CSE 373 Project 1 Group Writeup

Charles Tung Fang and Leon Zhang

Question 1

For the first turn in of our part1, we did not consider handling the null entries of keys. We only considered checking if the non-null key matches the key in dictionary by calling .equals(). We realized that equals() only checks to see if two objects are "content equal". When calling .equals on null object, the method will throw a null pointer exception.

To fix this, we added an extra test case to account for null case. To match null entries of keys, we need to use "==". "==" checks to see if two objects are "address equal". Two null keys must have the same address, therefore it will pass "==" test.

Our final version of handling both null and non-null input is the code shown below:

```
this.pairs[i].key == key || (this.pairs[i].key != null && this.pairs[i].key.equals(key))
```

The two test cases are separated by "||" logic. The first test `this.pairs[i].key == key` handles the null input, and the second test case `this.pairs[i].key != null && this.pairs[i].key.equals(key)` handles non-null input.

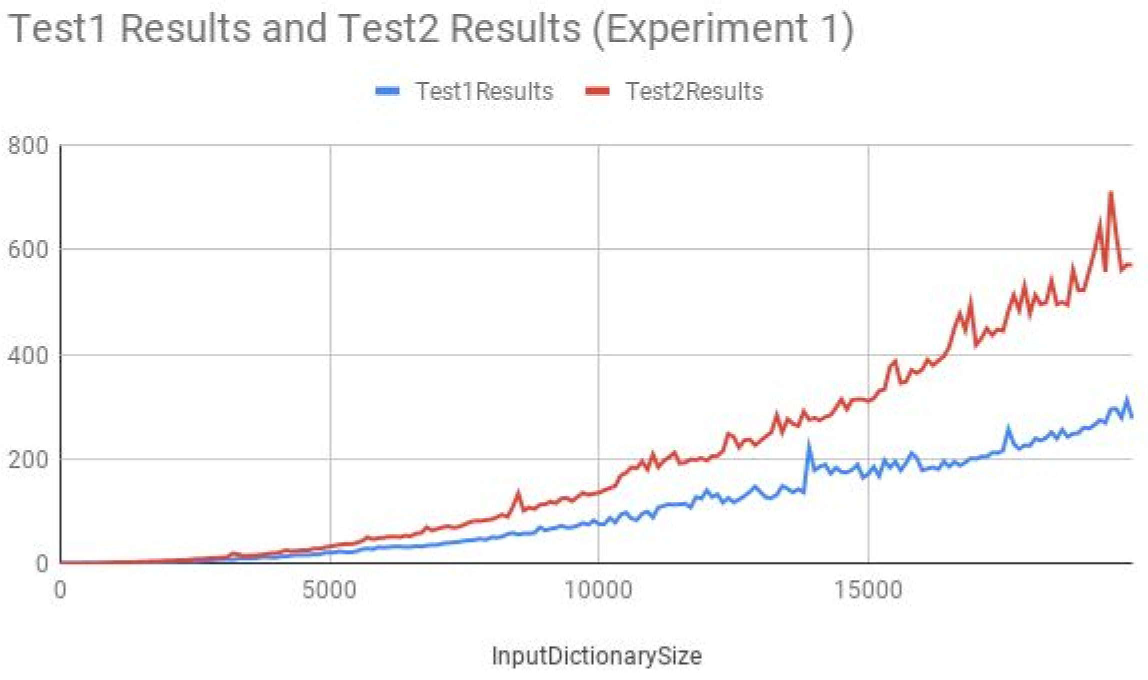
Question 2

Experiment 1:

1. This experiment is testing the time cost of the remove function in ArrayDictionary class. It fills the tested dictionary with elements first then run the first test, which traverse the array dictionary from the front to the end and remove element one after another. On the other hand, the second test traverse the array dictionary from the back, which mean the high index elements will be removed first.

2. Since there is an outer for loop and there is a for loop in the remove function, we predict that both tests will give quadratic curves.

3.



4.

