CSE 332 Project 2 Write up

* Note: The last 3 questions require you to write code, collect data, and produce graphs of your results together with relatively long answers. Do not wait until the last minute to start this write up!

1. **Who is in your group?**
   Austin Briggs and Nick Evans.

2. **What assistance did you receive on this project? Include anyone or anything *except* your partner, the course staff, and the printed textbook.**
   The course website for casting with generics. Wikipedia.

3. **a) How long did the project take?** Part A was completed mostly within a week and then wrapped up by the time it was due. Part B took most all of the time between Part A's due date and its own due date.

   b) Which parts were most difficult? The amount of code that had to be written was difficult. Also the process of abstracting JUnit test methods to TestDataCounter and getting it to work for all subclasses of TestDataCounter took while.

   c) How could the project be better? It could be better if above and beyond projects were implemented.

4. **What "above and beyond" projects did you implement? What was interesting or difficult about them? Describe in detail how you implemented them.**
   None.

5. **a) How did you design your JUnit tests & what properties did you test?** Generally we tested method by method for each class. The tests were designed to assess as small of an area of the class under test as possible.

   b) What properties did you NOT test? We believe we tested everything that is relevant to the assignment.

   c) What boundary cases did you consider? Resizing, resizing multiple times, filling arrays, null iterators, fully iterated iterators, empty arrays, single-element arrays, multiple calls on the same method to see if it changed anything, deleting elements from an empty heap, and hash functions returning the correctly modded number from the input.

6. **a) Why does the iterator for Binary Search Tree need to use a stack data structure?**
   So that recursive calls to children nodes can be made and still be able to return to the parent nodes. Then go down the other branch.

b) If you were to write an iterator specifically for the AVL Tree, how could you guarantee that no resizing of the stack occurs after iteration has begun (which may require changing the interface for GStack)?

We make the stack length the height of the tree then since the AVL tree's height doesn't vary greatly it would remain unchanged. Since the iterator does an in order traversal by going down each of the branches the largest size that the stack needs to be is the height of the AVL Tree.

7. **If DataCounter's iterator returned elements in "most-frequent words first" order, you would not need to sort before printing. For each DataCounter (BST, AVL, MoveToFrontList, HashTable), explain how you would write such an iterator and what its big-O running time would be.**

BST: Scan the tree into an array then put into a max heap using Floyd's method which in total is O(n*log(n)).

AVL: Scan the tree into an array then put into a max heap using Floyd's method which in total is O(n*log(n)).

MoveToFrontList: Scan the list into an array then put into a max heap using Floyd's method which in total is O(n*log(n)).

HashTable: Scan the table into an array then put into a max heap using Floyd's method which in total is O(n*log(n)).

8. **For your Hashtable to be <u>CORRECT</u> (not necessarily *efficient*), what must be true about the arguments to the constructor?**

The hash function and comparator must not be null. The hash function must also return the same positive (Java returns a negative number when a modulus calculation is made on a negative number) integer value for an input every time the input is passed to the hash function.

9. **Conduct experiments to determine which DataCounter implementation (BST, AVL, MoveToFrontList, HashTable) & Sorting implementation (insertionSort, heapSort, OtherSort) is the fastest for large input texts.**

a) Describe your experimental setup: 1) Inputs used, 2) How you collected timing information, 3) Any details that would be needed to replicate your experiments.

1) Inputs used will be each of the data counter and sorting implementation**.** Hamlet.txt will be our text file and will never change as our base.

