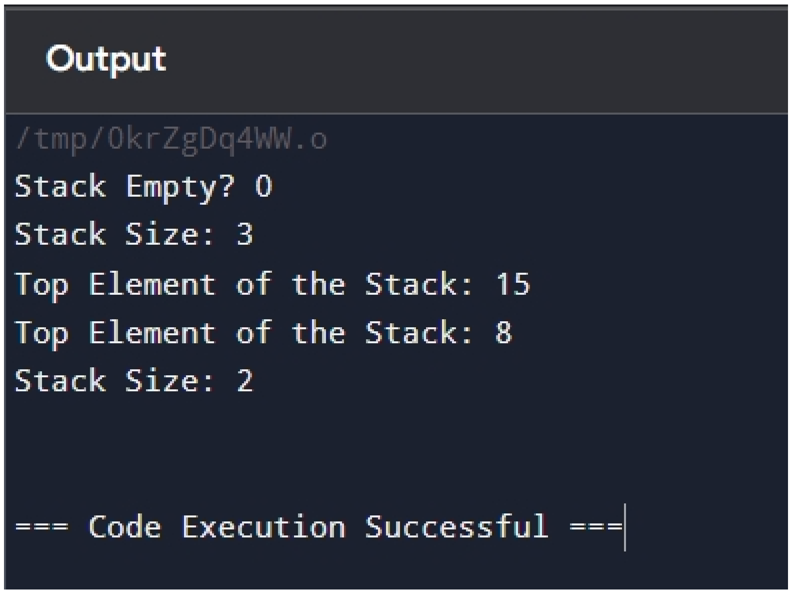
| Activity No. 4 | |
|---|---|
| **Hands-on Activity 4.1 Stacks** | |
| **Course Code:** CPE010 | **Program:** Computer Engineering |
| **Course Title:** Data Structures and Algorithms | **Date Performed: 04/10/24** |
| **Section: CpE21S4** | **Date Submitted: 04/10/24** |
| **Name(s): Alexzander J. Reyes** | **Instructor: Maria Rizette Sayo** |
| **6. Output** | |



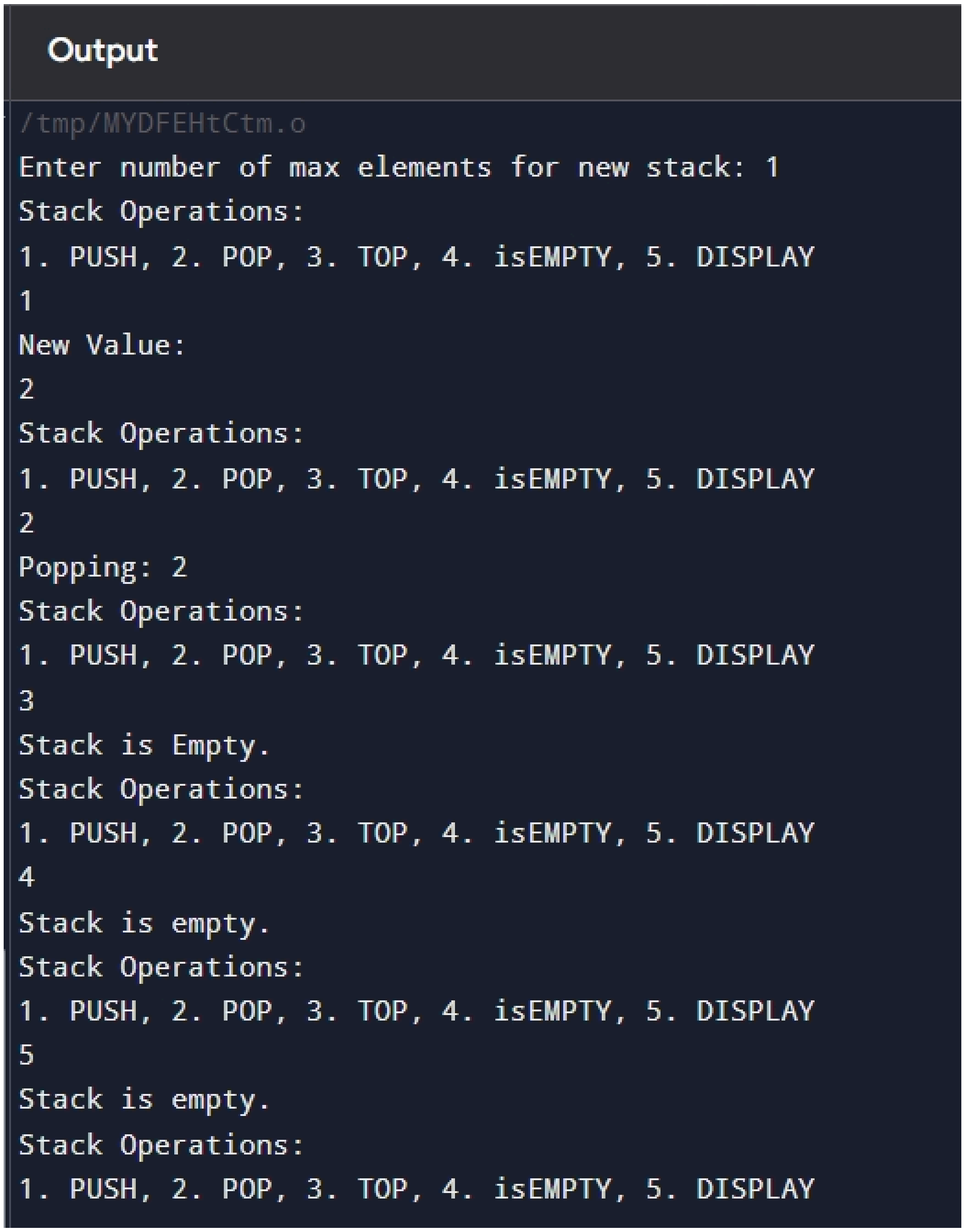
```
/tmp/0krZgDq4WW.o
Stack Empty? 0
Stack Size: 3
Top Element of the Stack: 15
Top Element of the Stack: 8
Stack Size: 2


=== Code Execution Successful ===
```

