Experiment No.1
Implement Stack ADT using array.
Name: Umang Borse
Roll no:03
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:

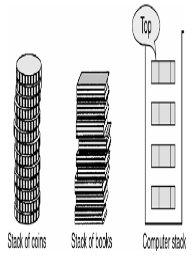
1) Understand the Stack Data Structure and its basic operators.

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

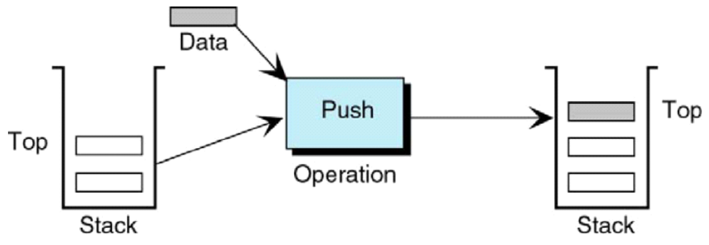
3) Learn how to create objects from an ADT and invoke member functions.

Theory:

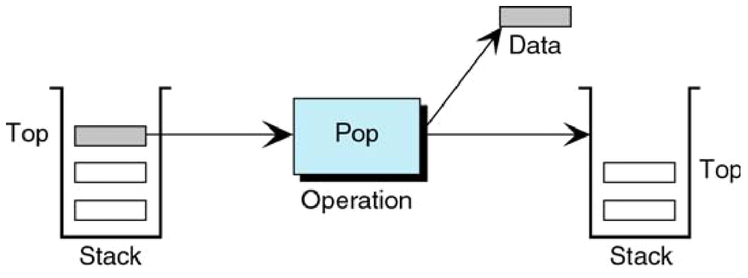
A stack is a data structure where all insertions and deletions occur at one end, known as the top. It follows the Last In First Out (LIFO) principle, meaning the last element added to the stack will be the first to be removed. Key operations for a stack are "push" to add an element to the top, and "pop"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



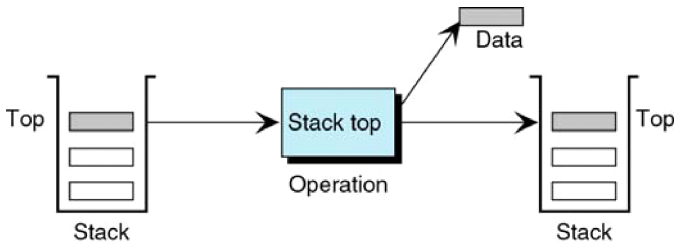
Push Operation



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>

#include<stdlib.h>


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

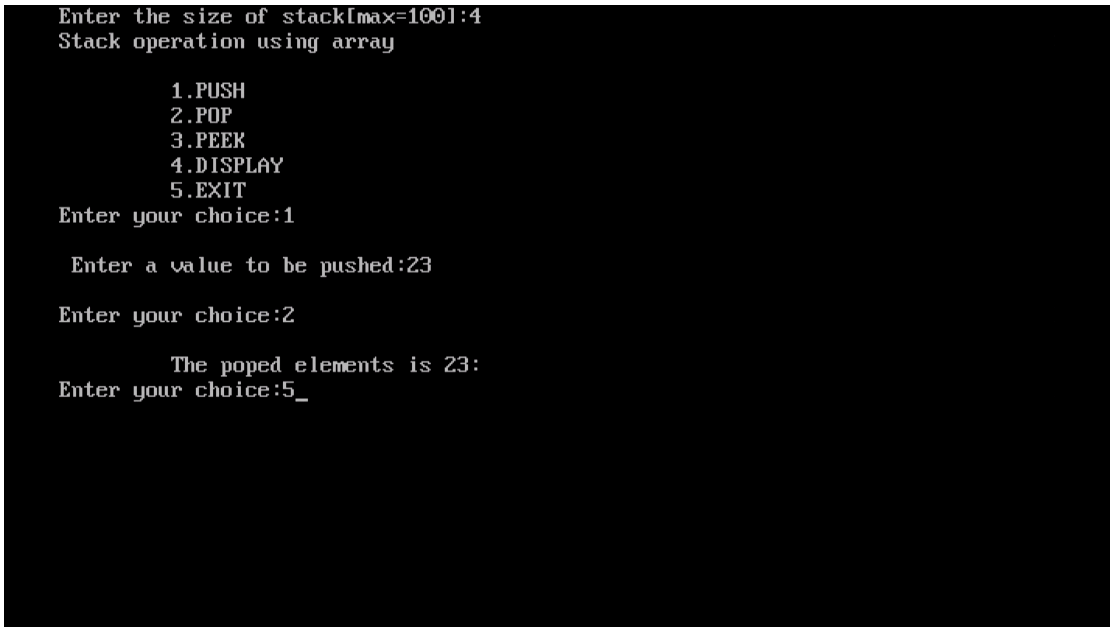
```
void push(void);

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_
```

