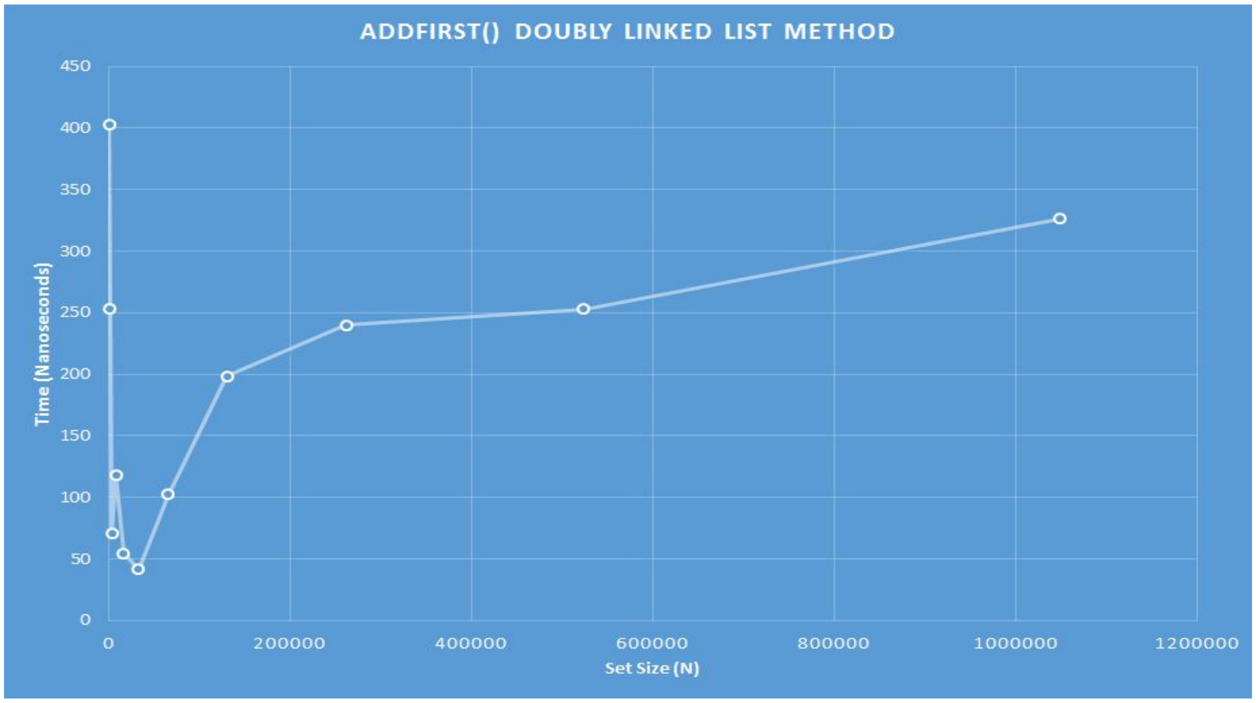
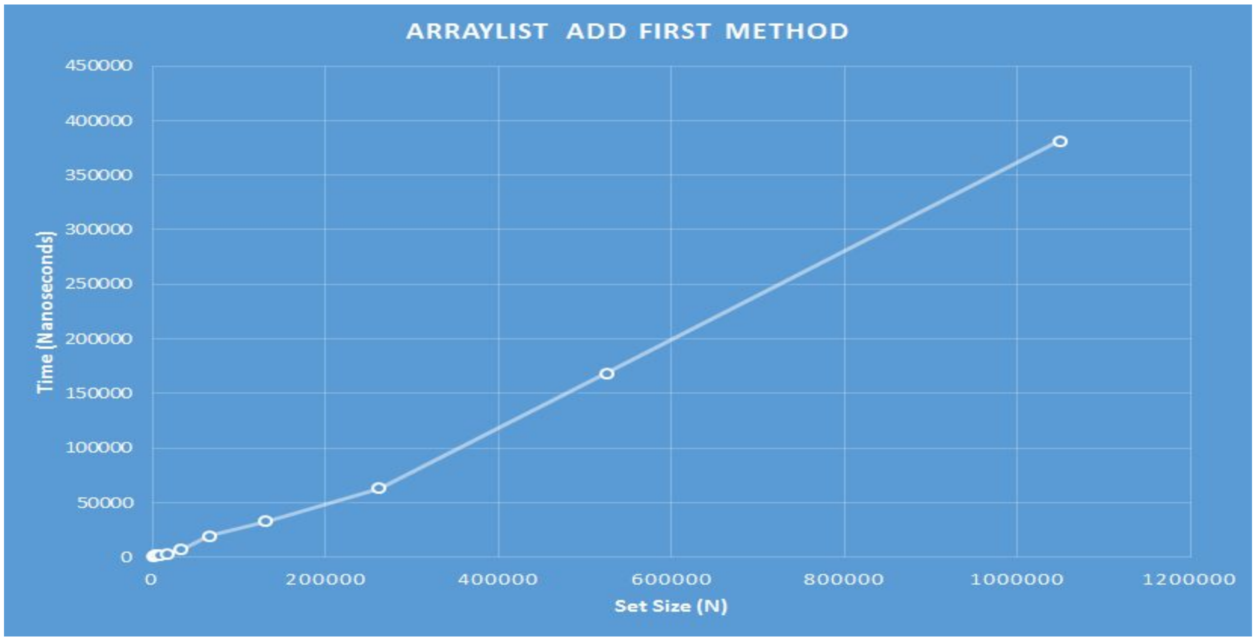