**Table 4-1. Output of ILO A**

**Observation**

The code uses the Standard Template Library (STL) stack container to demonstrate basic stack operations like push(), pop(), empty(), size(), and top(). It manipulates the stack with integer values.

**Remarks:**
The stack operates in a LIFO (Last-In, First-Out) manner, as expected.
push() successfully adds elements to the top of the stack.
empty(), size(), and top() work as expected and return correct values.
pop() removes the top element correctly.



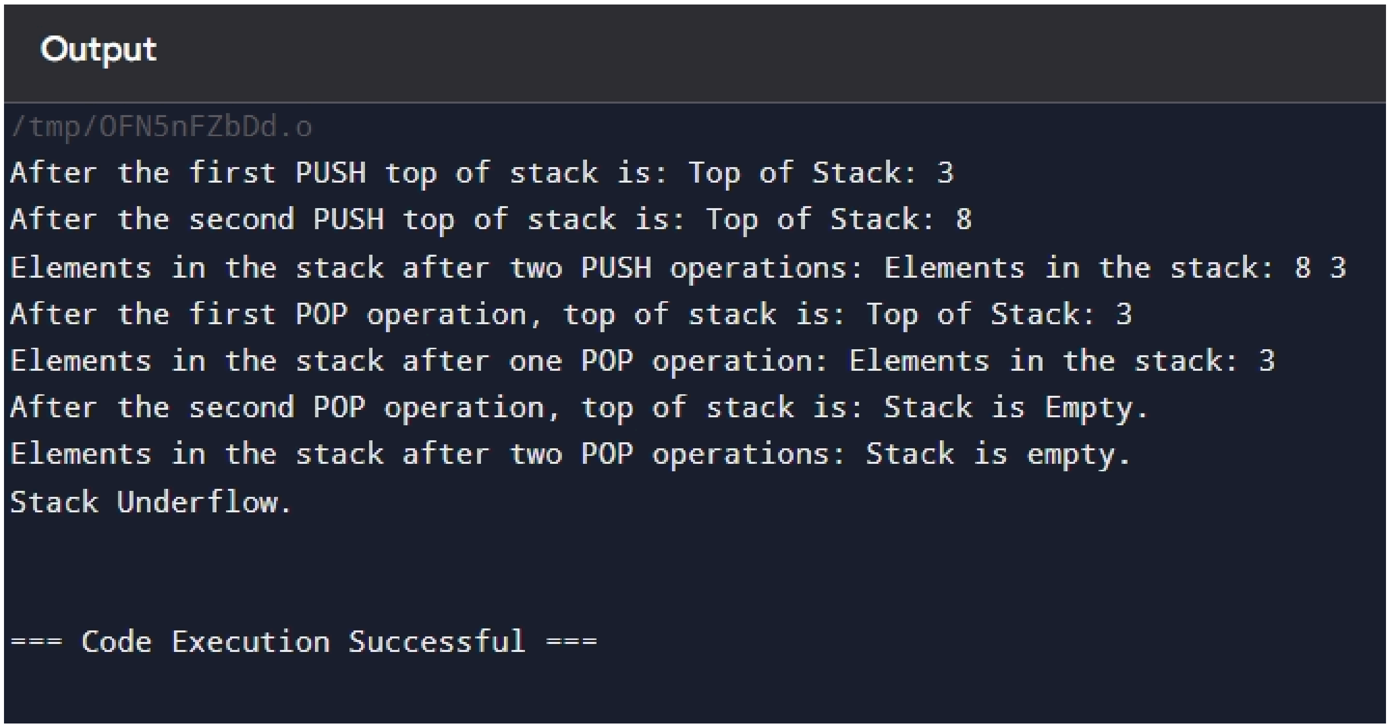
```
Output
/tmp/MYDFEHtCtm.o
Enter number of max elements for new stack: 1
Stack Operations:
1. PUSH, 2. POP, 3. TOP, 4. isEMPTY, 5. DISPLAY
1
New Value:
2
Stack Operations:
1. PUSH, 2. POP, 3. TOP, 4. isEMPTY, 5. DISPLAY
2
Popping: 2
Stack Operations:
1. PUSH, 2. POP, 3. TOP, 4. isEMPTY, 5. DISPLAY
3
Stack is Empty.
Stack Operations:
1. PUSH, 2. POP, 3. TOP, 4. isEMPTY, 5. DISPLAY
4
Stack is empty.
Stack Operations:
1. PUSH, 2. POP, 3. TOP, 4. isEMPTY, 5. DISPLAY
5
Stack is empty.
Stack Operations:
1. PUSH, 2. POP, 3. TOP, 4. isEMPTY, 5. DISPLAY
```

