Comp-2540: Lab Assignment 3: Stack and Dynamic Array

Your Marks (For TA Use Only)

3.1.1	3.1.2	3.1.3	3.2.1	3.2.2	Bonus	Total	Marked by

1 Due Date and Submission

The due time is your lab section during the week starting November 2. To be submitted 15 minutes before the end of your lab section. You can also submit in the labs one week earlier or during office hours. Make sure that you add your signature below to reaffirm that you followed Senate Bylaws 31 (click here for the document). Some PDF readers may not support digital signature well. We recommend you to use Acrobat Reader that is free for downloading.

Fill and Sign The Form

I, your name VEDANG PATEL **, verify that the submitted work is my own work.**

VEDANG PATEL Digitally signed by VEDANG PATEL Date: 2021.11.03 13:11:09 -04'00' Date NOV-03, Student Number 110035768 EMail ID patel3

2 Objective

The aim of this assignment is to obtain hands on experience in implementing the stack ADT. To understand an efficient implementation of stack, we use dynamic array to save space. Different from previous lab assignments that focused on time efficiency, this assignment also consider space efficiency, and the balance between the two. From this example, we learn amortized time complexity.

3 Tasks

3.1 Implement the stack ADT using array (4 marks)

3.1.1 Implement the `pop()` operation in the stack (1 mark)

Implement a stack class named `Stack2540Array` using array. The starter code is as follows. The instance variables and most operations are provided. You need to implement the `pop` operation. Make sure that your program checks whether the stack is empty in the pop operation. The implementation can be found in the book and in our lecture slides.

```
import java.io.*;
import java.util.*;

public class Stack2540Array {
        int CAPACITY = 128;
        int top;
        String[] stack;

        public Stack2540Array() {
                stack = new String[CAPACITY];
                top = -1;
        }

        public int size() {             return top + 1; }
        public boolean isEmpty() {              return (top == -1); }
```

```
        public String top() {
                if (top == -1)
                        return null;
                return stack[top];
        }

        public void push(String element) {
                top++;
                stack[top] = element;
        }
```

```
public String pop() {
    if (top == -1)
       return null;
    String str = stack[top];
    top--;
    return str;
  }
```

Figure 1: Write the `pop()` operation in the box. Also submit the code in the submission site.

3.1.2 Test Your Stack with the Parentheses Matching Problem (2 marks)

You will test your `Stack2540Array` on an application, i.e., parentheses matching. It takes a string that contains parentheses that are often found in arithmetic expressions, and checks whether the brackets are correct (balanced) or incorrect (unbalanced). Example of valid strings are:

```
final static String[] valid = {
        "( ) ( ( ) ) {( [ ( )  ] ) } ",
        "(3) (3 + (4 - 5) ) {( [ ( )  ] ) } ",
        "((()(())){([()])}))",
        "[(5+x)-(y+z)]"
};
```

Examples of invalid strings are:

```
final static String[] invalid={
        ")(())){([()])}",
        "({[])}",
        "("
};
```

You will write a method named `isMatched` like below. It will return true for valid strings, and false for invalid strings. You will use the stack named `Stack2540Array` that is implemented in the first step. The algorithm itself can be the same as in our lecture slides. Your program does not need to check the correctness of the arithmetic operators/operands. What you need to do is to check for balanced brackets.

```
        public static boolean isMatched(String expression) {
                final String opening = "({[";
                final String closing = ")}]";
                Stack2540Array buffer = new Stack2540Array();
                ... ...
                return  ...;
        }
```

3.1.3 Test Memory Usage on the Text File Reversing Problem (1 mark)

Test the performance of your Stack implementation on file reversing task. The code below is to reverse a text file using the stack. It has three lines missing, and you need to complete it. Then you submit it on our marking site here. You need to change the stack capacity according to input data size. On average each line has about 11 words. Our site prints out the total computer memory usage by your program when it runs on 5 data sets of different sizes.

The total memory usage from the submission site is 571

```
static String[] reverse(String filename) throws Exception{
                Scanner scanner = new Scanner(new File(filename)).useDelimiter("[^a-zA-Z]+");
                Stack2540Array stack = new Stack2540Array();
                while (scanner.hasNext())
                        stack.push(scanner.next().toLowerCase());
                String[] rev = new String[stack.size()];
                ...
                ...
                ...
                return rev;
        }
```

```
public static boolean isMatched(String expression) {
      final String opening = "({[";
      final String closing = ")}]";
      Stack2540Array buffer = new Stack2540Array();
      String temp;
      for(int i = 0; i < expression.length(); i++) {
         temp = expression.substring(i, i + 1);
         if(opening.indexOf(temp) != -1) {
            buffer.push(temp);
         }
         else if(closing.indexOf(temp) != -1) {
            if(buffer.isEmpty()) {
```

Figure 2: Write the `isMatch` method in the box. Also test your code here

3.2 Dynamic Array (4 marks)

To save computer memory, we need to use dynamic array. In the first version of our stack implementation, the instance variable has a fixed capacity, which is 128. There will be an error when the stack is bigger than 128. For bigger data, you increased the array size. But that very long array could be wasted for smaller data.

