DS Lab

Conversion of expression from one form to another

**To Design and implement an algorithm for conversion of an expression from one form to another. Demonstrate its working with suitable inputs.**

**Theory**

When we write mathematical expression in a program, we use infix expression. These expressions will be converted into equivalent machine instructions by compiler using stacks. Using stacks we can efficiently convert the expressions from infix to postfix, infix to prefix, postfix to prefix and postfix to infix.

**Infix expression**: In an expression if an operator is in between two operands, the expression is called an infix expression.Ex: a+b

**Postfix expression:** In an expression, if an operator follows the two operands (i.e. operator comes after the two operands), the expression is called postfix expression. Ex: ab+

**Prefix expression:** In an expression, if an operator precedes two operands (i.e operator comes before the two operands), the expression is called prefix expression. Ex: +ab

1. Infix to Postfix

Pseudocode

Scan infix from left to right

If char is operand

Print operand

Else if char is operator

If stack is empty

Push(operator)

Else

If operator == ‘(’

Push(operator)

Else if precedence(operator)>stack[top]

Push(operator)

Else if operator=’)’

While(stack[top]!=’(’)

Pop()

Pop()

Else

While(precedence(operator)<=stack[top])

Pop and print stack[top]

Push(operator)

Pop and print stack

**Code**

#include <stdio.h>

#include <string.h>

char stack[25];  //initialize stack

int top = -1;  //stack is empty

void push(char item)   //push function

{

        stack[++top]=item;

}

char pop()   //pop function

{

        return stack[top--];

}

int precedence(char sym)   //check precedence of operator

{

        switch(sym)

        {

                case '+':

                case '-':

                        return 2;

                        break;

                case '\*':

                case '/':

                        return 3;

                        break;

                case '^':

                        return 4;

                        break;

                case '(':

                case ')':

                case '#':

                        return 1;

                        break;

        }

}

int isOperator(char sym)  //to check if character is operator

{

        switch(sym)

        {

                case '+':

                case '-':

                case '\*':

                case '/':

                case '^':

                case '(':

                case ')':

                        return 1;

                        break;

                default:

                        return 0;

        }

}

void convert(char infix[], char postfix[])  //convert infix to postfix

{

        int i, j = 0;

        char symbol;

        stack[++top]='#';

        for(i=0; i<strlen(infix); i++)  //scanning infix from left to right

        {

                symbol = infix[i];  //current symbol

                if(isOperator(symbol)==0)  //if symbol is not an operator print it as it is, i.e print operand as it is

                {

                        postfix[j]=symbol;

                        j++;

                }

                else  //if symbol is an operator

                {

                        if(symbol=='(')   //if open brackets are encountered push it to stack

                        {

                                push(symbol);

                        }

                        else

                        {

                                if(symbol == ')')  //if closing brackets are encountered pop stack and print till open brackets

                                                   //are found

                                {

                                        while(stack[top]!='(')

                                        {

                                                postfix[j]=pop();

                                                j++;

                                        }

                                        pop();  //pop open brackets from stack

                                }

                                else

                                {

                                        if(precedence(symbol)>precedence(stack[top]))  //if precedence of symbol greater than

                                                                                       //stack top

                                                push(symbol);

                                        else

                                        {

                                                while(precedence(symbol)<=precedence(stack[top]))  //while precedence of symbol

                                                                                                   //is lower/equal to stack top

                                                {

                                                        postfix[j]=pop();

                                                        j++;

                                                }

                                                push(symbol);  //push symbol when done popping

                                        }

                                }

                        }

                }

        }

        while(stack[top]!='#')  //pop and print all operators left in stack

        {

                postfix[j]=pop();

                j++;

        }

        postfix[j]='\0';

}

void main()

{

        char infix[25], postfix[25];

        printf("Enter infix expression: ");

        scanf("%s", infix);

