CS 32, Winter 2023
Project 4: PnetPhlix

Due: 11 PM, Thursday, March 16

Make sure to read the entire document (especially Requirements and Other Thoughts) before starting your project.

Introduction

Before writing a single line of code or reading the rest of this document, you MUST first read AND THEN RE-READ the Requirements and Other Thoughts section. Print out this page, tape it to your wall or on the mattress above your bunk bed, etc. And read it over and over.

The NachenSmall Software Corporation, which has traditionally only built software for running senior-citizen bingo games, has decided to pivot into a new area. They've decided to disrupt the online movie recommendation market and build a new movie recommendation app called… PnetPhlix.

The evil co-CEOs, Carey and David (not to be confused with Harry and David, the purveyors of yummy gourmet gift baskets) have become increasingly frustrated with today's superficial, movie recommendations provided by NetFlix and Amazon Prime and have decided to create a new system to help people find the movies they really want to watch. They want their app to have the ability to provide recommendations for up to 20K movies for up to 100K users:

- Each user has a history of movies they've watched (up to hundreds of movies)
- Each movie has a set of attributes that describe it, including the movie title, release year, director(s), actor(s), genre(s) like horror, romance, etc., and a movie rating (e.g., 3.5 stars)

To identify recommended movies for a given person P, the PnetPhlix app determines what attributes are associated with the movies P has watched in the past, and then identifies new movies that also share as many of those attributes as possible. PnetPhlix rank orders movies by

the number of common attributes (e.g., if a user tends to watch horror films starring Selena Gomez, then the top recommendations should be horror films with Selena Gomez, followed by perhaps horror films with other actors or non-horror films starring Selena Gomez).

Given their eccentric ways, Carey and David initially asked a student at UC Berkeley to build their new movie recommendation app. The student was able to complete a set of interface declarations for the project, but then they resigned to go on a six-month Bikram Yoga and farming retreat in the Santa Cruz mountains to grow hydroponic hemp (you know those UCB undergrads).

The student was therefore unable to complete the actual implementation of these interfaces, so Carey and David will be providing you with his interfaces to build off of.

Your job for Project 4 is to build the six classes required to complete the movie recommendation app:

- **User**: A class that represents a user and their movie watching history
- **UserDatabase**: A class that stores all of the user objects and is searchable (e.g., find the user object associated with a given email address)
- **Movie**: A class that represents a movie, including a unique movie ID (e.g., ID15344), its title, release year, director(s), actor(s), genre(s), and star rating
- **MovieDatabase**: A class that stores all of the movie objects and is searchable (e.g., given a movie ID give me its movie object, or given an actor's name, give me all the movies starring that actor, etc.)
- **Recommender**: A class that produces rank-ordered movie recommendations for a given user given their previous watching history
- **TreeMultimap**: A templated class that implements a multimap that maps each key to one or more values, using a binary search tree[1] data structure

Here's what you might see when you run the program:

```
Enter a user's email for recommendations: KabiL@aol.com
Enter number of recommendations to provide: 8
User Kabir Luna has watched the following movies:
 Notes On Blindness
 Lana: Queen of the Amazons
 The Big Bang
 Homefront
 For Ellen
 Hitch
```

[1] Your TreeMultimap may use additional data structures, such as linked lists, as necessary.

… the rest of the movies are omitted for this example
The Valley of Gwangi

Here are the recommendations:
1. The Blues Brothers (1980)
   Rating: 3.79643
   Compatibility Score: 390

2. The Player (1992)
   Rating: 3.92309
   Compatibility Score: 281

3. Trust the Man (2005)
   Rating: 2.82645
   Compatibility Score: 250

4. Never So Few (1959)
   Rating: 3.11111
   Compatibility Score: 234

5. Knocked Up (2007)
   Rating: 3.40438
   Compatibility Score: 230

6. The Ghost Comes Home (1940)
   Rating: 2
   Compatibility Score: 230

7. Hot Fuzz (2007)
   Rating: 3.86654
   Compatibility Score: 227

8. Filth (2013)
   Rating: 3.64121
   Compatibility Score: 221

What Do You Need to Do?

Question: So, at a high level, what do you need to build to complete Project 4?

Answer: You'll be building six complete classes, detailed below:

Class #1: You need to build a class named **User** that holds a user's profile (e.g., their full name, email, and movie watching history).

Class #2: You need to build a class named **UserDatabase** that stores up to 100K user profiles and lets you search for a User object based on a user's email address.

Class #3: You need to build a class named **Movie** that holds a movie's details (e.g., the movie name, a unique ID for the movie, its release year, the director(s), actor(s), genre(s), and rating).

