CS1C Fall 2022 Assignment 6
Possible Approach

Overview: In this assignment you will create two hash tables, using quadratic probing, based on the songs in the MillionSongDataSubset. One table will be hashed based on the **song's Title** and the other table will be hashed based on the **song's Artist**. You will then be asked to find certain items in the hash tables based on a provided list of song Titles or a list of Artists. The bulk of the assignment is in creating 4 classes, (1)FHhashQPwFind.java which extends FHhashQP by providing a new generic element for a KeyType, (2)TableGenerator.java that generates the array of Artist names and the two hash tables, (3)SongCompTitle.java and (4)SongCompArtist.java that are wrapper classes for SongEntry objects. Take a quick look at MyTunes.java to understand what it is doing and read through the assignment description as it is quite helpful. There are several places in the Assignment description in Canvas where you are asked to respond, in your README.txt file, to certain questions. Don't miss these.

Start by cloning your repository. FHhashQP.java and HashEntry.java should already be in your repository. You'll need to make one small change to FHhashQP.java to keep track of the number of probes performed. This is explained in the Assignment description in Canvas.

FHhashQPwFind:
Next, create the new Java class FHhashQPwFind.java in the hashTable package (use the IntelliJ shortcut) and replace the class signature with the signature provided in the project description. Next, add the constructor and the shells for the 3 methods that are mentioned, public E find(KeyType key), private/protected int myHashKey(KeyType key) and private/protected int findPosKey(KeyType key).

Since you are extending FHhashQP, you need to call its constructor in the FHhashQPwFind.java constructor, using super(). FHhashQP has two constructors, one that takes a parameter for table size and one that uses the default. You will be using the one that requires a table size, it can't be smaller than the default. Your constructor for FHhashQPwFind.java should take a parameter and you would call super() inserting your parameter in the (). Look at FHhashQP and make sure you understand what it is doing. It does a lot of the work.

The method find(), utilizes the findPosKey() method by passing it the key and then checking if the returned index location of the array is active and if it is, returns the data of that index location, and if it's not active, throws an exception. Take a look at contains() in FHhashQP as a starter. Its one line returns a boolean, we want to check if the portion after return, is true and if it is, return the data portion. If it's not, throw the exception. Don't forget to change the name of the method called.

The method myHashKey() determines the hash value for the key passed in based on its KeyType, either a Title or an Artist. It is called by findPosKey(). It's a simple method that looks an awful lot like myHash() in FHhashQP. It just needs a minor tweak to the parameter.

The findPosKey() method finds the index location of an object based on the value of KeyType using quadratic probing. It calls myHashKey() to help in this. It is called by find() and its implementation looks an awful lot like findPos() in FHhashQP. It too needs a minor tweak to its parameter and a method call. You will want to use the compareTo() method rather than the equals() method which is not overridden. See below.

So now we can head back over to MyTunes.java to see what else we need to do. Let's start by dealing with the two classes MyTunes is looking for, SongCompTitle and SongCompArtist. Have IntelliJ create the shells in the hashTables package. You might also need to import cs1c.SongEntry.

Both classes need to implement Comparable, so update the class definitions to include that with the appropriate Types in the <> for Comparable. IntelliJ will require you to add the compareTo() method so go ahead and do that. Next, add the other methods that are required in the project description, equals(), hashCode() and toString(). Check the assignment description in Canvas for any guidance here. Define the equals() method as described in the module "Find() in Hash Tables" such that the equals() method of the wrapper class preserves the equals() provided by the embedded data.

The basic form of the wrapper classes are spelled out for you in the module, "Find() in Hash Tables", in Week 07 Part 02 in Canvas. Pretty straight forward. IntelliJ can help create the shells for these methods once you start typing. You should add the attribute(s) and constructor as mentioned in my next paragraphs before finalizing these methods. IntelliJ should automatically insert @overrride in the line above the overridden methods, but if it doesn't you should add it as described in the Assignments Additional Requirements.

SongCompTitle:
SongCompTitle is very straight forward and has a single SongEntry object as an attribute, its data. You also need to add a constructor (there is just one attribute) in addition to the other 4 methods you added above. You will have a separate instance of SongCompTitle for each song. You can now complete those methods. In the equals() method you are equating the data of a SongCompTitle object with the data of a SongCompTitle object passed in. For hashCode(), you need to return the hashcode of the Title for the SongEntry object. toString() should output all of the elements of a SongEntry object (which is the data attribute) and the SongEntry class has a toString() method so this one is easy.