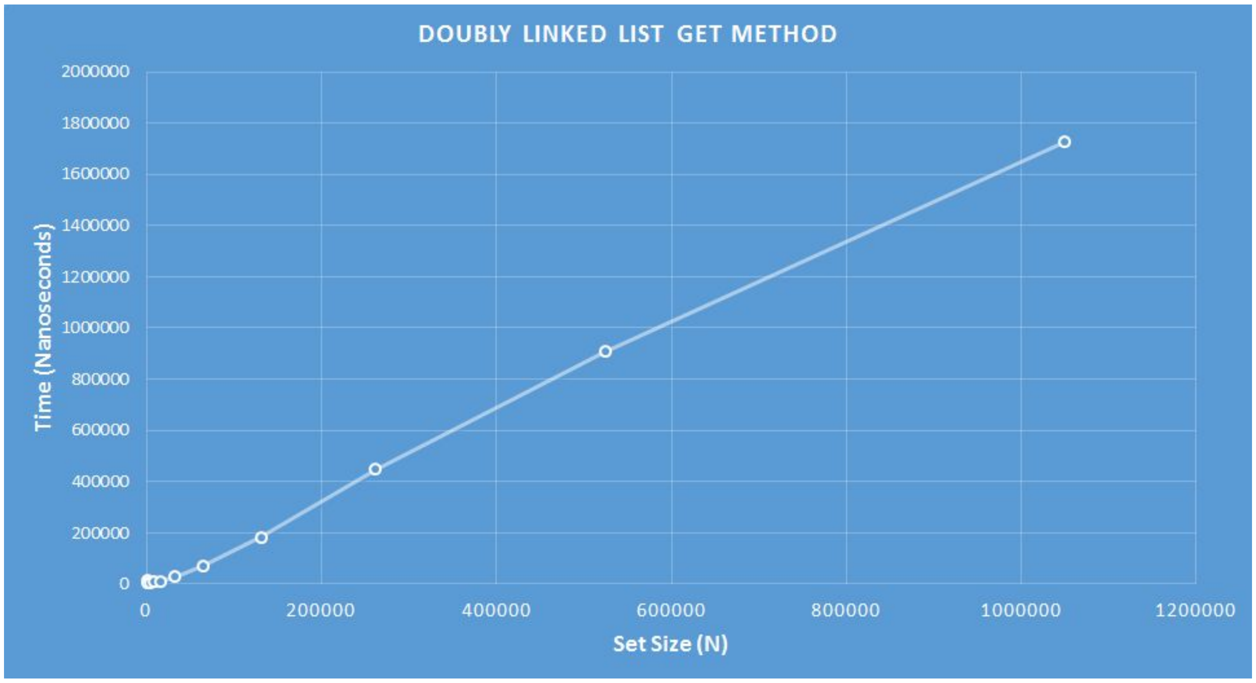
Cooper Pender (u0843147)
10/03/2016
Assignment 06 Analysis



