

Infix to Postfix Conversion Using Shunting Yard Algorithm

Mr. Vikas Kumar

Assistant Professor

Industry Embedded Program

Lesson Plan

Subject/Course	Competitive Coding
Lesson Title	Infix to Postfix Conversion Using Shunting Yard Algorithm
Lesson Objectives	
Understand infix, prefix, and postfix notations and their role in expression evaluation.	
Implement the Shunting Yard Algorithm to convert infix to postfix using a stack for operators and a queue for output.	
Handle operator precedence, associativity, and parentheses correctly.	
Apply this to real-world scenarios like compiler design and competitive problems.	

Problem Statement:

Write a Program for an infix expression, and convert it to postfix notation. Use a queue to implement the Shunting Yard Algorithm for expression conversion.

Given an infix expression (e.g., "A + B * C"), convert it to postfix (Reverse Polish Notation, e.g., "A B C * +") for easier evaluation without parentheses.

Input (Infix)	Output (Postfix)	Explanation
A + B * C	A B C * +	* has higher precedence than +.
(A + B) * C	A B + C *	Parentheses group A + B first.
A ^ B ^ C	A B C ^ ^	^ is right-associative.

Key Concepts & Prerequisites

Infix Expression: Operators appear between operands.

Example: $A + B * C$

Problem: Harder for computers to evaluate due to precedence rules.

Postfix Expression (Reverse Polish Notation): Operators come *after* operands.

Example: $A B C * +$

Advantage:

- No need for parentheses.
- Evaluation becomes simpler using a stack.

Key Concepts & Prerequisites

Shunting Yard Algorithm:

Uses:

- **Stack** → For operators and parentheses.
- **Queue** → For the final output (postfix form).

Prerequisites:

- Basic stack/queue operations
- Understanding of operator precedence (e.g., $*$ $>$ $+$) and
- associativity (left/right).

Algorithm/Logic

High-Level Approach

1. Scan input left-to-right.
2. Output operands immediately to queue.
3. Push '(' to stack; Pop until matching ')' for parens.
4. For operators: Pop lower/equal precedence from stack to queue, then push current.
5. At end, pop all stack to queue.

Algorithm/Logic

Detailed Algorithm

1. Initialize an empty stack (for operators) and an empty queue (for output).
2. Read tokens (operands/operators/parentheses) from left to right.
3. If token is operand → add to queue (output).
4. If token is operator (op1) →
 - While there is an operator (op2) at the top of the stack with greater or equal precedence, pop op2 and add to queue.
 - Push op1 onto the stack.

Algorithm/Logic

Detailed Algorithm

1. If token is '(' → push to stack.
2. If token is ')' → pop operators from stack to queue until '(' is found; discard both parentheses.
3. After reading all tokens → pop all remaining operators from stack to queue.

Operator Precedence:

^ (highest), *, /, then +, -

Visualization

Step	Input Token	Stack (Operators)	Queue (Postfix Output)
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6	End		A B C * +

Final Output:

☐ Postfix → **A B C * +**

Code Implementation

```
1  import java.util.Stack;
2
3  public class Practical5_InfixToPostfix {
4
5      static int precedence(char c) {
6          switch (c) {
7              case '+': case '-': return 1;
8              case '*': case '/': return 2;
9              case '^': return 3;
10         }
11         return -1;
12     }
13
14     static String convert(String exp) {
15         StringBuilder result = new StringBuilder();
16         Stack<Character> stack = new Stack<>();
```

```
17  ▼   for (char c : exp.toCharArray()) {
18      if (Character.isLetterOrDigit(c))
19          result.append(c);
20      else if (c == '(')
21          stack.push(c);
22  ▼   else if (c == ')') {
23          while (!stack.isEmpty() && stack.peek() != '(')
24              result.append(stack.pop());
25          stack.pop();
26  ▼   } else {
27          while (!stack.isEmpty() && precedence(c) <= precedence(stack.peek()))
28              result.append(stack.pop());
29          stack.push(c);
30      }
31  }
32  while (!stack.isEmpty())
33      result.append(stack.pop());
34  return result.toString();
35  }
```

```
36  ▾ public static void main(String[] args) {  
37      String exp = "a+b*(c^d-e)^(f+g*h)-i";  
38      System.out.println("Infix: " + exp);  
39      System.out.println("Postfix: " + convert(exp));  
40  }  
41 }
```

Output :

Infix: $a+b*(c^d-e)^{(f+g*h)}-i$

Postfix: $abcd^e-fgh*+^*+i-$

Complexity Analysis

Time Complexity: $O(n)$ — each token processed once

Space Complexity: $O(n)$ — for stack + queue

Variations & Competitive Tips

- Easier Variation: No associativity (assume all left).
- Harder: Add unary operators (e.g., -A); Handle variables/functions
- Competitive Angle: LeetCode #159: Expression Add Operators; Codeforces problems on parsers.
- Optimization: Use array for stack (faster in C++); Extend to full evaluator.

Practice Exercises

1 Evaluate Reverse Polish Notation — LeetCode #150

↪ <https://leetcode.com/problems/evaluate-reverse-polish-notation/>

Concept: Postfix (RPN) evaluation using stack operations.

Why Practice: It directly applies the postfix conversion and evaluation logic you learned.

Practice Exercises

2 Basic Calculator II — LeetCode #227

↪ <https://leetcode.com/problems/basic-calculator-ii/>

Concept: Stack-based infix expression evaluation with $+$ $-$ $*$ $/$.

Why Practice: Reinforces understanding of operator precedence and stack evaluation.

Practice Exercises

3 Basic Calculator — LeetCode #224

↪ <https://leetcode.com/problems/basic-calculator/Concept: Evaluate>

arithmetic expressions with parentheses. Why Practice: Strengthens your ability to parse infix expressions with nested operations.

Thanks