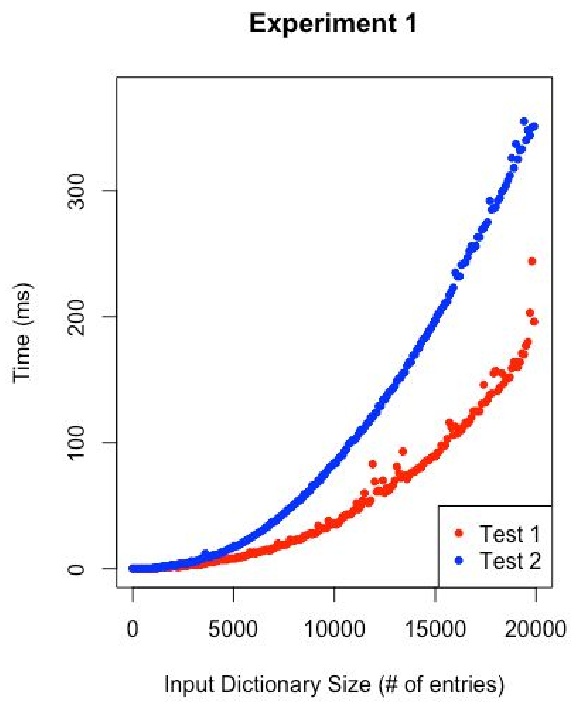
HW2-Bobcat
Tyler Kang 1667888
Andrew Song 1661524

When handling null and non-null entries, we first checked for edge cases. For example, for our DoubleLinkedList, we had to check if the front of the list was null or not to decide how to insert/set a certain node. With the ArrayDictionary's indexOf() helper method, if the the given key was null then we had to use a different comparison operator than if it wasn't. This was due to the fact that the Object.equals() method doesn't accept null inputs. To test if two nulls were equal, we decided to use the equality operator.

<u>Experiment 1</u>
This experiment is testing the efficiency and runtime of our ArrayDictionary's remove() method using differing dictionary sizes and starting indices (test1 starts from key value 0, and test2 starts from dictionarySize - 1).

The outcome of the experiment will be the average times that our remove() method takes to remove every element in a given dictionary for n trials. We predict that the average times will follow a linear relationship with the size of the dictionary, and we also predict that removing from the back will be faster than removing from the back.

Results:

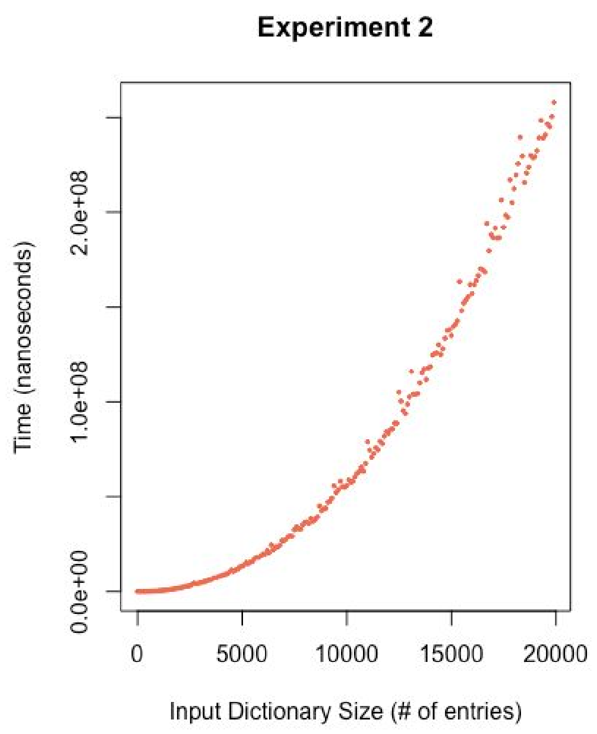


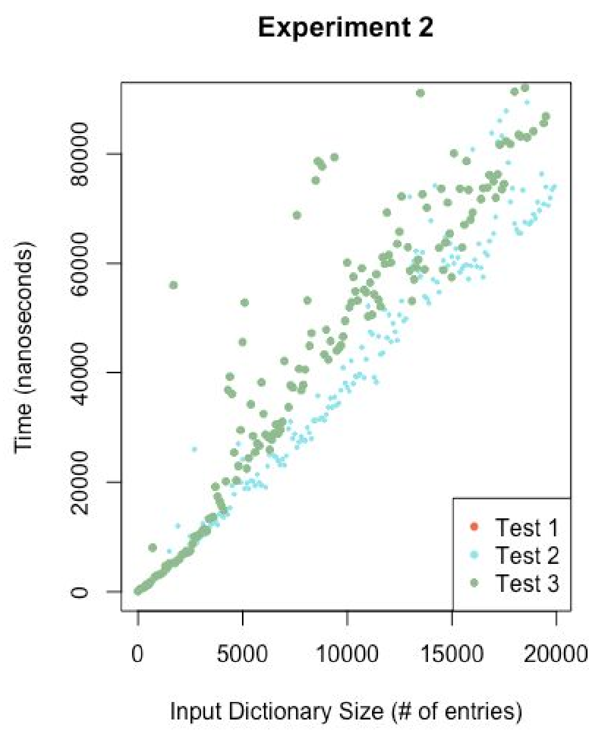
We predicted an O(n) relationship between the dictionary size and runtime, but our results show an O(n^2) relationship. This is because the amount of times we have to iterate through the list increases more and more as the dictionary size increases. (e.g. for dictionarySize = 3, we only have to iterate through the list 6 times. For dictionarySize = 4, we have to iterate through the list 10 times.. For dictionarySize = 5, we have to iterate through the list 15 times) The amount of times we iterate through the list increases at an increasing rate. Test1 is slower than Test2 because Test1 starts at the last possible index, forcing the dictionary to iterate through every element before reaching the end. Test2 is able to start from the front and is therefore quicker.

<u>Experiment 2</u>
This experiment tests the functionality and efficiency of iterating over our ArrayDictionary using different methods of iteration and adds the values of the dictionary to a temporary variable.

The outcome of the experiment will be the average times each iterator takes to go through our dictionary given different list sizes. We predict that, on average, the for loop will take the longest time, the Iterator object will be faster than the for loop, and the for each loop will be faster than than the Iterator. This is expected since the for loop has a runtime of *O(n)* meaning for larger list sizes, the for loop's runtime will follow linearly with the the number of elements in the list.

Results:





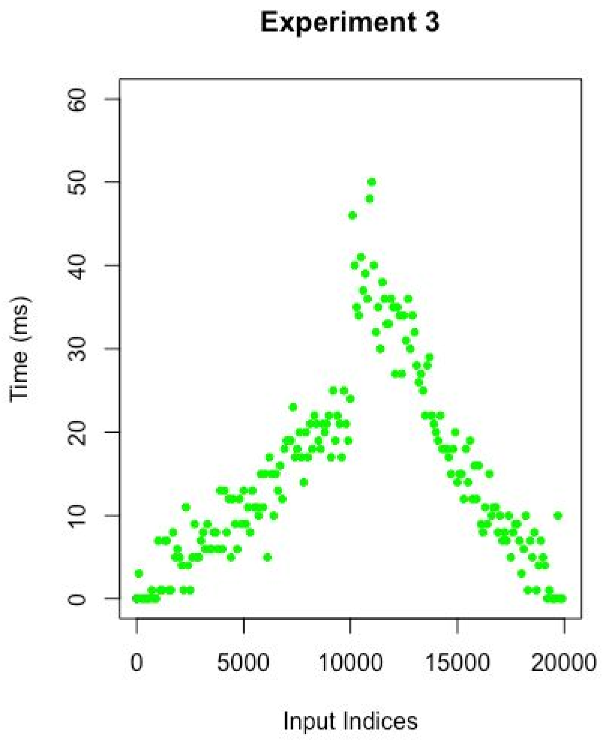
After reviewing the results, our hypothesis was only partially correct. We correctly hypothesized that the for loop iteration (Test 1) would take the longest amount of time, but we incorrectly hypothesized that the for each loop iteration (Test 3) would be faster than the Iterator object (Test 2). It was actually the Iterator object that iterated over the list the fastest, on average. We think that this is the case because the Iterator object does not keep track of indices, only pointers. Hence, "getting" to items in the dictionary, at worst, runs at *O(1)*, or constant time.

