594FinalProject

by Lu Liu and Kai Kleinbard
4/25/2020
UPENN MCIT

Dear TA:
Please note that our program on initialization, takes about 2 - 3 minutes to load. Then it works fast. During that time, it is reading the files, including the large properties csv file.

The Additional Feature:

We are finding the zipcode with the highest total parking fines (once we've added all the fines within the neighborhood. Then we use this data and find the same zip codes' total livable area per capita. We are curious how parking fines might relate to total livable area. We are using all three files: parking violations for the total fines, properties for the total livable area and population to find per capita. To ensure this data is correct, we ran a number of test cases using smaller data sets. Also, by using similar data to the required calculations of this project, this sata is extensively tested. Our parking fines reader automatically adds to a total within each zip code, so obtaining this total was already built into our code. We also double checked that our livable area per capita is correct by taking a ballpark of all zip codes per capita living area. The algorithm goes:

- Compare total fines for each zip code.
- Get the highest total.
- Find that zip codes' total livable area.
- Divide that by population.
- Add all the livable areas of each space and
- Divide that by the total population of that zipcode.

Three Data Structures:

HashMap: We used a hashmap in multiple places within the reader classes. We used a map, because we wanted to have a key, which was the zipcode, which was the key to a value. For example, for the property reader, we stored the zipcode as key, and the PropertyValue object as

value. We considered a number of other structures, such as an ArrayList and HashSet. The ArrayList we thought might be great if we created a more robust PropertyValue object that held zip codes. We also considered a HashSet, because it only stores one of each item -- since all zip codes are unique this seemed like a viable option. In the end, we chose HashMap since, like a HashSet, has only unique values, but we decided to keep zip codes separate from the PropertyValue, object, as it also allowed us to use HashMap for ParkingViolations and Population, using the same storage logic, with zipcode as key, and corresponding data as value. One disadvantage of the HashMap is it takes more overhead to store, however, this was okay with us in this case, as we were not concerned with memory. Additionally we used HashMaps for memoization, because it was easy to take a user input as key and use the value as the answer to return. A HashMap is O(1) for the `contains()` and `get()` method, making it quite efficient.

TreeMap: We used a treemap to store the parking violations per capita data. Since we were asked to store these based on ascending numerical order of zip codes. Since a TreeMap automatically sorts data, it was the best data structure for this case. We recognized that TreeMap has limitations and is slower for operations like `add()`, `remove()` and `contains()`. Another structure we considered here, HashMap, can do the same operations in O(1) (while TreeMap is O(log(N))]. However, we appreciated TreeMap's ability to sort items as they’re entered. We considered a HashSet here, since all the zipcodes are unique, however, it did not have the ability to sort automatically as a TreeMap does.

ArrayList: We used an ArrayList to store all the PropertyValues within a map entry. One zip code generally has many thousands of entries and we wanted a quick way to index and get property values if necessary. Since ArrayLists, unlike arrays, automatically resize, this was a good option as we did not know how many values were in each zip code. In addition, we wanted our program to be expandable in case we ever used a different city or file, etc. ArrayLists give us a quick way to index and get values. We considered an array here, but we would have to know the full size. We also considered a HashSet, but we realize that many properties have the same total livable area and market value, and HashSets don't allow duplicate values. Thus ArrayList was the best option for us.

Array: We used arrays in multiple areas, especially in readers to quickly split strings read from files by commas, spaces and quotations

(using the `String.split()` method). Arrays offered us a quick and relatively easy way to index and store the cells of our CSV. Direct indexing takes O(1). On the other hand searching slower at O(N), but since we were splitting each line as we were reading it, our ability to get the relevant data by index was key here (we did not have to sort or search for anything, we just had to get to a certain index). We considered an ArrayList here, which would have been sufficient, but the convenient `String.split()` method in Java makes Arrays relatively useful in this case. We also considered a StringBuilder, where we search character by character, adding into our StringBuilder, however, this proved cumbersome and required more code for the StringBuilder to constantly be turned into a String and then refreshed/cleared.