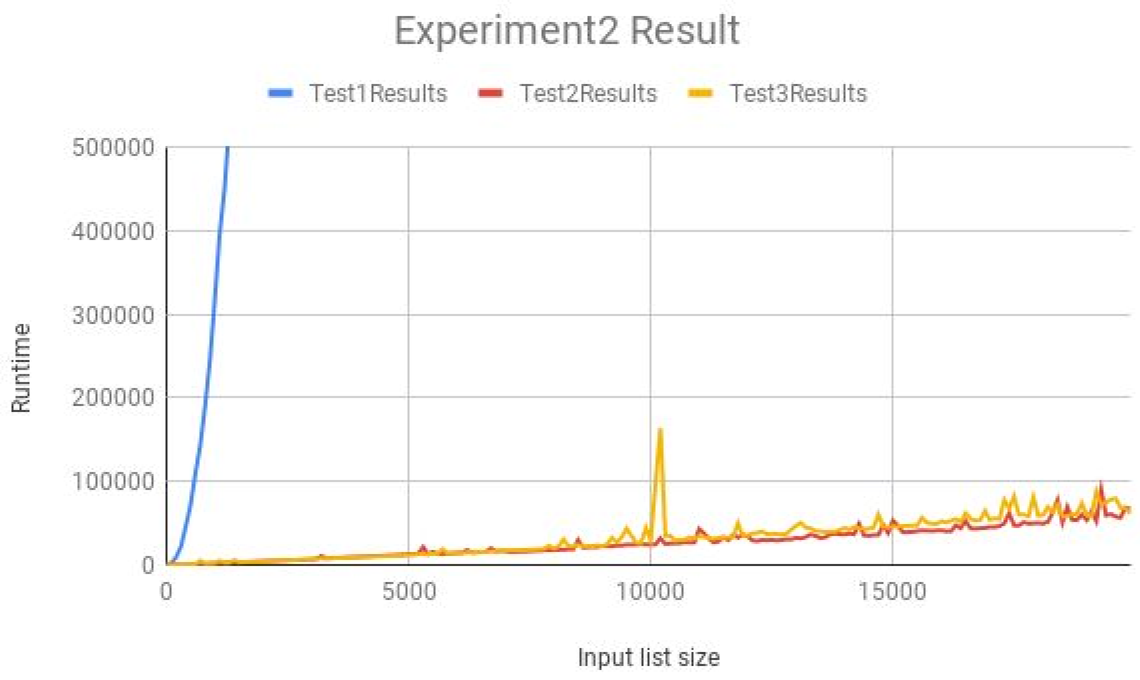
The graph we obtained after running experiment 1 confirmed our prediction. Both tests have quadratic curves, which represent that they are O(n^2). However, we notice that Test 2 is a bit more expensive than test 1. We believe the reason for that is because in order to remove starting from the end of the array, we need to traverse the whole array from index 0 to its size first before we do any actual remove work.

Experiment 2:

1. Test 1 tests for the runtime of get() method getting all the elements in the Double Linked List with a for loop. Test 2 tests for the runtime of iterator traversing all the elements in the list. Test 3 uses for each loop to traverse all the elements in the list. Again it is an implementation of iterator.

2. Test1 uses two nested for loop to scan through and get all the elements in data structure. The runtime lies in O(N^2) class. We predict the runtime will increase quadratically as the input size become greater. Test 2 and 3 both use iterator object that has a reference on the current node. The iterator has the access to the current node (it saves what was previously looked at, so it doesn't have to re-traverse through the list every time). We predict the runtime of test 2 and 3 is O(N). runtime will increase linearly as input size increase.

3.



4. The result we graphed is consistent with our hypothesis. Test 2 and 3 use iterator to traverse through list and they both show linear runtime with input list size. Test 1 shows a quadratic relationship. This is consistent with our knowledge of two nested for loop runs in O(N^2).

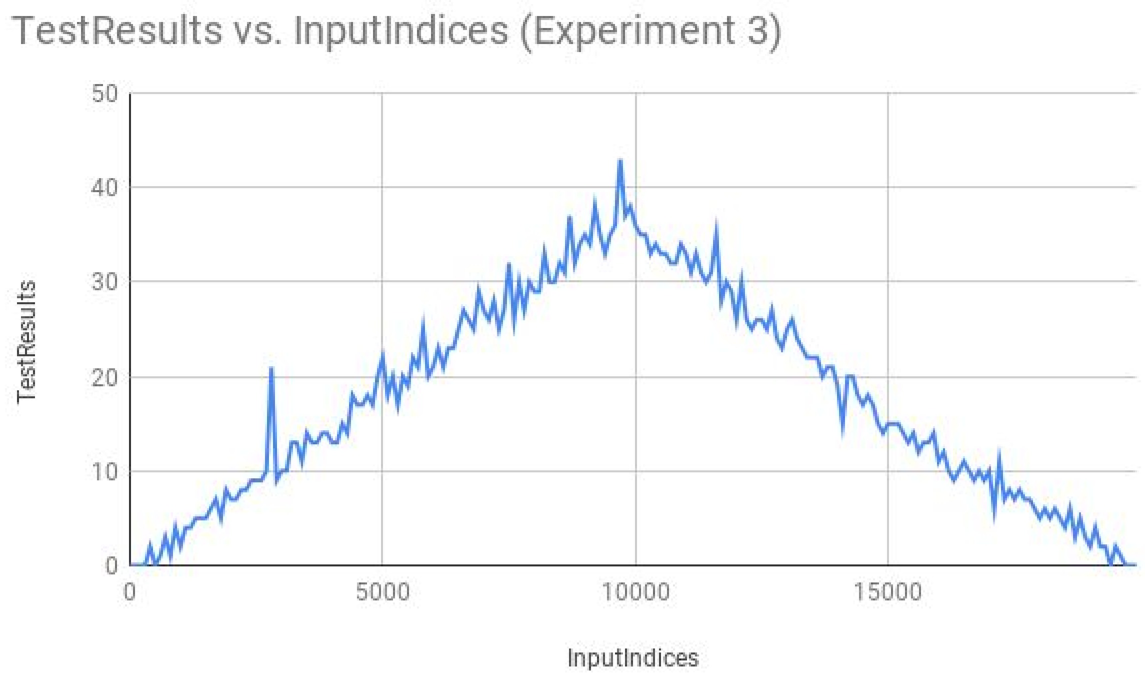
Experiment 3:

1. This experiment is testing the time cost of get function in the DoubleLinkedList class and the index parameter passing to it.

2. We are expect a linear curve. The curve should be increasing first then decreasing after it reach its middle index. The reason for that is because we implemented a helper method, skipTo(index), for our get method. The helper method check if the given index is greater than size/2 or not. We search from back of the list if it is and from the front of the list if the given index is smaller than size/2. Therefore, the cost of given index at the middle of

the list is the most expensive. We expected the curve to be linear because the time cost of get is proportional to the distance from the given index to the middle of the list.

3.



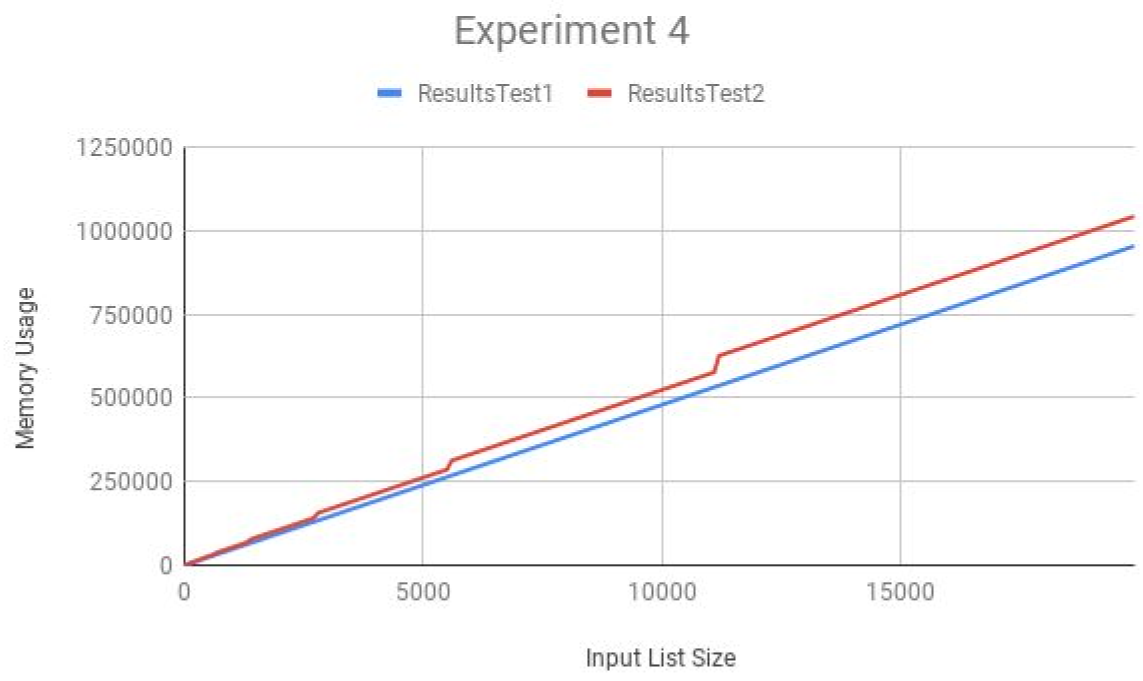
4. The result is consistent with our prediction. We are certain to get a peak at middle and linearly increasing from 0 to the middle and linearly decreasing from middle to the end.

Experiment 4:

1. Both tests check the memory usage of creating a data structure we implemented with a given size. Test 1 tests the memory usage of creating a double linked list. Test 2 tests the memory usage of creating an array dictionary.

2. We predict the memory usage of creating an Array dictionary will be double the amount of memory as double linked list because dictionary has key and value pairs (two types of data).

3.



4. Turns out the memory usage for creating an array dictionary (red) is only a little greater than creating a double linked list. The explanation for this is that they are both data structures that the memory usage is direct proportional to its size. The number of fields inside these data structures do not have big influences on memory usage. We noticed that for array dictionary there are certain places where there is a kink. This where is array has used up all the capacity, and it needs to create a new array with double the capacity, which takes up more memory.