Group Writeup:

Team Members: Aaron Nguyen & Anirudh Kumar Subramanyam

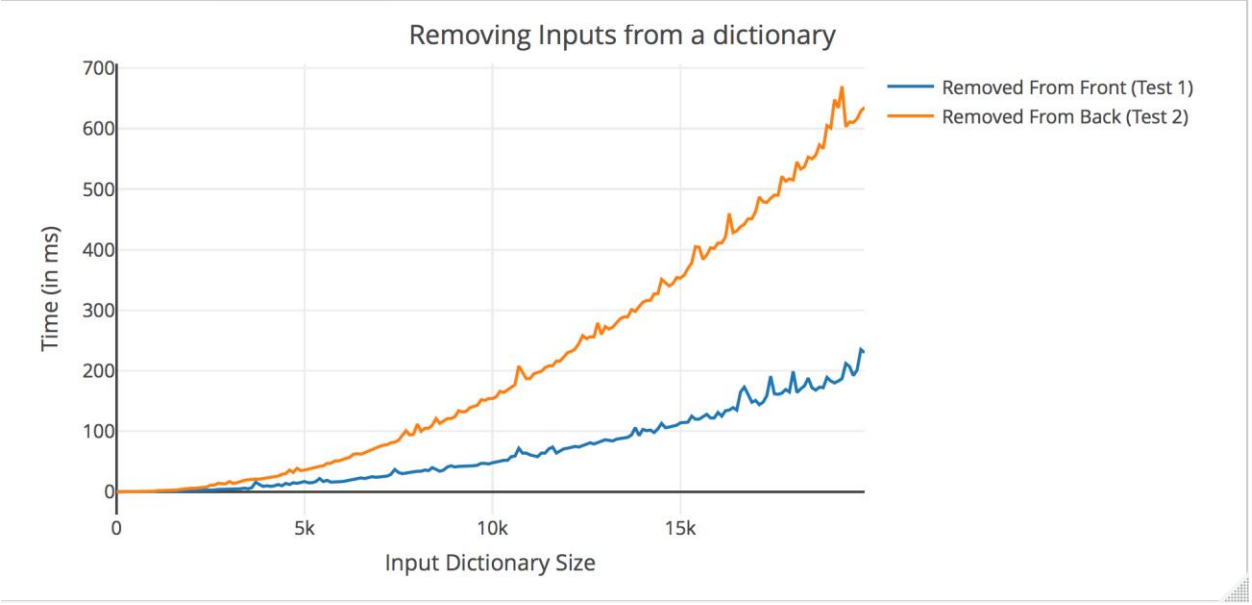
1) We compared the two objects using "==" to handle cases where the entry or key is null **before** we compared the two objects using ".equals()" for generic types. This was to ensure that null entries or keys were checked first to avoid "nullpointerexceptions" which occurs if we only used ".equals()" on null entries.

Experiment 1:

2.1) Experiment 1 is testing the speed of removing inputs from a dictionary. Test1 is testing how long it takes to remove all the inputs in the created dictionary from index 0 of the dictionary while Test2 tests how long it takes to remove the inputs from the end (index (size of dictionary) of the dictionary.

2.2) Test1 should be faster than Test2 because when removing from the front of the dictionary, it doesn't have to continually check for the matching key once it has been removed. It is simply then just shifting the rest of the dictionary to the left. Test2, however, has to do the check for every input in the dictionary until it reaches the end to remove the end input of the dictionary.

2.3)



