Convert Infix To Prefix Notation

geeksforgeeks.org/convert-infix-prefix-notation/

December 17, 2017

Last Updated : 27 Mar, 2023

Given an infix expression, the task is to convert it to a prefix expression.

> **Infix Expression:** The expression of type **a 'operator' b** (a+b, where + is an operator) i.e., when the operator is between two operands.
>
> **Prefix Expression:** The expression of type '**operator' a b** (+ab where + is an operator) i.e., when the operator is placed before the operands.

Examples:

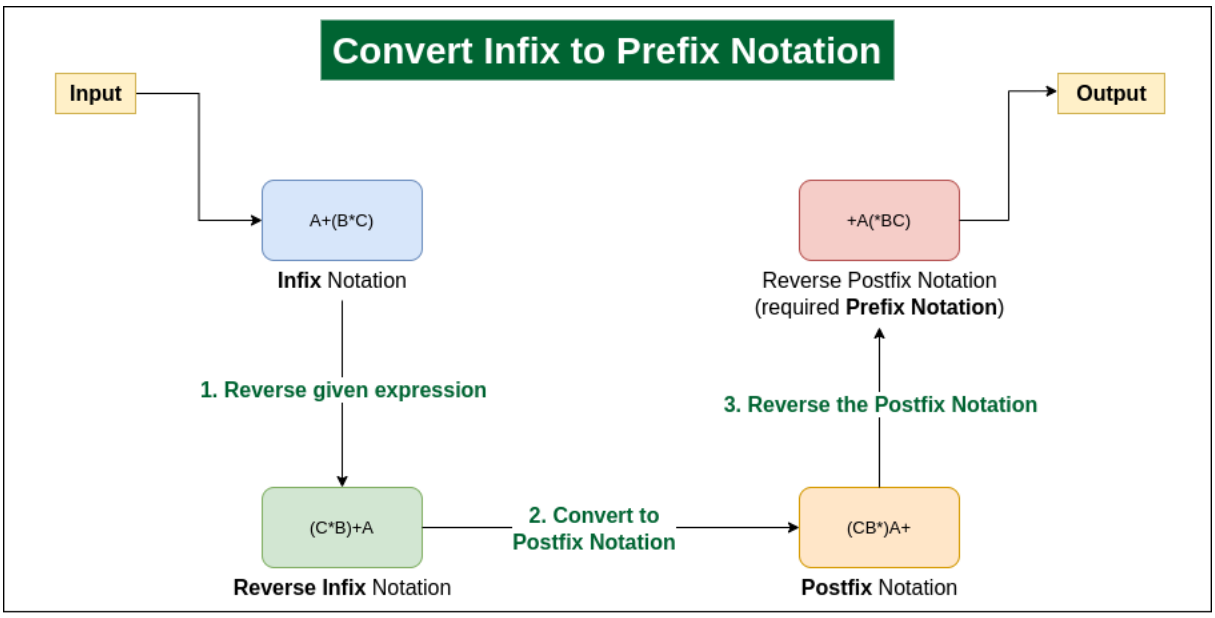
> **Input:** A * B + C / D
> **Output:** + * A B/ C D
>
> **Input:** (A – B/C) * (A/K-L)
> **Output:** *-A/BC-/AKL

How to convert infix expression to prefix expression?

> To convert an infix expression to a prefix expression, we can use the **stack data structure**. The idea is as follows:
>
> - **Step 1:** Reverse the infix expression. Note while reversing each '(' will become ')' and each ')' becomes '('.
> - **Step 2:** Convert the reversed **infix expression to "nearly" postfix expression**.
>
> While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.
> - **Step 3:** Reverse the postfix expression.
>
> The stack is used to convert infix expression to postfix form.

Illustration:

See the below image for a clear idea:



Convert infix expression to prefix expression

Below is the C++ implementation of the algorithm.

