Batuhan Kesikbaş-40573251614

Cansu Gürel-16157401836

Betülnaz HAYRAN-28354853660

**BIM213 - Data Structures and Algorithms**

Report-hw2

**What is the purpose of this project?**

Purpose of this project is to design a Spotify-like environment.

**How did we solve it?**

First, we created a class named Main for our operations and a class named LinkedList for linked list implementation. We created 4 linked list from LinkedList class. One for names without replication -<<name>> linked list-, one for all names with replication -<<allNames>> linked list-, one for without replication -<<song>> linked list- and one for all songs with replication -<<allSongs>> linked list- .

Main Class:

We use while(true) loop to make our operation scanner work as long as we want. We used scanner and switch case for to choose our operation.

In switch case we had “C” case for Create, “S” for set, “E” for Erase, “L” for List, “N” for to list all name of registered people, “M” for to list all the songs that liked by anyone and “R” for Recommend.

We used a default case for wrong inputs and in that case the system closes.

We also made the inputs case insensitive and used toLowerCase() and toUpperCase() methods to do so.

In case “C”, firstly we assign a string to input that will come from scanner which will be person’s name. We want to provide the name replication, so we add the inputs to <<name>> linked list if only it wasn’t already in it.

In case “S”, first we assign a string to input that will come from scanner which will be person’s name. We used scan.next() because person’s name will be the first word that will come after the operation character-“S”-. Then we assign a string to input that will come from scanner which will be song’s name. We used scan.nextLine() because song will be the words that will come after the person’s name. Then we add the person’s name to <<allNames>> linked list and song’s name to <<allSongs>> linked list. Also, to prevent replication, we only add person’s name to <<name>> linked list and song’s name to <<song>> linked list if only they weren’t already in the lists.

In case “E”, firstly we assign a string to input that will come from scanner which will be person’s name. Then, we assign a string to input that will come from scanner which will be song’s name. Then, we are iterating for the name and the song. When we found the name and the song in <<allNames>> linked list and <<allSongs>> linked list, we remove them from both linked lists by using deleteAtPosition() method.

In case “L”, firstly we assign a string to input that will come from scanner which will be person’s name. Then, we are iterating for the name that is written next to “L” operation character. When we found the name in <<allNames>> linked list we print the songs that is in <<allSongs>> linked list. We do this because the names index and the songs index are the same in their linked lists.

In case “N”, we print the <<name>> linked list which contains the person’s names without replication.

In case “M”, we print the <<song>> linked list which contains the song’s names without replication.

In case “R”, we store frequency of songs in a map. Because we want to know how many times a song liked. Then, we create a tree set with map entry by overriding compare() method in order to compare frequencies. After that, we use pollFirst() method to print 3 recommended songs.

LinkedList Class:

Firstly, we created a class Node and then we created methods for Main Class.

insert() – we add element to the list.

printList() – we printed the list.

deleteAtPosition () – we deleted node at given position(index).

findSize() – we get the size of the list.

contains() – we checked if element is already is in the list or not.

get()- we get the element in wanted position.

**Which environment did we used?**

We used JetBrains- IntelliJ environment for our project.

**Analysis of the time complexity of each data structure and function:**

**add() –** supports O(1) constant-time insertion at any position.

**get() –** searching for an element takes O(n) time.

**remove() –** removing an element also takes O(1) operation, as we provide the position of the element.

If you want to delete a specific element, the time complexity is O(n) (where n is the number of elements) because you have to find the element first.

**indexOf() -** takes O(n) time.

**size( )-** It returns the number of elements in the list. Its time complexity is O(1).

Storing and retrieving elements from the **HashMap takes constant O(1)** time.

**TreeSet has O(log(n))** time complexity.