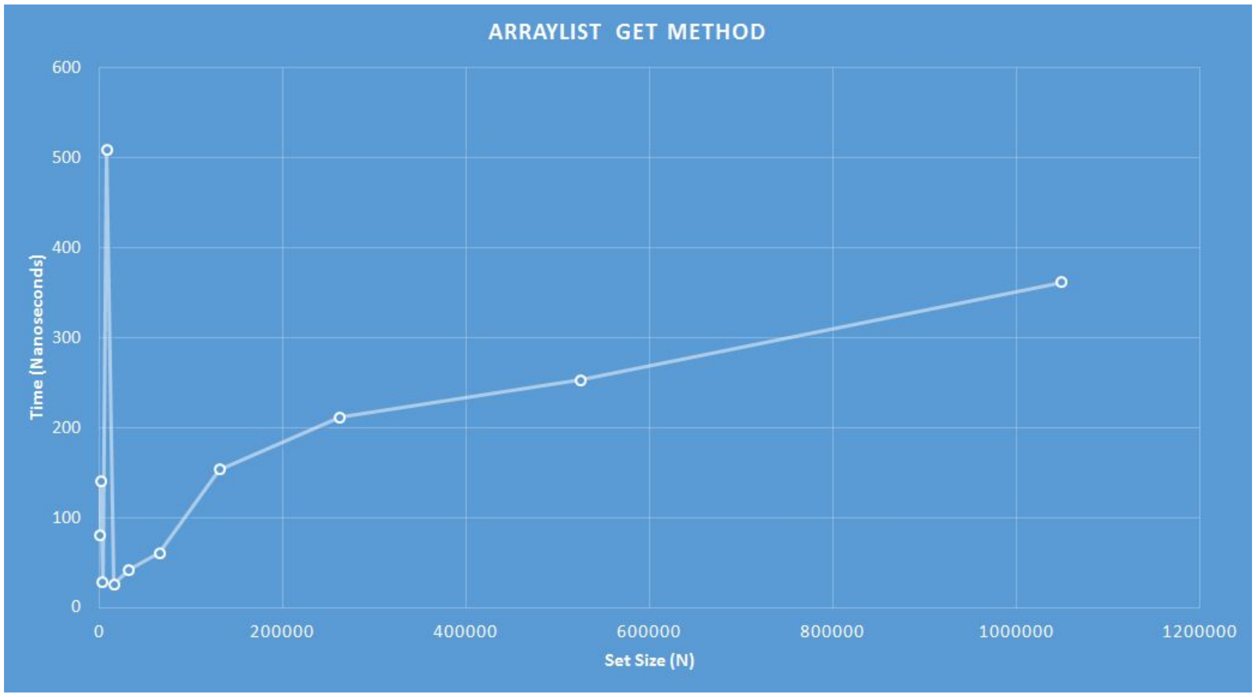
1. These running times **do** show that the addFirst method shows a **O(c)** complexity. The time on the left is in nanoseconds and the graph follows no defined pattern.