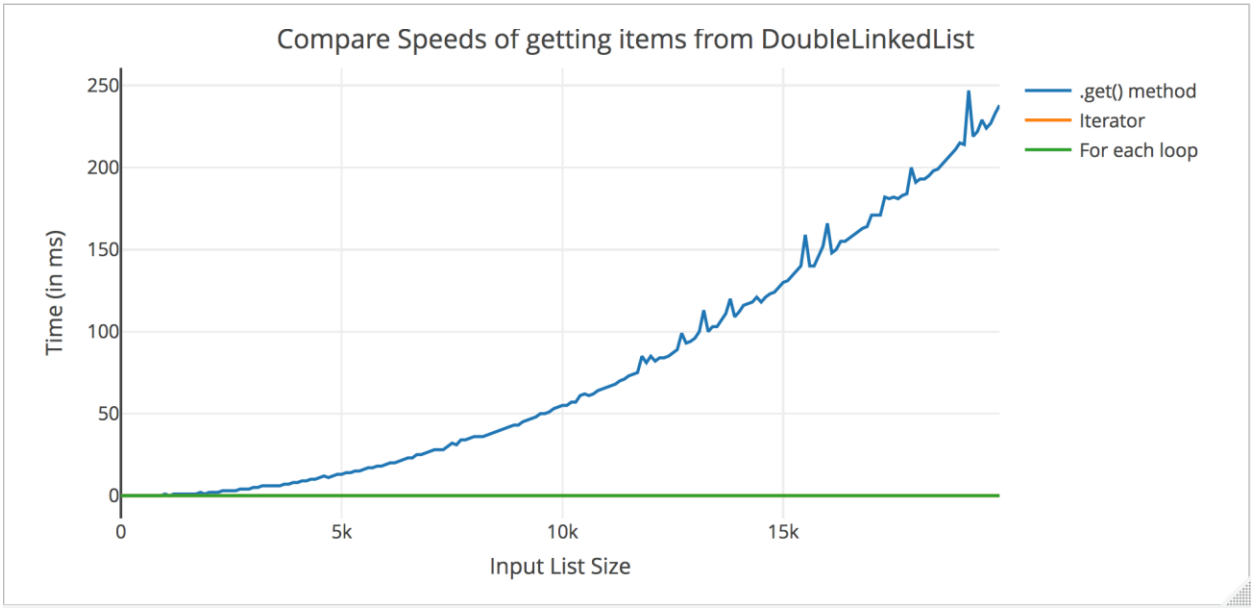
2.4) Our results went along the same lines as our hypothesis. It took test 2 much longer as the dictionary size grew to remove from the back than the time it took for test 1 to remove from the front of the dictionary. Again, these results are most likely from checking each input of the dictionary for test 2, versus checking the input once in the front and just shifting the remaining inputs to the left.

Experiment 2:

2.1) Experiment 2 is testing the speeds of the “list.get(int)” method, the list iterator, and the for each loop for different list sizes.

2.2) We predict that the .get() method will take longer and have an exponential-looking curve since it has to reiterate through the list as it gets the next item in the list. The curve should increase faster as more inputs are being added to the list. The iterator and for each loop should be much faster than the .get() method since they do not have to return to the front or back to reiterate. They simply traverse through the list once.

2.3)



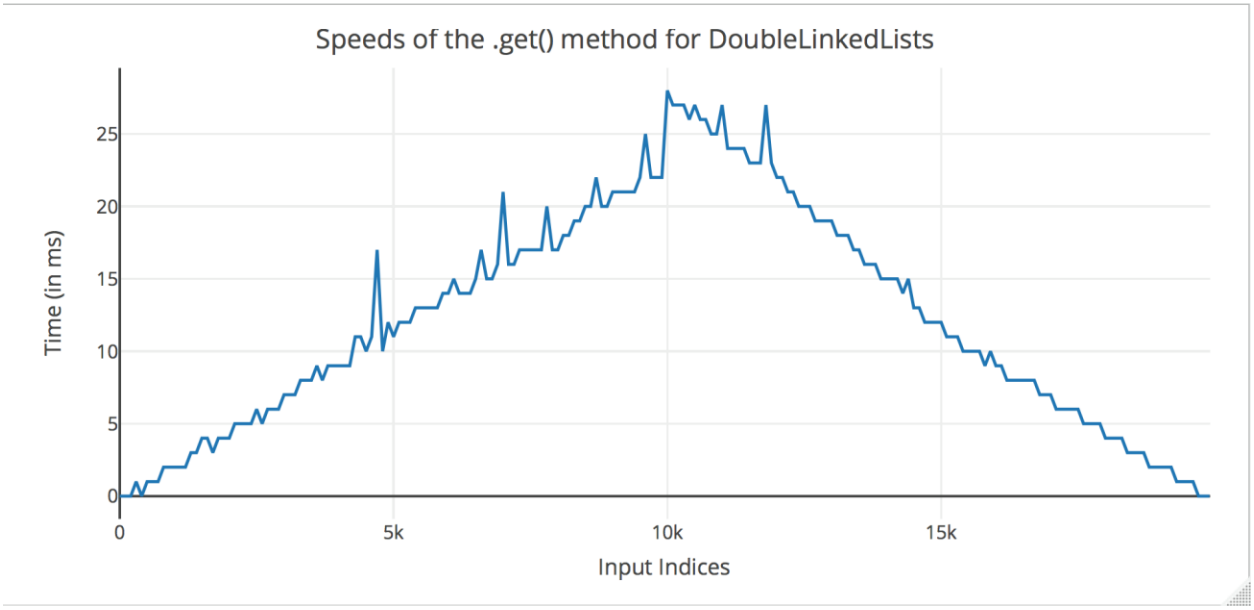
2.4) The results supported what we predicted in our hypothesis. The curve of the line for the .get() method did have an exponential look to it as more items were added to the list. This makes sense because the .get() method has to reiterate through the list as it gets each input from the list, which costs more time as the list increases in size. The iterator and for each loop, however, had a constant line at 0 milliseconds. It made sense that they were both much faster than the .get() method once the list grew larger because both these methods do not have to iterate through the list more than once. We think that the iterator and for each loop were rounded down to 0 ms because the list of items had not grown large enough to have a significant impact on the run times to make them not be rounded to zero.

