| ESPE UNIVERSIDAD DE LAS FUERZAS ARMADAS INNOVACIÓN PARA LA EXCELENCIA | INFORME DE PRÁCTICA DE LABORATORIO |
|---|---|

| | | | | | |
|---|---|---|---|---|---|
| **DEPARTAMENTO:** | **Ciencias de la Computación** | **CARRERA:** | | | |
| **ASIGNATURA:** | Estructura de datos | **NIVEL:** | 3 | **FECHA:** | 11-01-2025 |
| **DOCENTE:** | Margoth Guaraca | **PRÁCTICA N°:** | 2 | **CALIFICACIÓN:** | |

# Expression Converter and Evaluator using Custom Stacks

**Angel Steven Rodriguez Chavez**

**Summary**

Knowledge of linear data structures (stacks) was applied to solve the conversion and evaluation of arithmetic expressions. The purpose was to implement the StackNode and StackList classes from scratch, without using Java libraries, to manage the conversion logic. Algorithms were developed to convert infix expressions to postfix expressions and, as an additional challenge, to prefix expressions. Additionally, the evaluation of the postfix expression was implemented. All of this was integrated into a graphical user interface (GUI) following the Model-View-Controller (MVC) pattern. It was concluded that the stack is a fundamental LIFO structure for parsing and evaluating expressions, allowing operator precedence to be handled efficiently. The MVC pattern proved to be effective for organizing the code in a modular way.

**Keywords:** Stack, Postfix Expressions, Shunting Yard Algorithm.

## 1. INTRODUCTION:

In this lab exercise, knowledge of linear data structures, specifically stacks, was applied to solve common problems involving expression conversion and evaluation. The focus was on converting infix expressions to postfix notation and evaluating those expressions step by step. As an additional challenge, conversion to prefix notation was implemented. During development, key concepts such as operator precedence and parentheses handling were addressed. The practice was structured following the MVC (Model-View-Controller) architectural pattern to maintain clean and modular code organization, in accordance with laboratory discipline.

## 2. OBJECTIVE(S):

2.1 Implement the StackNode and StackList classes from scratch.
2.2 Apply algorithms to convert infix expressions to postfix notation.
2.3 Evaluate a postfix arithmetic expression step by step.
2.4 Implement a graphical user interface (GUI) to display the results of the conversion and evaluation.
2.5 Use the MVC pattern to maintain modular code organization.

## 3. THEORETICAL FRAMEWORK:

To support this practice, concepts from Joyanes' book Data Structures in Java (2008) were used, supplemented by the implementation of the project.

3.1 **Stack:**

A stack is a linear LIFO (Last-in, first-out) data structure, which means that the last item to enter is the first to leave. Elements can only be added (push or insert operation) or removed (pop or remove operation) from one end, known as the “top” of the stack. In this project, a dynamic stack was implemented using a linked list (StackNode and StackList classes), one of the standard representations for this structure.

3.2 **Arithmetic Expression Notations There are three main ways to write arithmetic expressions:**

- **Infix Notation**: The standard notation, where the operator is placed between its operands (e.g., A + B). It requires parentheses and precedence rules to resolve ambiguity.
- **Postfix Notation (Reverse Polish Notation):** The operator is placed after its operands (e.g., A B +).
- **Prefix Notation (Polish Notation)**: The operator is placed before its operands (e.g., + A B).

Both postfix and prefix notations have the advantage of not requiring parentheses, as the order of operations is explicitly defined by the position of the operators.

3.3 **Infix to Postfix Conversion Algorithm (Shunting-yard) The conversion algorithm utilizes a stack to manage operators and parentheses. The algorithm implemented in the NotationConverterEvaluator class follows these steps:**

- Read the infix expression from left to right.
- If an operand (letter or number) is read, it goes directly to the postfix output.
- If a '(' is read, it is pushed onto the stack.
- If a ')' is read, operators are popped from the stack and added to the output until the matching ( is found. The parentheses are then discarded.
- If an operator is read, its precedence is compared to the operator at the top of the stack. The project's Operator class defines two priorities: infix (precedence outside the stack) and stack (precedence inside the stack), as required by the theory. As long as the operator on the stack has greater or equal precedence, it is popped to the output. Then, the current operator is pushed onto the stack.
- At the end of the expression, any remaining operators on the stack are popped and added to the output.

