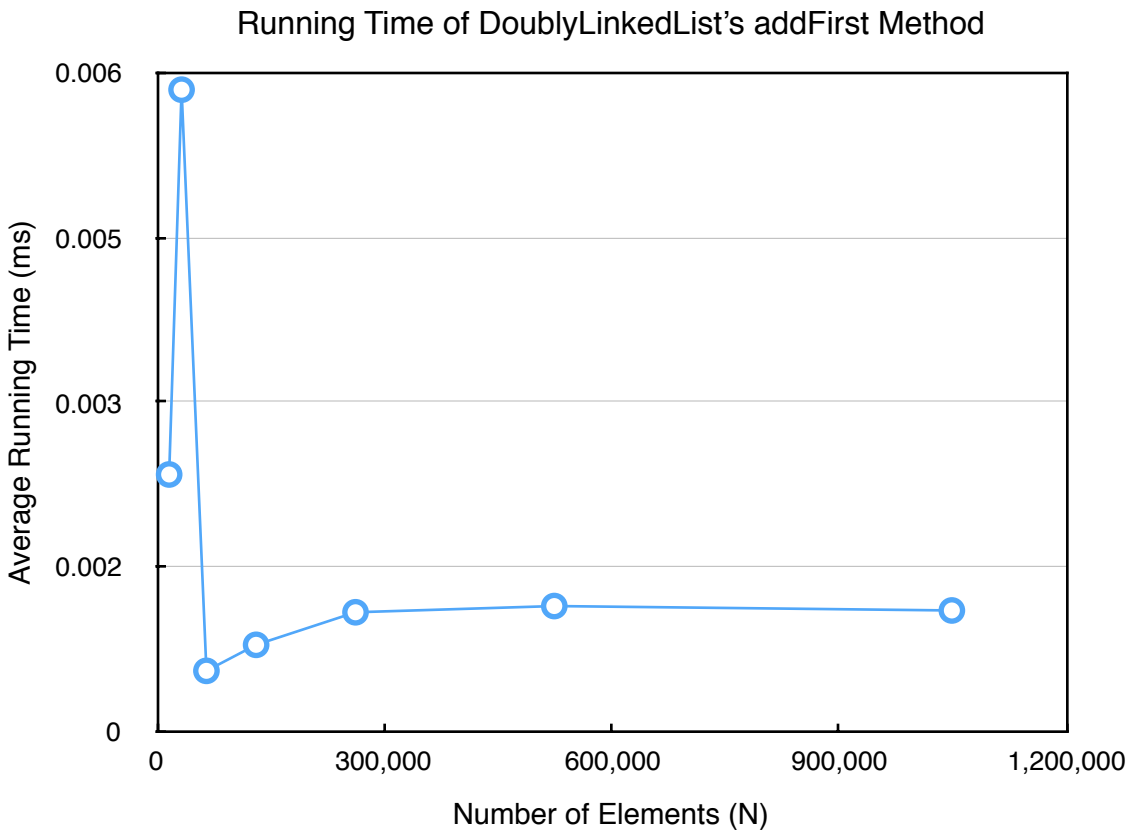
Analysis Document for Assignment 6

Philippe David (u0989696)

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.

