Muath Alhudhaybi
Grant Neuman
CSE373 Project 1 – Group Write-up
Summer 2018

1. Null and non-Null Question

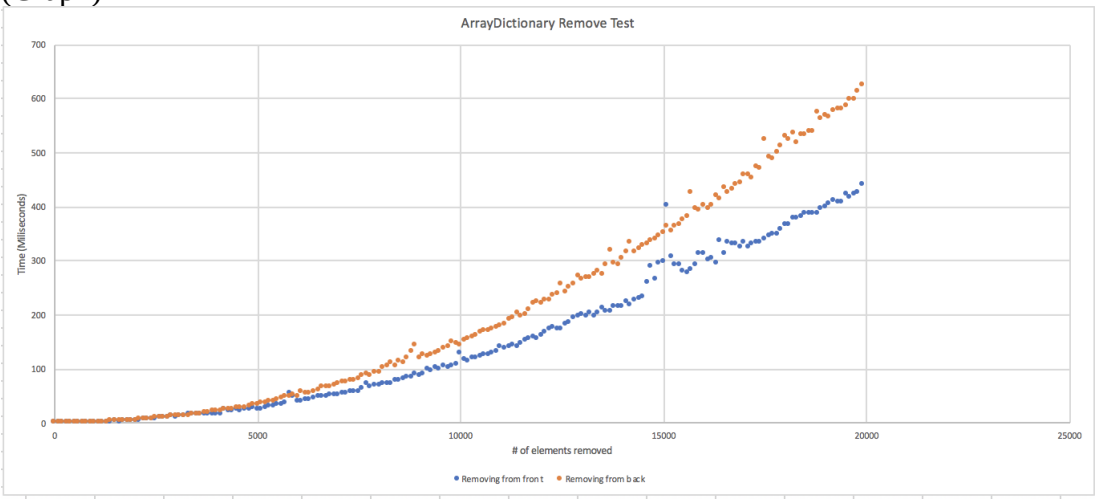
- In DoubleLinkedList, for the “get()” and “indexOf()”, the input could possibly be null or non-null results. Since comparing against values or objects in the DoubleLinkedList, we used the equals operator “==” then we need to see if the value is going to be null or non-null. If the item turned out to be null, we use “==” to compare it to an items that are already in the list. But if the item is turned out to be non-null, we can write “equals()” to see check for other items in the list.

- For ArrayDictionary, we decided to solve the null problem by comparison; when we need to find a keyVal in the array list, we iterate over it and compare and check if the wanted key matches the one we want. In Java, we use the quality testing function “object.equals(object1)”, but when it doesn’t work with a certain null, we use “==” to compare and check if we have a match null keyVal.

2. Experiments:

Experiment 1:

1. This experiment is testing on whether removing elements from the front or back of an ArrayDictionary object is faster / more efficient.
2. The test1 or removing from the front of the list should be faster because we start at the front of the array when removing elements from the ArrayDictionary. This means that we start our iterations of the array from the first index, regardless of where the specified index is in the array. If we want to remove from the end, we have to traverse the entire array and then remove that element. This means that if we are looking to remove an index that is at the very end, we would need to spend more time than if we wanted to remove an index at the front because we would traverse the array fewer times.
3. (Graph)

