Part A (Double Ended Queue Implementation)

I started by creating the StringDoubleEndedQueueImpl class, which is the implementation of the interface.

I created three class methods, “first”, “last”, and “size”. The first two indicate the beginning and end of the queue, respectively, while the last one indicates the size of the queue (an integer).

Then I created the nested class Node, which is used to indicate the nodes of the doubly linked list. The class contains a String object, a reference to the next node, and one to the previous one (“prev”). In this way, we join the nodes, and the list structure is obtained.

I initialized the three class fields via the constructor, and initially set them to NULL, NULL, and 0, respectively.

I continued by creating the isEmpty method, which returns the value TRUE if the queue is empty, and the value FALSE if it is not.

The add.First method constructs a new node, which is added to the start of the queue, and we give it a value (item) to put into the new node. It also changes the references to the next and previous nodes. If the queue is initially empty, the node we insert is the only node in the queue, so we define both the start (first) and the end of the queue (last) as the node we inserted. If the queue is not empty, we define the new node as the first node in the queue (first), and at the same time as the previous node of the node that was first before we inserted the node (first.prev). The method is completed in O(1) time, since we have direct access to the first node of the queue, and we can simply add the new first node by changing the pointers of the previous first node. There is no need for any kind of iteration for all the nodes of the queue: whether we have 100 nodes in the queue or 2, adding a new node to the beginning of the queue changes the pointers of only the first node, and is done in the same time.

Similarly, the add.Last method adds a new node to the end of the queue. If the queue is initially empty, the node we insert is the only node in the queue, so we define both the start (first) and the end of the queue (last) as the node we inserted. If the queue is not empty, we set the new node as the last node in the queue (last), and at the same time as the next node to the node that was last before we inserted the node (last.next). The method is also completed in O(1) time, as we have direct access to the last node in the queue, and we can simply add the new last node by changing the pointers of the previous last node. There is no need for some kind of iteration for all the nodes in the queue, i.e. the time does not depend on the size of the queue.

The remove.First method removes the first node in the queue. If the queue is initially empty, we get the error message NoSuchElementException, and we cannot remove the first node, since it does not exist. The deletion is done by

changing the pointer “first” to point to the second node in the list (first.next). If after this process the first node is empty, the queue is empty, so we set both the first and last nodes of the queue to NULL. If the list is not empty, the “prev” pointer of the new first node (i.e. the initially second) is set to NULL.

The remove.Last method works in a similar way, changing the pointers of the last and second to last nodes. Both methods finish in O(1) time, as we have direct access to the first and last nodes of the queue, respectively. The logic is the same as that of the insertion methods. The getFirst and getLast methods, which return the first and last elements of the queue, finish in O(1) time, because they also only deal with the first and last nodes to be executed. The size method is also O(1), as the size class field (initial value: NULL) is updated on each addition/removal of an element, so the time is constant on each update.

Finally, I print the elements of the queue (the elements inside the nodes), and return the size of the queue. I also changed the implementation of the interface, using generics. I modified the appropriate parameters so that my queue could accept any data type, not just String.

Part B (Prefix to Infix)

As we know, the implementation of part A is the implementation of a double ended queue. As a double ended queue, it has 2 ends. In my implementation these two ends are called First and Last, representing the first and last element of the queue respectively. As Part B requires the use of a stack to solve the problem, I used the double ended queue as a stack, pushing and popping elements only from one side of the queue, thinking of that particular side as the top of the stack (since the basic principle of the stack is that we push and pop exclusively from the top of the stack). In a stack, we push and pop from the top of the stack, where the top is the first element of the stack. I decided to think of the First end of the queue as the top of the stack, therefore I only insert and remove elements from the left end of the queue.

Then, after checking for valid user input, before using the implementation of part A for the conversion into an infix form, I reversed the prefix expression, essentially converting it into a postfix one, to be able to use part A to find the infix expression by following the following procedure (taking the prefix expression *+12/34 as an example). Since this particular sequence will be reversed, we get the sequence 43/21+*, which is in postfix form, which we will use to find the infix representation.

Below I show a conversion example, using the implementation from part A.

Prefix to Infix

Prefix: *+12/34. Conversion to Postfix -> Postfix: 43/21+*.

At this point, the implementation from part A begins (Having first converted, as mentioned previously, the result Postfix expression into a character array).

