**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 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 ~ 48+ hours

Tristan ~ 45

**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?**

Possibly less emphasis on experimentation. Though I see the value and role of the experiments. Overall fun project

* **(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-**

I tested simple external functions such as isEmpty and size, as well as trivial things like exceptions being caught.

Then I tested adding and removing one element from the heap, to ensure going back to an empty heap would not cause issues. Also rigorously tested deleteMin function, ensuring that the either the lowest value with the highest count was always returned, which in turn also tested the private helper functions 'percolate up' and 'percolate down'

**TestMoveToFrontList-**

I began by testing trivial functions such as size. Then I made sure that whenever a given word is accessed for any reason, it is moved to the front of the list. I also made sure that regardless of the order the elements are inputted into the list, the last element inputted is always at the front of the list. Also made sure any additional inserts on an element which already exists in the list would increment the count of that element.

**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 could not specifically test the private helper functions percolate up and percolate down, which makes up almost all the functionality of the class. However, if the functions like deleteMin and insert are working as expected, it is safe to assume that the percolate methods are working correctly as well.

**TestMoveToFrontList-**

I did not test the iterator, because there was no implementation in the class. I also could not test the protected node class used to implement the front list, but it is safe to assume if the Front list is implemented properly, the nodes are working as expected.

**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**- tested deleting with both 1 element and an empty heap, tested that count size held comparator superiority,tested finding min on an empty heap,

**TestMoveToFrontList**- not much edge case to consider, asserted an element was moved to front whenever it was touched, (inccount or getcount)

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-** In this case, the only option would be to traverse the entire linked list every time to select the highest remaining value. This would be O(n^2) time. However, there would be some amount of efficiency from the fact that the larger elements are more likely to be at 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 (all happening in the iterator instead of in seperate places). 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 type 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**

-the-new-atlantis ~17k total words

-hamlet ~34k total words

-same ~34k total words (all the same word)

-quest ~68K total words (collection of stories)

**2) How you collected timing information**

ran the combination of structures/sorts and timed each runtime individually (12 total) by creating arrays of Strings with the appropriate arguments to give main function in WordCount. To get the average of each runtime each combo was run 30 times and discarded the first 10 to avoid effects of JVM warm up.

**3) Any details that would be needed to replicate your experiments**

The quest file is a file I created by combining several stories to get an approximate word count that was double hamlet.txt.

**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?)

------AVL-----------

-------BST----------

-----HashTable--------

----MoveToFrontList----

**c) Interpretation of Experimental Results**

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

I expected a couple of things:

-I thought that MoveToFrontList would do the worst

-I thought that the HashTable would do the best

-I thought the same.txt file would always do the best

-I thought doubling the word count would very ROUGHLY double runtime

-I expected that the quest.txt file would NEVER be less than hamlet

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

Mostly. :

-It's apparent MoveToFrontList is the slowest.

-Hashtable wasn't necessarily the fastest in every category.

-I wasn't quite accurate when I thought same.txt would always be the fastest. It usually was, but the smaller file occassionally was faster. (sidenote-same.txt was almost always atleast twice as fast as hamlet.txt)

-Doubling the word count ROUGHLY doubled runtime except in a few exceptions like 2 combinations of BST and the MoveToFrontLists combos

-I was accurate as quest.txt was never less than hamlet

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

-Hashtable wasn't always the fastest because it did not depend on the input nearly as much as the other structure (no input would make it lightning fast like some specific inputs made other structures lightnight fast). It was much more constent.

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

-Binary Search Tree and Heapsort

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

**considering how MoveToFrontList works?**

Yes, the count of unique words REALLY matters in the case of MoveToFrontList. Also, if you had input that had a pattern of incrementing the count of an existing word that's at the end of the list, these traversals would be very costly.

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

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

-sum of the ascii values of the chars multiplied by 37 to the index of the char in the word.

-a really simple/bad hash function that sums the ascii values of each char in the word. An emphasis is put on the first letter by multiplying it by 10

**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.

--really basic hashing function--

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

-I expect the simple/bad hash function to do worse than the good one. The reason being that with large inputs implies a large array thats implementing the hash table. The range of these hash values is very small because there isn't that much variability in hash values compared to the size of the table. So only a small portion of the hashtable is getting used since they are all hashing to within the same range.