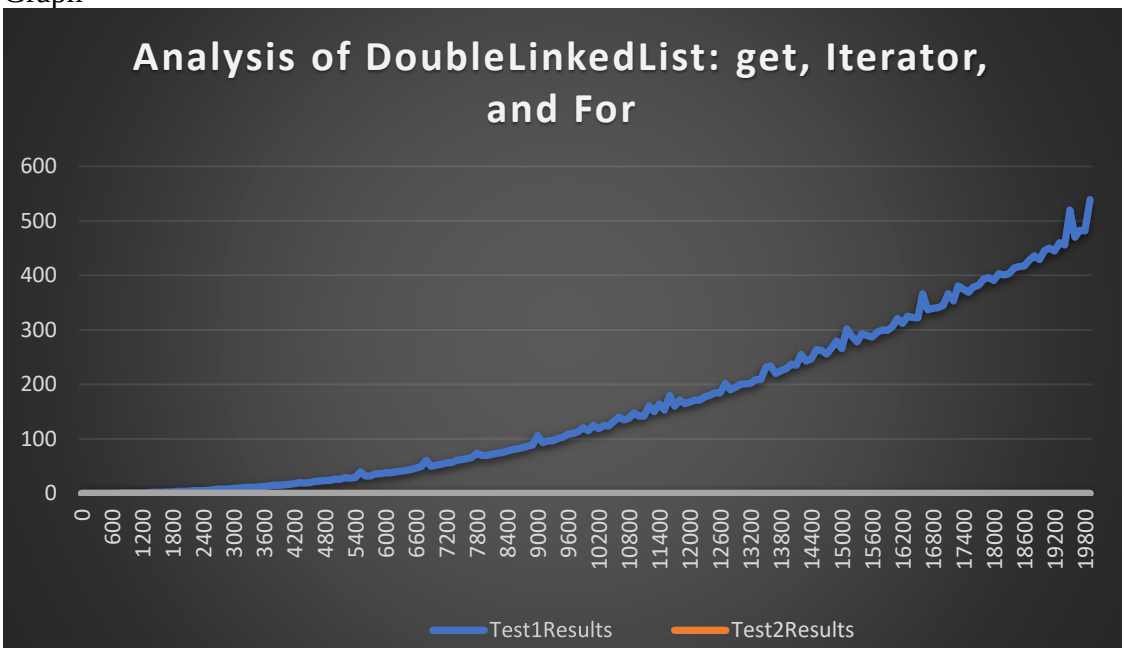


Muath Alhudhaybi
Grant Neuman
CSE373 Project 1 – Group Write-up
Summer 2018

4. My hypothesis was correct, but I did not correctly hypothesize that removing from the back will only become noticeably slower after 5000 elements in the array. Otherwise, my hypothesis was correct which confirms that starting from the beginning of the array regardless of where we are removing an element will greatly affect the efficiency of our data structure when we are dealing with a large number of elements and removing from the back of the list.

Experiment 2:

1. The first test was measuring the efficiency of the DoubleLinkedList while using the "get()" method from the front of the list. The second test was measuring the efficiency of the iterator of the DoubleLinkedList while using "hasNext()" and "next()". The third test was "hasNext()" and "next()", however, uses the "for:each". The values are the averages of the five runs.
2. We predict that the second test should be relatively faster with compared to the first test. We also predict that the results for both second and third test should be very simpler to each other. For the size of the list in the first test, it should increase for each run time. We can predict that the output will be relatively linear.
3. Graph

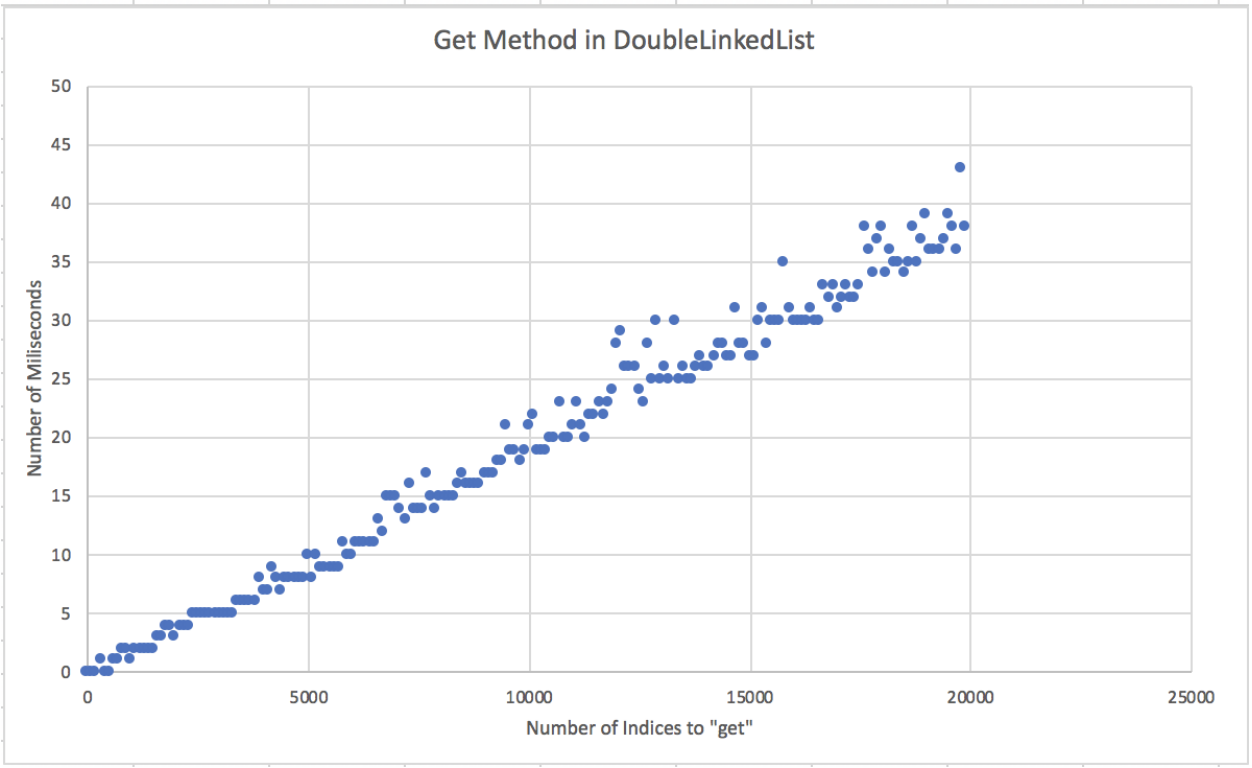


4. Looking at the graph, our prediction was correct. The second & third test were similar in output results. Also, our prediction of having a linear output was accurate, although, we can see that there are spikes in the time at certain sizes of the DoubleLinkedList, which means that it is optimized for certain sizes.

