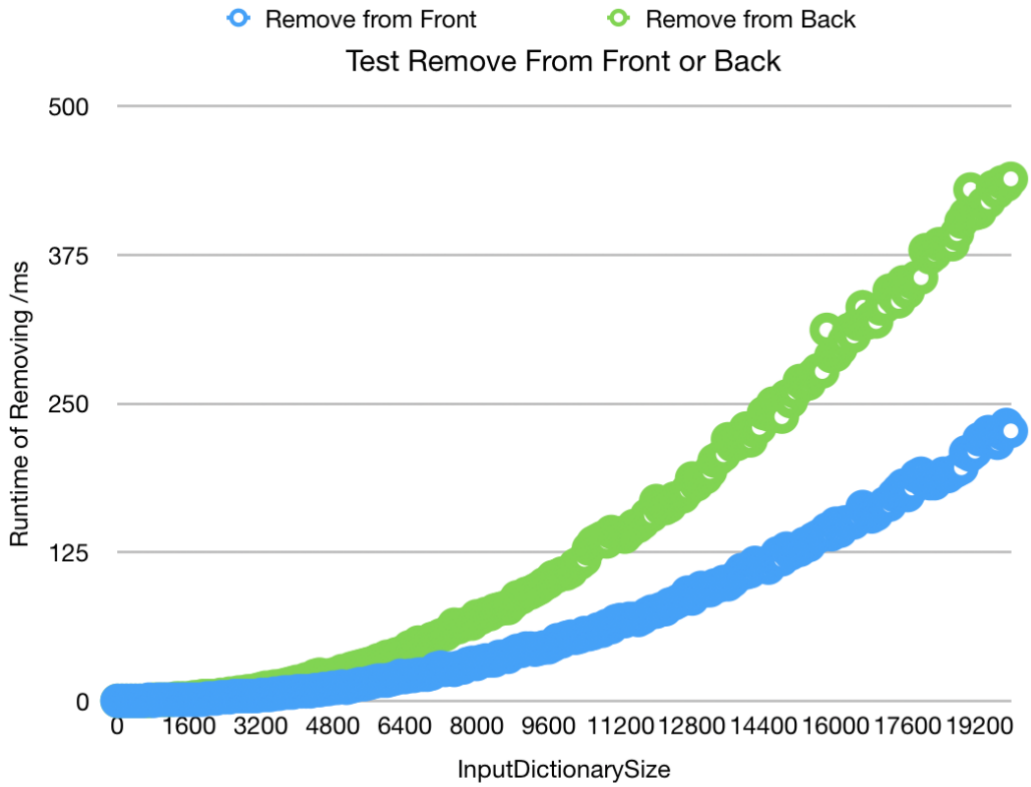
1. In DoubleLinkedList, to handle null case and non-null case, I use if statements to distinguish cases where null is at the front when inserting a new node. In ArrayDictionary, I realize that .equal() cannot be applied to null keys, so I add an if statement that returns true if both keys is null, in prior to applying .equal(). Since null can also be a key, it is unnecessary to check if the key is null before putting it into the ArrayDictionary.

2.

Experiment 1

1. Test remove items from dictionary by removing from the front or removing from the back to test its efficiency represented as the runtime as dictionary size increases. By loading in large data set into the the loop, the test would fail if the data structure doesn't implement a way to look up from the back.
2. According to our ArrayDictionary.remove(K), after checking whether the given key exists, which needs traversing ArrayDictionary, the a key that equals to the given key is then found by travesing the ArrayDictionary again. Since traversing the ArrayDictionary takes a runtime of O(n), which dominates our other statements in the method, the efficiency is largely determined by how large the size is and whether the key is remove from front or back. In test 2, since it's removing elements from back of the ArrayDictionary, it has to go to the end of the ArrayDictionary every time to check equality (runtime n) and find the key to be removed(runtime n). They always remove the last element. So the runtime will be like $n+n-1+n-2+n-3+\ldots n-n = n^2-(n+1)n/2$ and the complexity will be $O(n^2-n^2/2 = n^2/2)$. While in test 1, since it's removing elements from front, the key at the back will be moved to the front every time. So the first element will be removed, then second, then third…..When reached index = half of the dictionary size, the second half of the elements are already moved to the first half of the ArrayDictionary. So the dictionarySize/2 th element is removed, then dictionarySize/2 - 1 th….all the way down to the first element. So the runtime will be $(n/2 +1)n/2$ having complexity of $O(n^2/4)$.
So the runtime for both test should give a plot similar to quadratic equation, but runtime for test 1 is half of that of test 2

3.



4. The scatter plot result is similar to what we expected. Both lines are quadratic and the green line(remove from back) has runtime roughly double the runtime of blue line(remove from front). I think our prediction is correct, the runtime of this experiment is determined by the size of the dictionary. And that removing from front uses half of the runtime than that of removing from back.

