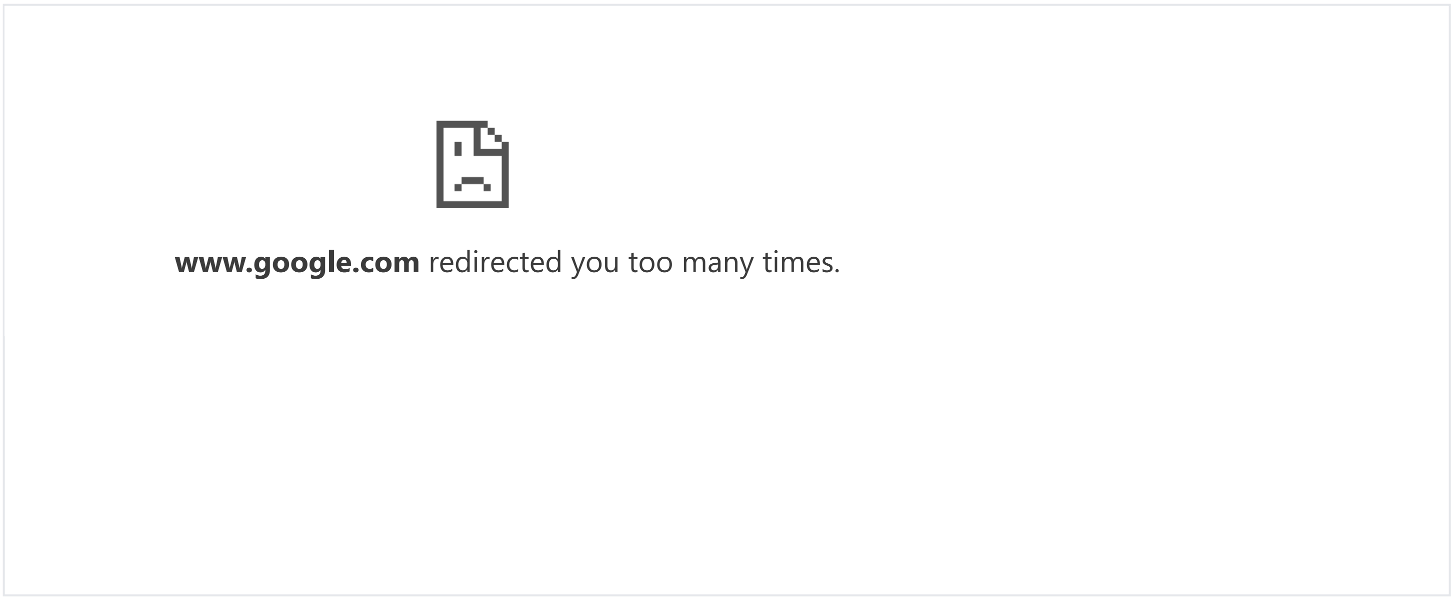


www.google.com redirected you too many times.

Show Course Contents ⊕

Overview Q&A Downloads Announcements

Infix To Postfix Using Stack

In the last tutorial, we had learned to convert an infix expression to its postfix and prefix equivalents manually. Following were the simple steps we followed.

1. Parenthesize the expression following the operators' precedence and their associativity.
2. From the innermost to outermost, keep converting the expressions.

But we didn't talk about their implementation using stacks; rather, we didn't even mention stacks in our last class. Today, we will learn how to convert an infix expression into its postfix equivalent using stacks.

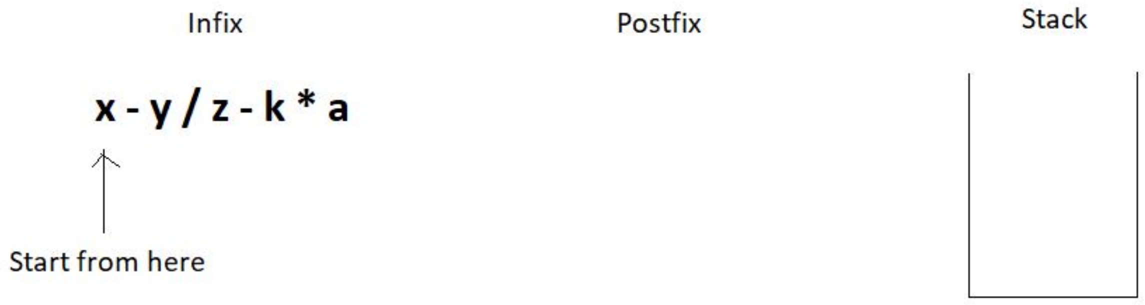
Converting an infix expression to its postfix counterpart needs you to follow certain steps. The following are the steps:

1. Start moving left to right from the beginning of the expression.
2. The moment you receive an operand, concatenate it to the postfix expression string.