Conclusion:

What is the structure of Stack ADT?

Stack ADT

A stack is a fundamental abstract data type in computer science, characterized by its Last-In-First-Out (LIFO) behavior. It is essentially a collection of elements with two primary operations: pushing and popping. The stack can be visualized as a vertical structure, similar to a stack of plates, where the last plate placed on top is the first one to be removed.

Structure of a Stack

1. Data Structure: Stacks can be implemented using various underlying data structures, with the most common being:

- Array-based Stack: In this implementation, the stack is realized as an array. The top of the stack corresponds to the last element in the array. When an element is pushed, it is added to the end of the array. Popping an element involves removing and returning the last element of the array. This implementation has a fixed size, and it can run into overflow or underflow issues if the stack size is not managed carefully.

- Linked List-based Stack: In this implementation, a stack is created using a linked list data structure. Each element in the stack is represented by a node, and these nodes are connected in a way that forms a stack. This implementation doesn't have a fixed size and can dynamically grow or shrink as needed.

2. Operations: The fundamental operations associated with a stack are:

- Push: This operation is used to add an element to the top of the stack. It involves creating a new element and placing it on top of the existing stack, effectively becoming the new top element.

- Pop: This operation is used to remove and return the element from the top of the stack. The element that was last pushed onto the stack is the one removed.

- Peek (or Top): This operation allows you to view the element at the top of the stack without removing it. It is often used to examine the top element before deciding whether to pop it.

- IsEmpty: This operation checks if the stack is empty. It returns true if the stack contains no elements and false otherwise.

- Size (or Length): This operation returns the number of elements currently in the stack.

3. Principle: The key principle behind a stack is the Last-In-First-Out (LIFO) behavior. The element that is most recently added is the first one to be removed. It mimics real-world scenarios where you pile up items, and the last item you placed on top is the one you'll access or remove first.

In practical programming, you would implement the Stack ADT using the chosen data structure (array or linked list) along with the associated operations. Stacks are widely used in algorithms, data processing, and managing function calls in computer memory. They offer a straightforward and efficient way to manage data in a particular order.

List various applications of stack?

1. Function call management in programming.

2. Expression evaluation, including infix to postfix conversion.

3. Backtracking in algorithms.

4. Undo/redo functionality in applications.

5. Parsing and compiling processes.

6. Delimiter matching in source code.

7. History management in web browsers.

8. Task management in operating systems.

9. Depth-First Search (DFS) in graph traversal.

10. Resource allocation and symbol table management in compilers.

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

When a recursive function call is returning to the calling function, the stack operation used is "Pop." This operation involves removing the current function call's activation record (or frame) from the call stack. The activation record contains information about the function's state, including local variables, parameters, and the return address.

Here's the typical sequence of events when a recursive function returns to the calling function:

1. The function call executes, and its activation record is pushed onto the stack.

2. When the recursive call within the function is completed, the function is ready to return.

3. The "Pop" operation is performed, removing the function's activation record from the stack.

4. The return address is retrieved from the popped activation record, and control is transferred back to the calling function at that return address.

5. The calling function can then continue its execution.

This process continues until all recursive calls have been completed, and the call stack becomes empty, effectively returning to the original caller. The "Pop" operation plays a crucial role in managing the call stack and controlling the flow of the program during recursive function calls.