**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.

1. **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.

1. **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.

1. **What "above and beyond" projects did you implement? What was interesting or difficult about them? Describe in detail how you implemented them.**

None.

1. **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?**

**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.

1. **a) Why does the iterator for Binary Search Tree need to use a stack data structure?**

**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)?**

1. **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:

AVL:

MoveToFrontList:

HashTable:

1. **For your Hashtable to be CORRECT (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 integer value for an input every time the input is passed to the hash function.

1. **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.**

**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?

**c) Are there (perhaps contrived) texts that would produce a different answer, especially**

**considering how MoveToFrontList works?**

**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?

1. **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?

**b) How could you modify your implementation to take advantage of your experimental**

**conclusion in a)?**

1. **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.**

1. **If you worked with a partner:**

**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 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.

**Appendix**

Place anything that you want to add here.