**Batch: B-1 Roll No.: 16010122104**

**Experiment / assignment / tutorial No.**

**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.

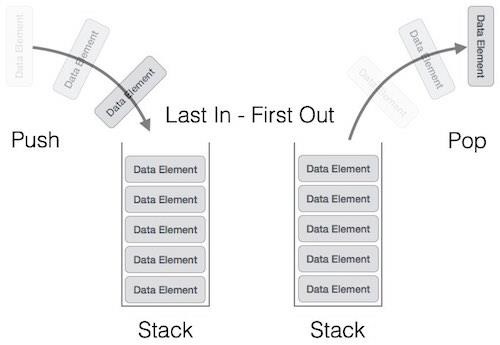
**Related Theory: -**

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:**

1. Stack of Plates: Imagine a stack of plates in a kitchen. New plates are added to the top, and when you need a plate, you take it from the top. The last plate you added will be the first one you use.
2. Browser Back Button: Web browsers use a stack to keep track of the pages you've visited. When you click the back button, it takes you to the previous page you visited.
3. Undo Functionality: Many software applications (like text editors or graphic design tools) provide an undo feature. Each action you perform is placed on a stack, and when you undo, it removes the most recent action (the top of the stack).
4. Call Stack in Programming: In programming, a call stack is used to manage function calls. When a function is called, it's pushed onto the stack, and when it returns, it's popped off. This ensures that the program knows where to continue after the function call.
5. Stack of Books: When you pile books on a table, the book you place last will be the first one you can easily pick up.

**Diagram:**



**Explain Stack ADT:**

A Stack, in the context of Abstract Data Types (ADT), is a linear data structure that follows a specific order for adding and removing elements. It's based on the Last-In-First-Out (LIFO) principle, which means that the last element added to the stack will be the first one to be removed.

The fundamental operations associated with a Stack ADT are:

1. Push: This operation is used to add an element to the top of the stack.

2. Pop: This operation removes and returns the top element from the stack.

3. Peek (or Top): This operation retrieves the top element without removing it from the stack.

4. isEmpty: It checks if the stack is empty or contains elements.

Stacks have a variety of practical applications in computer science and programming, such as:

- Function Call Stack: In programming languages like C/C++ or Java, the function call stack is used to manage function calls and their return addresses.

- Expression Evaluation: Stacks can be used to evaluate expressions, such as postfix or prefix expressions.

- Undo Functionality: Many software applications implement undo functionality using a stack to keep track of changes made.

- Backtracking Algorithms: Stacks are used in algorithms like depth-first search to explore and backtrack through paths or solutions.

- Memory Management: Stacks are used in memory management to allocate and deallocate memory for function calls and local variables.

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

1. Define the maximum size of the stack as MAX\_SIZE.

2. Define a structure named "Stack" with two members:

- An integer array named "data" with a size of MAX\_SIZE to store stack elements.

- An integer variable named "top" to keep track of the top element's index in the stack.

3. Create a function called "createStack" that takes a pointer to a Stack structure as a parameter:

- Initialize the "top" member of the stack structure to -1, indicating an empty stack.

4. Create a function called "isEmpty" that takes a pointer to a Stack structure as a parameter:

- Return true (1) if the "top" member of the stack is equal to -1; otherwise, return false (0).

5. Create a function called "isFull" that takes a pointer to a Stack structure as a parameter:

- Return true (1) if the "top" member of the stack is equal to MAX\_SIZE - 1; otherwise, return false (0).

6. Create a function called "push" that takes a pointer to a Stack structure and an integer value as parameters:

- Check if the stack is full using the "isFull" function:

- If the stack is full, print a message indicating that the stack is full, and return.

- Increment the "top" member of the stack.

- Add the integer value to the "data" array at the index indicated by the updated "top."

7. Create a function called "pop" that takes a pointer to a Stack structure as a parameter:

- Check if the stack is empty using the "isEmpty" function:

- If the stack is empty, print a message indicating that the stack is empty and return a sentinel value (e.g., -1) to indicate failure.

- Retrieve the top element from the "data" array using the "top" index.

- Decrement the "top" member of the stack.

- Return the retrieved element.

8. Create a function called "peek" that takes a pointer to a Stack structure as a parameter:

- Check if the stack is empty using the "isEmpty" function:

- If the stack is empty, print a message indicating that the stack is empty and return a sentinel value (e.g., -1) to indicate failure.

- Retrieve the top element from the "data" array using the "top" index.

- Return the retrieved element.

9. In the "main" function:

- Declare a Stack variable named "stack."

- Call the "createStack" function to initialize the stack.

- Demonstrate stack operations:

- Push elements onto the stack.

- Peek at the top element.

- Pop elements from the stack.

- Check if the stack is empty.

10. The program concludes after demonstrating the stack operations.

**Implementation Details:**

**Assumptions made for Input:**

1. MAX\_SIZE: Maximum stack size is predefined.

2. Push: Assumes a source of integer values.

3. Pop and Peek: Assume stack is not empty.

4. Input Validation: Limited validation.

5. User Interaction: Triggered by users.

6. Error Handling: Basic error messages.

7. Termination: Program eventually ends.

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

The algorithm only uses standard library functions like <stdio.h> for input/output or <stdlib.h> for memory allocation. It doesn't utilize functions like printf, scanf, malloc, or free.