<u>Experiment 3</u>

This experiment tests the efficiency of the DoubleLinkedLists's get() method given different list indices. For each trial, a new list is created with 20000 elements and from there, the test then calls get() at a certain index 1000 times, adding the retrieved values to a temporary variable.

The outcome will be the average time that get() takes at a given index. We predict that the time elapsed follows a negative quadratic relationship with the chosen index. At the front of the list, get() is a constant time operation. From an index of 0 and list.size() / 2, the runtime of get() will increase. From an index of list.size() / 2 to list.size(), runtime decreases. At the back of the list, get() is again a constant time operation.

Results:



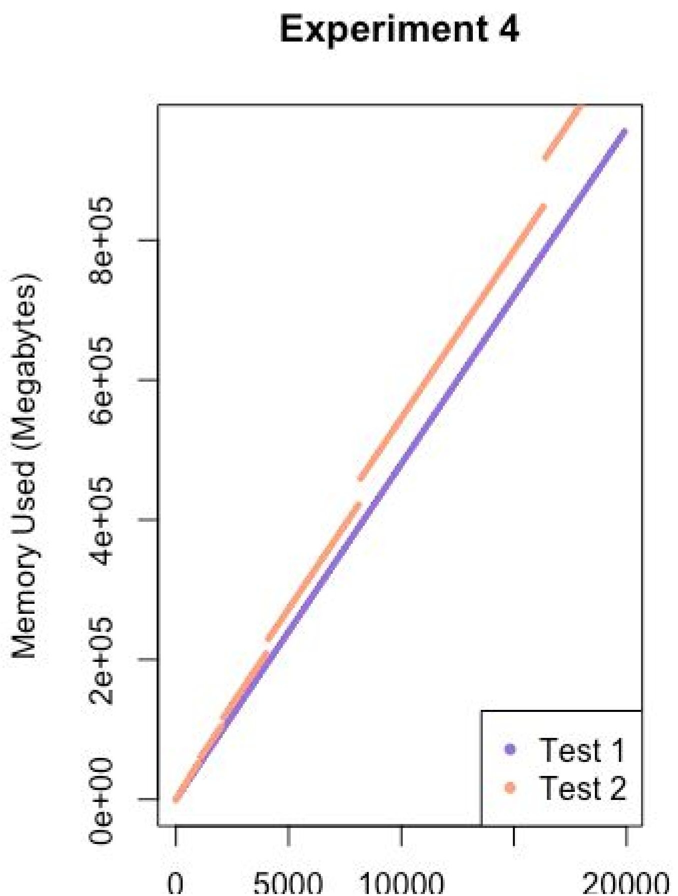
Our results were consistent with our hypothesis. This is because from indices 0 to list.size() / 2, the list must iterate through every element until the chosen index starting from the front of the list. Past an index of list.size() / 2, the list iterates from the back to the chosen index. Because there are internal pointers to the front and back of the list, access to the front and back has minimal runtime. What's surprising is the sharp jump in runtime at index 10000. Our prediction was that the runtime would follow a continuous relationship with input indices. This could be because iterating from the back requires checking if (size - index - 1 > 0), while iterating from the back only requires checking if (index > 0). The extra subtraction operation causes a jump in runtime by iterating from the back. Generally, this test follows a linear relationship from an index of 0 to an index of list.size() / 2, and an inverse linear relationship from an index of list.size() / 2 to list.size().

Experiment 4

This experiment approximates the amount of memory used to create DoubleLinkedLists and ArrayDictionaries of increasing sizes.

The outcome of the experiment will be the amount of memory taken up by both data structures. We believe that the ArrayDictionary will take up more memory than the DoubleLinkedList for all possible sizes of a given list. We think this because for a particularly large list with, say, 5,000 items in it, the ArrayDictionary has to take up a slot of memory for each of those 5,000 items.

Result:



The results of the experiment and our predictions are consistent. Given any list size, the ArrayDictionary takes up more memory than a DoubleLinkedList. This is most likely because memory is being stored as the list is being updated, therefore the dictionary has to constantly open up more memory to store items in order. On the other hand, DoubleLinkedLists are not sorted and therefore less memory is taken up keeping track of where particular data is located within the list.