Name: Zhenghao Guo, kuo Yan

January 31st ,2019

Group Writeup

1. for solving the null and non-null entries in doublylinkedlist, in my code, we only need to look at the

indexOf(items) method in which we are only given the items but should return the index of it. However, we can't

only do the statements like:

```
while(!head.data.equals(item)){
    head=head.next;
    index++
}
```

Because the given type is generic type T which is a object, We need to use .equals() to compare value.

However, when we are confronted with the case of head.data!=null, we can't use the .equals(null). Therefore, I used a structure like

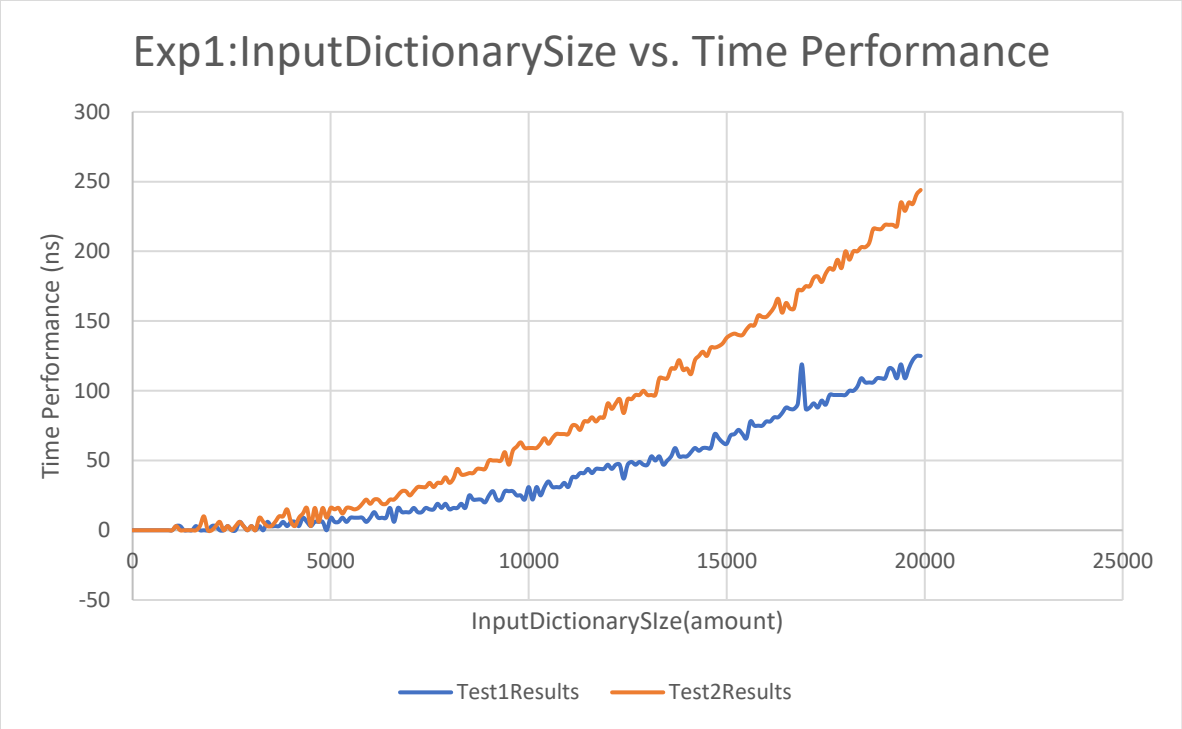
```
if(items==null){
  //iterate through the list to the end until we find the node whose data is null
}else{
  // while iterate through the list first check if the if(head.data!=null&&!head.data.equals(item))is true
}
```

For handling the null key in the dictionary, I used the same structure to seperate the case of key==null:

```
if (key==null){
    // we iterate through the whole array according to its size until we first find (key==null)
}
else {
    //iterate through until we first find the key in pairs which is not null then the pairs[i].key.equals(key).
    //by doing this we can continue to iterate through the whole list when meeting null key.
```

}

experiment 1:



hypothesis:

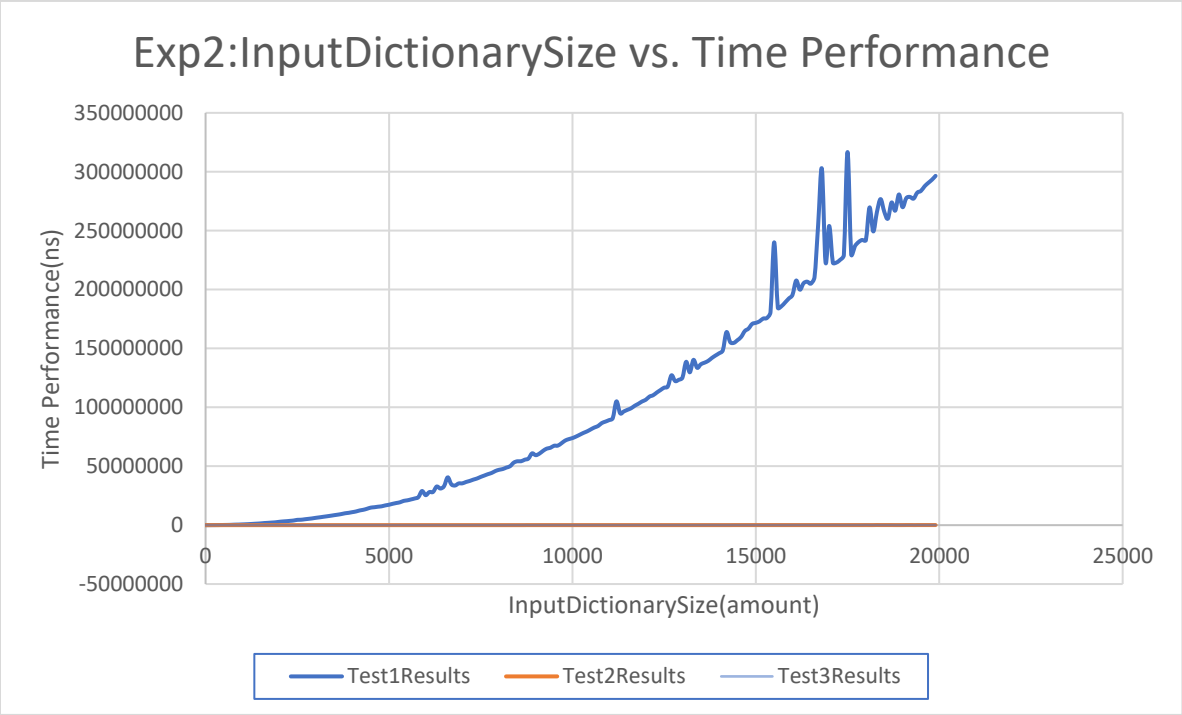
this is testing the performance while we remove data from the beginning and the end of the dictionary, respectively.

they all take constant time to remove items from array. However, test 2 might be faster than test1

because we should take one more step to replace the first element with the last element when we delete the beginning.

According to the graph, I am not correct. test 1 is always under test2. I think this is because while we remove from the beginning, we always find the index at without going to the end, however while we do remove from the end, we need to find the key at the very end. Which takes more time.

Experiment 2:



This experiment test time performance of three kinds of iteration (traditional loop, for each loop, iterator)

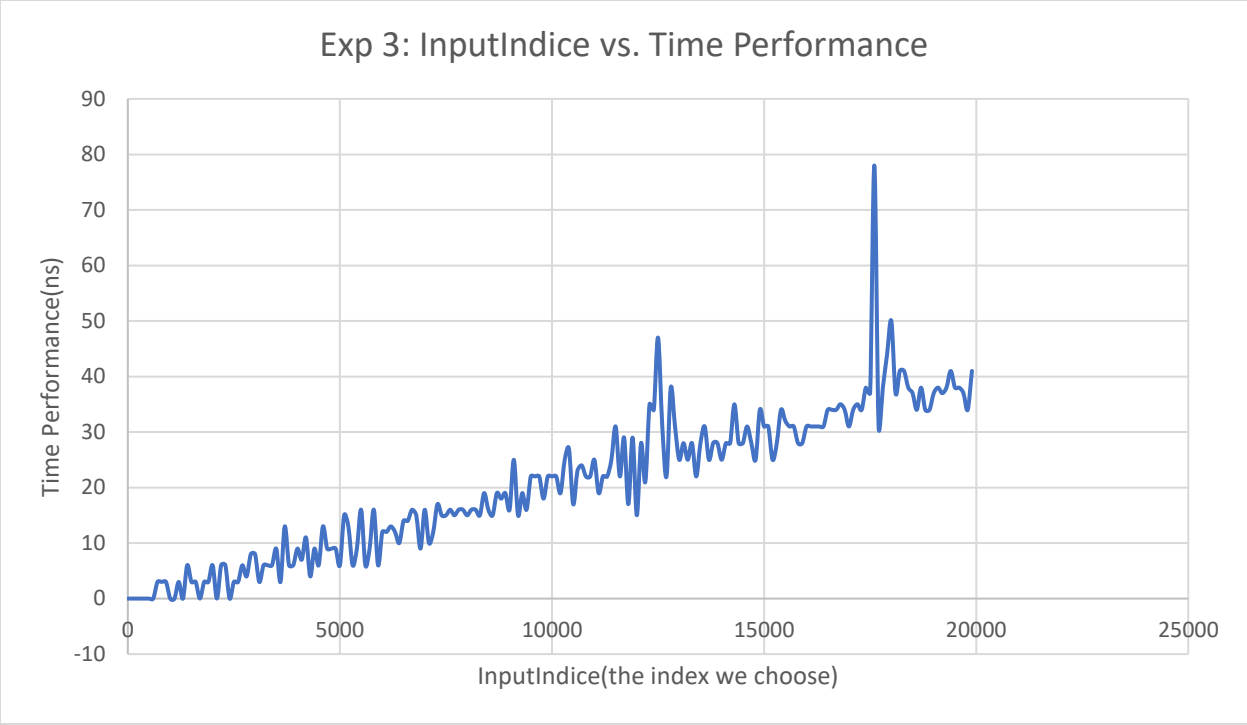
I predict that for loop will have O(n^2) because it has get(i) in it which is like a nested for loop.