C++

```
// C++ program to convert infix to prefix
#include <bits/stdc++.h>
using namespace std;
// Function to check if the character is an operator
bool isOperator(char c)
{
return (!isalpha(c) && !isdigit(c));
}
// Function to get the priority of operators
int getPriority(char C)
{
if (C == '-' || C == '+')
return 1;
else if (C == '*' || C == '/')
return 2;
else if (C == '^')
return 3;
return 0;
}
// Function to convert the infix expression to postfix
string infixToPostfix(string infix)
{
infix = '(' + infix + ')';
int l = infix.size();
stack<char> char_stack;
```

```
string output;
for (int i = 0; i < l; i++) {
// If the scanned character is an
// operand, add it to output.
if (isalpha(infix[i]) || isdigit(infix[i]))
output += infix[i];
// If the scanned character is an
// ‘(‘, push it to the stack.
else if (infix[i] == '(')
char_stack.push('(');
// If the scanned character is an
// ‘)’, pop and output from the stack
// until an ‘(‘ is encountered.
else if (infix[i] == ')') {
while (char_stack.top() != '(') {
output += char_stack.top();
char_stack.pop();
}
// Remove '(' from the stack
char_stack.pop();
}
// Operator found
else {
if (isOperator(char_stack.top())) {
if (infix[i] == '^') {
while (
```

```
getPriority(infix[i])
<= getPriority(char_stack.top())) {
output += char_stack.top();
char_stack.pop();
}
}
else {
while (
getPriority(infix[i])
< getPriority(char_stack.top())) {
output += char_stack.top();
char_stack.pop();
}
}
// Push current Operator on stack
char_stack.push(infix[i]);
}
}
}
while (!char_stack.empty()) {
output += char_stack.top();
char_stack.pop();
}
return output;
}
// Function to convert infix to prefix notation
```

```
string infixToPrefix(string infix)
{
// Reverse String and replace ( with ) and vice versa
// Get Postfix
// Reverse Postfix
int l = infix.size();
// Reverse infix
reverse(infix.begin(), infix.end());
// Replace ( with ) and vice versa
for (int i = 0; i < l; i++) {
if (infix[i] == '(') {
infix[i] = ')';
}
else if (infix[i] == ')') {
infix[i] = '(';
}
}
string prefix = infixToPostfix(infix);
// Reverse postfix
reverse(prefix.begin(), prefix.end());
return prefix;
}
// Driver code
int main()
{
string s = ("x+y*z/w+u");
```

```
// Function call

cout << infixToPrefix(s) << std::endl;

return 0;

}
```

