A Social-Network Based Recommendation System for last.fm

In this assignment, you are asked to design and implement a social network-based recommender system for last.fm. You'll learn and practice skills and gain knowledge in the following areas:

- Understanding and implementing a custom HashMap using separate chaining
- Using data structures like vectors, lists, and unordered sets to store and manage data
- Reading data from external files and parsing it to populate data structures.
- Creating and using classes like **User** and **Artist** to model real-world entities.
- Using C++ features such as templates for generic data structures (e.g., HashMap), operator overloading (e.g., **operator[]**), and streams for input/output.
- Developing algorithms for listing friends of a user, finding common friends between two users, and generating music artist recommendations based on user behavior and preferences.
- Formatting and displaying output messages to the console for user-friendly interaction.
- Solving complex problems related to data processing, recommendation systems, and data management.
- Understanding how data structures and algorithms are used in real-world applications, such as recommender systems used by platforms like music streaming services.

Overall, this implementation offers practical experience in building a basic recommender system, managing data, and applying various programming concepts in a real-world context. It is also a valuable addition to your resume!

Dataset:

You are given the following dataset (Reference: http://www.lastfm.com). *data.zip* file contains social networking, tagging, and music artist listening information from a set of 2K users from Last.fm online music system. http://www.last.fm

- There are 1892 users and 17632 artists
- There are 12717 user-friend relations
- There are 92834 user-listened artist relations [user, artist]

Files:

- artists.dat: This file contains information about music artists
  File format: `id \t name`
- user_artists.dat: This file contains the artists listened by each user. It also provides a listening count for each [user, artist] pair.
  File format: `userID \t artistID`
- user_friends.dat: These files contain the friend relations between users in the database.
  File format: `userID \t friendID`

Requirements

HashMap.h

HashMap is a template class that provides a basic hash table implementation for mapping keys to values. It supports insertion, retrieval, removal, and other operations on key-value pairs.

1. *Template Parameters*
   - **K** (Key Type): The type of keys stored in the hash table.
   - **V** (Value Type): The type of values associated with the keys.
   - **TableSize** (Table Size): The size of the hash table, which determines the number of buckets for storing key-value pairs
2. *Data Members*
   - **std::vector<std::list<HashEntry<K, V>>> table:** The main data structure used to store key-value pairs. It is a vector of linked lists (buckets), where each bucket holds key-value pairs with the same hash value.
3. *Constructor*
   - HashMap(): *Initializes the hash table with the specified size.*
4. *Public Member Functions*
   - **void insert(const K& key, const V& value):** Inserts a key-value pair into the hash table.
   - **bool get(const K& key, V& value) const:** retrieves the value associated with a given key from the hash table. Returns true if key is found, false otherwise.
   - **V& operator[](const K& key):** Subscript operator overloading to access and modify elements in the hash table.
   - **void remove(const K& key):** Removes a key-value pair from the hash table based on the key.
   - **bool contains(const K& key) const:** Checks if the hash table contains a specific key. Returns true if key is found, false otherwise
   - **std::vector<std::pair<K, V>> sortHashMapByValues():** Sorts the key-value pairs in the hash table by values in descending order and returns them as a vector of pairs.

Sample Test for HashMap implementation:

```
#include <iostream>
#include "HashMap.h"

int main(){
    // Create a HashMap with integer keys and string values
    HashMap<int, std::string, 100> myHashMap;

    // Insert key-value pairs
    myHashMap.insert(1, "Alice");
    myHashMap.insert(2, "Bob");

    // Retrieve a value by key
    std::string name;
    if (myHashMap.get(1, name)) {
        std::cout << "Name: " << name << std::endl;
    }

    // Check if a key exists
    if (myHashMap.contains(2)) {
        std::cout << "Key 2 exists." << std::endl;
    }

    // Remove a key-value pair
    myHashMap.remove(1);

    // Access elements using the subscript operator
    myHashMap[3] = "Charlie";

    // Sort the key-value pairs by values
    std::vector<std::pair<int, std::string>> sortedEntries =
                                myHashMap.sortHashMapByValues();

    // Print the contents of the vector
    for (const auto& entry : sortedEntries) {
    std::cout << "Key: " << entry.first << ", Value: " << entry.second <<
         std::endl;
    }

}
```

Artist.h

The Artist class in C++ is a straightforward data structure designed to encapsulate information about an artist. Here's the API description for this class:

1. *Data Members*
    - id: An integer variable used to store a unique identifier for each artist.
    - name: A string variable to store the name of the artist.
2. *Constructors*
    - Artist(): The default constructor.
    - Artist(int _id, std::string _name): A parameterized constructor that initializes an Artist object with a specific ID and name.
3. *Public methods*
    - Getter and setters for id and name

User.h

The User class is designed to represent a user in music recommendation systems. It provides mechanisms to manage a user's connections and their interactions with artists, aiding in features like friend lists and music preference tracking.

