CS126 WAFFLES Coursework Report [2014446]

CustomerStore

Overview

* I have used an ArrayList structure to store and process customers because it was easy to implement. Also the Customer Store is the smallest out of the three stores, and after running it with an Arraylist it was clear that, more data structure complexity it’s exaggeration. Also by using more advance data structures-like trees means increased Space Complexity. So we better leave trees for more complex operations
* I used MergeSort to sort customers by name and ID. It has time complexity O(n\*Logn) and its performance is very stable.

Space Complexity

| **Store** | **Worst Case** | **Description** |
| --- | --- | --- |
| CustomerStore | O(n) | I have used a two ArrayList. One to store customers and one to store the Blacklisted IDs when customers are added. Where n is total customers added. |

Time Complexity

| **Method** | **Average Case** | **Description** |
| --- | --- | --- |
| addCustomer(Customer c) | O(1) | Array add is constant time. Check customer valid(1).Search if customer id is in Blacklist(b-size of blacklist) |
| addCustomer(Customer[] c) | O(n) | Check if the customer’s Id exists in the Arraylist already(n) |
| getCustomer(Long id) | O(n) | Linear search n is total customers in the store |
| getCustomers() | O(nLogn) | MergeSort(nlongn) n is total customers in the store |
| getCustomers(Customer[] c) | O(nLogn) | MergeSort n is the length of the input array |
| getCustomersByName() | O(nLogn) | MergeSort n is total customers in the store |
| getCustomersByName(Customer[] c) | - | - |
| getCustomersContaining(String s) | O(n) | It searches all customers(n). Uses an AVLTree to add the found elements for fast sorting |

FavouriteStore

Overview

* I have used AVLTree data structures extensively in this Store. There are many benefites of this dataStructure. It has time complexity O(Logn) for both inserting, searching and deleting an element. Even though it is quite difficult to construct, it provides with huge flexibility and expansion potential
* I have used AVLTree  with added modifications to store favourites. Moreover the AVLTreeF extends a standard AVLTree and adds the ability to create new trees in tree’s nodes. The only disadvantage of this is that it takes up more space but it is sure worth it for the incredible performance we get
* I used AVLTree to sort in mostly all the methods that require sorting

Space Complexity

| **Store** | **Worst Case** | **Description** |
| --- | --- | --- |
| FavouriteStore | O(n2) | I have used AVLTreeF. Which basically it’s a tree that can have multiple dimensions(In this coursework I capped its dimension to 2) |
|  |  |  |

Time Complexity

| **Method** | **Average Case** | **Description** |
| --- | --- | --- |
| addFavourite(Favourite f) | O(Logn) | Checking if Restaurant is valid(1), then checking if FavouriteID is included in the tree with the blacklistIDs(1). Then it searches for the same FavouriteID in the tree of Restaurants and deletes it if found.(Logn).Also if found it will be deleted from the lookuptree too and be added to blacklist tree. IF not found in the FavouritetTree then proceed to see if its ID matches included in the Lookup tree then add it to that nodes’ subtree in the FavTree(logn). Else add it normally to the Tree |
| addFavourite(Favourite[] f) | O(nLogn) | Description Attemps to add n elements in the Favourites, which means executing n times the addFavourite(Favourite f)(Logn) |
| getFavourite(Long id) | O(Logn) | Search an element in the Favourite Tree(Logn) |
| getFavourites() | O(nLogn) | Description  Making a tree to List operation(n).Then the list to array(n).Uses mergeSort(nLogn) to sort the array in the end |
| getFavourites(Favourite[] f) | O(...) | Description |
| getFavouritesByCustomerID(Long id) | O(nLogn) | Description Find operations in tree(Logn.).Iteration of array(n). Merged sort is used |
| getFavouritesByRestaurantID(Long id) | O(nLogn) | Description Find operations in tree(Logn.).Iteration of array(n). Merged sort is used |
| getCommonFavouriteRestaurants(   Long id1, Long id2) | O(nLogn) | Description It calls getFavouritesByCustomerID(Long id) 2 times (n). Performs 2 array iteratons and calls sortedRestIds which uses mergeSort |
| getMissingFavouriteRestaurants(   Long id1, Long id2) | O(nlogn) | Description Iterates through arrays. Calls other methods which are-(nLogn). Uses mergeSort which is nLogn |
| getNotCommonFavouriteRestaurants(   Long id1, Long id2) | O(nLogn) | Description Iterates through arrays. Calls other methods which are-(nLogn). Uses mergeSort which is nLogn |
| getTopCustomersByFavouriteCount() | O(n) | Description Has to check all Customers.Adding elements to a tree(logn). Making a tree to a list(n). |
| getTopRestaurantsByFavouriteCount() | O(n) | Description Has to check all Restaurants.Adding elements to a tree(logn). Making a tree to a list(n). |

RestaurantStore

Overview

* I have used ArrayList to store restaurants . It works efficiently. Keeping in mind that when in testing the Website it loads
* Also a reason I chose ArrayList is because it requires sorting data in many different ways each. Therefore instead of making deferent trees each time, I just convert the arraylist to an array and sort it
* I used Mergesort to sort the list when needed

Space Complexity

| **Store** | **Worst Case** | **Description** |
| --- | --- | --- |
| RestaurantStore | O(n) | I have used ArrayList. |

