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?**
   The group consists of Owen Xia and Brian Griffith.

2. **What assistance did you receive on this project? Include anyone or anything *except* your partner, the course staff, and the printed textbook.**
   We did not receive any external assistance outside of code in the book.

3. **a) How long did the project take?**
   b) Which parts were most difficult?
   c) How could the project be better?

   a. Collectively, it took roughly ~30 hours.
   b. Correlator, AVLTree and FourHeap took quite a lot of time.
   c. Include more .txt files for our reading enjoyment.

4. **What "above and beyond" projects did you implement? What was interesting or difficult about them? Describe in detail how you implemented them.**
   We didn't do any of the “above and beyond” projects.

5. **a) How did you design your JUnit tests & what properties did you test?**
   b) What properties did you NOT test?
   c) What boundary cases did you consider?

   a. We thought about and tested inputs that would cause the data structure's properties to change, like rehashing or resizing array-based implementations. We tested our structures' “stability” by making sure the other function calls worked with the changed structures. We also designed our JUnit tests around more blackbox issues, like dealing with duplicates, large input and exceptions.

   b. For FourHeap, heap order was not tested.

   c. Again, we thought about boundary cases where the data structure's elements were moved in some way. For example, we tested the validity of the AVLTree after multiple inserts as well as whether an input hashes to the same index after rehashing.

6. **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`)?

a. Since we have to iterate through the tree recursively (which is bad for iterating only one element at a time), we have to use a stack to iterate through one node per call in O(n).

The LIFO property allows us to easily implement the traversal method since we don't have to keep track of an array index or anything to remind us of which index we're at in the traversal (postorder).

b. At any point, the maximum number of nodes in the stack is the height + 1. We just have to make sure that the stack is initialized to that value, so that the stack doesn't need to be resized.

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/AVL Trees can just be traversed in the stack-implemented post-order traversal in O(n) (like what was done in the BST class).

   For MoveToFrontList and HashTable, we could iterate through the list every time and return the next max value. However, that results in a O(n^2) runtime. Instead, every element can be added to an array and then use any O(nlogn) sorting algorithm on the array. This run time would be O(n + nlogn) or O(nlogn).

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

   The comparator must be able to effectively compare data objects if they're the same so that the count can be incremented instead of creating another DataCount object. The hasher must map the same data to the same integer as well.

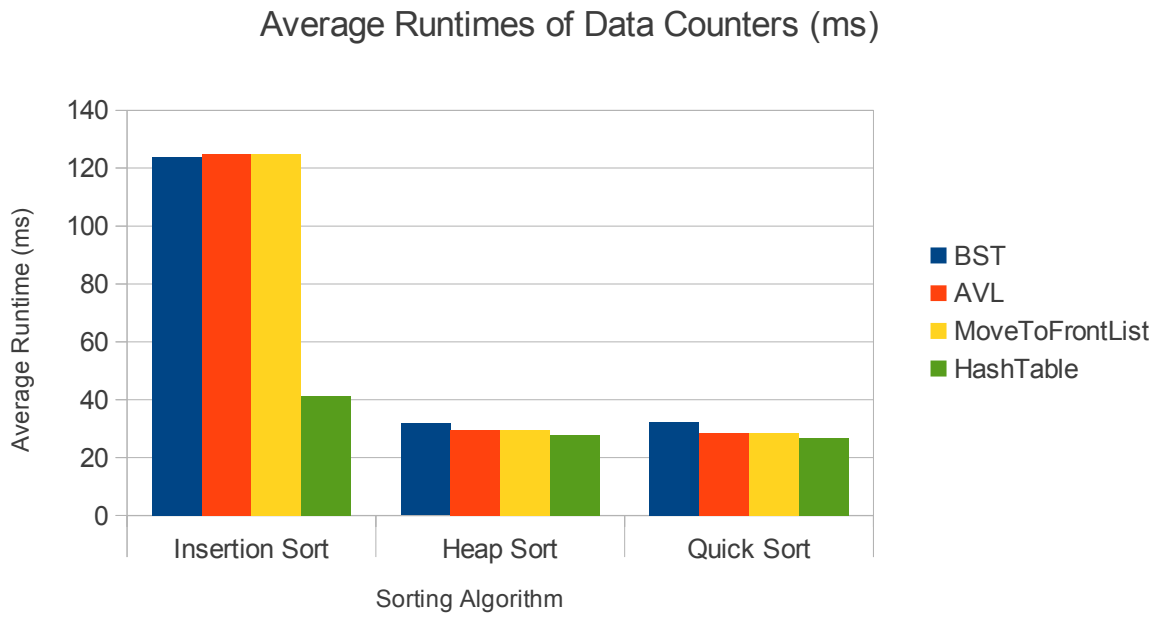
9. **Conduct experiments to determine which `DataCounter` implementation (BST, AVL, MoveToFrontList, HashTable) & Sorting implementation (insertionSort, heapSort, OtherSort) is the fastest for large input texts.**

   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.

   a. Our setup was running all twelve configurations on hamlet.txt using the provided getAverageRuntime() code. The average was collected out of 50 tests.
   b. getAverageRuntime() was used for timing.
   c. That should be everything

   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?

