Introduction

This weeks program asked us to implement a pre decided interface, which in turn would pass a number of J-Unit tests which had also been predefined.

Design/Implementation

Since the interface was given to us, very little design was done. However, free reign was given when it came to the way in which we would implement the methods.

Size: I only used a small loop to take me through the list. I did this as it kept both time and space complexities low, while being the simplest way to implement this method.

Count: I used a loop and a counter. The loop to take me through the list, and the counter was incremented every time the searched for element was found. I did it this way, as I felt that this was the most simple and time/space complexity efficient.

ConvertToString: I used a loop to take me through the list and a string to store the total final version. I pursued this technique, as I believed it was the best way to get through the list to create a string representation. Unfortunately this was not as efficient in terms of space complexity.

GetFromFront: I again used a simple loop to move through the list, while looking for the value in the nth position. Once it was found it was stored as an int and returned. I followed this method as I felt this was the most appropriate way to implement the method.

GetFromBack: I used a simple loop to read the list into a stack. I used this so that I could reverse the list. This meant I could work through it normally, reading from the front. Unfortunately, this made the method rather inefficient in terms of both time and space complexity.

DeepEquals: I used a while loop to check through the two lists. Initially checking to see if there were any equivalent elements which were not equal, then I checked if either were null lists. If these tests were incorrect then the method passed due to it finding no errors. I felt it was better to do this, almost backwards, instead of trying to find if they equalled each other, I found ways in which I could test if they didn't equal one another.

DeepCopy: I used an array list to store all of the objects which i would extract from the list. To do so, i looped through the list and added each element to the new array list. Once this task was completed, i added the values of the array list to a new list node and returned it. I felt this was a good way to implement this method, as it meant i could remove all pointers to previous lists. However, it was rather inefficient in terms of space complexity.

Contains: I used a loop to move through the list. If i encountered the element which was being searched for i returned true.

ContainsDuplicates: I used an array list to store the elements which I had already come across. I then looped through the list checking each element against each element in the array list. If it was there then I returned true, otherwise I added the element to the list and continued looping. This

worked well, but was quite inefficient in terms of time complexity, as I used a loop to go through the list, but contains also used a loop to move through the array list.

Append: I checked for any null values, and had different returns depending on what was null. If there were no nulls, then I used a loop to move the pointer on the list to the last element on the first list. I then set the next position in the list to be equal to the second list. I felt this was an effective method, as it had a constant space complexity.

Flatten: I used an array list to store the list containing the other lists. To populate it, I used a loop to go through the list and extracted each element. I then looped through the array list, filling a list node with the values of the array list. I thought this method was very inefficient, but could think of no better way to implement it.

IsCircular: I used a hash set to store the individual hashes of the list. To populate the hash set, I set the goal to be the hash Code of the first value in the list. I then looped through the rest of the list and checked if the node was equal to the first node, if it wasn't I checked if the node had indeed been found before. If it had, I returned false, as there was a circular pattern, but not to the first value. If both of these do not succeed then I add the nodes hash Code to the array list. I found this to be a pretty good way to implement the method, but I also feel that it could have been more efficient.

ContainsCycle: I found this method to be very similar to the isCircular method, with the only real difference being if I found a node I had already found previously, then I would return true instead of false. In the same way as I was with the previous method, I found it to be a good method, but I also feel that it could have been more efficient.

Map: For this method, I used an array list of objects to store the values in the list, which I got by looping through the list. I then created a new list node, and added values to it, by looping through the array list. I feel that this method could definitely have been more efficient, as I don't think it was entirely necessary to move to an array list and then back to a list node.

Reduce: I found this to be a good method. I used a loop to move through the list, and then using the different methods provided by the parameters, I managed to make this method relatively efficient.

Sort: This method was a horrible method to implement. I had many problems here and the final result was not one I had initially planned to use. Initially I had intended to use two while loops to move through the list to sort every item in said list, but in the end I had to read all the values into an array list, sort the array list then read the array list into a new list node. It could definitely have been more efficient, but when attempting to improve it, I found that when moving the pointer, I had not adequately moved the pointers on the list. So I re-implemented the method using the array list approach and managed to make it work, albeit in a significantly less efficient method.

Time/Space Complexity

Size:

Time – O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(C) constant – I thought it was constant, as there was only one variable which was of a fixed size in terms of memory.

Count:

Time – O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(C) constant - I thought it was constant, as there was only one variable which was of a fixed size in terms of memory.

ToString:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(N) linear – I thought it was linear, as the length of the string is determined on the size of the list

GetFromFront:

Time – O(N) linear – I thought it was linear, as the loop relied on the size of the list, and also the value of n.

Space - O(C) constant - I thought it was constant, as there was only one variable which was of a fixed size in terms of memory.

GetFromBack:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list, and also the value of n.

Space – O(N) linear – I thought it was linear as I used a stack, the size of which was determined by the size of the list

DeepEquals:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(C) constant – I thought it was constant, as there was nothing being stored in the method.

DeepCopy:

Time - O(N) linear – I thought it was linear, as the loops relied on the size of the list

Space – O(N) linear – I thought it was linear, as I used an array list to store the elements I was going to be copying

Contains:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(C) constant – I thought it was constant, as there was nothing being stored in the method.

ContaintsDuplicates:

Time – O(N * N) quadratic – due to the loop having a **.contains** method, which loops through the array list it is applied to, I thought that this was of quadratic time complexity

Space – O(N) linear – I thought it was linear, as I used an array list to store the elements I had already seen

Append:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(C) constant – I thought it was constant, as there was nothing being stored in the method.

Flatten:

Time - O(N) linear – I thought it was linear, as the loops relied on the size of the list

Space – O(N) linear – I thought it was linear, as I used an array list to store the list nodes that i extracted initially from the list

IsCircular:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(N) linear – I thought it was linear, as I used a hash set to store the hash codes of all the elements in the list I had seen, this means that the hash sets size is reliant on the size of the list

ContainsCycles:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(N) linear – I thought it was linear, as I used a hash set to store the hash codes of all the elements in the list I had seen, this means that the hash sets size is reliant on the size of the list

Map:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(N) linear – I thought it was linear, as I used an array list to store all the mapped objects from the lists

Reduce:

Time - O(N) linear – I thought it was linear, as the loop relied on the size of the list

Space – O(C) constant – I thought it was constant, as there was only one variable, which was of constant size in terms of memory

Sort:

Time – O(N * N) quadratic – I thought it was quadratic, as I had a for loop within a for loop in this method

Space – O(N) linear – I thought it was linear, as I had an array list which held all the elements during the actual sorting phase

Testing

All tests conducted were predefined, and held in the ListManipulationTest class.

Conclusion

During the implementation stage, I felt that I was doing rather well in completing the tasks relatively efficiently. However, upon reflection in this report, I have realised that there were a number of other paths I could have followed in order to make my program run significantly more efficiently. This has disappointed me slightly, but it has also helped me realise how I need to think about my coding, so I am sure that I will be able to make my future practicals more efficient as a result of this one.