Java

```
// Java program to convert infix to prefix

import java.util.*;

class GFG {

// Function to check if the character is an operand

static boolean isalpha(char c)

{

if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z') {

return true;

}

return false;

}

// Function to check if the character is digit

static boolean isdigit(char c)

{

if (c >= '0' && c <= '9') {

return true;

}

return false;

}

// Function to check if the character is an operator

static boolean isOperator(char c)

{

return (!isalpha(c) && !isdigit(c));

}

// Function to get priority of operators
```

```
static int getPriority(char C)
{
if (C == '-' || C == '+')
return 1;
else if (C == '*' || C == '/')
return 2;
else if (C == '^')
return 3;
return 0;
}
// Reverse the letters of the word
static String reverse(char str[], int start, int end)
{
// Temporary variable to store character
char temp;
while (start < end) {
// Swapping the first and last character
temp = str[start];
str[start] = str[end];
str[end] = temp;
start++;
end--;
}
return String.valueOf(str);
}
// Function to convert infix to postfix expression
```

```
static String infixToPostfix(char[] infix1)

{

String infix = '(' + String.valueOf(infix1) + ')';

int l = infix.length();

Stack<Character> char_stack = new Stack<>();

String output = "";

for (int i = 0; i < l; i++) {

// If the scanned character is an

// operand, add it to output.

if (isalpha(infix.charAt(i))

|| isdigit(infix.charAt(i)))

output += infix.charAt(i);

// If the scanned character is an

// ‘(‘, push it to the stack.

else if (infix.charAt(i) == '(')

char_stack.add('(');

// If the scanned character is an

// ‘)’, pop and output from the stack

// until an ‘(‘ is encountered.

else if (infix.charAt(i) == ')') {

while (char_stack.peek() != '(') {

output += char_stack.peek();

char_stack.pop();

}

// Remove '(' from the stack

char_stack.pop();
```

```
}
// Operator found
else {
if (isOperator(char_stack.peek())) {
while (
(getPriority(infix.charAt(i))
< getPriority(char_stack.peek()))
|| (getPriority(infix.charAt(i))
<= getPriority(
char_stack.peek())
&& infix.charAt(i) == '^')) {
output += char_stack.peek();
char_stack.pop();
}
// Push current Operator on stack
char_stack.add(infix.charAt(i));
}
}
}
while (!char_stack.empty()) {
output += char_stack.pop();
}
return output;
}
static String infixToPrefix(char[] infix)
{
```

```
// Reverse String and replace ( with ) and vice

// versa Get Postfix Reverse Postfix

int l = infix.length;

// Reverse infix

String infix1 = reverse(infix, 0, l - 1);

infix = infix1.toCharArray();

// Replace ( with ) and vice versa

for (int i = 0; i < l; i++) {

if (infix[i] == '(') {

infix[i] = ')';

i++;

}

else if (infix[i] == ')') {

infix[i] = '(';

i++;

}

}

String prefix = infixToPostfix(infix);

// Reverse postfix

prefix = reverse(prefix.toCharArray(), 0, l - 1);

return prefix;

}

// Driver code

public static void main(String[] args)

{

String s = ("x+y*z/w+u");
```

```
// Function call

System.out.print(infixToPrefix(s.toCharArray()));

}

}

// This code is contributed by Rajput-Ji
```

Python3

```
# Python code to convert infix to prefix expression

# Function to check if the character is an operator

def isOperator(c):

return (not c.isalpha()) and (not c.isdigit())

# Function to get the priority of operators

def getPriority(c):

if c == '-' or c == '+':

return 1

elif c == '*' or c == '/':

return 2

elif c == '^':

return 3

return 0


# Function to convert the infix expression to postfix

def infixToPostfix(infix):

infix = '(' + infix + ')'

l = len(infix)

char_stack = []

output = ""

for i in range(l):


# Check if the character is alphabet or digit

if infix[i].isalpha() or infix[i].isdigit():

output += infix[i]
```

```
# If the character is '(' push it in the stack
elif infix[i] == '(':
char_stack.append(infix[i])

# If the character is ')' pop from the stack
elif infix[i] == ')':
while char_stack[-1] != '(':
output += char_stack.pop()
char_stack.pop()

# Found an operator
else:
if isOperator(char_stack[-1]):
if infix[i] == '^':
while getPriority(infix[i]) <= getPriority(char_stack[-1]):
output += char_stack.pop()
else:
while getPriority(infix[i]) < getPriority(char_stack[-1]):
output += char_stack.pop()
char_stack.append(infix[i])
while len(char_stack) != 0:
output += char_stack.pop()
return output
# Function to convert infix expression to prefix
def infixToPrefix(infix):
```

```
l = len(infix)

infix = infix[::-1]

for i in range(l):

if infix[i] == '(':

infix[i] = ')'

elif infix[i] == ')':

infix[i] = '('

prefix = infixToPostfix(infix)

prefix = prefix[::-1]

return prefix

# Driver code

if __name__ == '__main__':

s = "x+y*z/w+u"


# Function call

print(infixToPrefix(s))
```

