**CSE 332 Project 2 Write up**

\* Note: Three of the last 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!

* **Who is in your group (Give name & UW NetID of each person)?**

Reggie Jones 10270248

Tristan Riddell 1166096

* **What assistance did you receive on this project? Include anyone or anything *except* your partner, the course staff, and the printed textbook.**

-Stackoverflow.com / other forums that were brought up when 'google-ing' a question.

-Java docs

* **a) How long did the project take?**

**Reggie ~ 40 hours**

**Tristan ~ 40 hours**

**b) Which parts were most difficult?**

**Debugging. Getting it "almost" working and then having to spend hours trying to figure out what small detail you missed that's creating madness.**

**c) How could the project be better?**

* **(OPTIONAL) What "above and beyond" projects did you implement? What was interesting or difficult about them? Describe in detail how you implemented them.**
* **a) How did you design your JUnit tests & what properties did you test?**

**Designed the tests for each data structure accordingly to its specific features. This also lead to writing several private function abstractions to reduce duplicated code.**

**TestAVLTree-**

**I tested the count property, the height property, multiple cases of the rotation correctly updating the height, the tree was always balanced especially after rotations and inserts, the overallroot was being correctly updated when you would rotate about the root, and that incCount() would add a new node vs incrementing an existing node correctly.**

**TestFourHeap-**

**The most important function of a heap is DeleteMin. Efficiently rotating new elements to the root after each delete is the defining feature of a heap. I began similarly to movetofrontlist, ensuring external functions like size and count work. Then I tested adding one or multiple elements and ensuring that delete min was giving the expected values.**

**TestMoveToFrontList-**

**I began by testing simple external functions like size, and count –generally testing on empty, one element, then multiple. Then I made sure that the fundamental functionality of a move to front list was correct. I added random elements, followed by the first element I had added, and ensured that it both moved to the front and had its count incremented correctly.**

**TestHashTable-**

**Tested the 'external' features of size, counts of data, and the iterator. Such as making sure new buckets are not created when they already exist in table. Also tested 'internal' pieces of the code such as properly rehashing, that the hash function was giving decent distributivity. Tested the StringHasher is constantly hashing two values that are the same to the same hash.**

**b) What properties did you NOT test (“I tested everything” is NOT a valid answer since it is**

**impossible to test every property with every possible input)?**

**TestAVLTree-**

**I did not test the iterator as there was no implimentation of it in this class. I did not directly test the finding of a node as I considered testing add new node vs. increment count of node as an indirect test of**

**finding if a node already exists in the tree.**

**TestFourHeap-**

**I did not explicitly test findMin, because I knew that it merely peeks at the root. And that if deleteMin is working perfectly so is findMin. I also could not explicity test my percolate methods, as they are private. However, again I can assume that given that the other functionality is present, the percolates are working correctly.**

**TestMoveToFrontList-**

**I did not test the iterator. I also could not test the protectednode class within MoveToFrontList, but I could assume that if all other functionality was present, that the nodes were working correctly.**

**TestHashTable- Did not test the constant array of primes if the numbers were actually prime or not, just assuming they are from the given website in spec. Did not test the data structure for inputs of larger than 200,000 since there is a finite hardcoded array of prime numbers. Did not test HashBucket innerclass that are essentially linked lists.**

**c) What boundary cases did you consider?**

TestAVLTree-Empty tree, single root as the tree, and all of the 4 cases that cause

an inbalance in the tree.

TestFourHeap- empty heap, adding and removing one element,

TestMoveToFrontList- Empty list, adding the same element more than once in a row,

HashTable- Empty table, right before rehashing the table and right after rehashing the table.

* **a) The iterator for Binary Search Tree used a Stack as an internal data structure.**

**Why does the BST iterator need to use an internal data structure?**

**-It needs an internal data structure to store all of the data of the BST so that it can traverse the tree. If it were just a reference and hasNext was only checking if it wasn't null, then once the reference got all the way to the bottom of the left side, it would have no way to process anything more right of this on the tree.**

**In other words, it needs a data structure in order to "work its way back up" during the traversal.**

**b) If you were to write an iterator specifically for the AVL Tree, how could you guarantee that no**

**resizing of the Stack (No size increase of the internal array) occurs after iteration has begun**

**(which may require changing the interface of GStack)? Start by thinking about what would be**

**the smallest size of an array that guarantees no resizing.**

* **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 iterator and what its big-O running time would be.**

**BST- When building up the BST do comparisons on the count and let that determine where the nodes are places instead of comparing by the data. Have the iterator simply do an in-order traversal and use the ordering property of BST to our benefit. O(n) you traverse and inspect every node once.**

