Homework 2 Part 2e: Group Writeup

Sanjeev Janarthanan

Dealing with Null and Non-null Entries

The main difference between null and non-null entries is dealing with equivalence. To test if two non-null entries are the same, you have to use the .equals() method. To test if two null entries are the same you have to use the == operator. As such, any time I needed to test the equivalence of two objects, I would have a separate case for if the object was null and for if it wasn't null. Having two separate cases for null and non-null entries has a negligible effect on efficiency for large lists, which is why I implemented two cases.

Experiments (Graphs for each experiment included in Figures section)

Experiment 1

This experiment tested the difference in efficiency in removing items from the front versus back of an ArrayDictionary.

I predicted that for a large ArrayDictionary, removing from the back would have a significantly larger runtime than from removing from the front. This is true because unlike a DoubleLinkedList, an ArrayDictionary has no pointer to the last index. The ArrayDictionary must traverse to the last item in its list every time it needs to remove something from the back of its list. This O(n) runtime is far costlier than the runtime needed to remove something from the front of the ArrayDictionary.

The results agree with my hypothesis. I think that my reasoning for my hypothesis was correct. Having to traverse to the end of the list to remove items from the back ended up being costly in terms of runtime.

Experiment 2

This experiment tested sequentially retrieving all the items in a list using a DoubleLinkedList, an iterator, and a for each loop

I predicted that for a large list, using the get() method from the DoubleLinkedList object would be far more costly than using the iterator or the for each loop. This is because both other approaches store your place in the list. This means you don't have to traverse to the current index in your list each time you need to retrieve an index. This constant transversal will significantly add to runtime.

The results agree with my hypothesis. My reasoning held up under testing. Not storing your place ends in the list ends up being costly for runtime.

Experiment 3

This experiment tested the get() from the DoubleLinkedList object on every location in a list.

I predicted that locations near the middle of the list would yield the highest runtime when using get(). This is because with a pointer at the front and back of the list, the middle of the list is further away from these pointers. Past that I didn't think there would be any complications to the result of this test.

For the most part, the results agree with my hypothesis. The general trend of the experiment is for runtime to increase as you approach the middle of the list. However, around a list of 16,000 elements and 17,600 elements there are two notable spikes. I can't account for these spikes in runtime.

Experiment 4
This experiment tested the memory usage of initializing an IList with the DoubleLinkedList and the ArrayDictionary class.

For the most part I figured that initializing either of these classes would yield similar memory usage. I predicted the memory usage for the ArrayDictionary initialization to be slightly bigger because it must create a new array for larger lists. The creation of these new array would add some memory usage to the initialization.

The results agree with my hypothesis. The initialization of ArrayDictionary is only slightly more expensive than DoubleLinkedList.