3.4 **Postfix Evaluation Algorithm Evaluating a postfix expression also requires a stack. The algorithm is as follows:**

- Read the postfix expression from left to right.
- If an operand (a number) is read, it is pushed onto the stack.
- If an operator is read, two operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack.
- When the expression is finished, the single value remaining on the stack is the final result.

3.5 **Infix to Prefix Conversion (Challenge) For the prefix conversion, an algorithmic trick was used that reuses the postfix logic. The process, implemented in the Expression and NotationConverterEvaluator classes, is:**

1. Reverse the original infix expression, swapping all (for ) and vice-versa.
2. Convert this new, reversed infix expression to postfix using the standard algorithm.
3. Reverse the resulting postfix expression to obtain the final prefix expression.

## 4. PROCEDURE DESCRIPTION:

A PC with Windows 10, 8 GB of RAM, and the Eclipse IDE with Java SE 8 was used. The project was structured into model, view, and controller packages as per the guide's instructions.

4.1 **Model Implementation (Package model):**

- The StackNode.java class was created, defining the structure for a simple linked list node, containing an Object and a next reference.
- The StackList.java class was created to implement the Stack ADT. It uses StackNode to manage elements via a top reference. The required LIFO methods were implemented: push(Object x) (inserts at top), pop() (removes from top), peek() (views top), and isEmpty() (checks if top == null).
- The Operator.java class was implemented to define the precedence of operators (^, *, /, +, -, () . Two distinct priorities (infix and stack) were assigned to correctly handle operator associativity, as described in the theoretical framework.
- Utility classes like Validator.java were created to check expression syntax (e.g., balanced parentheses, invalid characters) and EvaluationResult.java to store results and steps.

4.2 **Controller Implementation (Package controller):**

- The NotationConverterEvaluator.java class was created.
- convertToPostfix(String infixExpression) was implemented following the Shunting-yard algorithm, using the custom StackList to manage operators.
- evaluatePostfixStepByStep(String postfixExpression) was implemented to evaluate numeric postfix expressions, using a StackList to stack operands (Double).
- The challenge method convertToPrefix(String infixExpression) was implemented using the reverse-convert-reverse technique described in the framework.
- ...WithSteps methods were added to generate detailed ArrayList<String> logs of each algorithm's execution for display in the GUI.

4.3 **s View Implementation (Package view):**

- The MainWindow.java class was designed, extending JFrame.
- JTextField components were added for "Infix", "Postfix", and "Prefix", along with a JTextArea (in a JScrollPane) for the process steps .

- The required JButton components were added: "Convert to Postfix", "Evaluate Postfix", and "(Challenge) Convert to Prefix" .
- Button event listeners (ActionListeners) were configured to:
  1. Get the text from the infix field.
  2. Call the appropriate methods in NotationConverterEvaluator (e.g., validateAndConvertToPostfix or evaluatePostfixStepByStep).
  3. Receive the ValidationResult or EvaluationResult objects and update the GUI text fields (txtPostfix, txtPrefix, txtSteps, lblResult).

## 5. RESULTS ANALYSIS:

The test expressions provided in the guide were executed using the developed application .

Test 1: Complex Numeric Expression

- Infix Input: 4*(5+6-(8/2^3)-7)-11
- Postfix Output: 456+823^/7---*11-
- Prefix Output: -*4--+56/8^23711
- Postfix Evaluation: The evaluatePostfixStepByStep algorithm was executed.
  1. ...
  2. Stack: [4, 5, 6]
  3. Operate 5 + 6 = 11. Stack: [4, 11]
  4. Stack: [4, 11, 8, 2, 3]
  5. Operate 2 ^ 3 = 8. Stack: [4, 11, 8, 8]
  6. Operate 8 / 8 = 1. Stack: [4, 11, 1]
  7. Operate 11 - 1 = 10. Stack: [4, 10]
  8. Stack: [4, 10, 7]
  9. Operate 10 - 7 = 3. Stack: [4, 3]
  10. Operate 4 * 3 = 12. Stack: [12]
  11. Stack: [12, 11]
  12. Operate 12 - 11 = 1. Stack: [1]
- Final Result: 1.0

Test 2: Algebraic Expression

- Infix Input: A*(B+C)/D
- Postfix Output: ABC+*D/
- Prefix Output: /*A+BCD
- Postfix Evaluation: The Validator.validateForEvaluation method returned false, as the expression contains letters. The GUI displayed the corresponding warning message, indicating that evaluation is only possible with numbers.

