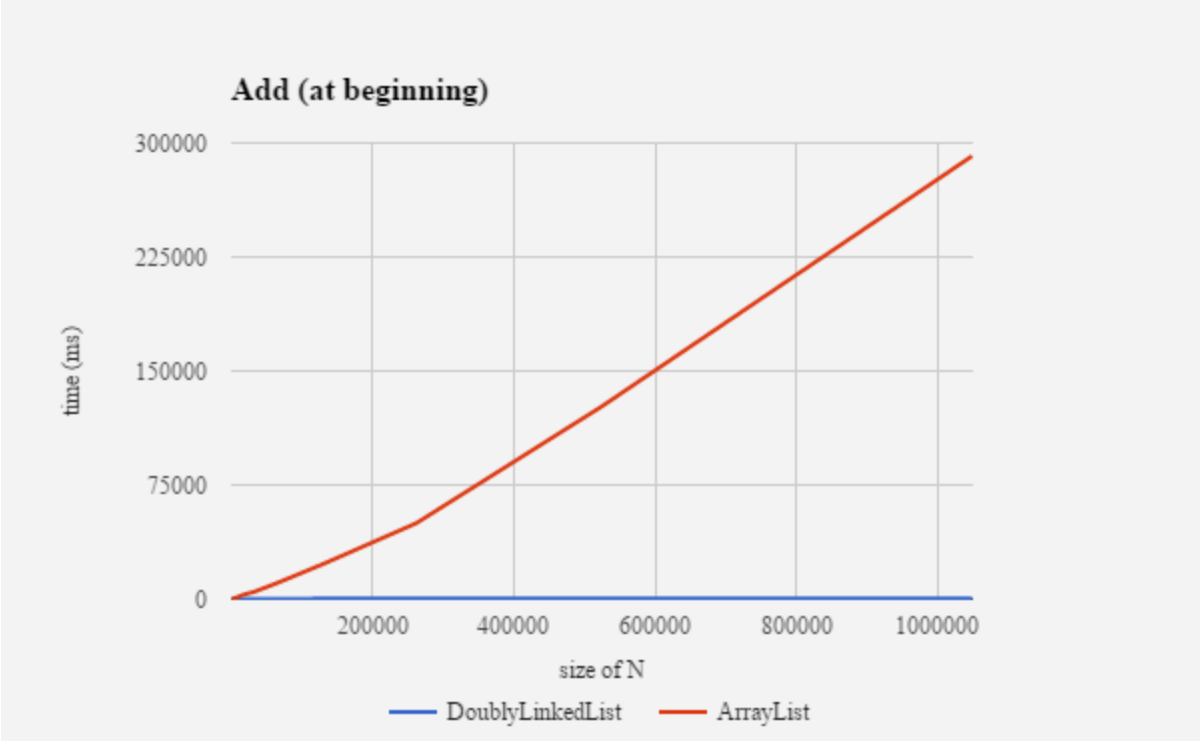
Rebekah Peterson
Assignment 6: Linked Lists
CS 2420

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.

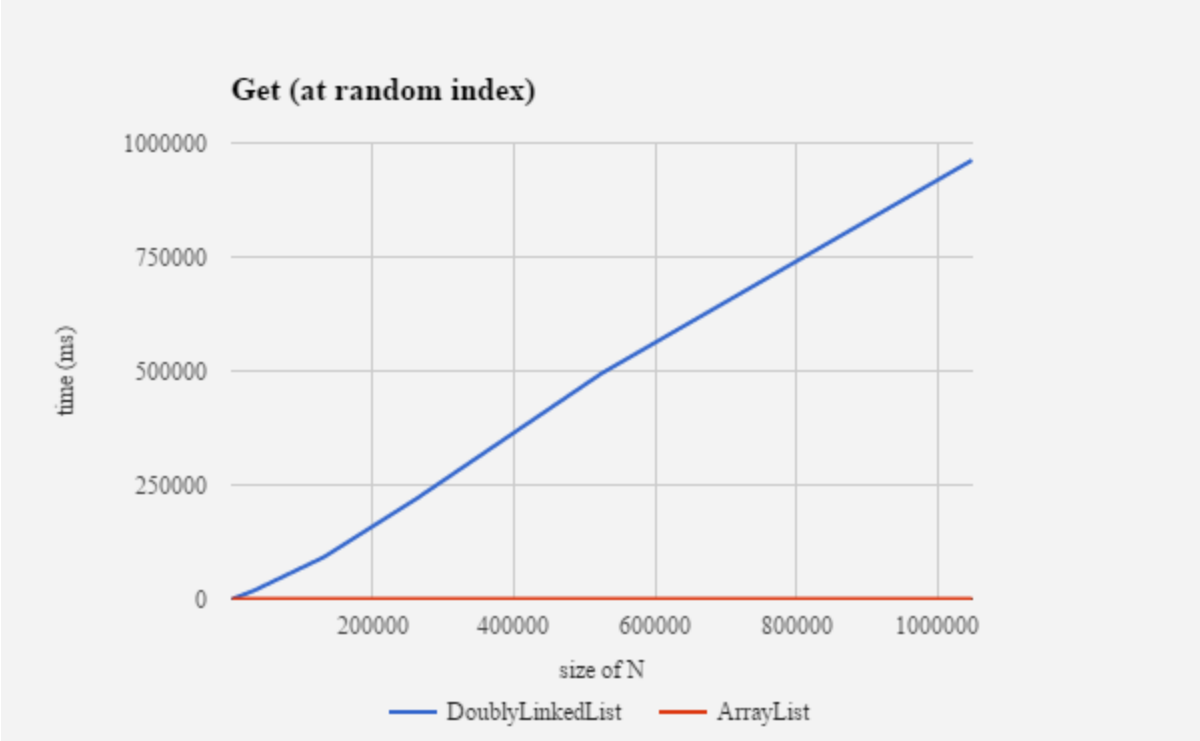
Is the running time of the addFirst method O(c) as expected? How does the running time of addFirst(item) for DoublyLinkedList compare to add(0, item) from ArrayList?

