**Batch:\_\_A3\_\_\_\_\_\_\_\_ Roll No.:\_\_\_\_\_1911034\_\_\_\_\_\_\_\_\_**

**Experiment No. 2**

**Grade: AA / AB / BB / BC / CC / CD /DD**

|  |
| --- |
| **Title:**  Implementation of infix to postfix transformation and its evaluation using stack |

**Objective:** To understand the application of Last In First Out Data Structure “Stack” in practical applications such as converting the infix expression to postfix and postfix evaluation using either arrays or linked list.

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **CO** | **Outcome** |
| 1 | Explain the different structures used in problem solving. |

**Books/ Journals/ Websites referred:**

1. Data Structures A Pseudocode Approach with C, Richard F. Gilberg & Behrouz A. Forouzan, second edition, CENGAGE learning.
2. Data Structures Using C & C++, Rajesh K. Shukla, Wiley- india.

**Abstract**:-

Different type of input expressions can be given as input for evaluation in some real mathematical models. General-Purpose programming languages like C, C++ do not have built-in data types or functions to convert the infix expressions and their evaluation directly. So, there by comes an important data structure Stack which is Last- In-First-Out (LIFO) concept, which is used for converting an infix expression to postfix or prefix expressions. Stacks are implemented using Arrays or Linked Lists. Computer also uses its own stack called System Stack.

**Related Theory: -**

Infix expression is the user inputted expression which may consist brackets or can be bracket-free. Infix is the one where operators are placed in between operands. Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules.

➢ Postfix Expression is the one where all the operators are placed after the operands and it is a bracket-free expression. This makes the evaluation of expression easy and less complicated as we don’t have to follow operator precedence rules while evaluating it.

**Related Theory (contd...): -**

**Infix expression:**The expression of the form a op b. When an operator is in-between every pair of operands.

**Postfix expression:**The expression of the form a b op. When an operator is followed for every pair of operands.

**Why postfix representation of the expression?**  
The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d  
If op1 = +, op2 = \*, op3 = +

The compiler first scans the expression to evaluate the expression b \* c, then again scan the expression to add a to it. The result is then added to d after another scan.

**Examples of conversion from Infix to Postfix :**

**Exp =** "a+b\*(c^d-e)^(f+g\*h)-i"

**Postfix Equivalent :** abcd^e-fgh\*+^\*+i-

**Example of Postfix Evaluation of Expression :**

**Exp =** A B \* C D \* +

**Evaluation** : A \* B + C \* D

**Diagram for Push, Pop operations on the stack:**





**Program Source Code:**

**Infix to Postfix :**

#define SIZE 50

#include <ctype.h>

#include <stdio.h>

char s[SIZE];

int top = -1;

void RemoveSpaces(char\* source) {

char\* i = source;

char\* j = source;

while(\*j != 0) {

\*i = \*j++;

if(\*i != ' ')

i++;

}

\*i = 0;

}

/\* Function for PUSH operation \*/

void push(char elem) {

s[++top] = elem;

}

char pop() {

return (s[top--]);

}

/\* Function for precedence \*/

int pr(char elem) {

switch (elem) {

case '#':

return 0;

case '(':

return 1;

case '+':

case '-':

return 2;

case '\*':

case '/':

return 3;

}

}

void infix\_to\_postfix(char \*infix, char \*postfix) {

char ch, elem;

int i = 0, k = 0;

RemoveSpaces(infix);

push('#');

while ((ch = infix[i++]) != '\n') {

if (ch == '(')

push(ch);

else if (isalnum(ch))

postfix[k++] = ch;

else if (ch == ')') {

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

postfix[k++] = pop();

elem = pop();

} else {

while (pr(s[top]) >= pr(ch))

postfix[k++] = pop();

push(ch);

}

}

while (s[top] != '#') //pop from the stack till it is empty

postfix[k++] = pop();

postfix[k] = 0;

}

int main() {

char infx[50], pofx[50];

printf("\nInput the infix expression: ");

fgets(infx, 50, stdin);