Test 3: Multiple Exponents (Right-Associativity)

- Infix Input: 3+4*2/(1-5)^2^3
- Postfix Output: 342*15-23^^/+
  - *Precedence Analysis:* The algorithm correctly handled the right-associativity of the ^ operator (defined in Operator.java with infix priority 4 and stack priority 3 ). When the second ^ was read, its infix priority (4) was greater than the stack priority of the ^ on the stack (3), allowing it to be pushed. This ensures 2^3 is evaluated before (1-5)^....
- Prefix Output: +3/*42^^-1523
- Postfix Evaluation:
  1. ...
  2. Stack: [3, 8, -4, 2, 3]
  3. Operate 2 ^ 3 = 8. Stack: [3, 8, -4, 8]
  4. Operate -4 ^ 8 = 65536. Stack: [3, 8, 65536]
  5. Operate 8 / 65536 = 0.000122... Stack: [3, 0.000122...]
  6. Operate 3 + 0.000122... = 3.000122... Stack: [3.000122...]
- Final Result: 3.00 (as displayed in the GUI, rounded)

## 6. GRÁFICOS O FOTOGRAFÍAS:

**1.** Screenshot of the graphical user interface (GUI) showing the numeric infix expression 4*(5+6-(8/2^3)-7)-11 successfully converted to Postfix and Prefix notation. The final evaluation result, 1.0, is displayed, validating the algorithm's correctness.

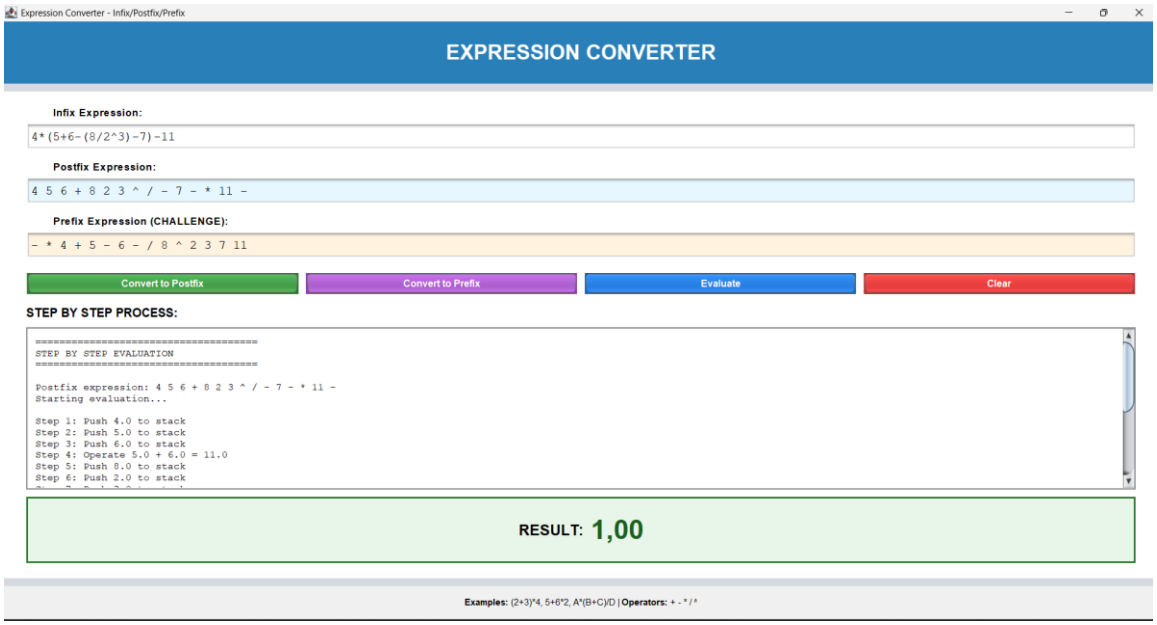


*Illustration 1 Full Execution of Test Case*

Screenshot of the GUI showing the conversion of the algebraic expression A*(B+C)/D. This result validates the Infix to Postfix (ABC+*D/) and Infix to Prefix (/*A+BCD) conversion algorithms, fulfilling the additional challenge.

