Evaluation of Expression

An **Expression** is made up of ***operands*** and ***operators***. The operations to be performed on the operands are described by the associated operator. In evaluating any given expression, operators of a higher precedence are processed first. When two adjacent operators in an expression have the same precedence, evaluation is performed from left to right.

How can a compiler accept an expression and produce a correct code? The answer is given by the rewriting the expression into a form called the ***Postfix Notation.***

Infix Notation is being used in writing all of the expressions. The ***Infix Notation*** is characterized by the sequence:

operand	operator	operand
C	**+**	**D**

The ***Postfix Notation***, on the other hand is characterized by the sequence

operand	operand	operator
C	**D**	**+**

Postfix notation, also known as reverse Polish notation, is a syntax for mathematical expressions in which the mathematical operator is always placed after the operands. Though postfix expressions are easily and efficiently evaluated by computers, they can be difficult for humans to read. Complex expressions using standard parenthesized infix notation are often more readable than the corresponding postfix expressions. Consequently, we would sometimes like to allow end users to work with infix notation and then convert it to postfix notation for computer processing. Sometimes, moreover, expressions are stored or generated in postfix, and we would like to convert them to infix for the purpose of reading and editing

Infix Expression

An infix expression may be directly translated into postfix form by beginning with the conversion of the sub-expression with the highest precedence and so on. One important rule to remember when converting expressions is that the order by which the operands appear in the infix expression must be the same when in the postfix form.

Why postfix representation of the expression?

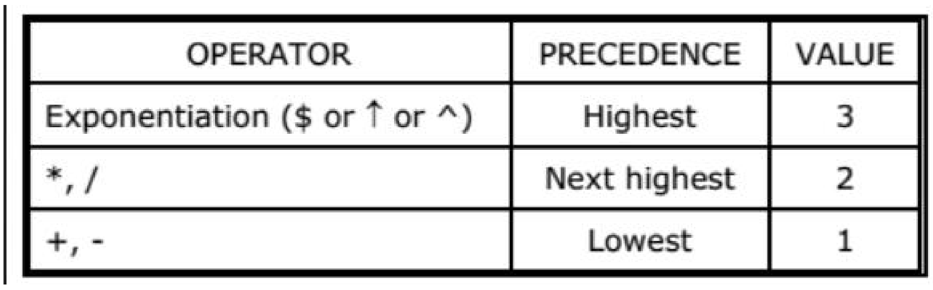
Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix(or prefix) form before evaluation. The corresponding expression in postfix form is **abc*+d+**. The postfix expressions can be evaluated easily using a stack.

Infix to Postfix

An infix expression may be directly translated into postfix form by beginning with the conversion of the sub-expression with the highest precedence and so on.
Example: The expression 1 + 2 × 3 is interpreted to have the value 1 + (2 × 3) = 7, and not (1 + 2) × 3 = 9

Order of Precedence or Operations

PEMDAS=**P**arenthesis **E**xponents **M**ultiplication **D**ivision **A**ddition **S**ubtraction.



OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

.

Note: One important rule to remember when converting expressions is that the order by which the operands appear in the infix expression must be the same when in the postfix form.

How to convert an Infix expression to Postfix expression using Stack

1. Scan all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is an operand, then immediately append it to the Postfix Expression.
3. If the reading symbol is left parenthesis '(', then Push it onto the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of the stack until the respective left parenthesis is popped and append each popped symbol to Postfix Expression.
5. If the reading symbol is an operator (+, –, *, /), then Push it onto the Stack. However, first, pop the operators which are already on the stack that have higher or equal precedence than the current operator and append them to the postfix. If

an open parenthesis is there on top of the stack then push the operator into the stack.

6. If the input is over, pop all the remaining symbols from the stack and append them to the postfix.

Infix to Postfix Example

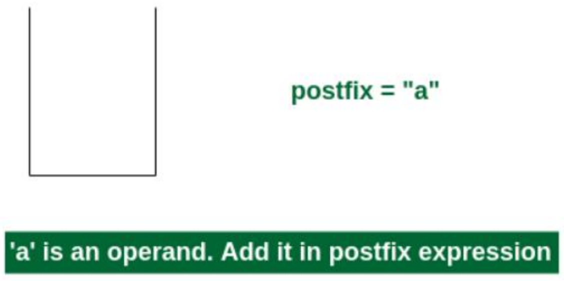
1.) a+b*c+d

Illustration:

Follow the below illustration for a better understanding

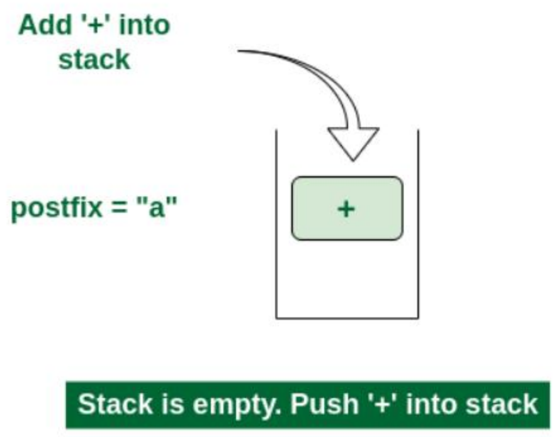
Consider the infix expression exp = **“a+b*c+d”** and the infix expression is scanned using the iterator i, which is initialized as i = 0.

