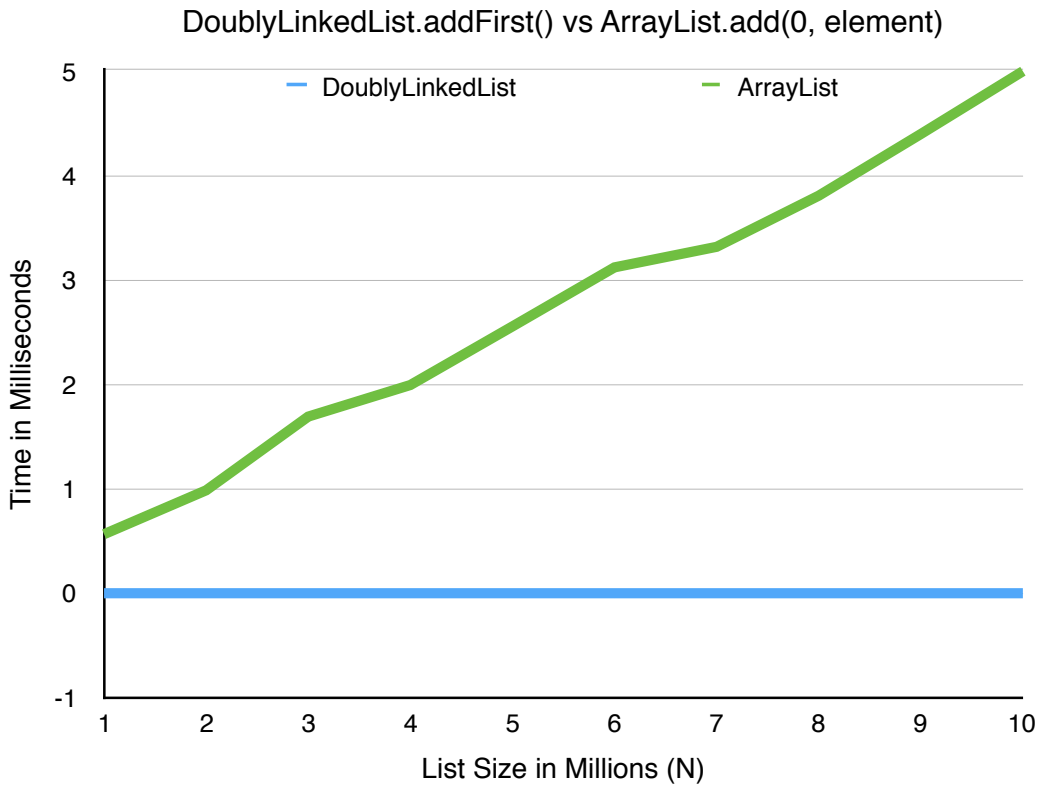
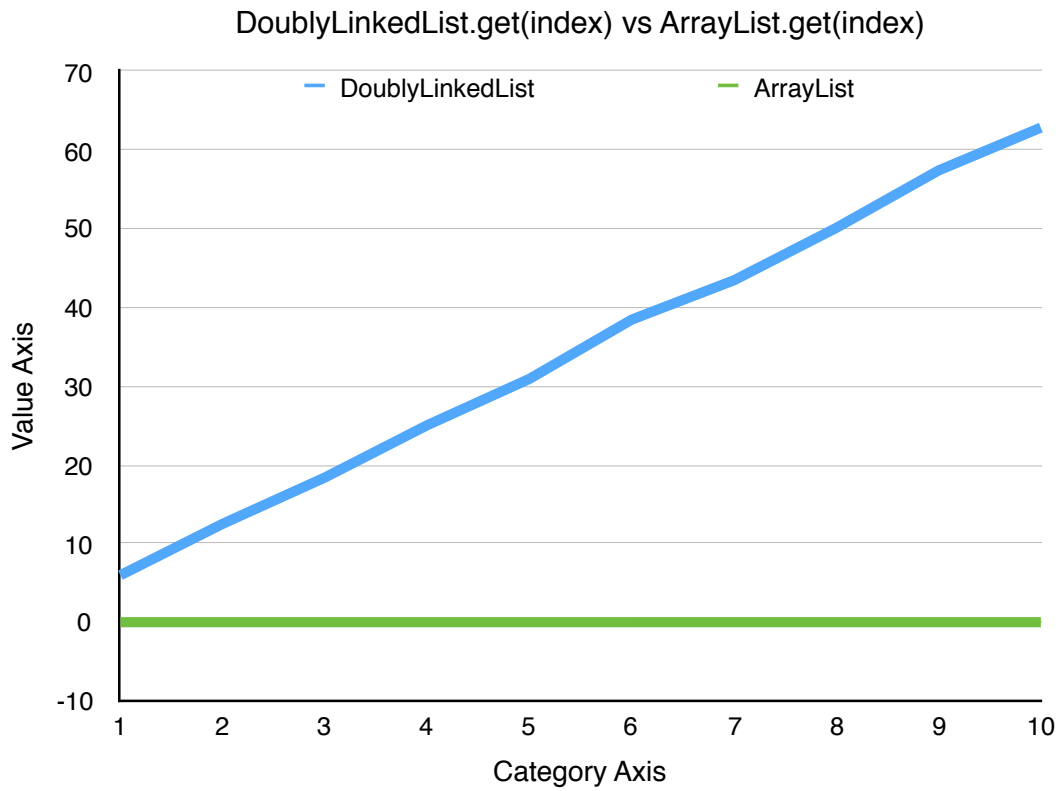
Samuel Teare
UID: u0663592
Assignment 6 Analysis
10/5/2016

