Practical 5

Aim : 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.

Algorithm :

1. **Initialize Data Structures**:
   - Create an empty stack for operators.
   - Create an empty queue for the output.
2. **Read the Infix Expression**:
   - Read the infix expression character by character.
3. **Process Each Character**:
   - For each character **token** in the infix expression:
     - **If token is an operand (number or variable)**:
       - Add **token** to the output queue.
     - **If token is a left parenthesis (**:
       - Push **token** onto the stack.
     - **If token is a right parenthesis)**:
       - While the stack is not empty and the top of the stack is not a left parenthesis **(**:
         - Pop the top of the stack and add it to the output queue.
       - Pop the left parenthesis **(** from the stack (do not add it to the output queue).
     - **If token is an operator (e.g., +, -, *, /, ^)**:
       - While the stack is not empty and the precedence of the operator at the top of the stack is greater than or equal to the precedence of **token**:

    - Pop the top of the stack and add it to the output queue.
  - Push **token** onto the stack.

4. **Pop Remaining Operators**:
   - After reading all characters in the infix expression, while the stack is not empty:
     - Pop the top of the stack and add it to the output queue.
5. **Build the Postfix Expression**:
   - The output queue now contains the postfix expression. Convert the queue to a string format (if necessary) and return it.

Example

Let's take an example **(5 * 4 + 3) - 1**

Step-by-Step Conversion Using the Shunting Yard Algorithm

1. **Initialize Data Structures**:
   - **Stack**: Empty (for operators)
   - **Queue**: Empty (for output)
2. **Process Each Character**:
   - Read the expression character by character.

Character Processing

- **Read (**:
  - Push **(** onto the stack.
  - **Stack**: **(**
  - **Queue**: (empty)
- **Read 5**:
  - Add **5** to the output queue.
  - **Stack**: **(**
  - **Queue**: **5**

- **Read ***:
    - Push ***** onto the stack.
    - **Stack**: **(***
    - **Queue**: **5**
- **Read 4**:
    - Add **4** to the output queue.
    - **Stack**: **(***
    - **Queue**: **5 4**
- **Read +**:
    - Pop ***** from the stack to the output queue (since ***** has higher precedence than **+**).
    - Push **+** onto the stack.
    - **Stack**: **(+**
    - **Queue**: **5 4 ***
- **Read 3**:
    - Add **3** to the output queue.
    - **Stack**: **(+**
    - **Queue**: **5 4 * 3**
- **Read)**:
    - Pop from the stack to the output queue until **(** is found.
    - Pop **+** and add it to the output queue.
    - Pop **(** (do not add it to the output queue).
    - **Stack**: (empty)
    - **Queue**: **5 4 * 3 +**
- **Read -**:
    - Push **-** onto the stack.
    - **Stack**: **-**

    - **Queue**: **5 4 * 3 +**
- **Read 1**:
    - Add **1** to the output queue.
    - **Stack**: **-**
    - **Queue**: **5 4 * 3 + 1**

Final Steps

- After processing all characters, pop any remaining operators from the stack to the output queue.
- Pop **-** from the stack and add it to the output queue.
- **Stack**: (empty)
- **Queue**: **5 4 * 3 + 1 -**

Code:-

```
import java.util.*;


public class InfixToPostfix {


    // Method to determine the precedence of operators
    private static int precedence(char operator) {
        switch (operator) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
```

```
            return 2;
        case '^':
            return 3;
        default:
            return 0;
    }
}

// Method to convert infix expression to postfix
public static String infixToPostfix(String infix) {
    Stack<Character> stack = new Stack<>();
    Queue<String> outputQueue = new LinkedList<>();

    for (int i = 0; i < infix.length(); i++) {
        char token = infix.charAt(i);

        // If the token is an operand, add it to the output queue
        if (Character.isLetterOrDigit(token)) {
            outputQueue.add(String.valueOf(token));
        }
        // If the token is '(', push it onto the stack
        else if (token == '(') {
            stack.push(token);
        }
        // If the token is ')', pop from stack to output queue until '(' is found
        else if (token == ')') {
            while (!stack.isEmpty() && stack.peek() != '(') {
                outputQueue.add(String.valueOf(stack.pop()));
            }
            stack.pop(); // Pop the '(' from the stack
        }
        // If the token is an operator
        else {
            while (!stack.isEmpty() && precedence(stack.peek()) >= precedence(token)) {
                outputQueue.add(String.valueOf(stack.pop()));
            }
            stack.push(token);
        }
    }

    // Pop all the operators from the stack to the output queue
    while (!stack.isEmpty()) {
        outputQueue.add(String.valueOf(stack.pop()));
    }

    // Build the final postfix expression
    StringBuilder postfix = new StringBuilder();
    while (!outputQueue.isEmpty()) {
        postfix.append(outputQueue.poll()).append(" ");
    }

    return postfix.toString().trim(); // Return the postfix expression
```

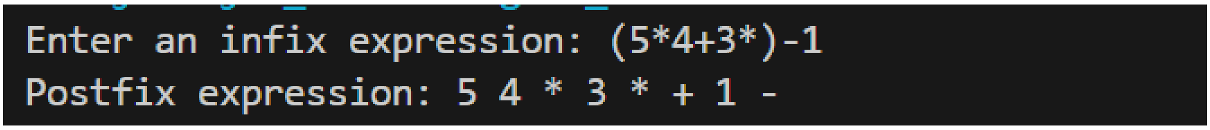
```
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter an infix expression: ");
        String infix = scanner.nextLine();

        String postfix = infixToPostfix(infix);
        System.out.println("Postfix expression: " + postfix);

        scanner.close();
    }
}
```

Output:-



```
Enter an infix expression: (5*4+3*)-1
Postfix expression: 5 4 * 3 * + 1 -
```