1st Step: Here i = 0 and exp[i] = ‘a’ i.e., an operand. So, add this in the postfix expression. Therefore, postfix = “a”.



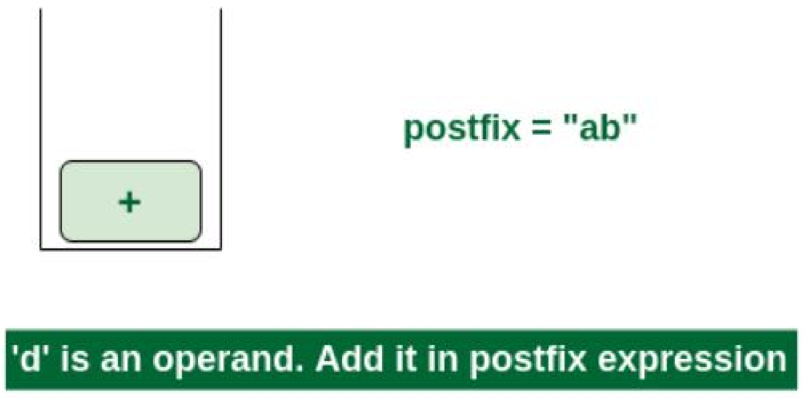
Add ‘a’ in the postfix

2nd Step: Here i = 1 and exp[i] = ‘+’ i.e., an operator. Push this into the stack. postfix = “a” and stack = {+}.



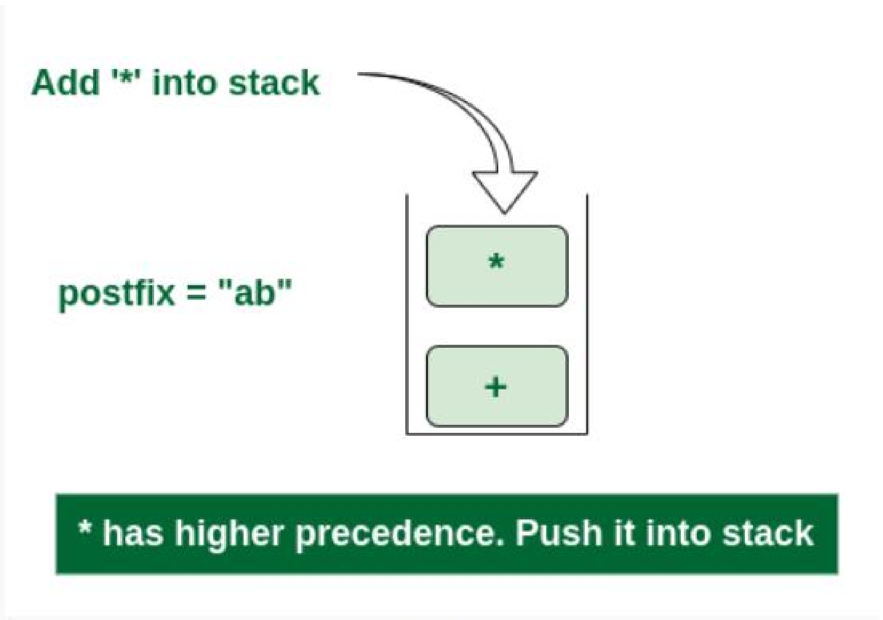
Push '+' in the stack

3rd Step: *Now i = 2 and exp[i] = 'b' i.e., an operand. So, add this in the postfix expression.* ***postfix = "ab"*** *and* ***stack = {+}****.*



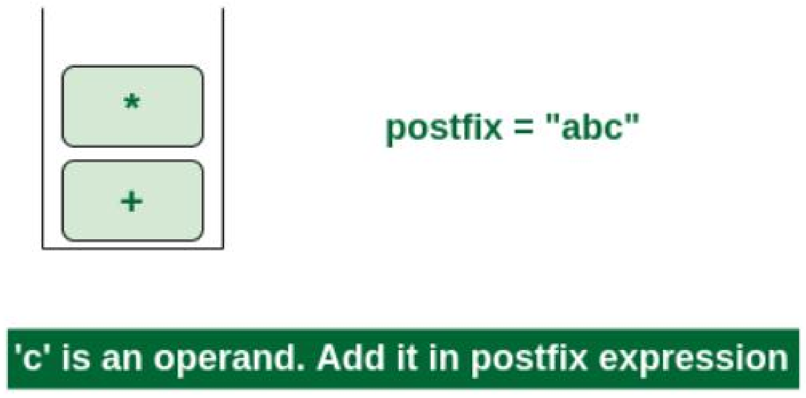
Add 'b' in the postfix

4th Step: *Now i = 3 and exp[i] = '*' i.e., an operator. Push this into the stack.* ***postfix = "ab"*** *and* ***stack = {+, *}****.*



