Experiment No. 1: To implement stack ADT using arrays

Aim: To implement stack ADT using arrays.

Objective:

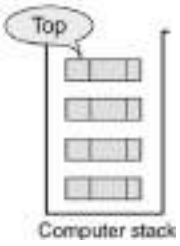
1) Understand the Stack Data Structure and its basic operators.

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

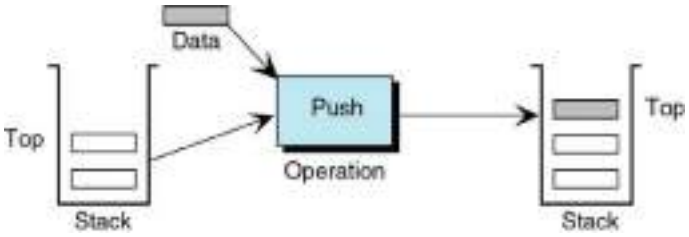
3) Learn how to create objects from an ADT and invoke member functions.

Theory:

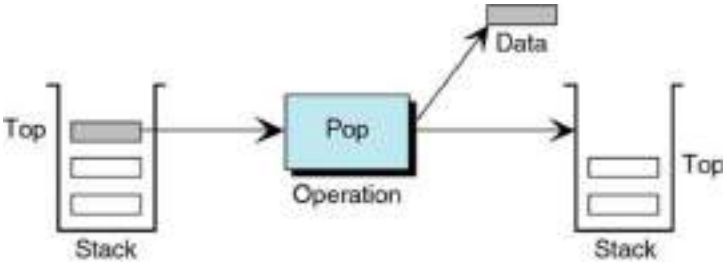
A stack is a data structure where all insertions and deletions occur at one end, known as the top. It follows the Last In First Out (LIFO) principle, meaning the last element added to the stack will be the first to be removed. Key operations for a stack are "push" to add an element to the top, and "pop"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



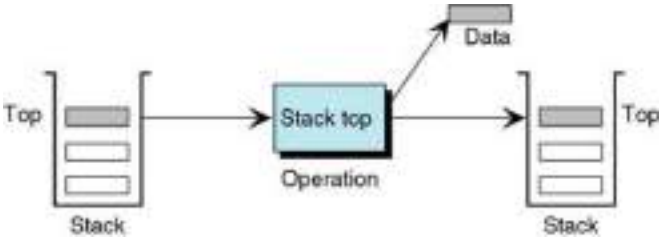
Push Operation



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>

#include<stdlib.h>

int stack[100],choice,n,top,x,i;

void push(void);

void pop(void);

void display(void);

void peek();


int main()

{

top=-1;

clrscr();

printf("Enter the size of stack[max=100]:");
```

```
scanf("%d",&n);
printf("Stack operation using array\n");
printf("\n\t 1.PUSH \n\t 2.POP \n\t 3.PEEK \n\t 4.DISPLAY \n\t 5.EXIT");
        do
        {
                printf("\nEnter your choice:");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1:
                        {
                                push();
                                break;
                        }
                        case 2:
                        {
                                pop();
                                break;
                        }
                        case 3:
                        {
                                peek();
                                break;
```

```
                }
                case 4:
                {
                        display();
                        break;
                }
                case 5:
                {
                        printf("\n\tEXIT POINT");
                        break;
                }
                default:
                {
                        printf("\n\t Please enter a valid choice(1/2/3/4)");
                }
        }
    }
    while(choice!=5);
return 0;
}

void push()
{
    if(top>=n-1)
```

```
        {
                printf("\n\t Stack is 'OVERFLOW' ");
        }
        else
        {
                printf("\n Enter a value to be pushed:");
                scanf("%d",&x);
                top++;
                stack[top]=x;
        }
}
void pop()
{
        if(top<=-1)
        {
                printf("\nStack is 'UNDERFLOW' ");
        }
        else
        {
                printf("\n\t The poped elements is %d:",stack[top]);
                top--;
        }
}
void display()
```

```
{
        if(top>=0)
        {
                printf("\n The element in stack:");
                for(i=top;i>=0;i--)
                {
                        printf("\n%d",stack[i]);
                        printf("\nPress next choice");
                }
        }
        else
        {
                printf("\nThe stack is empty");
        }
}
void peek()
{
        if(top<=-1)
        {
                printf("\n stack is Underflow");
        }
        else
        {
                printf("\n The peek element is %d:",stack[top]);
```

```
        }
}
```

Output:

Enter the size of stack[max=100]:4

Stack operation using array

1.PUSH

2.POP

3.PEEK

4.DISPLAY

5.EXIT

Enter your choice:1

Enter a value to be pushed:23

Enter your choice:2

The poped elements is 23:

Enter your choice:5

EXIT POINT

Conclusion:

Q1 What is the structure of Stack ADT?

```
if(top==N-1)
{
printf("stack overflow");
}
Else
```

```
{
top=top+1;
a[top]=x;
}
pop:
//let 'a' be the stack with the size'N'
if(top==-1)
{
printf("stack underflow");
}
else
{
x=a(top);
top=top-1;
print x as deleted
}
```

Q2 List various applications of stack?

Function calls and recursion: When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution.

Undo/Redo operations: The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To

undo the action, the top element of the stack is popped, and the reverse operation is performed.

Expression evaluation: Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements.

Browser history: Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.

Balanced Parentheses: Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced.

Backtracking Algorithms: The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack.

Q3 Which stack operation will be used when the recursive function call is returning to the calling function?

When a recursive function is called, a block of memory is created in the stack to hold the information of the currently executing function, and a different copy of local variables is created for each function call . When the recursive function call is returning to the calling function, the **pop** operation will be used to remove the block of memory from the stack that was created for the current function call . This will allow the program to return to the previous function call and continue its execution.