LAB 2

Experiment 1

OBJECTIVE: To perform various operations on a stack using array implementation.

THEORY:

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The basic operations performed on a stack are:

1. Push – Adds an element to the top of the stack.
2. Pop – Removes the top element from the stack.
3. Display – Shows all elements currently in the stack.

Stacks can be implemented using arrays or linked lists. In this lab, we use an array to implement the stack.

Algorithm:

1. Push Operation:
    - Check if the stack is full (Overflow condition).
    - If not full, increment the top pointer and insert the new element.

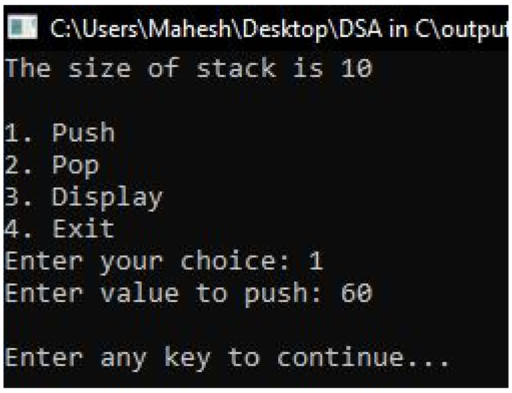
2. Pop Operation:
    - Check if the stack is empty (Underflow condition).
    - If not empty, remove the top element and decrement the top pointer.

3. Display Operation:
    - If the stack is empty, display an appropriate message.
    - Otherwise, traverse the stack from top to bottom and print all elements.

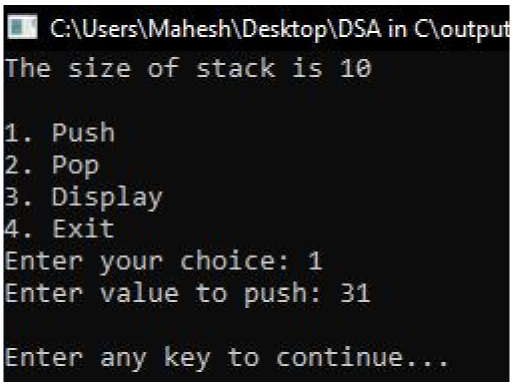
PROGRAMS

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

int main()
{
    int stack[100], top = -1, n;
    int i, choice, value;
    printf("Enter the size of stack: ");
    scanf("%d", &n);
    do
    {
        system("cls");
        printf("The size of stack is %d \n", n);
        printf("\n1. Push\n2. Pop\n3. Display\n4.
Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            if (top == n - 1)
                printf("Stack Overflow\n");
            else
            {
                printf("Enter value to push: ");
                scanf("%d", &value);
                stack[++top] = value;
            }
            break;
        case 2:
            if (top == -1)
                printf("Stack Underflow\n");
            else
            {
                value = stack[top--];
                printf("Popped value: %d\n",
value);
            }
            break;
        case 3:
            if (top == -1)
                printf("Stack is empty\n");
            else
            {
                printf("Stack elements are: ");
                for (i = top; i >= 0; i--)
                    printf("%d ", stack[i]);
                printf("\n");
            }
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice\n");
        }
        printf("\nEnter any key to continue...");
        getch();
    } while (choice != 4);
    return 0;
}
```

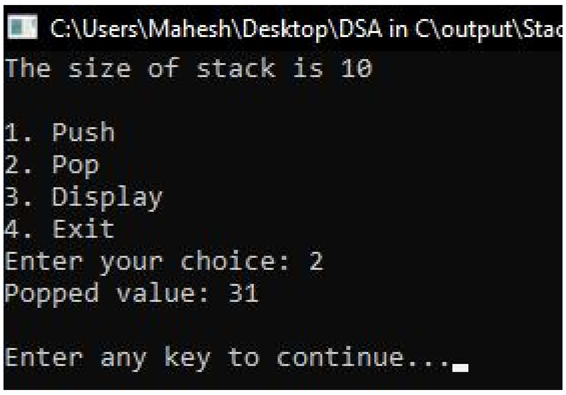
Output:



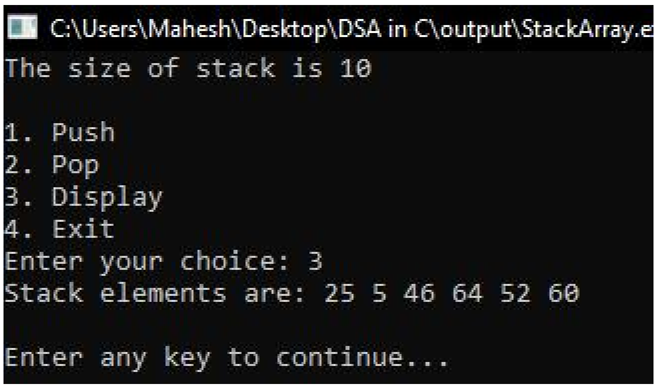
```
C:\Users\Mahesh\Desktop\DSA in C\output
The size of stack is 10