Class #4: You need to build a class named **MovieDatabase** that stores up to 20K movie objects and lets the user search for a movie object based on (a) the movie's unique ID, (b) a director of the movie, (c) an actor in the movie, or (d) the genre of the movie.

Class #5: You need to build a class named **Recommender** that can produce a set of movie recommendations for a given user based on their movie watching history.

Class #6: You need to build a class template named **TreeMultimap** that implements a binary search tree-based map (it can be more complex than a simple binary search tree if you like, but must be based on a binary search tree), capable of mapping any type of data (e.g., string, int, float) to any other type of data type (e.g., Movie, string, etc.).

You will find all of the gory details below in the section called Details.

What Will We Provide

We will provide the following for your use:

- A ***main.cpp*** file that has a main() function that lets you run/test your movie recommendation classes. You may modify this file for testing purposes, **BUT YOU WILL NOT TURN IT IN WITH YOUR SOLUTION**, so any changes must not be required for proper functioning of your solution**.**
- A ***users.txt*** file that contains a list of users and their movie watching history
- A ***movies.txt*** file that contains a list of movies and details for each movie (title, release year, director(s), actor(s), genre(s), and rating)

If you define your classes correctly, they should work perfectly with our main() driver and you'll have created your own awesome movie recommendation app!

Details

This section contains details about the classes you need to create.

User Class

You must build a class called User which implements a user's profile. A profile for a given user includes the following items:

- A person's name (e.g., David Sm0lbirg)
- A person's email address (e.g., sm0l_birg@gmail.com)
- One or more movies that the user has watched in the past

Your User class:

- **MUST NOT** add any new public member functions or variables
- MAY use any STL classes it likes
- MAY have any private methods/variables/structs/classes it needs (you can add these)

Your User class must have the following methods:

User(const std::string& full_name, const std::string& email, const std::vector<std::string>& watch_history)

This constructs a User object, specifying the user's name, email address and previous movie watching history (a vector of movie ID strings, e.g., "ID12345", indicating the movies the person has watched).

~User()

You may define a destructor for User if you need one.

std::string get_full_name() const

The get_full_name method returns the user's name.

std::string get_email() const

The get_email method returns the user's email address.

std::vector<std::string> get_watch_history() const

The get_watch_history method returns a vector of movie IDs, e.g., "ID00001" that the user has watched in the past.

UserDatabase Class

The UserDatabase class is responsible for loading and keeping track of all of PnetPhlix's users and making them easily searchable by their email address.

Your UserDatabase class:

- **MUST** meet the following big-O requirements. (You may assume that there is a constant that will not be exceeded by the length of any email address, user name, or movie ID string, so these string lengths need not be factored into your big-O calculations.)
    - Finding a user profile from an email address must run in O(log U) time, where U is the number of distinct users in the database.
    - Loading a user database of U users must run in O(U log U) time.
- **MUST** be case sensitive for all email address searches
- **MUST** be able to accommodate at least 100K users
- **MUST** use the TreeMultimap class that you will write to map each email address to a User object
- **MUST NOT** contain ANY member variables that use the standard associative types (std::set/map/multiset/multimap, std::unordered_set/map/multiset/multimap)
- **MUST NOT** add any new *public* member functions or variables
- MAY use the STL list and vector classes
- MAY have *private* member functions/variables/structs/classes you choose to add

The big-O requirements are for the average case assuming all orderings of user records in the user data file are equally probable. There will be a small percentage of orderings that would cause a worse time complexity than specified in a requirement, but you need not worry about them.

Your UserDatabase class has the following methods:

UserDatabase()

The user database constructor. It must be a default constructor (i.e., take no arguments).

~UserDatabase()

You may define a destructor if you need one. It must cause any dynamically allocated memory that the object holds to be freed.

bool load(const std::string& filename)

This method must load a user database from the specified input file, e.g., ***users.txt***. If it has not yet been called before for this UserDatabase object and it is successful in loading all of the user records in the file, it must return true. Otherwise, it must return false. (If it returns false, the state of the object may or may not have been changed.) The method must load the data into data structures that enable you to meet the big-O requirements specified in the section above.

The users data file is a text file with the following format:

```
Person 1’s name
Person 1’s email address
Count of number of movies for person 1
movieID#1
movieID#2
…
movieID#n

Person 2’s name
Person 2’s email address
Count of number of movies for person 2
movieID#1
movieID#2
…
movieID#n

etc.
```

The block of lines for one user's record is separated from that of the next by one empty line.

For example:

```
Carey Nachenberg
climberkip@gmail.com
4
ID22937
ID04742
ID02001
```

```
ID05573

David Sm0lbirg
sm0l_birg@gmail.com
3
ID09830
ID28583
ID07146
```

Strings in the file are case-sensitive, so `Carey` is not the same as `cAReY`. You can look at our synthetically-generated ***users.txt*** file for an example of what you will have to parse. We may try to load files other than the provided users.txt, so test your code carefully.

