**ACTIVITY NO. 4**

| **STACKS** | |
|---|---|
| **Course Code:** CPE010 | **Program:** Computer Engineering |
| **Course Title:** Data Structures and Algorithms | **Date Performed:** |
| **Section:** | **Date Submitted:** |
| **Name:** | **Instructor:** Engr. Roman M. Richard |

**1. Objective(s)**

- To implement the stack ADT in C++
- To create an implementation of stack with different internal representations

**2. Intended Learning Outcomes (ILOs)**

After this activity, the student should be able to:

a. Create a stack using the C++ STL
b. Develop C++ code that uses both arrays and linked lists to create a stack
c. Solve problems using an implementation of stack

**3. Discussion**

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

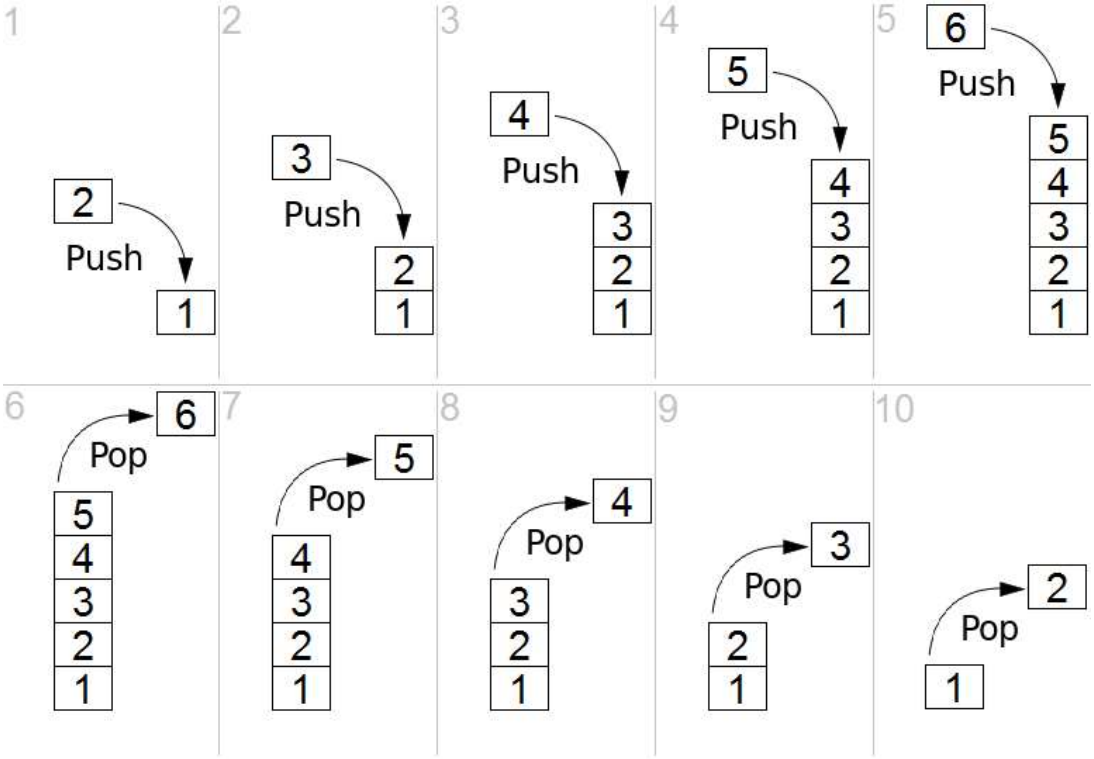


*Image Source: Wikipedia.org/wiki/Stack_(ADT)*

The following four basic operations can be performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

There are many real-life examples of a stack. Consider an example of plates stacked in the cupboard. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottom most position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out) / FILO(First In Last Out) order.

There are two ways to implement a stack:

- Using array
    - Pros: Easy to implement. Memory is saved as pointers are not involved.
    - Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.
- Using linked list
    - Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.
    - Cons: Requires extra memory due to involvement of pointers.

We will look at using a Linked list to implement a Stack. Most of what we need to know we have already covered in our discussion of Linked Lists - stacks only need a 'push' to build the stack. However, there are a couple of pieces that we need to add. In the "stack terminology" we have a 'pop' capability, which is just like the delete in our linked list. We also need to implement the 'peek' functionality - this simply returns the value that is sitting on the top of the stack but does not alter the stack in any way. Lastly, we will have to be able to determine if the stack is empty.



**4. Materials and Equipment**

Personal Computer with C++ IDE
Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

**5. Procedure**

**ILO A: Create a stack using the C++ STL**

Definition from the official CPP documentation for the stack STL.

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

**stacks** are implemented as container adaptors, which are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements. Elements are pushed/popped from the "back" of the specific container, which is known as the top of the stack.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall support the following operations:

```
empty
size
back
```

```
push_back
pop_back
```

The standard container classes vector, deque and list fulfill these requirements. By default, if no container class is specified for a particular stack class instantiation, the standard container deque is used. The member functions in the stack STL is shown in the figure below:

| | |
|---|---|
| (constructor) | Construct stack (public member function) |
| empty | Test whether container is empty (public member function) |
| size | Return size (public member function) |
| top | Access next element (public member function) |
| push | Insert element (public member function) |
| emplace | Construct and insert element (public member function) |
| pop | Remove top element (public member function) |
| swap | Swap contents (public member function) |

Member Functions

**Test the stack STL operations through the given code below. Provide screenshot per operation, your observations and any other remarks in table 4-1 provided in section 6.**

```
//Tests the push, empty, size, pop, and top methods of the stack library.

#include <iostream>
#include <stack>    // Calling Stack from the STL

using namespace std;

int main() {
    stack<int> newStack;

    newStack.push(3); //Adds 3 to the stack
    newStack.push(8);
    newStack.push(15);

    // returns a boolean response depending on if the stack is empty or not
    cout << "Stack Empty? " << newStack.empty() << endl;

    // returns the size of the stack itself
    cout << "Stack Size: " << newStack.size() << endl;

    // returns the topmost element of the stack
    cout << "Top Element of the Stack: " << newStack.top() << endl;

    // removes the topmost element of the stack
    newStack.pop();

    cout << "Top Element of the Stack: " << newStack.top() << endl;

    cout << "Stack Size: " << newStack.size() << endl;

    return 0;
}
```

**ILO B: Develop C++ code that uses both arrays and linked lists to create a stack**

In this section, we will have implementation of stack through an array and a linked list.