C#

```
// C# program to convert infix to prefix
using System;
using System.Collections.Generic;
public class GFG {
// Check if the given character is an alphabet
static bool isalpha(char c)
{
if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z') {
return true;
}
return false;
}
// Check if the current character is a digit
static bool isdigit(char c)
{
if (c >= '0' && c <= '9') {
return true;
}
return false;
}
// Function to check if the character is an operator
static bool isOperator(char c)
{
return (!isalpha(c) && !isdigit(c));
}
```

```
// Function to get the precedence order of operators
static int getPriority(char C)
{
if (C == '-' || C == '+')
return 1;
else if (C == '*' || C == '/')
return 2;
else if (C == '^')
return 3;
return 0;
}
// Reverse the letters of the word
static String reverse(char[] str, int start, int end)
{
// Temporary variable to store character
char temp;
while (start < end) {
// Swapping the first and last character
temp = str[start];
str[start] = str[end];
str[end] = temp;
start++;
end--;
}
return String.Join("", str);
}
```

```
// Function to convert infix to postfix notation

static String infixToPostfix(char[] infix1)

{

String infix = '(' + String.Join("", infix1) + ')';

int l = infix.Length;

Stack<char> char_stack = new Stack<char>();

String output = "";

for (int i = 0; i < l; i++) {

// If the scanned character is an

// operand, add it to output.

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

output += infix[i];

// If the scanned character is an

// ‘(‘, push it to the stack.

else if (infix[i] == '(')

char_stack.Push('(');

// If the scanned character is an

// ‘)’, pop and output from the stack

// until an ‘(‘ is encountered.

else if (infix[i] == ')') {

while (char_stack.Peek() != '(') {

output += char_stack.Peek();

char_stack.Pop();

}

// Remove '(' from the stack

char_stack.Pop();
```

```
}
// Operator found
else {
if (isOperator(char_stack.Peek())) {
while (
(getPriority(infix[i])
< getPriority(char_stack.Peek()))
|| (getPriority(infix[i])
<= getPriority(
char_stack.Peek())
&& infix[i] == '^')) {
output += char_stack.Peek();
char_stack.Pop();
}
// Push current Operator on stack
char_stack.Push(infix[i]);
}
}
}
while (char_stack.Count != 0) {
output += char_stack.Pop();
}
return output;
}
// Driver code
static String infixToPrefix(char[] infix)
```

```
{
// Reverse String Replace ( with ) and vice versa
// Get Postfix
// Reverse Postfix *
int l = infix.Length;
// Reverse infix
String infix1 = reverse(infix, 0, l - 1);
infix = infix1.ToCharArray();
// Replace ( with ) and vice versa
for (int i = 0; i < l; i++) {
if (infix[i] == '(') {
infix[i] = ')';
i++;
}
else if (infix[i] == ')') {
infix[i] = '(';
i++;
}
}
String prefix = infixToPostfix(infix);
// Reverse postfix
prefix = reverse(prefix.ToCharArray(), 0, l - 1);
return prefix;
}
// Driver code
public static void Main(String[] args)
```

```
{

String s = ("x+y*z/w+u");


// Function call

Console.Write(infixToPrefix(s.ToCharArray()));

}

}

// This code is contributed by gauravrajput1
```