**AVL- Similarly to BST compare on count when building up tree, then just do an in-order traversal since AVL tree's also have the BST ordering property. O(n)**

**MoveToFront- The order of movetoFront is already defined by the fact that more recently referenced elements are moved to the front. This means that the iterator would need to find the highest element and return that one every time. In terms of order of operations, this would be O(n^2), however it would be slightly more efficient than this realistically because you can expect the bigger values to be closer to the front of the list.**

**HashTable- I don't think there would be any fantastic way of doing this. Essentially all you would be doing was moving the location of the sorting process into the iterator by creating an aux structure, adding them to the structure, then sorting. So O(nlogn) since you would just be using a sorting algorithm. The reason there is no elegant solution to this is because a HashTable has absolutely no ordering property assuming you have a decent hash function.**

* **For your HashTable to be CORRECT (not necessarily *efficient*), what must be true about the arguments to the constructor (Think about the relationship between the two arguments)?**

The Comparator generic type has to be a superclass or same type of the generic that Hasher uses.

* **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 (Place your graphs and tables of results here).**

You need to conduct experiments for all possible combinations, 4 DataCounters X 3 Sorting

algorithms = 12 experiments if you measured the runtime of DataCounter and Sorting together, or 4

DataCounters + 3 Sorting algorithms = 7 experiments if you measured DataCounter and Sorting

runtimes separately. Don’t forget to give a title and label the axes for all graphs. Make sure to choose

appropriate graphs to clearly show the important points of your data (i.e. How do the runtimes of the 3

Sorting algorithms compare to each other?)

**c) Interpretation of Experimental Results**

**1) What did you expect about the results and why?**

**2) Did your results agree with your expectations?**

**3) If the results did not match with your expectations, why do you think this happened?**

**4) According to your experiments, which DataCounter & Sorting Algorithm combo is the best?**

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

**considering how MoveToFrontList works?**

* **Conduct experiments to determine if changing the hash function affects the runtime of your HashTable.**

**a) Brief description of your hash functions**

**b) Experimental Results (Place your graphs and tables of results here).**

Experiment with at least 2 hash functions (3 Sorting Algorithms X 2 Hashing functions = 6 OR

2 Hashing functions = 2 experiments depending on how you measured the runtime)

Don’t forget to give each graph a title and label the axes.

**c) Interpretation (Your expectations and why? Did it match your results? If not, why?)**

* **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 the same number of words (i.e. print top-k words for both n log k and n log n algorithms) to account for time it takes to print. You don’t have to measure runtime for every possible value of k; you can use something like increments of 10 or 20. Don’t forget to give a title and label the axes for graphs. Make sure to choose \*appropriate graphs\* to clearly show the important point of your data (i.e. Some Bar graphs may not be ideal to show how runtime changes as k increases).

**b) Interpretation of Experimental Results**

**1) What did you expect about the results and why?**

**2) Did your results agree with your expectations?**

**3) If the results did not match with your expectations, why do you think this happened?**

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

**conclusion in b)?**

* **Using Correlator, does your experimentation suggest that Bacon wrote Shakespeare's plays?**

**Show at least one (you can experiment with more texts if you want) correlation value for each of:**

**a) Shakespeare's work compared to Shakespeare's work**

**b) Bacon's work compared to Bacon's work**

**c) Shakespeare's work compared to Bacon's work**

**According to the results of your experiments, did Bacon write Shakespeare's plays?**

* **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).**

**For phase A, we split the work pretty evenly by splitting step 2 and 3 in half, then both contributed to JUnit tests. For phase B we did similarly to phase A. We just started picking tasks off the spec and let the other one know what we were doing. We mostly communicated via text message and talked after class.**

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

**Reggie: StringComparator**

**WordCount.getCountsArray**

**AVLTree**

**Sorter.heapSort**

**processing of cmd line args**

**HashTable**

**StringHasher**

**Corresponding JUnit tests**

**Correlator**

**...**

**Tristan: MoveToFrontList**

**FourHeap**

**Corresponding JUnit tests**

**...**

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

I think one good thing about working together is seeing how the other person solved one of the particular problems and seeing that's not particularly how you would've done it so it gives you a second angle on the problem.

One bad thing about working together is ..[fill in].. not having our schedules align so most of our work was done remotely at time that was convenient for each individual instead of working together at the same time. Keeping consistent style throughout the project was also tough since everyone has considerably different style from each other.

**Appendix**

Place anything else that you want to add here.