To evaluate an expression in postfix notation (Reverse Polish Notation), we can use a **stack**. The basic idea is to process the expression from left to right, and for each symbol, do the following:

1. **If the symbol is a number**, push it onto the stack.
2. **If the symbol is an operator** (e.g., +, -, \*, /), pop the necessary number of operands from the stack (in this case, two operands), apply the operator, and then push the result back onto the stack.
3. At the end of the expression, the stack will contain exactly one element, which is the result of the expression.

Here's a non-recursive algorithm for evaluating a postfix expression:

**Steps:**

1. **Initialize** an empty stack.
2. **For each token** in the postfix expression:
   * If it's a number, **push** it onto the stack.
   * If it's an operator, **pop** the two top elements from the stack, apply the operator, and **push** the result back onto the stack.
3. **After processing all tokens**, the result will be the only element left in the stack.

**Example:**

For the postfix expression 5 2 + 8 3 - \* 4 /, here’s the step-by-step breakdown:

1. Push 5 → Stack: [5]
2. Push 2 → Stack: [5, 2]
3. Apply +: Pop 2 and 5, compute 5 + 2 = 7, and push 7 → Stack: [7]
4. Push 8 → Stack: [7, 8]
5. Push 3 → Stack: [7, 8, 3]
6. Apply -: Pop 3 and 8, compute 8 - 3 = 5, and push 5 → Stack: [7, 5]
7. Apply \*: Pop 5 and 7, compute 7 \* 5 = 35, and push 35 → Stack: [35]
8. Push 4 → Stack: [35, 4]
9. Apply /: Pop 4 and 35, compute 35 / 4 = 8.75 (assuming floating-point division), and push 8.75 → Stack: [8.75]
10. Final result: 8.75 (the only element left in the stack).

**Java Implementation:**

import java.util.Stack;

public class PostfixEvaluation {

// Method to evaluate a postfix expression

public static double evaluatePostfix(String expression) {

// Stack to store operands

Stack<Double> stack = new Stack<>();

// Split the expression into tokens (assuming space-separated tokens)

String[] tokens = expression.split(" ");

// Iterate over each token

for (String token : tokens) {

// If the token is an operator

if (isOperator(token)) {

// Pop two operands from the stack

double operand2 = stack.pop();

double operand1 = stack.pop();

// Apply the operator and push the result back onto the stack

double result = applyOperator(operand1, operand2, token);

stack.push(result);

} else {

// If the token is a number, push it onto the stack

stack.push(Double.parseDouble(token));

}

}

// The result will be the only element left in the stack

return stack.pop();

}

// Method to check if a string is an operator

private static boolean isOperator(String token) {

return token.equals("+") || token.equals("-") || token.equals("\*") || token.equals("/");

}

// Method to apply an operator to two operands

private static double applyOperator(double operand1, double operand2, String operator) {

switch (operator) {

case "+":

return operand1 + operand2;

case "-":

return operand1 - operand2;

case "\*":

return operand1 \* operand2;

case "/":

return operand1 / operand2;

default:

throw new IllegalArgumentException("Invalid operator: " + operator);

}

}

public static void main(String[] args) {

String expression = "5 2 + 8 3 - \* 4 /";

double result = evaluatePostfix(expression);

System.out.println("Result: " + result);

}

}

**Explanation of the Code:**

1. **evaluatePostfix(String expression)**:
   * The expression is split into tokens using split(" ") to handle space-separated values.
   * For each token, if it's a number, it is pushed onto the stack. If it's an operator, the top two numbers are popped from the stack, the operator is applied, and the result is pushed back onto the stack.
2. **isOperator(String token)**:
   * Checks if the token is one of the valid operators (+, -, \*, /).
3. **applyOperator(double operand1, double operand2, String operator)**:
   * Applies the given operator to the operands and returns the result.

**Example Output:**

For the postfix expression "5 2 + 8 3 - \* 4 /", the output will be:

Result: 8.75

**Time Complexity:**

* The time complexity is **O(n)**, where n is the number of tokens in the expression, because each token is processed once.