
- Converting strings with only chains of operators all at the same level of precedence (e.g. “a + b + c + d” or “a * b / c * d”)
- Converting strings without parentheses (e.g. “a + b * c” or “a * b + c − d”)

If you are still stuck on Part 2 by 3/30, you may request a solution to the infix-to-postfix converter for a grade penalty.

Part 3: More 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 “4562−/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. \( 814 \ast + \)
2. \( 814 + \ast \)
3. \( 7642/ - \ast \)
4. \( 12 - 3 + 4 - 5 + 6 - 7+ \)
5. \( 63 - 7 \ast 3/ \)

Part 4: 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. \( ab + c \ast d+ \))
- Solve a postfix expression with all the same, non-commutative operator (e.g. \( ab - c- \))
Part 5: Additional Questions (20 points)

Please answer the following questions thoroughly:

1. Provide a table similar to the one on Slide #6 of Lecture #20 for the following sequence of Stack commands: isEmpty(), push(3), push(1), push(4), peek(), size(), pop(), push(1), push(5), peek(), size(), pop(), pop(), pop(), isEmpty()

2. Suppose an initially empty stack $S$ has performed a total of 35 push operations, 12 peek operations, and 10 pop operations, 3 of which returned null to indicate an empty stack. What is the current size of $S$?

3. Write a short method to copy the contents of a given stack $S$ into a new stack $T$. When complete, the contents in $T$ should have the same ordering that they had in $S$. You can either give pseudocode or Java code for this question.

4. 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?

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

6. 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 6: Extra Credit (10 points)

1. Up to 5 points of extra credit will be awarded for adding exponentiation support with the “^" character into your infix-to-postfix converter and to your final calculator method. Remember that, in the order of operations, exponentiation has precedence over multiplication, division, addition, and subtraction, but not over parentheses. You will need to determine how the system should behave when a “^" character is reached in the infix string – what should get added to the stack, what should get removed from the stack, and so on.

2. Up to 5 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 anywhere 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, please submit the following to your cs112s2017-<your user name> repository (and ensure that the instructor has access to your repository):

1. Your answers to the infix-to-postfix conversions in Part One.
2. Your source code for your converted in Part Two.
3. Sample output showing how your converter works on several different input strings.
4. Your solutions to the postfix expressions in Part Three.
5. Your source code for your calculator in Part Four.
6. Sample output showing how your calculator works on several different input strings.
7. The answers to the questions from Part Five