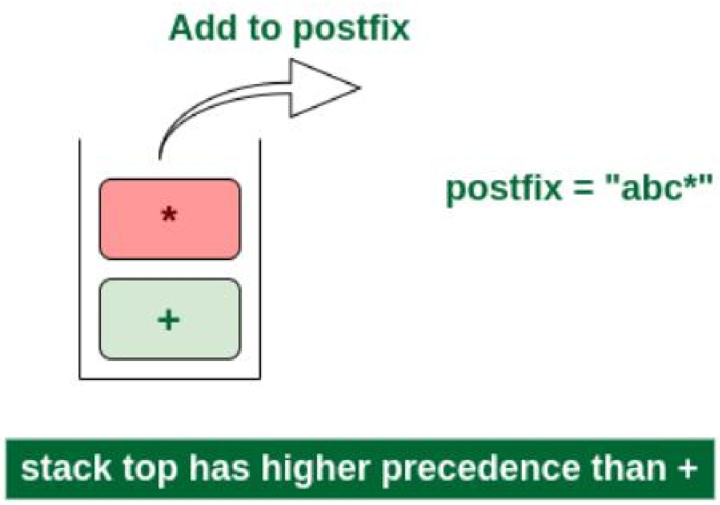
Push '' in the stack*

5th Step: *Now i = 4 and exp[i] = 'c' i.e., an operand. Add this in the postfix expression.* ***postfix = "abc"*** *and* ***stack = {+, *}***.



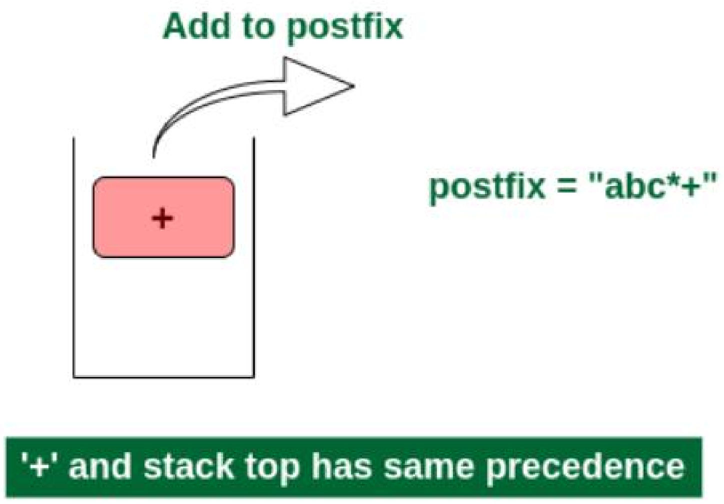
Add 'c' in the postfix

6th Step: *Now i = 5 and exp[i] = '+' i.e., an operator. The topmost element of the stack has higher precedence. So pop until the stack becomes empty or the top element has less precedence. '*' is popped and added in postfix. So* ***postfix = "abc*"*** *and* ***stack = {+}***.



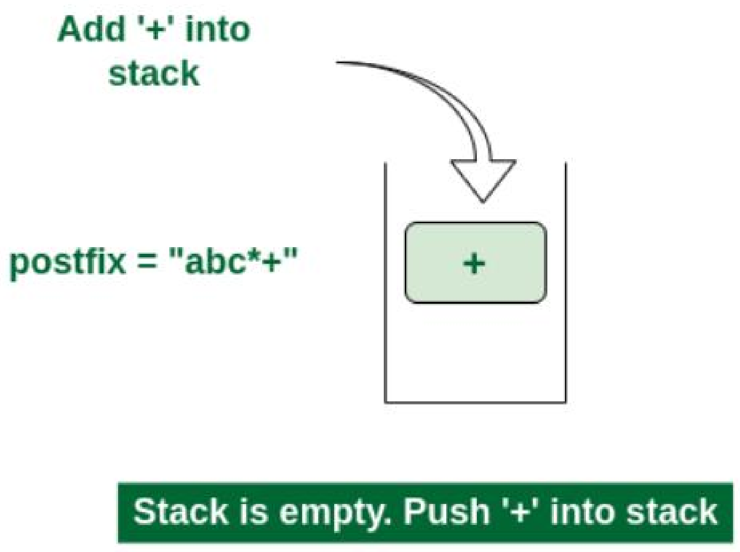
Pop '' and add in postfix*

*Now top element is '**+**' that also doesn't have less precedence. Pop it.* ***postfix = "abc*+"***.



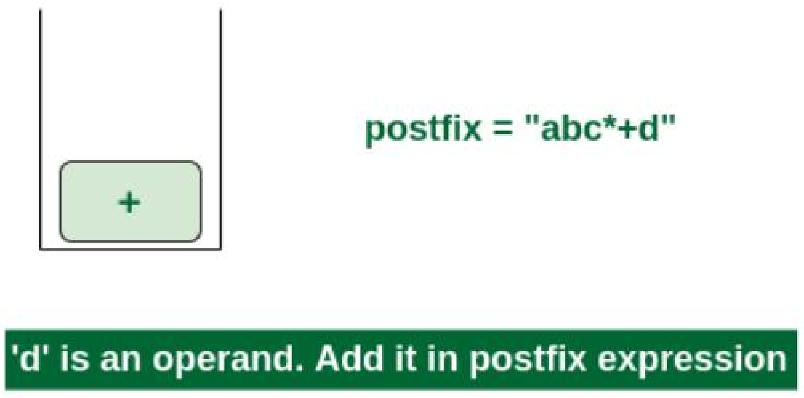
Pop '+' and add it in postfix

Now stack is empty. So, push ***'+'*** *in the stack.* ***stack = {+}***.



