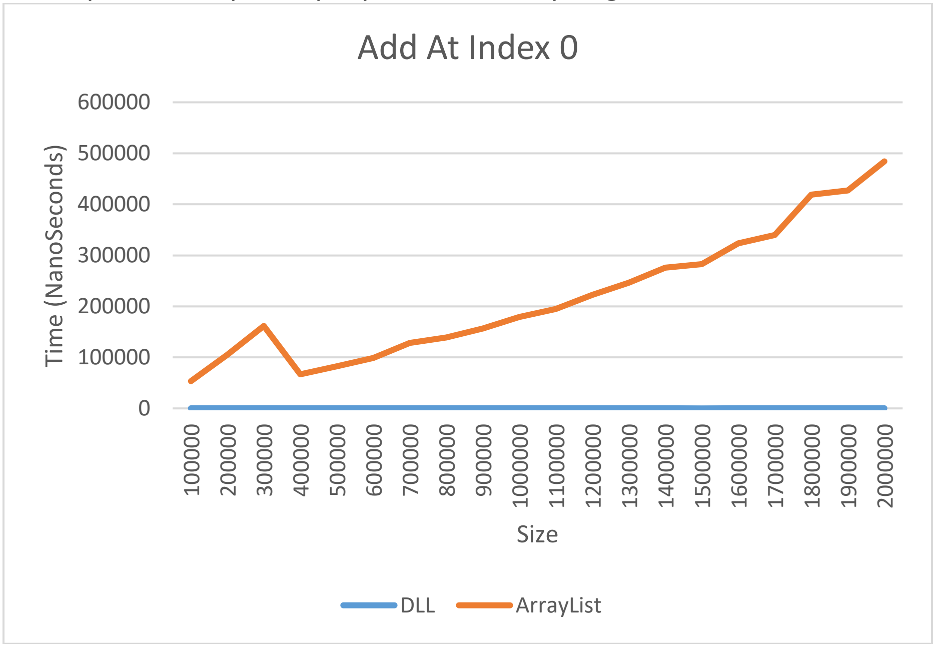
Kameron Service
U0963620

Assignment06 Analysis

1. a) Yes the run time for addFirst was O(c). The reason it is constant, is because there is only on operation every time. You never have to traverse the list or shift the elements in the list, you are only changing the head and a few other references.

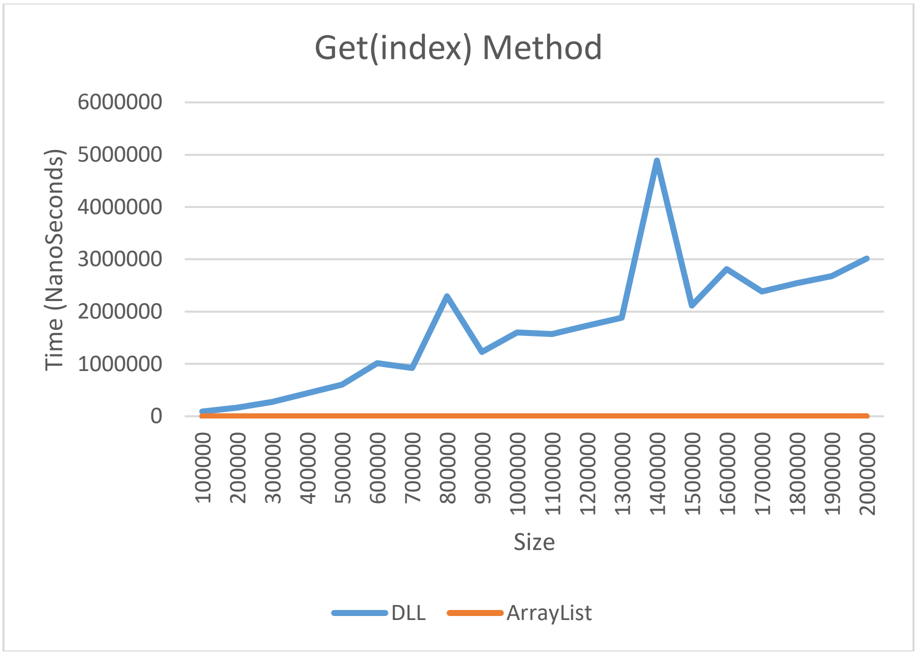
   AddFirst(item) method for DoublyLinkedList is a O(c) whereas add(0, item) for ArrayList is O(N). In an ArrayList, in order to add an element at index 0, you must shift every element in the arrayList over one spot. The act of shifting every element is what gives this method an O(N). If

   If you're adding to the beginning or end of a list, using a DoublyLinkedList is a much faster process, especially if you have a very large N.