**Table 4-2. Output of ILO B.1**

**Observation:**
The modified code provides a comprehensive implementation of a stack using an array, allowing the user to perform fundamental stack operations like push(), pop(), Top(), isEmpty(), and the newly added display() function. The code uses an infinite loop to continuously prompt the user for stack operations until they choose to terminate it.

**Remarks:**
Array-Based Stack:
The stack is implemented using a fixed-size array with a maximum capacity of 100 elements, though the user can define a smaller capacity (i) at runtime. This makes the stack size flexible within the array's limit.
Push Operation:
The push() function checks for stack overflow by ensuring the current top index is within bounds. If the stack is full, it displays a "Stack Overflow" message. Otherwise, it adds the new element and increments the top.
Pop Operation:

The pop() function checks for stack underflow by using the isEmpty() function. If the stack is empty, it displays a "Stack Underflow" message. If not, it removes the top element and decrements the top.
Top Operation:
The Top() function checks if the stack is empty before retrieving the top element. It avoids accessing invalid indices by ensuring that the stack has at least one element.
Display Function:
The newly added display() function iterates through the stack and prints all elements from the bottom (index 0) to the top. It provides a clear view of the entire stack, helping users better understand the stack's current state.
isEmpty Operation:
The isEmpty() function returns a boolean value, with true indicating that the stack has no elements and false indicating the stack has at least one element.
User-Friendly Loop:
The loop continuously prompts the user for a stack operation, making it user-friendly and interactive. It allows the user to test different stack operations in sequence.
Error Handling:
The code includes effective error handling for stack overflow and stack underflow scenarios, preventing the program from crashing when invalid operations are performed.



```
Output
/tmp/OFN5nFZbDd.o
After the first PUSH top of stack is: Top of Stack: 3
After the second PUSH top of stack is: Top of Stack: 8
Elements in the stack after two PUSH operations: Elements in the stack: 8 3
After the first POP operation, top of stack is: Top of Stack: 3
Elements in the stack after one POP operation: Elements in the stack: 3
After the second POP operation, top of stack is: Stack is Empty.
Elements in the stack after two POP operations: Stack is empty.
Stack Underflow.


=== Code Execution Successful ===
```

**Table 4-3. Output of ILO B.2**

**Observation:**
This code implements a stack using a singly linked list. Each element in the stack is represented by a Node that contains data and a pointer to the next node. The stack operations are performed using dynamic memory allocation (new and delete operators). It includes the basic stack operations: push(), pop(), and Top(), and tests them within the main() function.

