CS126 WAFFLES Coursework Report [2104534]

CustomerStore

Overview

- I have used an `ArrayList` structure to store and process customers because it was easy to implement and allowed a data structure to have a flexible size.
- I used `QuickSort` to sort customers by name and ID because I want my code to be efficient. I did not use mergeSort as it has high space complexity (O(n)) and quickSort is an in-place sorting algorithm (O(logn), requiring no additional space. On average, the quick sort takes O(nlogn) time. Furthermore, quicksort is much preferred for large, unsorted arrays, of which there exist many in this coursework, so it was an efficient choice to use Quicksort.
- I used a Hashmap to store the Blacklisted IDs so that there is instant lookup (O(1)) when searching for blacklisted IDs.
- I have also used individual methods for the sorting algorithm as it would take the same amount of time as a Comparable interface, with the same time complexity and space complexity. It was a design decision so that it is easier for a programmer to understand for maintenance of code.

Space Complexity

Store	**Worst Case**	**Description**
CustomerStore	O(n)	I have used a `ArrayList` to store customers. I have used a Hashmap to store IDs(blacklisted) Where `n` is total customers added.

Time Complexity

Method	Average Case	Description
addCustomer(Customer c)	O(1)	Array add is constant time
addCustomer(Customer[] c)	O(n)	Add all customers in linear time as the above method is called for each customer element in array. where `n` is the length of the input array
getCustomer(Long id)	O(n)	Linear search through array taking linear time. `n` is total customers in the store
getCustomers()	O(nlogn)	Quicksort, since no extra space is needed, and in real world applications, most programs find that quicksort perform much better than other method. `n` is total customers in the store
getCustomers(Customer[] c)	O(nlogn)	Quicksort, as above `n` is the length of the input array
getCustomersByName()	O(nlogn)	Quicksort, as above `n` is total customers in the store
getCustomersByName(Customer[] c)	O(nlogn)	Quicksort, as above `n` is total customers in the store
getCustomersContaining(String s)	O(1 + n*(1 + 1)) = O(n)	Searches all customers O(1) is the average time it takes to convert accents `n` is total customers

FavouriteStore

Overview

- I have initialised all the data structures/classes needed at the top of the file for ease of reading and would require less running time as no need to initialise during specific methods.
- I have used `arraylist` to store favourites as it allows for a flexible size
- I have used `hashmap` to store both blacklisted IDs and the backlog
- I used `quicksort` to sort all arrays in the store, as it takes O(nlogn) time on average.
- I did not use trees to sort as that would increase the time complexity.
- To get common/missing/notcommon restaurants, I used a hashmap to firstly store all of customer 1's restaurants and then compared it with customer 2's restaurants and if they were common/missing/notcommon depending on the method they would be appended to an arraylist. I use an arraylist as an intermediary because the size of the final array is unknown until we collect all data, so it is beneficial to use arraylist's flexibility.
- To get favourites by customer/restaurant ID I used a sorted arraylist, as that will place all the favourites in the desired location without manually changing the array each time the method is called.
- To get top customers/restaurants by favourite count, I make use of a Hashmap to store whether any favourite of a certain customer has been encountered before, i.e. the hashmap is used to check if that customer has had a favourite during the linear search of all favourites.
- If there is no customer in the hashmap, that means a new occurrence must be made, so I add the customerID and details as a concatenated string to an arraylist, because of its flexible size. Then I add the customerID as the key to the hashmap, and the value is the index that the customer was added to in the arraylist. This is helpful because when a customer has been encountered before, I extract the details from the arraylist, increment the count and the date favourited. This method allows me to efficiently create the data required, since both the arraylist and hash map will have O(1) lookup time(since we know the arraylist index). Then we simply sort the data by favourite amount and date if needed, using a quicksort algorithm, taking O(nlogn).

Space Complexity

Store	Worst Case	Description
FavouriteStore	O(n)	I have used `arraylist` to store the favourites I have used hashmaps to store blacklisted ID's and backlog of favourites, as it allows for instant O(1) lookup access time. I have used arraylists as an intermediary where the resulting data set does not have a determinate size.

Time Complexity

Method	Average Case	Description
addFavourite(Favourite f)	O(n logn)	The time complexity is nlogn because of the time it takes to remove a favourite and add a new one in its replacement. It will take this time to sort the favourites for the oldest favourite. However, the best time could be O(1) since no favourites would be needed to sort, so it is entirely dependent on the test data, and therefore the average case depends on if favourites need to be added again. So the range of average case is not clearly defined, as it depends on the data.
addFavourite(Favourite[] f)	O(n^2 logn)	Description Since we have to loop through the array where n is the the size of the array.