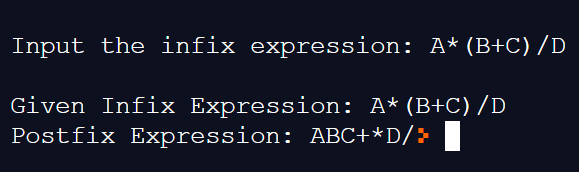
infix\_to\_postfix(infx, pofx);

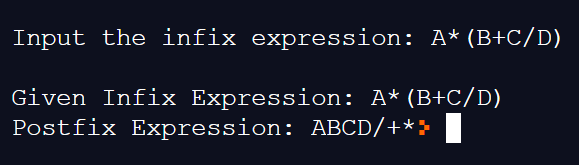
printf("\nGiven Infix Expression: %sPostfix Expression: %s", infx, pofx);

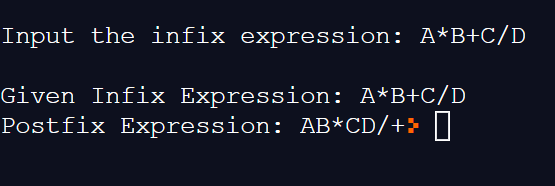
top = -1;

}

**Output screenshots: (Infix to Postfix)**

****

****

****

**Postfix Evaluation :**

#include <stdio.h>

#include<stdlib.h>

#include<string.h>

#include<math.h>

#define MAX 50

struct Stack

{

int top;

int a[MAX];

};

void push(struct Stack\*,int val);

int pop(struct Stack\*);

int PostfixtoInfix(char str[],struct Stack\*);

int main(void) {

struct Stack s;

struct Stack \*a=&s;

char str[50]

;

int ans=0;

s.top=-1;

int c;

printf("Enter the string\n");

fgets(str, sizeof(str), stdin);

ans= PostfixtoInfix(str,a);

printf("the answer is %d",ans);

}

void push( struct Stack \*b, int val)

{

b->top=b->top+1;

b->a[b->top]=val;

//since we only pass the pointer to the struct , thus any object of the struct that we reference has to be incremented with the help of the pointer itself

}

int pop(struct Stack \*f)

{float pvar=0;

if(f->top!=-1)

{

pvar=f->a[f->top];

f->top--;

}

return pvar;

}

int PostfixtoInfix(char str[],struct Stack\*a)

{

int i;

int res=0;

for(i=0;str[i]!='\0';i++)

{int temp=0;res=0;

if(str[i]=='0'||str[i]=='1'||str[i]=='2'||str[i]=='3'||str[i]=='4'||str[i]=='5'||str[i]=='6'||str[i]=='7'||str[i]=='8'||str[i]=='9')

{

push(a,(str[i]-48));

}

else if(str[i]=='+')

{

temp= pop(a);

res=pop(a)+temp;

push(a,res);

}

else if(str[i]=='-')

{

temp= pop(a);

res=pop(a)-temp;

push(a,res);

}

else if(str[i]=='\*')

{

temp= pop(a);

res=pop(a)\*temp;

push(a,res);

}

else if(str[i]=='/')

{

temp= pop(a);

res= (pop(a)/temp);

push(a,res);

}

else if(str[i]=='^')

{

temp= pop(a);

res=pow(pop(a),temp);

push(a,res);

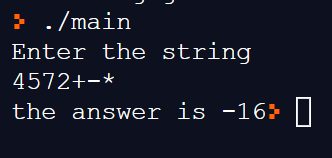
}

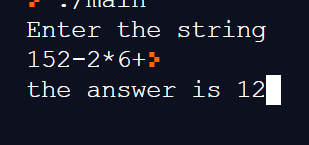
}

return(pop(a));

}

**Output Screenshots (Postfix Evaluation):**

****



**Implementation Details:**

1. **Enlist all the Steps followed and various options explored**

**For Infix to Postfix steps followed were as follows :**

1. First the Infix expression is accepted from the user in the main function in the form of a string.
2. It is then passed to the Infix to Postfix function where first we remove any spaces , if the user may have entered them.
3. Using an Algorithm that uses the stack data structure ( described below) , we have successfully converted the expression from Infix to Postfix and returned the value to the main() function which is printed on the console.
4. The option that the user can provide are that the user can enter the Infix Expression with or without the parentheses, and it gives the postfix expression for both the cases.
5. Also it is important to note that the addition of parentheses to any expression overrides the priority order of any preceding operators on the stack. The order is again restored , after the parentheses have been popped.

