CONVERSION FROM ONE FORM TO OTHER

INFIX TO POSTFIX CONVERSION

Aim: To convert an infix expression to postfix expression using stack.

Theory: The postfix expression is an expression in which the operator is written after the operands. For example, the postfix expression of infix notation (2+3) can be written as 23+.   Conversion of infix to postfix:-

Here, we will use the stack data structure for the conversion of infix expression to prefix expression. Whenever an operator will encounter, we push operator into the stack. If we encounter an operand, then we append the operand to the expression.

Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming

operator. If the associativity is from right to left then push the incoming operator.

At the end of the expression, pop and print all the operators.

Program:

```
#include<stdio.h>                                    //preprocessor directives

#include<stdlib.h>

#include<string.h>

#define MAX 100                                  //defining array size

char stack[100];

char infix[MAX],postfix[MAX];                //infix and postfix array declaration

int top=-1;                                   //declaring top as -1

void push(char);                            //function declaration

int precedence(char);                     //function declaration

char pop();                                 //function declaration

void inTopost();                           //function declaration

int space(char);                          //function declaration

void print();                             //function declaration

int isEmpty();                           //function declaration

int main()                               //main function

{

    printf("Enter the infix expression:");                  //prints
```

```
       scanf("%s",infix);                              //reads string
       inTopost();                                //function call
      print();                                      //function call
      return 0;
}
void inTopost()                         //called function
{
     int i,j=0;
     char next;
     char symbol;
     for(i=0;i<strlen(infix);i++)                  //for loop to accept input
   {
        symbol=infix[i];          //infix[i] is assigned to ‘symbol’ variable
       switch(symbol)                              //switch condition
     {
         case '(':
         push(symbol);                       //pushes symbol into stack
         break;                                         //terminates
        case ')':
        while((next=pop())!='(')
       postfix[j++]=next;                     //returns top of the stack until ‘(‘
       break;

      case '+':

      case '-':

      case '*':

      case '/':

     case '^':

//if operator has lower priority that top of stack then top ele gets popped and
printed then operator is pushed into stack

     while(!isEmpty()&&precedence(symbol)<=precedence(stack[top]))

     postfix[j++]=pop();

    push(symbol);

    break;

   default:

    postfix[j++]=symbol;                              //operand is pushed into postfix array

   }

 }

  while(!isEmpty())

  postfix[j++]=pop();                        //at end all elements are popped out

  postfix[j];

}

int precedence(char symbol)                  //called function for precedence

{
      switch(symbol)
     {
          case '^':
          return 3;
         case '/':
         case '*':
         return 2;
        case '+':
       case '-':
       return 1;
       default:
      return 0;
     }
  }
void print()
{
   int i=0;
   printf("The postfix expression is:\n");
   while(postfix[i])
  {
     printf("%c",postfix[i++]);                         //prints the elements of the array
  }
      printf("\n");
}
void push(char c)                              //called function for push
{
    int Max;
    if(top==Max-1)                                      //checks if stack is full
   {
       printf("stack overflow\n");
}
  top++;                                   //top incrementation
   stack[top]=c;                            //element is inserted at top
  }
 char pop()                                          //called function
{
   char c;
   if(top==-1)                                                       //checks if stack is empty
{
    printf("stack underflow\n");
   exit(1);
}
```

```
    c=stack[top];                              //top element is popped

    top=top-1;                                  //top decrementation

      return c;                                //returns top element

  }

int isEmpty()                           //called function

{

    if(top==-1)                                 //checks if it is empty or not

    return 1;

    else

    return 0;

}
```

Algorithm:

Step 1**:** If the scanned character is an operand, put it into postfix expression.

Step 2: If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.

Step 3: If the operator's stack is not empty, there may be following possibilities. If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.

If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack until we find a low precedence operator than the scanned character.

If the precedence of scanned operator is equal then check the associativity of the operator. If associativity left to right then pop the operators from stack until we find a low precedence operator. If associativity right to left then simply put into stack.

If the scanned character is opening round bracket ('('), push it into operator's stack.