Under normal circumstances, a program should not assume its input is correctly formed, since the person or program that produced that input may have made careless or deliberately malicious mistakes. For this project, though, so that you can focus your attention on other issues, your program may assume that:

- There are no extraneous spaces at the beginning or ending of any line.
- The first two lines of a user record are non-empty strings.
- No two user records have the same second line (email address).
- The third line of a user record consists of a nonnegative integer, let's call it *n*.
- The *n* lines after the third line are non-empty strings, no two of them the same.
- Between any two user records, there is exactly one empty line.

There might or might not be an empty line after the last user record.

User* get_user_from_email(const std::string& email) const

This method must find the user identified by the specified email address and return a pointer to that user's User object, or nullptr if the specified user cannot be found. This function performs a case-sensitive search.

Movie Class

You must build a class called Movie which holds the details about a movie:

- The ID of the movie. Every movie has a unique 7-character ID code, which always starts with "ID" and then has five numeric digits, e.g., "ID10782".
- The movie's name (e.g., "Back to The Future")
- The movie's release year (e.g., "1985")
- The movie's director(s)

- The movie's actor(s)
- The genre(s) of the movies, e.g., action, romance, mystery
- The star-rating of the movie (0-5), as rated by users

Your Movie class:

- **MUST NOT** contain ANY member variables that use the standard associative types (std::set/map/multiset/multimap, std::unordered_set/map/multiset/multimap)
- **MUST NOT** add any new *public* member functions or variables
- MAY use the STL list and vector classes
- MAY have *private* member functions/variables/structs/classes you choose to add

Your Movie class must have the following methods:

Movie(const std::string& id, const std::string& title,
const std::string& release_year,
const std::vector<std::string>& directors,
const std::vector<std::string>& actors,
const std::vector<std::string>& genres,
float rating);

This constructs a Movie object, specifying the Movie's ID, title, release year, director(s), actor(s), genre(s) and rating on a 5-star scale.

~Movie()

You may define a destructor for Movie if you need one.

std::string get_id() const

The get_id method returns the Movie's unique 7-character identifier (e.g., "ID12345").

std::string get_title() const

The get_title method returns the Movie's title (e.g., "Back to The Future").

std::string get_release_year() const

The get_release_year method returns the Movie's release year (e.g., "1985").

float get_rating() const

The get_rating method returns the Movie's rating on a 1-5 scale (may be fractional).

std::vector<std::string> get_directors() const

The get_directors method returns the Movie's director(s).

std::vector<std::string> get_actors() const

The get_actors method returns the Movie's lead actors(s).

std::vector<std::string> get_genres() const

The get_actors method returns the Movie's genre(s), (e.g. Romance or Horror).

MovieDatabase Class

The MovieDatabase class is responsible for loading and keeping track of all of PnetPhlix's Movies and makes it easy to perform a *case-insensitive* search by any of the following and locate all relevant movies:

- Movie ID: Give me the movie with this 7-character ID
- Director: Give me all movies directed by a particular director
- Actor: Give me all movies that a particular actor starred in
- Genre: Give me all movies that are of a particular genre

Your MovieDatabase class:

- **MUST** meet the following big-O requirements. (You may assume that there is a constant that will not be exceeded by the length of any movie ID, movie name, director name, actor name, or genre string, so these string lengths need not be factored into your big-O calculations. You may also assume that the number of directors, actors, and genres listed for each movie is small enough that they need not be factored into your big-O calculations. Further, you may assume that the number of distinct directors and distinct actors is proportional to the number of distinct movies; the number of distinct genres is much, much less than the number of distinct movies.)
    - Assuming there are M movies in the database, searching by a movie ID must run in $O(\log M)$ time.
    - Assuming there are D unique directors across all movies, and each director has an average of m_d movies they've directed, obtaining a list of movies that a given director directed must run in $O(\log D + m_d)$ time.
    - Assuming there are A unique actors across all movies, and each actor has an average of m_a movies they've acted in, obtaining a list of movies that a given actor acted in must run in $O(\log A + m_a)$ time.

