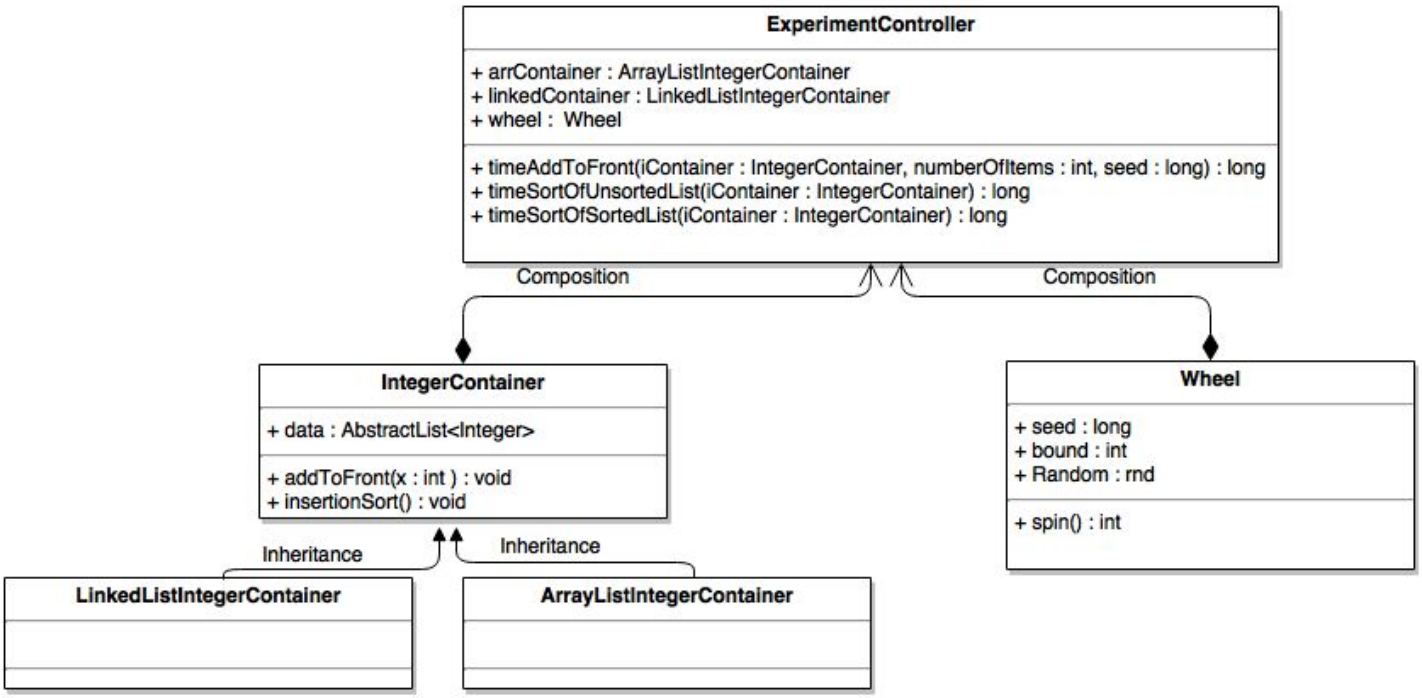
Zura Mestiashvili
Professor Liew
CS150 Data Structures and Algorithms
09/18/2016

Introduction:

With this experiment we are trying to evaluate performance of ArrayLists and LinkedLists. The program tries to perform this experiment by testing addToFront and insertionSort methods on each container. Main goal is to find out which container is better in specific case. For instance, in insertion of new element in front of the list may be easier for ArrayList and sorting could be easier for LinkedList or vice versa, or ArrayList could be better in both cases. My hypothesis is that LinkedList will be faster in adding new element in front than the ArrayList. As ArrayList needs to pull all its existing elements by one index right, it will need more time. Therefore, Arraylist's complexity for adding new element at index 0 will be O(n), while LinkedList's complexity will be constant O(1), as it needs just to change a reference to the next element. Although, linkedList needs less time for insertion at index 0, sorting will take less time for ArrayList than for LinkedList.