**Remarks:**
Class Node:

The Node class contains two members: data (the value stored in the node) and next (a pointer to the next node in the list).
Each Node represents an element in the stack, and the stack itself is represented by the linked list structure.
Push Operation:
The push() function adds a new element to the top of the stack.
A new Node is created dynamically using new.
The new node's next pointer is set to the current head (top of the stack), and head is updated to point to the new node.
If the stack is empty (head == NULL), both head and tail point to the newly created node.
Error: The push() function incorrectly sets newNode->next = head; twice. The first one (before the if-else block) is redundant and should be removed.
Pop Operation:
The pop() function removes the top element from the stack.
It first checks if the stack is empty (head == NULL), in which case it prints a "Stack Underflow" message and returns -1.
If the stack is not empty, it removes the current top node (pointed to by head), retrieves its value, updates head to point to the next node, and deletes the removed node from memory.
Issue: In the condition where head == NULL, both head and tail are unnecessarily reset to NULL because they are already NULL at this point.
Top Operation:
The Top() function displays the value at the top of the stack (the value in the node pointed to by head).
If the stack is empty, it prints "Stack is Empty."
Main Function:
The main() function tests the stack operations.
It pushes two elements (1 and 5), prints the top element after each push, then performs two pop operations and prints the top element after each pop.
After the second pop, the stack becomes empty, and attempting to pop again prints "Stack Underflow."

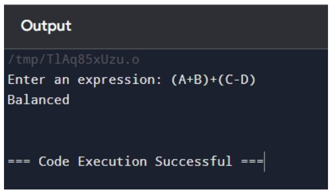
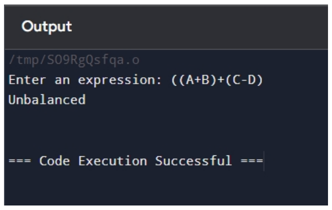
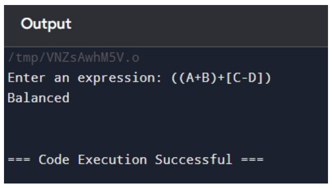
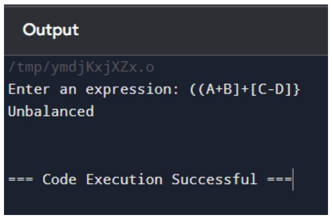
**7. Supplementary Activity**

**Array-Based Stack**

```
#include <iostream>
using namespace std;

class ArrayStack {
    int top;
    int maxSize;
    char* stackArray;

public:
    ArrayStack(int size) {
        top = -1;
        maxSize = size;
        stackArray = new char[maxSize];
    }

    ~ArrayStack() {
        delete[] stackArray; // Clean up dynamically allocated memory
    }
    int isEmpty() {
        return top == -1;
    }

    int isFull() {
        return top == maxSize - 1;
    }

    void push(char symbol) {
        if (!isFull()) {
            stackArray[++top] = symbol;
        } else {
            cout << "Stack Overflow" << endl;
        }
    }

    char pop() {
        if (!isEmpty()) {
            return stackArray[top--];
        } else {
            cout << "Stack Underflow" << endl;
            return '\0'; // Return a null character on error
        }
    }

    char peek() {
        if (!isEmpty()) {
            return stackArray[top];
        }
        return '\0'; // Return a null character on error
    }
};

// Function to check if symbols are balanced
int checkBalancedArrayStack(string expr) {
    ArrayStack s(expr.length());

    for (char& c : expr) {
        if (c == '(' || c == '{' || c == '[') {
            s.push(c);
        } else if (c == ')' || c == '}' || c == ']') {
            if (s.isEmpty()) return 0;
            char open = s.pop();
            if (!((c == ')' && open == '(') ||
                  (c == '}' && open == '{') ||
                  (c == ']' && open == '['))) {
                return 0;
            }
        }
    }
    if (!s.isEmpty()) return 0;
    return 1;
}

int main() {
    string expr;
    cout << "Enter an expression: ";
    cin >> expr;

    if (checkBalancedArrayStack(expr) == 1) {
        cout << "Balanced" << endl;
    } else {
        cout << "Unbalanced" << endl;
    }

    return 0;
}
```

