Infix, Prefix and Postfix Expressions

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-35

We have finished learning matching parentheses in the last tutorial. It is always great to see the applications of what you learn, and parentheses matching was one such application of stacks. Today we'll start another one, called infix, prefix, and postfix expressions.

What are these?

The three terms, infix prefix, and postfix will be dealt with individually later. In general, these are **the notations to write an expression**. Mathematical expressions have been taught to us since childhood. Writing expressions to add two numbers for subtraction, multiplication, or division. They were all expressed through certain expressions. That's what we're learning today: different expressions.

Infix:

This is the method we have all been studying and applying for all our academic life. Here the operator comes in between two operands. And we say, two is added to three. For eg: 2 + 3, a * b, 6 / 3 etc.

< operand 1 >< **operator** >< operand2 >

Prefix:

This method might seem new to you, but we have vocally used them a lot as well. Here the operator comes before the two operands. And we say, Add two and three. For e.g.: + 6 8, * x y, - 3 2 etc.

< **operator** >< operand 1 >< operand2 >

Postfix:

This is the method that might as well seem new to you, but we have used even this in our communication. Here the operator comes after the two operands. And we say, Two and three are added. For e.g.: 5 7 +, a b *, 12 6 / etc.

< operand 1 >< operand2 >< **operator** >

To understand the interchangeability of these terms, please refer to the table below.

	Infix	Prefix	Postfix
1.	a * b	* a b	a b *

2.	a - b	- a b	a b -

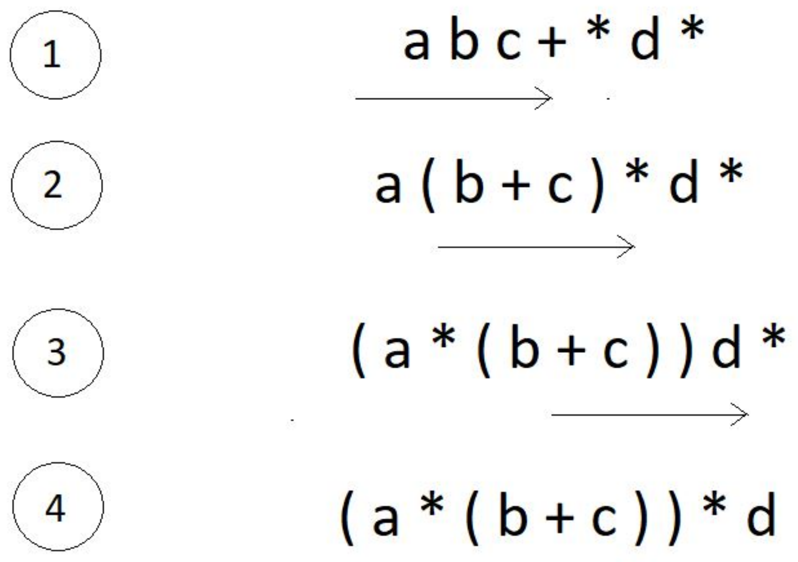
So far, we have been dealing with just two operands, but a mathematical expression can hold a lot more. We will now learn to change a general infix mathematical expression to its prefix and postfix relatives. But before that, it is better to understand why we even need these methods.

Why these methods?

When we evaluate a mathematical expression, we have a rule in mind, named BODMAS, where we have operators' precedence in this order; brackets, of, division, multiplication, addition, subtraction. But what would you do when you get to evaluate a 1000 character long-expression, or even longer one? You will try to automate the process. But there is one issue. Computers don't follow BODMAS; rather, they have their own operator precedence. And this is where we need these postfix and prefix notations. In programming, we use postfix notations more often, likewise, following the precedence order of machines.

Consider the expression a* (b + c) * d; since computers go left to right while evaluating an expression, we'll convert this infix expression to its postfix form.

Its postfix form is, a b c + * d *. You must be wondering how we got here. Refer to the illustration below.



We have successfully reached what we wanted the machine to do. Now the kick is in converting infixes to postfixes and prefixes.

Converting infix to prefix:

Consider the expression, **x - y * z**.

1. Parentheses the expression. The infix expression must be parenthesized by following the operator precedence and associativity before converting it into a prefix expression. Our expression now becomes **(x - (y * z))**.

2. Reach out to the innermost parentheses. And convert them into prefix first, i.e. **(x - (y * z))** changes to **(x - [* y z])**.

3. Similarly, keep converting one by one, from the innermost to the outer parentheses. **(x - [* y z]) → [- x * y z].**

4. And we are done.

Converting infix to postfix:

Consider the same expression, **x - y * z**.

5. Parentheses the expression as we did previously. Our expression now becomes **(x - (y * z))**.

6. Reach out to the innermost parentheses. And convert them into postfix first, i.e. **(x - (y * z))** changes to **(x - [y z *])**.

7. Similarly, keep converting one by one, from the innermost to the outer parentheses. **(x - [y z *]) → [x y z * -].**

8. And we are done.

Similarly the expression p - q - r / a, follows the following conversions to become a prefix expression:

p - q - r / a → ((p - q) - (r / a)) → ([- p q] - [/ r a]) → - - p q / r a

Quick Quiz: Convert the above infix expression into its postfix form.

Note: You cannot change the expression given to you. For eg. (p - q) * (m - n) cannot be changed to something like (p - (q * m) - n).

Let’s change this to its postfix equivalent.

(p - q) * (m - n) → ((p - q) * (m - n)) → ([p q -] *[m n -])* → ***p q-m n -***

We didn’t go to the programming part here. It was intended to be as simple as possible. I hope you understood everything. We’ll see automating this process in our next tutorial. Stay connected.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven’t checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we'll learn coding to achieve the conversions we talked about today. Till then, keep coding.