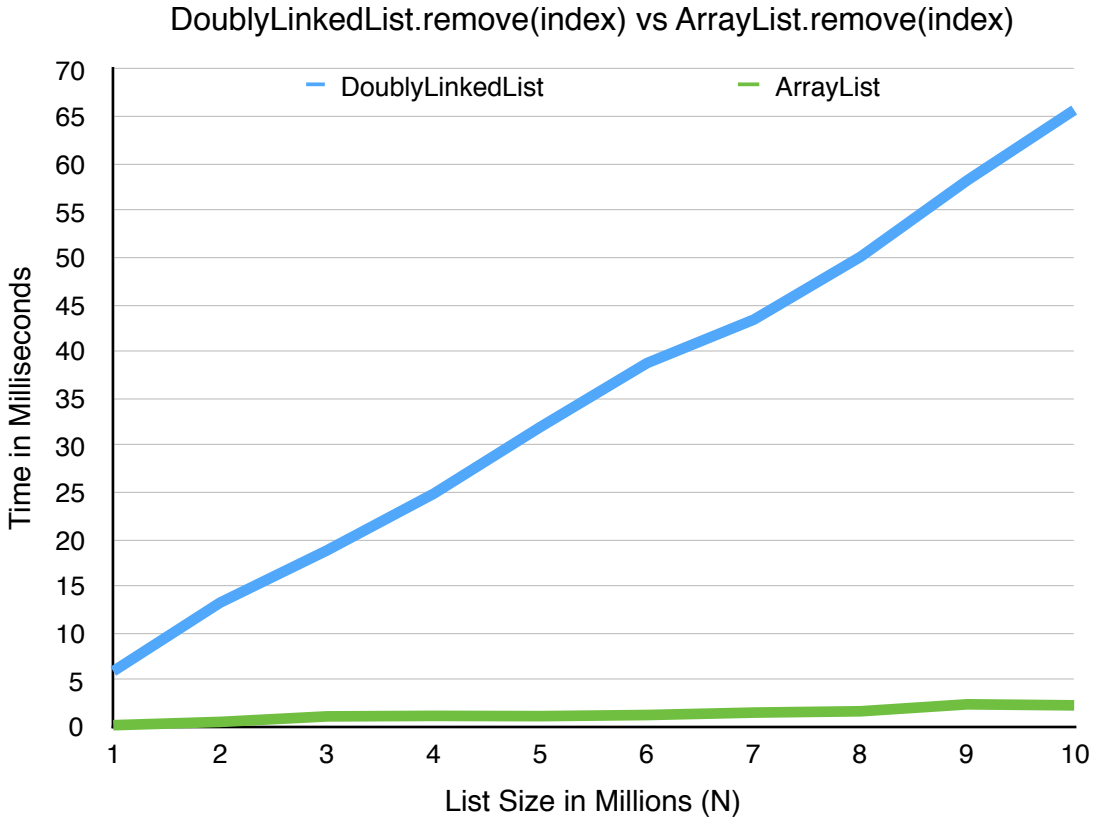
1. The run-time for the addFirst method in the DoublyLinkedList class I implemented was O(c) as it should be. I compared the run-time for this addFirst method with the run-time for adding an object to the front of an ArrayList over 10 different list sizes ranging from 1 million to 10 million. As shown in the graph below, the addFirst method for a DoublyLinkedList was O(c) and the much better choice compared with the O(N) run-time of add(0, element) in an ArrayList.