    - Assuming there are G unique genres across all movies, and each genre has an average of m_g movies associated with that genre, obtaining a list of movies of a given genre must run in O(log G + m_g) time.
    - Loading a database of M movies must run in O(M log M) time.
- **MUST** be case ***insensitive*** for all movie ID, director, actor, and genre searches
- **MUST** be able to accommodate at least 20K Movies
- **MUST** use the TreeMultimap class that you will write to map IDs, directors, actors, and genres to movies
- **MUST NOT** contain ANY member variables that use the standard associative types (std::set/map/multiset/multimap, std::unordered_set/map/multiset/multimap)
- **MUST NOT** add any new *public* member functions or variables
- MAY use the STL list and vector classes
- MAY have *private* member functions/variables/structs/classes you choose to add

The big-O requirements are for the average case assuming all orderings of movie records in the movie data file are equally probable. There will be a small percentage of orderings that would cause a worse time complexity than specified in a requirement, but you need not worry about them.

Your MovieDatabase class has the following methods:

MovieDatabase()

The movie database constructor. It must be a default constructor (i.e., take no arguments).

~MovieDatabase()

You may define a destructor if you need one. It must cause any dynamically allocated memory that the object holds to be freed.

bool load(const std::string& filename)

This method must load a movie database from the specified input file, e.g., ***movies.txt***. If it has not yet been called before for this MovieDatabase object and it is successful in loading all of the movie records in the file, it must return true. Otherwise, it must return false. (If it returns false, the state of the object may or may not have been changed.) The method must load the data into data structures that enable you to meet the big-O requirements specified in the section above.

The movies data file is a text file with the following format:

```
Movie ID 1
Movie Name
```

```
Movie Release Year
director1,director2,...,directorn
actor1,actor2,...,actorn
genre1,genre2,...,genren
rating out of 5 stars

Movie ID 2
Movie Name
Movie Release Year
director1,director2,...,directorn
actor1,actor2,...,actorn
genre1,genre2,...,genren
rating out of 5 stars
etc.
```

The block of lines for one movie record is separated from that of the next by one empty line.

For example:

```
ID25779
The Carpet of Horror
1962
Harald Reinl
Joachim Fuchsberger,Karin Dor,Werner Peters,Eleonora Rossi Drago,Carl
Lange
Crime,Horror
2.25

ID35763
Monkey Love Experiments
2014
Will Anderson
Tobias Feltus,Oliver Henderson
Animation,Drama
4
```

You can look at our synthetically-generated ***movies.txt*** file for an example of what you will have to parse. We may try to load files other than the provided movies.txt, so test your code carefully.

Your program may assume that:

- There are no extraneous spaces at the beginning or ending of any line, or before or after any comma.
- The first six lines of a movie record are non-empty strings.

- No two movie records have the same first line (movie ID).
- The fourth, fifth, and sixth lines are comma-separated lists of non-empty director names, actor names, and genres, respectively. No two director names on the fourth line are the same, no two actor names on the fifth line are the same, and no two genres on the sixth line are the same.
- The seventh line of a user record consists of a number.
- Between any two movie records, there is exactly one empty line.

There might or might not be an empty line after the last movie record.

Movie* get_movie_from_id(const std::string& id) const

This method must find the movie identified by the specified movie ID and return a pointer to that movie's Movie object, or nullptr if the specified movie cannot be found. This function performs a case-insensitive search.

std::vector<Movie*> get_movies_with_director(
const std::string& director) const

This method must return a vector of pointers to all Movie objects that have the specified director as one of its directors; if there are no such movies, it returns an empty vector.

std::vector<Movie*> get_movies_with_actor(
const std::string& actor) const

This method must return a vector of pointers to all Movie objects that have the specified actor as one of its actors; if there are no such movies, it returns an empty vector.

std::vector<Movie*> get_movies_with_genre(
const std::string& genre) const

This method must return a vector of pointers to all Movie objects that have the specified genre as one of its genres; if there are no such movies, it returns an empty vector.

Recommender Class

The Recommender class is the workhorse of this project. Given a user's email address and a number of movies to recommend, it must look up the movie watching history associated with that user (using the UserDatabase and User classes), identify candidate movies that the user hasn't yet watched (using the MovieDatabase and Movie classes), and then rank-order those movies as recommendations for the user. Finally, it must output the top N recommendations (including each movie's "Compatibility Score" and Movie ID of the recommended movies).

Your Recommender class:

- **MUST** run as efficiently as possible - we will not state any exact big-O requirements, but try to make your code as efficient as possible—avoid O(N) algorithms where better ones are possible, for example. If you code things correctly, you should be able to find matches in a few seconds even across tens of thousands of movie/user profiles!
- **MUST NOT** add any new *public* member functions or variables
- MAY use any STL containers you like
- MAY have *private* member functions/variables/structs/classes you choose to add

Your Recommender class has the following methods:

Recommender(const UserDatabase& user_database,
const MovieDatabase& movie_database)

The Recommender constructor. It takes in fully-loaded user and movie database objects as parameters (you must load the user database and movie database first before instantiating your Recommender object). These will be used to generate movie recommendations.

