CS-101 Programming Fundamentals

Lab Manual

Habib University

First printing, July 2018

Contents

1. Stacks

1.1 Objective

In this lab, we'll learn about a data structure, called Stack. We'll learn about its implementation, operations and uses.

1.2 Description

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).

Mainly the following three basic operations are 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.

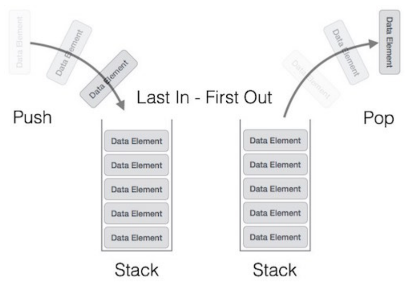

Figure 1.1: Representation of stack

1.2.1 Real-life example of stack

There are many real life examples of stack. Consider the simple example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

1.2.2 Time Complexities of operations on stack

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

1.2.3 Applications of stack

There are many possible applications of stack. Some are listed below:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver

1.2.4 Implementation

We will be implementing a linked list based stack. In a linked list based stack, we have nodes linked to each other. Each node stores a data and a link to the node next to it.

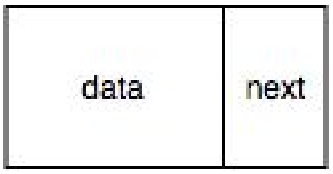

Figure 1.2: Representation of node

Implementing Node

To implement node, we will be making a struct to hold data and the link. In this implementation we are making a node to store integer type data. However, it can be implemented to store any data type or object.

```
struct Node
{
    int data; \\To store data
    Node* next; \\To store the address or link to the next node
};
```

Initializing the Stack

As a stack consists of many nodes, we should have many nodes inside our stack class, but we will have only one node, "head", which will act as a reference node and allows us to iterate through all the nodes till the end of the stack.

```
class Stack
{
    private:
        Node* head;

    public:
        Stack()
        {
            head = NULL;
        }
};
```

Our stack now consists of a Node pointer which will hold the address of the head (i.e the top element) of the stack. To initialize, it is set to **NULL**, as the stack is empty.

Figure 1.3: Stack's head pointer

Implementing Push() operation

Inside the class of Stack, we will declare and define a function "Push(int val)" in public.

```
void Push(int val)
{
    if (head == NULL)
    {
        head = new Node;
        head->data = val;
        head->next = NULL;
    }
    else
    {
        Node* temp = head;
        head = new Node;
        head->data = val;
        head->next = temp;
    }
}
```

■ **Example 1.1** Let's visualize an example Push() operation

```
Stack stack;
stack.Push(5);
```

Since the stack is empty and head is set to NULL. Body of 'if' will be executed.
It first creates a new node

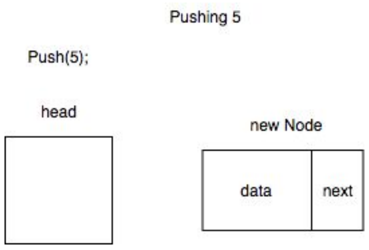

Figure 1.4: Creating new node for Push(5)

Then it sets head pointer to the new node.

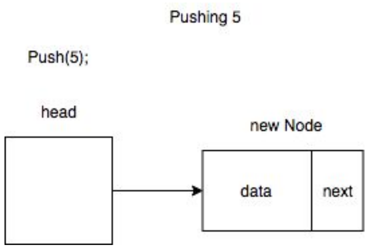

Figure 1.5: Setting head pointer

Then it sets the data and next pointer of new node.

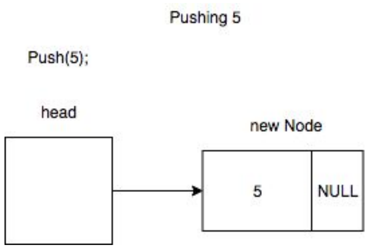

Figure 1.6: Setting data and next pointer

■

Since, head is not **NULL**, Push operation will now execute body of ’else’. Let’s see an example

■ Example 1.2

```
stack.Push(3);
```

It will first create a temporary node pointer, temp.

