**Batch: A2 Roll No.:16010123032**

**Experiment / assignment / tutorial No. 2**

**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 you're stacking plates on a table. You place each plate on top of the stack, and when you need to remove a plate, you take the top one off. This is a classic example of a stack in real life, where the last plate you put on the stack is the first one you remove.

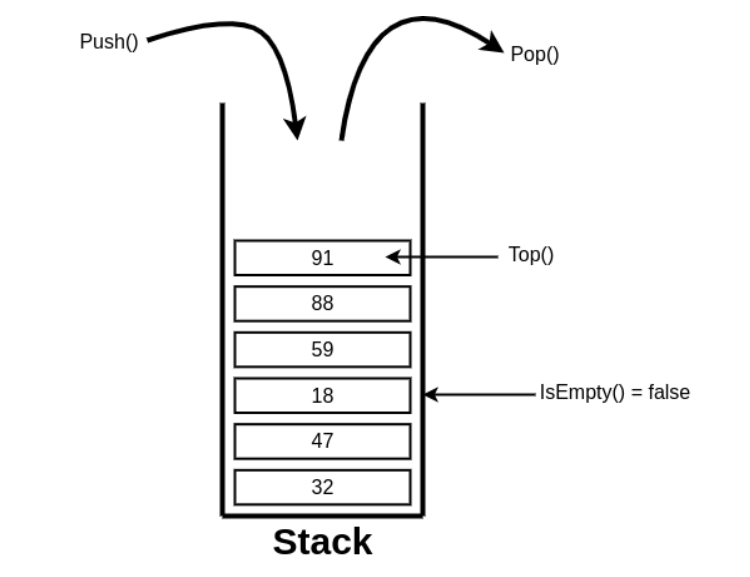
**2.** Book Pile: When you're organizing a pile of books, you might add books to the top of the pile and take books off the top when you want to read them. Just like a stack, the last book you added to the pile is the first one you'll pick up.

**3.** Browser Back Button: When you browse the internet, your web browser keeps track of the pages you visit in a stack-like manner. As you click on links, each new page is added to the stack of visited pages. When you press the back button, the most recently visited page (at the top of the stack) is displayed.

**4.** Undo/Redo Functionality in Software: Many software applications offer the ability to undo or redo actions. These actions are often managed using a stack. Each action you perform is pushed onto the stack, allowing you to revert the most recent action by popping it off the stack.

**5.** Call Stack in Programming: In programming, when functions are called, their information is pushed onto the call stack. This includes things like the function's return address and local variables. As functions complete, they are popped off the stack, allowing the program to return to the previous function

**Diagram:**

****

**Explain Stack ADT:**

**Stack ADT: Value definition**

Abstract typedef StackType(ElementType ele)

Condition: none

**Stack ADT: Operator definitions**

**1.** Abstract StackType CreateStack()

Precondition: none

Postcondition: EmptyStack is created

**2.** Abstract StackType PushStack(StackType Stack, ElementType Element)

Precondition: Stack not full or NotFull(Stack)= True

Postcondition: stack= stack + Element at the top

Or Stack= original stack with new Element at the top

**3.** Abstract ElementType PopStack(StackType stack)

Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: PopStack= element at the top,

Stack = stack - Element at the top

Or Stack= original stack without top Element

**4.** Abstract DestroyStack(StackType Stack)

Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: Element from the stack are removed one by one starting from top to bottom.

Empty(Stack)= True

**5.** Abstract Boolean NotFull(StackType stack)

Precondition: none

Postcondition: NotFull(Stack)= true if Stack is not full

NotFull(Stack)= False if Stack is full.

**6.** Abstract Boolean NotEmpty(StackType stack)

Precondition: none

Postcondition: NotEmpty(Stack)= true if Stack is not empty

~Empty(Stack)= False if Stack is empty.

**7.** Abstract ElementType Peep(StackType stack)

Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: PeepStack= element at the top,

Stack = original stack

A Stack ADT (Abstract Data Type) is a conceptual model that defines a collection of elements with two primary operations: "push" and "pop." It 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. The Stack ADT is often used to represent a linear data structure in which elements are stacked on top of each other.

**Here are the key operations associated with a Stack ADT:**

**1. Create:** This operation creates an empty stack.

**2. Push:** This operation adds an element to the top of the stack. The element that is pushed becomes the new "top" of the stack.

