**CSE 212 – Programming with Data Structures**

**W03 Prove – Response Document**

|  |  |
| --- | --- |
| **Name:** | Jorge Alberto Chavez Ponce |
| **Date:** | 09/30/2023 |
| **Teacher:** | Ephraim Kunz |

*It is a violation of BYU-Idaho Honor Code to post or share this document with others or to post it online. Storage into a personal and private repository (e.g. private GitHub repository, unshared Google Drive folder) is acceptable.*

**Question 1: From Part 1, describe what the Mystery Stack 1 code does and how the use of a stack helps in the implementation.**

The Mystery Stack 1 code reverses the entered text. It uses a stack data structure to achieve this in reverse.

First it starts an empty list called stack. Second it will loop through each character in the input text and add each character to the stack. This effectively pushes each character onto the stack, one by one.

After the loop, an empty string is created, followed by a while loop that continues if the stack length is greater than 0. Inside the while loop, pop (remove and return) the top element of the stack and return it. adds to the result string. This effectively reverses the order of the characters. This causes the order of the letters in the word to be reversed. When the while loop ends, the result string contains the inverted version of the input text.

**Question 2: From Part 1, what are the three outputs from the Mystery Stack 1 code for the following three different inputs?**

* **Racecar**

Output: “**racecaR”** (This word is a palindrome)

* **Stressed**

Output: “**dessertS”**

* **a nut for a jar of tuna**

Output: " **anut fo raj a rof tun a**”

**Question 3: From Part 2, describe what the Mystery Stack 2 code does and how the use of a stack helps in the implementation.**

**Question 4: From Part 2, answer the following regarding what the Mystery Stack 2 code does:**

* **What will the result be if the input parameter is: 5 3 7 + \***

The code interprets this postfix expression as: (3+7) \* 5

The stack is:

[5.0]

[5.0,3.0]

[5.0,3.0,7.0] => operator + => op2 = 7 y op1 = 3 res = [10]

[5.0,10.0] => operator \* => op2 = 5 y op1 = 10 res = [50]

[50.0]

So, the result is: **50.0**

* **What will the result be if the input parameter is: 6 2 + 5 3 - /**

The code interprets this postfix expression as: (6+2)/(5-3)

The stack is:

[6.0]

[6.0,2.0] => operator + => op2 = 2 y op1 = 6 res = [8]

[8.0]

[8.0,5.0]

[8.0,5.0,3.0] => operator - => op2 = 3 y op1 = 5 res = [2]

[8.0, 2.0] => operator / => op2 = 2 y op1 = 8 res = [4]

[4.0]

So, the result is: **4.0**

* **What input parameter would result in the display of “Invalid Case 1!”**

If an operator (+, -, \*, /) is encountered without at least 2 operands on the stack, it displays "Invalid Case 1!" and returns “N**one”**

For example, the next parameter: **5 +**

* **What input parameter would result in the display of “Invalid Case 2!”**

If a division operation is encountered with the second operand (op2) being 0, it displays "Invalid Case 2!" and returns “N**one”**

For example, the next parameter: **5 0 /**

* **What input parameter would result in the display of “Invalid Case 3!”**

If a non-numeric and non-operator item is encountered, it displays "Invalid Case 3!" and returns “N**one”**

For example, the next parameter: **5 a b c +**

* **What input parameter would result in the display of “Invalid Case 4!”**

If there is more than one value left on the stack after processing all items, it displays "Invalid Case 4!" and returns “N**one”**

For example, the next parameter: **5 3 + 5**