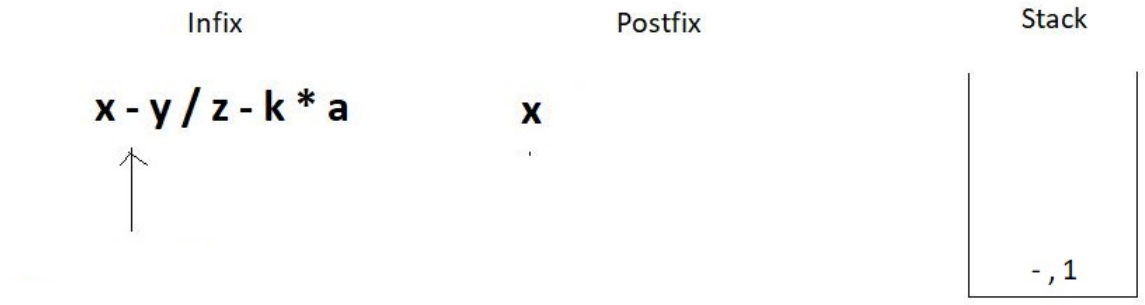
3. And the moment you encounter an operator, move to the stack along with its relative precedence number and see if the topmost operator in the stack has higher or lower precedence. If it's lower, push this operator inside the stack. Else, keep popping operators from the stack and concatenate it to the postfix expression until the topmost operator becomes weaker in precedence relative to the current operator.
4. If you reach the EOE, pop every element from the stack, and concatenate them as well. And the expression you will receive after doing all the steps will be the postfix equivalent of the expression we were given.

For our understanding today, let us consider the expression **x - y / z - k * a.** Step by step, we will turn this expression into its postfix equivalent using stacks.

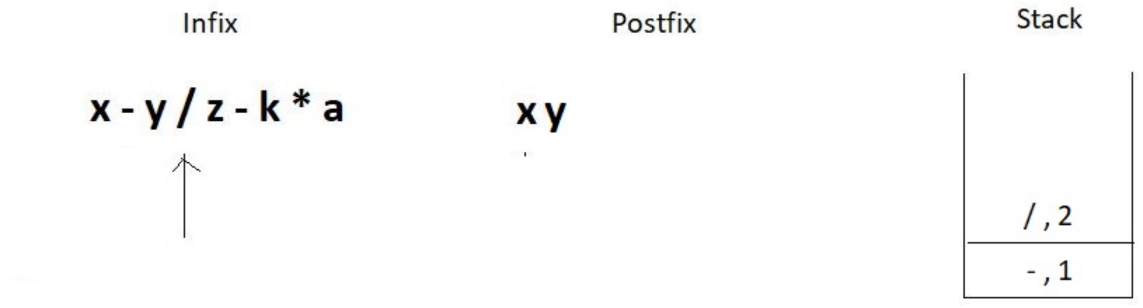
1. We will start traversing from the left.



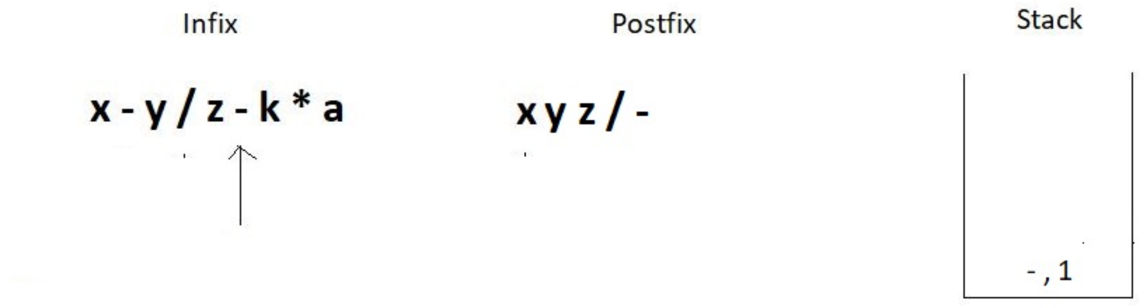
2. First, we got the letter 'x'. We just pushed it into the postfix string. Then we got the subtraction symbol '-', and we push it into the stack since the stack is empty.