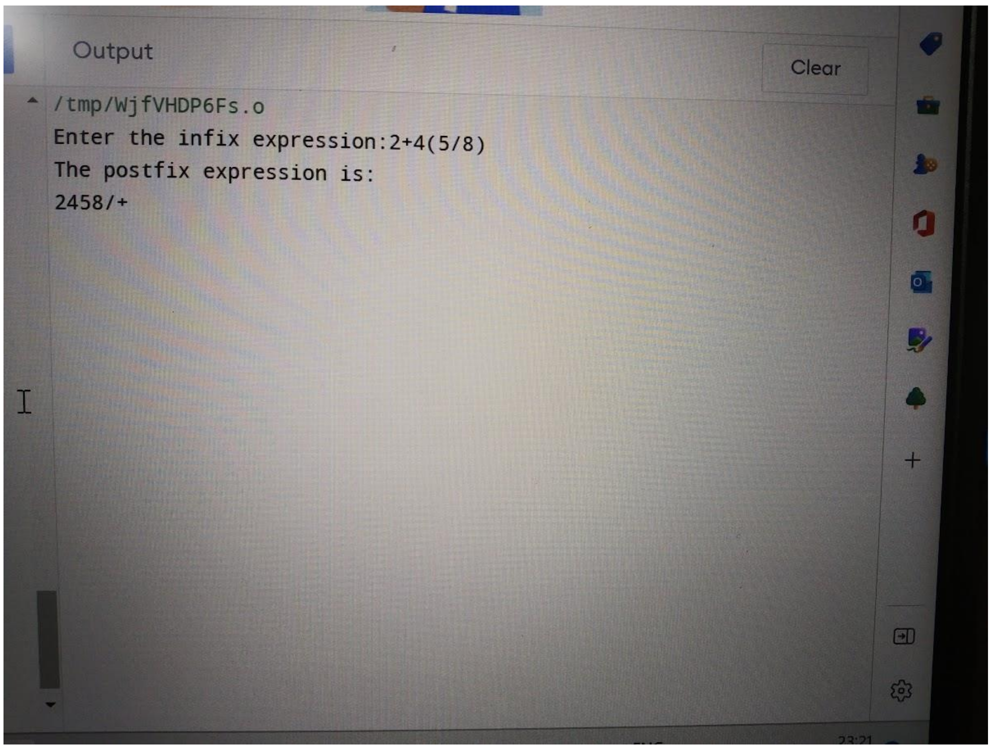
If the scanned character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').

Repeat Step 1,2 and 3 till expression has character

Step 4: Now pop out all the remaining operators from the operator's stack and push into postfix expression.

Step 5: Exit

Output:



```
/tmp/WjfVHDP6Fs.o
Enter the infix expression:2+4(5/8)
The postfix expression is:
2458/+
```

INFIX TO PREFIX CONVERSION

Aim: To convert an infix expression to prefix expression using stack.

Theory: To solve expressions by the computer, we can either convert it in postfix form or to the prefix form. Here we will see how infix expressions are converted to prefix form.

At first infix expression is reversed. Note that for reversing the opening and closing parenthesis will also be reversed.

for an example: The expression: A + B * (C - D)

after reversing the expression will be:) D – C (* B + A

so we need to convert opening parenthesis to closing parenthesis and vice versa.

After reversing, the expression is converted to postfix form by using infix to postfix algorithm. After that again the postfix expression is reversed to get the prefix expression.

Program:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define ss 20

char infix[20], res[20], mstack[20];

int top ;

//Push Operation

void push(char x){
     if(top == ss-1){

         printf("Stack is full \n");

}

     else{

    top++;

    mstack[top] = x;

   }

}

//Pop Operation

   char pop(){

    if(top == -1){

     return -1 ;

}

else{

       return (mstack[top--]) ;

     }

}

//Reversing the expression

void reverse(char *s, int begin, int end){

     char x ;

if(begin >= end){
      return ;

   }

while(begin<end){

    x = *(s+begin) ;

    *(s+begin) = *(s+end) ;

    *(s+end) = x ;

    ++begin ; --end;

   }

}

//Conversion to prefix

void prefixconversion(char infix[]){

    int i=0, j=0;

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

     if(isdigit(infix[i]) || isalpha(infix[i])){

        res[j++] = infix[i] ;

       }

    else{

    switch(infix[i]){

       case ')':

       push(infix[i]) ;