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 60

Enter any key to continue...
```



```
C:\Users\Mahesh\Desktop\DSA in C\output
The size of stack is 10

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 31

Enter any key to continue...
```



```
C:\Users\Mahesh\Desktop\DSA in C\output\Stac
The size of stack is 10

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped value: 31

Enter any key to continue...
```



```
C:\Users\Mahesh\Desktop\DSA in C\output\StackArray.e
The size of stack is 10

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements are: 25 5 46 64 52 60

Enter any key to continue...
```

CONCLUSION:

In this lab, we successfully implemented stack operations using an array. We performed push, pop, and display operations and handled stack overflow and underflow conditions effectively. This implementation helped in understanding the fundamental working of the stack data structure.

Experiment 2

OBJECTIVE: To perform various operations on a stack with Linked List operation.

THEORY:

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The main operations of a stack are:

- Push: Inserts an element onto the stack.
- Pop: Removes and returns the top element of the stack.
- Display: Shows all the elements in the stack.

A linked list-based stack dynamically allocates memory and does not require a predefined size. It consists of nodes where each node contains:

- data: The value stored in the node.
- next: A pointer to the next node.

Algorithm

Push Operation

1. Create a new node.
2. Assign the given value to the node.
3. Set the next pointer of the new node to the current top.
4. Update the top pointer to the new node.
5. Increment the stack size.

Pop Operation

1. Check if the stack is empty.
2. If not, store the top node's data.
3. Update the top pointer to the next node.
4. Free memory allocated to the previous top node.
5. Decrement the stack size.
6. Return the popped value.

Display Operation

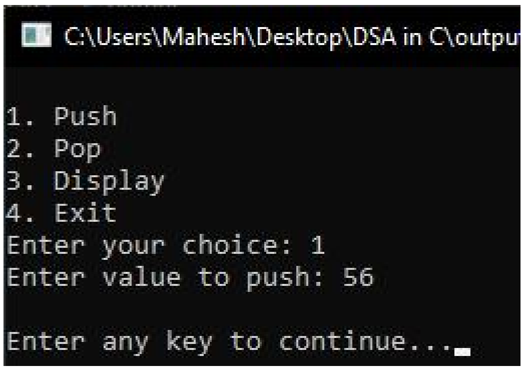
1. Traverse the stack from the top to the bottom.
2. Print each node's data.

PROGRAMS

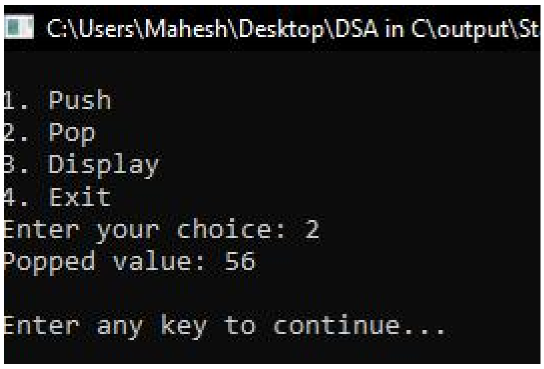
```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node Node;

