SOMAIYA
VIDYAVIHAR
K J Somaiya Institute of Technology
An Autonomous Institute Permanently Affiliated to the University of Mumbai
DEPARTMENT OF INFORMATION TECHNOLOGY

Course: Data Structures Lab (ITL405) B.Tech. (Information Technology) – Semester III Academic Year: 2024-25(Odd Semester)

Experiment No: 3

Aim: Implementation of Infix to Postfix Expression for real-world applications**.**

Lab Objective:

1. To introduce the concepts of data structures and analysis procedure.
2. To conceptualize linear data structures and its implementation for various real-world applications.

Theory:

1. Introduction

Infix and postfix expressions are two different notations for writing mathematical expressions. Infix notation, which is commonly used, places operators between operands (e.g., A + B). Postfix notation (or Reverse Polish Notation), however, places operators after their operands (e.g., A B +). Postfix expressions have the advantage of eliminating the need for parentheses and the complexities of operator precedence and associativity, making them ideal for computer processing.

2. Infix Notation

In infix notation, operators are placed between operands, and operator precedence dictates the order of evaluation. For example, in the expression A + B * C, multiplication is performed before addition due to its higher precedence. Parentheses can be used to explicitly dictate the order of operations (e.g., (A +B) * C).

3. Postfix Notation

Postfix notation does not require parentheses to dictate the order of operations, as the position of operators relative to their operands implicitly defines the order. In postfix notation, the expression A + B* C is written as A B C * +. The operands are pushed onto a stack, and when an operator is encountered, it operates on the top elements of the stack, which simplifies evaluation.

4. Conversion Algorithm

The Shunting Yard algorithm, developed by Edsger Dijkstra, is a widely used method for converting infix expressions to postfix notation. The algorithm uses a stack to temporarily hold operators and ensures the correct order of operations:

- Initialization: Create an empty stack for operators and an empty list for the output.
- Processing: Read the infix expression from left to right:
  - Operands: Directly add operands (numbers, variables) to the output list.
  - Left Parenthesis: Push onto the stack.
  - Right Parenthesis: Pop from the stack to the output list until a left parenthesis is encountered.
  - Operators: Pop operators from the stack to the output list while they have higher or equal precedence compared to the current operator. Then push the current operator onto the stack.
- End of Expression: Pop all remaining operators from the stack to the output list.

5. Application in Real-World Scenarios

Converting infix expressions to postfix notation is crucial in various applications:

- Compiler Design: Postfix notation simplifies parsing and code generation in compilers.
- Calculator Systems: Postfix expressions facilitate efficient evaluation in calculators and expression evaluators.
- Mathematical Software: Many mathematical libraries and software tools use postfix notation to optimize the execution of complex expressions.

Code:

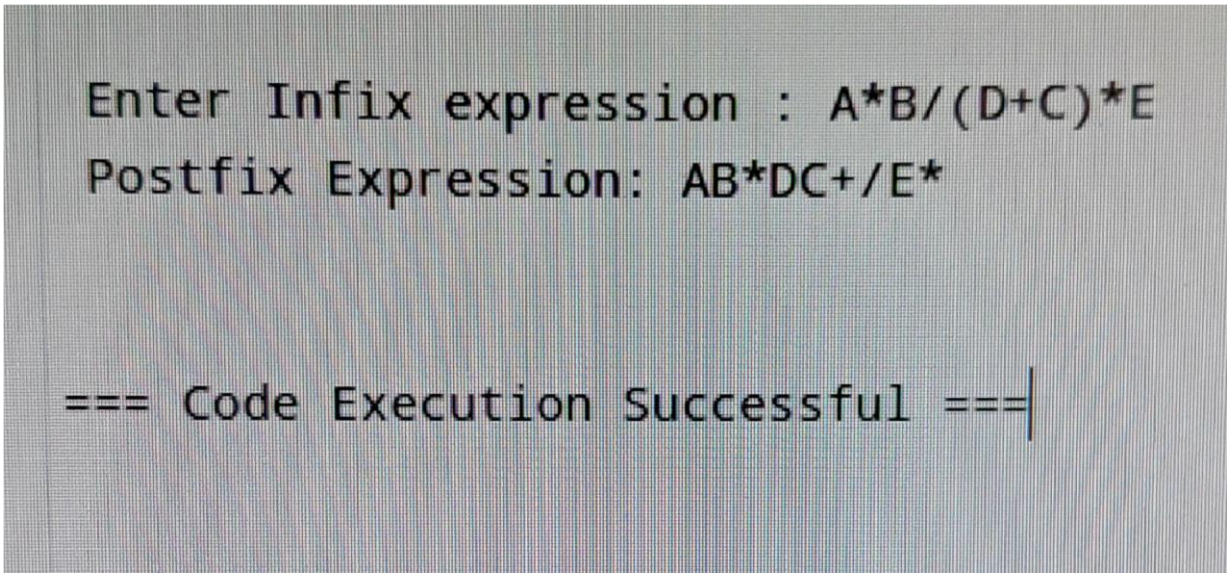
```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define SIZE 100

char stack [SIZE];

int top = -1;

/* === define push operation === */

void push (char item){

   if (top >= SIZE - 1)
   {
      printf("\n Stack Overflow.");
   }
   else
   {
      top = top + 1;
      stack[top] = item;
   }
}
/* === define pop operation === */

char pop()

{
   char item;
   if (top < 0)
   {
      printf("stack underflow: invalid infix expression");

      get char();

      /* underflow may occur for invalid expression */
      /* where ( and ) are not matched */
      exit(1);
   }
   else
   {
```

```
        item = stack[top];

        top = top - 1;

        return (item);
    }
}
/* === define function that is used to determine whether any symbol is operator or

not this function returns 1 if symbol is operator else return 0 === */

int is operator (char symbol){

    if (symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || symbol == '-')
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
/* === define function that is used to assign precedence to operator. Here ^ denotes
exponent operator. In this function we assume that higher integer value means higher
precedence === */

 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, j;
char item;
```

```
    char x;

     push('(');

/* push '(' onto stack*/strcat(infix_exp,")");/*

add ')' to infix expression */

i = 0; j = 0;

    item = infix_exp[i];

    while (item != '\0')

    {
        if (item == '(')
        {
            push(item);
        }
        else if (isdigit(item) || isalpha(item))
        {
            postfix_exp[j] = item;          /* add operand symbol to postfix expr */
            j++;
        }
        else if (is_operator(item) == 1)    /* means symbol is operator */
        {
            x = pop(); while (is_operator(x) == 1 && precedence(x) >=

             precedence(item))

            {
                postfix_exp[j] = x;         /* so pop all higher precedence operators and */
                j++;

                                            /* add them to postfix expression */

                x = pop();
            }
           push(x);

                                             /* push current operator symbol onto stack */
            push(item);
        }

                                             /* if current symbol is ')' then */
        else if (item == ')')
        {
```

```
        x = pop();                  /* pop and keep popping until */

        while (x != '(')            /* '(' encountered */
        {
           postfix_exp[j] = x;

           j++;

           x = pop();
        }
      }
      else
      {  /* if current symbol is neither operand, nor '(', nor ')', nor operator */

         printf("\nInvalid infix Expression.\n"); getchar(); exit(1);

      } i++; item =

      infix_exp[i];

   }
   if (top > 0)
   { printf("\nInvalid infix Expression.\n");

      getchar(); exit(1);

   }
   postfix_exp[j] = '\0'; /* add sentinel else puts() function */
   /* will print entire postfix[] array up to SIZE */
}
/* === main function begins === */

int main()

{
   char infix[SIZE], postfix[SIZE]; printf("\n Enter Infix expression :

   "); fgets(infix, SIZE, stdin); infix[strcspn(infix, "\n")] = 0; //

   remove trailing newline character InfixToPostfix(infix, postfix);

   printf(" Postfix Expression: "); puts(postfix); return 0;

}
```

Output:



Conclusion:

In conclusion, converting infix expressions to postfix notation streamlines mathematical computation by eliminating the need for parentheses and operator precedence rules. Implementing the Shunting Yard algorithm efficiently transforms expressions into a format that simplifies evaluation in computational systems, making it vital for applications like compilers and calculators. This process not only optimizes expression handling but also underscores the practical importance of effective expression parsing in real-world scenarios.

Lab Outcome:

Apply the concepts of Stack for real-world applications.

Submission Details: -

Name of Student: Awani Goyal

Roll No.: 31

Date of Performance: 8/08/2024