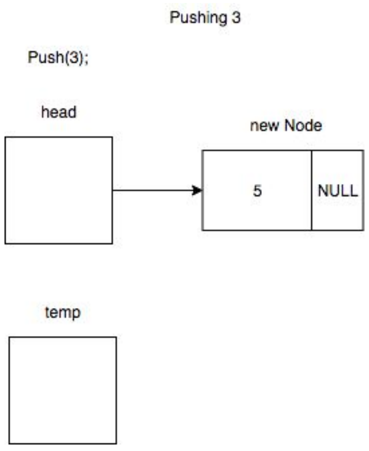

Figure 1.7: Creating temp pointer

Then it sets the temp to the node where head is pointing

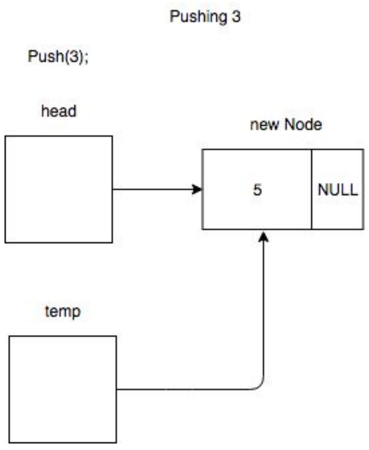

Figure 1.8: Setting temp node

It now creates a new node and sets head to the newly created node.

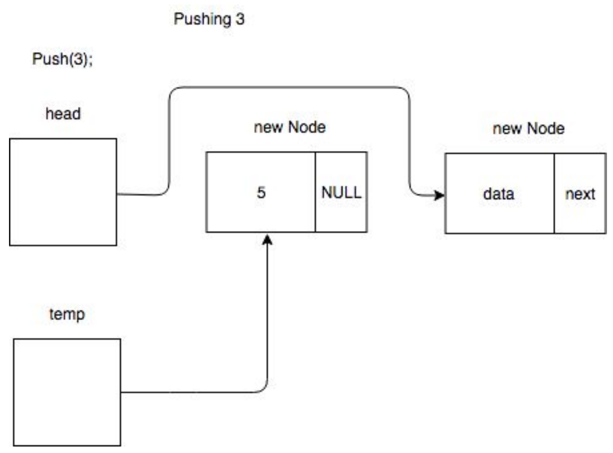

Figure 1.9: Setting head to the new node

It sets the newly created node's next pointer to the node being pointed by temp.

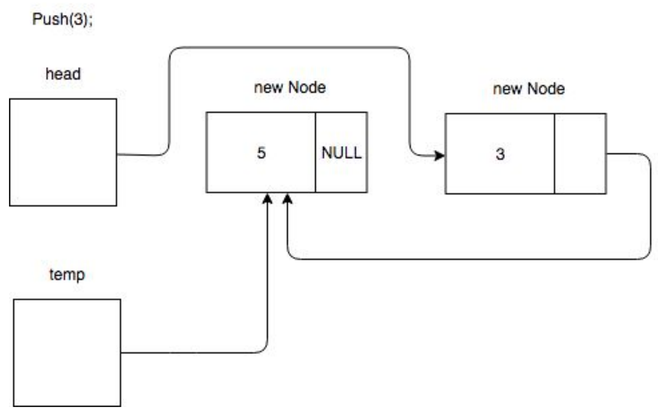

Figure 1.10: Setting next pointer

The temp gets destroyed since it was a local variable of the function and 3 is added succesfully.

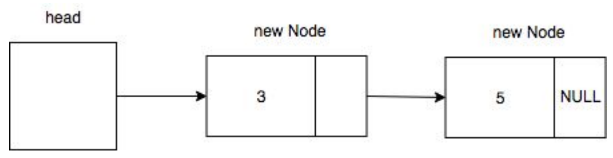

Figure 1.11: Succesful Push(3) operation

■

Implementing Pop() Operation

Inside the class of Stack, we will declare and define a function "Pop()" in public.

```
int Pop()
{
    int val = -1;
    if(head != NULL)
    {
        Node* temp = head;
        head = head->next;
        val = temp->data;
        delete temp;
        temp = NULL;
    }
    return val;
}
```

Let's visualize an example of Pop() operation

■ **Example 1.3**

```
stack.Pop();
```

This is the stack on which pop() will be called.

