Experiment No.1
Implement Stack ADT using array.
Name : Saurabh Mane
Roll no:25
Date of Performance:
Date of Submission:
Marks:
Sign:

Experiment No. 1: To implement stack ADT using arrays

Aim: To implement stack ADT using arrays.

Objective:

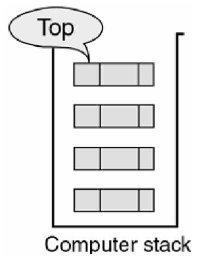
1) Understand the Stack Data Structure and its basic operators.

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

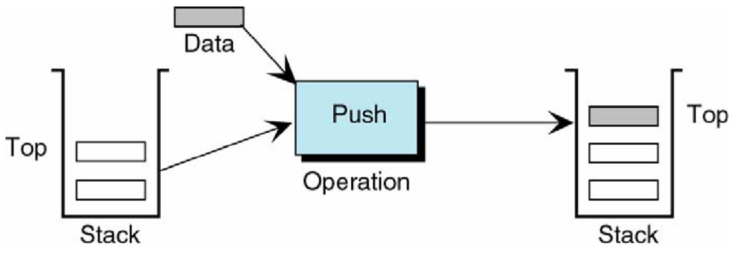
3) Learn how to create objects from an ADT and invoke member functions.

Theory:

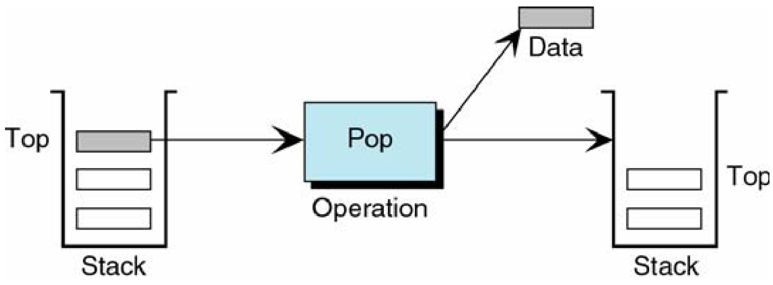
A stack is a data structure where all insertions and deletions occur at one end, known as the top. It follows the Last In First Out (LIFO) principle, meaning the last element added to the stack will be the first to be removed. Key operations for a stack are "push" to add an element to the top, and "pop" 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.



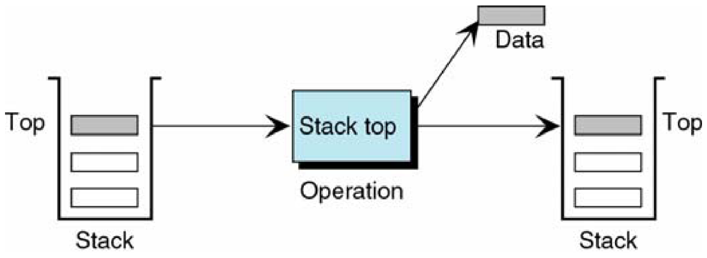
Push Operation



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()
1. If(top = max)then
        return 1
2. return 0
```

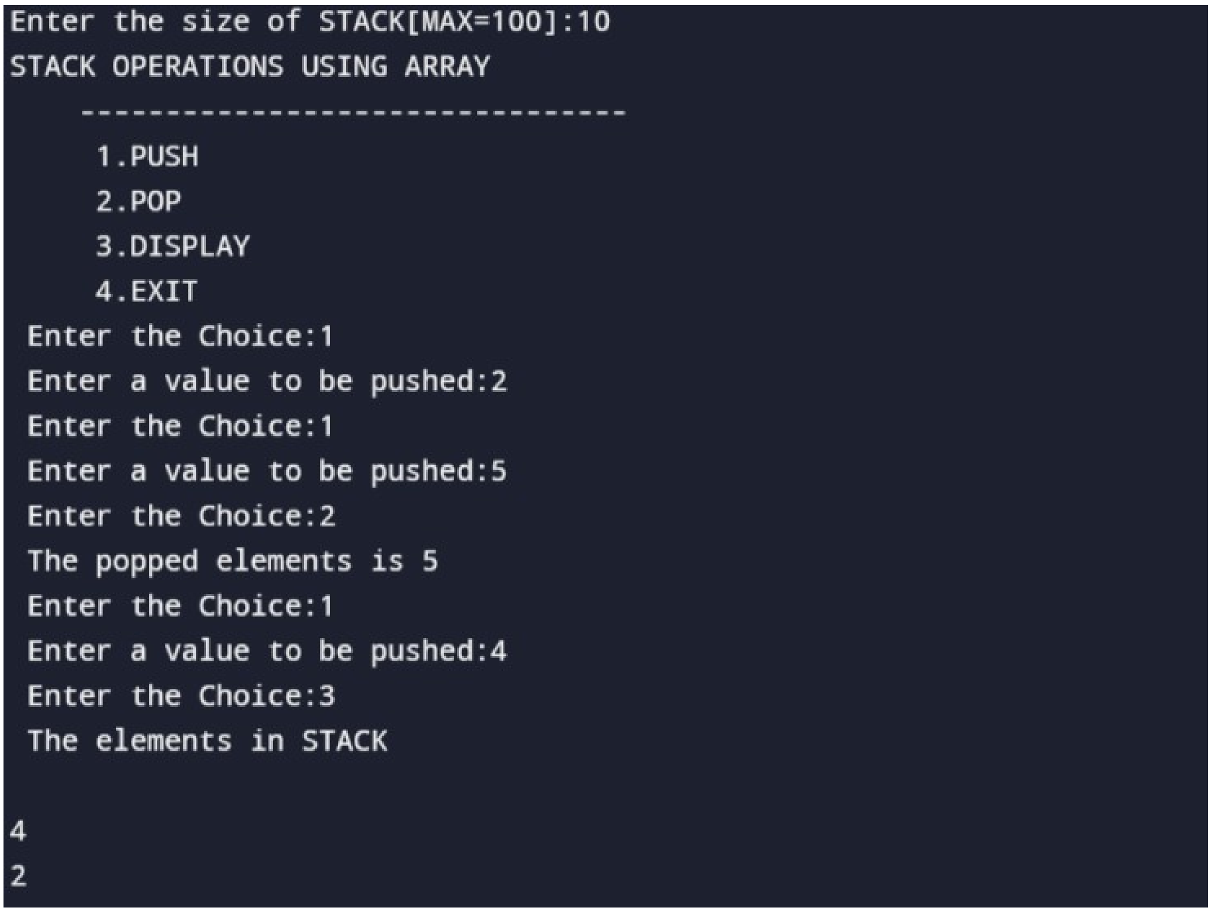
Code:

```
#include <stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
 void pop(void);
 void display(void);
 int main() {
 //clrscr();
 top=-1;
 printf("\n Enter the size of STACK[MAX=100]:");
 scanf("%d",&n);
printf("\n\t        STACK        OPERATIONS        USING        ARRAY");
printf("\n\t--------------------------------");
printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
 do { printf("\n Enter the Choice:");
```

```
scanf("%d",&choice);
switch(choice) {
 case 1: {
 push();
 break;
 } case 2: {
 pop();
break;
} case 3: {
 display();
 break;
 } case 4: {
 printf("\n\t EXIT POINT ");
 break;
} default: {
 printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
}
 }
 } while(choice!=4);
 return 0;
} void push() {
if(top>=n-1) {
 printf("\n\tSTACK is over flow"); }
 else {
 printf(" Enter a value to be pushed:");
 scanf("%d",&x);
```

```
top++; stack[top]=x; }
 }
 void pop() { if(top<=-1) {
 printf("\n\t Stack is under flow"); }
 else {
printf("\n\t The popped elements is %d",stack[top]);
 top--;
}
} void display() {
 if(top>=0) {
 printf("\n The elements in STACK \n");
 for(i=top; i>=0; i--)
printf("\n%d",stack[i]);
 printf("\n Press Next Choice"); }
else { printf("\n The STACK is empty"); } }
```

Output:



```
Enter the size of STACK[MAX=100]:10
STACK OPERATIONS USING ARRAY
    --------------------------------
     1.PUSH
     2.POP
     3.DISPLAY
     4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:2
 Enter the Choice:1
 Enter a value to be pushed:5
 Enter the Choice:2
 The popped elements is 5
 Enter the Choice:1
 Enter a value to be pushed:4
 Enter the Choice:3
 The elements in STACK

4
2
```

Conclusion:

The Stack Abstract Data Type (ADT) typically has the following structure:

1. Push: Adds an element to the top of the stack.

2. Pop: Removes and returns the element from the top of the stack.

3. Peek (or Top): Allows you to view the element at the top of the stack without removing it.

4. IsEmpty: Checks if the stack is empty.

5. Size: Returns the number of elements in the stack.

These operations are used to manage and manipulate the stack data structure according to the Last-In-First-Out (LIFO) principle.

Various applications of the stack data structure include:

1. Expression evaluation: Used for evaluating mathematical expressions in infix, postfix, or prefix notation.

2. Function call management: To keep track of function calls and their return values in a program.

3. Backtracking algorithms: Often used in algorithms like depth-first search.

4. Undo functionality: To implement undo and redo operations in applications.

5. Memory management: In computer systems, to manage function call stacks and local variables.

6. Syntax parsing: In parsing and evaluating programming language expressions.

7. Browser history: To maintain a history of web pages visited.

8. Task management: In managing tasks and their execution order.

9. Expression matching: To check for balanced parentheses and brackets in expressions.

10. Postfix calculations: To perform arithmetic calculations in postfix notation.

11. Undo/redo in text editors: To support undo and redo operations in text editing.

12. Playlists: To implement features like "back" and "next" in media players.

13. Call history: In mobile phones to keep track of recent calls.

14. Task scheduling: In operating systems for managing process execution.

15. Routing algorithms: In networking for routing decisions.

16. Symbol balancing: To check for balanced symbols in code (e.g., in programming languages).

17. Navigation systems: For maintaining routes and directions in GPS navigation.

When a recursive function call is returning to the calling function, the "Pop" operation is used to remove the information related to the completed function call from the call stack, allowing the program to continue executing from the point where the recursive call was made. This ensures that the program doesn't get stuck in an infinite recursion and that the stack space is freed up as functions return.