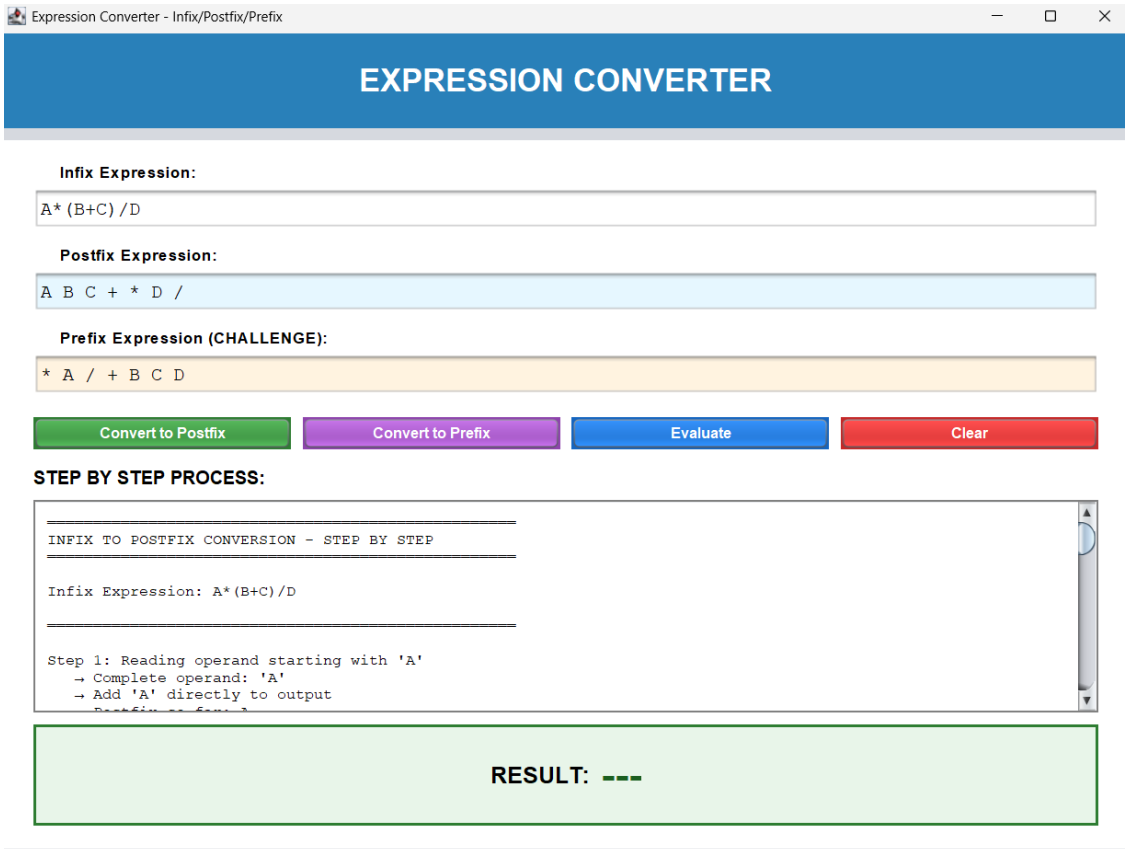


*Illustration 2 Algebraic Expression and Prefix Challenge*

The Graphical User Interface (GUI) displaying the final execution for the test expression: 3+4*2/(1-5)^2^3. This figure validates the algorithm's ability to correctly handle complex operator precedence and the right-associativity of the exponentiation operator (^). The GUI confirms the successful conversion to the Postfix

notation 342*15-23^^/+ and the final numerical evaluation result of approximately **3.00**.