In comparison to the Doubly linked list, adding at the first element in an ArrayList exhibits behavior much closer to a complexity of **O(N).** This is because regardless of how many items are in the set, adding to the beginning of an ArrayList requires shifting everything over to make room for the new element, resulting in **O(N)** complexity.

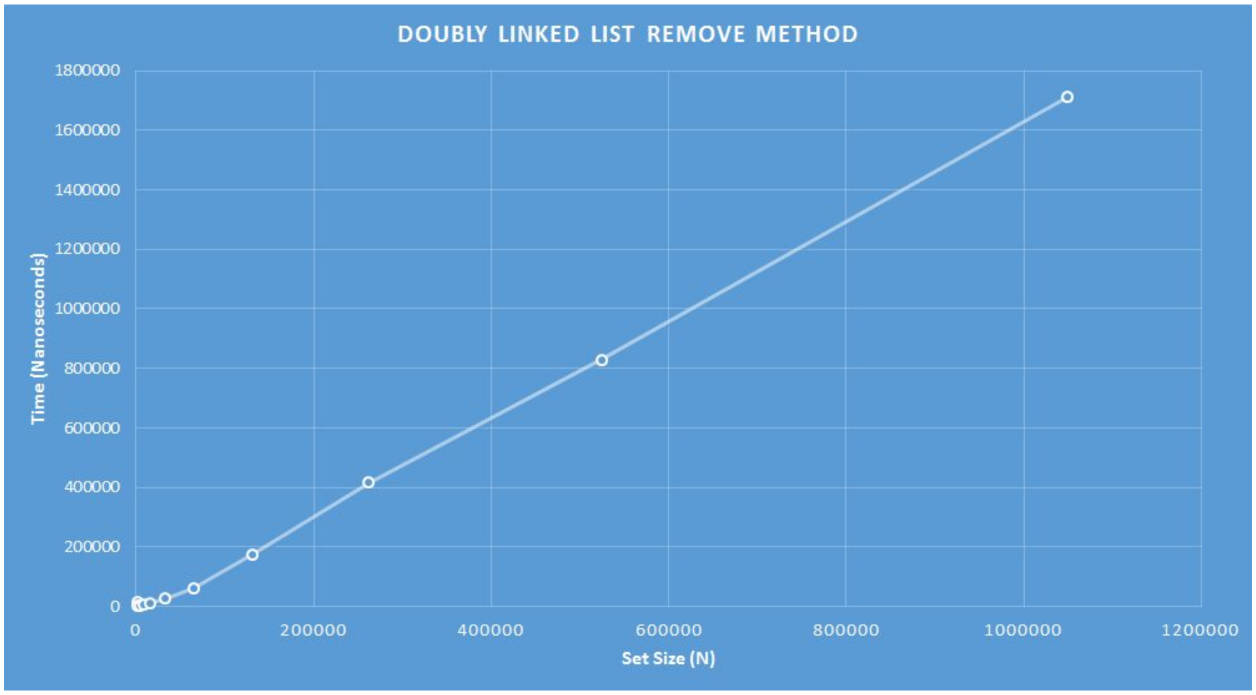




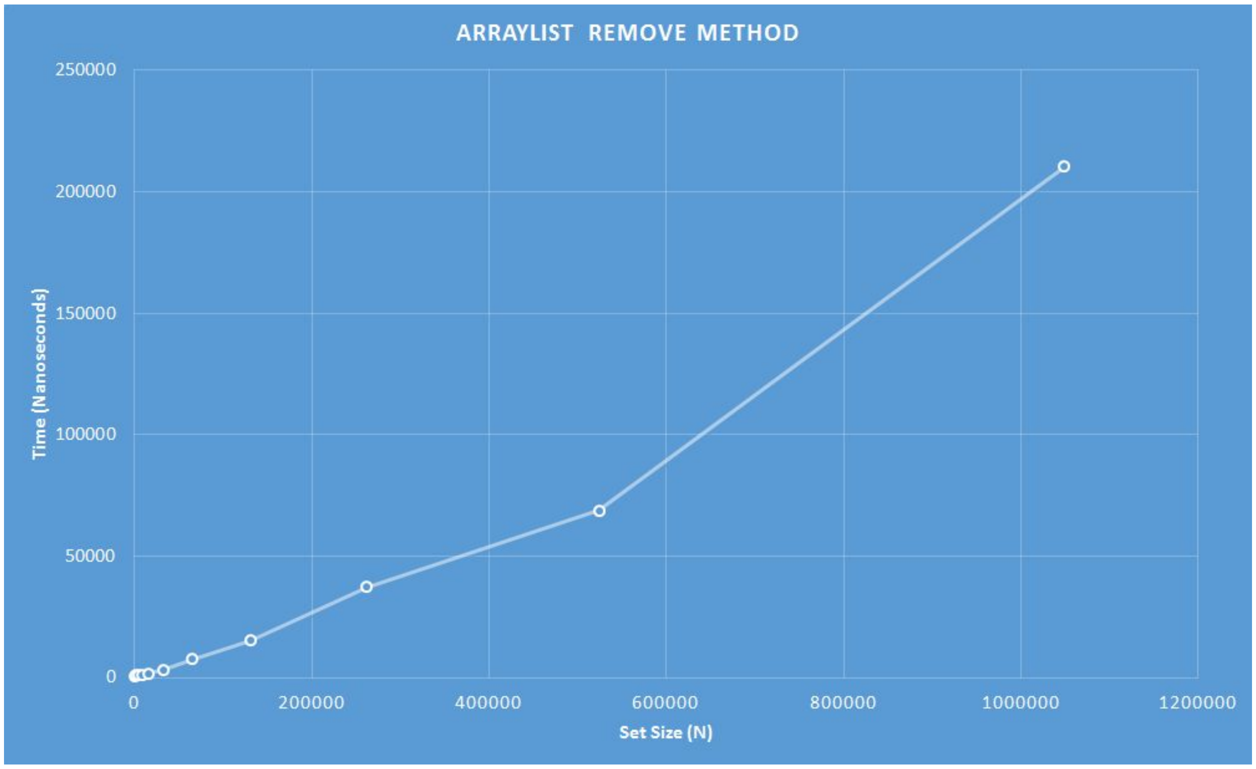
The get method for a doubly linked list is as expected. It shows a **O(N)** complexity.



The get method for an arraylist shows that it has a complexity of **O(c)** as expected. The get method for a doubly linked list has a higher complexity due to having its data **not** stored continuously.



The complexity of the remove method for a doubly linked list shows a complexity of **O(N).**



The remove method for an ArrayList also shows a complexity **close to** but not quite of **O(N)**. This data would be closer to **O(c)**. This has a higher complexity than a normal O(c) due to having to shift things over in the array to rearrange it after the removal.

2. In general each a doubly linked list compared to java's arraylist has pros and cons. Depending on the operation you are doing you might want to reconsider which data structure to use. For example if you are adding or removing from the beginning of a data set a doubly linked list will prove much more useful time wise since it does **not** require all following elements to be shifted over to make room for the insertion of removal. In these cases doubly linked lists show a **O(c)** complexity whereas arraylist shows a **O(N)** complexity.