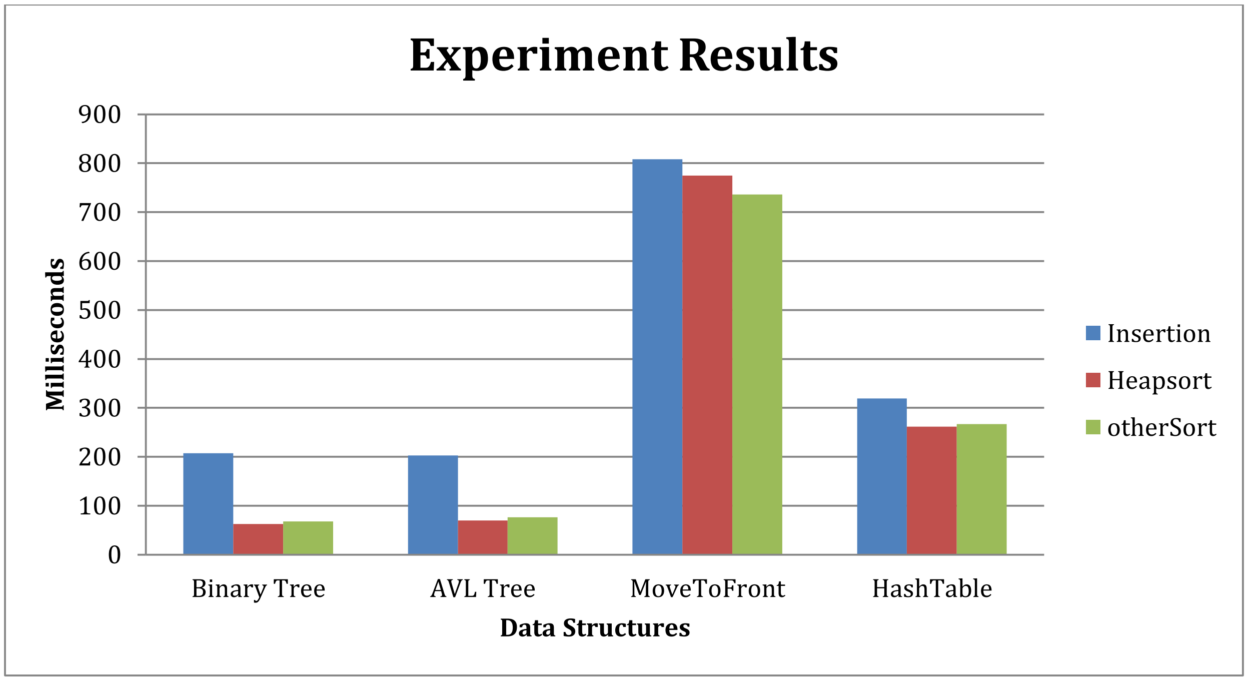
2) We used the provided getAverageRuntime method provided on the website then created an array that collects the returned values of the 12 string arguments.

3) Run each combination 50 times in a row using 5 as the warm up number and take the average time of the rest of the experiments.

b) Experimental Results (Your graph and table of results & Interpretation).

You need to conduct experiments for all possible combinations, 4 DataCounter X 3 Sorting algorithms = 12 experiments. Don't forget to give title and label axis for graphs and state which combination is the best. Does the result match your expectation? If not, why?

Experimental Results			
	Insertion	Heapsort	otherSort
Binary Tree	207.2	62.89	67.96
AVL Tree	203.04	69.73	76.76
Move To Front List	808.29	774.8	735.84
Hash Table	319.02	261.38	266.89



No the results do not meet our expectations. We had predicted that hash table would be significantly faster. The reason for our results is likely do to an exorbitant amount of collision despite are measures to prevent it most likely stemming from the double hash not adjusting in size.

c) Are there (perhaps contrived) texts that would produce a different answer, especially considering how MoveToFrontList works?
Hash Table could take longer by having specific inputs that would always hash to the same location.

A text file with only new words would cause MoveToFrontList to take O(N) for each insert as it would have to traverse the array each time.

A text file that was inserted in alphabetical order would cause insertion into a binary tree to behave as a link list and as such take O(N).

d) Does changing your hashing function affect your results?
(Provide graph/table & interpretation)
Conduct 6 experiments using HashTable (3 Sorting Algorithms X 2 Hashing functions = 6)
Does the result match your expectation? If not, why?

