CIT594 - Final Project Report

December 08, 2019

Additional Feature

The additional feature we implement in problem 6 is to show the top parking violation reason for the top 5 zip codes with least average livable area per capita.

Our feature uses all three data sets: 1) property file is used to get and calculate the sum of livable area in a certain zip code, 2) then, population file is used to calculate livable area per capita for a certain zip code, 3) parking file is used to count the top parking violation for a certain zip code. We use zip code as a unique identifier to link all three data sets.

To verify the accuracy and completeness of our additional feature functionality, we extracted limited sample data from the three data inputs (four files including two parking violation file types). The test data we used covers different edge cases, such as invalid data, very large data value, and very small data value. The result from our program matches our manual calculation. In addition, we carefully choose the optimized data structure to make sure we don't need to put unnecessary effort.

Data Structure analysis

1. **LinkedList:** Used to store data from input files.
    a. Parking Violations .json file is read in and stored in ArrayList: *parkingViolationsList* in class: *ParkingViolationsJSONReader*
    b. Parking Violations .csv file is read in and stored in ArrayList: *parkingViolationsList* in class: *ParkingViolationsCSVReader*

c. Property file is read in and stored in Linked List: *residenceList* in class: *PropertyReader*

We chose this data structure because we need to store data as a collection that is iterable. The reason why we choose LinkedList because there are more than 580k elements in property file as an example and linked list is dynamic in size. Also, the linked list is accessed by “for each ” loop instead of using indexing method, so the time complexity for traversing the data is O(n).

ArrayList is not preferred in this case because its sizing strategy is to double the size once it reaches the current size limit. The advantage of ArrayList is to access element by index. However, we don’t utilize this advantage. Though both collection framework has the same run time O(1) to add an element, and O(n) to traverse data, LinkedList is better in saving space.

2. **Array:** Used to store processed data.

Array is used to store information of a fixed number of residences. It is implemented with name *marketValuePerZIP* in class *ResidentialMarketValueCollector* and with name *livableAreaPerZIP* in class *ResidentialTLACollector*.

The size of the array is given before the value is assigned, so it is a good chance to use array with fixed size. Also, it stores primitive data type instead of Objects, and this saves space.

No alternative can behave better than array given the condition here.

3. **List:** Used as a general list type for interface class.

   a. *ParkingViolationsJSONReader* and *ParkingViolationsCSVReader* both implemented interface *ParkingViolationsReader*. In this interface, we used list as return data structure for function *getAllParkingViolation()*.

   b. *ResidentialMarketValueCollector* and *ResidentialTLACollector* both implemented interface ResidentialInformationCollector. In this interface, we used list as return data structure for function *getInformation()*.

We chose this data structure because we need to store data as a collection that is iterable. Since the child classes use ArrayList or LinkedList, it is natural to use List in the interface class.

4. **HashMap:** Used to store key-value data pair from input files.

    a. Population file is read in and stored in HashMap: *populationMap* in class: *PopulationReader*

    b. A HashMap data structure is used to store key-value pair: zip code - total parking fine as *parkingViolationFineMap* in class: *ParkingViolationsProcessor*

    c. A HashMap data structure is used to store key-value pair: zip code - violation reason as *violationReasonMap* in class: *ParkingViolationsProcessor*

    d. A HashMap data structure is used to store key-value pair: zip code - sum of total livable area as *residentialMap* in class: *ResidentialProcessor*

    We chose this data structure because we need to store data as a key value pair. Our alternatives include other collection framework in map - TreeMap, HashTable. The reason why we choose HashMap because the other two alternatives are not not more optimized in this case. We don't need to keep the data in sorted order by key, so we don't need to use TreeMap. We later used a helper function to sort the HashMap by value where we used LinkedHashMap. It is easy to swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if we were using Hashtable.

5. **TreeMap:** Used to store key-value data pair from input files and sorted in order by key

    a. A tree map data structure is used to store key-value pair: zip code - average parking fine per capita as *avgFineMap* in class: *ParkingViolationsProcessor*

    We chose this data structure because we need to store data as a key value pair and also keep this map in a sorted order by key. Treemap takes care of the sorting and there is no other sorted map in this collection.