Data Structures : Algorithms and Applications(Lab)

EXPERIMENT No:2

Aim: Conversion of Infix Expression to Postfix Expression Using Stack

Theory:
Infix notation is the conventional way we write expressions, such as A + B. However, for computational purposes, especially in parsing expressions, postfix notation (or Reverse Polish Notation) is more efficient. In postfix notation, the operators follow the operands, such as A B +.

Benefits to use:
Eliminates Ambiguity: No need for parentheses to denote operation precedence.
Simplifies Computation: Easier for computers to evaluate without needing to backtrack.

Conversion Algorithm:
Initialize:
Use a stack to temporarily hold operators.
Prepare an empty output list for the result.
Process the Expression:
Operands: Directly add to the output.
Operators: Push onto the stack but first pop operators with higher or equal precedence from the stack to the output.
Parentheses:
Push opening parentheses onto the stack.
Pop from the stack to the output until the matching opening parenthesis for a closing parenthesis.
Finalize:
Pop any remaining operators from the stack to the output.

Precedence Rules:
Operators have a precedence that determines their order of evaluation:
Highest: ^ (Exponentiation)
Middle: *, / (Multiplication and Division)
Lowest: +, - (Addition and Subtraction)

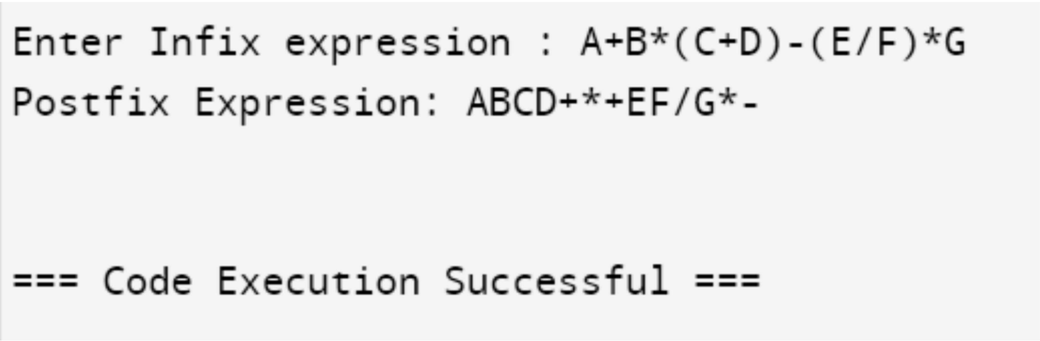
Example:
Convert the infix expression A + (B * C) to postfix:
Input: A + (B * C)
Process:
A → Output: A
+ → Push to stack.
(→ Push to stack.
B → Output: A B
* → Push to stack (inside parentheses).
C → Output: A B C
) → Pop from stack until (. Output: A B C *
Pop remaining operators. Output: A B C * +
Result: A B C * +

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define SIZE 100

char stack[SIZE];
int top = -1;

void push(char item) {
    if (top >= SIZE - 1) {
        printf("\nStack Overflow.");
    } else {
        top = top + 1;
        stack[top] = item;
    }
}

char pop() {
    char item;
    if (top < 0) {
        printf("Stack underflow: invalid infix expression");
        exit(1);
    } else {
        item = stack[top];
        top = top - 1;
        return item;
    }
}

int is_operator(char symbol) {
    return (symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' ||
symbol == '-');
}

int precedence(char symbol) {
    if (symbol == '^') {
        return 3;
    } else if (symbol == '*' || symbol == '/') {
        return 2;
    } else if (symbol == '+' || symbol == '-') {
        return 1;
    } else {
        return 0;
    }
}
```

```
void infixToPostfix(char infix_exp[], char postfix_exp[]) {
    int i = 0, j = 0;
    char item, x;

    strcat(infix_exp, ")");
    push('(');

    item = infix_exp[i];

    while (item != '\0') {
        if (item == '(') {
            push(item);
        } else if (isdigit(item) || isalpha(item)) {
            postfix_exp[j++] = item;
        } else if (is_operator(item)) {
            x = pop();
            while (is_operator(x) && precedence(x) >= precedence(item)) {
                postfix_exp[j++] = x;
                x = pop();
            }
            push(x);
            push(item);
        } else if (item == ')') {
            x = pop();
            while (x != '(') {
                postfix_exp[j++] = x;
                x = pop();
            }
        } else {
            printf("\nInvalid infix Expression.\n");
            exit(1);
        }
        i++;
        item = infix_exp[i];
    }
    if (top > 0) {
        printf("\nInvalid infix Expression.\n");
        exit(1);
    }

    postfix_exp[j] = '\0';
}

int main() {
    char infix[SIZE], postfix[SIZE];

    printf("\nEnter Infix expression : ");
    fgets(infix, SIZE, stdin);
    infix[strcspn(infix, "\n")] = '\0';

    infixToPostfix(infix, postfix);
    printf("Postfix Expression: ");
    puts(postfix);

    return 0;
}
```

OUTPUT :



```
Enter Infix expression : A+B*(C+D)-(E/F)*G
Postfix Expression: ABCD+*+EF/G*-



=== Code Execution Successful ===
```

CONCLUSION:

Stacks efficiently manage operators in infix-to-postfix conversion, optimizing expression evaluation and supporting various algorithms. This technique is essential for enhancing computational efficiency and effectiveness.