~Recommender()

You may define a destructor if you need one. It must cause any dynamically allocated memory that the object holds to be freed.

std::vector<MovieAndRank> recommend_movies(
const std::string& user_email, int movie_count) const

Outside your declaration of the Recommender class, your Recommender.h file must declare the following:

```
struct MovieAndRank
{
```

```
    MovieAndRank(const std::string& id, int score)
     : movie_id(id), compatibility_score(score)
    {}
    std::string movie_id;
    int compatibility_score;
};
```

The recommend_movies method is responsible for:

- Taking as parameters an email address for a user, and a maximum count of how many movies to recommend to the user based on the user's watching history
- Using the provided email address to find all of the movies the user has watched in the past
- For each movie m the user has previously watched, determining a compatibility score:
    - For each director d associated with m, each movie in the movie database that has d in its list of directors contributes 20 points to the compatibility score
    - For each actor a associated with m, each movie in the movie database that has a in its list of actors contributes 30 points to the compatibility score
    - For each genre g associated with m, each movie in the movie database that has g in its list of genres contributes 1 point to the compatibility score
- For the movies that have a compatibility score of at least 1, filtering out the movies that the user has already watched (so you don't recommend them)
- Rank ordering all candidate movies that have a compatibility score of at least 1, breaking ties as follows:
    - Movies with a higher compatibility score must ordered higher on the recommendation list than movies with a lower compatibility score
    - If two or more movies have the same compatibility score, they must be further ordered based on:
        - The movie's rating; a movie with a higher 5-star rating must be ordered higher on the recommendation list than a movie with a lower 5-star rating
        - If two or more movies have the same compatibility score and the same 5-star rating, then they must be further ordered based on the movie's name, alphabetically, in ascending order (e.g., Avalanche comes before The Matrix)
- Returning a vector with the requested number of the most highly ranked movie recommendations, ordered as described above. If fewer compatible movies were found than the requested number, then the vector must have all movies with a compatibility score of at least 1, ordered as described above. (If the movie_count parameter is negative, act as if it were 0, returning an empty vector.)
    - Each recommendation (using the MovieAndRank struct) has the movie's ID (which can be used to look up the movie's details), and the compatibility score.

Here’s how your method might be called.

```
void findMatches(const Recommender& r,
                 const MovieDatabase& md,
                 const string& user_email,
                 int num_recommendations) {
      // get up to ten movie recommendations for the user
    vector<MovieAndRank> recommendations =
                  r.recommend_movies(user_email, 10);
    if (recommendations.empty())
       cout << "We found no movies to recommend :(.\n";
    else {
      for (int i = 0; i < recommendations.size(); i++) {
        const MovieAndRank& mr = recommendations[i];
        Movie* m = md.get_movie_from_id(mr.movie_id);
        cout << i << ". " << m->get_title() << " ("
             << m->get_release_year() << ")\n   Rating: "
             << m->get_rating() << "\n   Compatibility Score: "
             << mr.compatibility_score << "\n";
      }
    }
}
```

Example Results from Recommender

Here is an example of what the example function above would output assuming we have the following users and movies, and asked for the top 10 movies for climberkip@gmail.com:

users.txt

```
Carey Nachenberg
climberkip@gmail.com
2
ID00001
ID00003
```

movies.txt

```
ID00001
Back to The Future
1985
Robert Zemeckis
```

```
Michael J. Fox,Christopher Lloyd,Lea Thompson,Crispin Glover
Action,Adventure,Sci-Fi
4.25

ID00002
Rocky Horror Picture Show
1975
Jim Sharman
Tim Curry,Richard O'Brien,Susan Sarandon,Meat Loaf
Horror,Comedy
3.7

ID00003
The Matrix
1999
Andy Wachowski,Larry Wachowski
Laurence Fishburne,Keanu Reeves,Hugo Weaving,Carrie-Anne Moss
Action,Sci-Fi,Thriller
4.15952

ID00004
The Matrix Reloaded
2003
Andy Wachowski,Larry Wachowski
Keanu Reeves,Laurence Fishburne,Carrie-Anne Moss,Hugo Weaving
Action,Adventure,Sci-Fi,Thriller,IMAX
3.36812

ID00005
Collider
2018
Justin Lewis
Christine Mascolo,Jude Moran,Conner Greenhalgh,Heath C. Heine
Sci-Fi
3

ID00006
The Psychotronic Man
1980
Jack M. Sell
Peter Spelson,Chris Carbis,Curt Colbert,Robin Newton
Horror,Sci-Fi
2

ID00007
Alien Outlaw
1985
Phil Smoot
```

```
Stephen Winegard,Kimberly Mauldin,Stuart Watson,Sunset Carson,Kari Anderson
Comedy,Horror,Mystery,Sci-Fi,Western
3

ID00008
Running with the Devil
2019
Jason Cabell
Nicolas Cage,Leslie Bibb,Clifton Collins Jr.,Barry Pepper,Laurence Fishburne
Crime,Drama,Thriller,Adventure
2.93182
```