Muath Alhudhaybi
Grant Neuman
CSE373 Project 1 – Group Write-up
Summer 2018

Experiment 3:

1. This experiment is testing the time it takes to use the get method on a DoubleLinkedList by varying the number of times the get method is called.
2. I hypothesize that the test will take the longest when the get function is called the greatest number of times. The test will be the quickest when it is called fewer times. Or in other words, I think that there will be a linear relationship between the number of indices and the time it takes to access them using the get method of a DoubleLinkedList. I think this because the get function in our DoubleLinkedList implementation starts at the front and iterates to the index that it is trying to get. The more indices inputted into the test, the more times the get method is called which would then increase the time of the test.
3. (Graph)



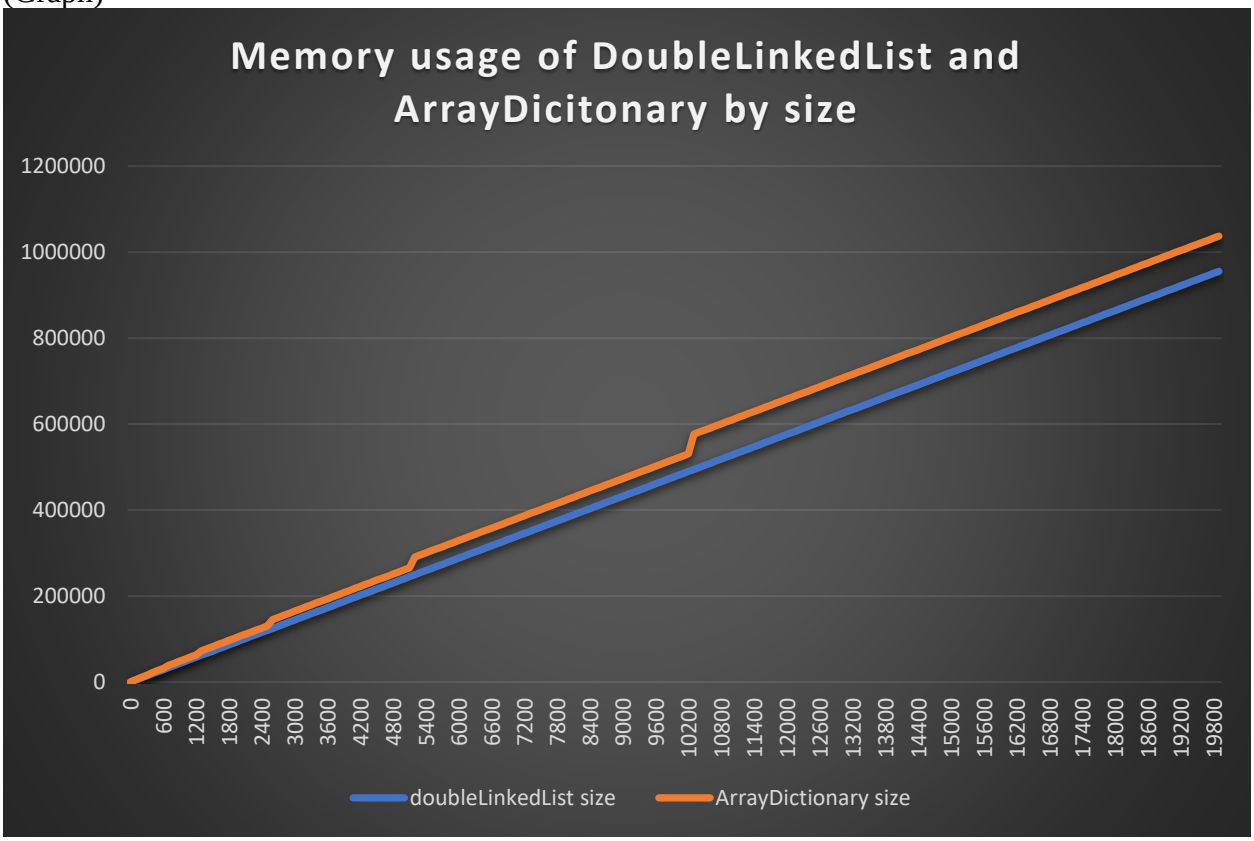
4. My hypothesis was correct. This makes sense because any method that starts at the beginning of a list until it reaches the index in question will increase linearly with the number of elements in must iterate over to reach the specified index.

Experiment 4:

1. The test measures the memory usage of the DoubleLinkedList and the ArrayDictionary as their size increases.
2. We predict that the ArrayDictionary should require more memory than the DoubleLinkedList, given that the Array Dictionary must hold a Key-Value pair per

"node", as compared to a DoubleLinkedList which can only have a single data per node. But we think that the size of memory needed will increase linearly as number gets larger.

3. (Graph)



4. 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.