Original Function:

```
public class StringHasher implements Hasher<String> {
         int sum = 0;
```

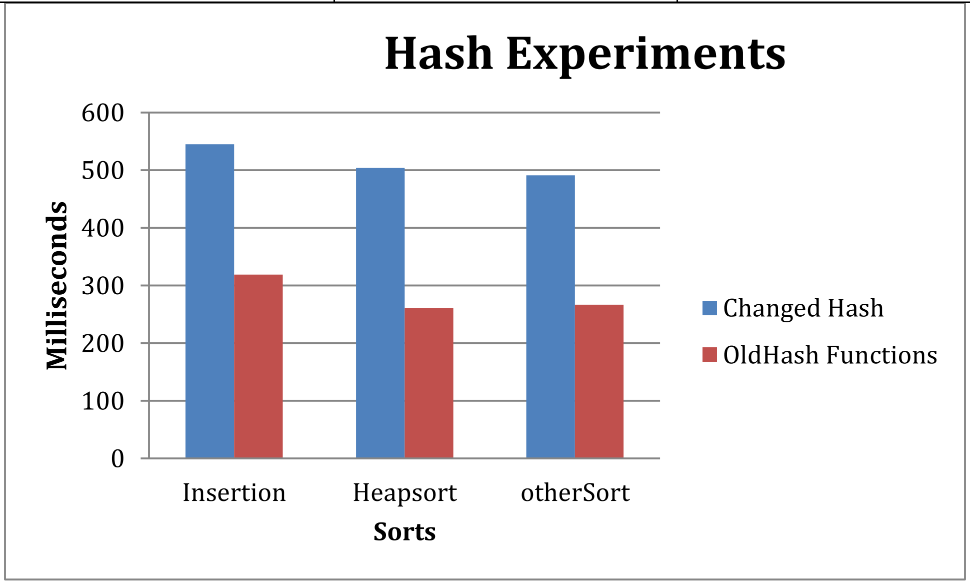
```
        for (int i = 0; i < s.length(); i++) {
                //Subtract 48 to get ASCII char value for 0 equal to 0
                int asciiVal = (int) s.charAt(i) - 48;
                //In case ASCII val is less than 48 (prevents a negative hash code)
                if (asciiVal < 0) asciiVal += 79;
                sum += asciiVal*i;
        }
        return sum;
```

Secondary Hash function: 17 – (sum mod 17)

New Hash function:
```
public class StringHasher implements Hasher<String> {
        int sum = 0;
        for (int i = 0; i < s.length(); i++) {
                //Subtract 48 to get ASCII char value for 0 equal to 0
                int asciiVal = (int) s.charAt(i) - 48;
                //In case ASCII val is less than 48 (prevents a negative hash code)
                if (asciiVal < 0) asciiVal += 79;
                sum += asciiVal*i;
        }
        return sum % 79;
```

Secondary Hash functions: 29 – (sum mod 29)

Hash Function Experiments			
	Insertion	Heapsort	otherSort
Changed Hash	545.044	503.8	491.3
OldHash Functions	319.02	261.38	266.89



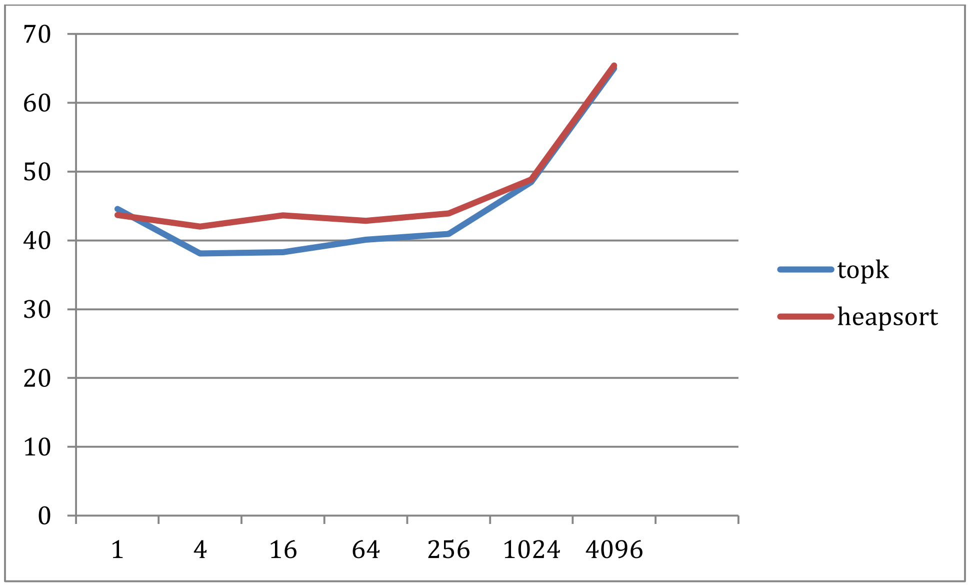
Yes this does match our predicted outcome which was that having the String Hasher mod its own output which limits the amount.