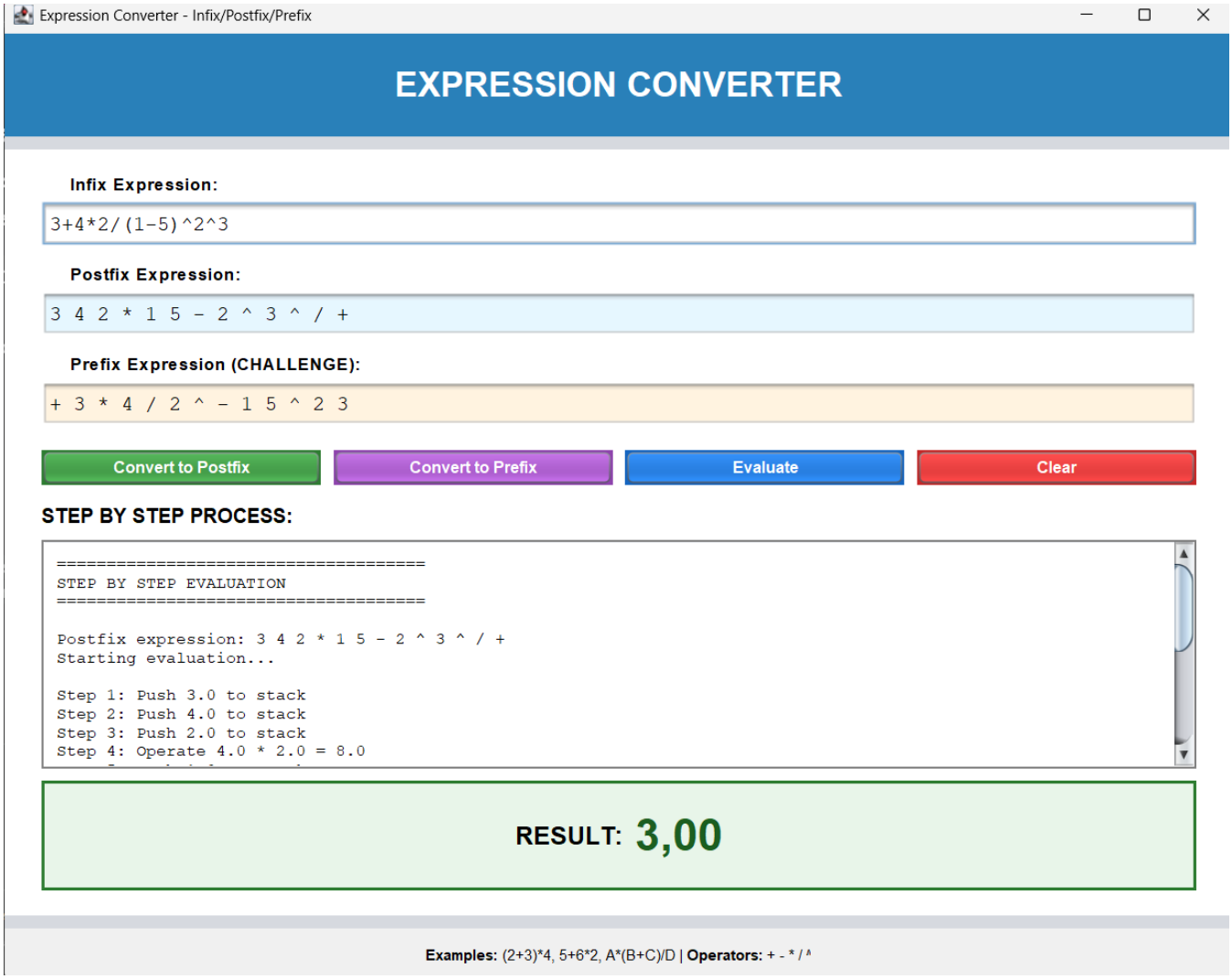


*Ilustración 3 Execution of Test Case 3*

## 7. CHALLENGE:

To achieve 100% of the functionality objectives, the conversion of mathematical expressions from infix notation to prefix notation (also known as Polish notation) was implemented and documented. This challenge represents a significant extension of the system's core functionality.

**Implemented Conversion Strategy**

The solution leverages the standard **"reverse-convert-reverse"** algorithm, which builds directly upon the existing convertToPostfix method.