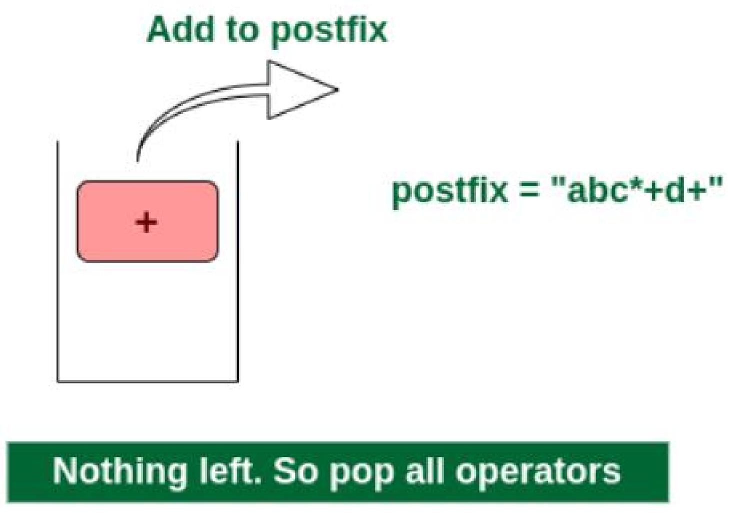
Push '+' in the stack

7th Step: *Now i = 6 and exp[i] = 'd' i.e., an operand. Add this in the postfix expression.* ***postfix = "abc*+d"****.*



Add 'd' in the postfix

Final Step: *Now no element is left. So empty the stack and add it in the postfix expression.* ***postfix = "abc*+d+"****.*



Pop '+' and add it in postfix

Input Token	Stack	Postfix Expression	Action
(	(		Push '(' into stack
A	(	A	Add A into Postfix
+	(+	A	Push '+' into stack
(	(+(	A	Push '(' into stack
(	(+((	A	Push '(' into stack
(	(+(((	A	Push '(' into stack
B	(+(((	AB	Add B into Postfix
*	(+(((*	AB	Push '*' into stack
C	(+(((*	ABC	Add C Postfix
)	(+((*)	ABC*	')' has come so pop * and add it into Postfix
+	(+((+	ABC*	Push '+' into stack
D	(+((+	ABC*D	Add D into Postfix
)	(+((+)	ABC*D+	')' has come so pop '+' and add it into Postfix
/	(+(/	ABC*D+	Push '/' into stack
E	(+(/	ABC*D+E	Add E into Postfix
)	(+(/)	ABC*D+E/	')' has come so pop / and add it into Postfix
)	(+)	ABC*D+E/+	')' has come so pop '+' and add it into Postfix

Example no. 2 (A+(((B*C)+D)/E))

Postfix to Infix

How to convert Postfix expression to Infix expression using Stack

1. Start Iterating the given Postfix Expression from Left to right
   If Character is operand then push it into the stack.
2. If Character is operator then pop top 2 Characters which is operands from the stack.
3. After popping create a string in which coming operator will be in between the operands.
4. push this newly created string into stack.
5. Above process will continue till expression have characters left

Example to convert postfix to Infix

1. **Postfix Expression: abc-+de-+**

Input Token	Stack	Action
a	a	push a in stack
b	a, b	push b in stack
c	a, b, c	push c in stack
—	a, b — c	pop b and c from stack and put — in between and push into stack
+	a + b — c	pop a and b-c from stack and put + in between and push into stack
d	a + b — c d	push d in stack
e	a + b — c, d, e	push e in stack
—	a + b — c, d — e	pop d and e from stack and put — in between and push into stack

+	((a + (b — c)) + (d — e))	pop a + b — c and d — e from stack and put + in between and push into stack

