**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!

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

Ruijia Wang 1360765

Mengyuan Huang 1360368

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

Data Structures and Algorithm Analysis in Java

Stackoverflow.com

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

Phase A: 2 days

Phase B: 3 days

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

Debug, test code

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

Add more data counter implementations and more sorting algorithm

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

1. **a) How did you design your JUnit tests & what properties did you test?**

**AVL tree**: test size, getCount, iterator, height, balance

**Move to front list**: test size, getCount, iterator, order

**Hash table**: test size, getCount, iterator, rehash, string hasher

**Sorter**: For each sorting routine, we tested null exception and regular array sorting. In addition to that, we also tested several different k input for topK sort.

**FourHeap:** we tested each method in the FourHeap class. For insert method, we tested four different cases for input: in order, duplicate, null and without order.

**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)?**

We did not test the data counter with large input, private method and internal structure.

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

**AVL tree:** test balance and height after four different cases of rotation, test empty tree

**Move to front list:** test the order after getCount and incCount, test empty root

**Hash table:** test the length and size of the table before and after rehash, and whether it rehash right before and after the size is greater than the length, test empty table

**FourHeap:** we tested null insertion and deletion of an empty heap.

1. **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?**

The BST iterator need to use an internal data structure because it need to traverse the BST. When it get the bottom left of the tree, it can process the right of the tree.

**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.**

Let the height of the tree to n, the max number of the nodes is 2^(n+1) - 1. Initialize the size of the array be 2^(n+1) - 1. Because there are at most 2^(n+1) - 1 nodes, that guarantees no resizing.

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

**MoveToFrontList:** We have to traverse the entire linked list to get the highest frequent value. Therefore, the runtime would be O(n^2).

**BST:** We can have the iterator do an in-order traversal through  the binary tree so that we will get all the nodes in ascending order. The runtime of this will be O(n).

**AVLTree:** We will have the iterator do an in-order traversal through the tree. Since the AVL tree has the property of BST, we can get all the nodes in the order we want. The runtime for this is O(n).

**HashTable:** We have to get the items from the table and sort them. Therefore, the runtime would be O(nlogn).

1. **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 hashing function must hash the same value for the same string.

The generic type of comparator have to been same or superclass that hasher uses.

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**

the-new-atlantis.txt

bacon.txt

king-lear.txt

hamlet.txt

**2) How you collected timing information**

We conduct the experiments by measuring DataCounter and Sorting runtimes separately. We first run each experiment 50 times to avoid JVM warmup. Then we will run each experiment 100 times to calculate the average runtime.

**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?)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DataCounter | BST | AVLTree | MTFL | HashTable |
| Time(ms) | 29.12 | 32.46 | 321.58 | 30.64 |

For BST, compare each sort algorithm:

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting | HeapSort | OtherSort | InsertionSort |
| Time(ms) | 2.88 | 1.34 | 63.7 |

**c) Interpretation of Experimental Results**

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

We expected that MTFL should do the worst.

We expected AVL tree would be slower than BST.

We expected running more times would give us more stable result.

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

The MTFL has the worst perfomance.

AVL tree is a little bit slower than BST.

Running more times does give us more stable result.

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

The result matches with our expectation.

**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, if we increase the counts of an existing word, especially at the end of the list, the traversal would be really slow for MTFL.

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

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

1. (in Phase B) hashValue = s.charAt(0) \* 31^(s.length - 1) + s.charAt(1)) \* 31^(s.length - 2) + s.charAt(2)) \* 31^(s.length - 3)+....+ s.charAt(s.length - 1)
2. hashValue = s.charAt(0) \* 127 \* s.length + s.charAt(1) \* 127 \* (s.length - 1) + s.charAt(2) \* 127 \* (s.length - 2) +....+s.charAt(s.length - 1) \* 127

**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.

hash function:            1     |     2

average time (ms): 30.34  |  30.62

hash function 1

average time for hash function 1 is: 30.34

hash function 2

average time for hash function 2 is: 30.62

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

expectation: there isn’t too much difference between those two hash functions, because both of the function run s.length times for a string. The running time should be the same.

Yes, it matches.

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 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).

X-axes: number of k (increments of 10)

Y-axes: times(ms)

**b) Interpretation of Experimental Results**

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

We expected to see that with the increase of k, the runtime for topK sorting will be greater. But the runtime for topK sort will still be less than merge sort of n elements. When k is greater, the runtime of topK sort is getting closer to runtime of merge sort.

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

Yes, the results agree with our expectation.

**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)?**

If n is not very big, we can use either merge sort or topK sort for finding the top k elements because the runtime nlogk is very close to nlogn when n is not very big. However, when n is being significantly great, topK sort is more efficient than merge sort since the difference between nlogn and nlogk is greater.

1. **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**

2.1294534043685116E-4 compare hamlet.txt with king-lear.txt

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

4.2919399932124345E-4

compare the-new-atlantis.txt and *the essays* (bacon.txt)

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

5.657273669233966E-4 compare hamlet.txt with the-new-atlantis.txt

6.873599481302955E-4 compare king-lear.txt with the essays (bacon.txt).

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

According to the data above, Bacon didn’t write Shakespeare’s plays.

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).**

One of us wrote data counter implementations and the other wrote sorting algorithm. For phase A, we spent one day for code and one day for test code. For phase B, we spent one day for code, one day for test code and one day for writeup.

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

Ruijia wrote java code and test code for data counter implementations.

Mengyuan wrote java code and test code for sorting algorithm.

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

One good thing is our process is very efficient, because different data counter implementations and different sorting algorithms are similar to each other.

One bad thing, we do not know too much each other’s work because our works do not relate too much.

**Appendix**

Place anything else that you want to add here.