Time Complexity

| **Method** | **Average Case** | **Description** |
| --- | --- | --- |
| addRestaurant(Restaurant r) | O(n) | Description Extract the trueId of the Restaaurant.Check the ID is valid.Check if the restautant is Valid(1).Check if id is blacklisted(b-size of blacklist). Check if Restaurant Id already exists in the list(n). |
| addRestaurant(Restaurant[] r) | O(n) | Description  Performes the addRestaurant ID b times( where b is the size of the Restaurant array) |
| getRestaurant(Long id) | O(n) | Description Iterates through the List to find the the Restaurant with the same Id. |
| getRestaurants() | O(nLogn) | Description  Checking if the List of stored Restaurants is empty(1). Copy the List into the array(n). Sorting the array of Restaurant by Restaurant Id, with mergeSort(nLogn). |
| getRestaurants(Restaurant[] r) | O(nLogn) | Description Checking if the Restaurant Array is null(1).Sorting an array of Restaurant by Restaurant Id, using MergeSort(nLogn) |
| getRestaurantsByName() | O(nLogn) | Description Checking if the List of stored Restaurants is empty(1). Copy the List into the array(n). Sorting the array of Restaurant by Restaurant Name, with mergeSort(nLogn). |
| getRestaurantsByDateEstablished() | O(nLogn) | Description Checking if the List of stored Restaurants is empty(1). Copy the List into the array(n). Sorting the array of Restaurant by Date Established, with mergeSort(nLogn). |
| getRestaurantsByDateEstablished(   Restaurant[] r) | O(nlogn) | Description Checking if the Array of stored Restaurants is empty(1).Sorting the array of Restaurant by Date Established, with mergeSort(nLogn). |
| getRestaurantsByWarwickStars() | O(nlogn) | Description Iterating through the Restaurant List and adding each Restaurant in an array of Arraylist according to the number of stars it has(n) .Calls geRestaurantByName() to sort the 3 arraylists (nLogn) .Compines 3 arrays into one array(n). |
| getRestaurantsByRating(Restaurant[] r) | O(nlogn) | Description Checking if the Array of stored Restaurants is empty(1).Sorting the array of Restaurant by Rating, with mergeSort(nLogn). |
| getRestaurantsByDistanceFrom(   float lat, float lon) | O(nLogn) | Description Checks if list is null(1). Creates an array with the newly calculated RestaurantDistances(n). Sorts the array with MergeSort(nLogn) |
| getRestaurantsByDistanceFrom(   Restaurant[] r, float lat, float lon) | O(nLogn) | Description Checks if array null(1). Creates an array with the newly calculated(1) RestaurantDistances(n). Sorts the array with MergeSort(nLogn) |
| getRestaurantsContaining(String s) | O(n) | Description It Iterates through an arraylist of all Restaurants. And It sorts by adding to an AVL tree(Logn) |

Util

Overview

* **ConvertToPlace**
  + Constant operations. Therefore not many things that could make it faster
* **DataChecker**
  + Constant operations. Therefore not many things that could make it faster
* **HaversineDistanceCalculator (HaversineDC)**
  + Constant operations. Therefore not many things that could make it faster
* **KeywordChecker**
  + ...
* **StringFormatter**
  + Very fast performance when using the Hashmap

Space Complexity

| **Util** | **Worst Case** | **Description** |
| --- | --- | --- |
| ConvertToPlace | O(n) | A HashMap to store all of the places |
| DataChecker | O(n) | Uses arrays. |
| HaversineDC | O(1) | Constant operations |
| KeywordChecker | O(...) |  |
| StringFormatter | O(n) | A HashMap to store all of the charachters |

Time Complexity

Generally Util methods are expected to have O(1) time complexity. That is because these kind of methods are added within the main methods to provide some kind of checking.

| **Util** | **Method** | **Average Case** | **Description** |
| --- | --- | --- | --- |
| ConvertToPlace | convert(float lat, float lon) | O(1) | It’s just searching for an element in the already created array. |
| DataChecker | extractTrueID(String[] repeatedID) | O(1) | Constant 3 Loops to check the 3 Strings per call. It can be considered a constant operation |
| DataChecker | isValid(Long id) | O(1) | Constant number of operations. It’s just checking if a Long data type is not 0L |
| DataChecker | isValid(Customer customer) | O(1) | Constant number of operations. It’s just checking if the fields of the Customer class |
| DataChecker | isValid(Favourite favourite) | O(1) | Constant number of operations. It’s just checking if the fields of the Favourite class |
| DataChecker | isValid(Restaurant restaurant) | O(1) | Constant number of operations. It’s just checking if the fields of the Restaurant data type |
| DataChecker | isValid(Review review) | O(1) | Constant number of operations. It’s just checking if the fields of the Review class |
| HaversineDC | inKilometres(   float lat1, float lon1,   float lat2, float lon2) | O(1) | Constant number of mathematical operations |
| HaversineDC | inMiles(   float lat1, float lon1,   float lat2, float lon2) | O(1) | Constant number of mathematical operations |
| StringFormatter | convertAccentsFaster(String s) | O(n) | Iteration to get all String parameter’s letters(n). Matches it’s letter with the hashmap(1) |