**Batch: B4 Roll No.: 16010122221**

**Experiment / assignment / tutorial No. 4**

**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

1. *Data Structures A Pseudo Approach with C –* Richard F. Gilberg & Behrouz A. Forouzan
2. [*https://www.cprogramming.com/tutorial/computersciencetheory/stack.html*](https://www.cprogramming.com/tutorial/computersciencetheory/stack.html)
3. [*https://www.geeksforgeeks.org/stack-data-structure-introduction-program/*](https://www.geeksforgeeks.org/stack-data-structure-introduction-program/)
4. [*https://www.thecrazyprogrammer.com/2013/12/c-program-for-array-representation-of-stack-push-pop-display.html*](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:**

**Diagram:**

**Explain Stack ADT:**

The Stack Abstract Data Type (ADT) is a fundamental concept in computer science and programming. It represents a linear data structure that follows the Last-In-First-Out (LIFO) principle, which means that the last element added to the stack is the first one to be removed. Stacks are commonly used for managing data in a way that resembles a stack of items, like a stack of plates or a stack of books.

A Stack ADT provides two primary operations:

1. Push: This operation adds an element to the top of the stack. As a result, the element becomes the new top, and any existing elements are pushed down. It's analogous to placing an item on top of a physical stack.

2. Pop: This operation removes the element from the top of the stack. The element that was added most recently (i.e., the top element) is removed, and the element just below it becomes the new top. It's like removing the top item from a physical stack.

Additionally, there are often auxiliary operations associated with stacks:

- Peek/Top: This operation retrieves the top element from the stack without removing it. It allows you to examine the element at the top without altering the stack's contents.

- isEmpty: This operation checks whether the stack is empty or not. It returns a Boolean value indicating whether there are any elements in the stack.

Stacks are commonly used in various applications, including but not limited to:

1. Expression Evaluation: Stacks can be used to evaluate arithmetic expressions, such as converting infix expressions to postfix or prefix notation and then calculating their values.

2. Function Call Management: When a function is called, its local variables and information about where to return after the function finishes need to be stored. Stacks are used to manage this information, known as the call stack.

3. Undo/Redo functionality: Stacks can be used to implement undo and redo operations in applications, where the current state of an action is pushed onto the stack, and undoing involves popping the stack.

4. Browser History: The back and forward navigation in web browsers can be implemented using stacks. Each visited page is pushed onto the stack and navigating backward (going back in history) involves popping elements from the stack.

The implementation of a stack can be done using various data structures such as arrays or linked lists. Depending on the use case and the specific requirements, different implementations might be preferred for efficiency and ease of use.

To summarize, a Stack ADT is a simple yet powerful data structure that follows the LIFO principle and is widely used in programming to manage data and control flow in various applications.

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

CreateStack:

Initialize an empty array called stackArray

Initialize a variable topIndex = -1

Push(element):

if topIndex is at the maximum size:

Display "Stack Overflow"

else:

Increment topIndex

stackArray[topIndex] = element

Pop:

if topIndex is -1:

Display "Stack Underflow"

else:

Remove element = stackArray[topIndex]

Decrement topIndex

return element

Peek:

if topIndex is -1:

Display "Stack is empty"

else:

return stackArray[topIndex]

**Implementation Details:**

**Assumptions made for Input:**

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

**Program source code:**

#include<stdio.h>

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

void push(void);

void pop(void);

void display(void);

int main()

{

top=-1;

printf("\n Enter the size of STACK:");

scanf("%d",&n);

printf("\n\t 1.PUSH\t 2.POP\t 3.DISPLAY\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 Screenshots:**

**A screenshot of a computer program

Description automatically generated**

**Applications of Stack:**

Stacks find applications in various domains of computer science and programming due to their Last-In-First-Out (LIFO) nature and their ability to efficiently manage data and control flow. Here are some common applications of stacks:

1. \*\*Expression Evaluation:\*\* Stacks are used to evaluate arithmetic expressions, convert between different notations (infix, postfix, prefix), and calculate their values.

2. \*\*Function Call Management:\*\* Stacks are essential for managing function calls and local variables. The call stack keeps track of the order in which functions are called and their local variables.

3. \*\*Undo/Redo Operations:\*\* Stacks are used in applications that require undo and redo functionality. The current state is pushed onto the stack, allowing users to go back in history by popping the stack.

4. \*\*Parsing and Syntax Checking:\*\* Stacks are used in compilers and interpreters to parse and check the syntax of programming languages. They help track opening and closing parentheses, brackets, and braces.

5. \*\*Backtracking Algorithms:\*\* Backtracking algorithms, like depth-first search and recursive algorithms, often utilize stacks to manage the exploration of solution spaces and backtrack when needed.

6. \*\*Browser History:\*\* Stacks are used to implement back and forward navigation in web browsers. Each visited page is pushed onto the stack, allowing users to navigate through their history.

7. \*\*Memory Management:\*\* Stacks play a role in memory management for programs. Local variables and function call information are stored in the stack memory.

**Conclusion:-**

Learnt to implement Basic Operations on Stack i.e., Create, Push, Pop, Peek

**PostLab Questions:**

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

Stacks are essential in backtracking algorithms as they facilitate systematic exploration of solution spaces. During backtracking, choices made at decision points are pushed onto the stack. If a choice leads to a dead end, the stack is popped, and alternatives are explored. For instance, consider the N-Queens problem: Placing N queens on an N×N chessboard without threats. Using a stack, you start with the first column, pushing queen positions and backtracking upon conflicts. When a valid solution isn't possible, you backtrack to the previous column, adjusting choices as needed. The stack stores the state, aiding in decision-making and efficient exploration. This mechanism enables solving complex combinatorial problems where finding the right solution requires trying out various options and reverting when necessary, forming an integral part of the backtracking paradigm.

1. **Illustrate the concept of Call stack in Recursion.**

The call stack plays a crucial role in understanding recursion. Recursion is a programming technique where a function calls itself in order to solve a problem. The call stack keeps track of function calls and their parameters, enabling the program to remember where to return after each recursive call.

Let's illustrate this with a simple example of calculating the factorial of a number using recursion:

```python

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

result = factorial(5)

print(result)

```

1. When `factorial(5)` is called:

- The function checks if `n` is 0, which is not the case.

- The function returns `5 \* factorial(4)`.

2. `factorial(4)` is called:

- The function checks if `n` is 0, which is not the case.

- The function returns `4 \* factorial(3)`.

3. This pattern continues until `factorial(0)` is reached, which returns 1.

Now, let's visualize the call stack:

```

| | |

| | |

| | |

|factorial| n=0 | <- factorial(0) returns 1

|factorial| n=1 | <- factorial(1) returns 1 \* 1

|factorial| n=2 | <- factorial(2) returns 2 \* 1

|factorial| n=3 | <- factorial(3) returns 3 \* 2

|factorial| n=4 | <- factorial(4) returns 4 \* 6

|factorial| n=5 | <- factorial(5) returns 5 \* 24 (final result)

```

As the recursive calls are made, each function call is pushed onto the call stack along with its parameters. The return values are computed as each call returns, eventually leading to the computation of the final result.

However, it's important to note that if the recursion goes too deep without proper base cases, the call stack could overflow, causing a "stack overflow" error. To prevent this, recursion should always have base cases that terminate the recursive calls.