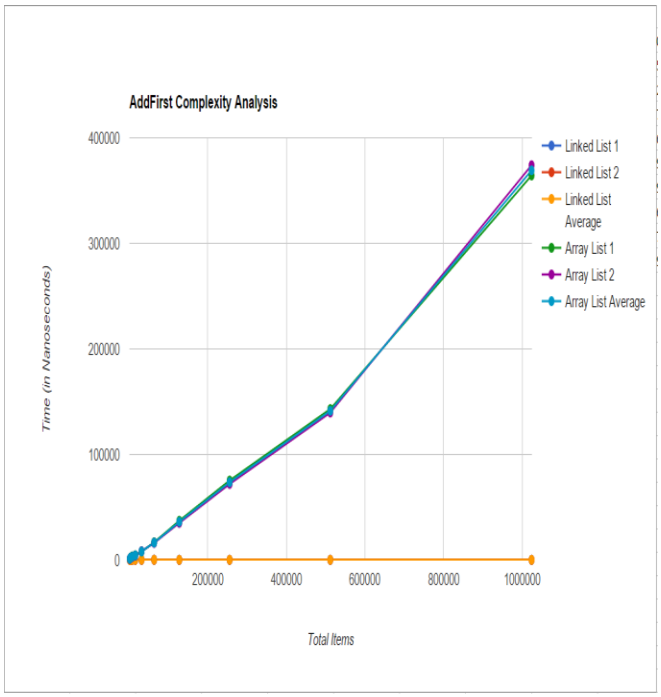
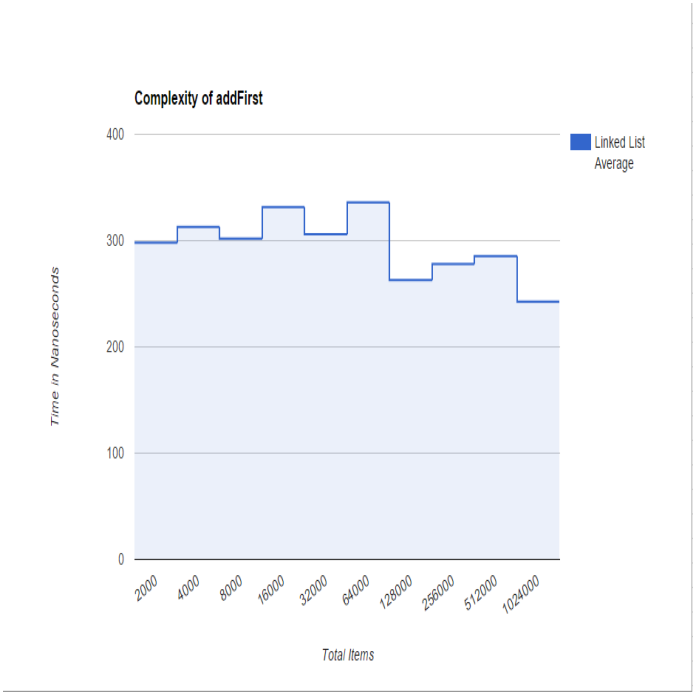
Alan Hansing
U0668729

1. **Collect and plot running times in order to answer each of the following questions. Note that this is this first assignment that does not specify the exact procedure for creating plots. You must design your own timing experiments that sufficiently analyze the problems. Be sure to explain all plots and answers.**

- **Is the running time of the addFirst method O(c) as expected? How does the running time of addFirst(item) for DoublyLinkedList compare to add(0, item) for ArrayList?**

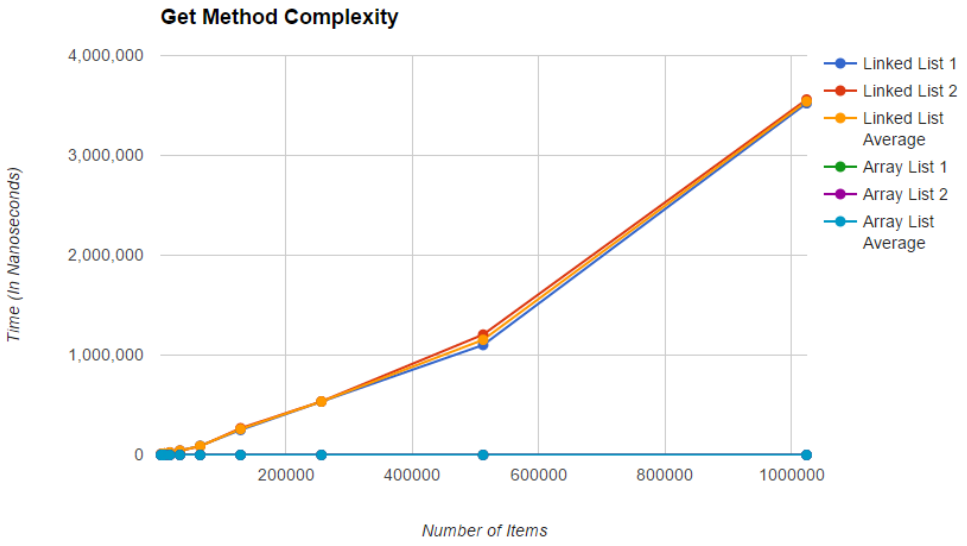



From the two charts above clearly demonstrate the O(C) of the addFirst function. The graph on the right shows that as the size of the LinkedList increases, the time to add a new first item varies between 200 nanoseconds and 400 nanoseconds, which is a margin of error that is easily acceptable with the size of our lists. The graph on the left clearly demonstrates the advantage of LinkedList over ArrayList when it comes to adding a new first item. Because the LinkedList needs to shift every item in the array over, it has a complexity of O(N).

