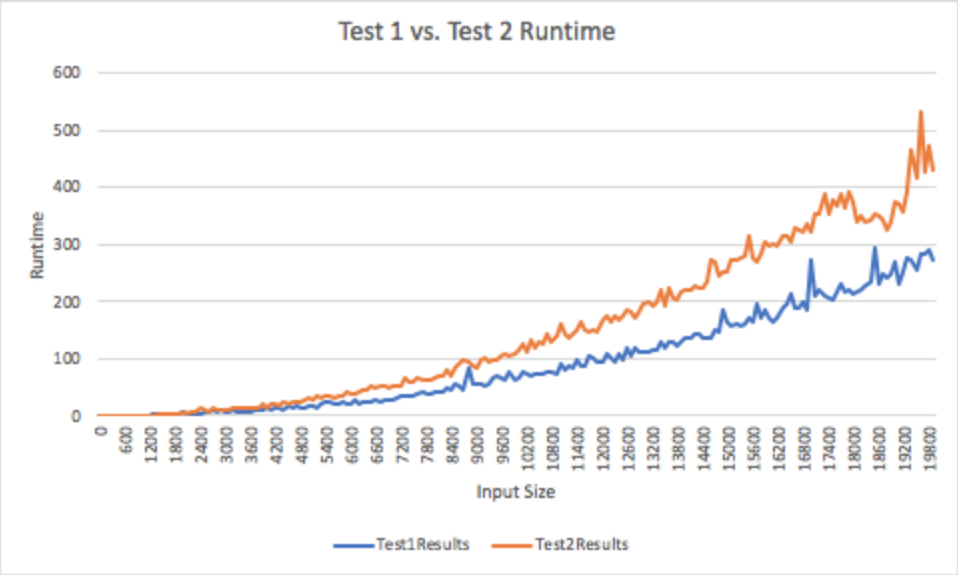
Hoang Nguyen and Andrew Yu
Project 1 Group Writeup
1.
When we write the function indexOf, we have to implement a logic to check if the data of a node is equal to what we are looking for. First, we just write:

```
if  (temp.data == item || temp.data.equals(item))
```

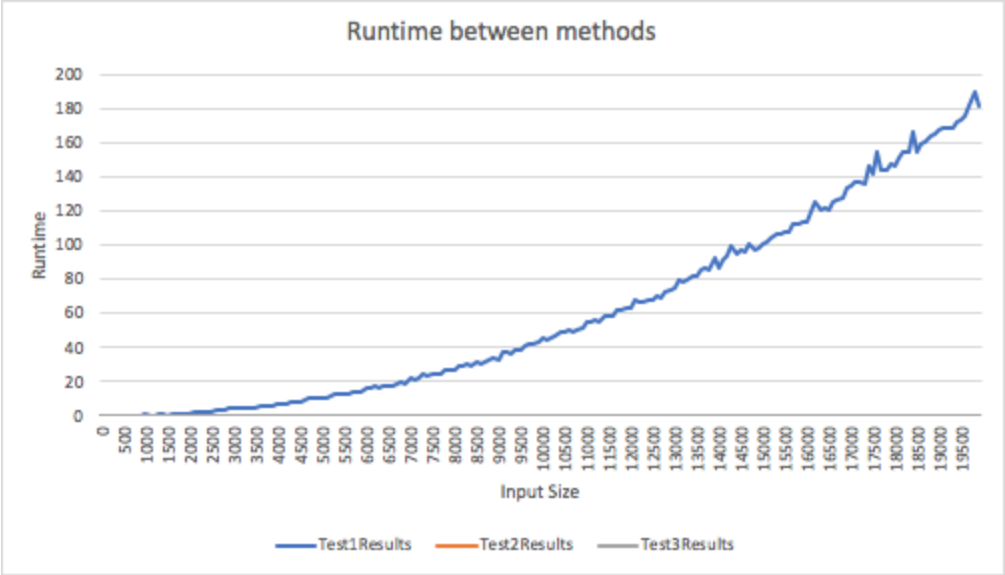
For cases when temp.data is not null, this code does simple comparison. However, when temp.data is null, and the code tries to do temp.data.equals() which is null.equals(), the code throws a NullPointerException(). We know that we have to check if the temp.data is null first, and then do the equals() method. At last, our code looks like this:

```
if  (temp.data == item || (temp.data != null && temp.data.equals(item)))
```

