**Experiment no 3:** Evaluation of postfix Expression using stack ADT

**Aim:** Implementation of Evaluation of Postfix Expression using stack ADT

**Objective:**

1. Understand the use of stack
2. Understand importing an ADT in an application program
3. Understand the instantiation of stack ADT in an application Program
4. Understand how the member function of an ADT are accessed in an application program

**Theory:** Postfix expression evaluation using a stack ADT (Abstract Data Type) involves utilizing a stack data structure to process and calculate the value of the expression. The stack helps manage the order of operations and operands, ensuring correct evaluation. Here's a detailed explanation of the theory behind postfix expression evaluation using a stack ADT

**Algorithm:** Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from

stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands

are consumed

Step 6 − pop the stack and perform operation

Postfix Evaluation Algorithm

1) Add ) to postfix expression.

2) Read postfix expression Left to Right until ) encountered

3) If operand is encountered, push it onto Stack

[End If]

4) If operator is encountered, Pop two elements

i) A -> Top element

ii) B-> Next to Top element

iii) Evaluate B operator A

push B operator A onto Stack

5) Set result = pop

6) END

**Code :**

#include<stdio.h>

int stack[20];

int top = -1;

void push(int x)

{

stack[++top] = x;

}

int pop()

{

return stack[top--];

}

int main()

{

char exp[20];

char \*e;

int n1,n2,n3,num;

printf("Enter the expression :: ");

scanf("%s",exp);

e = exp;

while(\*e != '\0')

{

if(isdigit(\*e))

{

num = \*e - 48;

push(num);

}

else

{

n1 = pop();

n2 = pop();

switch(\*e)

{

case '+':

{

n3 = n1 + n2;

break;

}

case '-':

{

n3 = n2 - n1;

break;

}

case '\*':

{

n3 = n1 \* n2;

break;

}

case '/':

{

n3 = n2 / n1;

break;

}

}

push(n3);

}

e++;

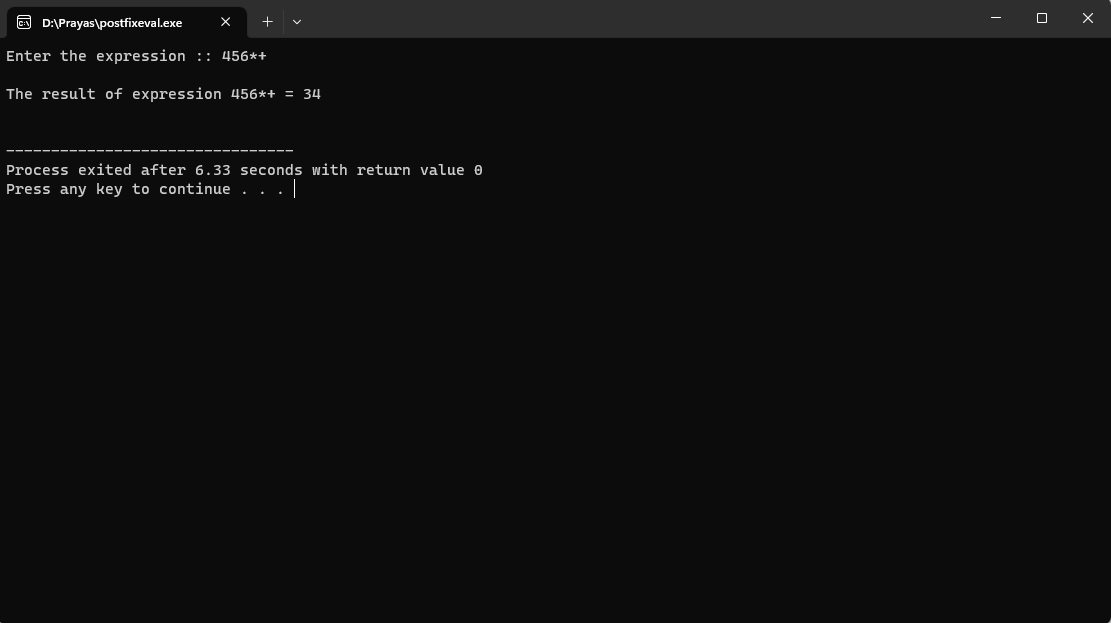
}

printf("\nThe result of expression %s = %d\n\n",exp,pop());

return 0;

}

**Output:**



**Conclusion :** The process involves systematically scanning the expression, pushing operands onto the stack and performing calculations with operators. The stack's Last-In-First-Out (LIFO) behavior ensures that the correct order of operations is maintained, even in complex expressions.

This methodology eliminates the need for parentheses and operator precedence rules, simplifying the evaluation process. The algorithm's time complexity of O(n), where n is the number of tokens in the expression, demonstrates its efficiency, making it a practical choice for real-time calculations.