**For Postfix Evaluation the steps followed were as follows:**

1. First we accept the postfix expression from the user in the form of a string.
2. Using the stack data structure , we implement the postfix evaluation , wherein if we encounter any operand in the string , we push it onto the stack and if we encounter any operator after any two operands we pop the previous two operands and perform the operation that is indicated by the operator.
3. The final result is stored in the stack as a single element which is finally returned to the main() function and the result is printed onto the console window.
4. **Explain your program logic, classes and methods used.**
5. **For Infix to Postfix**
6. **Program Logic (Algorithm is as follows ):**

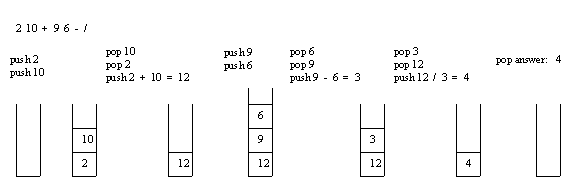
* Scan input string from left to right character by character.
* If the character is an operand, put it into output stack.
* If the character is an operator and operator's stack is empty, push operator into operators' stack.
* If the operator's stack is not empty, there may be following possibilites.
  + If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operand's stack.
  + If the precedence of scanned operator is less than or equal to the top most operator of operator's stack, pop the operators from operand's stack untill we find a low precedence operator than the scanned character. Never pop out (**'('**) or (**')'**) whatever may be the precedence level of scanned character.
  + If the character is opening round bracket (**'('**), push it into operator's stack.
  + If the character is closing round bracket (**')'**), pop out operators from operator's stack until we find an opening bracket (**'('** ).
  + Now pop out all the remaining operators from the operator's stack and push into output stack.

1. **Methods used in Infix to Postfix Conversion are as follows :**
2. int main ();
3. void infix\_to\_postfix(char \*infix, char \*postfix)// function that converts infix to postfix
4. int pr(char elem) // function for assigning priority to the operators.
5. void RemoveSpaces(char\* source) // function to remove whitespaces if user has entered any.
6. void push( char elem) // function to push the operators on the stack.
7. **Program Logic (Algorithm) for postfix Evaluation:**

Create an empty stack and start scanning the postfix expression from left to right.

* If the element is an operand, push it into the stack.
* If the element is an operator **O**, pop twice and get A and B respectively. Calculate B**O**A and push it back to the stack.
* When the expression is ended, the value in the stack is the final answer.

Evaluation of a postfix expression using a stack is explained in below example:



**Methods used were as follows** :

1.int main(void)

2. void push(struct Stack\*,int val);// to push the operands on the stack and to push the result after performing the operation

3. int pop(struct Stack\*); // to pop the final result of the stack

4.int PostfixtoInfix(char str[],struct Stack\*); // driver function to evaluate the postfix expression

1. **Explain the Importance of the approach followed by you**
   * 1. In this approach, we have applied our knowledge of the stack data structure to convert an infix expression to it’s Postfix Type and evaluate it.
     2. The basic operations on the stack ADT i.e, Push and Pop were implemented in the code.
     3. We have also considered the effect of adding parentheses to the infix expression which overrides the priority of any operators that were already on the stack.
     4. The code was organized and broken down into different functions like InfixtoPostfix(), Push() and Pop() , which was handled by the main function , for easier access to individual operations in the code like pushing , popping as well as assigning priority to the operators.
     5. We have also taken into account the special cases such as popping from a stack that is not already empty and handling other exceptions too.

**Conclusion:-** Through this experiment , we have successfully applied our knowledge of Stack Data Structure and the operations that can be performed on it , to convert an infix expression to it’s corresponding postfix equivalent as well as evaluation of the postfix expression.