Node *createNode(int data)
{
    Node *newNode = (Node
*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct LinkedListStack
{
    Node *top;
    int size;
};
typedef struct LinkedListStack LinkedListStack;

void push(LinkedListStack *stack, int data)
{
    Node *newNode = createNode(data);
    newNode->next = stack->top;
    stack->top = newNode;
    stack->size++;
}
int pop(LinkedListStack *stack)
{
    if (stack->top == NULL)
    {
        printf("Stack Underflow\n");
        return -1; // Indicate stack is empty
    }
    Node *temp = stack->top;
    int poppedValue = temp->data;
    stack->top = stack->top->next;
    free(temp);
    stack->size--;
    return poppedValue;
}

void display(LinkedListStack *stack)
```

```
{
    if (stack->top == NULL)
    {
        printf("Stack is empty\n");
        return;
    }
    Node *temp = stack->top;
    printf("Stack elements are: ");
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main()
{
    LinkedListStack stack;
    stack.size = 0;
    stack.top = NULL;

    int choice, value;
    int poppedValue;
    do
    {
        system("cls");
        printf("\n1. Push\n2. Pop\n3. Display\n4.
Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(&stack, value);
            break;
        case 2:
            poppedValue = pop(&stack); // Use
the declared variable
            if (poppedValue != -1)
                printf("Popped value: %d\n",
poppedValue);
            break;
        case 3:
            display(&stack);
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice\n");
        }
        printf("\nEnter any key to continue...");
        getch();
    } while (choice != 4);
    return 0;
}
```

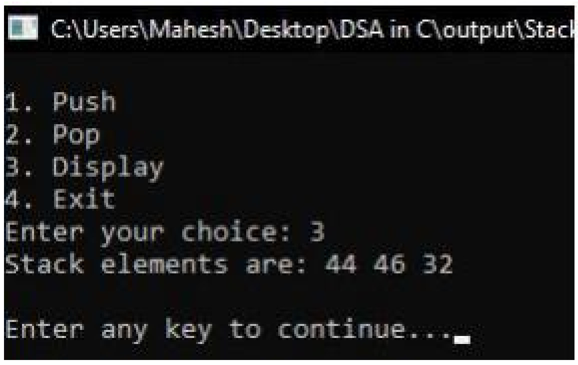
Output:



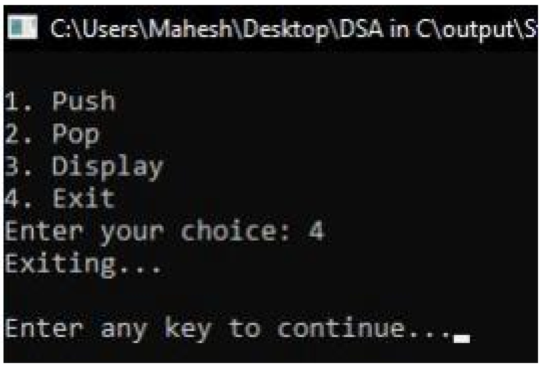
```
C:\Users\Mahesh\Desktop\DSA in C\outpu

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 56

Enter any key to continue...
```



```
C:\Users\Mahesh\Desktop\DSA in C\output\St

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped value: 56

Enter any key to continue...
```



```
C:\Users\Mahesh\Desktop\DSA in C\output\Stac

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements are: 44 46 32

Enter any key to continue...
```



```
C:\Users\Mahesh\Desktop\DSA in C\output\S

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting...

