Report

PART A

The implementation was done using a Generic Node class and a *double-linked list*. The class includes the following member variables:

1. *head*: reference to the first object
2. tail: reference to the last object
3. *size:* the number of elements in the queue

The above variables are updated in every queue modification method and help achieve O(1) time complexity in the **insertion/removal** and **size** operations*,* using the properties of the *double-linked list,* which allow constant-time updates by simply changing the references to the next and previous elements. These operations have constant runtime, as they do not depend on the number of items in the queue.

PART B

Using the implementation of Part A, we can process a prefix numerical expression using a stack-like structure. We first read the expression and start analyzing it from end to beginning, checking three cases:

1. If we encounter a number (0-9), we store it at the beginning of our structure using **addFirst()**.
2. If we encounter an operator (+, -, *, /), we pop the first 2 elements, by calling **removeFirst**() twice and store the resulting expression as a new string (this is why the structure must always contain at least 2 elements before popping).
3. If we encounter an unknown character (e.g. "c"), the loop stops and an error message is printed.

Before entering the for loop, a check is performed to ensure that the last two elements of the expression are numbers, so that the structure starts with at least two elements before popping begins.

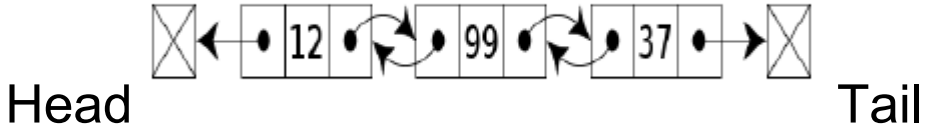
Let's assume that we have the prefix *+357. We start inserting elements into the structure from **left to right**, so after the third iteration the structure will look like:
Head → [3] ↔ [5] ↔ [7] ← Tail.

In the next iteration, the symbol "+" is encountered. We pop "3" and "5" and insert a new processed value at the head. The structure now looks like:
Head → [(3+5)] ↔ [7] ← Tail

In the final iteration, due to the character " * ", we pop twice again and insert a new element. The structure becomes:
Head → [((3+5)*7)] ← Tail

Since there are no more symbols, the loop ends, and we print the contents of the structure at the Head.

PART C



Input Control and Queue Initialization

The Input Control processes one character at a time and immediately terminates the loop if an invalid character is detected. At the same time, when the character is valid, it is **immediately placed** into *the Double-Ended Queue,* eliminating the need for a second loop and thus improving the algorithm's performance, especially in cases of a long sequence.

Using the Deque to solve the Palindrome

If we observe the examples, which are Watson-Crick complemented palindromes ATATATAT, AAAACGTTTT, the characters marked with the same color are complements of each other and always form **pairs**.

Based on this observation, the program should compare the two ends of the input Sequence when:

1. It is **not empty** (since the empty string is always a Watson-Crick complemented palindrome).

2. When the length of the sequence is even (if it is **odd**, that means there is a central character without a pair, and after converting to its complement, it will differ from the corresponding character in the original sequence — for example, AGT becomes ACT after complementing).

The most efficient way, taking advantage of the capabilities of StringDoubleEndedQueue, is to use **removeFirst()** and **removeLast()** methods, which return the characters and set new ends ready for comparison in the next Loop.

We observe that the **isComplemantaryCouple()** method performs logical checks with O(1) time complexity and calls two O(1) methods, **removeFirst()** and **removeLast()** per iteration. Essentially, we have a constant time cost of c1*N, so the deque processing is done in O(N) time. The **isInputValidAndQueuePopulated()** method also uses logical operations, assignments, method calls and return statements, all of O(1) complexity. So, we have a constant c2*N and therefore execution time O(N).
In total we have c1*N+c2*N+c3+N (where c3+N all the other Main functions) and thus the program runs in c*N time, that is, O(N).