Name: Sasha Babayan & Brian Park
Date: 5/14/2013
Project 2 Write-Up

1. Our group consists of Brian Park and Sasha Babayan

2. The tutor at CLUE for CSE

3. This project took a total of about 65 hours for Phase A and Phase B amongst the two of us. I think the hardest part of the project was implementing the AVL tree, especially because of the rotations and testing the structure property. Since most of the code was taken from the pseudo code in the textbook it didn't really seem like a necessary data type to implement so perhaps to improve the project you could remove AVLTree.

4. None

5. I designed the JUnit tests to test for various different edge cases and samples of "normal" cases. Generally speaking, I always tested each method to see that it behaved correctly when an empty data structure is passed, as well as a data structure with a single element and multiple elements. For many of the test cases, I attempted to check for duplicate values, as well as negative values (when applicable). I tried to test the functionality of every public method of each ADT, as well as various cases for each algorithm. I also did my best to avoid redundant tests; I did not attempt to test every possible value. I attempted to be as specific as possible in my testing; if something was wrong in our debugging, I would isolate the issue to the simplest test case that still failed.

6. BinarySearchTree uses an explicit stack because we must iterate through each node individually and recursion doesn't return a single node easily while continuing to iterate through the structure. Since a stack is never larger than the height of the tree and we know the height for an AVL Tree, we could avoid resizing in the midst of iteration for an AVL Tree by initially setting the stack's size equal to the height of the root node.

7. AVLTree/BSTree: Since we know that the maximum values of the tree are the most bottom-right nodes, we can iterate in reverse order, starting at the bottom-right most node, these would both be O(n) time where n is the number of elements in the AVLTree/BSTree. Hashtable/MoveToFrontList: Go through each element to find the maximum and return it, for the next highest count go through each element (that wasn't a previously returned one) and return that. Follow those steps until all elements are returned. These are both in $O(n^2)$ time where n is the number of elements in the Hashtable/MoveToFrontList.

8. The hasher passed must return an integer and the comparator must properly compare words so that it can distinguish between words. For example, if our comparator always returns a 0 (the two

elements are equal) then regardless of the number of words our file has, our hashtable will always contain only 1 DataCount object with a count equal to the total number of words in the file.

9. We found that hashtables and quicksort produce the fastest results in terms of running time. We determined this by calculating the running time for each possible combination of DataCounters and sorting algorithms for two different texts. We ran 3 trials then took the average time of those three trials and based our answer off of that. The charts providing the running times are listed below.

Hamlet					
DataCounter	Sort	Running Time (in ms) Trail 1	Running Time (in ms) Trail 2	Running Time (in ms) Trail 3	Average Running Time (in ms)
BinarySearchTree	Insertion Sort	1219	1366	1186	1257
	Heap Sort	313	324	375	337
	Quick Sort	238	163	201	201
AVLTree	Insertion Sort	728	881	585	731
	Heap Sort	168	202	216	195
	Quick Sort	196	181	184	187
MoveToFrontList	Insertion Sort	14692	7744	18570	13669
	Heap Sort	6775	15206	7000	9660
	Quick Sort	13977	15260	15360	14866
Hashtable	Insertion Sort	492	531	433	485
	Heap Sort	150	120	150	140
	Quick Sort	135	81	106	107

The New Atlantis					
DataCounter	Sort	Running Time (in ms) Trail 1	Running Time (in ms) Trail 2	Running Time (in ms) Trail 3	Average Running Time (in ms)
BinarySearchTree	Insertion Sort	1065	790	1001	952
	Heap Sort	228	190	223	214
	Quick Sort	86	82	98	89
AVLTree	Insertion Sort	342	341	305	329
	Heap Sort	163	171	180	171
	Quick Sort	85	109	98	97
MoveToFrontList	Insertion Sort	3074	3204	3120	3133
	Heap Sort	3049	1429	1471	1983
	Quick Sort	3313	3211	3235	3253
Hashtable	Insertion Sort	180	179	177	179
	Heap Sort	56	76	69	67
	Quick Sort	45	48	47	47

One specific contrived text that would produce a different running time output for MoveToFrontList is if all repeated instances of words occurred right next to each other. This would allow for easy look-up and incrementing because the DataCount node containing the desired word would already be located at the front of the list. As you can see MoveToFrontList produces our slowest output in terms of running time. However, when running times were calculated using the new contrived file with only 5 unique words and about 200 copies of each word next to each other, MoveToFrontList runs considerably faster. In fact it even runs faster than Binary Search Tree and AVL Tree on average.

