Kayli O'Keefe

Data Structures: Project 6

# Written Report: Performance Analysis

Before implementing alternate data structures, I computed the average execution time for both reading and storing data, and for computation of results from the existing code. I took an average time over thirty runs. These times were a basis for analyzing the time performance for alternate data structures.

**Existing Code: Performance Analysis (average over 30 executions)**

**Smallest File (all_2014-2015.csv)**

| | |
|---|---|
| *Reading and storing data:* | *1,446,639,853 nanoseconds* |
| *Computation of results:* | *56,690,879 nanoseconds* |
| *Average for total program completion:* | *1,503,330,732 nanoseconds* |

**Largest File (all_2012-2015.csv)**

| | |
|---|---|
| *Reading and storing data:* | *4,067,989,107 nanoseconds* |
| *Computation of results:* | *152,078,371 nanoseconds* |
| *Average for total program completion:* | *4,220,067,478 nanoseconds* |

The first alternate data structure that I chose to implement was a LinkedList. I simply replaced every occurrence of "ArrayList<String>" with "LinkedList<String>". I was expecting there to be only a slight, if any, improvement in time performance, as both ArrayList and LinkedList implement the List interface. There ended up being a slight change in performance. Reading and storing data was slightly faster than the ArrayList implementation, while the computation of results was (very) slightly slower. In considering the time performance for reading and storing data, I expected that LinkedList would have a faster execution time, because efficiency for adding to LinkedList is always O(1), whereas the average case for adding to ArrayList is O(n).

**LinkedList: Performance Analysis (average over 30 runs)**

*Smallest File (all_2014-2015.csv)*

| | |
|---|---|
| *Reading and storing data:* | *996,639,853 nanoseconds* |
| *Computation of results:* | *60,100,771 nanoseconds* |
| *Average for total program completion:* | *1,056,740,624 nanoseconds* |

These results were found using the smallest file (all_2014-2015.csv). I then ran the program with a larger file, thinking that perhaps the LinkedList implementation would be more noticeably time efficient with a larger file, but this was not the case.

**LinkedList: Performance Analysis (average over 30 runs)**

*Largest File (all_2012-2015.csv)*

| | |
|---|---|
| *Reading and storing data:* | *4,402,347,296 nanoseconds* |
| *Computation of results:* | *113,936,846 nanoseconds* |
| *Average for total program completion:* | *4,516,284,142 nanoseconds* |

The implementation that I found that worked the fastest was a combination of LinkedList and HashMap. Instead of storing collisions in an ArrayList, the CollisionList class stores the collisions in a HashMap, where the key is based on the hashCode method (given in the original code), and the value was based on the Collision associated with the key. In the methods for computing results, I used the Collections adAll() method to retrieve the values from the HashMap and store them in a LinkedList. At first, I repeated this step in every method that involved computing the results. I then realized that it would be more efficient to implement a method that created a new LinkedList<ZipCodeList> from the HashMap<Integer, ZipCodeList> and returned the new LinkedList, which could then call the Collections.sort method. While the average time for reading and storing data decreased significantly, the average time for computation of results was still slightly longer than the original ArrayList implementation. However, with a larger file, the computation of results was **significantly** faster than that of the ArrayList.

**LinkedList and HashMap: Performance Analysis (average over 30 runs)**

**Smallest File (all_2014-2015.csv)**

| | |
|---|---|
| *Reading and storing data:* | *788, 739,014 nanoseconds* |
| *Computation of results:* | *58,313,809 nanoseconds* |
| *Average for total program completion:* | *847,052,823 nanoseconds* |

**Largest File (all_2012-2015.csv)**

| | |
|---|---|
| *Reading and storing data:* | *2,440,758,765 nanoseconds* |
| *Computation of results:* | *106,775,346 nanoseconds* |
| *Average for total program completion:* | *2,547,534,111 nanoseconds* |

As is clearly evident, when dealing with larger files, the LinkedList/HashMap implementation is significantly more time efficient that the ArrayList implementation that was given in the original code, while the the pure LinkedList implementation did not improve performance.

Consider the Big-O efficiency for ArrayList, LinkedList, and HashMap. The table contains the average time performance.  Considering the O(1) performance for the HashMap methods, it is consistent that the best time performance included the HashMap implementation.

| Data Structure | Access | Search | Insertion |
|---|---|---|---|
| **ArrayList** | O(1) | O(n) | O(n) |
| **LinkedList** | O(n) | O(n) | O(1) |
| **HashMap** | O(1) | O(1) | O(1) |
| | | | |