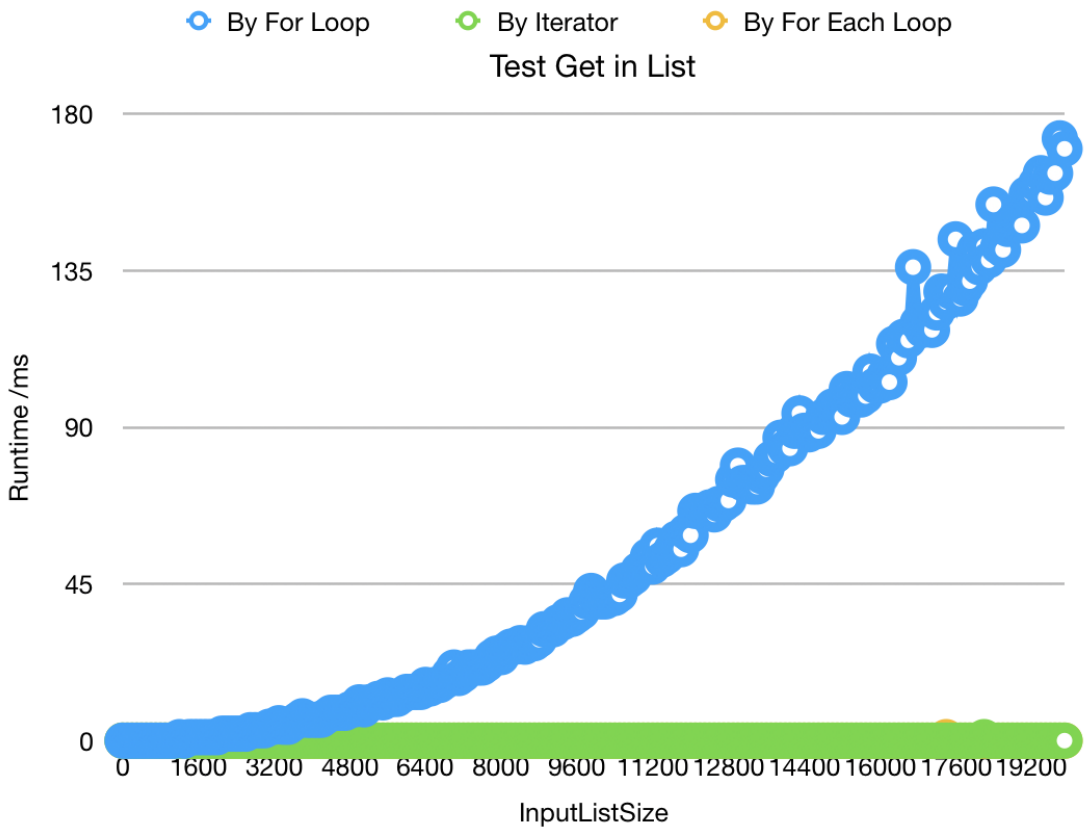
Experiment 2

1. Experiment 2 is trying to add all elements in the double linked list together by using a for loop or by calling an iterator to reach every value in the iterator. And check their runtimes as size of the list increases.
2. From our perspectives, test 1 will take significantly more time than 2 and 3 as the size goes up. This is because test 1 is trying to add elements using for loop that runs n times, and get to the value to be added by using get(i) in every loop. In our method, get(i), we traverse the list from front or back to get the element we want, so the runtime is roughly equal to that of removing from front in experiment 1, which is $O(n^2/2)$, and since it is inside the for loop in test 1, a

loop inside a loop will take significantly more time as size goes up. We think that it is like $O(n^3/2)$ since for loop takes n time to complete. The graph will be like a n^3 graph.

Test 2 and 3 are roughly the same since in test 3 a for each loop is used, which is like calling an iterator. Both of these operation only traverse the list once in the iteration, so the runtime is O(n) and both lines for 2 and 3 looks like a straight line.

3.



4. The plot is similar to our hypothesis since the plot of green line(iterator) and yellow line(for each loop) is the same and is a straight line. Because they are doing the similar operation and the runtime is O(n), which doesn't increase much as n increases. But the plot of blue line(for loop) looks like a n^2 line which confused us a bit. We think that the get(i) method in double linked list is probably less than $O(n^2)$.