2.
Experiment 1:
Experiment 1 is testing if removing starting from the end or removing starting from the start is faster.
We think that deleting from the start is faster. The reason for this is that each time, we call delete, we have to call indexOf(), which will go from the start to the end to find the value. If we delete from the start, it will be faster for indexOf() to find the value.



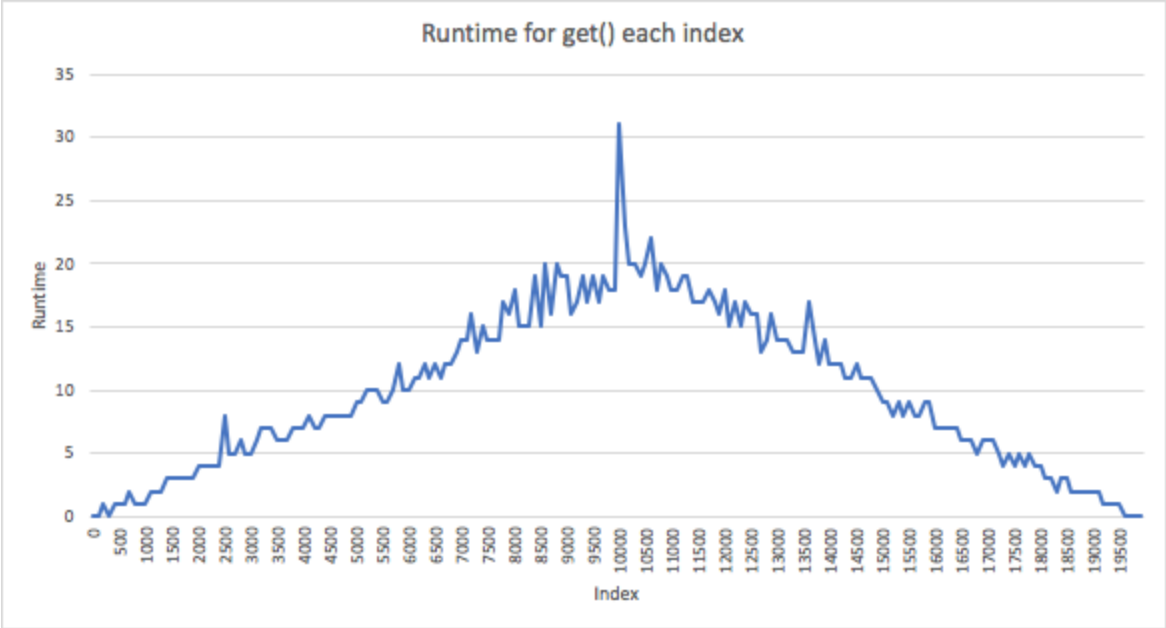
The results agrees with our assumptions. Test 2 delete from the end of the list and therefore slower than test 1 which deletes from the beginning of the list. We also finds out that the difference is more noticeable when the input size is larger
Experiment 2:
Experiment 2 is testing the runtime between using get(), iterator, and for each loop while accessing the element of the whole list.
We think that using for each loop and iterator would be faster than the get() method. The get() method has to get to an element every iteration of the for loop, while the iterator and for each already have access to the next element in constant time.