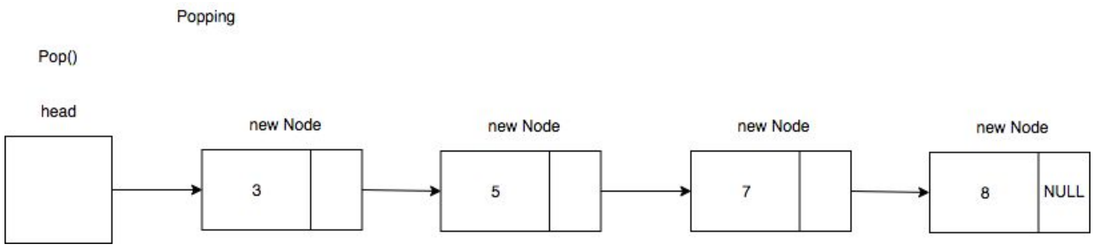

Figure 1.12: Stack before pop()

After checking the head pointer if it is not NULL, it creates a temp pointer and stores the address of node pointed by head in it.

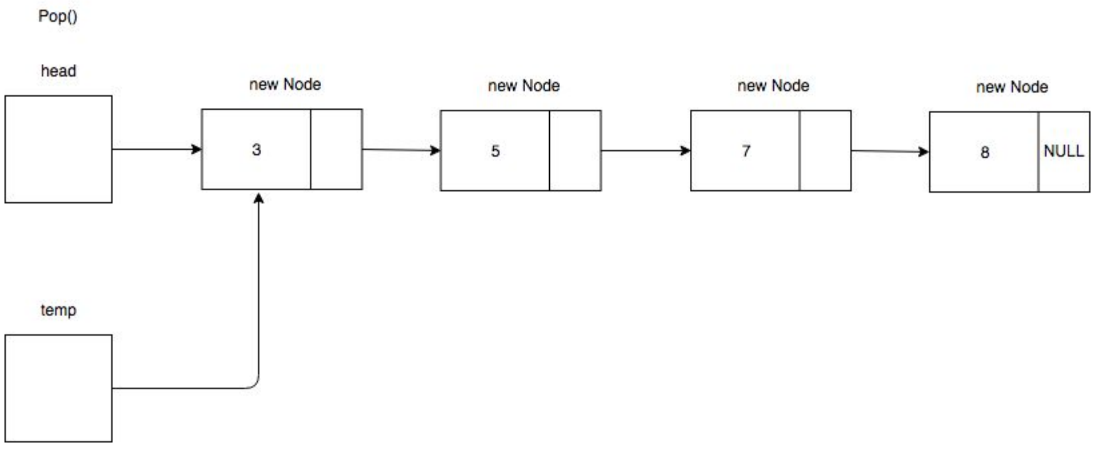

Figure 1.13: Creating temp pointer

It updates the head pointer to the node next to head

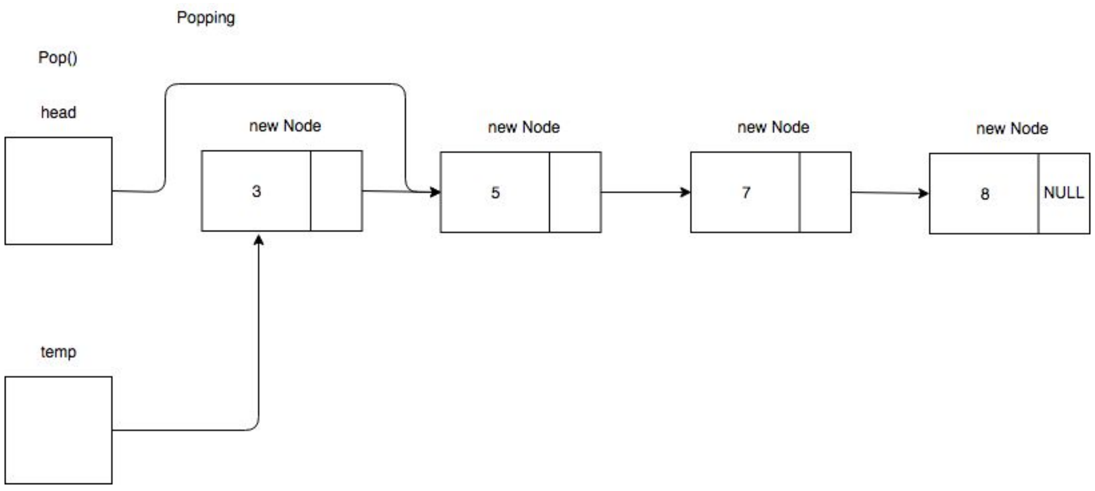

Figure 1.14: Updating head pointer

It now deletes the node pointed by temp.

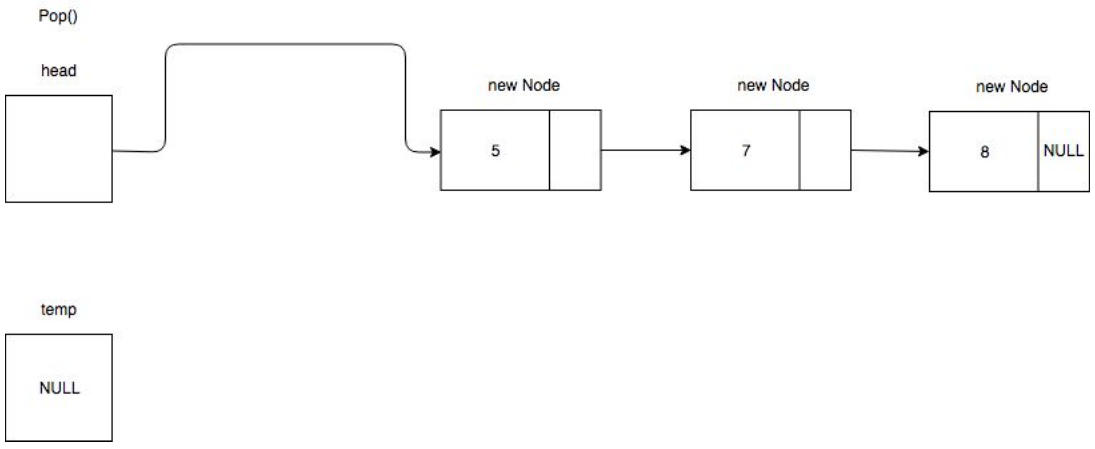

Figure 1.15: Deleting the node pointed by temp

The temp pointer then gets destroyed and stack is updated successfully.

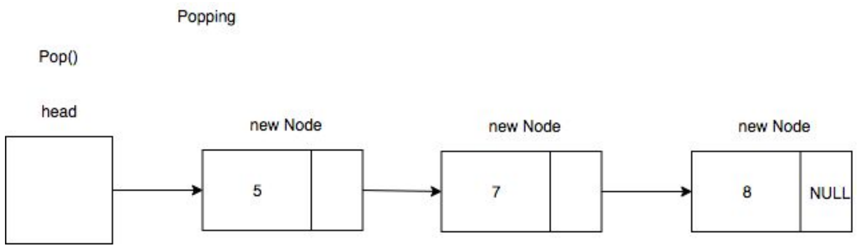

Figure 1.16: Succesful pop() operation

■

Implementing Show() function

The Show() function allows to output all the values in the stack.
Inside the class of Stack, we will declare and define a function "void Show()" in public.

```
void Show()
     Node* temp = head;
     while(temp!=NULL)
     {
          std::cout<<temp->data<<std::endl;
          temp = temp->next;
     }
```

Let's visualize an example of Show() operation

■ **Example 1.4**

```
     stack.Show();
```

Creating the temp pointer and points it to the node pointed by head.