SongCompArtist:
SongsCompArtist is a bit more complicated. It serves two purposes, it compares songs based on their Artist, but this class also is used to determine the number of songs for each Artist. This class has two attributes, a String for the name of the Artist and an ArrayList<SongEntry> for all the songs for that Artist. The ArrayList is the data for this class. You need to add a constructor in addition to the other 4 methods. The constructor takes two parameters, one for each instance variable. You will end up with an instance of this class for each Artist, each instance will have the name of the Artist and an ArrayList of all the songs for that Artist. You can now complete those 4 methods that you added. In the equals() method you are equating the data of a SongCompArtist object with the data of a SongCompArtist object passed in. For hashCode(), you need to return the hashCode() of the Artist name. For toString(), just output the Artist name and the number of songs in its array. Look at the sample output in the assignment description.

You have one more method to add to SongCompArtist, public/protected void addSong(SongEntry someVaraible). This method will be called by the last class we need to create, TableGenerator. The addSong() method's purpose is to add a song to an existing SongCompArtist's ArrayList< SongEntry >, first checking to make sure its Artist name is the same as the Artist name of the requesting

SongCompArtist object. You can do this using the String class equals() method or the compareTo() method. Probably want to print out an error message if their Artists are not the same.

TableGenerator:
Back we go to MyTunes. There are still a couple of red underlines. There is one more class, TableGenerator.java, to create and several methods as well. This class generates hash tables for the two instances of the class FHhashQPwFind, one (SongCompTitle) that uses a String as the key and a class that includes a String key, and another (SongCompArtist) that also uses a String as the key and a class that includes a String key. It also generates a String ArrayList of Artist names. Have IntelliJ generate the TableGenerator class in the hashTables package and its 3 methods. You can return null for now for the first 2 methods. You can add the attribute - private ArrayList<String> artistNames - and return it for the getArtists() method. Woohoo! No more red. Just empty methods.

You also need to add to TableGenerator, as attributes, the two FHhashQPwFind objects, tableOfSongTitles and tableOfArtists specifying the appropriate <parameters>, check the return types of two populate methods (populateTitleTable() and populateArtistTable()) for a hint. Your constructor takes one parameter, an initial table size and will initialize the three attributes. In the constructor create a new FHhashQPwFind object for tableOfSongTitles and pass in the initial table size. Create a new FHhashQPwFind object for tableOfArtists and pass in the initial table size. Create a new ArrayList<> for the artistNames ArrayList<> that you defined as a class attribute.

The populateTitleTable() method is fairly straight forward. Loop through each song creating a new SongCompTitle object for each song and then calling insert() (FHhashQPwFind inherited it from FHhashQP) on tableOfSongTitles and passing in the object you created. Return the table at the end.

The populateArtistTable() method is a bit more complicated. I think there are two good approaches that have a lot in common. The main difference is when you insert the SongCompArtist objects into the hash table. The result is that you will have a unique SongCompArtist object for each Artist, which will hold their name and an ArrayList of songs. These SongCompArtist objects will then need to be hashed into the hash table based on the Artist name.

In my preferred approach I first created an ArrayList<SongCompArtist> as a temporary container for each SongCompArtist object while I processed the songs. I then looped through each song in the list of songs, got its Artist and checked if its Artist was in the list of artistNames, if it wasn't, I added the Artist to the artistNames ArrayList and then created the ArrayList of SongEntry objects for that Artist and added the song to it, and then created a new SongCompArtist object to hold the name of the Artist and their ArrayList of songs that I just created, and added that SongCompArtist object to my temporary container. If the Artist name already existed, I would get the appropriate SongCompArtist object from my temporary container (based on the Artist's index in the artistNames array) and then add that song to the ArrayList of that SongCompArtist object, calling the addSong() method of that class. Once all the songs were processed, I looped through the ArrayList < SongCompArtist > holding the SongCompArtist objects and hashed the objects into the hash table calling the insert() method for each SongCompArtist object. That's a mouthfull!

An alternative approach is to avoid creating the temporary container and immediately inserting the new SongCompArtist object into the hash table when you create it. Then when you encounter a song

whose Artist has already been found, you need to find their SongCompArtist object in the hash table and add the song to that object's ArrayList. This means you will have to call find() on the hash table for almost every song in the SongEntry array. However, I tested the execution time of both approaches, and they seem to be about the same, so take your pick.

One last thought. How much collision is going on in the Title hash table and in the Artist hash table? How unique are each of the respective hash values? Also, how big does the hash table for the Artist objects need to be, how about for the Title objects?