Experiment No.
Aim: To implement stack ADT using arrays.
Name: Nikhil Kamalaji Shingade
Roll no: 57
Date of Performance:
Date of Submission:

Experiment No. 1: To implement stack ADT using arrays

Aim: To implement stack ADT using arrays.

Objective:

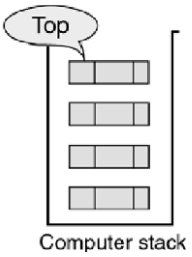
1) Understand the Stack Data Structure and its basic operators.

2) Understand the method of defining stack ADT and implement the basic operators.

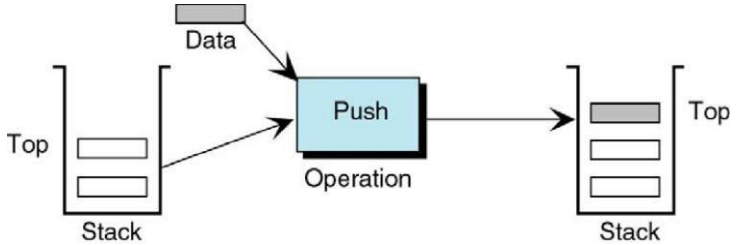
3) Learn how to create objects from an ADT and invoke member functions.

Theory:

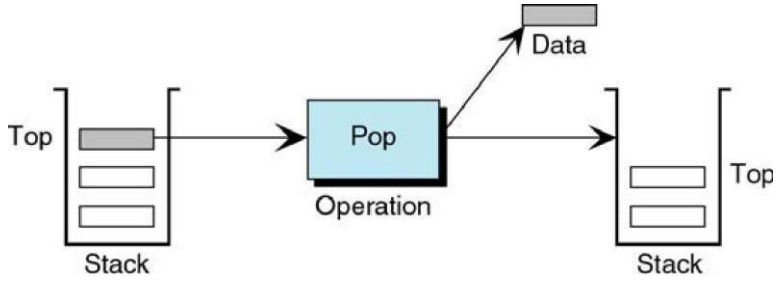
A stack is a data structure where all insertions and deletions occur at one end, known as the top. It follows the Last In First Out (LIFO) principle, meaning the last element added to the stack will be the first to be removed. Key operations for a stack are "push" to add an element to the top, and "pop" to remove the top element. Auxiliary operations include "peek" to view the top element without removing it, "isEmpty" to check if the stack is empty, and "isFull" to determine if the stack is at its maximum capacity. Errors can occur when pushing to a full stack or popping from an empty stack, so "isEmpty" and "isFull" functions are used to check these conditions. The "top" variable is typically initialized to -1 before any insertions into the stack.



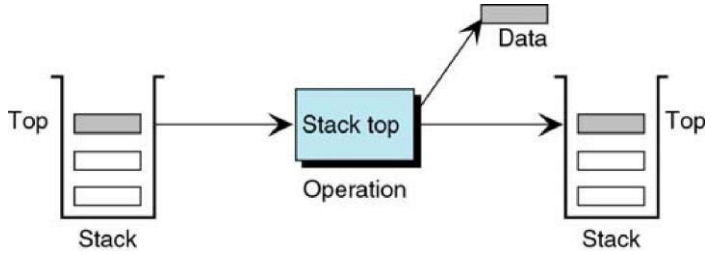
Push Operation



Pop Operation



Peek Operation



Algorithm:

PUSH(item)

1. If (stack is full)

Print "overflow" 2.

top = top + 1

3. stack[top] = item

Return

POP()

1. If (stack is empty)

Print "underflow"

2. Item = stack[top]

3. top = top – 1

4. Return item

PEEK()

1. If (stack is empty) Print "underflow"

2. Item = stack[top]

3. Return item

ISEMPTY()

1. If(top = -1)then

   return 1

2. return 0 ISFULL()

If(top = max)then return1

1. return 0

Code:

```
#include<stdio.h>

#define max 50 int
num,top=-1,a[max],i;
void push()

{
printf("enter element:"); scanf("%d",&num);
if(top==max-1)
{ printf("stack is
full");
}
else
{
top=top+1;
a[top]=num;
}
}
int pop()
{
if(top==-1)
{
printf("stack is empty"); return
0;
}
else
{
```

```
num=a[top];
top=top-1; return
num;
 }
}
void display()
{
if(top==-1)
 {
printf("stack is empty");
 }
else
 {
for(i=top;i>=0;i--)
{
printf("%d\t",a[i]);
   }
 }
}
void main()
{

push(); push();
push();
display();
printf("\n%d is poped",pop());

}
```

Output:

```
Output
/tmp/ILoe9VZ6LC.o
enter element:30
enter element:20
enter element:50
50  20  30
50 is poped

=== Code Exited With Errors ===
```

Conclusion:

1) What is the structure of Stack ADT?

Ans:

A Stack Abstract Data Type (ADT) is pretty intriguing. It's like a pile of plates last one placed is the first one removed. This "Last In, First Out" (LIFO) principle means it's all about maintaining order. You've got two main operations:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the top element.

There are auxiliary operations like **peek** (or top) to view the top element without removing it and **isEmpty** to check if the stack is empty

2) List various applications of stack?

Ans:

Application of stack

- **Expression Evaluation**: Used in parsing expressions, particularly for evaluating postfix expres sions.
- **Backtracking**: Useful in scenarios like mazesolving algorithms, where you need to backtrack to a previous position.
- **Undo Mechanism**: Many applications use stacks to implement undo functionalities— think text editors where you can revert to a previous state.
- **Function Call Management**: Most programming languages use a stack to manage function ca lls and local variables.
- **Parenthesis Checking**: Helps in checking balanced parentheses in compilers.
- **Browser History**: Maintains the history of visited pages in web browsers.

3) Which stack operation will be used when the recursive function call is returning to the calling function?

Ans:

When a recursive function call is returning to the calling function, the **pop** operation on the st ack is used. This removes the function call from the top of the stack, restoring the previous st ate. It's like peeling back the layers to get back to where you were. Recursive functions depen d on this mechanism to keep track of their calls.