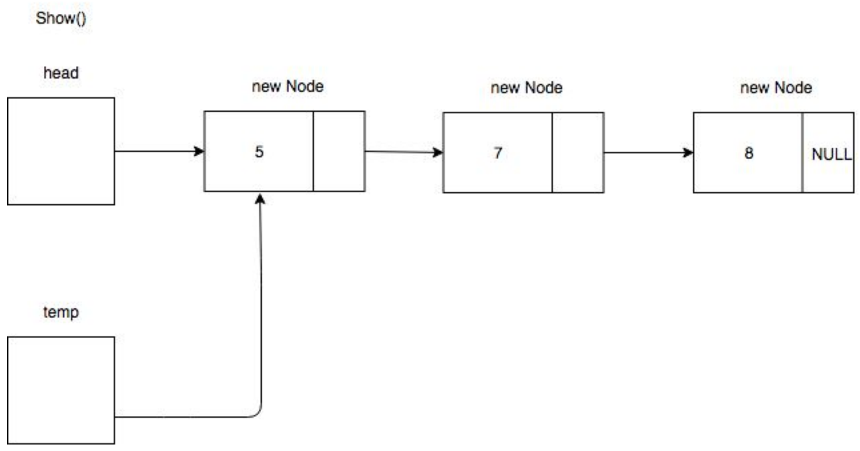

Figure 1.17: Creating temp pointer

Now it outputs the vaue stored in the node pointed by temp.

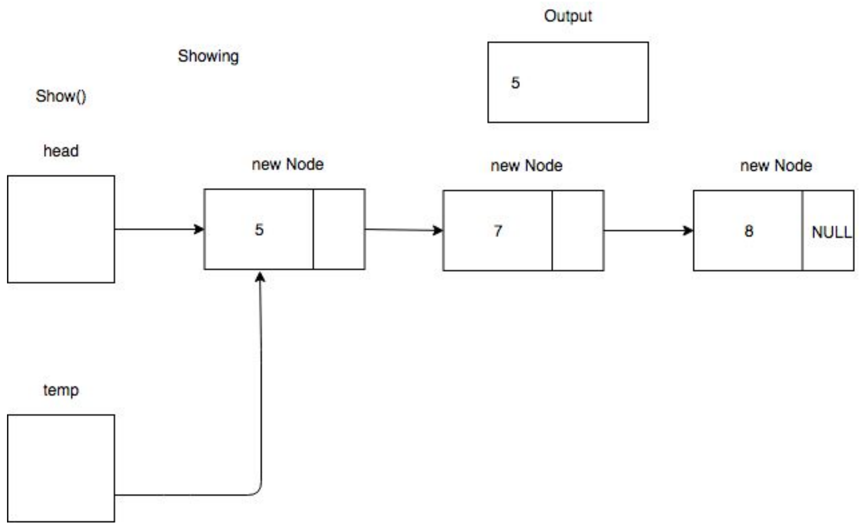

Figure 1.18: Outputting first value

It updates the temp pointer and moves to the next node. Then it outputs the value of the node pointed by temp.