Figures

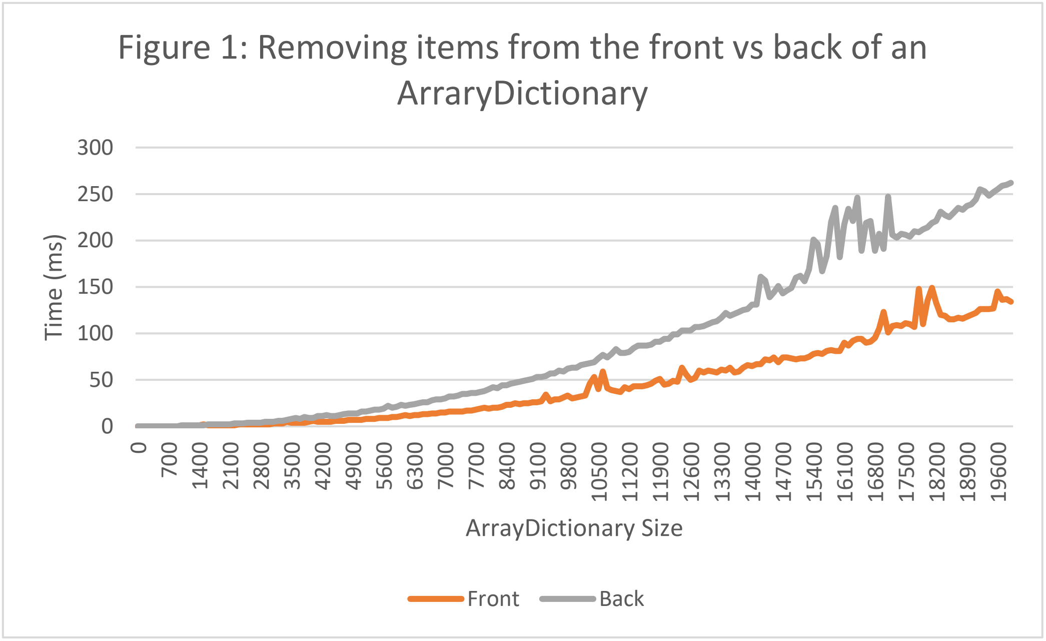


Figure 2: Sequentially retrieving items in a list

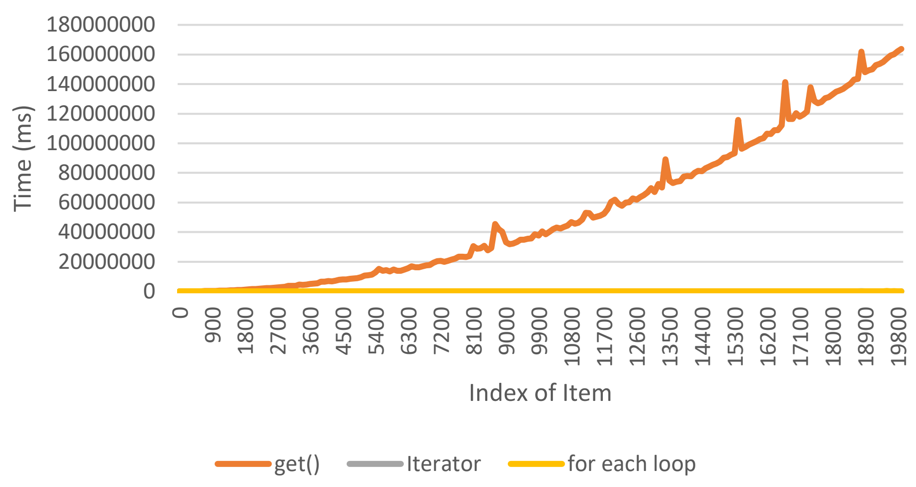


Figure 3: get() on different locations in a DoubleLinkedList

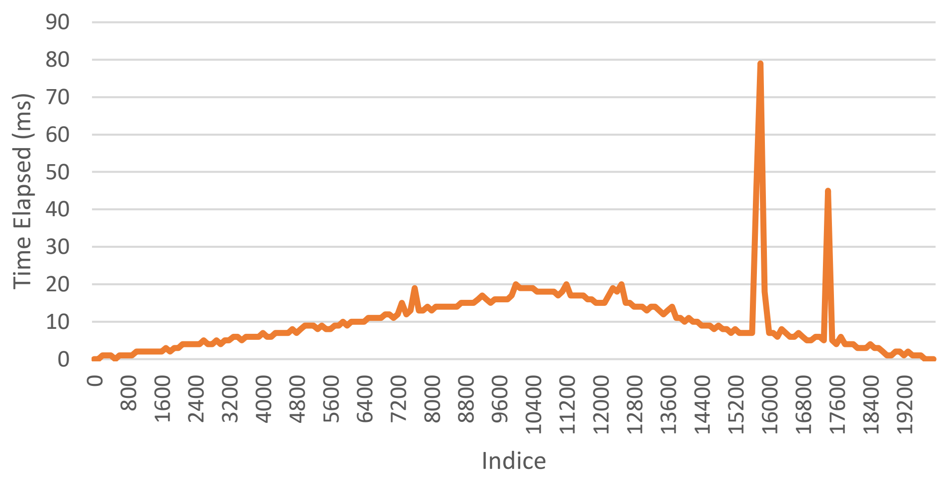
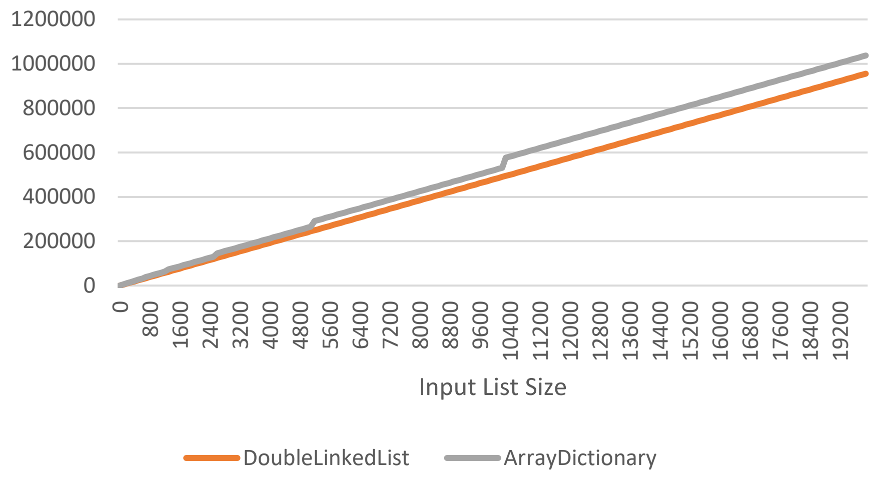


Figure 4: Initializing an IList with Different Classes


Memory Used
Input List Size
DoubleLinkedList
ArrayDictionary