2. **postfix expression:** 752+*415–/–

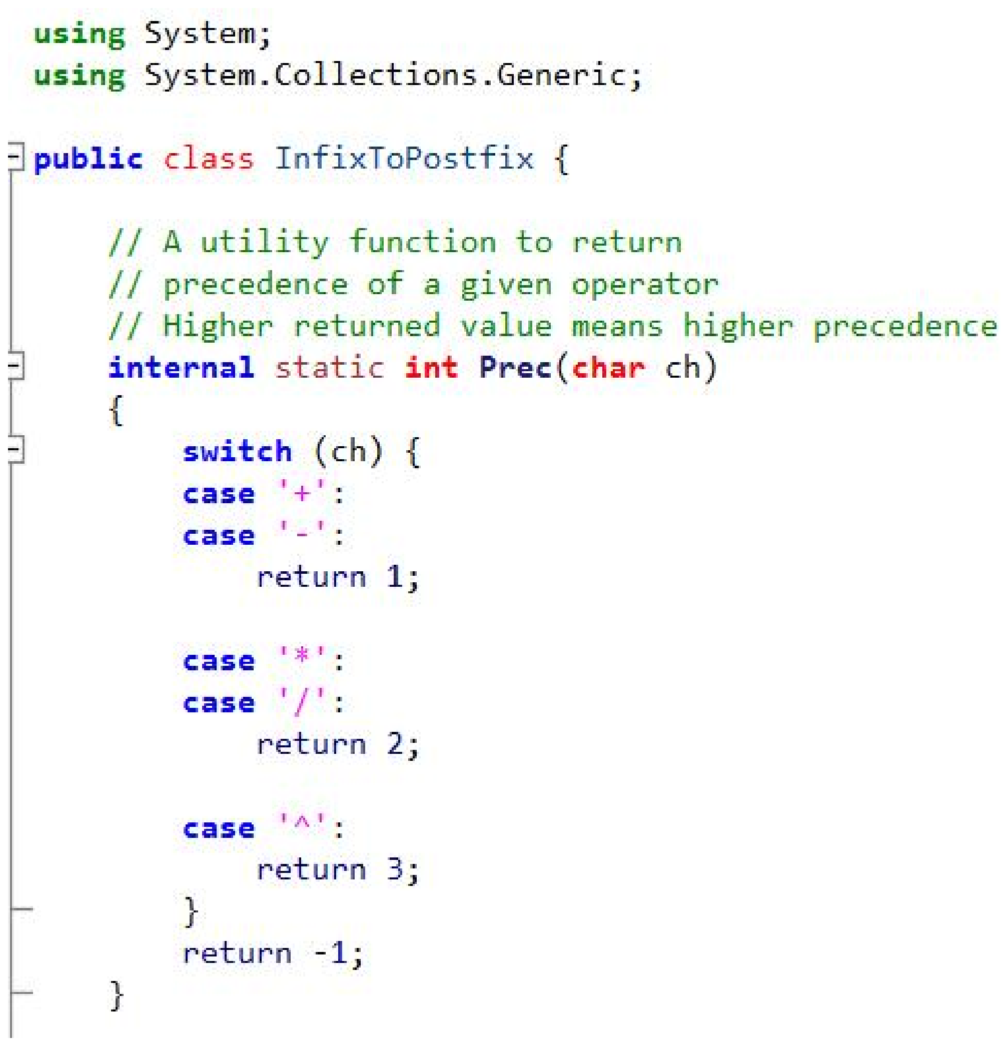
Token	Stack	Action
7	7	push 7 in stack
5	7, 5	push 5 in stack
2	7 , 5, 2	push 2 in stack
+	7, 7	pop 2 and 5 from stack, sum it and then again push it
*	49	pop 7 and 7 from stack and multiply it and then push it again
4	49, 4	push 4 in stack
1	49, 4, 1	push 1 in stack
5	49, 4, 1, 5	push 5 in stack
—	49, 4, –4	pop 5 and 1 from stack
/	49, –1	pop 4 and 4 from stack
—	50	pop 1 and 49 from stack

Evaluation/Generalization

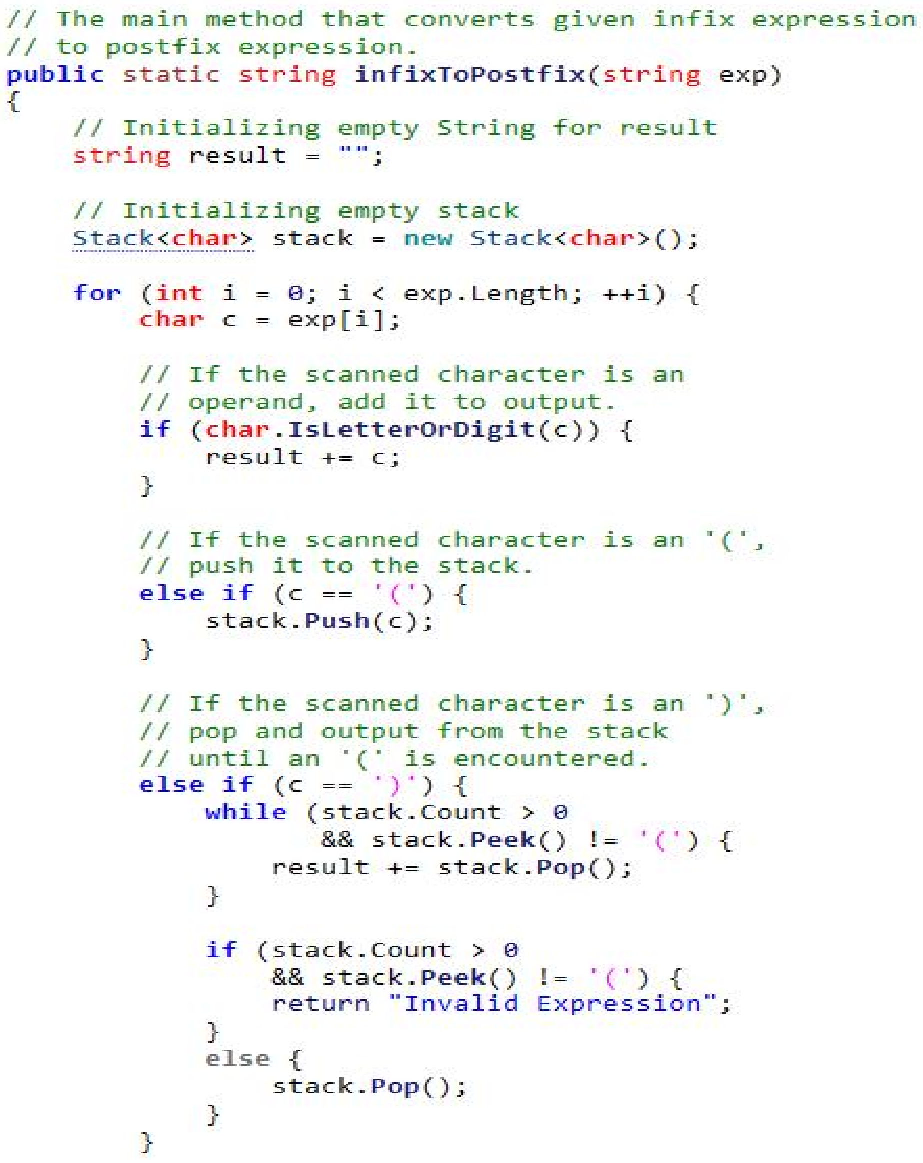
- ✓ In Stack, only one data item can be accessed, it is the last item inserted.
- ✓ Stacks are also useful in implementing recursion.
- ✓ The important stack operations are pushing (inserting) an item onto top of the stack and popping (removing) the item that's on the top.