Here is an API description for this class:

- *Data Members*
    - id: An integer variable used to store the unique identifier for each user.
    - friendIDs: A set containing the IDs of the user's friends.
    - artistIDs: A set to store the IDs of artists that the user has listened to.
- Constructors
    - User(): The default constructor.
    - User(int _id): A parameterized constructor that initializes a User object with a specific ID.
- Member Functions
    - void addFriend(friendID): This function takes a friendID as an argument and inserts it into the friendIDs set.
    - void addArtist(artistID): This function takes an artistID as an argument and inserts it into the artistIDs set.
- Usage and Functionality
    - The User class is primarily used in systems that require tracking of user relationships and preferences, such as social networking sites or music streaming services.

Recommender.h: is a comprehensive C++ class designed for a music recommender system. It provides functionalities to manage user and artist data, and generate music recommendations based on users' listening habits and their social network. Here's a detailed API description:

1. *Data Members*
    - **users**: A hash map mapping user IDs to **User** objects. You'll keep the user objects in a linked list
    - **artists**: A hash map mapping artist IDs to **Artist** objects. You'll keep the artist objects in a linked list
2. *Constructor*
    - Initializes the **Recommender** instance by loading artists, user artists, and user friends data from specified files given.
3. *Private Methods*
    - **void loadArtists(fileName)**: Loads artist data from a file into the **artists** map.
    - **void loadUserArtists(fileName):** Loads user-artist relationships from a file, updating the **users** map.
    - **void loadUserFriends(fileName)**: Loads user-friend relationships from a file, updating the **users** map.
4. *Public Methods*
    - **void listFriends(userID)**: Lists all friends of a given user by their ID.
    - **void commonFriends(user1ID, user2ID)**: Prints common friends between two users.
    - **void listArtists(user1ID, user2ID)**: Lists artists listened to by both specified users.
    - **void recommend(int userID, int top)**: Generates and prints music artist recommendations for a given user, based on the listening habits of the user and their friends. It limits the recommendations to the top 'n' artists specified by the 'top' parameter.

Sample Run:

```
Recommender recommender("artists.dat","user_artists.dat","user_friends.dat");

recommender.listFriends(2);
recommender.commonFriends(2, 124);
recommender.listArtists(2, 4);
recommender.recommend(2, 10);
```

Sample Output:

```
Friends of User 2: 1869 1625 1585 1327 909 831 761 1209 1210 428 1230 515 275
Common friends of User 2 and User 124: 831 1230 275
Artists listened to by both User 2 and User 4:
 George Michael
Depeche Mode
Moby
Coldplay
Röyksopp
Air
Duran Duran

Recommendations for User 2:
Artist Name: Depeche Mode, Listen Count: 10
Artist Name: Duran Duran, Listen Count: 9
Artist Name: Madonna, Listen Count: 8
Artist Name: Simple Minds, Listen Count: 7
Artist Name: New Order, Listen Count: 6
Artist Name: a-ha, Listen Count: 6
Artist Name: Pet Shop Boys, Listen Count: 6
Artist Name: Erasure, Listen Count: 6
Artist Name: Michael Jackson, Listen Count: 6
Artist Name: The Human League, Listen Count: 5
```

Important Notes:

- For hash map, you are required to use your own implementation of HashMap
- For the set, it's permissible to utilize the C++ library's unordered_set to compile a collection of unique items.

What to Submit

For the music recommendation system project, please submit the following components:

1. *Source Code:*
    - Include all **.cpp** and **.h** files that make up your project. Ensure the code is well-commented to explain the functionality and logic behind your implementation.
    - A **main.cpp** file that demonstrates the functionality of the recommendation system through a series of tests.
2. **Documentation**:
    - A **README.md** file that provides an explanation of the data structures and algorithms used, and the rationale behind your design choices. And include what you've learned in this assignment.
    - .

How to Evaluate

The evaluation will be based on the following criteria:

1. **Functionality**:
    - Does the system correctly implement the features outlined in the project description?
2. **Code Quality**:
    - Is the code well-organized and consistently formatted?
    - Are variables and functions named clearly and descriptively?
    - Are there comments explaining complex sections of code?
3. **Performance**:
    - How efficiently does the system perform with large data sets?
    - Are there any noticeable performance bottlenecks or areas that could be optimized?
4. **Documentation**:
    - Does the **README.md** clearly explain the design and the reasons behind it?