Enter any key to continue...
```

CONCLUSION:

In this lab, we successfully implemented a stack using a linked list. We performed operations such as push, pop, and display, demonstrating the Last In, First Out (LIFO) principle. This approach allows dynamic memory allocation, eliminating the limitations of a fixed-size stack array.

Experiment 3

OBJECTIVE: To convert an infix expression into a postfix expression using stack.

THEORY:

An infix expression is a mathematical notation where operators are placed between operands (e.g., A + B). A postfix expression places operators after the operands (e.g., AB+). Postfix notation eliminates the need for parentheses, making evaluation easier.

To convert an infix expression to a postfix expression, we use a stack to manage operator precedence and parentheses. The main steps involve:

- Scanning the expression from left to right.
- Pushing operators onto a stack while maintaining precedence order.
- Popping operators when encountering lower precedence operators or closing parentheses.
- Appending operands directly to the postfix expression.

Algorithm

Infix to Postfix Conversion

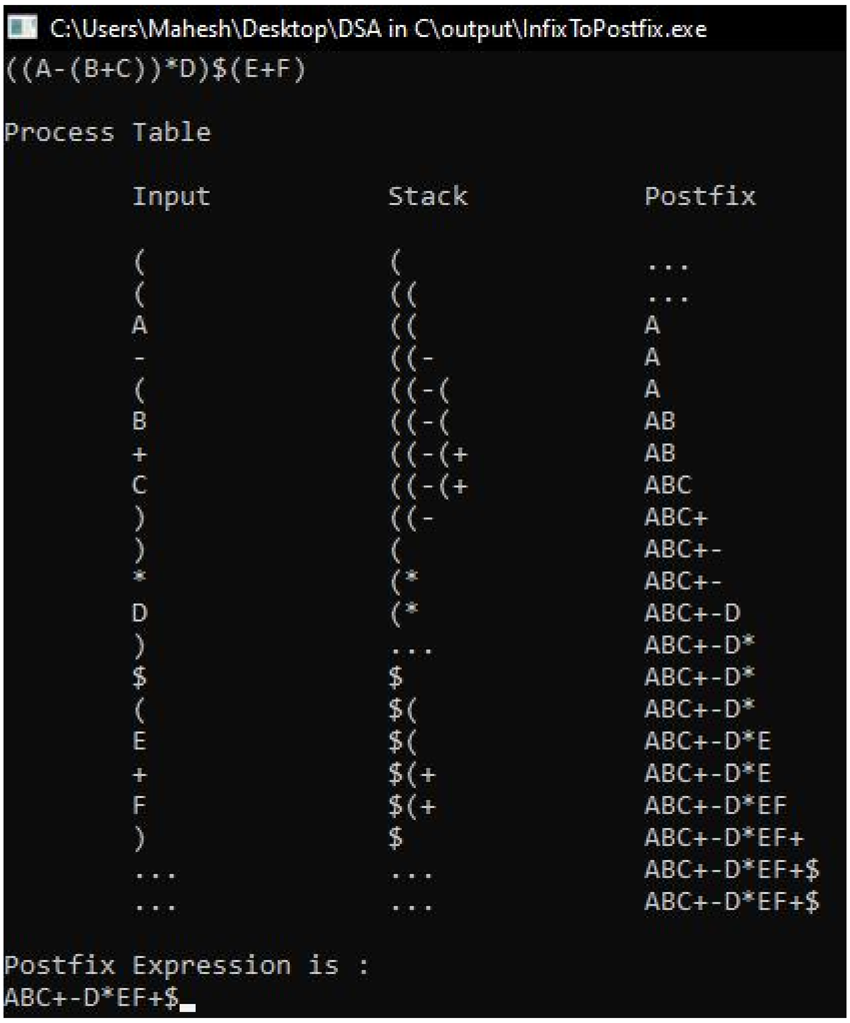
1. Read the infix expression character by character.
2. If the character is an operand (A-Z, a-z), append it to the postfix expression.
3. If the character is an opening parenthesis (, push it onto the stack.
4. If the character is a closing parenthesis), pop from the stack and append to postfix until (is encountered.
5. If the character is an operator:
   - Pop operators from the stack to postfix if they have higher or equal precedence.
   - Push the current operator onto the stack.
6. After scanning, pop any remaining operators from the stack to the postfix expression.

PROGRAMS

```
#include<stdio.h>
#include<conio.h>
#include<stdbool.h>
#define MAX 50

char stack[MAX];
int tos = -1;

bool isempty(){
    return tos == -1;
}
void push(char ch){
    stack[++tos] = ch;
}
char pop(){
    if(!isempty())
        return stack[tos--];
    return ' ';
}
char peek(){
    return stack[tos];
}
```

```
int precedende(char op){
    if(op=='!') return 1; //highest precedence
    if(op=='^' || op=='$') return 2;
    if(op=='*' || op=='/' || op=='%') return 3;
    if(op=='+' || op=='-') return 4;
    else return 5;
}

bool CheckPrecedende(char op1, char op2){
    return precedende(op1) < precedende(op2);
}

int whatCharacterIs(char ch){
    if((ch>=65 && ch<=90) || (ch>=97 &&
ch<=122))
        return 1; // if the character is Letter.