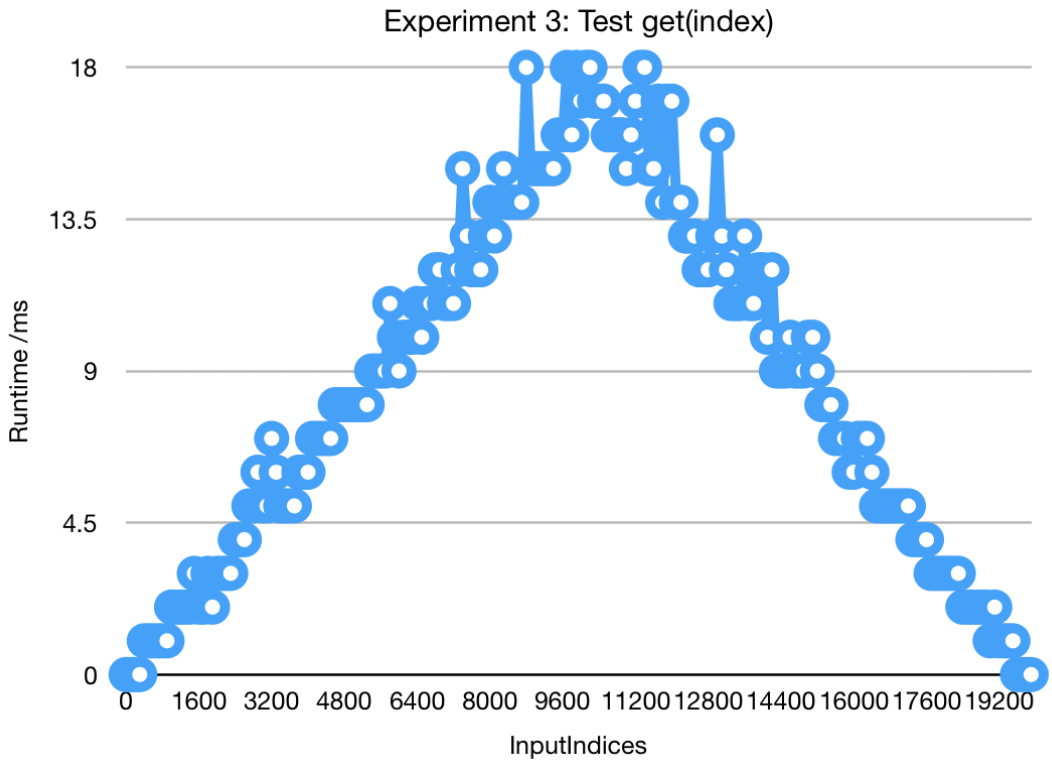
Experiment 3

1. It is trying to get an element using get(index) in double linked list multiple

times and record the runtime as the index increases.

2. We think that since we handle both the cases where index > size/2, which will traverse from back to get to the corresponding position of index, and index < size/2, which will traverse from front, the runtime will be the most when index is near the middle of the list which similar to n/2. And in the first half of the list, runtime increases linearly from 1 to n/2 and in the second half decreases from n/2 to 1.

3.

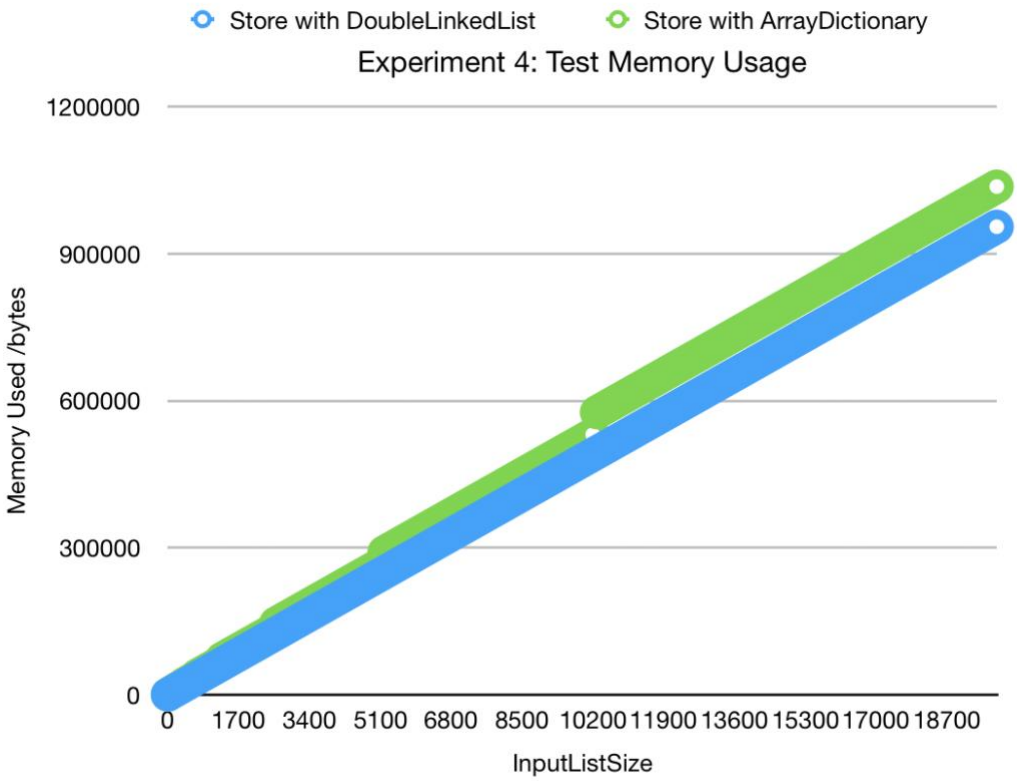


4. This is the same as our hypothesis since when indices < 10000 the runtime increases linearly and decreases linearly otherwise.

Experiment 4

1. Test memory usage of double linked list and array dictionary tests memory usage when adding random items to DoubleLinkedList as the size increases
2. we think that dictionary may use more space than list. As the size increases, both of their memory usage will increases linearly since each iteration through the loop would take up additional memory to store the data., but for the dictionary, their may be a sudden boost of usage because everytime the when the size reaches capacity, it will double the capacity.

3.



4. It is the same as our hypothesis.