Infix to Postfix

Write a program to Convert Infix expression to Postfix.

> **Infix expression:** The expression of the form a operator b (a + b). When an operator is in-between every pair of operands.
> **Postfix expression:** The expression of the form a b operator (ab+). When an operator is followed by every pair of operands.

Examples:

> **Input:** A + B * C + D
> **Output:** ABC*+D+
>
> **Input:** ((A + B) - C * (D / E)) + F
> **Output:** AB+CDE/*-F+

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.
Consider the expression: **a + b * c + d**

The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it. The result is then added to d after another scan. The repeated scanning makes
are easily readable and solvable by humans whereas the computer can
parenthesis easily so, it is better to convert the expression to postfix(or prefix) for
The corresponding expression in postfix form is **abc*+d+**. The postfix expressions can be evaluated easily using a stack.

Steps to convert Infix expression to Postfix expression using Stack:

- Scan the infix expression from left to right.
- If the scanned character is an operand, output it.
- Else,
  - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack(or the stack is empty or the stack contains a '('), then push it.
  - '^' operator is right associative and other operators like '+','-','*' and '/' are left-associative. Check especially for a condition when both, operator at the top of the stack and the scanned operator are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack. In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
  - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
- If the scanned character is an '(', push it to the stack.
- If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
- Repeat steps 2-6 until the infix expression is scanned.
- Print the output

- Pop and output from the stack until it is not empty.

Below is the implementation of the above algorithm:

C++ Java

```
/* Java implementation to convert
 infix expression to postfix*/
// Note that here we use ArrayDeque class for Stack
// operations

import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Stack;

class Test {

    // A utility function to return
    // precedence of a given operator
    // Higher returned value means
    // higher precedence
    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.
    static String infixToPostfix(String exp)
    {
        // initializing empty String for result
        String result = new String("");

        // initializing empty stack
        Deque<Character> stack
            = new ArrayDeque<Character>();

        for (int i = 0; i < exp.length(); ++i) {
            char c = exp.charAt(i);

            // If the scanned character is an
            // operand, add it to output.
            if (Character.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.isEmpty()
                       && stack.peek() != '(') {
                    result += stack.peek();
                    stack.pop();
                }

                stack.pop();
            }
            else // an operator is encountered
            {
                while (!stack.isEmpty()
                       && Prec(c) <= Prec(stack.peek())) {

                    result += stack.peek();
                    stack.push(c);
                }
            }
```

```
        // pop all the operators from the stack
        while (!stack.isEmpty()) {
            if (stack.peek() == '(')
                return "Invalid Expression";
            result += stack.peek();
            stack.pop();
        }

        return result;
    }

    // Driver's code
    public static void main(String[] args)
    {
        String exp = "a+b*(c^d-e)^(f+g*h)-i";

          // Function call
        System.out.println(infixToPostfix(exp));
    }
}
```

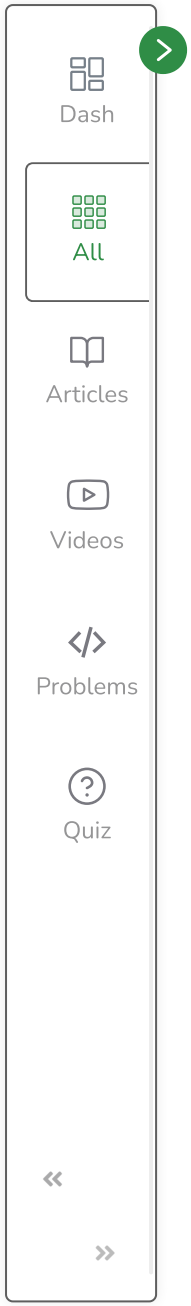

Output

```
abcd^e-fgh*+^*+i-
```

Time Complexity: O(N), where N is the size of the infix expression
Auxiliary Space: O(N)

Marked as Read

Article marked as read.



Report An Issue

If you are facing any issue on this page. Please let us know.