1. **Explain briefly how you implement the DoubleStack. Explain how your implementation guarantees an efficient usage of the allocated memory.**

Answer: I implemented the DoubleStack data structure by using only a single array, which used minimum memory. Because in the file that the program will read will give the size of the stack. So, first I get the size of the stack (array), then because of we need implement a double stack. So I made RED stack start at the beginning of the array (start with index of 0), and BLUE stack start at the ending of the array (start with index of the size of array - 1). Two counters count the size of RED and BLUE stacks' size separately. Because I only used one single array to implement it, the memory usage is the minimum and efficient.

2. **Describe the algorithm you implemented for the conversion from infix to prefix and justify its correctness.**

Answer: In order to convert from infix to postfix I implemented a double stack to put operands in red stack and the operators in blue stack. Then I simply push values on to the two stacks depending on whether it's an operator or an operator. But once the algorithm encounters a closing parenthesis ')' it pops values until an opening parenthesis is encountered. Once the closing parenthesis is encountered I evaluate the expression using a double array map that has all the various precedencies of the operators hard coded. This therefore allows the algorithm to make computations while taking the precedence in consideration.

3. **Explain the algorithm you implemented for the evaluation of prefix expressions.**

Answer: First, in order to evaluate the value of the given prefix expressions easily. I inverse the order of the expression, which makes it become a postfix expression. Store it in an array and then use DoubleStack to evaluate. If it is an operator, push to RED stack and pop an operator check the operation is +-*/, do the computation, push the result to BLUE stack, else (if it is an operand) push it to BLUE stack. At the end, pop an element in BLUE stack; it will be the only element in stack, which is the final result we want.

4. **Explain how you implemented MyQueue and justify the constant time complexity of your "lookup" function.**

Answer: I used a single array to implement MyQueue. Every time when enqueue a new element in it, just add it to the array. But every time when I dequeue an element, based on the first in first out rule of queue, we need pop the first element of the array. So, I pop it out (dequeue), then move the rest elements in the array move to the left one step. Before enqueue and dequeue, I always check the size of the queue, cause the size of queue is dynamic, so I written a resize function which will store all the elements in a new array (different size). If the array is full, then double the size of array; similarly, if all the elements only occupied 1/4 size of the array, then narrow the size to 1/2 of the original size. Lookup function will take an integer as a parameter and return the Nth element that we want (if this element is available). So, it will always take a constant time.