Jonathan Barton
Mitchell Orsucci
CSE 373 Project 1
Winter 2018

1. Handling Null and non-Null entries

In DoubleLinkedList.java, for the 'get()' and 'indexOf()' methods, we may receive an input that is null or not. Because we are comparing against objects in the DoubleLinkedList, we first started by using the 'equals()' method contained in the Object class. However, when using this method on a null object, Java throws a Null Pointer Exception. Thus, to adequately check against any null items in the double linked list, we can use the equals operator: '=='. So, to fully implement this in Double Linked List, we first check whether the item we're looking for in the DoubleLinkedList is null or not. If it is null, we use '==' to compare it against items already in the list. However, if the item is not null, we use the 'equals()' method to check against other items in the list.

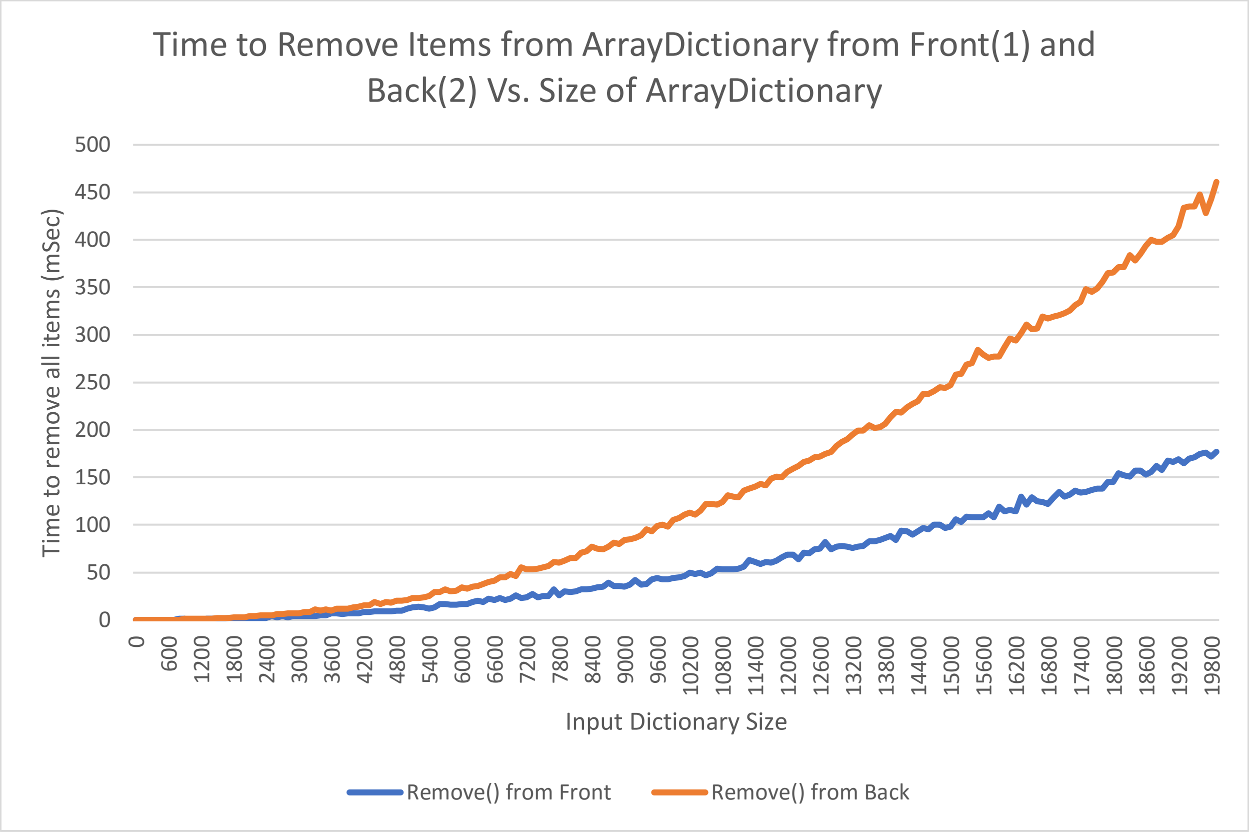
For ArrayDictionary we handled the "NULL" case in a very simple way. Every time we need to find a key within the Array, we must iterate over it and make a comparison to check if the stored key matches the target key. A given search must compare two unknown object types for equality using Java's special object equality testing function, "obj1.equals(obj2)", however this does not work with the NULL edgecase, so to get around this we simply use the standard "==" comparison, to check first if we have matching NULL keys.

