Aaron Walker

Section 5

Improving Runtime of Project 1

When I first started thinking about this project it was pretty quickly apparent that one of the fastest ways to get through the inputted data would maybe be to set up a HashMap in the CollisionList class where the Key, Value pairs would be a string, representing a zip code, and then a ZipCode object as the value. Rather than continuously add Collision objects to their respective CollisionLists you would be able to just increment the values held by a ZipCode. Of course this would be too simple and ignores the whole point of the project, namely using data structures!

My next thought was to cut down on the read in and storing time of the program. Although adding to an ArrayList is highly efficient, having an amortized constant time, "adding n elements requires O(n) time". After reading some StackOverFlow answers it became clear that adding in amortized constant time is pretty good…but not the best. If you wish to add m items to an ArrayList the total time taken to add those m items to the array is O(m), and the amortized time, or the time per insertion is O(1). Can we do better than O(m)?

I have shown we can! Instead of using an ArrayList as the overarching container for a given zip's collisions we can use a LinkedList. A LinkedList has a guaranteed time complexity of O(1) when adding. More often than not though this nice performance level of the LinkedList's *add* is given up in favor of the ArrayList's ability to provide O(1) *access* through indices. The project specifications do not require further access to the actual Collision objects themselves once they have been read in and their data fields added to the ZipCodeList's object so we do not need to worry about access times. If for some reason access times became more important or were heavily used by some of the calculations, then it would not make sense to switch over to a LinkedList…as the requirements are though it makes perfect sense in order to cut down on the *add* time used by this program. Without any other changes to the program I was able to achieve slight improvements.

LinkedList instead of ArrayList in the ZipCodeList class (Over 100 Runs)	Original Implementation (Over 100 Runs)
AVG. Read & Store (Minus First 2 Runs): 1, 486, 853, 680 (nanoseconds)	AVG. Read & Store (Minus First 2 Runs): 1, 505, 695, 970 (nanoseconds)
AVG. Computation (Minus First 2 Runs): 44,502,400 (nanoseconds)	AVG. Computation (Minus First 2 Runs): 42,888,550 (nanoseconds)

This chart is interesting because even though the computational time is faster in the Original Implementation, despite having not changed anything dealing with the computational tasks, the LinkedList Implementation was able to perform better by 17,192,440 nanoseconds. Just looking at the Reading and Storing section of the program the LinkedList Implementation was able to outperform the original ArrayList by a whooping 18,842,290 nanoseconds! These times come from running the program on the 2014-2015 data file, but remain consistent throughout the other two files. After implementing a LinkedList instead of an ArrayList in the ZipCodeList class I was content to leave it as because no other structure was going to offer the same *add* time complexity and still retain the overall structure of the program.

Moving onto the CollisionList class I saw that the program was once again storing data within an ArrayList. Once again rather than use the amortized constant time which really boils down to being O(n) for adding, I decided to use a structure that allows O(1) adding, a HashMap. Like I suggested above I used the ZipCodeList's zip field as the Key and then used the actual ZipCodeList as the Value. Adding a Collision as a whole then becomes an O(1) operation because the program is essentially using the HashMap containsKey() method, which reduces to the HashMap get() method which runs in O(1) then using the HashMap put() method, which also runs in O(1) and then adds to a LinkedList, another O(1) operation. If the Key is not yet in the HashMap the addition is still O(1) because all that happens is a put into the HashMap O(1) and then the creation of a LinkedList in O(1). The decision to use a HashMap over a LinkedList was fairly simple, it just makes sense to have a Key to be able to access the rest of the data associated with a given ZipCodeList.

After working through all the places in which reading and adding take place it was time to take a shot at improving the computation time. My first thought also turned out to be the best implementation. I researched several different data structures and although a TreeSet seemed promising with its O(nlogn) sorting and adding after reading some more about them I figured that because I was going to eventually have to dig out the top or bottom k items a PriorityQueue would be the best structure. The tricky part here was being able to retrieve all the ties. Before implementing this algorithm I was able to get the following times for each of the three data sets.

My Implementation Using LinkedList/HashMap/PriorityQueue (100 Runs)	Original Implementation (100 Runs)
Smallest File: (100 Runs) AVG Read/Store: 1,402,818,210 AVG Computation: 46,274,370 AVG Total: 1,449,092,580	Smallest File: (100 Runs) AVG Read/Store: 1,504,300,600 AVG Computation: 44,420,660 AVG Total: 1,548,721,260
Middle File: (100 Runs) AVG Read/Store: 2,640,354,550 AVG Computation: 75,730,530 AVG Total: 2,716,085,080	Middle File: (100 Runs) AVG Read/Store: 2,956,196,590 AVG Computation: 80,813,900 AVG Total: 3,037,010,490
Largest File: (100 Runs) AVG Read/Store: 3,868,920,150 AVG Computation: 106,390,450 AVG Total: 3,975,310,600	Largest File: (100 Runs) AVG Read/Store: 4,410,464,750 AVG Computation: 113,611,910 AVG Total: 4,524,076,660

Overall I think I showed how using the three data structures, a LinkedList and HashMap for storing and a PriorityQueue for doing computational tasks how the program could be modified to run faster.