**Core Algorithm Implementation:** The convertToPrefix method orchestrates this three-phase process:

1. Reverse with Parenthesis Swap: The input infix expression is reversed using expression.reverseWithParenthesis()
2. Postfix Conversion: The reversed expression is processed by the convertToPostfix method (as shown in screenshots 224211.png and 224224.png), which implements the standard stack-based algorithm
3. Final Reverse: The resulting postfix string is reversed to yield the final prefix expression

   Key Features of the convertToPostfix Method:
   - Uses StackList for operator management
   - Handles multi-digit numbers and variables through character-by-character reading
   - Implements proper operator precedence checking using the priority method

- Correctly manages parentheses matching and nesting
- Includes robust error handling with try-catch blocks

```java
// Converts infix expression to postfix using StackList
public static String convertToPostfix(String infixExpression) {
    StringBuilder postfix = new StringBuilder();
    StackList stack = new StackList();

    // Traverses each character of the infix expression
    for (int i = 0; i < infixExpression.length(); i++) {
        char symbol = infixExpression.charAt(i);

        if (symbol == ' ') {
            continue; // Ignores spaces
        }

        // If it's a letter or number, read the complete number/variable
        if (Character.isLetterOrDigit(symbol)) {
            // Read complete number or variable
            while (i < infixExpression.length() && Character.isLetterOrDigit(infixExpression.charAt(i))) {
                postfix.append(infixExpression.charAt(i));
                i++;
            }
            i--; // Step back one position
            postfix.append(' '); // Add space as separator
        }
        // If it's '(' push to stack
        else if (symbol == '(') {
            stack.push(symbol);
        }
        // If it's ')' pop operators until finding '('
        else if (symbol == ')') {
            try {
                while (!stack.isEmpty() && (char)stack.peek() != '(') {
                    postfix.append((char)stack.peek()).append(' ');
                    stack.pop();
                }
                if (!stack.isEmpty()) {
                    stack.pop(); // Removes the '('
                }
            } catch (Exception e) {}
        }
        // If it's operator pop those with higher or equal priority
        else if (isOperator(symbol)) {
            try {
                while (!stack.isEmpty() &&
                       (char)stack.peek() != '(' &&
                       priority((char)stack.peek()) >= priority(symbol)) {
                    postfix.append((char)stack.peek()).append(' ');
                    stack.pop();
                }
                stack.push(symbol);
            } catch (Exception e) {}
        }
    }

        // Empties the stack at the end
        try {
            while (!stack.isEmpty()) {
                postfix.append((char)stack.peek()).append(' ');
                stack.pop();
            }
        } catch (Exception e) {}

        return postfix.toString().trim();
    }
```

## 8. DISCUSIÓN

The results obtained from the tests validate the correct implementation of the theoretical algorithms. The Shunting-yard algorithm, as described by Joyanes, was successfully implemented in NotationConverterEvaluator.java. The key

to this success was the correct definition of operator precedence (both infix and in-stack) in the Operator.java class , which properly handled both precedence (* before +) and associativity (right-associativity for ^).

Furthermore, the postfix evaluation algorithm performed as expected. The custom StackList.java implementation served as the core data structure for the entire practice; without a functional LIFO structure, neither of the primary algorithms would have worked. The prefix conversion challenge demonstrated the flexibility of these algorithms, allowing for their reuse in a novel way (reverse-postfix-reverse). The MVC pattern was a sound choice, enabling the stack logic (model.StackList) and the conversion logic (controller.NotationConverterEvaluator) to be developed and tested independently of the graphical interface (view.MainWindow).

## 9. CONCLUSIONS:

- The StackNode and StackList classes were successfully implemented from scratch, demonstrating the ability to build fundamental LIFO data structures without relying on Java's built-in collections.
- The Shunting-yard algorithm was correctly applied to convert infix expressions to postfix notation, properly managing operator precedence and parentheses as verified by the test cases.
- Numeric postfix expressions were successfully evaluated step-by-step, using an operand stack to store intermediate results and compute the final value.
- The prefix conversion challenge was implemented, demonstrating a deeper understanding of the algorithm by reversing the postfix conversion logic.
- A functional GUI was developed that interacts with the controller to display all conversions, evaluations, and the detailed step-by-step processes.
- The MVC pattern was successfully used, resulting in organized, modular, and maintainable code by separating the data logic (Model), application logic (Controller), and presentation (View).

## 10. BIBLIOGRAPHY:


- Joyanes Aguilar, Luis, and Ignacio Zahonero Martínez. 2008. *Estructuras de datos en Java*. McGraw-Hill.
- Date of consultation: November 1, 2025.


Link github: [angeldev7/P1Lab2xpression-Converter-and-Evaluator-using-Custom-Stacks-and-Queues](angeldev7/P1Lab2xpression-Converter-and-Evaluator-using-Custom-Stacks-and-Queues)