2. Experiment Data

a. Experiment 1

i. Test 1 tests the efficiency of the remove() method when removing from the front of the dictionary. Test 2 tests the efficiency of the remove() method when removing from the back of the dictionary. Both of these tests begin with a test on an ArrayDictionary of size 0 and increase the size of the ArrayDictionary by 100 each test until the maximum ArrayDictionary size is reached.

ii. We expect the outcome of the experiment to be that removing all elements from the back of the list will take generally more time than removing elements from the front of the list (especially as the array gets very large). The reasoning for this is that based on how remove() is implemented with an Array based dictionary, searching for a key will involve starting at the beginning of the array and iterating forward until the target key is found and then you have to make object comparisons on keys all the way through the array. Removing from the front involves making a single object comparison at index zero, and then having to shift the entire array to the left. However, removing from the end involves interating over the entire array and making n object comparisons. Given these two behavioral patterns, we assume removing from the end takes longer because making n object comparisons is slower than making n array shifts.

iii. 

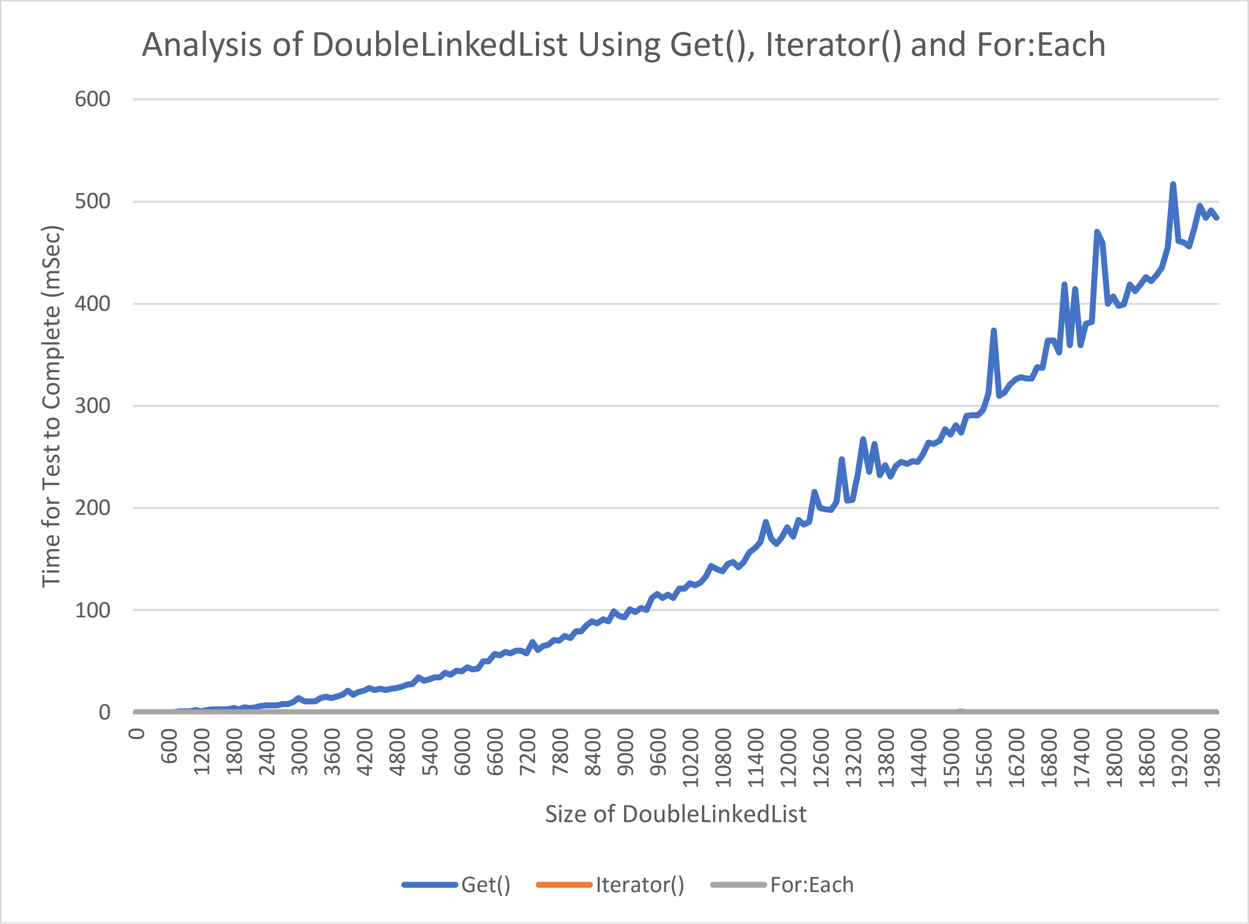
iv. As per our hypothesis, the removal from the back took a fair amount longer than removing from the front, we still assume this is because time to do n object comparisons on remove() from back takes much longer than n array shifts on remove() from front.