### B.1. Stacks using Arrays

```
#include<iostream>

const size_t maxCap= 100;
int stack[maxCap]; //stack with max of 100 elements
int top = -1, i, newData;

void push();
void pop();
void Top();
bool isEmpty();

int main(){
    int choice;
    std::cout << "Enter number of max elements for new stack: ";
    std::cin >> i;

    while(true){
        std::cout << "Stack Operations: " << std::endl;
        std::cout << "1. PUSH, 2. POP, 3. TOP, 4. isEMPTY" << std::endl;
        std::cin >> choice;

        switch(choice){
            case 1: push();
                    break;
            case 2: pop();
                    break;
            case 3: Top();
                    break;
            case 4: std::cout << isEmpty() << std::endl;
                    break;
            default: std::cout << "Invalid Choice." << std::endl;
                     break;
        }
    }

    return 0;

}

bool isEmpty(){
    if(top==-1) return true;
    return false;
}

void push(){
    //check if full -> if yes, return error
    if(top == i-1){
        std::cout << "Stack Overflow." << std::endl;
        return;
    }

    std::cout << "New Value: " << std::endl;
    std::cin >> newData;
    stack[++top] = newData;
```

```
}

void pop(){
    //check if empty -> if yes, return error
    if(isEmpty()){
        std::cout << "Stack Underflow." << std::endl;
        return;
    }

    //display the top value
    std::cout << "Popping: " << stack[top];
    //decrement top value from stack
    top--;
}

void Top(){
    if(isEmpty()) {
        std::cout << "Stack is Empty." << std::endl;
        return;
    }

    std::cout << "The element on the top of the stack is " << stack[top] <<
std::endl;
}
```

**Tasks:**

- Modify the code given above to include a function that will display all elements in the stack.
- Provide a description of each operation provided.
- Include your output in section 6.

### B.2. Stacks using Linked Lists

```
#include<iostream>

class Node{
public:
    int data;
    Node *next;
};

Node *head=NULL,*tail=NULL;

void push(int newData){
    Node *newNode = new Node;
    newNode->data = newData;
    newNode->next = head;

    if(head==NULL){
        head = tail = newNode;
    } else {
        newNode->next = head;
        head = newNode;
    }
}

int pop(){
    int tempVal;
    Node *temp;
```

```
        if(head == NULL){
            head = tail = NULL;
            std::cout << "Stack Underflow." << std::endl;
            return -1;
        } else {
            temp = head;
            tempVal = temp->data;
            head = head->next;
            delete(temp);
            return tempVal;
        }
    }

    void Top(){
        if(head==NULL){
            std::cout << "Stack is Empty." << std::endl;
            return;
        } else {
            std::cout << "Top of Stack: " << head->data << std::endl;
        }
    }

    int main(){

        push(1);
        std::cout<<"After the first PUSH top of stack is :";
        Top();
        push(5);
        std::cout<<"After the second PUSH top of stack is :";
        Top();
        pop();
        std::cout<<"After the first POP operation, top of stack is:";
        Top();
        pop();
        std::cout<<"After the second POP operation, top of stack :";
        Top();
        pop();

        return 0;
    }
```

**Tasks:**

- Modify the code given above to include a function that will display all elements in the stack.
- Provide a description of each operation provided.
- Include your output in section 6.

**6. Output**

Table 4-1. Output of ILO A

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

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

**7. Supplementary Activity**

**ILO C: Solve problems using an implementation of stack:**

The following problem definition and algorithm is provided for checking balancing of symbols. Create an implementation using stacks. Your output must include the following:

a. Stack using Arrays
b. Stack using Linked Lists
c. (Optional) Stack using C++ STL

**Problem Definition:**

Stacks can be used to check whether the given expression has balanced symbols. This algorithm is very useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [- then it is written to the stack. When a closing delimiter is encountered like ), }, or ]-the stack is popped. The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line.

**Steps:**

1. Create a Stack.
2. While(end of input is not reached) {
   a) If the character read is not a symbol to be balanced, ignore it.
   b) If the character is an opening symbol, push it onto the stack.
   c) If it is a closing symbol:
      i) Report an error if the stack is empty.
      ii) Otherwise, pop the stack.
   d) If the symbol popped is not the corresponding opening symbol, report an error.
3. At the end of input, if the stack is not empty: report an error.

**Self-Checking:**

For the following cases, complete the table using the code you created.

| Expression | Valid? (Y/N) | Output (Console Screenshot) | Analysis |
|---|---|---|---|
| (A+B)+(C-D) | | | |
| ((A+B)+(C-D) | | | |

| | | | |
|---|---|---|---|
| ((A+B)+[C-D]) | | | |
| ((A+B]+[C-D]} | | | |

**Tools Analysis:**

- How do the different internal representations affect the implementation and usage of the stack?

**8. Conclusion**

Provide the following:

- Summary of lessons learned
- Analysis of the procedure
- Analysis of the supplementary activity
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?

**9. Assessment Rubric**