Lab 3: Stacks and Queues

3.1 (Stack) Create a program which show a stack of characters. The program will implement the stack using an **array**. Do not use stack library. Test functions by push, pop some characters. Note that testing empty case and full case.

3.2 (Queue) Create a program which show a a queue of integers. The program will implement the queue using an array. Do not use queue library. Note that testing empty case and full case.

3.3 (Palindromes) Use a stack and queue to implement a more powerful version of the is_palindrome() function. This function will return true if text is a palindrome, false otherwise. A palindrome is a string that is identical to itself when reversed. For example, "madam", "dad", and "abba" are palindromes. Note: the empty string is a palindrome, as is every string of length one. (Read more: https://en.wikipedia.org/wiki/Palindrome)

- Level 1: check string is strict palindromes (normal case).
- Level 2: check string is loose palindromes. It means that the function is more flexible than level 1. Your program should ignore whitespace and punctuation, and all comparisons should be case-insensitive. Examples of valid palindromes:

    ""

    "a"

    "aa"

    "aaa"

    "aba"

    "abba"

    "Taco cat"

    "Madam, I'm Adam"

    "A man, a plan, a canal: Panama"

    "Doc, note: I dissent. A fast never prevents a fatness. I diet on cod."

3.4 (Postfix Creation) Reverse Polish notation (RPN) is an alternative way to represent arithmetic expressions like those you might type into a calculator. At one point in time, it was popular to use RPN in physical calculators because it was easier to implement a calculator that read RPN as input. Similarly, it will be easier for you to do this assignment with RPN than with the normal style of writing arithmetic expressions. (Read more: https://en.wikipedia.org/wiki/Reverse_Polish_notation)

In RPN, arithmetic operators such as +, *, and - come after their two op perands, rather than in between them. Here are some examples of infix notation vs. RPN.

"1 + 2" == "1 2 +"

"3 - 1" == "3 1 -"

"2 * 3" == "2 3 *"

"3 + (4 * 5)" == "4 5 * 3 +"

"5 + ((1 + 2) * 4) - 3" == "5 1 2 + 4 * + 3 -"

You write program to convert a given infix mathematical expression to RPN. We will discuss the shunting-yard algorithm for converting. Just like the evaluation of RPN, the algorithm is stack-based . For the conversion we will use queue for output. Additionally we will use a stack for operators that haven't been yet added to the output.

```
For all the input tokens
    If token is an operator (x):
        While there is an operator (y) at the top of the operators stack
        and either (x) is left-associative and its precedence is less or
        equal to that of (y), or (x) is right-associative and its
        precedence is less than (y):
            Pop (y) from the stack;
            Add (y) output buffer;
            Push (x) on the stack;
    Else If token is left parenthesis, then push it on the stack;
    Else If token is a right parenthesis:
        Until the top token (from the stack) is left parenthesis, pop
        from the stack to the output buffer;
        Also pop the left parenthesis but don't include it in the output
        buffer;
    Else add token to output buffer.
While there are still operator tokens in the stack, pop them to output
```

Read more detail and example in:

https://en.wikipedia.org/wiki/Shunting-yard_algorithm#The_algorithm_in_detail

Suppose that our expression will only include integer numbers, standard arithmetic operators (+, -, *, /), and parenthesis.

3.4 (Postfix Evaluation) Postfix notation (sometimes called "Reverse Polish Notation" or "RPN") for explicitly specifying the operation order of a mathematical expression. We also discussed an algorithm for evaluating postfix expressions using a stack:

```
for each token in expression
    if token is a number
        stack.push(token)
    if token is an operator
        op2 = stack.pop()
        op1 = stack.pop()
        stack.push(op1 <op> op2)
return stack.pop()
```

Implement this algorithm by writing a function which return the value of evaluating 'expr' as a postfix expression. All operands should be integer numbers, and all standard arithmetic operators (+, -, *, /) are supported. All operands and operators should be separated in 'expr' by a single space.

Include some tests in a main() function. Examples:

"1 2 +" = 3

"15 8 -" = 7

"2 3 *" = 6

"4 5 * 3 +" = 23

"5 1 2 + 4 * + 3 -" = 14