b. Experiment 2

i. Test 1 tests the efficiency of the DoubleLinkedList when using the get() method from the front of the list. Test 2 tests the efficiency of the iterator implementation of the DoubleLinkedList when using the hasNext() and next() methods. Test 3 tests the same thing as test 2 but without using the iterator explicitly. Test 3 utilizes the for:each syntax contained in Java. Each test runs 5 times on different list sizes, starting at 0 and increase to 20000 by steps of 100. The final values are the averages of the five runs.

ii. As the iterator should run in close to constant time, we expect that test 2 should happen relatively quick with respect to test 1. Also, we expect that the results for test 3 should be identical to test 2 as they are implemented using the same underlying code. We expect to see that get() should increase in time as the size of the list increases as get() does not run in constant time. As the size of the list in test 1 increases for each run, we expect to see output results that are relatively linear, with some quadratic elements.

iii.



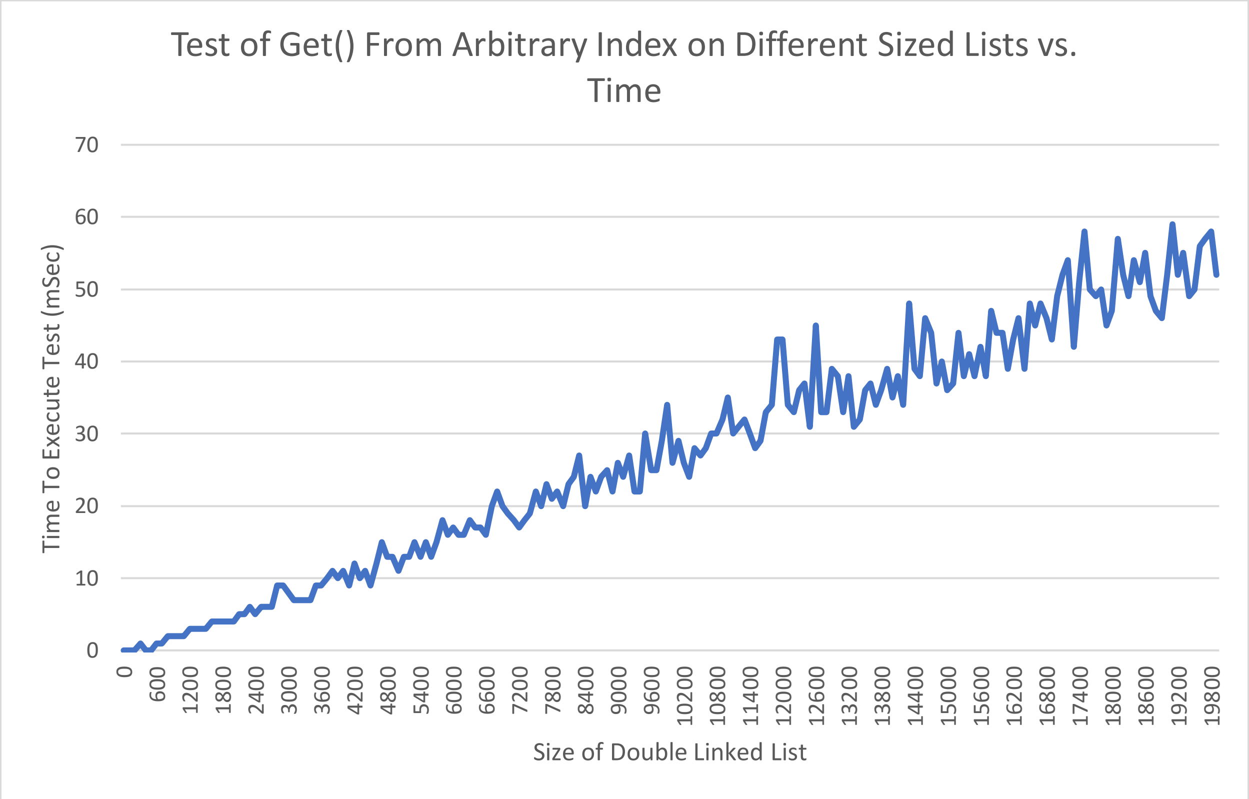
iv. We see from the output plot of the tests that our assumptions were pretty much correct. Tests 2 and 3 took the same amount of time and do not even appear on the graph with respect to test 1. Moreover, we see that the time it takes to execute test 1 does indeed have a mostly linear relationship with time when accounting for the size of the DoubleLinkedList. However, we do see spikes in the testing time at certain sizes of the DoubleLinkedList, which leads us to believe that the DoubleLinkedList is optimized for certain sizes. There does seem to be some quadratic elements to the relationship between time and DoubleLinkedList size. This is because get() runs in constant time from the front or the back of the list, but it runs in linear time when the item is in the middle of the list. Since there is already a linear element to the test, the linear element of the test and the linear behavior of get() when getting from the middle of the list combine to show some quadratic behavior.