**3. Pop:** This operation removes and returns the element from the top of the stack.

After the pop operation, the element that was below the removed element

becomes the new "top."

**4. Destroy:** This operation removes elements one by one from the stack till it becomes empty.

**5. NotFull:** This operation checks the stack and returns true if stack is not full and false if the stack is full.

**6. NotEmpty:** This operation checks the stack and returns true if stack is not

empty and false if the stack is empty

**7. Peek (or Top):** This operation returns the element at the top of the stack

without removing it.

**8. Size:** This operation returns the number of elements currently present in the stack.

The Stack ADT can be visualized as a collection of items stacked one on top of another, resembling a physical stack of plates or books. The most recently added item is at the top, and the least recently added item is at the bottom.

Here's a simple example of using a stack to demonstrate the push and pop operations:

Initial Stack: Empty

• Push 1: Stack becomes [1]

• Push 2: Stack becomes [1, 2]

• Push 3: Stack becomes [1, 2, 3]

• Pop: Returns 3, and stack becomes [1, 2]

• Pop: Returns 2, and stack becomes [1]

• Pop: Returns 1, and stack becomes empty

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

**Stack Operations:**

**1. Create a Stack:**

Algorithm:

1. Initialize an empty array to represent the stack.

2. Return the stack.

**2. Push (Insertion):**

**Algorithm:**

1. Accept the element to be pushed.

2. Append the element to the end of the stack.

**3. Pop (Deletion):**

**Algorithm:**

1. If the stack is empty, return an error indicating underflow.

2. Retrieve and store the top element of the stack.

3. Remove the top element from the stack.

4. Return the stored element.

**4. Display:**

Display stack.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

int arr[MAX];

int top=-1;

int isFull(){

return top == MAX -1;

}

int isEmpty(){

return top ==-1;

}

void push(int num){

if(isFull())

printf("Stack is Full\n");

else{

top=top+1;

arr[top] = num;

printf("%d is pushed to Stack",num);

}

}

void pop(){

if (isEmpty())

printf("Stack Underflow\n");

else

{

printf("Popped element is : %d\n",arr[top]);

top=top-1;

}

}

void display(){

int i;

if (isEmpty())

printf("Stack is empty\n");

else

{

printf("Stack elements :\n");

for(i = top; i >=0; i--)

printf("%d\n", arr[i] );

}

}

void peek(){

if (isEmpty())

printf("Stack is empty\n");

else

printf("Top element is %d",arr[top]);

}

int main() {

int choice, num;

while (1) {

printf("\n Stack Operations:\n");

printf("1. Push\n");

printf("2. Pop\n");

printf("3. Peek\n");

printf("4. Display\n");

printf("5. Exit\n");

printf("Enter your choice(1-5): ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to push: ");

scanf("%d", &num);

push(num);

break;

case 2:

pop();

break;

case 3:

peek();

break;

case 4:

display();

break;

case 5:

printf("Exit the Program\n");

exit(0);

default:

printf("Invalid choice! Please enter a number between 1 and 5.\n");

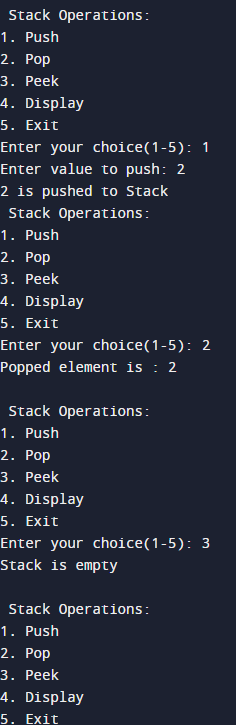
}

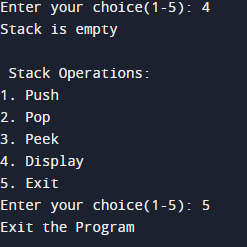
}

return 0;

}

**Output:**

****



**Applications of Stack:**

1. **Function Call Management (Call Stack):** Stacks are used in programming languages to manage function calls. When a function is called, its information, including local variables and the return address, is pushed onto the stack. When the function returns, this information is popped from the stack.
2. **Expression Evaluation:** Stacks can be used to evaluate arithmetic expressions, especially those with parentheses. They help in maintaining the order of operations and can be used to convert infix expressions to postfix or prefix notation for efficient evaluation.
3. **Backtracking:** In algorithms like depth-first search (DFS), backtracking, and maze solving, a stack is used to keep track of the path or state at each step. If a dead-end is reached, the algorithm can backtrack to the last known good state stored on the stack.
4. **Undo Mechanisms:** Many software applications, including text editors and graphic design tools, implement undo functionality using a stack. Each action is pushed onto the stack, and undoing involves popping actions from the stack.
5. **Compiler Syntax Checking:** Stacks are used to check the syntax of programming code. For example, they can be used to ensure that opening and closing brackets, braces, and parentheses are properly balanced.