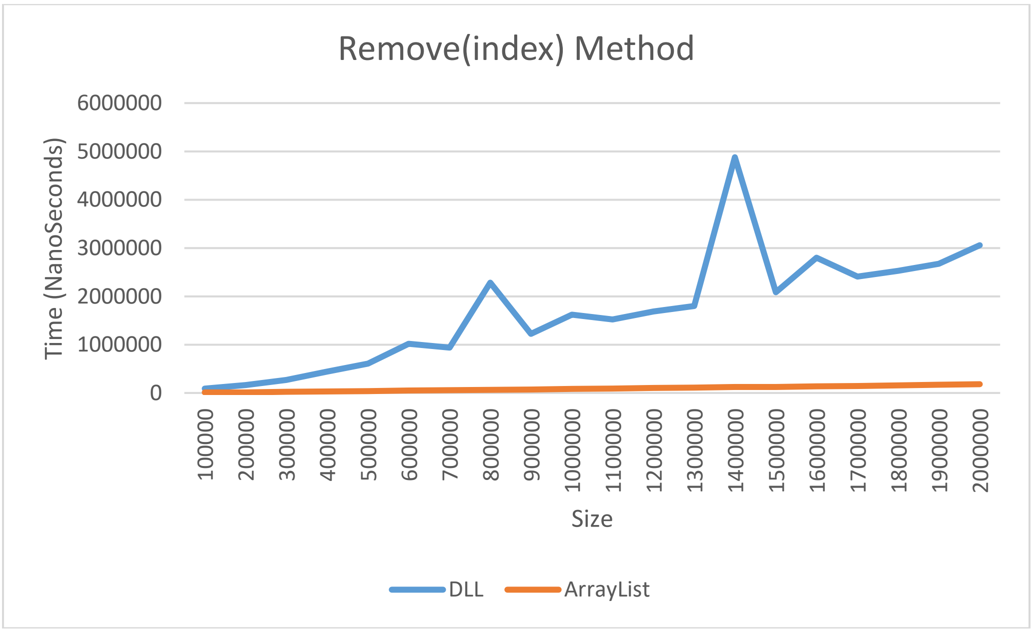
b) Yes the run time for get(item) is O(N). The reason is because unlike the addFirst method, where you know exactly where you're adding, the get() method must traverse the entire list in order to find the element. Using a DoublyLinkedList instead of a SinglyLinkedList allows you to optimize the get() method a little bit. Being able to work backwards from the tail allows you to find an element at an index closer to the tail much quicker. The complexity with this optimization is N/2, but that gives us the same big-O notation of O(N).

The running time of get() method for DoublyLinkedList is O(N), whereas the running time for ArrayList is O(c). Unlike our DoublyLinkedList get() method, the ArrayList get() method doesn't need to traverse the whole list to find the index. It has values stored at each index so it is much quicker to find the element at that index. For our get() method, we have to traverse the whole list index number of times to find that index.



c)Yes our remove() method is O(N). The reason this is the case, is the same reason our get() method was O(N). In order for us to remove an element from our DoublyLinkedList, we must first find the element at the certain index. We have to traverse the whole list until we reach the correct index. Having to go through each element until we get to the correct index is why our remove method is O(N).

The complexity would both be the same O(N). An array list can find the value to remove in constant time, but then must shift all the elements behind it down one spot. A doublyLinkedList takes longer to find the element, but once it finds the element it doesn't have to do anything that isn't constant time.



2. DoublyLinkedList are faster than ArrayLists in some aspects, and slower in others. Areas where doublyLinkedLists out perform ArrayLists would be insertion and removal of an object. For an arrayList to do either of these things, it would have to shift every element that comes after it. A doublyLinkedList performs so well because there is no shifting of any kind. It is simply changing a few references which is a constant speed.

   The areas where ArrayList is quicker are in get() and set() methods. An array list doesn't need to go through the list to access an item at a certain index because every element has a corresponding index. So accessing these items are done at constant speed. A DoublyLinkedList on the other hand, must go through each element in the list to find an element at a given index.

   Every difference between the two stems down to having the arrayList backed to an array. To find certain elements, you just call their indices. A DoublyLinkedList does not save each element with a corresponding index. It has one element point to the next and the next element point to itself. The only way to find elements at certain indices is to step through each element to the next.

3. DoublyLinkedLists have one distinct difference between the two. Singly linked lists can only traverse in one direction, first-last, whereas DoublyLinkedLists can move in both directions, first-last and last-first. They are very similar in functionality, but not performance. DoublyLinkedLists can complete task much quicker because they can start at a tail and work back, or start at a head and work forward. DoublyLinkedLists are essentially just an optimized SinglyLinkedList.
4. If DoublyLinkedLists were used as our data structure for binarySearchSet, we wouldn't even be able to use Binary Search. Binary search is all about taking half of the list setting that to equal our index and determining if we should go up or down. We wouldn't be able to use binary search because DoublyLinkedLists don't have easily accessible indices. You can't just take half of a DoublyLinkedList and start there. You would have to step through each element in the list to get to that point.

   Now with that being said, if you used a DoublyLinkedList, your add method would be faster, however, the big O complexity is O(N) for both. The add complexity would be O(N) because it can step through until it sees that the next element is greater than itself, and the previous element is less than itself. It doesn't need to find, then shift the elements. The add complexity of an arrayList using binary search is O(N) because finding the place has a logN complexity, then shifting every element down has a N complexity and N + logN = O(N).

   The complexity of the remove method would the same as the add method. For an arrayList, to remove the item, it has to find the element (O(logN)) and then shift every item that follows down (O(N)) giving it a complexity of O(N). For a DoublyLinkedList, you have to find the element to remove and that's what gives it the complexity O(N); stepping through every element until it's found.

The contains method would be faster using an arrayList. Being able to use binary search to find an element makes your contains method by O(logN) whereas a doubly list would be O(N) because it has to step through each element until it finds the target.

5. I probably spent around 10 hours on this assignment.