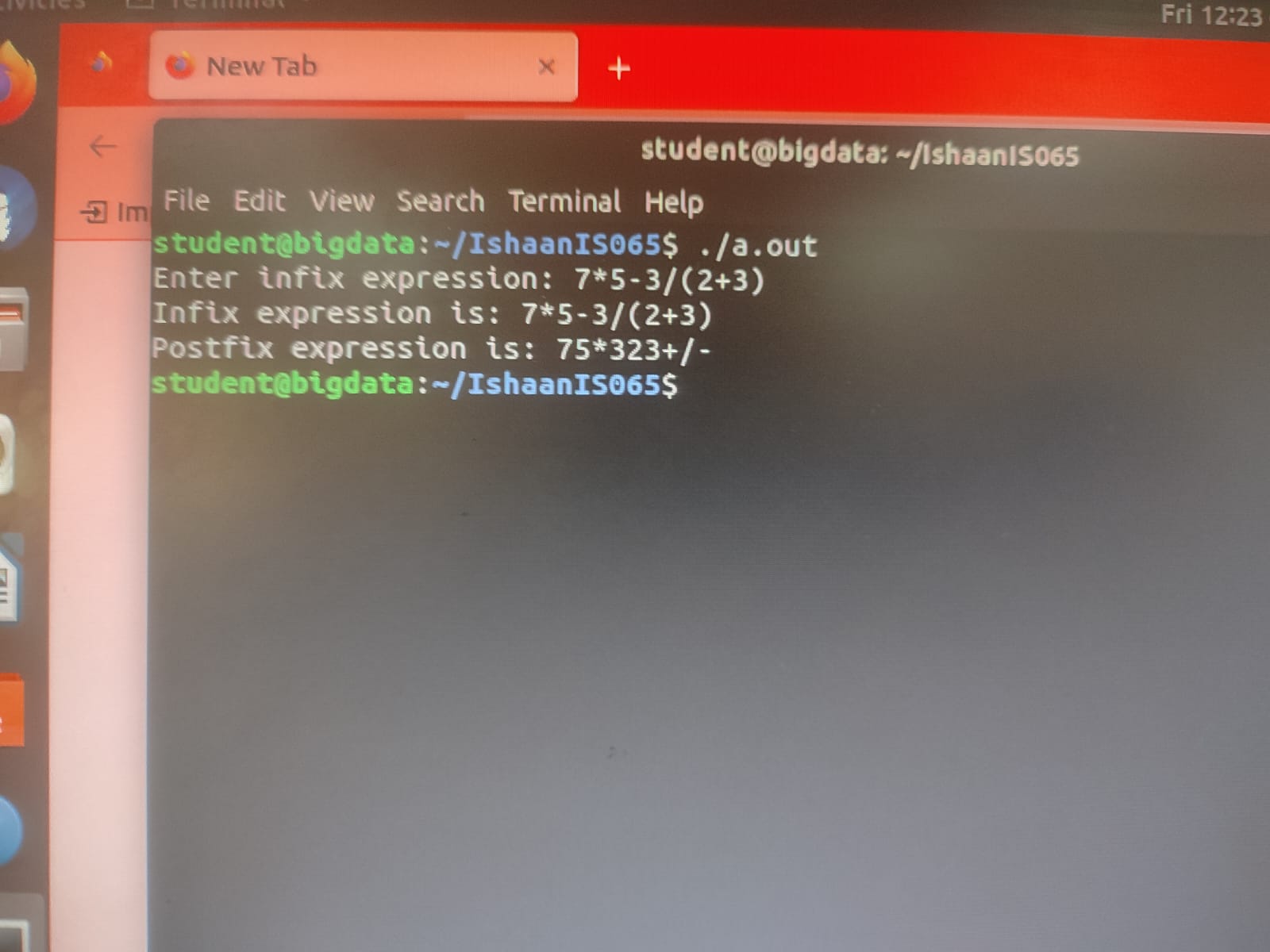
        convert(infix, postfix);

        printf("Infix expression is: %s\n", infix);

        printf("Postfix expression is: %s\n", postfix);

