Task 1 – Design and Implement Stacks and Queues

1. Practice modular programming. That is, compartmentalize the data structures and operations by storing the implementation codes in separate source code files.
2. Create separate file/s for each data structure
    1. For example, if the project was implemented with Java, then the code for the stack must be in a separate file *stack.java*, and the code for the queue must be in another separate file *queue.java*
3. Implement the operations of Stacks and Queues yourselves.

Task 2 - Implement Parser

1. Create a separate file for the implementation of the parser.
2. Accept a mathematical expression in INFIX notation (fully, unambiguously parenthesized) as input.
    - Example input #1: (1+(2*3))
    - Example input #2: ((8+2)/3)
    - Example input #3: (5>=3)
    - Example input #4: (1&&0)
    - Example input #5: (!((2>1)&&(3<2)))
    - Example input #6: (123/0)
3. Assume the following:
    - the tokens making up the INFIX expression are stored in a string with at most 256 characters including the NULL byte.
    - NO spaces are separating the tokens and that the expression is syntactically valid.
    - Operands are limited to non-negative integers (i.e. 0 and higher).
    - Operators include:
        - Arithmetic operators: **+**, **-**, *, /,**%**, **^** (caret is for exponentiation)
        - Relational operators: **>**, **<**, **>=**, **<=**, **!=**, **==**
        - Logical operators: **!**, **&&**, **||**
4. Implement the algorithm for parsing mathematical expressions from fully parenthesized INFIX notation to POSTFIX notation/Reverse Polish Notation (RPN). You can modify the algorithm for INFIX evaluation in the notebook, or you may also check out Shunting Yard Algorithm.
5. Output the POSTFIX expression. Separate consecutive tokens with exactly one SPACE character.
    - For example input #1, the output should be: 1 2 3 * +
    - For example input #5, the output should be: 2 1 > 3 2 < && !
    - For example input #6, the output should be: 123 0 /

Task 3 - Implement Postfix Evaluation

1. Create a separate file for the implementation of the Postfix evaluation.
2. Implement the algorithm for evaluating Postfix expressions.
3. Output the value.
    - For example input #1, the evaluated value should be: 7
    - For example input #2, the evaluated value should be: 3
    - For example input #3, the evaluated value should be: 1
    - For example input #4, the evaluated value should be: 0
    - For example input #5, the evaluated value should be: 1
    - For example input #6, there is no evaluated value. Print instead: Division by zero error!
4. Follow these restrictions:
    - The division operator is an integer/floor division. For example, 10/3 gives a result of 3 (see Example input #2).
    - In case the denominator is zero, do not perform the actual division since it will cause a run-time error. Your program should check this case, and if it occurs, the program should print the constant string “Division by zero error!” (see Example input #6).
    - For relational and logical operators, the result should be limited to either a 1 or a 0 which mean True or False respectively (see Example input #3 to #5).

Task 4 - Testing

1. Create test cases and perform exhaustive testing.
2. Test input validation.
3. Test each operator, and the grouping symbols.
4. Test the operator precedence and associativity
5. Test the division by zero error.

Tip: It would be easier to do testing if you write all test inputs in a text file.