As seen in the graph below, the addFirst method of DoublyLinkedList performs with O(c) complexity. This makes sense since the implementation involves steps that don't involve N, the size of the list. ArrayList shows O(N) complexity because of the shifting of the backing array that has to happen.



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?

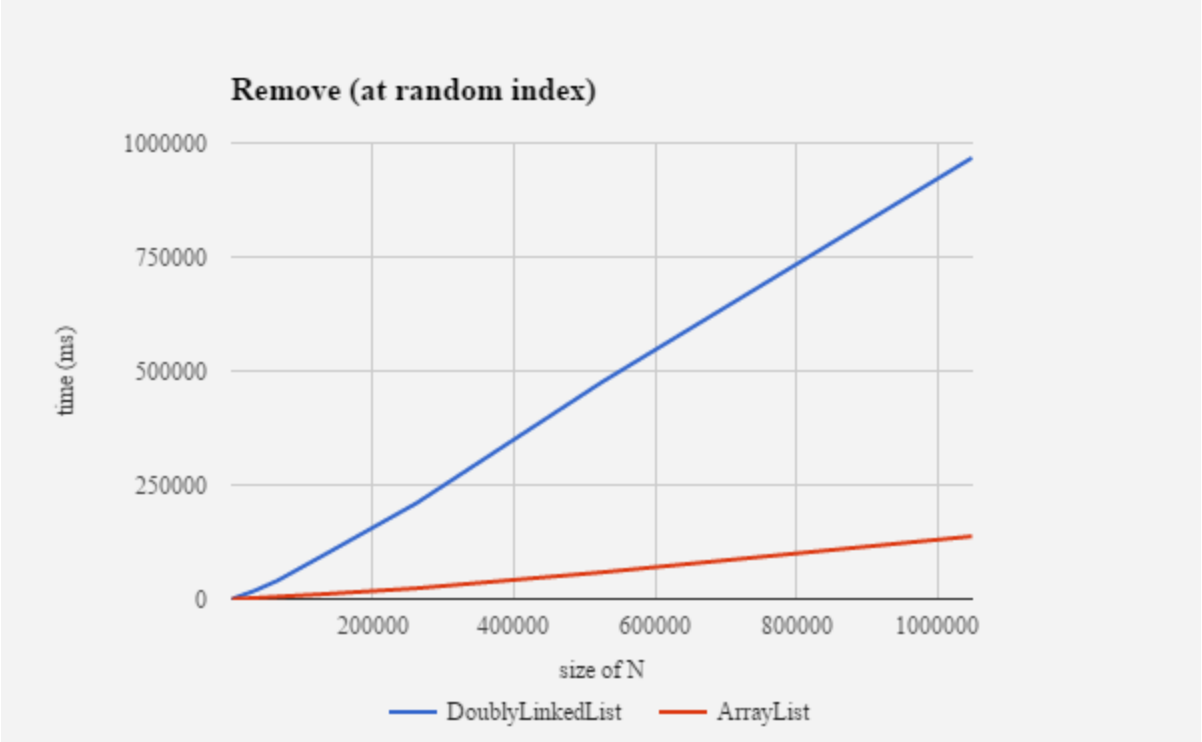
The graph below illustrates that the get method of DoublyLinkedList performs with O(N) complexity. This is because “getting” involves stepping through the list. ArrayList outperforms DoublyLinkedList for getting at a random index, because ArrayList’s get method doesn’t require stepping through the list, it can just retrieve the information stored at that index.



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?

As seen in the graph below, the remove method of DoublyLinkedList performs with O(N) complexity. This makes sense since the implementation requires stepping through the list of size N. ArrayList performs better when removing random items. Although it can go straight to the item, it must shift items still

which results in O(N) complexity.



2. In general, how does DoublyLinkedList compare to ArrayList, both in functionality and performance? Please refer to Java's ArrayList documentation.

DoublyLinkedList stores data with Nodes, which contains the value and links to the Node before and after. ArrayList stores data through a backed array, then as the list outgrows its array, it resizes the array. DoublyLinkedList has better complexity for methods that deal with the beginning and end of the list, while ArrayList requires shifting of items in such instances. ArrayList performs better when dealing with indexes in the entire list, because it can go straight there. DoublyLinkedList acquires O(N) complexity from having to traverse the list to accomplish things, while ArrayList acquires O(N) complexity from having to shift items when removing or adding.

3. In general, how does DoublyLinkedList compare to Java's LinkedList, both in functionality and performance? Please refer to Java's LinkedList documentation.

Java's LinkedList is a doubly linked list. It functions and performs much the same as our DoublyLinkedList. Java's LinkedList has implemented many more methods, such as descendingIterator(). It uses Nodes to store data and links, and

has a head and tail, making it possible to start at the beginning or the end of the list.

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?

Using an ArrayList as the backing data structure, we saw O(N) complexity for add, O(N) for remove, and O(log N) for contains. Because having the data sorted would not improve the complexity of a linked list, the complexity would be O(N) for add, O(N) for remove, and O(N) for contains. To implement contains we would have to use sequential search. So using an ArrayList would be better for our BinarySearchSet still.

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

10 - 12 hours