1. The primary design difference that we had to implement to handle null entries was when traversing through the list, say for instance if we wanted to check the indexOf(null), we had to include two checks in our if statement. It would be (if temp.data == null || (temp.data != null && temp.data.equals(item))). This is because in order to check if something is null, "==" must be used, and if .equals is called on a null entry, a null pointer is thrown, so before we call .equals, we had to check that what we were calling .equals on wasn't null. For our Array, the exact same idea was used, where we called (pairs[i].key == key || (pairs[i].key != null && pairs[i].key.equals(key)).

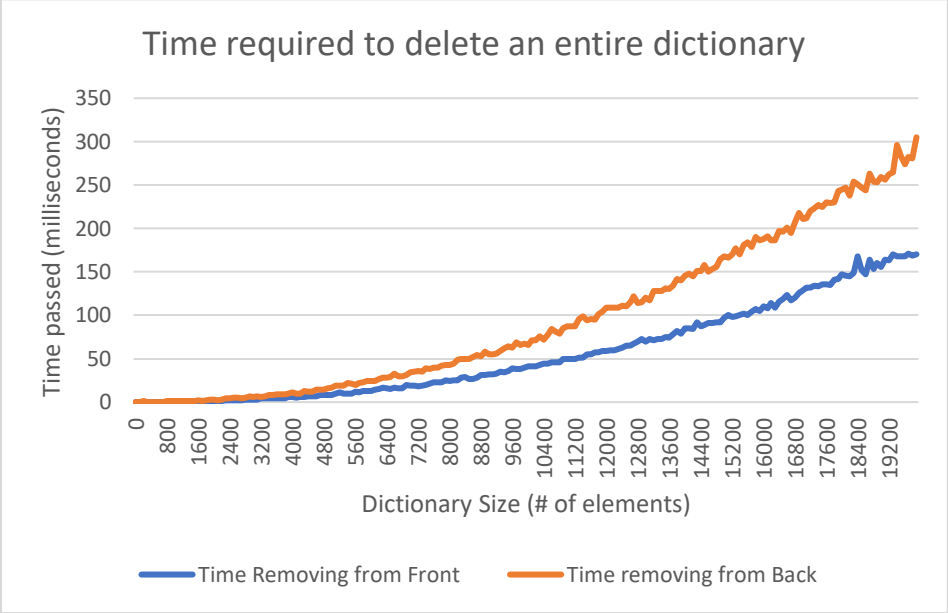
2.

Experiment 1

1) It's testing the amount of time needed to remove everything in a dictionary from zero index to the last index and also in the reverse order.

2) The amount of time needed to remove everything from front to back and from back to front should be the same because accessing every index is O(1) in an array, so regardless of accessing index[20000] or index[0], it should take the same amount of time.

3)



4) Removing from the front was quicker than removing from the back, since "I" is a key and not an index, we have to loop through the whole list to find "I" first. Consider removing from the list

0 1 2 3 4 5 6 7 8 9

Then to remove from the back each time would be roughly 10 (9 indices have to be checked before reaching 9, and then removing 9) + 9 + … 1 = 55 operations

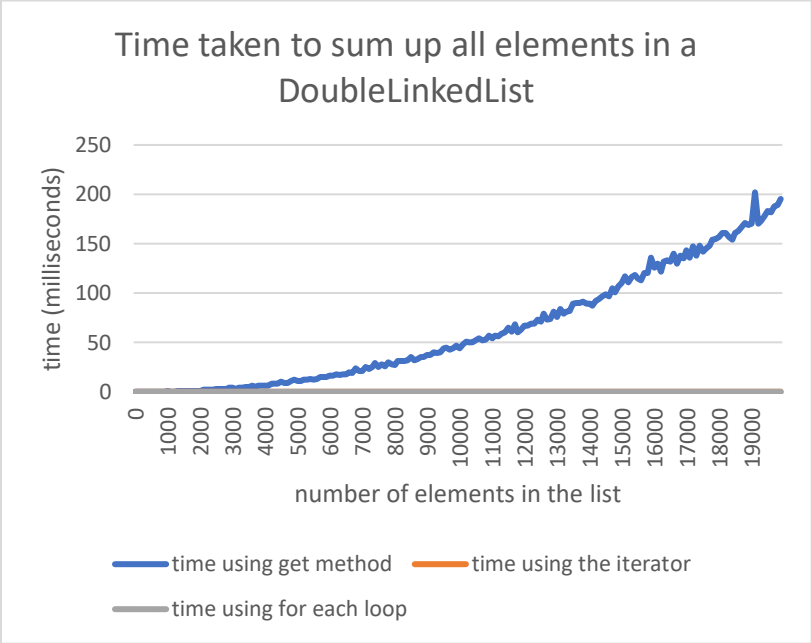
But when removing from the front we get something like

9 1 2 3 4 5 6 7 8 which is 1 operation (since we can access the last index right away)
9 8 2 3 4 5 6 7 which is 2 operations and so on, until we reach
9 8 7 6 5. At this point we have a total of 15 operations, now to remove 5 it takes 5 operations, to remove 6 it takes 4 operations, and so on for a total of 30 operations. This also accounts for the fact that for large dictionaries it's a little under 2x the time taken.