Another situation which favors arraylist is calling the get method. Since data in an arraylist is continuous (next to each other) in memory, it's much quicker to lookup a index and find the element there which leads to a **O(c)** complexity. For a doubly linked list however you have to iterate through the data to find that specific element, this leads to a **O(N)** complexity.

3. A doubly linked list compared to a singly linked list offers several benefits. Having pointers to the start **and** end of a list allows for traversal both ways across the list. This allows you to optimize some of the methods in a list, such as the *get()* method, where you can traverse from front to back **and** back to front, to more quickly find the desired element.

The biggest downside to using a doubly linked list compared to a singly linked list is implementation time and difficulty. With a singly linked list you only have to worry about the *head* node and a next pointer reference, whereas with a doubly linked list a *head* and *tail* node need to be kept track of, along with forward and backward pointers.

4. For our binary search assignment the methods described **would change.** Since for a backing data type using an array, the add and remove methods have a complexity of **O(N),** but if you were to use a linked list instead, would change that complexity of **O(N)** to **O(logN)**. This is becuase add and remove for a linked list have a complexity of **O(c)**, so the remaining complexity for add and remove would be **O(logN)**.

The contains method would also change. Since getting an element in an array is **O(c),** the complexity of the contains method would go from **O(logN)** due to the binary search, to **O(N)**. This is because finding an element in a linked list requires traversal through that list, and can not just be looked up.

5. I spent around 13 hours on this assignment.