**Exp No 2: Conversion of Infix to postfix expression using stack ADT**

**NAME: KUMUD ZIMAL, SUB: DATA STRUCTURE , DIV : COMPS C -3, ROLL NO : 68**

**Aim**: To convert infix expression to postfix expression using stack ADT

**Objective**:

1) Understand the use of stack

2) Understand how to import 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**:

Infix : An infix expression is a mathematical or logical expression in which operators are placed between operands. For example: 3 + 4 \* 2.

Postfix (also known as Reverse Polish Notation): A postfix expression is a mathematical or logical expression in which operators are placed after their operands. For example: 3 4 2 \* +.

Converting an infix expression to a postfix expression involves rearranging the operators and operands so that the expression can be evaluated using a stack. This process helps to remove the need for parentheses to indicate the order of operations.

**Algorithm:**

1. Initialize an empty stack for operators.

2. Initialize an empty string for the postfix expression.

3. Scan the infix expression from left to right.

4. For each symbol in the infix expression:

a. If the symbol is an operand (numeric value or variable), add it to the postfix string.

b. If the symbol is an operator (+, -, \*, /, etc.):

i. While the stack is not empty and the operator at the top of the stack has equal or higher precedence than the current operator, pop the operator from the stack and add it to the postfix string.

ii. Push the current operator onto the stack.

c. If the symbol is an opening parenthesis '(', push it onto the stack.

d. If the symbol is a closing parenthesis ')':

i. Pop operators from the stack and add them to the postfix string until an opening parenthesis '(' is encountered. Pop and discard the opening parenthesis.

5. After scanning the entire expression, pop any remaining operators from the stack and add them to the postfix string.

6. The postfix expression is the final output.

**CODE:**

#include<stdio.h>

#include<conio.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 is overflow\n");

}

else

{

top=top+1;

stack[top]=item;

}

}

char pop()

{

char item;

if(top<0)

{

printf("stack is underflow\n");

getchar();

exit(1);

}

else

{

item=stack[top];

top=top-1;

return item;

}

}

int is\_operator(char symbol)

{

if(symbol=='^'||symbol=='+'||symbol=='\*'||symbol=='-'||symbol=='/'||symbol=='+')

{

return 1;

}

else

{

return 0;

}

}

int precendence(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('(');

strcat(infix\_exp,")");

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;

j++;

}

else if(is\_operator(item)==1)

{

x=pop();

while(is\_operator(x)==1&&precendence(x)>=precendence(item))

{

postfix\_exp[j]=x;

j++;

x=pop();

}

push(x);

push(item);

}

else if(item==')')

{

x=pop();

while(x!='(')

{

postfix\_exp[j]=x;

j++;

x=pop();

}

}

else

{

printf("invalid infix expression\n");

getchar();

exit(1);

}

i++;

item=infix\_exp[i];

}

if(top>0)

{

printf("invalid infix expression\n");

getchar();

exit(1);

}

postfix\_exp[j]='\0';

}

int main()

{

char infix [size],postfix [size];

clrscr();

printf("ASSUMPTION:the infix expression contain single letter variable & single digit constant only\n");

printf("enter infix expression\n");

gets(infix);

infixtopostfix(infix,postfix);

printf("postfix expression ");

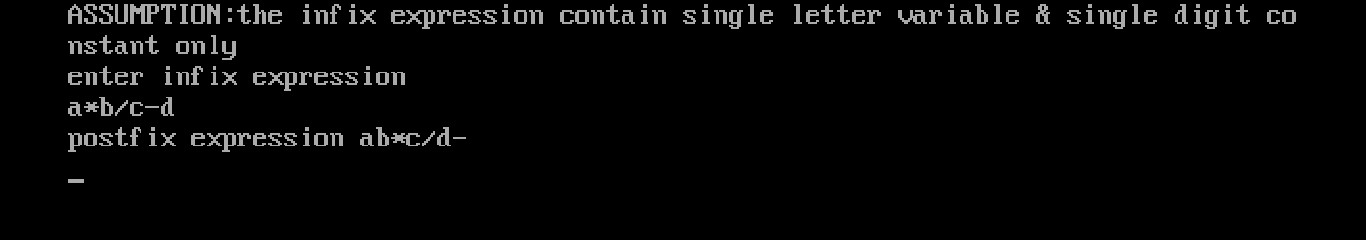
puts(postfix);

getch();

return 0;

}

**OUTPUT:**



**CONCLUSION:**

**In conclusion, this experiment demonstrated the effective use of the stack ADT for converting infix expressions to postfix notation. Through this process, we learned to manage operators, operands, and parentheses while preserving the correct order of operations. The experiment provided practical insights into the implementation of data structures and their role in solving computational challenges. By mastering this conversion technique, we've gained a valuable skill for expression evaluation and algorithm design in various programming scenarios.**