10. Conduct experiments to determine whether it is faster to use your *O(n log k)* approach to finding the top *k* most-frequent words or the simple *O(n log n)* approach (using the fastest sort you have available).

a) Produce a graph showing the time for the two approaches for various values of *k* (where *k* ranges from 1 to n).

If you measure runtime including the time it takes to print, you should print same number of words (i.e. print top-k words for both n long k and n log n algorithm) to account for time it takes to print. Be sure to give your interpretation of the result. Does the result match your expectation? If not, why?

TopKSort Experiment							
	1	4	16	64	256	1024	4096
topk	44.6	38.11	38.27	40.11	40.93	48.51	65
heapsort	43.71	42.02	43.64	42.87	43.93	48.87	65.42



b) How could you modify your implementation to take advantage of your experimental conclusion in a)?

That for values of k less than the n use topKSort otherwise the difference is negligible between the two sorting algorithms.

11. Using Correlator, does your experimentation suggest that Bacon wrote Shakespeare's plays? We do not need a fancy statistical analysis. This question is intended to be fun and simple. Give a 1-2 paragraph explanation.

If we assume that in a vacuum our use of Latent Semantic Indexing was the most appropriate choice for determining if Bacon wrote Hamlet then we must conclude by the result of 5.657273669233966E-4 that Bacon wrote Hamlet. This is due to the fact that the difference between the two is such a small number.

However our implementation ignores words that are not used in both file which is a severe weakness. This is because it might be possible that Shakespeare uses a word that Bacon never would. Therefore our results throw away key words that could provide us with key information on the authorship of Hamlet. Using Correlator we also discovered that “The Matrix” is 5.833310583051085E-4 to Hamlet and “The Lion King” is 4.409971909456395E-4 when compared to Hamlet which is even lower which calls into question our method of determining authorship. Unless of course we assume these are all from the same author.

12. <u>If you worked with a partner</u>:
a) Describe the process you used for developing and testing your code. If you divided it, describe that. If you did everything together, describe the actual process used (eg. how long you talked

about what, what order you wrote and tested, and how long it took).

On part A Nick wrote the front-end code and Austin wrote the testing code. On part B we both wrote front-end and testing code. During the project the two of us bounced ideas off of each other to work towards solutions for our parts and occasionally worked in the same room on our different parts.

b) Describe each group member's contributions/responsibilities in the project.

Nick wrote the front-end code (AVLTree, FourHeap, MoveToFrontList, WordCount.getCountsArray), heapSort, and TestAVLTree, and Austin wrote the testing code (TestFourHeap, TestMoveToFrontList, TestDataCounter), StringComparator, and updated WordCount.main on part A.

On part B, Nick wrote otherSort and topKSort, updated WordCount to accommodate for phase B's changes from phase A, and the tests for the sorts. Austin wrote Correlator, HashTable, StringHasher, and TestHashTable for part B. Both of us worked on the write up portion of the project.

c) Describe at least one good thing and one bad thing about the process of working together.

Some good things are working together is practice for our future professional careers where we will always be collaborating with others, we learned how to use GitHub, and we had practice in communication on deadlines and status reports.

One bad thing is it can be hard to understand a partner's code due to unfamiliarity with it.

Not getting feedback on phase A fast enough.

Appendix

Place anything that you want to add here.