}

**Output**

****

|  |  |  |
| --- | --- | --- |
| Expression | Stack | Postfix |
| 7 |  | 7 |
| \* | \* | 7 |
| 5 | \* | 75 |
| - | - | 75\* |
| 3 | - | 75\*3 |
| / | - / | 75\*3 |
| ( | -/( | 75\*3 |
| 2 | -/( | 73\*32 |
| + | -/(+ | 75\*32 |
| 3 | -/(+ | 75\*323 |
| ) |  | 75\*323+/- | |

**Tracing**

1. Infix to Prefix

Pseudocode

Infix=strrev(infix)

Scan reverse infix from left to right

If char==’(’

Replace with ‘)’

Else if char==’)’

Replace with ‘(’

Postfix=intopost(infix)

Prefix=strrev(Postfix)

Printf(Prefix)

**Code**

#include<stdio.h>

#include<string.h>

#include<limits.h>

#include<stdlib.h>

# define MAX 100

int top = -1;

char stack[MAX];

int isFull()    // check if stack is full

{

    return top == MAX - 1;

}

int isEmpty()   // check if stack is empty

{

    return top == -1;

}

void push(char item)   //push function for stack

{

    if (isFull())

        return;

    top++;

    stack[top] = item;

}

int pop()    //pop function returning the top value of stack after deleting it

{

    if (isEmpty())

        return INT\_MIN;

    return stack[top--];  //decrement top value

}

int peek()   //returns top value of stack without removing it

{

    if (isEmpty())

        return INT\_MIN;

    return stack[top];

}

int checkIfOperand(char ch)    //function to check if character is operand

{

    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

int precedence(char ch)    //to check precedence of operators

{

    switch (ch)

    {

    case '+':

    case '-':

        return 1;

    case '\*':

    case '/':

        return 2;

    case '^':

        return 3;

    }

    return -1;

}

int getPostfix(char\* expression)   //convert infix to postfix

{

    int i, j;

    for (i = 0, j = -1; expression[i]; ++i)

    {

        if (checkIfOperand(expression[i]))    //check if character is operand

            expression[++j] = expression[i];  //adding to output

        else if (expression[i] == '(')     //check for opening brackets

            push(expression[i]);           //push to stacks

        else if (expression[i] == ')')     //check for closing brackets. pop the stack and add to output

                                           //until closing brackets are found

        {

            while (!isEmpty(stack) && peek(stack) != '(')

                expression[++j] = pop(stack);

            if (!isEmpty(stack) && peek(stack) != '(')

                return -1; // invalid expression

            else

                pop(stack); //pop closing brackets

        }

        else // if character is an opertor

        {

            while (!isEmpty(stack) && precedence(expression[i]) <= precedence(peek(stack)))  //if precedence is smaller or equal

                expression[++j] = pop(stack);                                                //to top of stack

            push(expression[i]);

        }

    }

    // Once all inital expression characters are traversed

    // adding all left elements from stack to exp

    while (!isEmpty(stack))

        expression[++j] = pop(stack);

    expression[++j] = '\0';

}

void reverse(char \*exp)   //to reverse a string

{

    int size = strlen(exp);

    int j = size, i=0;

    char temp[size];

    temp[j--]='\0';

    while(exp[i]!='\0')

    {

        temp[j] = exp[i];

        j--;

        i++;

    }

    strcpy(exp,temp);

}

void brackets(char\* exp)    //to change brackets after string reversal

{

    int i = 0;

    while(exp[i]!='\0')

    {

        if(exp[i]=='(')

            exp[i]=')';

        else if(exp[i]==')')

            exp[i]='(';

        i++;

    }

}

void InfixtoPrefix(char \*exp)  //to change infix to prefix

{

    int size = strlen(exp);

    // reverse string

    reverse(exp);

    //change brackets

    brackets(exp);

    //get postfix

    getPostfix(exp);

    // reverse string again

    reverse(exp);

}

int main()   //main function

{

    char expression[]="((a/b)+c)-(d+(e-f))";

    printf("The infix is: ");

    printf("%s\n",expression);