DataCounter	Sorting Algorithm	Average Runtime (ms)
BST	Insertion Sort	123.8695
BST	Heap Sort	31.7826
BST	Quick Sort	32.3478
AVL	Insertion Sort	124.9565
AVL	Heap Sort	29.5217
AVL	Quick Sort	28.6086
MoveToFrontList	Insertion Sort	563.1304
MoveToFrontList	Heap Sort	486.4347
MoveToFrontList	Quick Sort	444.0869
HashTable	Insertion Sort	41.3478
HashTable	Heap Sort	27.7391
HashTable	Quick Sort	26.6956



Quick-Sorted Hash Table is the best implementation.

AVL Tree and BST are similar in performance to HashTable but BST is not guaranteed to have a O(logn) runtime for insertions and AVLTree has to do many rotations per insert. Although HashTable has to rehash elements, it does it very infrequently since it at least doubles in size every time.

For sorting, quick sort is slightly faster than heap sort because quick sort accesses elements very close to one another (better spacial locality). As a result, there are fewer cache misses for quick sort although both are O(nlogn) runtime.

Just to mention, the AVL Tree and Binary Tree iterators work in a post-order traversal, which is the worst possible input for insertion sort.

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

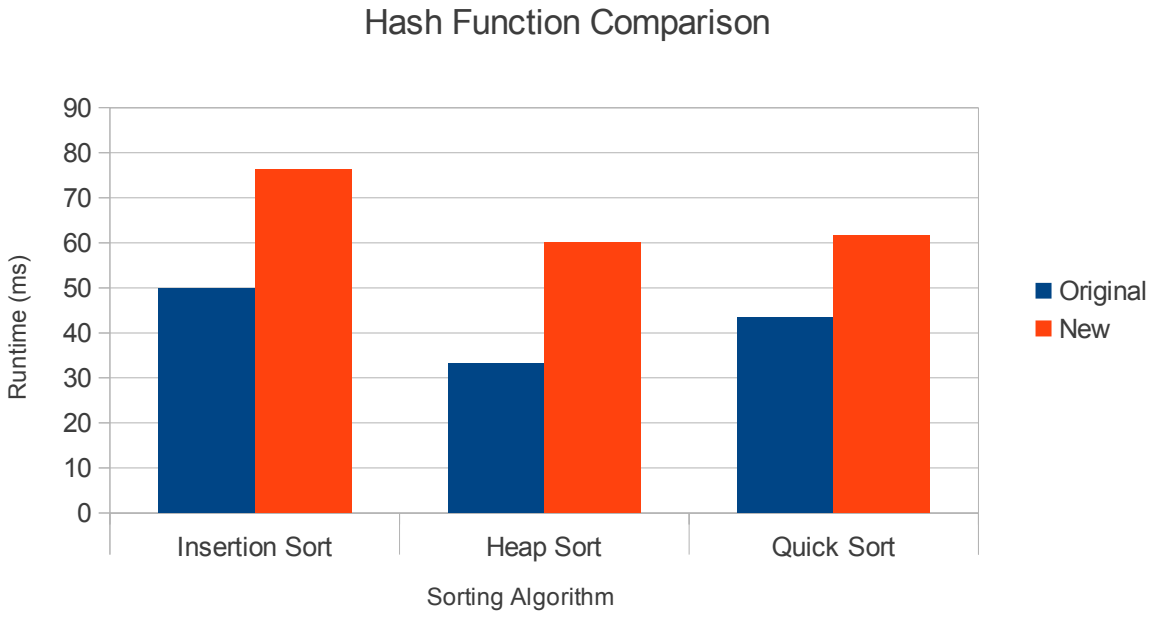
considering how MoveToFrontList works?

If the words in the text are in sorted order, BST would have a O(n) runtime for insertion and AVL Tree would have to do many more rotations. If there are more repeated words already at the front of the MTFList, it would only have a O(1) insertion instead of a O(n) insertion.

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?