Test File for MoveToFrontList					
DataCounter	Sort	Running Time (in ms) Trail 1	Running Time (in ms) Trail 2	Running Time (in ms) Trail 3	Average Running Time (in ms)
BinarySearchTree	Insertion Sort	142	151	161	151
	Heap Sort	155	160	119	145
	Quick Sort	43	81	40	55
AVLTree	Insertion Sort	93	42	92	76
	Heap Sort	19	20	78	39
	Quick Sort	28	20	52	33
MoveToFrontList	Insertion Sort	87	23	80	63
	Heap Sort	5	16	6	9
	Quick Sort	5	14	5	8
Hashtable	Insertion Sort	14	24	15	18
	Heap Sort	74	155	110	113
	Quick Sort	5	5	6	5

Changing the hashing function also affects our results. We designed an experiment where we calculated the running times with a new hash function of h(k) = 0. This is essentially a linked list because only 1 index is being used and our hashtable now produces one of the slowest results.

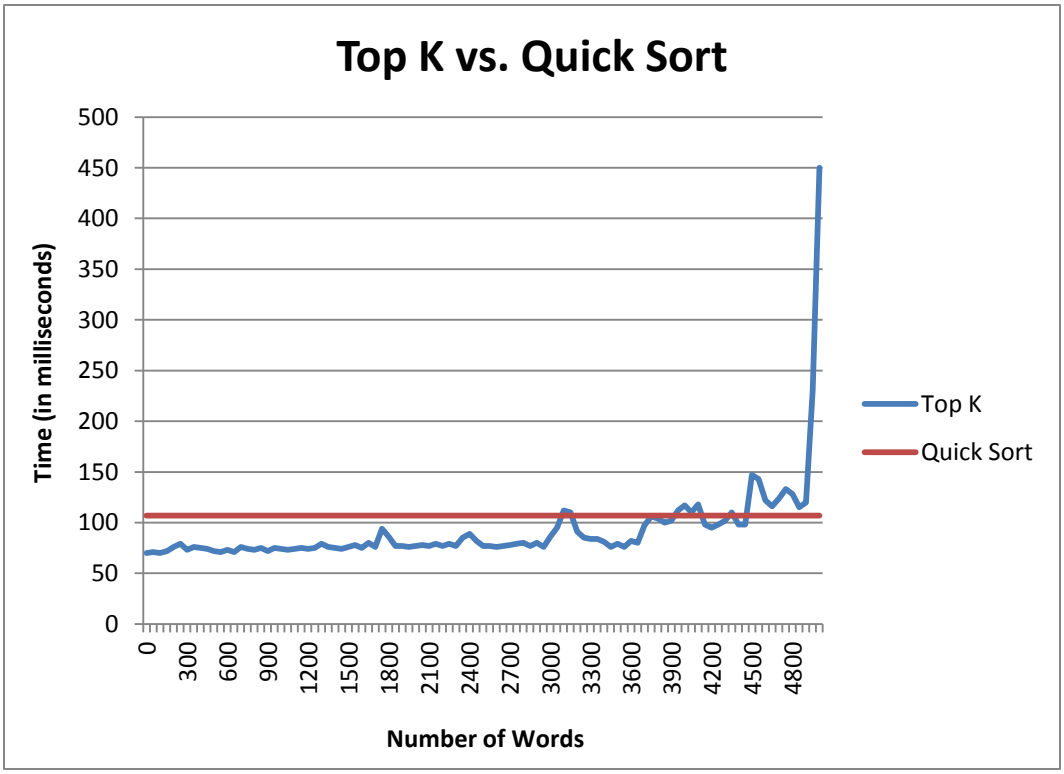
Hamlet.txt (using h(k) = 0)					
DataCounter	Sort	Running Time (in ms) Trail 1	Running Time (in ms) Trail 2	Running Time (in ms) Trail 3	Average Running Time (in ms)
BinarySearchTree	Insertion Sort	1129	1177	1074	1127
	Heap Sort	294	283	324	300
	Quick Sort	207	161	156	175
AVLTree	Insertion Sort	581	612	909	701
	Heap Sort	193	185	194	191
	Quick Sort	203	481	497	394
MoveToFrontList	Insertion Sort	14840	14738	7330	12303
	Heap Sort	6640	6585	14622	9282
	Quick Sort	14606	14327	14626	14520
Hashtable	Insertion Sort	6445	6484	6451	6460
	Heap Sort	7001	6484	3421	5635
	Quick Sort	3748	3968	6663	4793

Each experiment used a different text, but we essentially calculated the difference in time using System.currentTimeMillis() before we began running our code, and after. We then took the difference of the two times and stored those values in our table. We ran 3 trials and took an average to ensure better quality of data. We essentially had 12 blocks of code, each one corresponding to a possible combination of DataCounter and sorting algorithm which we ran 3 times for each text file/experiment.