Ranked Output

```
1. The Matrix Reloaded (2003)
   Rating: 3.36812
   Compatibility Score: 166

2. Running with the Devil (2019)
   Rating: 2.93182
   Compatibility Score: 32

3. Alien Outlaw (1985)
   Rating: 3
   Compatibility Score: 2

4. Collider (2018)
   Rating: 3
   Compatibility Score: 2

5. The Psychotronic Man (1980)
   Rating: 2
   Compatibility Score: 2
```

Output Explained

Carey has watched two movies, Back to the Future and The Matrix.

Back to The Future has:

Directors: Robert Zemeckis
Actors: Michael J. Fox,Christopher Lloyd,Lea Thompson,Crispin Glover

Genres: Action,Adventure,Sci-Fi

The Matrix has:

Directors: Andy Wachowski,Larry Wachowski
Actors: Keanu Reeves,Laurence Fishburne,Carrie-Anne Moss,Hugo Weaving
Genres: Action,Sci-Fi,Thriller

If we enumerate all of the movies in our database with one or more of the same directors, actors, or genres of Carey's two movies, we find the following movies that are candidates for recommendations (Carey hasn't watched them yet):

- The Matrix Reloaded
- Running with the Devil
- Alien Outlaw
- Collider
- The Psychotronic Man

The Matrix Reloaded would get a compatibility score of 166 and thus ranked first:

- 20 points for sharing the same director with The Matrix, Andy Wachowski
- 20 points for sharing the same director with The Matrix, Larry Wachowski
- 30 points for sharing the same actor with The Matrix, Laurence Fishburne
- 30 points for sharing the same actor with The Matrix, Keanu Reeves
- 30 points for sharing the same actor with The Matrix, Hugo Weaving
- 30 points for sharing the same actor with The Matrix, Carrie-Anne Moss
- 1 point for sharing the same genre with The Matrix, Action
- 1 point for sharing the same genre with The Matrix, Thriller
- 1 point for sharing the same genre with The Matrix, Sci-Fi
- 1 point for sharing the same genre with Back to The Future, Action
- 1 point for sharing the same genre with Back to The Future, Adventure
- 1 point for sharing the same genre with Back to The Future, Sci-Fi

Running With the Devil would get a compatibility score of 32 and thus ranked 2nd:

- 30 points for sharing the same actor with The Matrix, Laurence Fishburne
- 1 point for sharing the same genre with The Matrix, Thriller
- 1 point for sharing the same genre with Back to The Future, Adventure

Alien Outlaw, Collider and The Psychotronic Man would all get a compatibility score of 2 because each shares two common genres with Carey's movies. Since Alien Outlaw and Collider both have a 5-star rating of 3, they will be the next highest ranked movies, and since there's a tie on their 5-star rating, they will be ordered alphabetically (Alien Outlaw before Collider). Finally, since The Psychotronic Man has a rating of 2, it will be ranked last. None of the other

movies in the database share any genre, director or actor in common, so they are not recommended.

TreeMultimap Class

You MUST write a templated class named *TreeMultimap<KeyType, ValueType>* that implements a multimap data structure based on a hand-coded binary search tree. Your multimap must be able to efficiently map any value of *KeyType* to one or more values of the *ValueType*. Unlike a traditional map that maps each key to a single value, a multimap can map a given key to potentially many values. Your TreeMultimap class must support both inserting new items and searching for items, but not deleting a particular item.

Here's the interface that you MUST implement in your TreeMultimap class:

```
template <typename KeyType, typename ValueType>
class TreeMultimap {
public:

  class Iterator {
    public:
      Iterator();  // you may add additional constructors
      ValueType& get_value() const;
      bool is_valid() const;
      void advance();
  };

  TreeMultimap();
  ~TreeMultimap();
  void insert(const KeyType& key, const ValueType& value);
  Iterator find(const KeyType& key) const;
};
```

As you can see, this class has a nested Iterator class. When you search for a particular key in the multimap, you receive an Iterator object as the result. The iterator object can then be used to obtain all values that were associated with the searched-for key (since there may be more than one value associated with a key). Here's how your class could be used:

```
      int main() {
        TreeMultimap<std::string, int> tmm;

        tmm.insert("carey", 5);
        tmm.insert("carey", 6);
```

```
  tmm.insert("carey", 7);
  tmm.insert("david", 25);
  tmm.insert("david", 425);

  TreeMultimap<std::string,int>::Iterator it = tmm.find("carey");
  // prints 5, 6, and 7 in some order
  while (it.is_valid()) {
   std::cout << it.get_value() << std::endl;
   it.advance();
  }

  it = tmm.find("laura");
  if (!it.is_valid())
    std::cout << "laura is not in the multimap!\n";
}
```