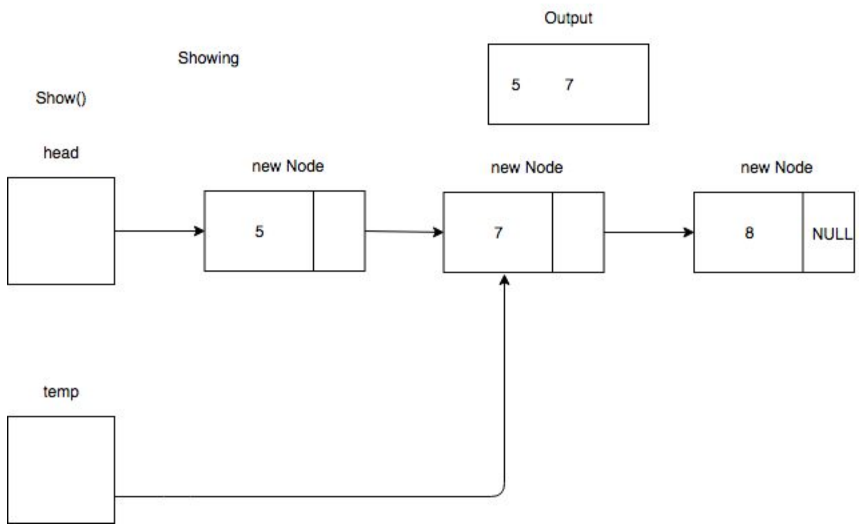

Figure 1.19: Updating temp pointer and outputting value

It repeats the updating of temp pointer and outputs the value

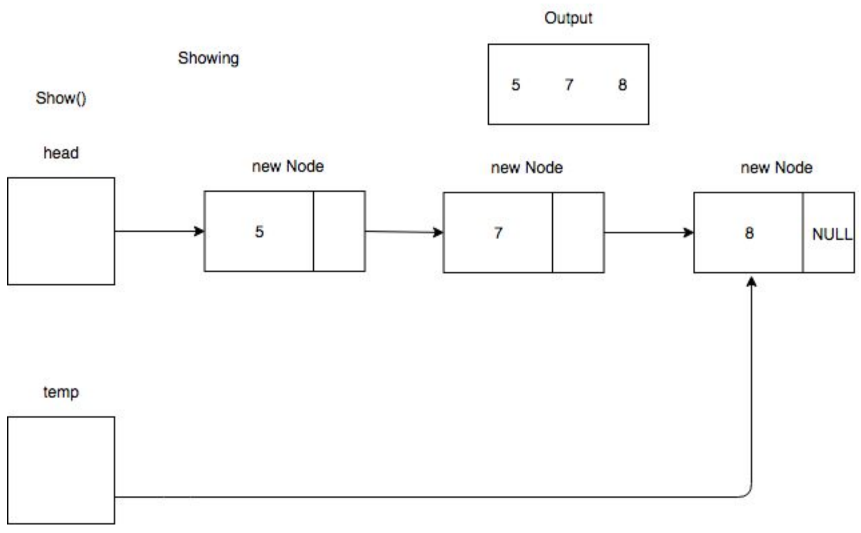

Figure 1.20: Updating the temp pointer to the third node and outputs the value

The loop gets stopped now because the temp is now set to NULL.

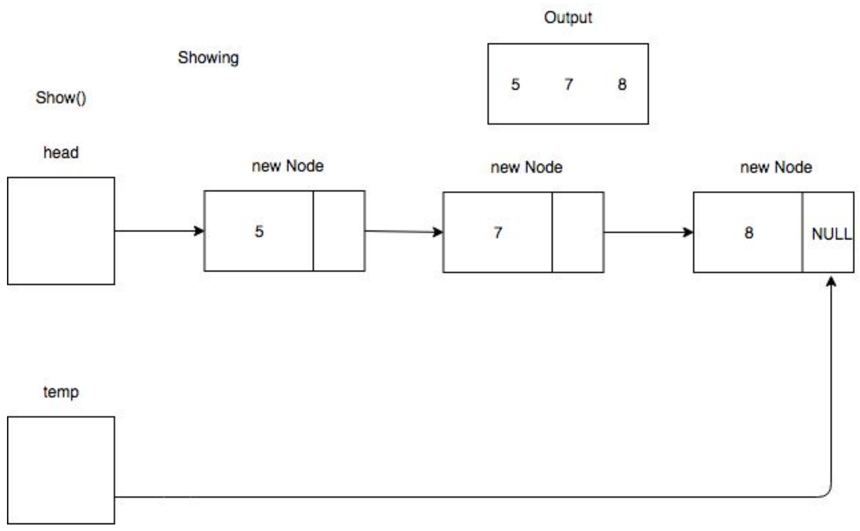

Figure 1.21: Updating the temp pointer to the third node and outputs the value

■

1.3 Problems

Problem 1.1 Implement the isEmpty() function in your Stack class. The function should return true if the stack is empty and return false if not.

Instructions

- Think about the value stored in head pointer when the stack is Empty

Problem 1.2 Implement the functions Top() and Bottom() in your Stack class. The Top() function should return the top element of the stack. The Bottom() function should return the element on the bottom of the stack.

Instructions

- Top() function is easy to implement after doing the problem above. Use head pointer to implement this function
- For Bottom() function, think how to iterate over the stack till the bottom using only head pointer and a temporary pointer.

1.4 Feedback

Please write the things you've learned from this lab and suggestions to make it more better and easy to learn.