K. J. Somaiya College of Engineering, Mumbai-77
(Autonomous College Affiliated to University of Mumbai)

Batch: D2	**Roll No.: 16010122323**
Experiment / assignment / tutorial No. 3	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Implementation of Basic operations on stack using Array and Linked List- Create, Insert, Delete, Peek.

Objective: To implement Basic Operations on Stack i.e. Create, Push, Pop, Peek

Expected Outcome of Experiment:

CO	Outcome
1	Explain the different data structures used in problem solving

Books/ Journals/ Websites referred:

1. *Fundamentals Of Data Structures In C* – Ellis Horowitz, Satraj Sahni, Susan Anderson-Fred
2. *An Introduction to data structures with applications* – Jean Paul Tremblay, Paul G. Sorenson
3. *Data Structures A Pseudo Approach with C* – Richard F. Gilberg & Behrouz A. Forouzan
4. *https://www.cprogramming.com/tutorial/computersciencetheory/stack.html*
5. *https://www.geeksforgeeks.org/stack-data-structure-introduction-program/*
6. *https://www.thecrazyprogrammer.com/2013/12/c-program-for-array-representation-of-stack-push-pop-display.html*

Abstract:

A Stack is an ordered collection of elements , but it has a special feature that deletion and insertion of elements can be done only from one end, called the top of the stack(TOP). The order may be LIFO(Last In First Out) or FILO(First In Last Out).
Students need to first try and understand the implementation of using arrays. Once comfortable with the concept, they can further implement stacks using linked list as well.

Related Theory: -

Stack is a linear data structure which follows a particular order in which the operations are performed. It works on the mechanism of Last in First out (LIFO).

List 5 Real Life Examples:

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.

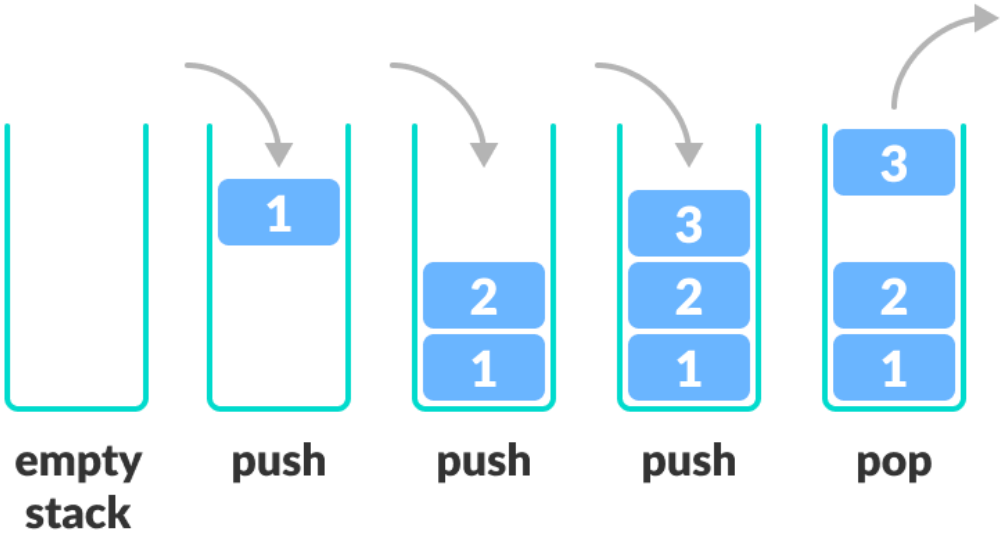
Undo/Redo operations: The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.

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.

Diagram:



Explain Stack ADT:

STACK is a simple linear data structure used for storing data. Stack follows the LIFO(Last In First Out) strategy that states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first. It can be implemented through an array or linked lists. Some of its main operations are: push(), pop(), top(), isEmpty(), size(), etc.

Algorithm for creation, insertion, deletion, displaying an element in stack:

INSERTION:-

1 . Checks if the stack is full.

2 . If the stack is full, produces an error and exit.

3 . If the stack is not full, increments top to point next empty space.

4 . Adds data element to the stack location, where top is pointing.

5 . Returns success.

DELETION:-

1 . Checks if the stack is empty.
2 . If the stack is empty, produces an error and exit.
3 . If the stack is not empty, accesses the data element at which top is pointing.
4 . Decreases the value of top by 1.
5 . Returns success.

PEEK:-

1.Checks if the stack is full.
2. If the stack is full, produces an error and exit.
3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.