My implementation of the get method for a DoublyLinkedList had the expected run-time of O(N). I also compared this to the similar method in an ArrayList. This time the run-times were the reverse of the addFirst. The ArrayList showed O(c) and the DoublyLinkedList was O(N). It did was not affected by the size of the list as it can quickly access any item in the list with the index. It is obviously the quicker answer to retrieve an object from an ArrayList, than it is for a DoublyLinkedList. I compared both get methods in attempts to get the item stored at the middle of the list.



My implementation of the remove method for a DoublyLinkedList also had the expected run-time of O(N). In the comparison with the ArrayList remove method shown in the graph below, the ArrayList remove was also O(N), but had a vastly superior run time.



2. After reviewing the comparison between DoublyLinkedList and ArrayList, they both have performances that are better than the other. The DoublyLinkedList is most efficient at adding something to the beginning of the list or even to most places within the list because it does not have to shift everything else. It is also more efficient at increasing its size as it does not have to run through a lengthy process of doubling the size every so often as compared to the ArrayList. The ArrayList was more efficient at locating an item, O©, while DoublyLinkedList is linear, O(N). ArrayList was also more efficient at removing an item. Both ArrayList and DoublyLinkedList had linear behavior, but DoublyLinkedList had a much steeper linear behavior. Both lists have similar methods and functionality. ArrayList does have more methods for dealing with a large number of objects such as removeALL, addALL, and removeRange. It might be convenient to add similar methods for DoublyLinkedList.

3. DoublyLinkedList and Java's LinkedList are doubly-linked list, which allow both of them to move through the list from front to back and back to front. LinkedList contains all of the same methods that DoublyLinkedList has. In addition to these methods, LinkedList has several methods that would be very conducive to a Stack, including pop and peek. As both lists are doubly-linked, they should have similar, if not the same run-time behavior. The biggest run-time behavior any of the methods for either list should have is O(N). The DoublyLinkedList that we wrote is a basic version of Java's LinkedList.

4. Using a LinkedList or an ArrayList to back the BinarySearchSet would have different effects on the efficiency. Using an ArrayList to back the BinarySearchSet would have the same Big-Oh complexity for add, remove, and contains as the original backing with an Array. It would be O(LogN) for contains, and O(NLogN) for add and remove. If the BinarySearchSet was backed by a LinkedList, it would not be possible to have the desired O(LogN) for the contains method. This is due to a LinkedList having an O(N) for accessing any object contained in side it, other than the first or last element. The adding and removing of objects would have better efficiency to an ArrayList as these two methods would be O(N) to add or remove. I think that the important part of a BinarySearchSet, is to implement the binary search when searching for an object. Since this would not be plausible with a LinkedList, it would be best to stick with an ArrayList to back the BinarySearchSet.

5. I spent between 10 and 12 hours on this assignment.