    if(ch=='(') return 2;
    if(ch==')') return 3;
    if(ch=='+' || ch=='-' || ch=='*' || ch=='/' ||
ch=='$' || ch=='^' || ch=='%')
        return 4;

    return 0;
}

void TableRow(char infix[], char postfix[], int i,
int j){
    if(infix[i]=='\0') printf("\n\t...");
    else printf("\n\t%c",infix[i]);

    printf("\t\t");
        if(isempty()) printf("...");
        else
            for(int a=0; a<=tos;a++)
                printf("%c",stack[a]);

    printf("\t\t");
    if(j==0) printf("...");
    else
        for(int a=0; a<j; a++)
            printf("%c",postfix[a]);
}

int main(){
    char infix[MAX],postfix[MAX];
    int i=0;
    char ch;

    printf("Enter the Infix expression:\n");
    do{
        ch = getch();
        printf("%c",ch);

        if(ch != 13)
            infix[i++] = ch;

    }while(ch != 13);
    infix[i] = '\0';

    //print table
    printf("\n\nProcess Table\n");
    printf("\n\tInput\t\tStack\t\tPostfix\n");

    int j = 0;
    for(i=0; infix[i] != '\0'; i++){
        switch(whatCharacterIs(infix[i])){
            case 1:
                postfix[j++] = infix[i];
                break;
            case 2:
                push(infix[i]);
                break;
            case 3:
                while(peek()!='('
&& !isempty()){
                    postfix[j++] = pop();
                }
                if(!isempty()) pop();
                break;
            case 4:
                if(isempty() || peek()=='(')
                    push(infix[i]);
                else{

                    if(CheckPrecedende(peek(),i
nfix[i])){
                        postfix[j++] = pop();
                        push(infix[i]);
                    }else{
                        push(infix[i]);
                    }
                }

                break;
        }

        TableRow(infix,postfix,i,j);
    }
    while(!isempty()){
        if(peek()!='(')
            postfix[j++] = pop();
        else
            pop();
        TableRow(infix,postfix,i,j);
    }
    TableRow(infix,postfix,i,j);

    postfix[j] = '\0';

    printf("\n\nPostfix Expression is :\n");
    for(i=0; postfix[i]!='\0'; i++)
        printf("%c",postfix[i]);

    getch();
    return 0;
}
```

Output:



```
C:\Users\Mahesh\Desktop\DSA in C\output\InfixToPostfix.exe
((A-(B+C))*D)$(E+F)

Process Table

        Input           Stack           Postfix

        (               (               ...
        (               ((              ...
        A               ((              A
        -               ((-             A
        (               ((-(            A
        B               ((-(            AB
        +               ((-(+           AB
        C               ((-(+           ABC
        )               ((-             ABC+
        )               (               ABC+-
        *               (*              ABC+-
        D               (*              ABC+-D
        )               ...             ABC+-D*
        $               $               ABC+-D*
        (               $(              ABC+-D*
        E               $(              ABC+-D*E
        +               $(+             ABC+-D*E
        F               $(+             ABC+-D*EF
        )               $               ABC+-D*EF+
        ...             ...             ABC+-D*EF+$
        ...             ...             ABC+-D*EF+$