Your TreeMultimap class:

- **MUST** be a class template, implemented fully in ***treemm.h***.
- **MUST** be based on a binary search tree in some way, where each node of the BST contains a key (and other members, up to you).
- **MUST** have finding a value that matches a particular key run in O(log K) time, where K is the number of distinct keys in the tree.
- **MUST** be able to accommodate any number of key-value pairs
- **MUST NOT** contain ANY member variables that use the standard associative types (std::set/map/multiset/multimap, std::unordered_set/map/multiset/multimap)
- **MUST NOT** add any new *public* member functions or variables
- MAY use the STL list and vector classes
- MAY have *private* member functions/variables/structs/classes you choose to add

Your Iterator class must be a nested class of your TreeMultimap class and must adhere to the following requirements:

- **MUST** have a big-O of O(V) for iterating through all V values associated with a given key (via calls to the is_valid(), get_value() and advance() methods) after you have found a matching value
- **MUST NOT** contain ANY member variables that use the standard associative types (std::set/map/multiset/multimap, std::unordered_set/map/multiset/multimap)
- **MUST NOT** add any new *public* member functions or variables
- MAY use the STL list and vector classes
- MAY have *private* member functions/variables/structs/classes you choose to add
- MAY have any number or type of parameters in its constructor, as well as multiple constructors if necessary

The big-O requirements are for the average case assuming all orderings of insertions into the tree are equally probable. There will be a small percentage of orderings that would cause a worse time complexity than specified in a requirement, but you need not worry about them.

You may assume that the *KeyType* has comparison operators: < defining an ordering relation, == defining an equivalence relation, and the related >, <=, >=, and !=. All the builtin types and strings have these, of course, but a user who wants a key type of some class of their own is responsible for defining the comparison operators for that type.

Here are the specs for your TreeMultimap methods:

TreeMultimap()

The TreeMultimap constructor. It must take no arguments.

~TreeMultimap()

The TreeMultimap destructor. It must free any dynamically allocated memory TreeMultimap methods created to implement the tree data structure. It is NOT responsible for freeing any dynamically allocated objects to which pointers were passed in to TreeMultimap::insert. For example:

```
class Dog { … };

int main() {
  Dog* d1 = new Dog("Koda");
  Dog* d2 = new Dog("Spot")

  TreeMultimap<std::string, Dog*>* tmmptr =
                            new TreeMultimap<std::string, Dog *>();
  tmmptr->insert("carey", d1);
  tmmptr->insert("cindy", d2);
  delete tmmptr;   // TreeMultimap's destructor runs
                   // It must not free the dog objects, just tree nodes

    // User is responsible for freeing any dynamically allocated
    // objects that were passed to insert to be added to the tree
  delete d1;
  delete d2;
}
```

void insert(const std::string& key, const ValueType& value)

The insert method must update the multimap to associate the specified key with a **copy** of the passed-in value. (Note that if ValueType is some pointer type, this means the pointer is copied, not the object pointed to.) If the same key is associated with more than one value via successive calls to insert(), then the tree should maintain all such associations. For example, insert(k,v1); insert(k,v2); insert(k,v1) should result in the tree mapping k to two instances of v1 and one of v2.

Iterator find(const KeyType& key) const

The find() method is responsible for searching your multimap for the specified key. The search function must return a TreeMultimap::Iterator object which can be used to retrieve values associated with the searched-for key. If the specified key was not found, the method must also return a TreeMultimap::Iterator object, but one which is "invalid" (an invalid Iterator object is one which does not refer to any value).

```
TreeMultimap<int, int> tmm;
  // insert a bunch of stuff in the tree
Iterator it = tmm.find(42);
  // your code can now use the iterator to access the values associated
  // with a key of 42
```

TreeMultimap::Iterator

The Iterator class is a nested class which must be defined inside of the TreeMultimap class. The two classes must be designed together such that the iterator understands how to access the contents of the tree's nodes and identify all potential matches with a key during iteration.