Javascript

```
// Javascript program to convert infix to prefix
// program to implement stack data structure
class Stack {
constructor() {
this.items = [];
}
// add element to the stack
push(element) {
return this.items.push(element);
}
// remove element from the stack
pop() {
if (this.items.length > 0) {
return this.items.pop();
}
}
// view the last element
top() {
return this.items[this.items.length - 1];
}
// check if the stack is empty
isEmpty() {
return this.items.length == 0;
}
// the size of the stack
```

```
size() {
return this.items.length;
}
// empty the stack
clear() {
this.items = [];
}
}
function isalpha(c) {
if ((c >= "a" && c <= "z") || (c >= "A" && c <= "Z")) {
return true;
}
return false;
}
function isdigit(c) {
if (c >= "0" && c <= "9") {
return true;
}
return false;
}
function isOperator(c) {
return !isalpha(c) && !isdigit(c);
}
function getPriority(C) {
if (C == "-" || C == "+") return 1;
else if (C == "*" || C == "/") return 2;
```

```
else if (C == "^") return 3;

return 0;

}

function infixToPostfix(infix) {

infix = "(" + infix + ")";

var l = infix.length;

let char_stack = new Stack();

var output = "";

for (var i = 0; i < l; i++) {

// If the scanned character is an

// operand, add it to output.

if (isalpha(infix[i]) || isdigit(infix[i])) output += infix[i];

// If the scanned character is an

// ‘(‘, push it to the stack.

else if (infix[i] == "(") char_stack.push("(");

// If the scanned character is an

// ‘)’, pop and output from the stack

// until an ‘(‘ is encountered.

else if (infix[i] == ")") {

while (char_stack.top() != "(") {

output += char_stack.top();

char_stack.pop();

}

// Remove '(' from the stack

char_stack.pop();

}
```

```
// Operator found
else {
if (isOperator(char_stack.top())) {
if (infix[i] == "^") {
while (getPriority(infix[i]) <= getPriority(char_stack.top())) {
output += char_stack.top();
char_stack.pop();
}
} else {
while (getPriority(infix[i]) < getPriority(char_stack.top())) {
output += char_stack.top();
char_stack.pop();
}
}
// Push current Operator on stack
char_stack.push(infix[i]);
}
}
}
while (!char_stack.isEmpty()) {
output += char_stack.top();
char_stack.pop();
}
return output;
}
function infixToPrefix(infix) {
```

```
/* Reverse String
* Replace ( with ) and vice versa
* Get Postfix
* Reverse Postfix  *  */
var l = infix.length;
// Reverse infix
infix = infix.split("").reverse().join("");
// Replace ( with ) and vice versa
var infixx = infix.split("");
for (var i = 0; i < l; i++) {
if (infixx[i] == "(") {
infixx[i] = ")";
} else if (infixx[i] == ")") {
infixx[i] = "(";
}
}
infix = infixx.join("");
var prefix = infixToPostfix(infix);
// Reverse postfix
prefix = prefix.split("").reverse().join("");
return prefix;
}
// Driver code
var s = "x+y*z/w+u";
console.log(infixToPrefix(s));
```

Output

```
++x/*yzwu
```

Complexity Analysis:

- **Time Complexity:** O(n)
  - Stack operations like push() and pop() are performed in constant time.
  - Since we scan all the characters in the expression once the complexity is linear in time
- **Auxiliary Space**: O(n) because we are keeping a stack.

S

Sayan Mahapatra


Improve

Similar Reads

- Stack Data Structure

  A Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first

  3 min read

- What is Stack Data Structure? A Complete Tutorial

  Stack is a linear data structure that follows LIFO (Last In First Out) Principle, the last element inserted is the first to be popped out. It means both insertion and deletion operations happen at one end only. LIFO(Last In First Out) PrincipleHere are some real world examples of LIFO Consider a sta

  4 min read

- Applications, Advantages and Disadvantages of Stack

  A stack is a linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack. Applications of Stacks:Function calls: Stacks are used to keep track of the return addresses of function calls, allowing the

  2 min read

- Implement a stack using singly linked list

  To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO(last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list. So we need to follow a simple rul

  15+ min read

- Introduction to Monotonic Stack - Data Structure and Algorithm Tutorials

  A monotonic stack is a special data structure used in algorithmic problem-solving. Monotonic Stack maintaining elements in either increasing or decreasing order. It is commonly used to efficiently solve problems such as finding the next greater or smaller element in an array etc. Table of Content Wh

  12 min read

- Difference Between Stack and Queue Data Structures

  In computer science, data structures are fundamental concepts that are crucial for organizing and storing data efficiently. Among the various data structures, stacks and queues are two of the most basic yet essential structures used in programming and algorithm design. Despite their simplicity, they

  4 min read

Stack implementation in different language

Some questions related to Stack implementation

Intermediate problems on Stack

Hard problems on Stack

Practice Tags :

- Stack
- Strings