Approach:

As we are going to compare ArrayList and LinkedList, which are subclasses of AbstractList we can use inheritance. A new class IntegerContainer would have both methods - addToFront and insertionSort, and a data - class variable, that would store data structure(ArrayList or LinkedList), later running methods on those objects.



In order not to write separate timeAddToFront, timeSortOfUnsortedList, timeSortOfSortedList methods for ArrayListIntegerContainer and LinkedListIntegerContainer we could pass IntegerContainer type object to each of those methods, as both ArrayListIntegerContainer and LinkedListIntegerContainer are children of IntegerContainer, later passing them to those methods as arguments would not cause any error. Wheel class would provide random integers to fill our containers.

Methods:

For this experiment I used to test each of those three methods for different numbers of elements, several times for each. The initial value of number of items, in my case, was 100, that I multiplied on 2 each time until loop condition was valid. This way I used to test methods for following numbers of items: 100 * 2^0, 100 * 2^1, 100 * 2^2, 100 * 2^3… In addition, test class contained a constant which regulated when the loop had to end and at the same time showed the highest number of items, that the program could test for this specific case. For instance, if the constant equaled 5, I would run methods on IntegerContainers having following number of items: 100 * 2^0, 100 * 2^1, 100 * 2^2, 100 * 2^3, 100 * 2^4, where the highest number would be 100 * 2^4, the same as 100 * 2^(constant - 1). The first time I tried to assign 20 to the constant, so the highest number of items i would run test for would be 100 * 2^19, which is 52428800. Considering that my algorithm would run tests on containers with following numbers of items too: 100 * 2^0, 100 * 2^1, 100 * 2^2...100 * 2^18, the total number of elements I would use for my tests would be the sum of geometric sequence - Σ100 * 2^i, where i changes from 0 to 19. This test case failed, because the number of items was too big. After, running this case for approximately 12 hours I tried to use smaller number. My new constant was 15, but 100 * 2^15, was also a big number for my experiment. Finally, I started testing for lower numbers - 5 to 10(inclusive). I ran test for same constant value several times, following graph describes the results of the test. The numbers on the graph are average values of those tests' results:

Figure 1:

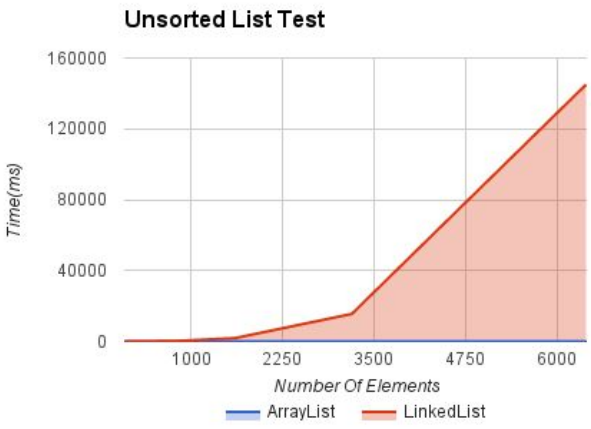


Figure 2:

