THEORY – 4

PROGRAM -1

AIM- STACK OPERATIONS

|  |
| --- |
| Stack is a fundamental data structure that follows the Last In, First Out (LIFO) principle. It can be imagined as a stack of plates where you can only add (push) or remove (pop) items from the top of the stack. Here are the key operations associated with a stack:  ### Stack Operations:  1. \*\*Push Operation (Insertion):\*\*  - This operation adds an element to the top of the stack.  - If the stack is full (in case of a fixed-size array-based implementation), it results in a "Stack Overflow" condition.  - Time complexity: O(1)  2. \*\*Pop Operation (Deletion):\*\*  - This operation removes and returns the top element from the stack.  - If the stack is empty, it results in a "Stack Underflow" condition.  - Time complexity: O(1)  3. \*\*Peek or Top Operation (Access):\*\*  - This operation returns the top element of the stack without removing it.  - It checks the element at the top of the stack without modifying the stack itself.  - Time complexity: O(1)  4. \*\*isEmpty Operation (Check if stack is empty):\*\*  - This operation checks if the stack is empty.  - Returns `true` if the stack is empty, otherwise `false`.  - Time complexity: O(1)  ### Implementation Approaches:  1. \*\*Array-based Implementation:\*\*  - In this approach, the stack is implemented using a fixed-size array.  - Operations (push, pop, peek) are performed by manipulating the array indices.  - Efficient for a known maximum capacity but can lead to stack overflow if the capacity is exceeded.  - Time complexity: O(1) for push, pop, peek.  2. \*\*Linked List-based Implementation:\*\*  - In this approach, the stack is implemented using a singly linked list where each node contains a data element and a pointer to the next node.  - Push and pop operations involve inserting or deleting nodes at the beginning of the list.  - Memory allocation is dynamic, and the stack can grow as needed (no fixed size limitation).  - Time complexity: O(1) for push, pop, peek.  ### Example of Stack Usage (Array-based Implementation in C):  ```c  #include <stdio.h>  #define MAXSIZE 10  int stack[MAXSIZE];  int top = -1;  void push(int value) {  if (top == MAXSIZE - 1) {  printf("Stack Overflow\n");  } else {  stack[++top] = value;  printf("Pushed %d to the stack\n", value);  }  }  int pop() {  if (top == -1) {  printf("Stack Underflow\n");  return -1; // Return -1 for underflow condition  } else {  return stack[top--];  }  }  int peek() {  if (top == -1) {  printf("Stack is empty\n");  return -1; // Return -1 if stack is empty  } else {  return stack[top];  }  }  int isEmpty() {  return top == -1;  }  int main() {  push(10);  push(20);  push(30);  printf("Top element: %d\n", peek());  while (!isEmpty()) {  printf("Popped %d from the stack\n", pop());  }  printf("Stack is empty: %s\n", isEmpty() ? "true" : "false");  return 0;  }  ```  In the above example:  - `push(value)` adds `value` to the stack.  - `pop()` removes and returns the top element of the stack.  - `peek()` returns the top element without removing it.  - `isEmpty()` checks if the stack is empty.  This is a basic implementation of a stack using an array in C, demonstrating the essential stack operations. |