**LAB 1**

**Implementation of stack using array**

**Theory :**

A stack is a simple data structure that works on the **LIFO (Last In, First Out)** rule. This means that the last item you add to the stack is the first one you take out. Stacks are super handy in many situations, like keeping track of function calls, solving math expressions, or handling backtracking problems.

In this implementation, the stack is represented using a fixed-size array (static stacking). A variable, top, is used to track the index of the last added element. The primary operations performed on a stack include:

Push:

Adds an element to the top of the stack.

Pop:

Removes and retrieves an element from the top of the stack.

Peek: Returns the top element of the stack without removing it.

IsEmpty:

Checks whether there are no elements in the stack.

IsFull:

Checks whether the stack has reached its maximum capacity. **Algorithm:**

1. Push operation: Use isFull() to check if the stack is full. If it is full, print an error message and exit the operation. Otherwise, the top is increased and new elements are allocated to the stack [top].
2. Pop operation: Use iSEMPTY () to check if the stack is empty. If it is empty, print an error message and exit. Otherwise, get the element stack [top], decrement the top, and return the value.

Display Action If the stack is empty, print \"Stack is empty\".

Otherwise, iterate from 0 up and print all the elements on the stack.

1. End operation: End program execution.

**Source Code:**

#include<stdio.h>

#include<stdbool.h>

#include<stdlib.h>

#define MAXSIZE 10

int stack[MAXSIZE];

int top = -1;

bool isEmpty(){

if(top == -1){

return true;

}

else{

return false;

}

}

bool isFull(){

if(top == MAXSIZE - 1){ // Corrected condition

return true;

}

else{

return false;

}

}

void push(int num){

if(!isFull()){

top += 1;

stack[top] = num;

printf("Pushed %d successfully!\n", num);

}

else{

printf("Couldn't insert, stack is full!\n");

}

}

int pop(){

int num;

if(!isEmpty()){

num = stack[top]; // Get the top item BEFORE decrementing

top -= 1;

printf("Popped %d\n", num);

return num;

}

else{

printf("Couldn't pop, stack is empty!\n");

return -1;

}

}

int main(){

int choice, num;

while (1)

{

printf("\nStack Operations Menu:\n");

printf("1. Push (max limit is 10 datas)\n");

printf("2. Pop (max limit is 10 datas)\n");

printf("3. Display\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("Enter the number you want to push: ");

scanf("%d", &num);

push(num);

break;

case 2:

pop();

break;

case 3:

if (isEmpty()) {

printf("Stack is empty!\n");

} else {

printf("Stack contents:\n");

for(int i = 0; i <= top; i++){

printf("%d ", stack[i]);

}

printf("\n");

}

break;

case 4:

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

**Discussion and Conclusion:**

The stack is a widely used linear data structure that operates on the principle of Last In, First Out (LIFO). Implementing a stack using an array involves creating a fixed-size array and performing operations such as push, pop, and peek. This approach is straightforward, as arrays provide a contiguous memory allocation, allowing for fast access to elements based on their indices.The program effectively implements stack operations using a fixed-size array, performing push, pop, and display functions efficiently. It ensures data integrity with isFull() and isEmpty() checks, preventing overflow and underflow. This simple and reliable static stack design is ideal for small datasets but limited by its fixed size, making it unsuitable for dynamic or larger applications. To improve scalability, future iterations could incorporate dynamic memory allocation or a list-based implementation, enabling the stack to grow or shrink as needed. These enhancements would provide greater flexibility and efficiency, making the program more versatile and capable of handling larger or unpredictable data volumes.

**Name : Biraj Pandey**

**Roll no : ACE079BCT023**