Experiment 2

1. It's comparing the time needed for summing all the elements in the list using different methods: calling get method in a for loop, using an iterator in a while loop, and using a for each loop.

2. Calling the first method should take the most time among the three approaches. Because the first method loops through the list from 0 each time, before it arrives at the index, while the iterator can reach the next element right away since it's pointing to the previous one, and a for each loop is basically an iterator.

3. 

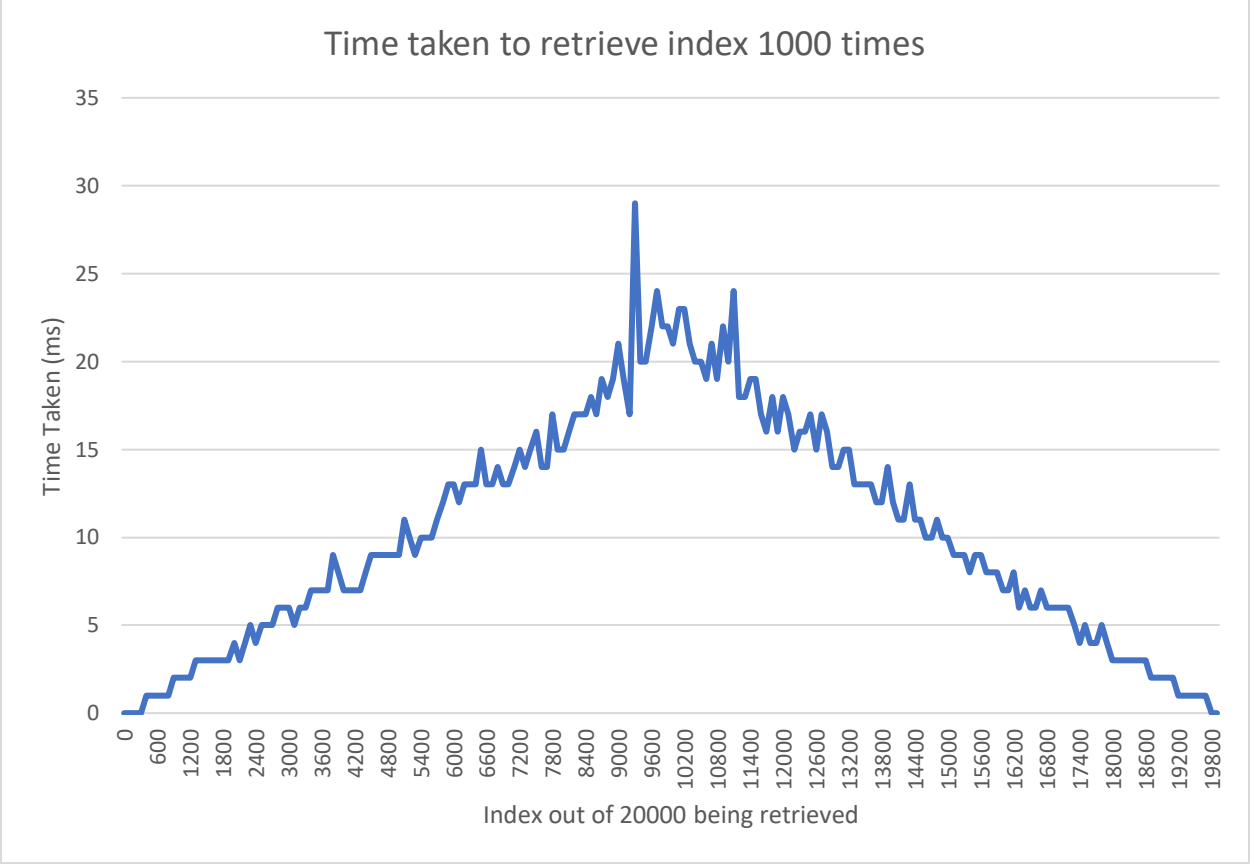
4. The results are similar to our hypothesis, time taken using the iterator and using for each loop are much less than calling the get method and both come up to be zero for all

cases. For the first method, we are using a for loop, so when we get whatever is at index(i), we start from the start for each loop. However, an iterator and the for each loop don't start over, they have a pointer to whatever is previous, so the iterator and for each loop only have to perform ~40000 operations – going to the next index and summing, whereas the for loop has to perform 1 + … + 10000 + 10000 + … + 1 operations (since if its closer to the back, we start from the back, it doesn't go to 20000).