Display operation:

1.If top is -1 (empty stack), indicate underflow.
2. Otherwise, iterate from top to 0 and display each element.

Create a stack:

Initialize an empty array to hold stack elements.
Initialize a variable top and set it to -1 (indicating an empty stack).

Implementation Details:

Built-In Functions/Header Files Used: (exit() etc):-

BUILT IN FUNCTIONS:-

1.MALLOC

2.SCANF

3.EXIT

4.FREE

HEADER FILES:-

1.stdlib.h

2.stdio.h

3.stdbool.h

Programm source code:-

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int info;
   struct node *ptr;
}*top,*top1,*temp;

int count = 0;
void push(int data) {
    if (top == NULL)
   {
      top =(struct node *)malloc(1*sizeof(struct node));
      top->ptr = NULL;
      top->info = data;
   }
   else
   {
```

```
        temp =(struct node *)malloc(1*sizeof(struct node));
        temp->ptr = top;
        temp->info = data;
        top = temp;
    }
    count++;
    printf("value is Inserted\n\n");
}

int pop() {
     top1 = top;

    if (top1 == NULL)
    {
        printf("\nStack Underflow\n");
        return -1;
    }
    else
        top1 = top1->ptr;
    int popped = top->info;
    free(top);
    top = top1;
    count--;
    return popped;
}

void display() {

    top1 = top;

    if (top1 == NULL)
    {
        printf("\nStack Underflow\n");
```

```
    return;
  }

  printf("The stack is \n");
  while (top1 != NULL)
  {
    printf("%d--->", top1->info);
    top1 = top1->ptr;
  }
  printf("NULL\n\n");

}

int main() {
  int choice, value;
  printf("\nDynamic stack\n");
  while (1) {
    printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("\nEnter your choice : ");
    scanf("%d", &choice);
    switch (choice) {
    case 1:
      printf("\nEnter the value to insert: ");
      scanf("%d", &value);
      push(value);
      break;
    case 2:
      printf("Popped element is :%d\n", pop());
      break;
    case 3:
      display();
      break;
    case 4:
```

```
            exit(0);
            break;
        default:
            printf("\nWrong Choice\n");
        }
    }
}
```

Output Screenshots:

```
Dynamic stack

1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1
Enter the value to insert: 25
value is Inserted
```

```
1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1
Enter the value to insert: 45
value is Inserted
```

```
1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 2
Popped element is :45
```

```
1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 2
Popped element is :25
```

```
Enter your choice : 2
Stack Underflow
```

APPLICATION 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.

Explain the Importance of the approach followed by you:-

The approach I used to explain the stack data structure involved introducing its Last-In-First-Out behaviour, showcasing real-world examples of its usage, explaining its Abstract Data Type with operations, presenting an algorithm for its operations, reviewing provided code, discussing assumptions and their importance, highlighting its

wide-ranging applications, offering concise information about used functions and header files, and ensuring tailored clarity and customization for understanding.

Conclusion:-

Exploring the stack data structure illuminates its core properties and practical applications, bridging theory and implementation for effective problem-solving in various domains.

PostLab Questions:

1) Explain how Stacks can be used in Backtracking algorithms with example.

Ans. Stacks are crucial in backtracking algorithms for managing the state ofexploration and efficiently implementing the "undo" mechanism. For instance, in the"Depth-First Search" algorithm, a stack keeps track of nodes to be explored, enabling asystematic traversal of paths and backtracking when necessary. This approach efficiently explores all possible paths before revisiting decisions, ensuring a comprehensive search

2) Illustrate the concept of Call stack in Recursion.

Ans:- The call stack is a data structure that the JavaScript interpreter uses to keep track of its place in a script that calls multiple functions. JavaScript has a single call stack, therefore only one function can be executed at a time. This is what makes JavaScript synchronous. The call stack works on a Last-In First-Out principle. This means that the last function to be added to the call stack will be the first to execute and pop off the stack. When a function is called, it is added to the call stack. Any functions that are called from within that function are then added to the top of the call stack. Multiple functions can be added to the stack in this manner.

For example, consider the factorial function `factorial(n)`, where `factorial(5)` results in a call stack like:

1. factorial(5) pushed with n=5

2. factorial(4) pushed with n=4

3. factorial(3) pushed with n=3

4. factorial(2) pushed with n=2

5. factorial(1) pushed with n=1

6. Base case reached, pop stack frames: factorial(1) popped, factorial(2) continues, and so on.

.