c. **Experiment 3**

i. This single test tests the efficiency from calling get() multiple times on a DoubleLinkedList. However, in difference from Experiment 2, this get() method always pulls an item from a specified index, rather than from the beginning or back of the list. The test then runs multiple times with each run getting from a successive single index.

ii. We expect that the amount of time to get() an item from the list will increase as a function of the size of the list. However, as we can traverse from the front and back of the list to find an item, we do not expect this to be a quadratic relationship. We expect this to be somewhere between log(n) < our test < linear, meaning that the performance of this test should be faster than linear, but not quite as fast as log(n)

iii.



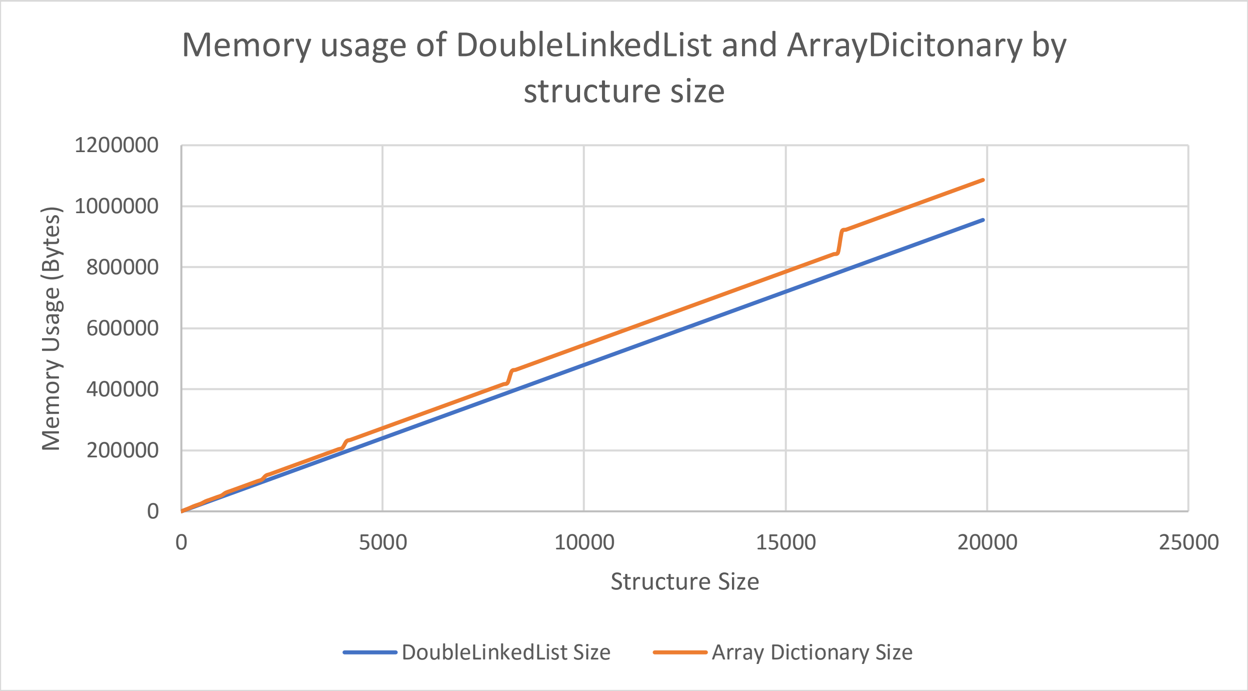
iv. The output plot shows the behavior that we expected and outlined in part ii. We see that the time to execute the test grows relatively linearly as a function of the size of the double linked list. However, the test itself runs in linear time, which leads us to believe that the behavior of get() in our double linked list actually executes in better than linear time.

d. **Experiment** 4

i. This single test measures the memory usage of both the DoubleLinkedList and the ArrayDictionary as the size of each one increases.

ii. We expect that the ArrayDictionary will probably require more memory than the DoubleLinkedList, given that the ArrayDictionary must hold a Key-Value pair per “node”, as compared to a DoubleLinkedList which by design only holds a single data set per node. However we expect that the amount of memory needed will increase roughly linearly as n gets very large.

iii.



iv. As expected, the ArrayDictionary indeed takes up more space, and while both increase mostly linearly, the ArrayDictionary has an interesting stepped linearity, with jumps in memory usage, but maintaining the same rate of increase even after the jump. This is surely due to the doubling in size of the base array each time the dictionary approaches the size limit. The Array dictionary thus makes the most efficient use of memory when it is fully filled but not yet resized.