✓ The efficiency of stack is that no comparisons or moves are necessary to perform any operations of it.

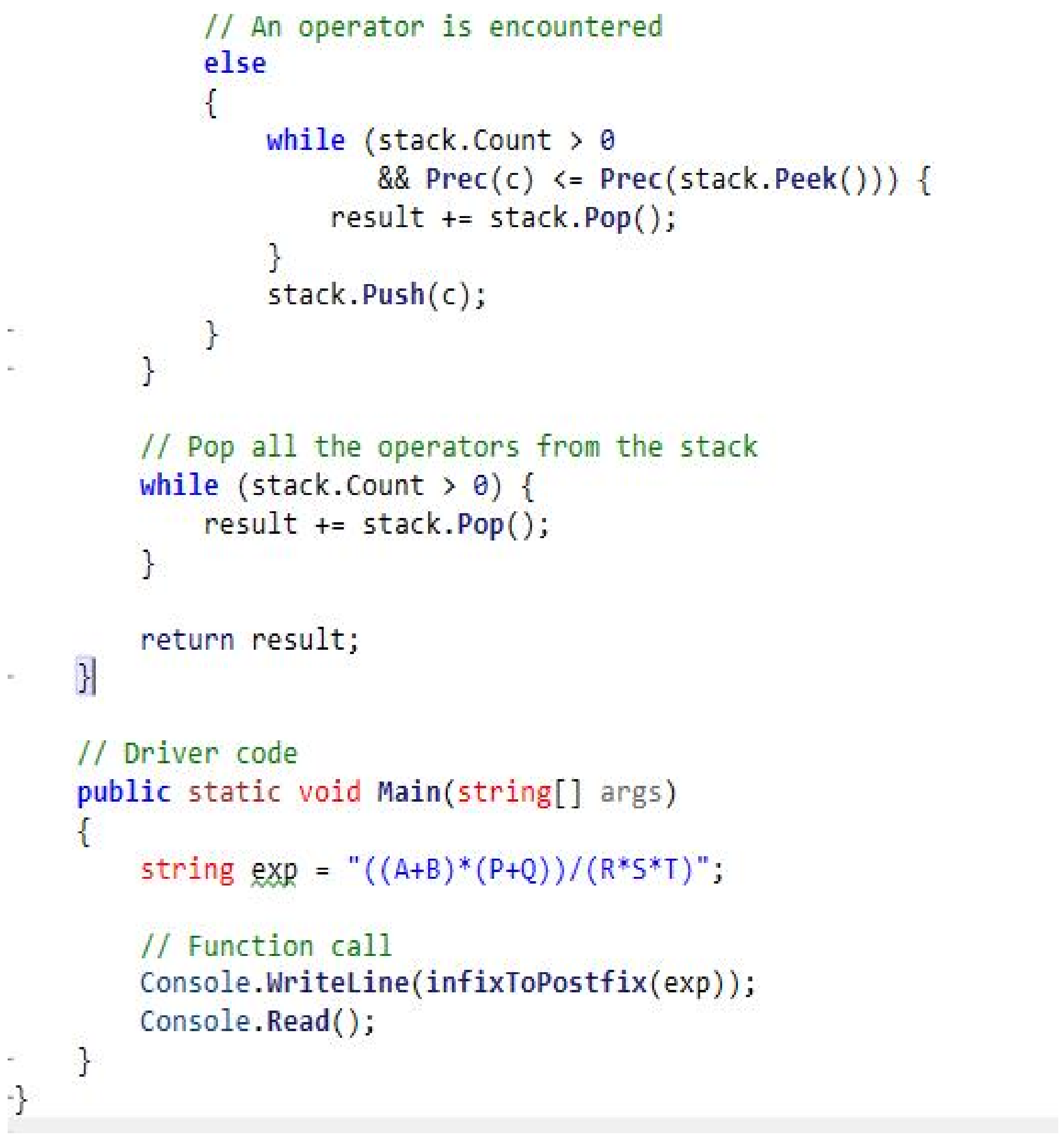
INFIX TO POSTFIX SAMPLE PROGRAM



```
using System;
using System.Collections.Generic;

public class InfixToPostfix {

    // A utility function to return
    // precedence of a given operator
    // Higher returned value means higher precedence
    internal static int Prec(char ch)
    {
        switch (ch) {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;

        case '^':
            return 3;
        }
        return -1;
    }
```

