Experiment 2

Name : Vaibhav R Tatkare

Div / Roll No.: Comps 3 (C) / 52

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 expressions are readable and solvable by humans. We can easily distinguish
the order of operators, and also can use the parenthesis to solve that part first
during solving mathematical expressions. The computer cannot differentiate the
operators and parentheses easily, that's why postfix conversion is needed.

To convert infix expression to postfix expression, we will use the stack data
structure. By scanning the infix expression from left to right, when we will get
any operand, simply add them to the postfix form, and for the operator and
parenthesis, add them in the stack maintaining the precedence of them.

Algorithm :

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, put it in the postfix expression.

3. Otherwise, do the following

- If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a '('], then push it in the stack. ['^' operator is right-associative and other operators like '+-' and '/' are left-associative].

- Check especially for a condition when the operator at the top of the stack and the scanned operator both are '^' In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.

- In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.

- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.

- After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is a '(, push it to the stack.

5. If the scanned character is a Y, pop the stack and output it until a '(' is encountered, and discard both the parenthesis.

6. Repeat steps 2-5 until the infix expression is scanned.

7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.

8. Finally, print the postfix expression.

Code :

```c
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
#define size 100
char stack[size];
int top=-1;
void push(char item)
{
if(top>=size-1)
{
printf("\n stack verification");
}
else
{
top=top+1;
stack[top]=item;
}
}
char pop()
{
char item;
if(top<0)
{
```

```c
printf("stack is underflow");

getchar();

exit(1);

}

else

{

item=stack[top];

top--;

return(item);

}

return 0;

}

int is_operator(char symbol)

{

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

return 1;

}

else

{

return 0;

}

}

int precedence(char symbol)

{

if(symbol=='^')

{
```

```c
    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')

{
```

```c
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 && precedence(x)>=precedence(item))  {

postfix_exp[j]=x;

j++;

x=pop();

}

push(x);

push(item);

}

else if(item==')')

{

x=pop();

while(x!='(')

{

postfix_exp[j]=x;
```

```c
j++;

x=pop();

}

}

else

{

printf("\n invalid infix expression\n");

getchar();

exit(1);

}

i++;

item=infix_exp[i];

}

if(top>0)

{

printf("\n invalid infix expression\n");

getchar();

exit(1);

}

postfix_exp[j]='\0';

}

int main()

{

char infix[size],postfix[size];

printf("assumption the infix expression contain single letter variable and  single digit constact omly\n");

printf("\n enter infix expression:");
```
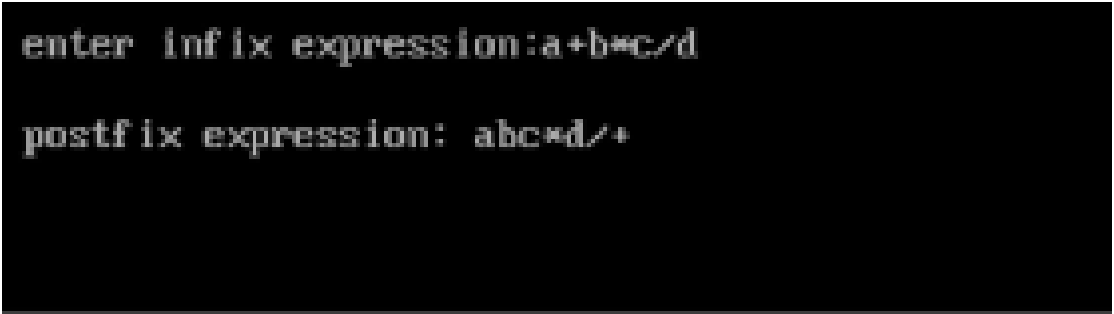
```
gets(infix);

InfixToPostfix(infix,postfix);

printf("\n postfix expression: ");

puts(postfix);

return 0;

}
```

Output :


```
enter infix expression:a+b*c/d

postfix expression: abc*d/+
```

Conclusion:

Infix notation is the notation in which operators comes between the required operands. Postfix notation is the type of notation in which operators comes after the operand. Infix expression can be converted to posŋix expression using stack. We saw how to convert infix to posŋix by writing code in the language of c.