ASSIGNMENT 1: Implementing a Calculator

Due Date: Sunday February 6th, 11:59 pm.

ASSIGNMENT 1(A): STACK Using Growable Arrays

Problem Statement: Your first task is to implement a generic Stack class named **MyStack**. As is to be expected, MyStack must implement the Stack ADT we provided in class. The exact interface will be as follows:

```
public interface StackInterface {

    public void push(Object o);

    public Object pop() throws EmptyStackException;

     public Object top() throws EmptyStackException;

     public boolean isEmpty();

    public String toString();

}
```

Your implementation must be done in terms of an array that grows by doubling as needed. Your initial array must have a length of one slot only!

Your implementation must support all stack operations except insertion in (worst-case) constant time; Push operation can take longer every now and then (when you need to grow the array), but overall all insertion operations must be constant amortized time as discussed in lecture.

You should provide a toString method in addition to the methods required by the Stack interface. A stack with 1,2,3 from top to bottom order should return "[1, 2, 3]" while an empty Stack should print as "[]". Note that this method will take time linear in the number of elements on the stack.

Ensure that the version of your code you hand in does not produce any extraneous debugging output.

ASSIGNMENT 1(B): Postfix Calculator

As explained in the lecture, in a postfix notation, the operator comes after the operands. For example,

(i) 7 9 + : Answer is 16

(ii) 4 6 + 9 * 5 7 * + 3 * : We scan this from left to right and whenever we see an operator, we apply it on the previous two operands. Hare, we first replace 4 6 + by 10 to get 10 9 * 5 7 * + 3 * Then we replace 10 9 * by 90 to get 90 5 7 * + 3 * After this we get 90 35 + 3 *, 105 3 * and finally 315.

Implement a class **Calculator** which has the following method

public int evaluatePost-Fix(**String** s);

This method takes as input a postfix expression in the form of a String. The method should analyze the String, and use the implementation of the Stack interface to print the value of the expression. You can assume that the only allowed operators are + , - , * and the operands are positive integers. All operators/operands will be separated by one or more whitespaces in the input string. You should throw an InvalidPostfixException in case the string is not a valid expression (refer to JAVA exception handling).

ASSIGNMENT 1(C): Implementing a Calculator

We would now like to have a calculator which evaluates expressions written in the notation we normally use, which we call the **standard notation**.

Implement the following method in the Calculator class:

public String convertExpression (**String** s);

This method should take as input a String representing an arithmetic expression written in the standard notation, and return a String representing the expression in postfix notation with operators/operands separated by a whitespace. Note that the input string may contain 0 or more white spaces between operands/operators.

Again you can assume that the expression in the standard notation contains + , - , * , (,) and positive integers only. You should throw an InvalidExprException in case the input string is not a valid arithmetic expression.

Note In the standard notation, multiplication has precedence over addition or subtraction. For example, the value of 6+3*4 is 18.

Further, for operators with equal precedence, you should evaluate them in the left to right order, e.g., 5+6+8 should be evaluated in the order (5+6)+8 and NOT as 5+(6+8).

Note that the arithmetic expression: (-3) + 4 is invalid since only positive integers are allowed.

Following are some examples of valid expressions:

2 + (((3)))
(5+5-5)
2+(2-(3-0))

For the above expressions, valid postfix expressions, respectively, are:

2 3 +
5 5 + 5 -
2 2 3 0 - - +

Evaluation Criteria

The assignment is worth 12 points. Your code will be autograded at the demo time against a series of tests. 8 points will be provided for correct output and 4 points will be provided for your explanations of your code and your answering demo questions appropriately. Your code should be as efficient as possible (primarily think about efficiency in terms of the time and memory complexity, and removing any obvious inefficiencies/redundant operations resulting in very slow implementations). Marks will be deducted for inefficient code/implementations.

What is allowed? What is not?

1. This is an individual assignment.
2. Your code must be your own. You can browse online resources for any general ideas/concepts, but you are supposed to search for/look at specific code meant to solve these or related problems.
3. You should develop your algorithm using your own efforts. You should not Google search for direct solutions to this assignment. However, you are welcome to Google search for generic Java-related syntax.
4. You must not discuss this assignment with anyone outside the class. **Make sure you mention the names in your write-up in case you discuss with anyone from within the class.** Please refer to the plagiarism related guidelines covered in the first lecture and follow them carefully. In case of any doubts, you are free to contact the TAs or the instructors.
5. You are not allowed to use built-in (or anyone else's) implementations of stacks, queues, vectors, growable arrays and/or other similar data structures - it is ok to use fixed size arrays as covered

in the class. A key aspect of the course is to have you learn how to implement these data structures.

6. Your submitted code will be automatically evaluated against another set of benchmark problems. You get a significant penalty if your output is not automatically parsable and does not follow input-guidelines.
7. We will run plagiarism detection software. Anyone found guilty will be awarded a suitable penalty as per IIT rules.

What to submit?

1. You should create individual class files for each part as mentioned above. Submit your code in a .zip file named in the format **<EntryNo>_G<GrpNo>.zip, e.g., e.g. 2018ME10000_G3.zip.** Make sure that when we run “unzip yourfile.zip”, there should be a directory <EntryNo>_G<GrpNo> created, and the working directory should contain at least 2 java files: MyStack.java and Calculator.java. You can create a separate class file Main.java for writing your starter code and testing your functions (this may not be evaluated). In addition to your code “writeup.txt” should be produced in the working directory. Thus the directory structure should be as follows after unzipping:

   <EntryNo>_G<GrpNo>

   - MyStack.java
   - Calculator.java
   - Main.java (Optional)
   - any other files needed for implementation
   - writeup.txt

2. You will be penalized for any submissions that do not conform to this requirement.

3. The writeup.txt should have a line that lists names of all students you discussed/collaborated with (see guidelines on collaboration vs. cheating on the course home page). If you never discussed the assignment with anyone say None. After this line, you are welcome to write something about your code, though this is not necessary.