Example:

```
long before = System.currentTimeMillis();
countWords(new String[]{"-b","-is", args[2]}); //args[2] is the file name
long after = System.currentTimeMillis();
System.out.println(after-before);
```

10.



For this experiment we created a for loop that decremented by 50 on each iteration of the loop and started at 5000 (the approximate number of unique words in hamlet.txt). We then plotted the running times relative to K. Our fastest sorting algorithm is quicksort and, from question 9, quicksort is fastest when using a hashtable so we plotted that average on our graph as well. As K grows topK also grows and after a certain threshold topK is slower than our nlog(n) algorithm. We attributed the initial large running time to caching (since our for loop starts with K ≈ n elements then decrements to k = 0). Since our nlog(n) approach becomes faster after a certain K, we could modify topK to use an nlog(n) algorithm if K is greater than or equal to some predefined x.

Example-

```
for(int i = 5000; i >= 0 ; i= i - 50 ) {
    String f = Integer.toString(i);
    long before = System.currentTimeMillis();
    countWords(new String[]{"-h","-k", f, args[2]});
    long after = System.currentTimeMillis();
    System.out.println(after-before);
}
```

11.

Text 1	Text 2	Correlation
Othello	Hamlet	1.7647691627533388E-4
Romeo & Juliet	Hamlet	2.620455785497559E-4
Macbeth	Hamlet	1.8931214147456797E-4
The Temptest	Hamlet	2.669301347631808E-4

Average: 3.58E-4

Text 1	Text 2	Correlation
Othello	The New Atlantis	6.687040024769987E-4
Romeo & Juliet	The New Atlantis	7.053996062610786E-4
Macbeth	The New Atlantis	4.7076072984178794E-4
The Temptest	The New Atlantis	5.039603559242517E-4

Average: 5.87E-4

Hamlet and The New Atlantis: 5.657273669233964E-4

Using Correlator our experimentation suggests that Shakespeare wrote Shakespeare's plays under the assumption that Hamlet was undoubtedly written by Shakespeare. Since Correlator calculates a running sum of the squared differences of each word's standardized frequency that occurs in both texts, texts that are extremely similar have a correlation closer to 0 because they have fewer differences. We gathered several works that were claimed to have been written by Shakespeare and tested to calculate their correlation between The New Atlantis (Bacon) and Hamlet (Shakespeare). From our calculations, for each individual comparison (and thus on average) all of Shakespeare's "supposed" works resulted in a smaller correlation when compared against Hamlet than when compared against The New Atlantis. This is evidence to support the claim that Shakespeare wrote his own plays.

12.a.) We tried to write as much code together in person as we could and we began with first discussing what was required of us so that we understood the problem the same way and there would be no confusion if we had to split up the work later. We then discussed our algorithm and pros/cons in terms of efficiency. Afterwards one of us would type up the code and we would debug together. We wrote AVL Tree, String Comparator, Correlator, TopKWords, and QuickSort together. The rest of the work was split up amongst the two of us. We found that it

was easier if one of us wrote an entire class or both of us wrote it together in each other's presence. Otherwise writing the code became very confusing and sometimes one of us had trouble interpreting the other's algorithm. We used GitHub to make sure we had the latest versions of the files and synced after every modification.

b.) Brian wrote all of the testing code because of his background in JUnit from 331 and I wrote the code for heapsort, MoveToFrontList, FourHeap, Hashtable, StringHasher, and TopKComparator. We wrote String Comparator, QuickSort, Correlator, TopKWords, and AVLTree together and debugged all the code together as well.

c.) One good thing about working together was having the added resource of someone else looking over your code. There were countless times where we were able to catch the other's bugs because we were looking at the code through a fresh pair of eyes. One bad thing about working together was trying to understand each other's algorithms for classes we didn't write.