    InfixtoPrefix(expression);

    printf("The prefix is: ");

    printf("%s\n",expression);

    return 0;

}

**Ouput**

****

**Tracing**

((a/b)+c)-(d+(e\*f))

Reverse – ))f\*e(+d(-)c+)b/a((

Switch brackets – ((f\*e)+d)-(c+(b/a))

Convert to postfix

|  |  |  |
| --- | --- | --- |
| Expression | Stack | Postfix |
| ( | ( |  |
| ( | (( |  |
| f | (( | f |
| \* | ((\* | f |
| e | ((\* | fe |
| ) | ( | fe\* |
| + | (+ | fe\* |
| d | (+ | fe\*d |
| ) |  | fe\*d+ |
| - | - | fe\*d+ |
| ( | -( | fe\*d+ |
| c | -( | fe\*d+c |
| + | -(+ | fe\*d+c |
| ( | -(+( | fe\*d+c |
| b | -(+( | fe\*d+cb |
| / | -(+(/ | fe\*d+cb |
| a | -(+(/ | fe\*d+cba |
| ) | -(+ | fe\*d+cba/ |
| ) |  | fe\*d+cba/+- |

Reverse Postfix-

Prefix = -+/abc+d\*ef

1. Prefix to Postfix

Pseudocode

Scan prefix from right to left

If char is operand

Push(operand)

Else if char is operator

Operand2=pop()

Operand1=pop()

Temp=operand1+operand2+operator

Push(temp)

Print(stack)

**Code**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

# define MAX 20

char str[MAX],stack[MAX];

int top=-1;

void push(char c)

{

   stack[++top]=c;

}

char pop()

{

   return stack[top--];

}

void pre\_post()

{

   int n,i,j=0; char c[20];  //c is postfix expression

   char a,b,op;

   printf("Enter the prefix expression\n");

   gets(str);

   n=strlen(str);

   for(i=0;i<MAX;i++)

   stack[i]='\0';  //set stack to null stack

   printf("Postfix expression is:\t");

   for(i=0;i<n;i++)

   {

      if(str[i]=='+'||str[i]=='-'||str[i]=='\*'||str[i]=='/')

      {

         push(str[i]);  //push operator to stack

      }

      else

      { c[j++]=str[i];  //add operand to postfix expression

        while((top!=-1)&&(stack[top]=='@'))

        {

            a=pop(); c[j++]=pop();  //pop'@' and add operator to postfix

        }

        push('@');  //push '@' to stcak

      }

   }

   c[j]='\0';  //add null character at the end of postfix to make it a string

   printf("%s",c); //print postfix

}

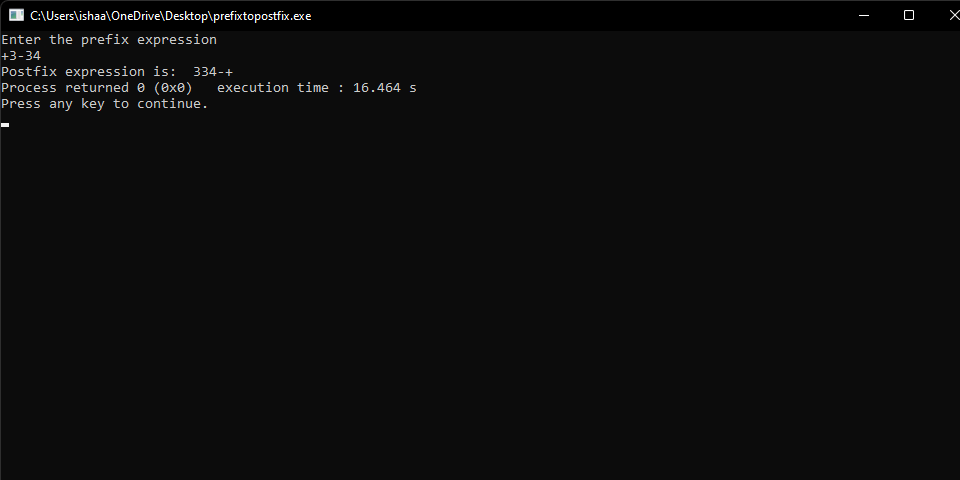
main()

