Name: Sasha Babayan & Brian Park

Date: 5/13/2013

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 60 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 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(n2) 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.

10.

11.

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, 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, topKwords, MoveToFrontList, FourHeap, Hashtable, StringHasher, and TopKComparator. We wrote String Comparator, QuickSort, Correlator, 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.