Experiment 3

1. This experiment is comparing the time taken to get every 100th index 1000 times.

2. Our hypothesis is that it will peak in the middle indices and fall off as it gets closer to the outer indices, because our get method takes into account if its closer to the back or to the front. If its closer to the front, we begin at the front, and if its closer to the back, we begin from the back.

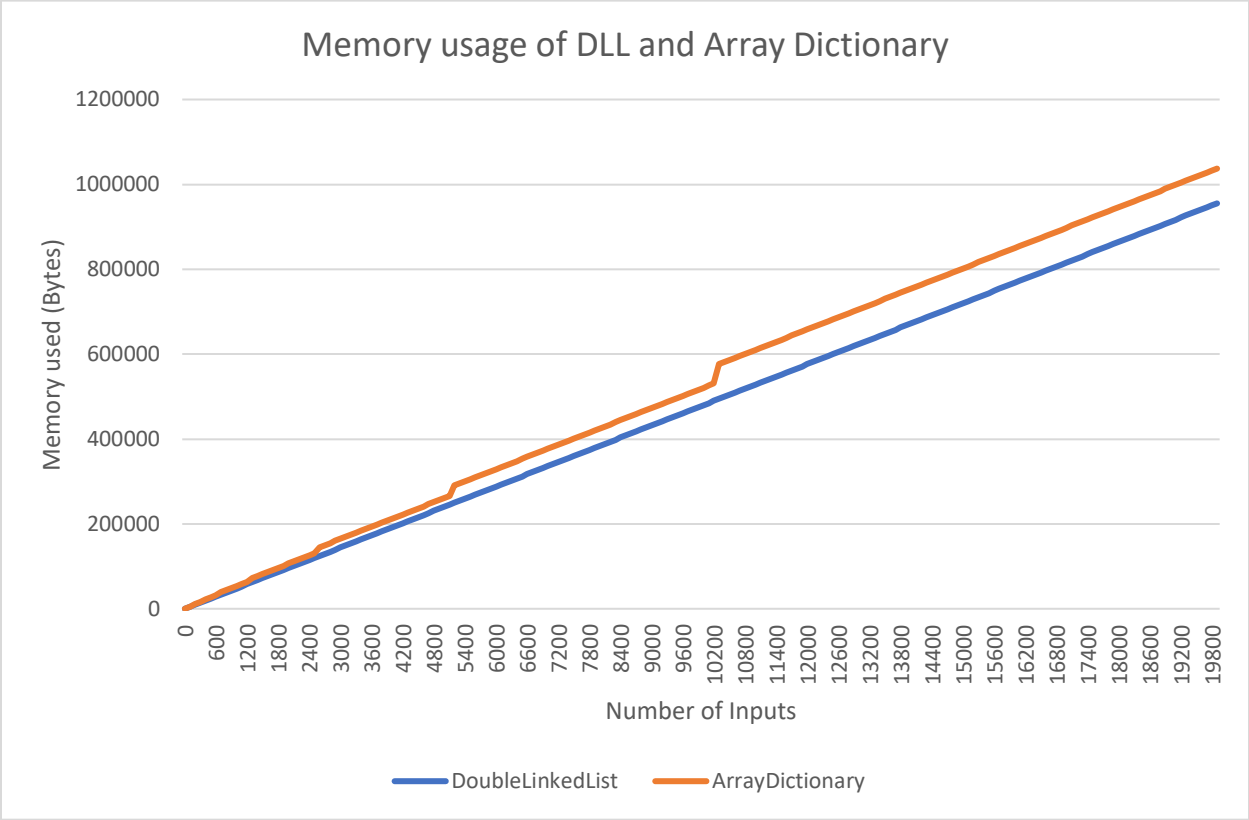
3. 

4. Our hypothesis was correct, and its reflected by the peak in the middle and the taper off from side to side.

Experiment 4

1. These trials are measuring memory usage by Dictionary size and DoubleLinkedList

2. Our prediction is that the memory is basically linear, since for the dictionary every new element added stores a new data point at each index, and will occasionally need to increase in size, but as the dictionary becomes larger and larger this increase in size rarely occurs. Our prediction is that a DoubleLinkedList will require more data since each link stores what's previous, what's next, and its input data whereas an array only stores what's at its current index.

3. 

4. Our hypothesis was incorrect and ArrayDictionary used a bit more memory, but both were approximately linear. An interesting thing to note is you can see where the jumps occur when we double the size of our array Dictionary. We think the Dictionary uses more memory because each index is a unique Key, Value pair whereas the DoubleLinkedList stores pre-existing nodes for its previous and front, and the items are the only things that are unique.