3.2.1 Array Resizing (2 marks)

In order to solve this dilemma, we start with a small array, and increase the capacity only when it is needed. You will modify the `push` operation by resizing the array when it overflows. And you also need to implement the `resize(...)` method. The details of the code are in the slides. Submit your dynamic stack in the submission site here. Your can use the doubling strategy in resizing.

```
public void resize() {
      CAPACITY = 2 * CAPACITY;
      String[] str = new String[CAPACITY];
      for (int i = 0; i < stack.length; i++)
         str[i] = stack[i];
      stack = str;
   }
```

Figure 3: Wtite the `resize(...)` operation in the box. Also test your code on our submission site here

To make sure that your resizing is correct, you will test your `Stack2540ArrayDynamic` on the reversing task in our site. The reverse method is called `reverseDynamic`, and it will use the dynamic version of the stack implementation. The total space used by dynamic array is now 570.26

3.2.2 Comparison of doubling strategy and constant incremental approach (2 marks)

There are two approaches to array resizing, i.e., doubling and increasing by a constant. Run `reverseDynamic` program on two versions of Stack2540ArrayDynamic. In one version, array is resized by doubling (CAPACIT*2). In another version, it is resized by adding a constant 128 (CAPACITY+128), and plot the running time against the data size in Figure 4. Data for this plot should be obtained from your local machine.

We shall see that two strategies causes very different performance. In theory, suppose that the data size (the number of words in the text file) is n. The time complexity of `reverseDynamic` program when doubling strategy is used is O(n) in big O notation. The time complexity of `reverseDynamic` when constant addition is used is O(n^2). A brief explanation of complexity of the doubling strategy can be explained by the total copying cost, which is calculated by the following formula:

2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 ... + 2^n = 2^(n+1) - 1

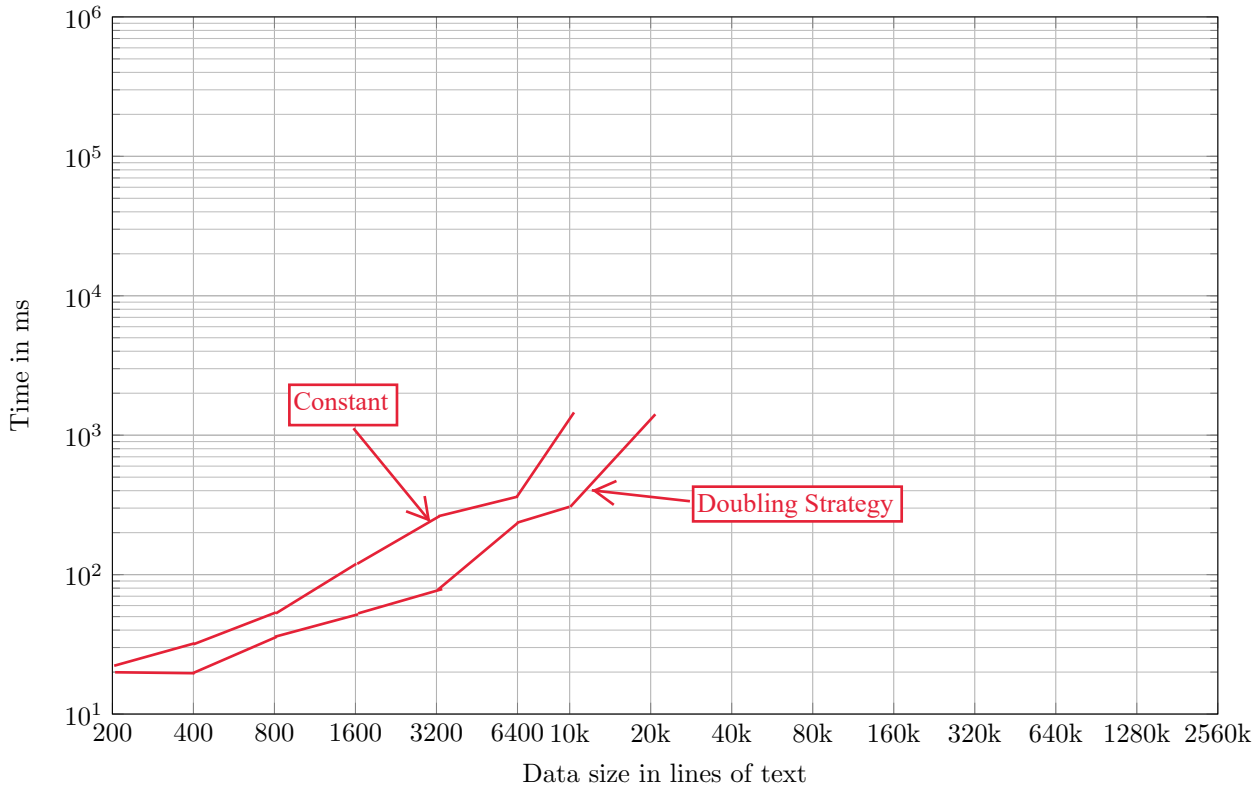

Figure 4: Plot the run time for doubling strategy and constant (128) incremental strategy in array resizing when running `reverseDynamic` on different data sizes. You can skip the data points that take too much time. You can run either on our site or on your machine, but you need to run at least once successfully on our site to demonstrate your program is correct.

4 Bonus (1 mark for each of the top 6 students)

When you run the dynamic array, we print out both memory usage and running time, and the summation between the two. You will receive one bonus mark if you are among the top 6 programs (most efficient in both space and time). A simple improvement is to change the initial stack size, and improve text read similar to Lab A2.