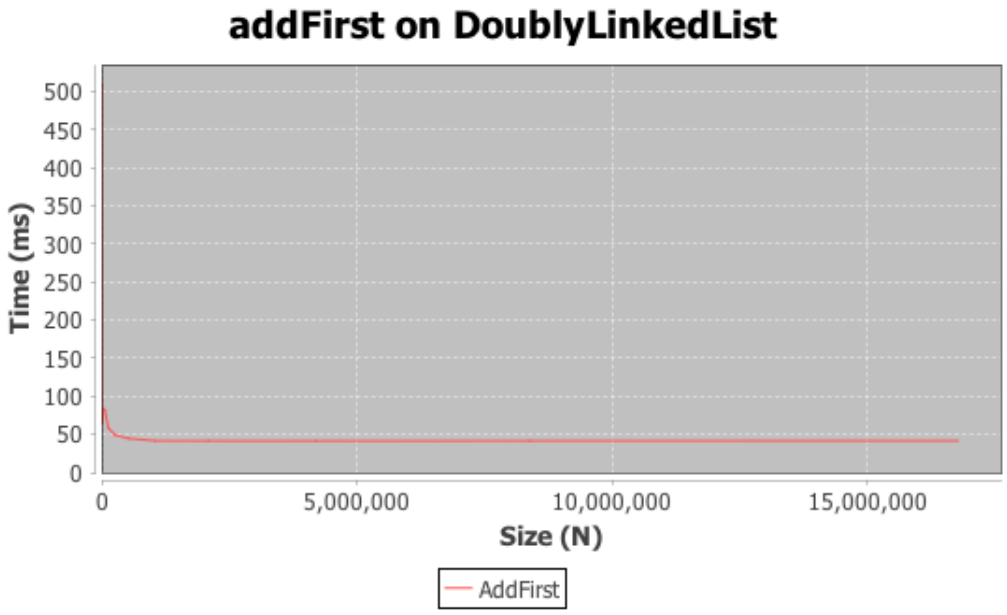
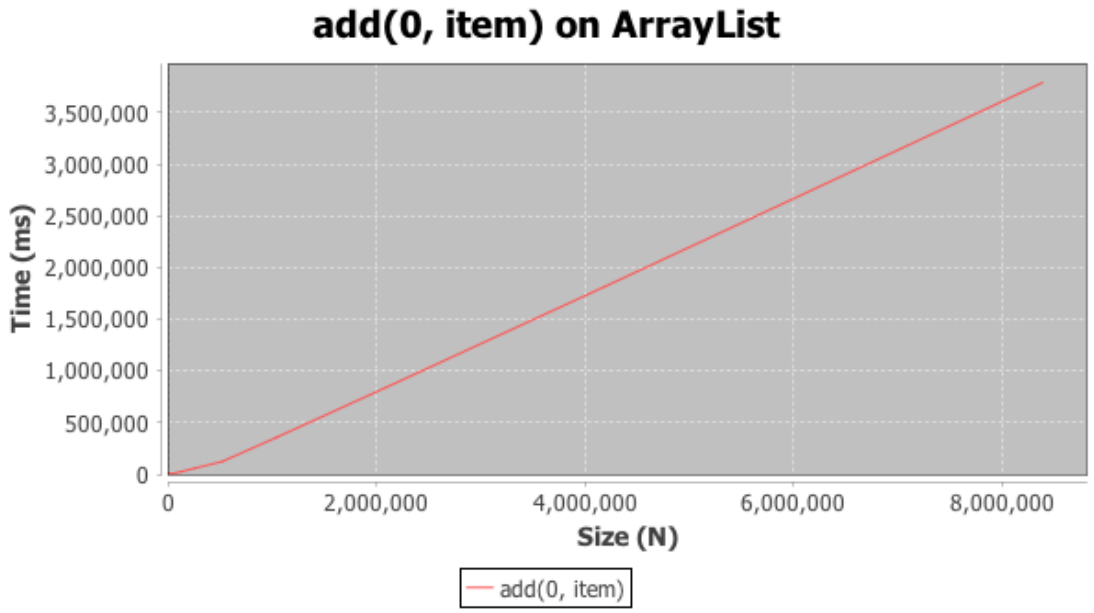
Kyle Price
U1018878
Assignment06
Analysis document