Hash Function	Sorting Algorithm	Average Runtime (ms)
Original	Insertion Sort	49.8695
Original	Heap Sort	33.2173
Original	Quick Sort	43.3913
New	Insertion Sort	76.3478
New	Heap Sort	60.1739
New	Quick Sort	61.6521



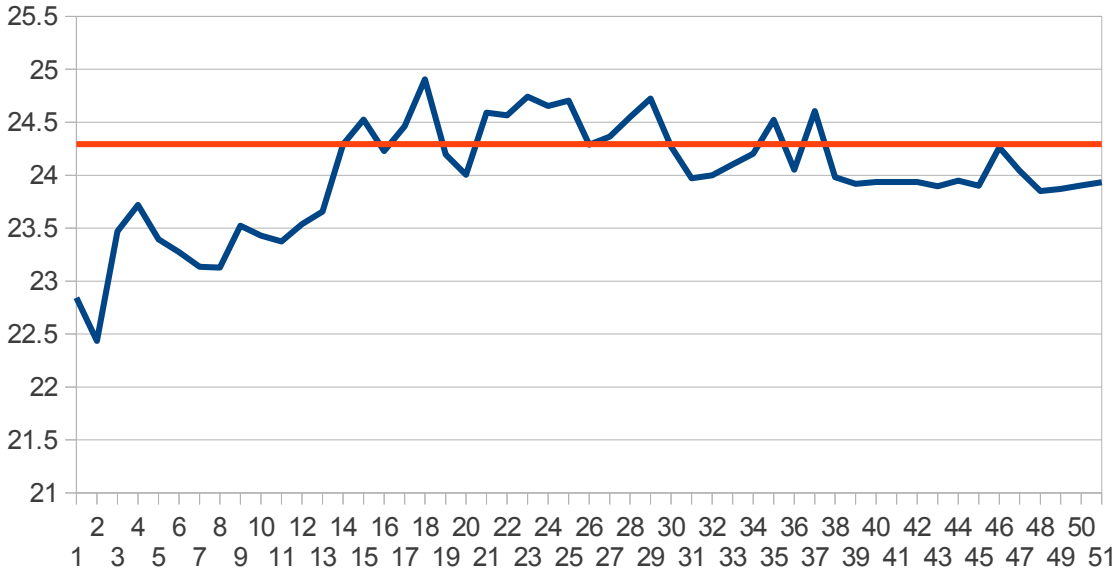
The original hashing function multiplied each char in the string by 37 and summed them all up. The second hash function only takes in the first character in the string. The second one only uses the first char value in the string. Since some characters are used more than others, each word starting with the same letter is hashed to the same index (how many words start with “a” as opposed to “x”?). As a result, there are much more collisions.

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?



This displays results of top k sort in 100 increments: e.g., top k sort for values of 100, 200, 300, etc. As we can see, top K sort is faster than our best (quick sort) full sorting algorithm for values of k less than about 1400. The reason this is not more pronounced is likely due to the constants factors; e.g., O(n log k) might actually be 3*log(k). Also, for values of n less than 5000 (words in hamlet.txt), the logorithmic term never grows very high. (This was performed with any printing)

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

Based on our experiments, we could change our implementation to simply use a quick sort for large values of k (e.,g., about 1400 here) and then print only the first k elements. However, in smaller k cases, our topK is the correct search algorithm.

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.

Variance Between Works

Text 1	Text 2	Variance
Hamlet	Othello	1.76E-4
Hamlet	Macbeth	1.89E-4
Hamlet	The Advancement of Learning	4.621E-4

Based on the above results, the variance between Shakespeare's works is much lower than the variance between Shakespeare and Bacon's works. Under the assumption that Bacon didn't have an alternate personality that anonymously authored plays in the same style, Shakespeare isn't a fake. In addition, Anne Hathaway (who was absolutely wonderful in Les Mis) married Shakespeare and Anne Hathaway doesn't marry fakes.

12. **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).

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

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

a. We divided our code because the abstract classes provided enough abstraction for us to work without knowing the exact implementations of the data structures. However, this meant that we were responsible for our own bugs and that meant we had to be more responsible for writing correct code. A lot of classes use other classes and oftentimes, we had to add functionality to classes we didn't work on.
b. Brian wrote MoveToFrontList/AVLTree/TopKSort/OtherSort and all of their tests. Owen implemented WordCount/Sorter/FourHeap/StringComparator/HashTable/Correlator and all their tests.
c. Good: It's more fun to work together and less work.
Bad: It's individually harder to work when we don't know exact implementations.

Appendix

Place anything that you want to add here.