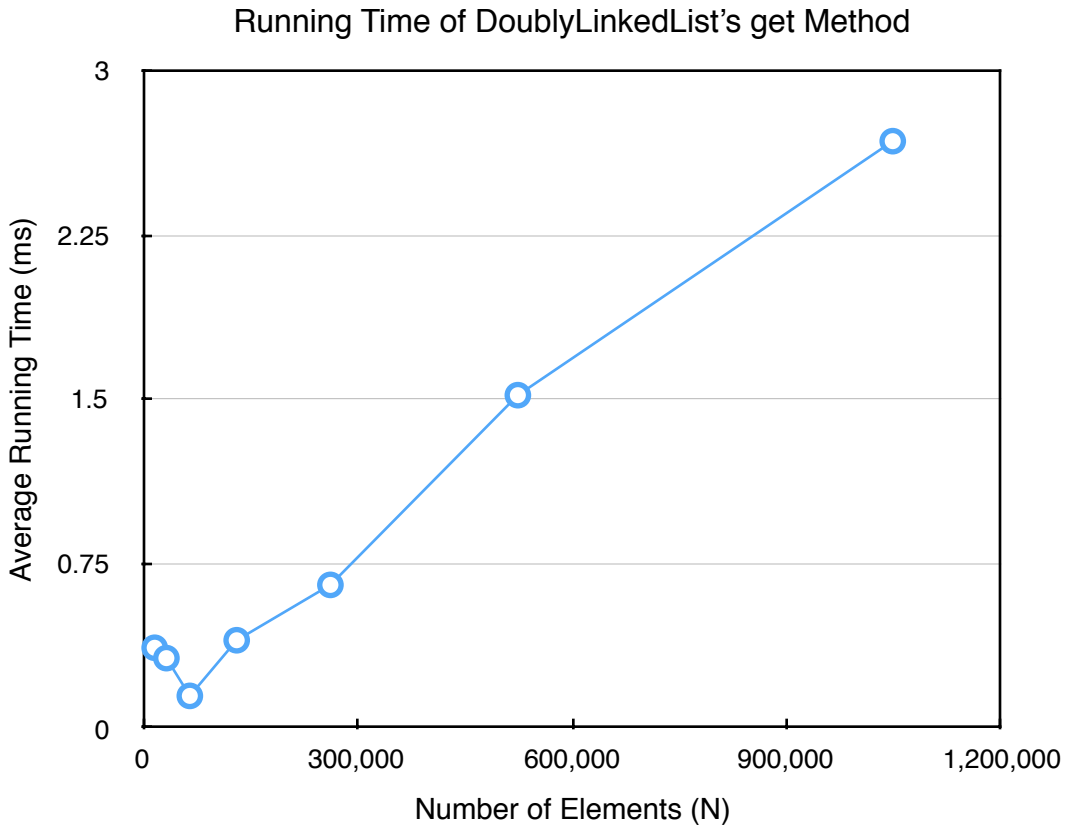
1.1 Is the running time of the addFirst method O(c) as expected? How does the running time of addFirst(item) for DoublyLinkedList compare to add(0, item) for ArrayList?



The run-time complexity for the addFirst method is O(c) (constant time) in this implementation of a doubly linked list as elements are not adjacent in memory and the

operation only requires re-assigning a few references. An ArrayList has run-time complexity of O(N) for the same operation because of the additional requirement of shifting all the elements in the list one index to the right to fit the new element.

1.2 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?

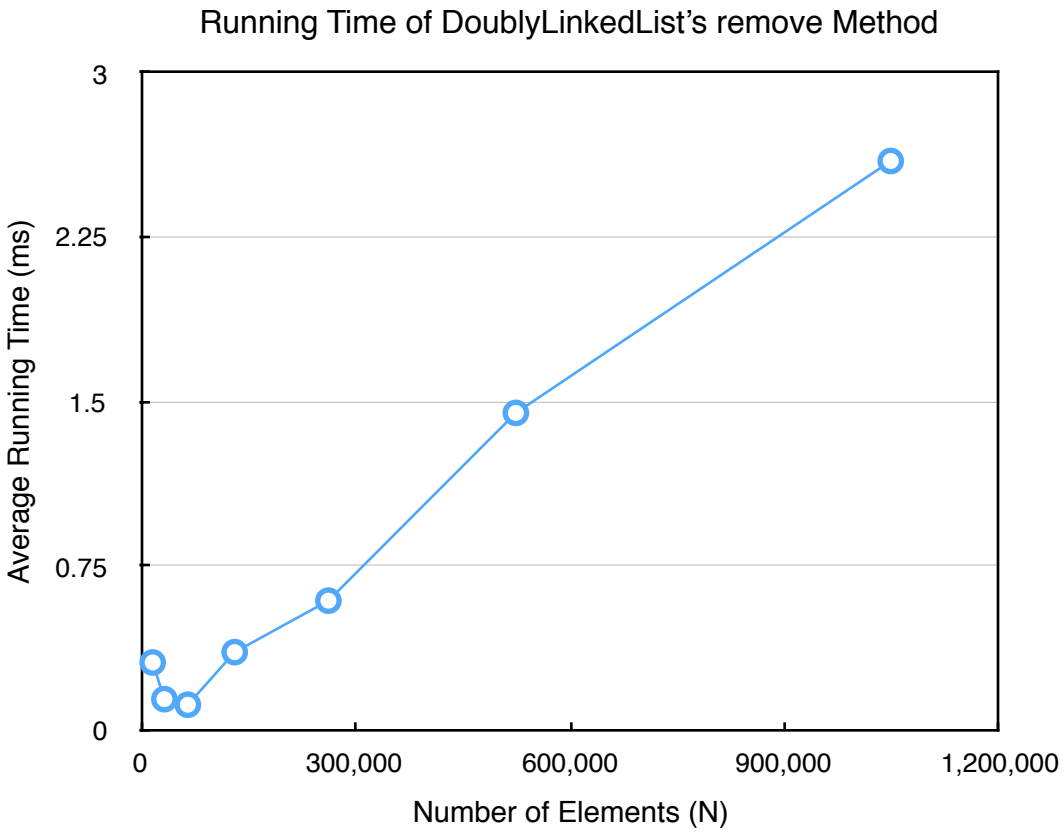


The average and worst case run-time complexity for the get(i) method is O(N) (linear time) in this implementation of the DoublyLinkedList. This is because the linked list requires that the program transverse the list to find the element at the specified position. The best case, with the element being at the beginning or end of the list, has constant time run-time complexity.