{

    pre\_post();

}

**Output**



1. Postfix to Prefix

Pseudocode

Scan postfix from left to right

If char is operand

Push(operand)

Else if char is operator

Operand1=pop()

Operand2=pop()

Temp=operator+operand1+operand2

Push(temp)

Print(stack)

**Code**

#include<stdio.h>   //include header files

#include<string.h>

#include<stdlib.h>

#define MAX 20   //global declaration of MAX

char str[MAX], stack[MAX];   //initialize stack and str

int top = -1;  //initialize top to -1 i.e stack is empty

void push(char c)  //push funtion for stack

{

  stack[++top] = c;

}

char pop()  //pop function for stack

{

  return stack[top--];

}

int checkIfOperand(char ch)  //check whether charactor is operand or not

{

  return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

int isOperator(char x)  //function to check if character is operator

{

  switch (x) {

  case '+':

  case '-':

  case '/':

  case '\*':

    return 1;

  }

  return 0;

}

void postfixToprfix()  //postfix to prefix conversion function

{

  int n, i, j = 0;

  char c[20]; //c is postfix expression

  char a, b, op;

  printf("Enter the postfix expression\n");

  scanf("%s", str);  //str stores postfix expression

  n = strlen(str);  //n is string length for str

  for (i = 0; i < MAX; i++)

    stack[i] = '\0';  //set all elements of stack to null character

  printf("Prefix expression is:\t");

  for (i = n - 1; i >= 0; i--)

  {

    if (isOperator(str[i]))

    {

      push(str[i]);  //push operator

    } else

    {

      c[j++] = str[i];  //add operand to c

      while ((top != -1) && (stack[top] == '#'))

      {

        a = pop();  //pops # from top of stack

        c[j++] = pop();  //adds the operator from stack to c

      }

      push('#');  //push # to stack

    }

  }

  c[j] = '\0';

  i = 0;

  j = strlen(c) - 1;

  char d[20];  //d is prefix

  while (c[i] != '\0')  //reverses the c string and stores in d string which is prefix

  {

    d[j--] = c[i++];

  }

  printf("%s\n", d);

}

int main()

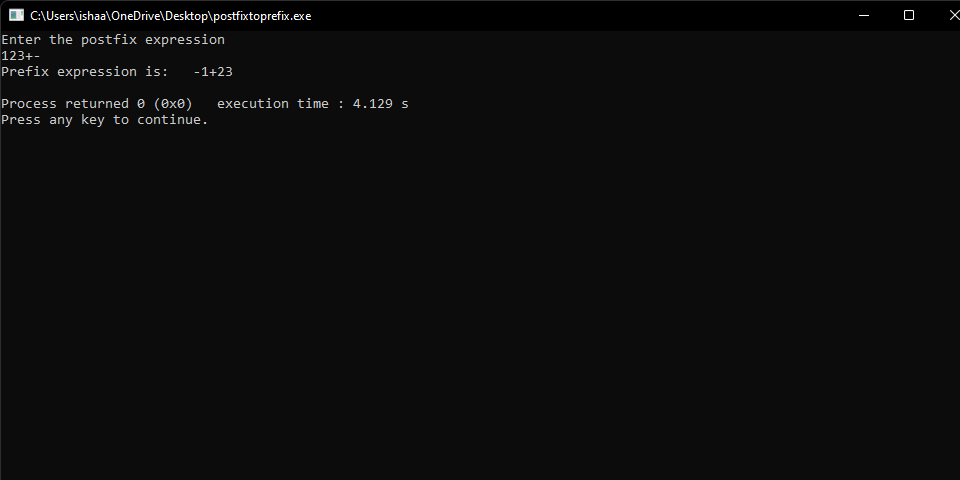
{

  postfixToprfix();

  return 0;

}

**Output**

****