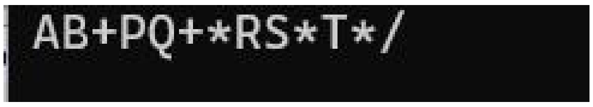


```
// The main method that converts given infix expression
// to postfix expression.
public static string infixToPostfix(string exp)
{
    // Initializing empty String for result
    string result = "";

    // Initializing empty stack
    Stack<char> stack = new Stack<char>();

    for (int i = 0; i < exp.Length; ++i) {
        char c = exp[i];

        // If the scanned character is an
        // operand, add it to output.
        if (char.IsLetterOrDigit(c)) {
            result += c;
        }

        // If the scanned character is an '(',
        // push it to the stack.
        else if (c == '(') {
            stack.Push(c);
        }

        // If the scanned character is an ')',
        // pop and output from the stack
        // until an '(' is encountered.
        else if (c == ')') {
            while (stack.Count > 0
                   && stack.Peek() != '(') {
                result += stack.Pop();
            }

            if (stack.Count > 0
                && stack.Peek() != '(') {
                return "Invalid Expression";
            }
            else {
                stack.Pop();
            }
        }
```



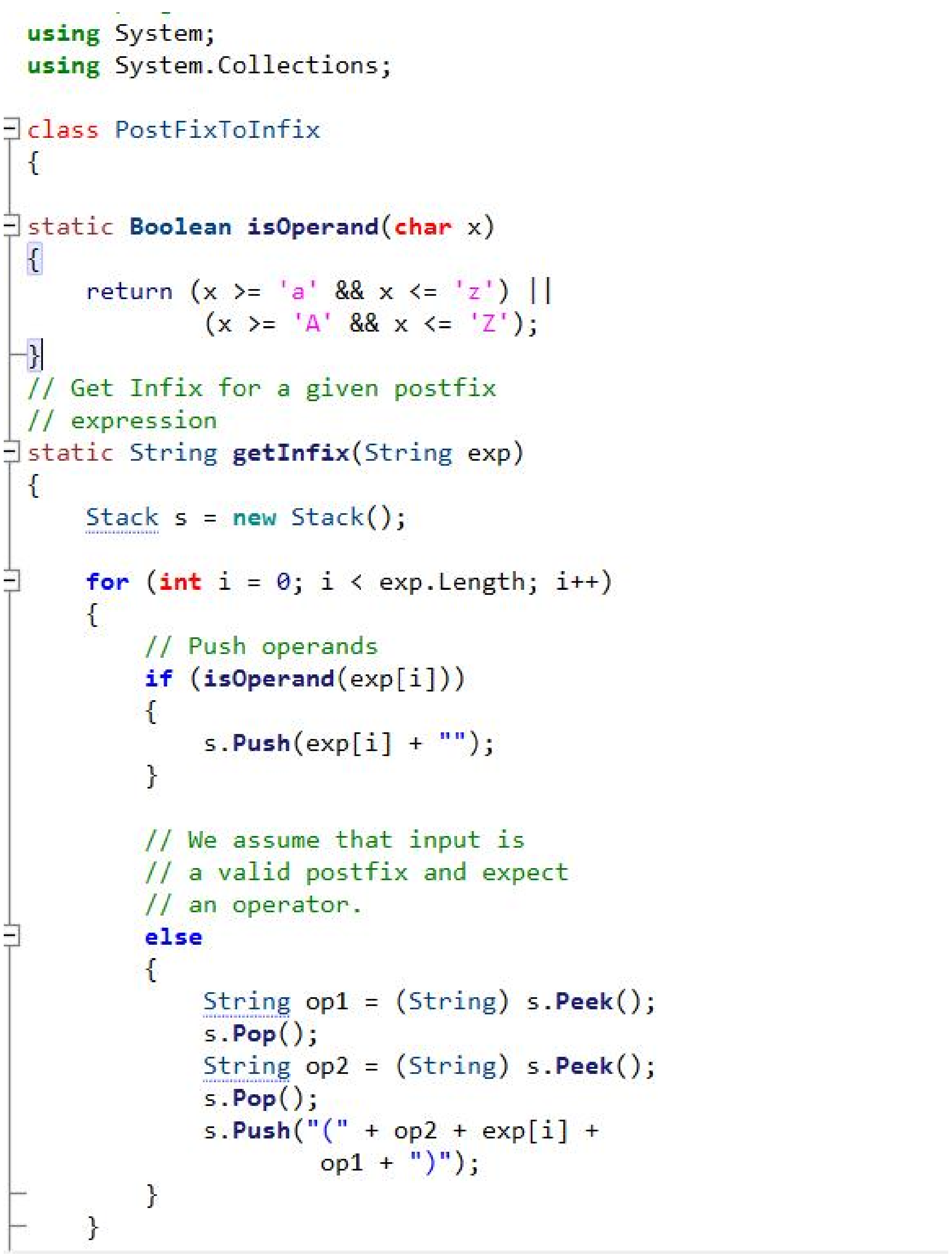
```
            // An operator is encountered
            else
            {
                while (stack.Count > 0
                       && Prec(c) <= Prec(stack.Peek())) {
                    result += stack.Pop();
                }
                stack.Push(c);
            }
        }

        // Pop all the operators from the stack
        while (stack.Count > 0) {
            result += stack.Pop();
        }

        return result;
    }

    // Driver code
    public static void Main(string[] args)
    {
        string exp = "((A+B)*(P+Q))/(R*S*T)";

        // Function call
        Console.WriteLine(infixToPostfix(exp));
        Console.Read();
    }
}
```