The result agrees with our guess. The line in blue is the runtime of the get() which take linear time. The orange and gray is exactly 0 time (constant time) because the methods have quick access to the next element

Experiment 3:

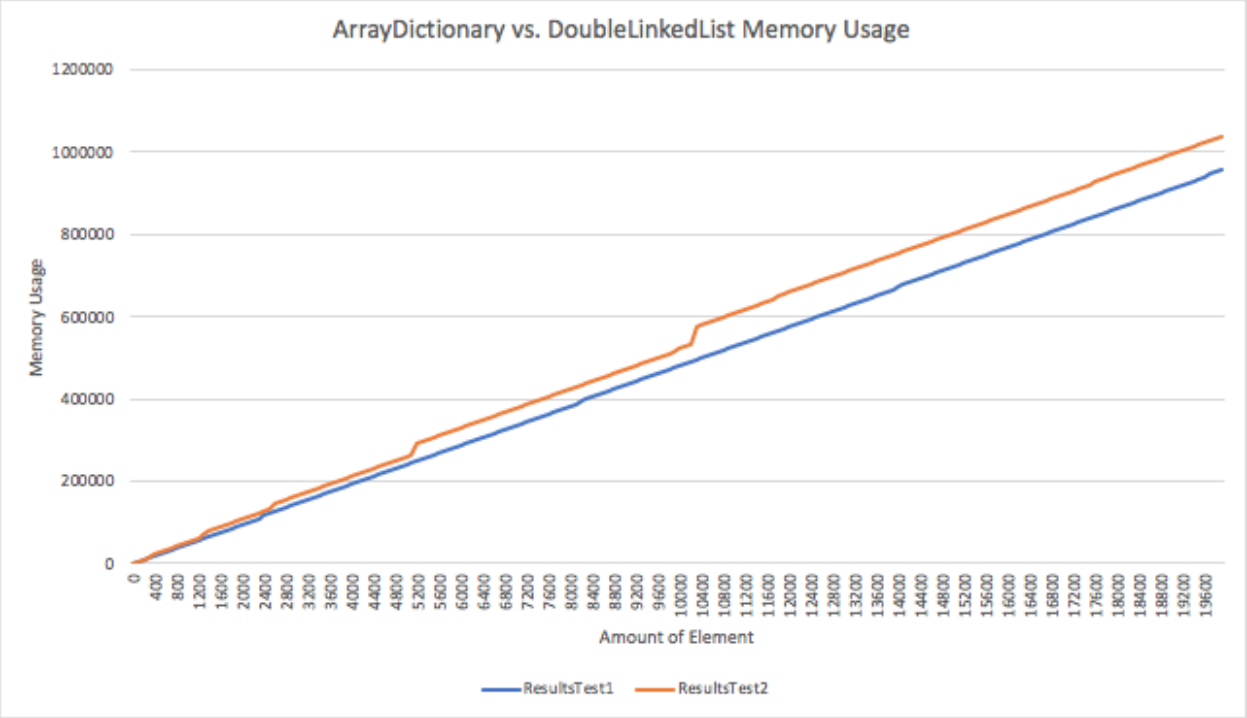
Experiment 3 compares the runtime of the get() operation for each index of DoubleLinkedList. We guess that get() near the front and near the back would be really fast because we instructed the method to check if the index is near each end. Then it would place a pointer at the end that is closer. As a result, get() an index near the middle of the list takes the most time.



The graph agrees with our guess. The graph looks like a triangle, and the highest point is in the middle. This makes sense because get() near the middle would take the most time.

Experiment 4:

This experiment tests DoubleLinkedList and ArrayDictionary's usage of memory to see which one takes more.
We guess the the ArrayDictionary takes more memory because of two reasons. First, the ArrayDictionary store and Pair Object for each index while the LinkedList only store one data. Second, the ArrayDictionary almost always has unused memory because it has to double it's size every time one inserts past the capacity



The graph agrees with our guess. The orange line express the memory usage of Array Dictionary, which is higher than the blue line - the memory usage of DoubleLinkedList

Extra Credit 1: drawSpline()
Originally, the graph was a scatter plot (multiple points on the graph). We added curved lines to connect the dots. Doing this would help the user to recognize the shape of the graph and what function the graph refer to.
We used the JFreeChart library, the XYPlot library and the XYSplineRenderer. We changed the original JFreeChart to an XYPlot instance, and then changed the rendering of the XYPlot.

Extra Credit 2: Control flow
We followed the instruction on the extra credit page number 4, and add a while(cond, body) function that will continue to evaluate body if cond doesn't equal to zero.