The traversal of the character array is done through the use of the “for” loop.

When the element of the array is an operand, we push it onto the “stack” with the addFirst() method. When the element of the array is an operator,, we “pop” 2 elements from the “stack” with the removeFirst() method and create the resulting expression having the operand that we just encountered between these two elements (operands). Then we “push” the specific expression onto the “stack” using the addFirst() method. This process is followed until the traversal of the character array is completed following the process described previously, and we end up with a single element (which is a numeric expression) on the stack, which will be the Infix form of the Prefix expression that the user provided as input.

Part C (DNA Palindrome)

To check whether a DNA sequence is a Watson-Crick palindrome or not via the queue, we need to insert the elements into the queue, and do the following:

→ Replace each nucleotide with its complementary element and

→ Reverse the order.

If the original sequence is obtained, we have a Watson-Crick palindrome sequence.

As a first step, I created an empty list named “queue”. My first thought was to find the complementary element for each element of the sequence, and insert the sequence with the complementary elements into the list with the add.First method (so from the front). Then remove them one by one with the remove.First method to obtain the reversed sequence, so that we can check if it is identical to the original. However, I ultimately preferred to use both ends of the queue, in order to take full advantage of it.

First, I defined the values that the nucleotides can take (A, T, C, G), and created the method isComplementary to check whether each pair of them is complementary (output TRUE) or not (output FALSE). I created a “for” loop to check whether each element of the sequence is a value that a nucleotide can take or not. In each iteration of the loop, I take the elements of the sequence one by one, until I reach the last one. If the value of each element of the sequence is valid, it gets added to the queue via the method add.Last, so as not to spoil the order of the original sequence. Otherwise, this error message appears”: "Invalid DNA sequence: " + the invalid value". Since we know that the empty string “” is a Watson-Crick palindrome, I check with the “if” condition whether I have an empty string and if this is true, I print that we have a W-C palindrome. We also

use a new “if” condition to check whether the sequence consists of an odd number of elements. If this is the case, I print that it is not a W-C palindrome. Since we have filled the queue with elements and we do not have an empty string or an odd number of elements, we create a “while” loop that takes as a parameter the size of the queue. As long as we have a queue with a number of elements greater than or equal to 2 (which is not odd), we keep removing one element from the beginning of the queue and one from the end, and check whether they are complementary. This procedure directly checks the complementarity after the reversal, since during the reversal the last element will go to the position of the first, while the first element will go to the last position. This will apply for all iterations of the “while” loop, until the queue is empty. Therefore, checking the complementarity of the first and last elements in each iteration examines both the complementarity and reversal conditions. If the final sequence is the same as the initial one, we get true (and we have a W-C palindrome sequence), otherwise false (and we do not have a W-C palindrome sequence). We put the above procedure in a Boolean method, isWatsonCrickPalindrome, which returns the value TRUE if we have a W-C palindrome sequence, and the value FALSE if we do not. Finally, we put all of the above in the DNAPalindrome class, and add the Scanner helper class to the code for user input. If the user input is a W-C palindrome sequence, the message "Yes, it is a Watson-Crick complemented palindrome." is printed, while if it is not, the message "No, it is not a Watson-Crick complemented palindrome." is printed.

Example: Let's imagine the sequence ATCGAT. The “for” loop first checks element A, and sets it as currentNucleotide. The method isValidNucleotide tells us that it is an accepted nucleotide value, so it is added to the empty queue from the end, and is currently its only element. Then, the loop checks element T, and sets it as currentNucleotide. With the same logic, since it is an accepted nucleotide value, the element is added to the end of the list, after A (now the list has 2 elements). The loop is repeated for each element of the sequence, until we reach the last one. Since all elements are accepted, they are all put into the list, and we do not get an error message. The order of the elements is preserved in the queue. We do not have an empty string or an odd number of elements, so the process proceeds to the while loop. First while loop: We remove the first and last elements (A and T) from the queue. These are complementary, so the process continues. Second while loop: We remove the first and last elements (T and A) from the queue. These are complementary, so the process continues. Third while loop: We remove the first and last elements (C and G) from the queue. These are complementary, so the process continues. The queue has no more elements in it, so the loop stops. The loop returns true, so the message "Yes, it is a Watson-Crick complemented palindrome." is printed.