      break ;
    case '(':
   while(mstack[top]!=')'){
   res[j++] = pop() ;
  }
    pop();
   break ;
   case '*':
   case '/':
          while(mstack[top]=='$'){
          res[j++] = pop() ;
}
push(infix[i]) ;
break;
  case '+':
case '-':
          while(mstack[top]=='$'||mstack[top]=='*'||mstack[top]=='/'
){
     res[j++] = pop() ;
}
       push(infix[i]) ;
       break;
     }
  }
}
      while(top >-1){
      res[j++] = pop() ;
      res[j] = '\0';
    }
     reverse(res,0,strlen(res)-1) ;
     printf("Prefix Expression: %s\n", res) ;
}
//Main Function
int main(){
     top = -1 ;
    printf("Enter the Infix Expression:") ;
     scanf("%s", infix) ;
     reverse(infix, 0, strlen(infix)-1) ;
     printf("Reversed Infix: %s\n", infix) ;
     prefixconversion(infix) ;
     return 0 ;
}
```

Algorithm:

Step 1: First reverse the given expression

Step 2: If the scanned character is an operand, put it into prefix expression.

Step 3: If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.

Step 4: If the operator's stack is not empty, there may be following possibilities.

If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.

If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack untill we find a low precedence operator than the scanned character.

If the precedence of scanned operator is equal then check the associativity of the operator. If associativity left to right then simply put into stack. If associativity right to left then pop the operators from stack until we find a low precedence operator.

If the scanned character is opening round bracket ('('), push it into operator's stack.

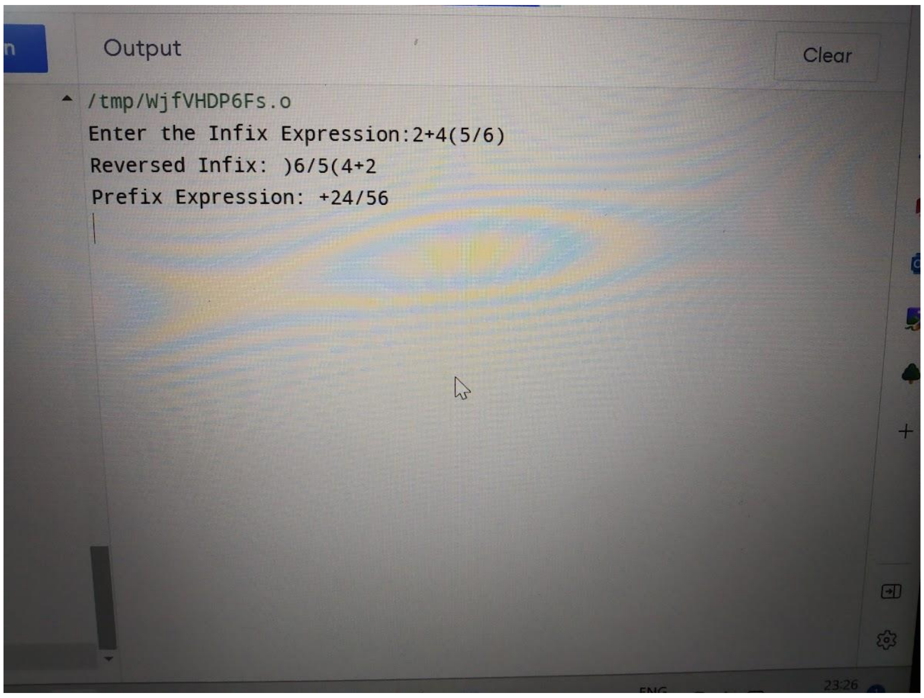
If the scanned character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').

Repeat Step 2,3 and 4 till expression has character

Step 5: Now pop out all the remaining operators from the operator's stack and push into postfix expression.

Step 6: Exit

Output:



```
/tmp/WjfVHDP6Fs.o
Enter the Infix Expression:2+4(5/6)
Reversed Infix: )6/5(4+2
Prefix Expression: +24/56
```

PRIFIX TO POSTFIX CONVERSION

Aim: To convert a prefix expression into postfix expression.

Theory: To solve this problem, we will first traverse the whole postfix expression in an reverse order. And we will be using the stack data structure for our processing. And do the following for cases of elements found of traversal

Case: if the symbol is operand -> push(element) in stack.

Case: if symbol is operator -> 2*pop(element) from stack. And then push sequence of operand - operand - operator.

Rules for prefix to postfix expression using stack data structure:

- Scan the prefix expression from right to left, i.e., reverse.
- If the incoming symbol is an operand then push it into the stack.
- If the incoming symbol is an operator then pop two operands from the stack. Once the operands are popped out from the stack, we add the incoming symbol after the operands. When the operator is added after the operands, then the expression is pushed back into the stack.
- Once the whole expression is scanned, pop and print the postfix expression from the stack.

Program:

```
#include<stdio.h>                    //preprocessor directives

#include<string.h>

#include<math.h>

#include<stdlib.h>
```

```
#define BLANK ' '

#define TAB '\t'

#define MAX 50

char *pop();                              //function declaration

char prefix[MAX];

char stack[MAX][MAX];

void push(char *str);                     //function declaration

int isempty();

int white_space(char symbol);

void prefix_to_postfix();

int top;

int main()

{

      top = -1;

      printf("Enter Prefix Expression : ");

      gets(prefix);                        //gets string

      prefix_to_postfix();                 //function call

}/*End of main()*/

void prefix_to_postfix()                    //called function

{

      int i;
```

```
char operand1[MAX], operand2[MAX];

char symbol;

char temp[2];

char strin[MAX];

for(i=strlen(prefix)-1;i>=0;i--)                    //for loop

{

        symbol=prefix[i];

        temp[0]=symbol;

        temp[1]='\0';

        if(!white_space(symbol))