Postfix Expression is :
ABC+-D*EF+$
```

CONCLUSION:

In this lab, we successfully converted an infix expression to a postfix expression using a stack. This conversion helps eliminate parentheses and simplifies expression evaluation in computer systems.

Experiment 4

OBJECTIVE: To evaluate a postfix expression using a stack.

THEORY:

A postfix expression is evaluated using a stack by processing the expression from left to right:

1. If the character is an operand, push it onto the stack.
2. If the character is an operator, pop the top two elements, perform the operation, and push the result back onto the stack.
3. The final result is the only value left in the stack.

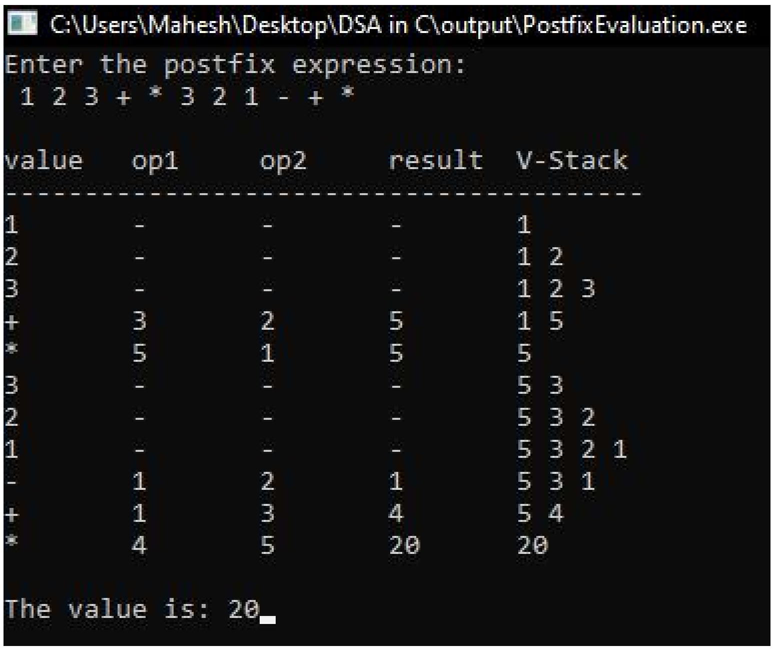
<u>Algorithm</u>

1. Read the postfix expression character by character.
2. If the character is an operand, push it onto the stack.
3. If the character is an operator:
   - Pop the top two elements.
   - Perform the operation.
   - Push the result back onto the stack.
4. The final result is the only value left in the stack.

PROGRAMS

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define MAX 50

int main()
{
    int vstack[MAX];
    int tos = -1;

    char postfix[MAX];
    int i = 0;

    char ch;
    printf("Enter the postfix expression:\n");
    while (1)
    {
        ch = getch();
        printf(" %c", ch);
        if (ch == 13)
            break;

        postfix[i++] = ch;
    }
    postfix[i] = '\0';
    printf("\n");

    printf("\nvalue\top1\top2\tresult\tV-
Stack\n");
printf("---------------------------------------\n");

int op1, op2, result;
for (i = 0; postfix[i] != '\0'; i++)
{
    op1 = -1;
    if (postfix[i] >= '0' && postfix[i] <= '9')
        vstack[++tos] = postfix[i] - '0';
    else
    {
        op1 = vstack[tos--];
        op2 = vstack[tos--];
        switch (postfix[i])
        {
        case '+':
            result = op2 + op1;
            break;
        case '-':
            result = op2 - op1;
            break;
        case '*':
            result = op2 * op1;
            break;
        case '/':
            result = op2 / op1;
            break;
        case '^':
            result = pow(op2, op1);
            break;
            }
            vstack[++tos] = result;
        }

        if (op1 != -1)
            printf("%c\t%d\t%d\t%d\t",
postfix[i], op1, op2, result);
            else
            printf("%c\t-\t-\t-\t", postfix[i]);
        for (int j = 0; j <= tos; j++)
        {
            printf("%d ", vstack[j]);
        }
        printf("\n");
    }

    printf("\nThe value is: %d", vstack[tos]);
    getch();
}
```

Output:



```
C:\Users\Mahesh\Desktop\DSA in C\output\PostfixEvaluation.exe
Enter the postfix expression:
 1 2 3 + * 3 2 1 - + *

value   op1     op2     result  V-Stack
----------------------------------------
1       -       -       -       1
2       -       -       -       1 2
3       -       -       -       1 2 3
+       3       2       5       1 5
*       5       1       5       5
3       -       -       -       5 3
2       -       -       -       5 3 2
1       -       -       -       5 3 2 1
-       1       2       1       5 3 1
+       1       3       4       5 4
*       4       5       20      20

The value is: 20
```

CONCLUSION:

In this lab, we successfully evaluated a postfix expression using a stack. These techniques are fundamental in expression evaluation in computer systems, providing efficiency and clarity in computation.