- **Is the running time of the get method O(N) as expected? How does the running time of get(i) for DoublyLinkedList compare to get(i) for ArrayList?**

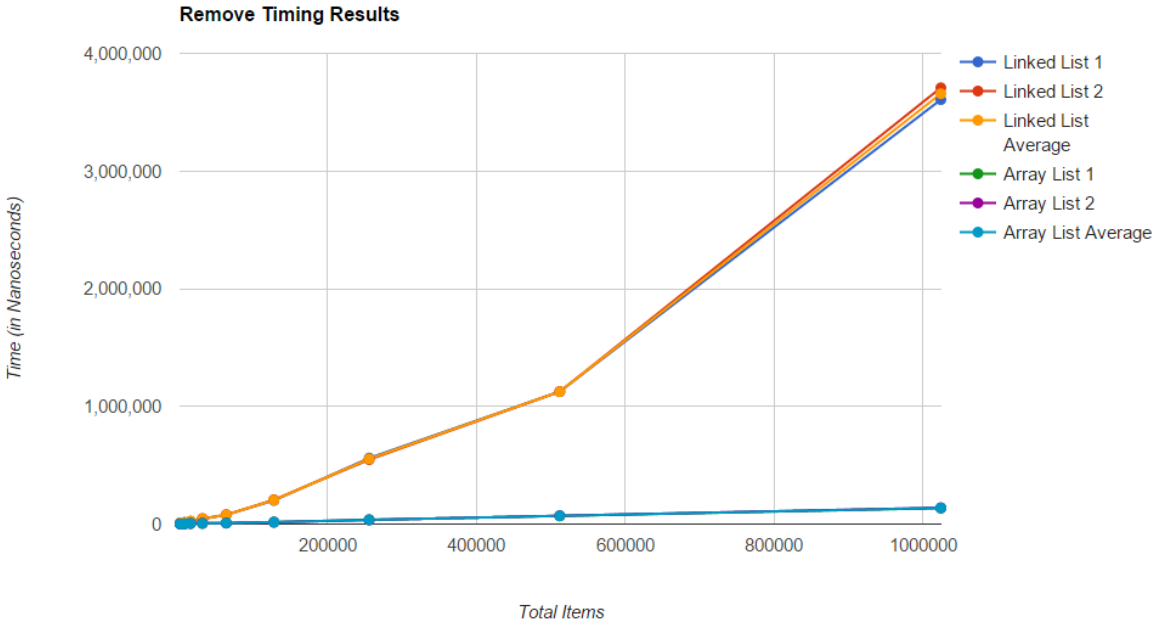
Alan Hansing
U0668729




The complexity of our get method was O(N) as demonstrated in the graph above. There is a slight hitch around 500,000 but overall the trend through one million items is linear. Due to the huge advantage of ArrayLists to be able to access any data point in the structure almost instantaneously they have a get complexity of O(C). In contrast, a LinkedList has to traverse the entire distance to the item you're looking for which gives it the O(N) complexity.

Alan Hansing
U0668729

- **Is the running time of the remove method O(N) as expected? How does the running time of remove(i) for DoublyLinkedList compare to remove(i) for ArrayList?**




The run time of our LinkedList was definitely O(N), again with a hitch around 500,000 but the overall complexity is linear. This is due to the need to traverse the List until we hit the index that we are removing, the actual remove function should be O(C) because it is only one action of removing that item from the list, no shift is necessary.

Not exactly sure how ArrayList works under the hood at removal, but it also has a linear removal rate, which is to be expected. As it can instantly access any data point O(C), but has to then shift everything over O(N). It has a much faster performance than our LinkedList, possibly by breaking it into smaller arrays? I don't know how ArrayList does its removal, but it is O(N) just like LinkedList, but quicker.

2. **In general, how does DoublyLinkedList compare to ArrayList, both in functionality and performance?**

As we have discussed in class, both have strengths and weaknesses that need to be evaluated to make an informed decision about which data structure is best in each situation. If we are always adding items to the front, then a LinkedList is the obvious choice. If we are only going to be adding things once but accessing them many times then an ArrayList would be more useful. It depends on the situational usage. There is no one "Best" Data Structure.

3. **In general, how does DoublyLinkedList compare to Java's LinkedList(Single Linked List), both in functionality and performance?**

   Doubly Linked Lists have the advantage of being able to traverse in both directions, which can cut down the time it takes to get to an index. Linked Lists would have to traverse from the front only because they don't keep track of a tail variable and there is not ".previous" to be able to go backwards. This could be useful in a situation where you don't really care about going both ways, you wouldn't have to keep track of as many variables and pointers. Other than that, I think a Doubly Linked List has all the function of a LinkedList but with the added capacity to go backwards.

4. **Compare and contrast using a LinkedList vs an ArrayList as the backing data structure for the BinarySearchSet (Assignment 3). Would the Big-Oh complexity change on add / remove / contains?**

   The complexity would definitely change, as the main ability of BinarySearchSet is based on being able to jump around to specific indices of an array, this does not work in a linked list. You would have to traverse to each point in list that you need to get to, adding a complexity of N items every jump. It would completely destroy the added optimization of a BinarySearchSet.

5. **How many hours did you spend on this assignment?**

   I would estimate that I spent anywhere from 3-5 hours on this assignment.