# 1 Lab 1 Implementation of Stack Operations

## 1.1 Theory

### 1.1.1 Stack:

A stack is a fundamental abstract data type that adheres to the Last In, First Out (LIFO) principle. It models a collection of elements with two primary operations:

Push: Adding an element to the top of the stack.

Pop: Removing the topmost element from the stack.

A third operation, commonly supported, is:

Peek or Top: Viewing the top element without removing it.

Stacks are pivotal in computer science and software engineering for their role in managing function calls (known as the call stack), parsing and evaluating expressions, and supporting various algorithms like depth-first search. They find application in diverse scenarios such as compiler parsing, undo functionality in software, and managing memory in operating systems. Their simplicity and efficiency make them a cornerstone of data structure design and implementation.

## 1.2 Pseudo Code:

* **For Push Operations**

push(){  
 if (point = size -1)  
 print "Stack Overflow"  
 else  
 point++  
 arr[point] = value;  
}

* **For Pop Operations**

pop(){  
 if (point = -1)  
 print "Stack underflow"  
 else  
 print arr[point]  
 point--  
}

* **For Peek Operations**

peek(){  
 print arr[size-1]  
}

* **To display the entire stack**

showState(){  
 print "["  
 for (i = 0; i < size; i++)  
 print arr[i]  
 print "]"  
}

## 1.3 Source Code:

#include <stdio.h>  
  
void push(int\* arr, int val, int size, int\* point) {  
 if (\*point == size - 1)  
 printf("Stack overflow!");  
  
 else  
 arr[++(\*point)] = val;  
}  
void pop(int\* arr, int size, int\* point) {  
 if (\*point == -1)  
 printf("Stack underflow!");  
  
 else {  
 printf("%d", arr[(\*point)]);  
 arr[(\*point)--] = 0;  
 }  
}  
  
void peek(int\* arr, int size, int\* point) {  
 if (\*point == -1)  
 printf("Stack underflow!");  
  
 else  
 printf("%d", arr[\*point]);  
}  
  
void showState(int\* point, int\* arr, int size) {  
 printf("[");  
 int i, corr=0, n;  
  
 for (i = 0; i < size; i++)  
 printf(" %d", arr[i]);  
  
 printf("]\n");  
 /\* printf(" "); \*/  
  
 for (i = 0; i < (\*point)+1; i++) {  
 //count no of digits in arr[point]  
 n = arr[i];  
 corr=0;  
  
 while (n!=0) {  
 n=n/10;  
 corr++;  
 }  
  
 int j;  
  
 printf(" ");  
 /\* printf("corr: %d\n", corr); \*/  
  
 for (j=0; j<corr; j++) {  
 /\* printf("%d\n", j); \*/  
  
 printf(" ");  
 }  
  
 }  
  
 printf("^");  
}  
  
int main() {  
 int point = -1;  
 int inp, noexit = 1, val;  
 char exitence;  
 int arr[20];  
  
 while (noexit) {  
  
 printf("Choose a number.\n");  
 printf("1) Push value\n");  
 printf("2) Pop value\n");  
 printf("3) Peek value\n");  
 printf("4) Display Stack\n");  
 printf("> ");  
 scanf("%d", &inp);  
  
 switch (inp) {  
 case 1:  
 scanf("%d", &val);  
 push(arr, val, 10, &point);  
 break;  
  
 case 2:  
 pop(arr, 10, &point);  
 break;  
  
 case 3:  
 peek(arr, 10, &point);  
 break;  
  
 case 4:  
 showState(&point, arr, 10);  
 break;  
 }  
  
 printf("\nDo we exit?(y/n) ");  
 getchar();  
 scanf("%c", &exitence);  
 noexit = exitence == 'y' ? 0 : 1;  
 }  
  
 return 0;  
}

Output:

Choose a number.

1) Push value

2) Pop value

3) Peek value

4) Display Stack

> 1

12

Do we exit?(y/n) n

Choose a number.

1) Push value

2) Pop value

3) Peek value

4) Display Stack

> 1

3

Do we exit?(y/n)

Choose a number.

1) Push value

2) Pop value

3) Peek value

4) Display Stack

> 3

3

Do we exit?(y/n) n

Choose a number.

1) Push value

2) Pop value

3) Peek value

4) Display Stack

> 2

3

Do we exit?(y/n) n

Choose a number.

1) Push value

2) Pop value

3) Peek value

4) Display Stack

> 4

: [ 12 0 0 0 0 0 0 0 0 0]

^

Do we exit?(y/n) y

DISCUSSION:

The C code demonstrated here offers a direct implementation of a stack with essential functions like push, pop, displaying all elements, showcasing the top element, and providing an exit option. It employs a menu-based interface, enabling users to input numeric selections for different operations. Notably, the program effectively manages stack overflow and underflow scenarios, presenting clear messages to the user. The code utilizes basic input handling using getchar() and scanf(), contributing to its user-centric design. This implementation serves as a practical introduction to stack principles in C programming. Although functional, the program could be enhanced by incorporating error handling for invalid user inputs and refining the display format for improved readability. Overall, it establishes a sturdy foundation for comprehending and advancing stack concepts in C programming.

CONCLUSION:

This C program offers a functional implementation of a stack, empowering users to execute fundamental stack operations. It diligently checks for stack overflow and underflow situations, delivering informative messages to users. Its menu-driven approach fosters user-friendliness, simplifying interaction with the stack for individuals.