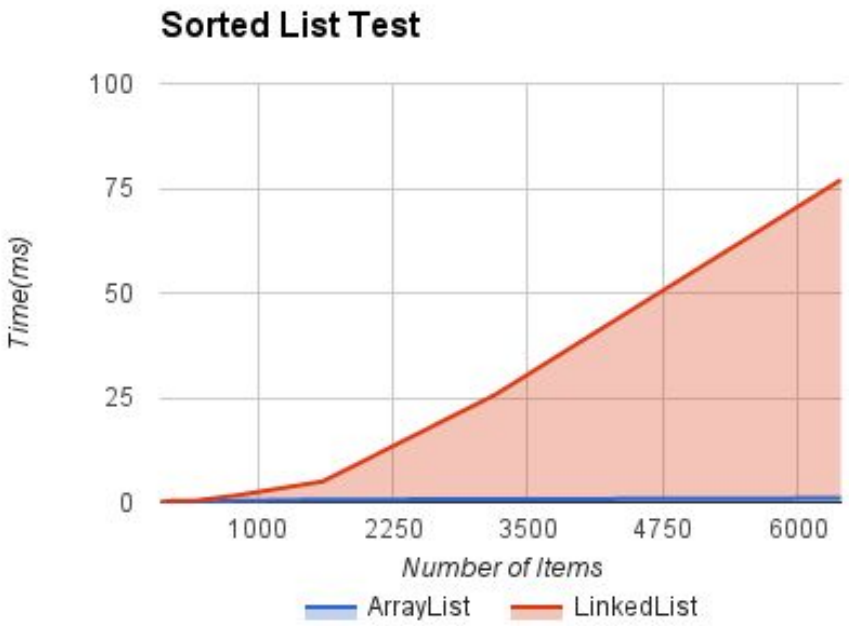
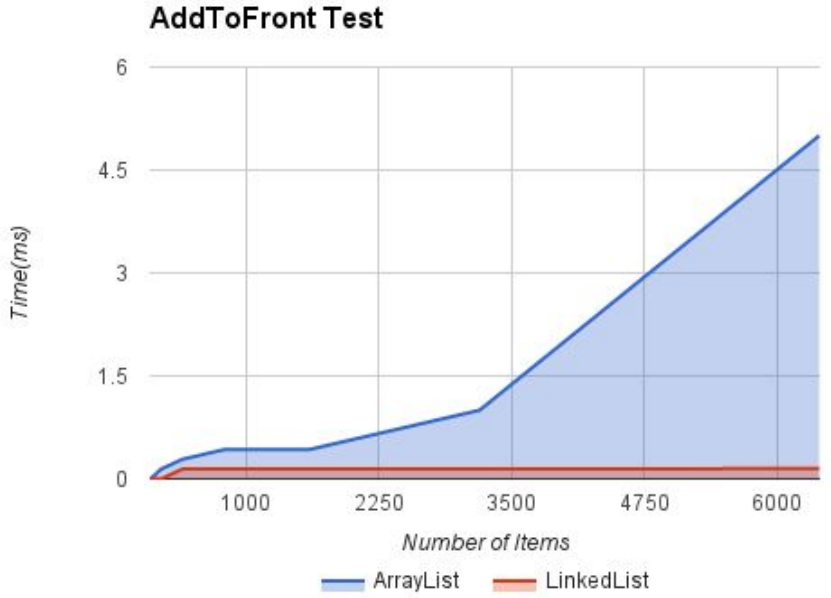


Figure 3:



After that, I created another testing algorithm by making initial number of items equal to 1000, and increasing it by 500 on each step. For instance, if the constant is 20, this would test for integer containers having 1000, 1000 + 500, 1000 + 2*500, 1000 + 3*500,

… 1000+19*500 elements. I ran this test the twice for different constants and the following graphs illustrate the average values based on both tests' results. **Figure 4:**

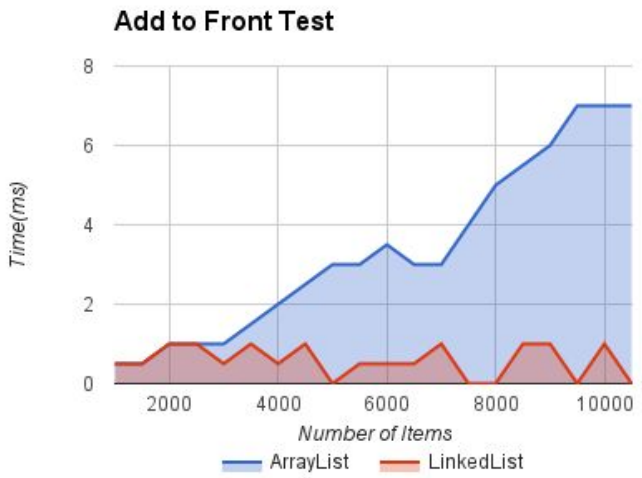


Figure 5:

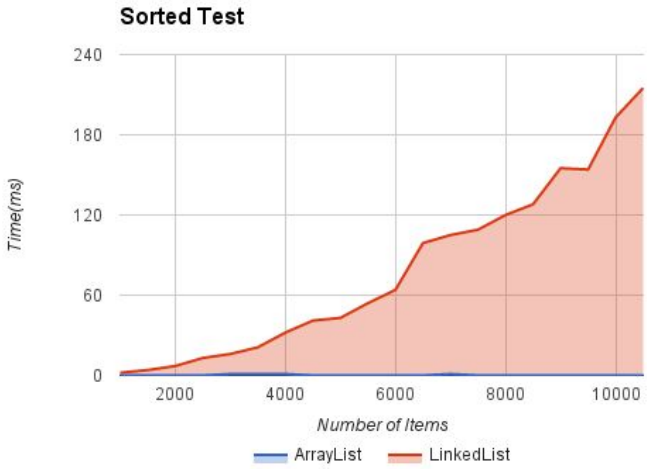
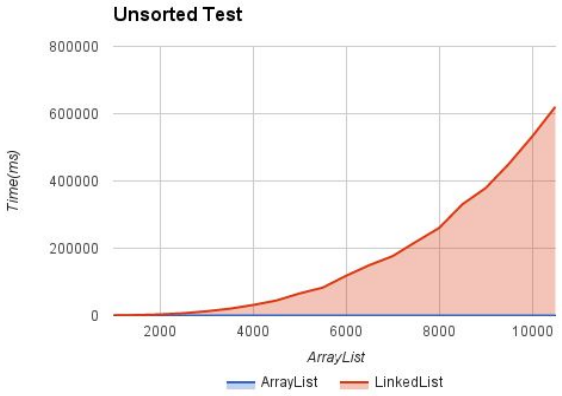


Figure 6



The first method gave me possibility to quickly increase the constant value and observe bigger numbers of items, I used that method mainly because I thought testing for many points would take more time, and multiplying constant by powers of two could give me bigger number in less time, though as I still had time left until the project submission, I tried the other method too, adding a constant value to my initial value instead of multiplication, this gave me more points in the same range. For instance, If I set my maximum number of items to 6000, the first method would give me 6 points(100,200,.. 3200), while the second method could provide 11 points(1000,1500...6000). Therefore this first method of solving this problem takes less time but provides less points than the second method of solution.

Data and Analysis

Both methods provide almost the same results. ArrayList needs more time to add new element in front. According to the Figure 3, complexity of adding new element in front of list is linear for ArrayList while it is constant for LinkedList. Figure 4 describes the same process too, time for addition in LinkedList is between 0-1 ms, while for ArrayList it increases with the number of items(there are some points where time is less than at the previous point, but in my opinion this was caused by other processes running on computer). According to Figure 1 and Figure 6, LinkedList needs much more time for sorting array, while ArrayList needs much less time. Figure 2 and Figure 5 show the same thing, that LinkedList still needs more time to sort sorted array, while ArrayList time is near 0(approximately between 0 and 5).

Conclusion:

In conclusion, answer to the question - which data structure is more efficient, depends on the case. As we saw above, inserting new elements takes less time for LinkedList than for ArrayList, and the complexity for this operation is constant for LinkedList and linear for ArrayList, as I assumed in my hypothesis. Although, sorting takes more time for LinkedList. The cause for this is the LinkedList's structure, each element in LinkedList contains reference to the next element, therefore we find an element by finding the previous element. This process provides O(n) complexity, while for ArrayList, which is based on array structure, getting element by index is easier and has O(1) complexity. So, why does sorting take more time for LinkedList, if replacement is easier? Sorting itself contains two parts - finding elements, than comparing and replacing with adjacent element if they are not at correct places. If we observe Figure 2 , we will find the answer. When we are testing for sorted list, there is no second part - program does not need to replace anything, which means the running time mainly depends on getting element from list and as we mentioned above, getting element takes more time for LinkedList than for ArrayList. This fact illustrates that getting element is a

key factor for defining the running time of each process. In my opinion, that’s why ArrayList is more efficient in sorting large amount of data than LinkedList.