but the for each loop and iterator can get the element during iteration which means they have will consume same

time.

According to the graph, for each loop and iterator are overlapped and the traditional for loop is behaving like n^2

experiment 3:



hypothesis: this test is trying to the time performance of get() method, with different amounts out input.

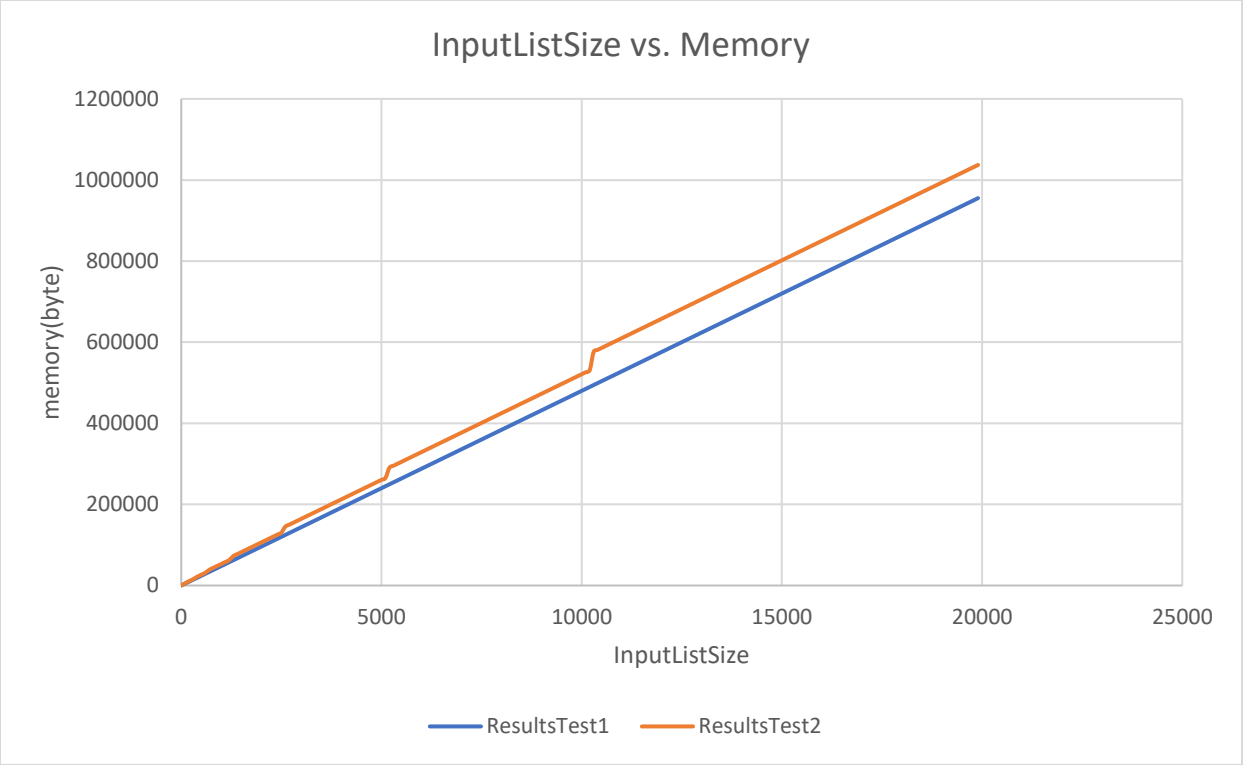
My guess will be that the time will grows linearly with the input.

According to the graph, I know I missed something in my prediction. With the input amount increasing, there are

some spikes in the graph, these situations are because while we are getting the the value in the middle of the doublylinkedlist, even though we can iterate through the list from the start and end. we still need to

iterate through halfway to get the middle value.

experiment 4:



hypothesis:

This test is testing how much memory (in bytes) is taken up by initializing the same size of array dictionary and

doublylinkedlist. obviously, doublylinkedlist is compacted every time while array might have size*2 every time we

are out of space. arrayDictionary will have more memory used.

According to the graph, yes, my hypothesis is correct.