An ArrayList has run-time complexity of O(c) for this method because Arrays are laid sequentially in memory. This means, if it is an array of integers that uses 4 bytes each, and starts at memory address 1000, the next element will be at 1004, and the next at 1008, and so forth. Thus, in order to return the element at position 20 in an

array, the code in get(20) will have to compute: 1000 + 20 * 4 = 1080 (a constant-time operation) to have the exact memory address of the element.

1.3 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?



The average and worst case run-time complexity for the remove(i) method is O(N) (linear time) in this implementation of a doubly linked list. This is because the linked list data structure requires that the program transverse the list to find the correct index of the element to remove. The best case, for an element at the beginning or end at the list, would have O(c) run-time complexity.

An ArrayList has run-time complexity of O(N) for this method because of the additional requirement of shifting all the elements in the list one index to the left to fill

the gap left by the removed element. Finding the correct index is a constant time operation because of the nature of the data-structure.

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

DoublyLinkedList and ArrayList are two different implementations of the List interface. DoublyLinkedList implements it with a doubly-linked list and ArrayList implements it with a dynamically re-sizing array.

DoublyLinkedList<E> allows for constant-time insertions or removals using iterators, but only sequential access of elements. In other words, it is possible to walk the list forwards or backwards, but finding a position in the list takes time proportional to the size of the list. The elements in a LinkedList have references to the next and previous elements allowing the insertion and removal of elements from list to only require a few assignment statements.

ArrayList<E>, on the other hand, allow fast random read access, so it can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap. Additionally, if more elements are added to the ArrayList than the capacity of the underlying array, a new array is allocated, and the old array is copied to the new one, Adding to an ArrayList is O(N) operation in the worst case.

Memory usage is also different. Each element of a LinkedList has more overhead since references to the next and previous elements are also stored. ArrayLists don't have this overhead.

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

Java's LinkedList is a doubly-linked list implementation of the List and Deque interfaces. It implements all optional list operations, and permits all elements (including null). All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Java implantation is smarter than DoublyLinkedList for a number of reasons. When the list needs to find an element at a specified index it will, in Java's implantation, will transverse from the side that is closer to the index. This gives it approximately twice the average performance of DoublyLinkedList for the get(int index), add(int index, E element), and remove(int index) operations.

Java iterator implementation is also smarter and has more features and than my own. The iterators returned by Java's LinkedList has fail-fast protection: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. This feature is not implemented in DoublyLinkedList.

Many additional, useful methods also are implemented in java version such as the addAll(Collection c) method that adds all the elements of the specified collection c to 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?

An ArrayList implements the List Abstract Data Type using an array as its underlying implementation which is essentially what we implemented in assignment three. Access speed for ArrayList is virtually identical to an array, with the additional advantages of being able to add elements to the list without worrying about the size. Arrays are static in size and are fixed length data structures, whereas ArrayLists are dynamic in size. In fact, the ArrayList was specifically designed to replace the low-level array construct in most contexts. Essentially, using an ArrayList would not change the BinarySearchSet class at all.

Using a LinkedList however would change the BinarySearchSet a lot. The contains method would suffer a relatively large change in run-time performance because of the fact that is is not possible to implement a binary search algorithm on basic doubly linked-list. The Binary search algorithm is based on the logic of reducing

the input size by half in every step until the search succeeds or input gets exhausted. The important point here is "the step to reduce input size should take constant time". In case of an array, it's always a simple comparison based on array indexes that takes O(c) time. LinkedLists however don't have indices to access items. To perform any operation on a list item, the method would first have to reach it by traversing all items before it. An implementation of binary search on a doubly-linked list would work by computing the indices to look up on each iteration (just like in the array case), then access each one by starting at the front of the list and scanning forward the appropriate number of steps. This is indeed very slow and would have O(n*log(n)) run-time complexity.

For the add() and remove() methods, run-time complexity is equivalent for both data structures. In the case of an ArrayList adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap. This gives average complexity of O(N) not counting the use of binary search to check if the item already exists.

For the LinkedList, calling either of these methods would require traversing the list to find if the element is located in the list. This is an O(N) operation. Of course, once the element or index has been found, inserting or a new element or the specific element is an O(c) operation.

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

I spent about 12 hours in total working on this assignment.