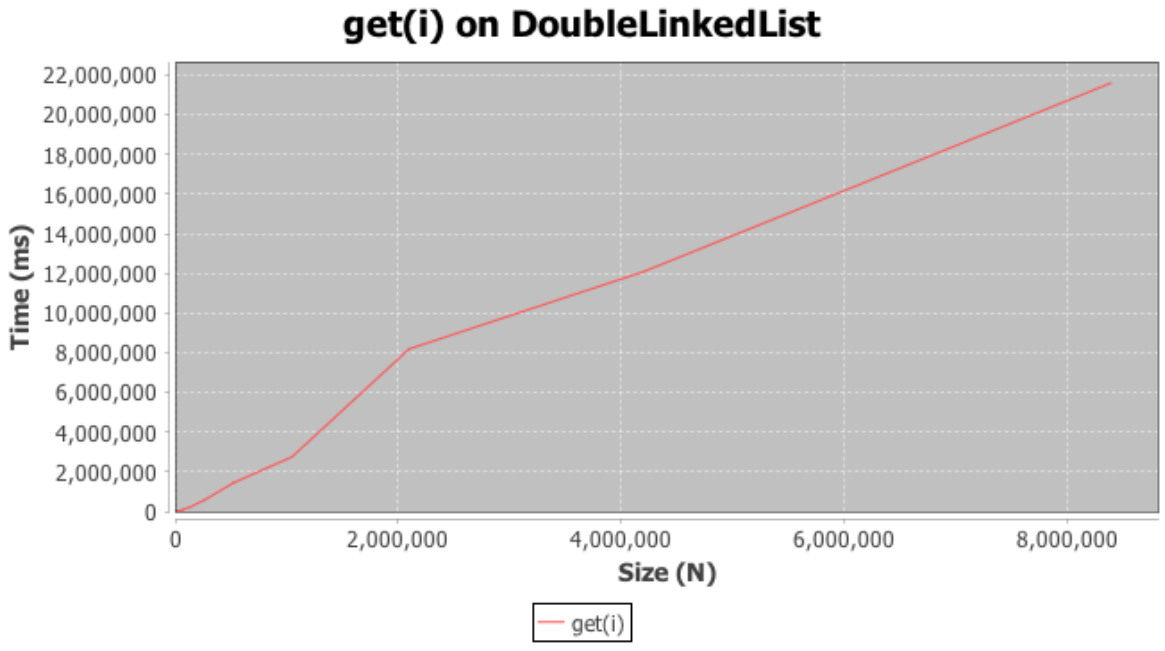
1. Collect and plot running times in order to answer each of the following questions. Note that this is the first assignment that does not specify the exact procedure for creating plots. You must design your own timing experiments that sufficiently analyze the problems. Be sure to explain all plots and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

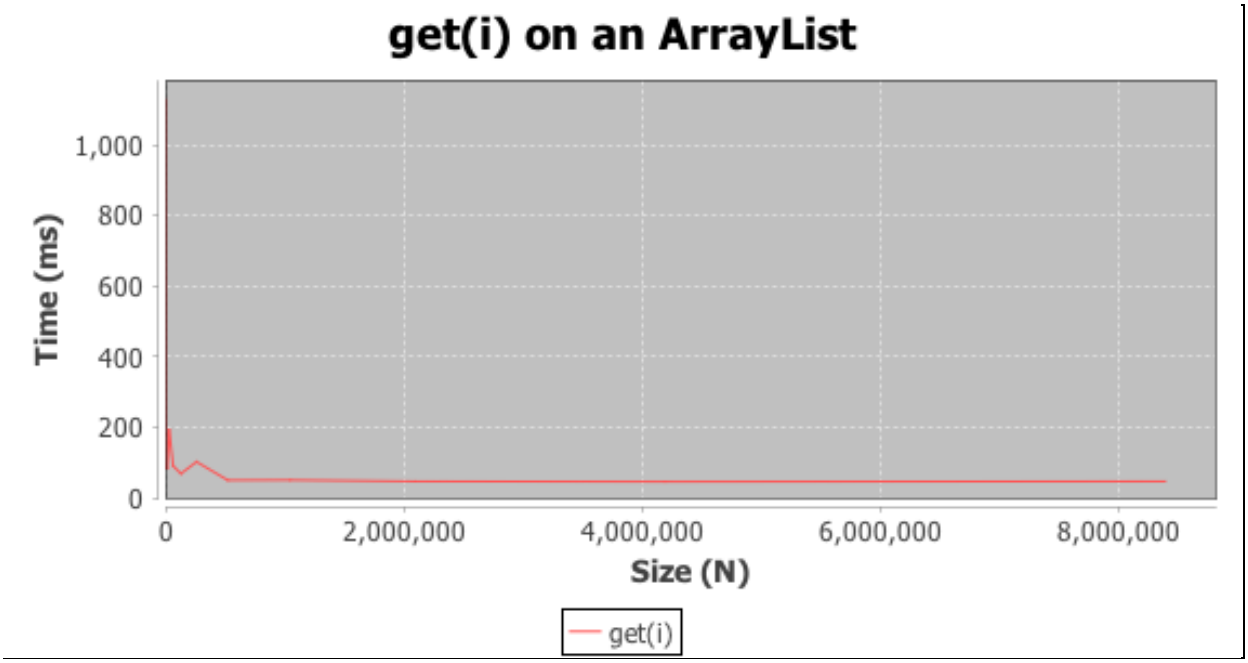
Is the running time of the addFirst method O(c) as expected? How does the running time of addFirst(item) for DoublyLinkedList compare to add(0, item) for ArrayList? I split the first question into three parts for clarity. Yes. The expected time was O(c) and that's exactly what the timing experiments showed. Essentially it didn't matter how large the list was that we were adding to. The time stayed constant. The difference between my addFirst and ArrayList's add(o, item), is after the ArrayList inserts the item in the correct spot, it has to move along the length of the list and shift everything over one index. That means it's running time is an abysmal O(N). My addFirst simply has to do some reference swapping. Therefore, it's running time is O(c). You can see both of these running times neatly plotted below.