Here are the specs for your TreeMultimap::Iterator methods:

Iterator()

The TreeMultimap::Iterator constructor. You may define multiple Iterator constructors and/or specify any set of arguments you like for these constructors, since the only class that will instantiate a TreeMultimap::Iterator is your TreeMultimap class. No matter whether you declare other constructors, your TreeMultimap::Iterator MUST have a default constructor that initializes an invalid Iterator object (that doesn't refer to any values), so this code can work:

```
TreeMultimap<int,int>::Iterator it;  // doesn't point to any value
if (!it.is_valid()) std::cout << "This will print!\n";
```

~Iterator()

You may define a destructor for TreeMultimap::Iterator if you need one. If you do, it must free any dynamically allocated memory TreeMultimap::Iterator methods created to implement the iterator. It is NOT responsible for freeing any dynamically allocated objects to which pointers were passed in to any TreeMultimap::Iterator constructor.

ValueType& get_value() const

This method returns a reference to the current value referred to by the iterator. If the iterator is invalid (e.g., doesn't refer to a valid value), the behavior of this method is not defined; this means your code can do anything it likes, or not do anything special about this case at all.

bool is_valid() const

This method returns a true if the iterator is valid (currently refers to a value) or false otherwise.

void advance()

This method advances the iterator to refer to the next value associated with the searched-for key; if there is no next value, the iterator becomes invalid. When you advance through all the values associated with a particular key, the order in which those values are visited is up to you; this spec does not require any particular ordering. If the iterator is invalid at the time this method is called, the behavior of this method is not defined.

Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

- Every time you make progress, back up your code to Google Drive, iCloud, a flash drive, or some other place not on your computer. WE WILL NOT ACCEPT CRASHED COMPUTERS/LOST FILES AS AN EXCUSE. Use the SEASnet lab machines if your computer crashes!
- If you use the Visual Studio or Xcode debugger, you will probably shave off about 50% of your development time for this project. Use the debugger!
- The entire project can be completed in roughly 300 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.

- You must not add any public member variables/functions to your classes (except additional constructors where the spec explicitly permits this). You may add private member variables/helper methods/structs/classes.
- Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. How will you use these data structures? Plan before you program!
- Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
- Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
- To get full credit, you may use only those STL containers that are explicitly permitted for each class. However, if you're having trouble building a data structure from scratch, feel free to use the STL containers to help you make progress, and replace them later with your own code if you can. A class you turn in that uses a banned STL container will not earn any points, but won't affect the score that other classes earn.

If you don't think you'll be able to finish this project, then take some shortcuts. For example, if you can't get your TreeMultimap class working with a hand-built tree, use the STL map or unordered_map class to temporarily implement your TreeMultimap class so that you can proceed with implementing other classes, and go back to fixing your TreeMultimap class later.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to fully debug a class (e.g., MovieDatabase), we will provide a correct version of that class and test it with the rest of your program (e.g., we'll test *our* correct MovieDatabase class with *your* Recommender class). If you implemented the rest of the program properly, our version of the MovieDatabase class should work perfectly with your version of the Recommender class and we can give you credit for those parts of the project you completed.

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors under both g32 and either Visual Studio or clang++!

What to Turn In

You must turn in twelve to fourteen files:

User.h	Contains your declaration of User
User.cpp	Contains your implementation of User
Movie.h	Contains your declaration of Movie
Movie.cpp	Contains your implementation of Movie

UserDatabase.h	Contains your declaration of UserDatabase
UserDatabase.cpp	Contains your implementation of UserDatabase
MovieDatabase.h	Contains your declaration of MovieDatabase
MovieDatabase.cpp	Contains your implementation of MovieDatabase
Recommender.h	Contains your declaration of Recommender
Recommender.cpp	Contains your implementation of Recommender
treemm.h	Contains your TreeMultimap class template (there is no treemm.cpp)
utility.h	Contains any utility function declarations used by multiple classes
utility.cpp	Contains any utility function implementations used by multiple classes
report.docx, report.doc, or report.txt	Contains your report

If you write a helper function used by only one class, put it in that class's .cpp file. If you write a helper function used by more than one class (e.g., operator<), put its prototype in utility.h and its implementation in utility.cpp. If you have no need for such utility functions, do not submit utility.h or utility.cpp.

If you choose to implement some functions as inline functions, put their implementations in the appropriate header file, not a .cpp file. If you do this for all of the implementations from a .cpp file, leaving the .cpp file empty, turn in the empty .cpp anyway.

Comment any complicated part of your code.

You must submit a brief (You're welcome!) report that details the parts of the project:

- that you were unable to finish
- that have bugs that you have not yet been able to fix
- that use banned STL components (perhaps because you used them for placeholder implementations but didn't have time to rewrite the code to remove them)

The report must also document how you tested your various classes. Either a paragraph or two about how you tested each class or a list of test cases is fine. For example, you might provide a list of items like this: "I inserted *car* then *carey*, then searched to make sure both were found in the TreeMultimap."

Grading

- 95% of your grade will be assigned based on the correctness and efficiency of your solution
- 5% of your grade will be based on your report

Good luck!