**Post Lab Questions:**

1. **List 5 Applications of Stack Data Structures.**

 **Expression Evaluation and Conversion:**

* Stacks are used to evaluate expressions, particularly postfix and prefix expressions. They are also used in converting infix expressions (common in arithmetic) to postfix or prefix forms.

 **Function Call Management:**

* The call stack is a fundamental part of most programming languages, keeping track of active subroutines or function calls. It stores return addresses, local variables, and parameters.

 **Undo Mechanism in Text Editors:**

* Stacks are used to implement undo features in text editors. Each action (e.g., typing a character) is pushed onto a stack, and when undo is triggered, the last action is popped from the stack and reverted.

 **Parenthesis Matching:**

* Stacks help in checking the balance of parentheses in an expression. By pushing opening brackets onto the stack and popping them when closing brackets are encountered, the stack can verify proper nesting.

 **Browser History Navigation:**

* Web browsers use stacks to manage the back and forward navigation history. When a user visits a new page, it is pushed onto the stack, and when the back button is clicked, the current page is popped off the stack, returning to the previous page.

2) **Convert the given Infix Expression into Postfix Expression using Stack:**

**(A-B/C)\*(D\*E-F)**

| **Step** | **Symbol** | **Action** | **Stack** | **Postfix Expression** |
| --- | --- | --- | --- | --- |
| 1 | ( | Push to stack | ( |  |
| 2 | A | Add operand to postfix expression | ( | A |
| 3 | - | Push to stack | (- | A |
| 4 | B | Add operand to postfix expression | (- | AB |
| 5 | / | Push to stack (higher precedence) | (- / | AB |
| 6 | C | Add operand to postfix expression | (- / | ABC |
| 7 | ) | Pop stack until ( is found | (- -> ( | ABC/ -> ABC/- |
| 8 | \* | Push to stack | \* | ABC/- |
| 9 | ( | Push to stack | \*( | ABC/- |
| 10 | D | Add operand to postfix expression | \*( | ABC/-D |
| 11 | \* | Push to stack | \*( \* | ABC/-D |
| 12 | E | Add operand to postfix expression | \*( \* | ABC/-DE |
| 13 | - | Pop \* and push - to stack | \*( - | ABC/-DE\* |
| 14 | F | Add operand to postfix expression | \*( - | ABC/-DE\*F |
| 15 | ) | Pop stack until ( is found | \* | ABC/-DE\*F- |
| 16 | \* | Pop remaining stack | Empty | ABC/-DE\*F-\* |

3) **Explain How stack can be used in both Nested Function calls and Recursion using suitable examples for each. Further Define Activation Records used for Function Calling.**

**Stack in Nested Function Calls**

1. **Stack Frame Creation**:
   * When a function A calls another function B, a new stack frame for B is created and pushed onto the stack, holding information such as local variables, parameters, and the return address.
2. **Return to Caller**:
   * After B completes, its stack frame is popped off, and control returns to A using the return address stored in A's stack frame.
3. **Nested Calls Management**:
   * This process continues for any number of nested calls, ensuring that each function completes in the reverse order of how it was called, maintaining proper execution flow.

**Stack in Recursion**

1. **Stack Frame for Each Call**:
   * In a recursive function, each recursive call generates a new stack frame, similar to how it would in nested function calls, but this time, the same function is re-entered with different parameters.
2. **State Preservation**:
   * The stack helps preserve the state of each recursive call, allowing the function to return to the exact point where it left off after completing the deeper recursive calls.
3. **Termination and Unwinding**:
   * Once the base case is reached (termination condition), the stack begins to unwind, popping off each stack frame and returning the results step-by-step to the previous calls, eventually resolving the entire recursive sequence.