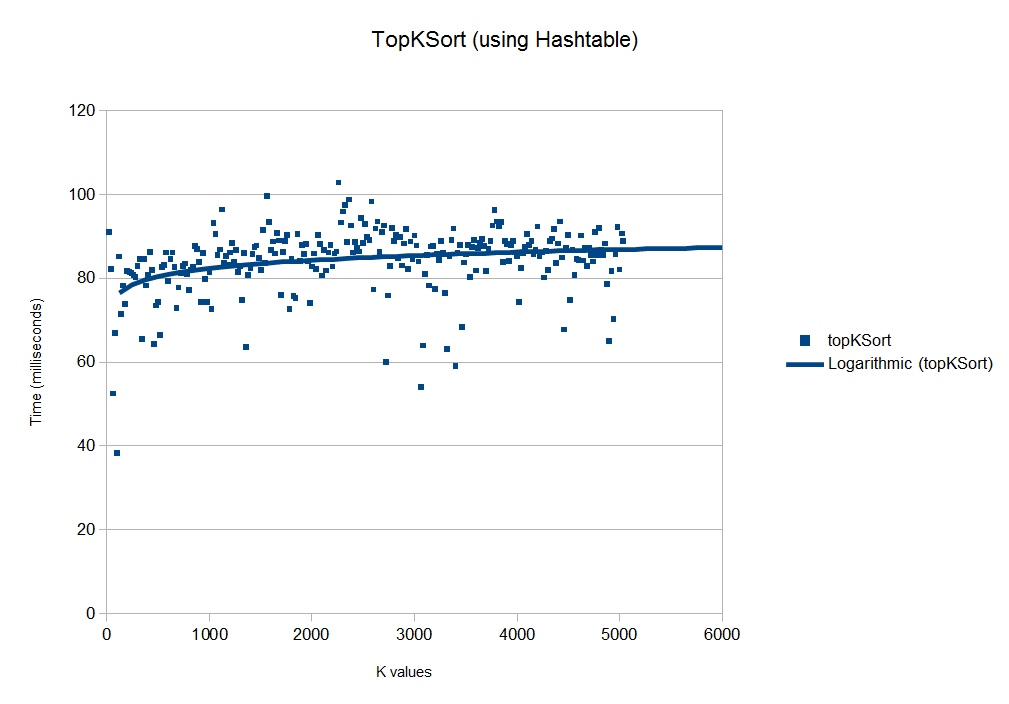
-No! It did not match my expectation.

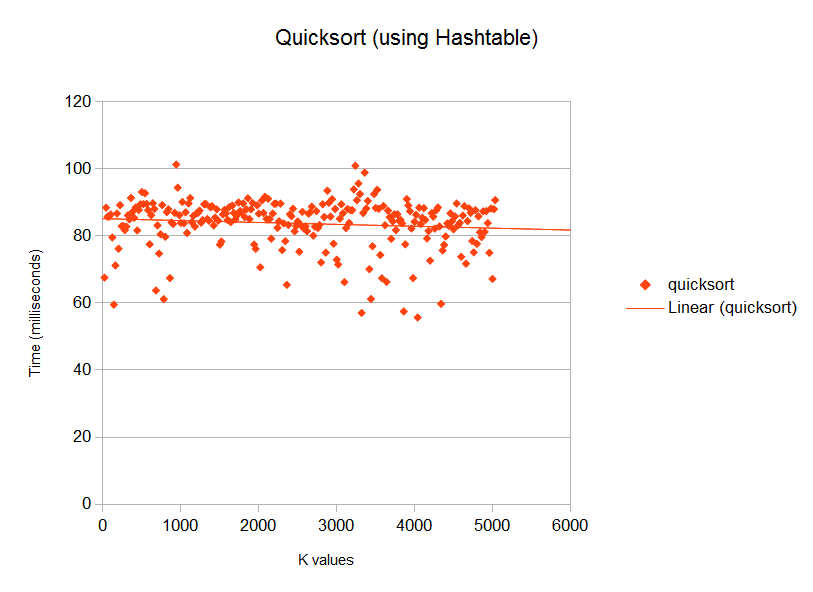
-The time 'saved' by only doing a very small amount of work in the bad hashing function made up for the time that I thought would be lost in the process of my initial explanation above.

* **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).**





**b) Interpretation of Experimental Results**

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

I expected to see topKSort be slightly quicker than quicksort because of the potential advantage n(log(k)) could have over n(log(n). I expected to see quicksort's completion time to remain constant as k increased, and expected to see topKSort's begin as more efficient than quicksort, then taper off and become less efficient as k increases. This is because quicksort is our fastest sort, so as K has less and less variance to n, you could expect quicksort to win out.

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

It is a little difficult to tell with the sample size, but it seem that what I expected was the outcome. The logorathmic line of best fit for the topKsort graph begins slightly below the beginning of the quicksort's linear line of best fit. But as K increases, quicksort begins to beat out topKsort in efficiency, simply because quicksort is a more effective sorting algorithm than the topKsort's heap algorithm.

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

While they did match, I expected to see a little bit more variance. I suspect if I had used less efficient sort than quicksort, and used a larger sample size of n, that I would have seen topKsort begin significantly below the other NlogN sort, and gradually meet the other sort in efficiency as k approaches n.

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

**conclusion in b)?**

According to my conclusion in b, I should simply use quicksort over topKsort if n is not too large, because it is faster to use quicksort over the entire data set than to use a heap and get nlogk time. However, as the size of n increases,topKsort would gain a greater advantage over quicksort, because the nlogK would become noticeably more efficient than quicksort's nlog(n). I would modify my implementation by having topKsort call quicksort if n is smaller than a certain size, or if k is a certain large percentage of n.

* **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**

2.6204557854975593E-4

Compared Romeo and Juliet to Hamlet

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

1.3796952709500807E-4

Compared The Essays Or Counsels, Civil And Moral, Of Francis Ld. to The Advancement of Learning

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

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

7.053996062610793E-4

compared Romeo and Juliet to The New Atlantis. According to the data, Bacon did NOT write Shakespeares play! Congratulations Shakespeare, you're not a fraud!!

* **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**

**HeapSort**

**processing of cmd line args**

**HashTable**

**StringHasher**

**Corresponding JUnit tests**

**Correlator**

**Datastructure/Sortingtype combos experimentation**

**Hashing experimentation**

**Correleator experimentation**

**Tristan: MoveToFrontList**

**FourHeap**

**Corresponding JUnit tests**

**TopKSort**

**TopKComparator**

**TopKPrint**

**OtherSort**

**Corresponding TestSorter tests**

**TopKSort Experimentation**

**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 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.