**Program source code:**

#include <stdio.h>

#include <stdlib.h>

// Define the maximum size of the stack

#define MAX\_SIZE 100

// Define the stack structure

struct Stack {

int data[MAX\_SIZE];

int top;

};

// Function to initialize an empty stack

void createStack(struct Stack \*stack) {

stack->top = -1; // Initialize the top pointer to -1 (empty stack)

}

// Function to check if the stack is empty

int isEmpty(struct Stack \*stack) {

return stack->top == -1;

}

// Function to check if the stack is full

int isFull(struct Stack \*stack) {

return stack->top == MAX\_SIZE - 1;

}

// Function to push an element onto the stack

void push(struct Stack \*stack, int value) {

if (isFull(stack)) {

printf("Stack is full. Cannot push.\n");

return;

}

stack->data[++stack->top] = value;

printf("%d pushed onto the stack.\n", value);

}

// Function to pop an element from the stack

int pop(struct Stack \*stack) {

if (isEmpty(stack)) {

printf("Stack is empty. Cannot pop.\n");

return -1; // Return a sentinel value to indicate failure

}

int value = stack->data[stack->top--];

return value;

}

// Function to peek at the top element of the stack without removing it

int peek(struct Stack \*stack) {

if (isEmpty(stack)) {

printf("Stack is empty. Cannot peek.\n");

return -1; // Return a sentinel value to indicate failure

}

return stack->data[stack->top];

}

int main() {

struct Stack stack;

createStack(&stack);

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

printf("Top element: %d\n", peek(&stack));

printf("Popped element: %d\n", pop(&stack));

printf("Popped element: %d\n", pop(&stack));

printf("Popped element: %d\n", pop(&stack));

printf("Is stack empty? %s\n", isEmpty(&stack) ? "Yes" : "No");

return 0;

}

**Output Screenshots:**

A white screen with black text

Description automatically generated

**Applications of Stack:**

1. Task Scheduling: In multitasking environments, stacks help manage the execution of tasks or threads. They provide a way to switch between tasks and return to the previously executing context.
2. Parentheses Matching: Stacks can check if parentheses in a text (or code) are balanced. An open parenthesis is pushed onto the stack, and when a closing parenthesis is encountered, it's popped if matched.
3. Evaluation of Postfix Expressions: After converting expressions to postfix, stacks are used to evaluate them efficiently.
4. Browser History: Web browsers use stacks to keep track of visited pages. When you click back, you pop the current page from the stack to return to the previous one.
5. Undo/Redo Functionality: Stacks can be employed in applications like text editors to implement undo and redo functionality. Changes to the text are pushed onto the undo stack and can be popped to redo or undone.

**Explain the Importance of the approach followed by you**

This program defines a stack using a structure and provides functions for creating a stack, checking if it's empty or full, pushing elements onto the stack, popping elements from the stack, and peeking at the top element without removing it.

.

**Conclusion:-**

We have learned 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 fundamental data structures that play a crucial role in backtracking algorithms. Backtracking is a technique used to find solutions to problems through systematic exploration of all possible options. Stacks are employed to keep track of the choices made at each step and to facilitate the process of backtracking when a dead end is encountered. Here's how stacks are used in backtracking algorithms with an example:

Example: The N-Queens Problem The N-Queens problem is a classic example often used to illustrate how stacks are employed in backtracking. The problem is to place N chess queens on an N×N chessboard so that no two queens threaten each other.

How Stacks Are Used:

1. Choice Representation: At each step, the algorithm needs to make a choice. For example, in the N-Queens problem, it needs to decide where to place the next queen. The stack is used to keep track of these choices.
2. Exploration: The algorithm explores one choice and pushes it onto the stack. It then proceeds to the next choice, pushing it onto the stack as well. This process continues until a solution is found or a dead end is reached.
3. Backtracking: When a dead end is encountered (i.e., a situation where the current choice cannot lead to a solution), the algorithm pops the stack to return to the previous choice point. This is the essence of backtracking. It allows the algorithm to backtrack to a previous state and explore alternative choices.
4. Revisiting Choices: By popping the stack, the algorithm revisits a previous choice and explores other options. This process continues until a valid solution is found or all possible choices have been exhausted.

Illustration: Suppose we're solving the N-Queens problem with N = 4. The stack would record the positions of queens at each step. Here's a simplified representation of the stack during exploration:

1. Push (1, 1) onto the stack. (Queen at row 1, column 1)
2. Push (2, 3) onto the stack. (Queen at row 2, column 3)
3. Encounter a dead end.
4. Pop (2, 3) from the stack and backtrack to the previous choice.
5. Push (2, 4) onto the stack. (Queen at row 2, column 4)
6. Push (3, 2) onto the stack. (Queen at row 3, column 2)
7. Encounter a dead end.
8. Pop (3, 2) from the stack and backtrack to the previous choice.

This process continues until a valid solution is found, or all possible combinations are exhausted.

In summary, stacks provide a structured way to manage choices and backtrack when solving problems using backtracking algorithms. They allow efficient exploration of all possible solutions while maintaining a record of choices made along the way.

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

In recursion, the call stack plays a vital role in managing function calls. Let's take the calculation of the factorial of 4 (4!) as an example. We start by calling the factorial(4) function. Inside factorial(4), it calls factorial(3) and awaits the result. The call stack at this point contains only factorial(4). Within factorial(3), it calls factorial(2) and awaits its result, adding factorial(3) to the call stack. Inside factorial(2), it calls factorial(1), adding factorial(2) to the call stack. factorial(1) directly returns 1, so factorial(2) can compute (2 \* 1 = 2) and return it to factorial(3). The call stack now contains only factorial(4) and factorial(3). factorial(3) computes (3 \* 2 = 6) and returns it to factorial(4), leaving only factorial(4) in the stack. Finally, factorial(4) calculates (4 \* 6 = 24) and returns it as the final result, emptying the call stack. The call stack keeps track of pending function calls and their results, allowing the program to execute recursive functions effectively.