**Linked List-Based Stack**

```
#include <iostream>
using namespace std;

struct Node {
    char data;
    Node* next;
};

class LinkedListStack {
    Node* top;

public:
    LinkedListStack() {
        top = nullptr;
    }

    ~LinkedListStack() {
        while (!isEmpty()) {
            pop(); // Clean up the stack on destruction
        }
    }

    int isEmpty() {
        return top == nullptr;
    }

    void push(char symbol) {
        Node* newNode = new Node();
        newNode->data = symbol;
        newNode->next = top;
        top = newNode;
    }
    char pop() {
        if (!isEmpty()) {
            Node* temp = top;
            char symbol = top->data;
            top = top->next;
            delete temp;
            return symbol;
        } else {
            cout << "Stack Underflow" << endl;
            return '\0';
        }
    }

    char peek() {
        if (!isEmpty()) {
            return top->data;
        }
        return '\0';
    }
};

// Function to check if symbols are balanced using LinkedList stack
int checkBalancedLinkedListStack(string expr) {
    LinkedListStack s;

    for (char& c : expr) {
        if (c == '(' || c == '{' || c == '[') {
            s.push(c);
        } else if (c == ')' || c == '}' || c == ']') {
            if (s.isEmpty()) return 0;
            char open = s.pop();
            if (!((c == ')' && open == '(') ||
                  (c == '}' && open == '{') ||
                  (c == ']' && open == '['))) {
                return 0;
            }
        }
    }
    if (!s.isEmpty()) return 0;
    return 1;
}

int main() {
    string expr;
    cout << "Enter an expression: ";
    cin >> expr;

    if (checkBalancedLinkedListStack(expr) == 1) {
        cout << "Balanced" << endl;
    } else {
        cout << "Unbalanced" << endl;
    }

    return 0;
}
```

| Expression | Valid? (Y/N) | Output (Console Screenshot) | Analysis |
|---|---|---|---|
| (A+B)+(C-D) | Y |  Output<br>/tmp/TlAq85xUzu.o<br>Enter an expression: (A+B)+(C-D)<br>Balanced<br>=== Code Execution Successful === | Both opening and closing parentheses match correctly. |
| ((A+B)+(C-D) | N |  Output<br>/tmp/SO9RgQsfqa.o<br>Enter an expression: ((A+B)+(C-D)<br>Unbalanced<br>=== Code Execution Successful === | Missing closing parenthesis for the second opening. |
| ((A+B)+[C-D]) | Y |  Output<br>/tmp/VNZsAwhM5V.o<br>Enter an expression: ((A+B)+[C-D])<br>Balanced<br>=== Code Execution Successful === | Parentheses and brackets match correctly. |
| ((A+B]+[C-D]} | N |  Output<br>/tmp/ymdjKxjXZx.o<br>Enter an expression: ((A+B]+[C-D]}<br>Unbalanced<br>=== Code Execution Successful === | Mismatch between parentheses and brackets. |

**Tools Analysis:**
• How do the different internal representations affect the implementation and usage of the stack?

**Array-Based Stack:** This implementation uses a fixed-size array. The primary benefit of adopting an array-based stack is that indexing and accessing elements are quick since arrays offer constant-time access. However, one constraint is the fixed size, which requires that the maximum number of items be determined. If the stack is full, no additional elements can be added, resulting in a potential overflow problem. This can be inefficient if the amount of data to be processed is unknown or changing. Furthermore, resizing an array (if dynamic resizing is used) incurs additional computational complexity because the entire stack must be moved to a larger array when the capacity is reached.

**Linked List-Based Stack:** This implementation employs a dynamic data structure in which each element (or node) carries both data and a pointer to the next node. The linked list-based stack's main advantage is its memory flexibility. The stack grows dynamically, so there is no need to set a limit size, and it only takes as much memory as is required. There is no chance of an overflow unless the machine runs out of memory. However, due to the expense of utilizing pointers in each node and the additional memory required for each pointer, this method is significantly less memory efficient than the array-based stack.

| 8. Conclusion |
|---|
| The activity gave students practical experience implementing stacks using linked lists and arrays. Although the array-based stack showed fast element access and retrieval, it was limited in size, which could have resulted in stack overflow. The linked list-based stack, on the other hand, provided dynamic memory allocation, enabling it to expand without bounds and at the expense of increased pointer memory utilization. Both implementations showed how to check for balanced parentheses in expressions and handled simple stack operations like pop, push, and peek with success. The exercise also demonstrated the trade-offs between the two approaches' performance and memory efficiency. |
| **9. Assessment Rubric** |
| |