        {

                switch(symbol)

                {

                case '+':

                case '-':

                case '*':

                case '/':

                case '%':

                case '^':

                        strcpy(operand1,pop());

                        strcpy(operand2,pop());
                        strcpy(strin,operand1);

                        strcat(strin,operand2);

                        strcat(strin,temp);

                        push(strin);

                        break;

                default: /*if an operand comes*/

                    push(temp);

                }

            }

        }

        printf("\nPostfix Expression :: ");

        puts(stack[0]);

}/*End of prefix_to_postfix()*/

void push(char *str)                                //called function

{

        if(top > MAX)

        {

                printf("\nStack overflow\n");

                exit(1);

        }

        else
        {
                top=top+1;
                strcpy( stack[top], str);
        }
}/*End of push()*/
char *pop()
{
        if(top == -1 )
        {
                printf("\nStack underflow \n");
                exit(2);
        } else
                return (stack[top--]);
}/*End of pop()*/
int isempty()                       //called function checks if stack is empty or not
{
        if(top==-1)
                return 1;
        else
                return 0;
}
```

```
int white_space(char symbol)

{

        if(symbol==BLANK || symbol==TAB || symbol=='\0')

                return 1;

        else

                return 0;

}/*End of white_space()*/
```

Algorithm:

Step 1: Start

Step 2: Read the Prefix expression from right to left.

Step 3: If the scanned character is an operand, then push it onto the Stack.

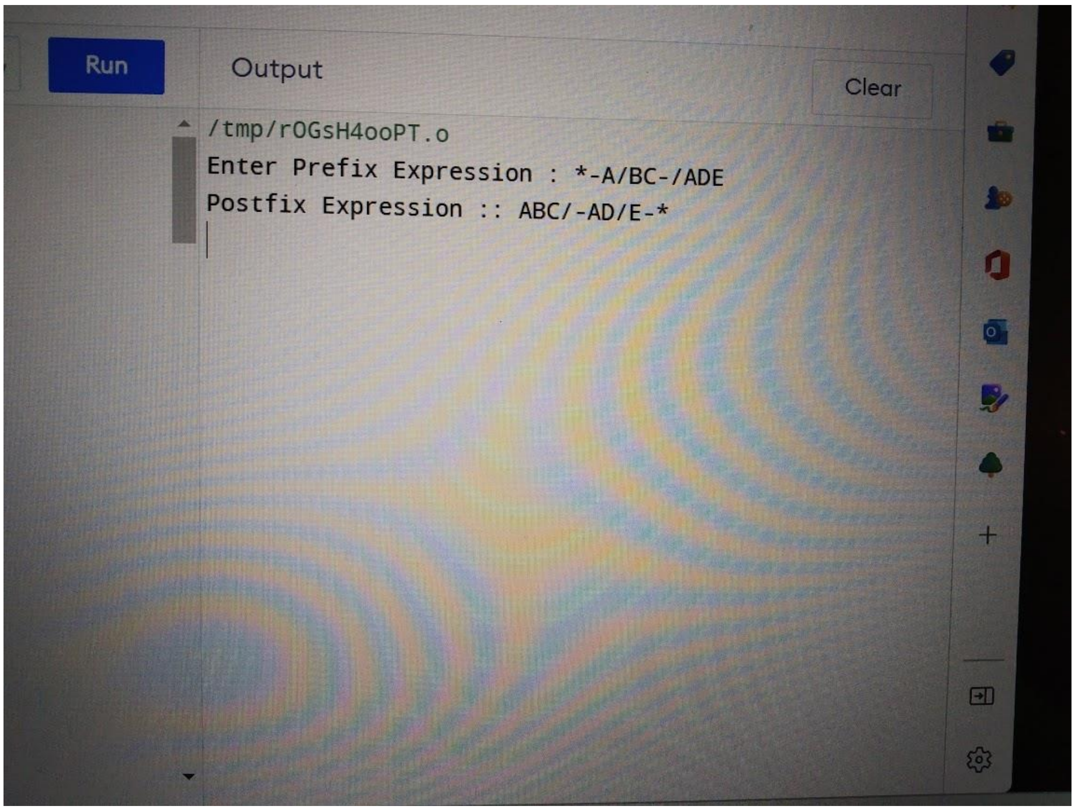
Step 4: If the scanned character is an operator, pop two operands from the stack and concatenate them in order of 'operand1, operand2, operator'. Push the result into the stack.

Step 5: Repeat steps 2-4 until the prefix expression ends.

Step 6: Pop all the rest elements of the stack if any.

Step 7: Stop

Output:



```
/tmp/rOGsH4ooPT.o
Enter Prefix Expression : *-A/BC-/ADE
Postfix Expression :: ABC/-AD/E-*
```