Experiment 3:

2.1) Experiment 3 is testing the speed of the get method by repeatedly getting the same input numerous times for each input in the list.

2.2) We predict that the times should be fastest near the beginning or at the end of the dictionary. This is due to how we designed our code to start from the front or back based on the index parameter that is taken in by the ".get()" method. Times should increase when the index is around the middle of the list because it would have to traverse starting from the front or back of the list.

2.3)



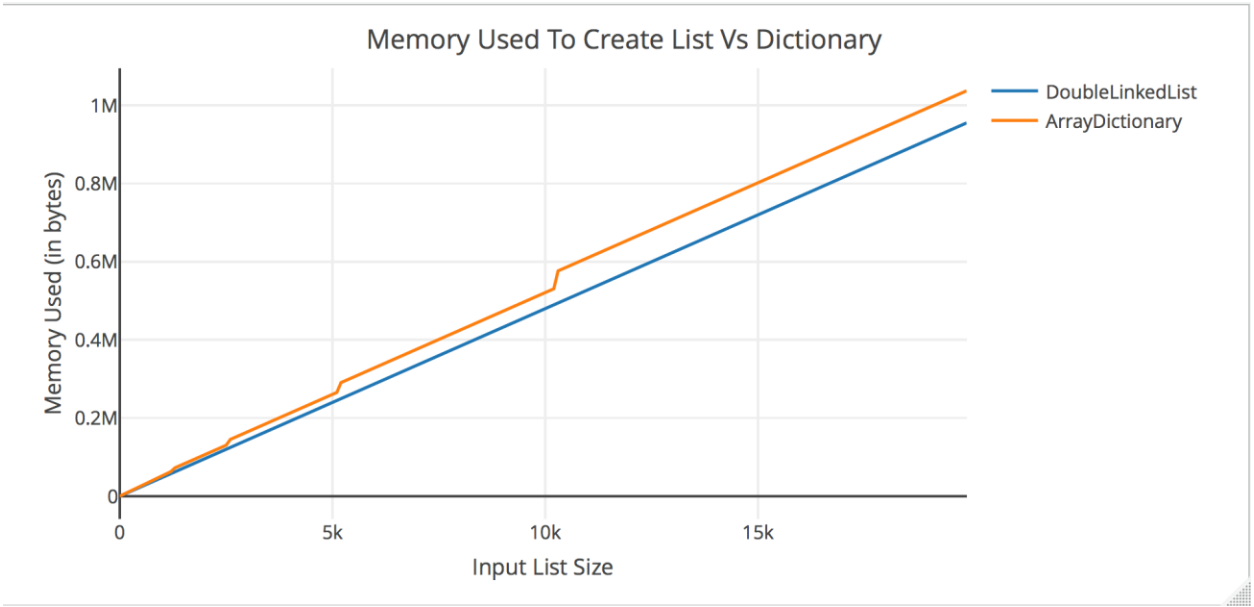
2.4) The results were along the lines of what we predicted. It created a triangle shape line graph which indicated that it was much faster if the index was closer to the front or back of the list. The longest times were recorded when the index was near the middle of the list.

Experiment 4:

2.1) Experiment 4 is testing the memory usage when constructing a list (our double linked list) and when constructing a dictionary (using our Array Dictionary). Test 1 corresponds to the memory usage for a list, and test 2 corresponds to the memory usage for a dictionary.

2.2) We predict that the ArrayDictionary will take up more memory since we designed our dictionary to double in size when it runs out of space to add an input. Our DoubleLinkedList does not require a space doubling effect since it can just add a node. This means that it will only take up enough space for its nodes.

2.3)



2.4) The results supported our hypothesis. You can see how the ArrayDictionary doubles when the line jumps suddenly upwards. It can also be seen how this causes the ArrayDictionary to take up more space than the DoubleLinkedList. The DoubleLinkedList had a steady increase in memory usage due to the steady increase of input list size.