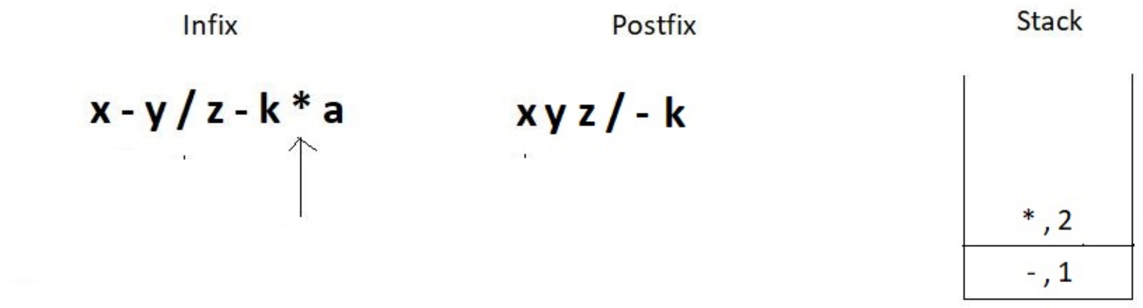
3. Similarly, we push the division operator in the stack since the topmost operator has a precedence number 1, and the division has 2.



4. The next operator we encounter is again a subtraction. Since the topmost operator in the stack has an operator precedence number 2, we would pop elements out from the stack until we can push the current operator. This leads to removing both the present operators in the stack since they are both greater or equal in precedence. Don’t forget to concatenate the popped operators to the postfix expression.

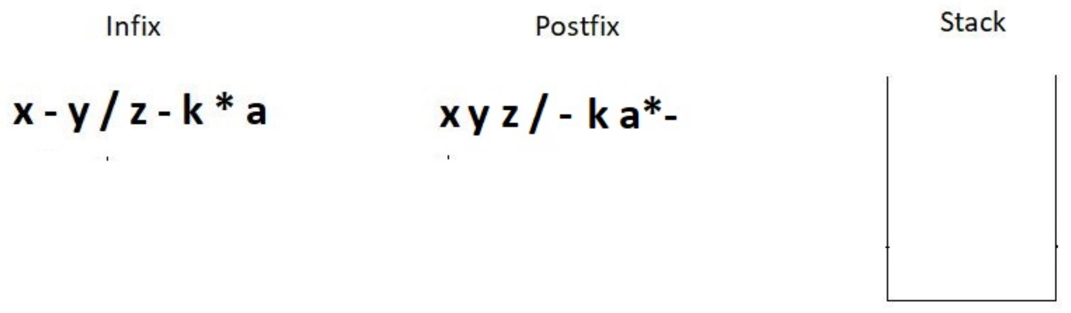


5. Next, we have a multiplication operator whose precedence number is 2 relative to the topmost operator in the stack. Hence we simply push it in the stack.



6. And then we get to the EOE and still have two elements inside the stack. So, just pop them one by one, and concatenate them to the postfix. And this is when we

succeed in converting the infix to the postfix expression.



Follow every step meticulously, and you will find it very easy to master this. You can see if the answer we found at the end is correct manually.

- **x - y / z - k * a** → ((x - (y / z)) - (k * a)) → ((x - [y z /]) - [k a *]) → [x y z / -] - [k

This was visualizing the conversion process of infix to postfix using stacks. We will see the programming part in the next tutorial. It would be best if you practiced converting a few expressions of your own. If you still feel diffident about using stacks, you must check out the previous videos where we discussed stacks in detail.
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 see the program to fetch these conversions from infix to postfix. Till then, keep coding.

Previous Next