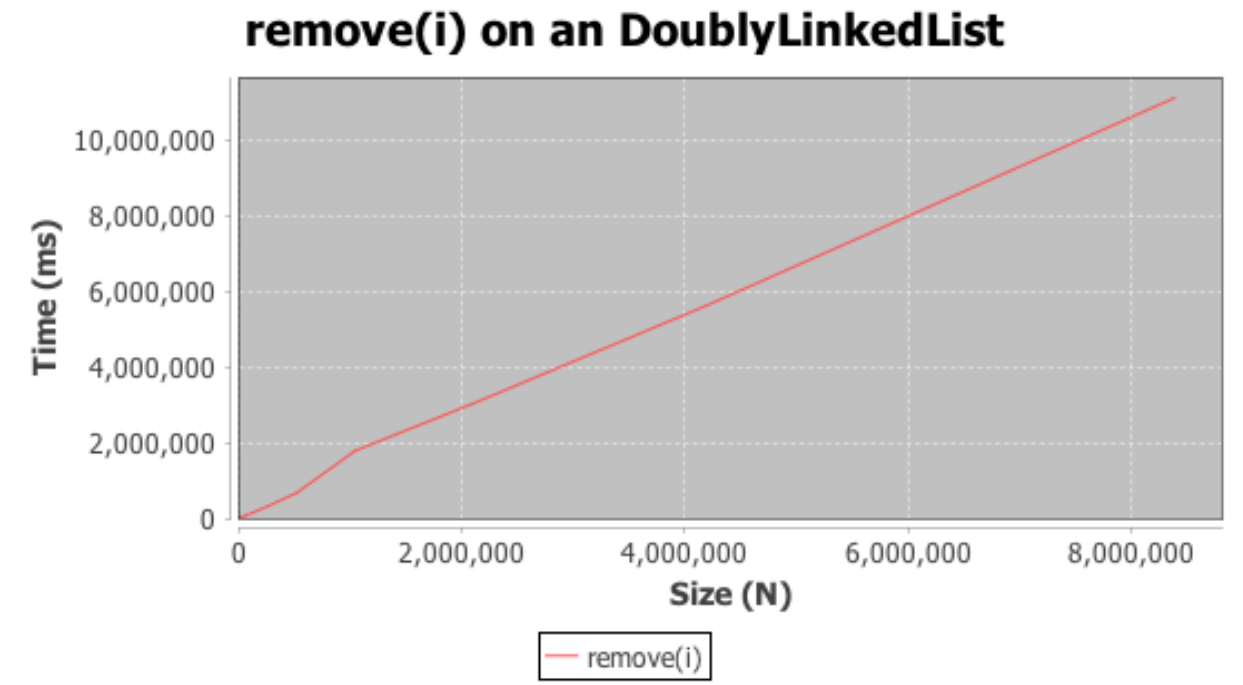


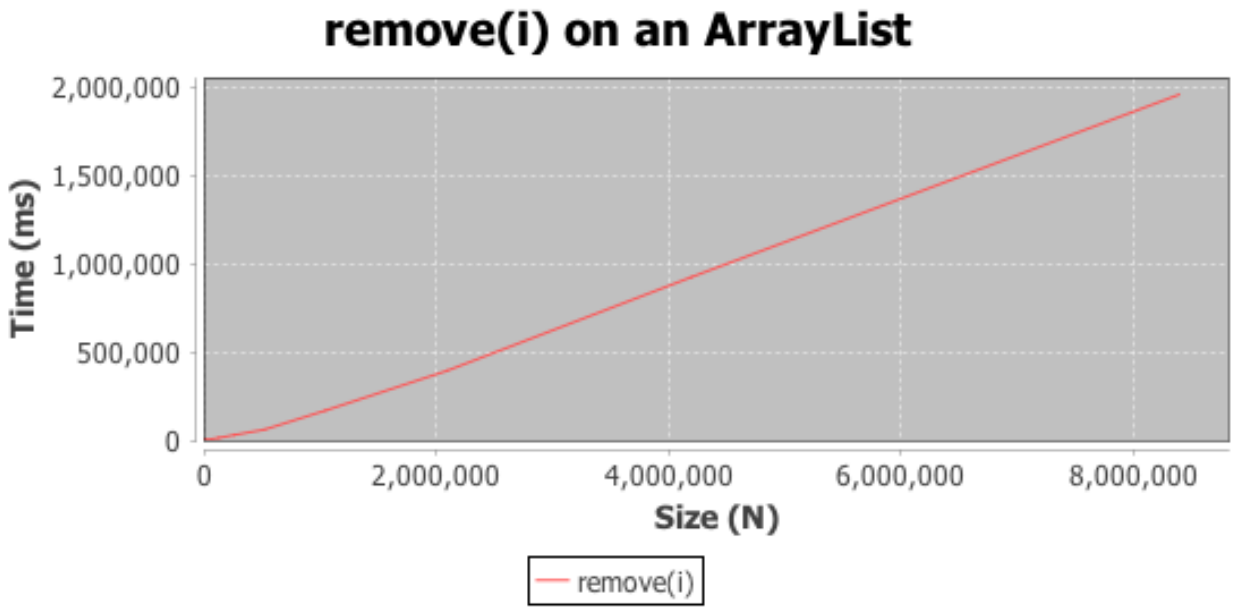
Is the running time of the get method O(N) as expected? How does the running time of get(i) for DoublyLinkedList compare to get(i) for ArrayList? Yes. As you can see from below, the runtime looks rather linear. The reason for this is that on a linked list, you have to iterate through the list to get an item. You can't just index into the middle like an ArrayList. Java's ArrayList performed get(i) in O(c). If accessing items in the middle of the list is going to be necessary, an ArrayList is definitely the better option.





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? Yes. From below you can see that my remove method meets the O(N) requirement. It's O(N) because like get(i), remove has to traverse through the list in order to index it. ArrayList is also O(N), but because after it removes an item, it has to shift the items to the left.





2. In general, how does DoublyLinkedList compare to ArrayList, both in functionality and performance? Please refer to Java's ArrayList documentation. It's really a trade-off between the two. If the user wants a list that they can index into immediately and retrieve values, Java's ArrayList is the perfect option. Now if the user wants to add an element to the beginning of a list very quickly, due the DoublyLinkedList's awesome referencing nature, the ArrayList would fall short of the DoublyLinkedList. Another attribute of Java's ArrayList is that it has to resize itself if it gets too big. Although that probably wouldn't affect the overall asymptotic behavior of the program, it is another sizeable step that ArrayList has to perform that DoublyLinkedList doesn't have to worry about.

3. In general, how does DoublyLinkedList compare to Java's LinkedList, both in functionality and performance? Please refer to Java's documentation. DLL allows you to go in two directions instead of just one. Java's LL takes up a little less memory as it only has one pointer. In the DDL, if we know the element we need to access near the end of the list, we can start from the tail and call .prev. This can increase get() times substantially. The DDL also has a removeFirst and removeLast, as well as addFirst and AddLast, since it has constant and exact knowledge of the first and last node, it can perform any of these functions in O(c). In order to add or remove anything from the end of the list, Java's LL would have to traverse the list first and that would make its runtime O(N).

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? Performing binary search on a linked list would be a nightmare. Trying to divide the array in half and traversing back and forth would take a lot of time and would ruin the O(log(N)) nature of binary search. If you needed to add something to the front of the list, the DDL would work amazingly. But since BinarySearchSet enters the set sorted, that would likely never happen. Remove would perform about the same as an ArrayList because they both perform in O(N). It

wouldn't perform as well on contains. Contains is a O(log(N)) method. The items would have to be sorted but you'd still have to traverse the list to find it. The ArrayList is the clear winner.

5. How many hours did you spend on this assignment? I'd say I spent around 18 hours.