OUTPUT

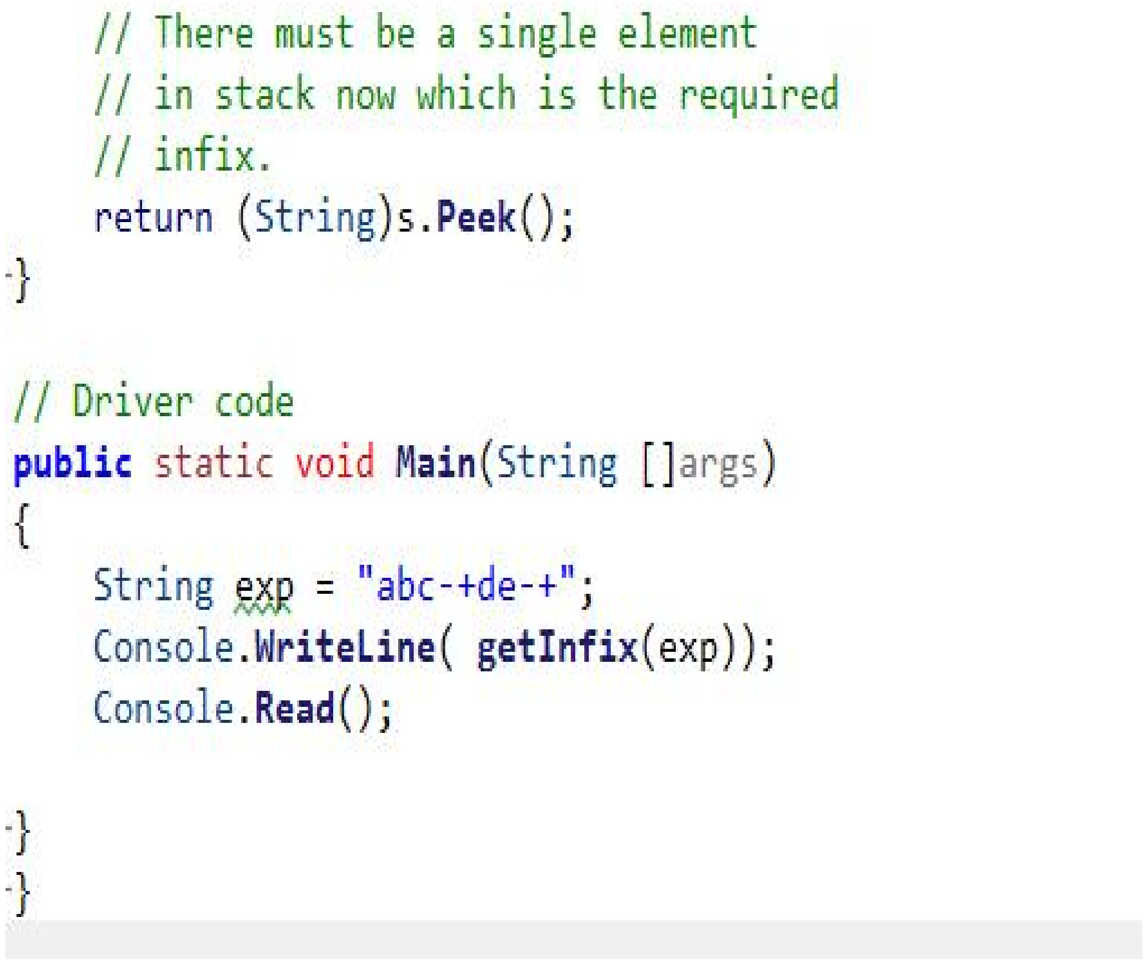


```
AB+PQ+*RS*T*/
```

POSTFIX TO INFIX

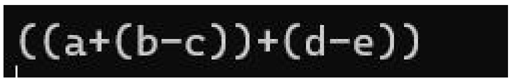


```
using System;
using System.Collections;

class PostFixToInfix
{

static Boolean isOperand(char x)
{
    return (x >= 'a' && x <= 'z') ||
           (x >= 'A' && x <= 'Z');
}
// Get Infix for a given postfix
// expression
static String getInfix(String exp)
{
    Stack s = new Stack();

    for (int i = 0; i < exp.Length; i++)
    {
        // Push operands
        if (isOperand(exp[i]))
        {
            s.Push(exp[i] + "");
        }

        // We assume that input is
        // a valid postfix and expect
        // an operator.
        else
        {
            String op1 = (String) s.Peek();
            s.Pop();
            String op2 = (String) s.Peek();
            s.Pop();
            s.Push("(" + op2 + exp[i] +
                    op1 + ")");
        }
    }
```



```
        // There must be a single element
        // in stack now which is the required
        // infix.
        return (String)s.Peek();
    }

    // Driver code
    public static void Main(String []args)
    {
        String exp = "abc-+de-+";
        Console.WriteLine( getInfix(exp));
        Console.Read();

    }
}
```

OUTPUT



```
((a+(b-c))+(d-e))
```