Method	Average Case	Description
getFavourite(Long id)	O(n)	Description Linear time for iterating through the arraylist, with size n.
getFavourites()	O(nlogn)	Description Quicksort method takes nlogn time on average
getFavouritesByCustomerID(Long id)	O(n)	Description Using a sorted arraylist to add favourites but linear traversal takes n time
getFavouritesByRestaurantID(Long id)	O(n)	Description Using a sorted arraylist to add favourites but linear traversal takes n time
getCommonFavouriteRestaurants(Long id1, Long id2)	O(nlogn)	Description Quicksort method takes nlogn time on average
getMissingFavouriteRestaurants(Long id1, Long id2)	O(nlogn)	Description Quicksort method takes nlogn time on average
getNotCommonFavouriteRestaurants(Long id1, Long id2)	O(nlogn)	Description Quicksort method takes nlogn time on average
getTopCustomersByFavouriteCount()	O(nlogn)	Description Takes O(n) to traverse and create data, but O(nlogn) to sort.

Method	Average Case	Description
getTopRestaurantsByFavouriteCount()	O(nlogn)	Description Takes O(n) to traverse and create data, but O(nlogn) to sort.

RestaurantStore

Overview

- I have used an `arraylist` to store restaurants array
- I also used arraylist as an intermediate when there would be a varying number of restaurants, e.g. in the restaurants matching the query searchTerm.

- I used `quicksort` to sort all arrays in the store, as it takes O(nlogn) time on average.
- I did not use trees to sort as that would increase the time complexity.
- I have initialised all the data structures/classes needed at the top of the file for ease of reading and would require less running time as no need to initialise during specific methods.

Space Complexity

Store	Worst Case	Description
RestaurantStore	O(n)	I have used `arraylist` to store restaurnts I used `hashmap` to store blacklisted ID's. Where `n` is number of restaurants

Time Complexity

Method	Average Case	Description
addRestaurant(Restaurant r)	O(1)	Description `Adding to array is constant time`

Method	Average Case	Description
addRestaurant(Restaurant[] r)	O(n)	Description Runs n times to call the add, where n is the arraysize
getRestaurant(Long id)	O(n)	Description Linear time for iterating through the arraylist, with size n.
getRestaurants()	O(nlogn)	Description `Quicksort` takes nlogn time on average
getRestaurants(Restaurant[] r)	O(nlogn)	Description `Quicksort` takes nlogn time on average
getRestaurantsByName()	O(nlogn)	Description `Quicksort` takes nlogn time on average
getRestaurantsByDateEstablished()	O(nlogn)	Description `Quicksort` takes nlogn time on average
getRestaurantsByDateEstablished(Restaurant[] r)	O(nlogn)	Description `Quicksort` takes nlogn time on average
getRestaurantsByWarwickStars()	O(nlogn)	Description `Quicksort` takes nlogn time on average. Takes linear time to create arraylist of restaurants with at least one star.

Method	Average Case	Description
getRestaurantsByRating(Restaurant[] r)	O(nlogn)	Description `Quicksort` takes nlogn time on average
getRestaurantsByDistanceFrom(float lat, float lon)	O(nlogn)	Description `Quicksort` takes nlogn time on average
getRestaurantsByDistanceFrom(Restaurant[] r, float lat, float lon)	O(nlogn)	Description `Quicksort` takes nlogn time on average
getRestaurantsContaining(String s)	O(nlogn)	Description `Quicksort` takes nlogn time on average. Takes linear time to find relevant restaurants But this is overwritten by the sorting algorithm

Util

Overview

- **ConvertToPlace**
  - Initialise array in the constructor, so that the data is not transferred to array every time.
  - Use a hashmap to store the concatenated string coordinates as a key, and the location in the array as value.
  - This allows for O(1) lookup
- **DataChecker**
  - ...
- **HaversineDistanceCalculator (HaversineDC)**
  - ...
- **KeywordChecker**
  - ...

- **StringFormatter**
    - ...

Space Complexity

Util	**Worst Case**	**Description**
ConvertToPlace	O(n)	Stores values in hashmap, allows for instant access.
DataChecker	O(1)	No data stored
HaversineDC	O(1)	Only use one variable to store a constant.
KeywordChecker	O(...)	...
StringFormatter	O(1)	Uses hashmap to store accents and split input term into its characters and loop through the character array to search and replace accents.

Time Complexity

Util	**Method**	**Average Case**	**Description**
ConvertToPlace	convert(float lat, float lon)	O(n) OR O(1) after initialisation	Takes O(n) time for the default initialisation, but to find the place only takes O(1) time.
DataChecker	extractTrueID(String[] repeatedID)	O(1)	Up to 3 if statements can be called, each taking constant time.
DataChecker	isValid(Long id)	O(1)	For loop runs up till 16 characters input maximum

Util	Method	Average Case	Description
DataChecker	isValid(Customer customer)	O(1)	Two if statements, not based off input size, so constant time.
DataChecker	isValid(Favourite favourite)	O(1)	Two if statements, not based off input size, so constant time.
DataChecker	isValid(Restaurant restaurant)	O(1)	Multiple if statements, but no dependency on input for time.
HaversineDC	inKilometres(float lat1, float lon1, float lat2, float lon2)	O(1)	Performs calculation and returns value
HaversineDC	inMiles(float lat1, float lon1, float lat2, float lon2)	O(1)	Performs calculation and returns value
KeywordChecker	isAKeyword(String s)	O(...)	...
StringFormatter	convertAccentsFaster(String s)	O(n)	Using hashmap allows for O(1) lookup time, but in order to read the characters in each string, where n is the number of characters.