CSCI-1200 Data Structures — Spring 2022
Homework 9 — IMDB Search

The Internet Movie Database (IMDB) keeps track of data related to films. We have processed some of the data from https://www.imdb.com/interfaces/ since the full dataset has some interesting problems and is far too large for us to work with.

Your program will take input from `std::cin` which constitutes a list of commands. Some of these commands will describe how your hash table for the assignment will work, and other commands will request output to `std::cout`.

The goal is to write our own simple search engine for movie data. In a search, we may have exact information, or we may only have partial information (more on this later). In order to make our search time-efficient we will use extra memory in our hash table to store more "keys".

Movie Data

When your program receives the `movies` command, it will be given a filename to read from. Each entry in the file will look like:

```
Title
Year of Release
Runtime in Minutes
GenreCount GenreList
ActorCount ActorList
RoleCount RoleList
```

There will be no spaces in any of the data, for example instead of "John Wayne" we would have "John_Wayne". None of the fields will be blank, and `GenreCount`, `ActorCount`, `RoleCount` will all be ≥ 1. `GenreList` will have `GenreCount` entries, `ActorList` will have `ActorCount` entries, and `RoleList` will have `RoleCount` entries. Refer to the sample for more concrete examples.

One strange thing about the data is that instead of the actors having names, they all have strings starting with "`n`" which we will refer to as *"nconsts"*. This is because there might be two actors with the same name, so we instead use a unique identifier. However, there is another command `actors` which also gives a filename. The structure of the actors file is much simpler with each line consisting of:

```
nconst ActorName
```

The third major operation is `query`. After the word `query` your program can expect input that looks like an entry from the movie file. However, it is also possible to have unknown values. If the title, runtime, or year are "?" it means that they are a wildcard and will match any movie. If the GenreCount, ActorCount or RoleCount are 0, it means they are a wildcard and will match any movie. When your program receives a query, it should search the hash table (more on that in a moment) and return all results that match. In order for two GenreLists, ActorLists, or RoleLists to match, they must have the same number of values, every value must match, and the values must be in the same order.

Choice of Hash Function and Table Implementation

The choice of the hash function is up to you. A good hash function should be fast, O(1) computation, and provide a random, uniform distribution of keys throughout the table. You may use one of the hash functions mentioned in lecture, one found on the internet, or one of your own devising. If you choose to download a hash function from the internet, you must provide the URL in your README and include the source code with your submission. If the downloaded file requires a copyright notice, you MUST include that notice. Be sure to observe any copyright restrictions on the use of the code. In your README file, describe your hash function and table implementation.

You may write as many classes as you feel are necessary, however you must have exactly one hash table and the representation must be a `vector<pair<QUERY, list<MOVIE_DATA> > >` where `QUERY` is your representation of a single query (may have wildcards), and `MOVIE_DATA` should be all the information about one movie. To handle collisions, use one of the open addressing methods described in lecture (linear probing, quadratic probing, or secondary hashing). Linear probing is the simplest of these three methods. Since you are not separate chaining, you might be confused about why the value has a list. Each entry in the hash table consists of a query, and a list of all movies that match that specific query. You should not add a movie's data to a particular list in the hash table unless it matches the corresponding key (query). **You will receive a significant penalty if you do not follow this structure.**

A common point of confusion about this assignment comes from the fact that our hash table's value type is `list<MOVIE_DATA>` meaning that a value may consist of information about several movies in a linked list. This is **not** separate chaining, because separate chaining resolves hash collisions. All the movies represented within a given list correspond to a single key (for example, for the `QUERY` that is "all movies from 1986", there could be multiple movies in the linked list). The key (which is what we do our hashing based on) is the `QUERY`. If we try to insert a query into a slot in the hash table and that slot is already being used by a different, then open addressing must be found to find an empty spot in the hash table.

One of your challenges will be figuring out how to hash the queries. We will not be picky about which approach you take for the value, but it is strongly recommended that your values in your hash table are pointers, since you will otherwise duplicate the movie data many times and the duplicates may cause you to run out of memory quickly, particularly on Submitty. *Hint: If you use an appropriate STL data structure to hold the data in main, then you will not need to use new/delete in your code.*

The key in your hash table should be a unique identifier of a query, which is based on movie data. Where this gets interesting is that in addition to a query consisting of the full movie data (no wildcards), you will need to hash every partial version of a movie object (with one or more wildcards), since a given movie can be an answer in approximately 64 different queries! Recall from the description above of the `query` command that you may be given only some information with others left as unknowns. If implemented correctly, constructing the hash table should be where most of the work happens, with a lookup being very quick.

For example, the second query in `top250_example_input.txt` looks for any movie from 1994, so all movies in the dataset that have a year of 1994 should be values associated with the `QUERY` corresponding to the partial movie data where everything is empty except the year, which is set to 1994. Representing partial queries is part of the challenge in this assignment though one approach is simply to use empty strings/containers. Generating all of the partial queries may be made easier by using the code in *starter_code.cpp* which generates all possible sequences of 6 numbers that are each 0 or 1.

You may not use *std::hash*, *std::unordered_map*, *std::unordered_set*, *std::map* or similar STL functions/containers. **Exception:** You can use a map to store actor data, but not movie data.

When implementing the hash table, set the initial size of the table. As you enter data in the table, calculate the occupancy ratio:

$$occupancy = \frac{number\,of\,unique\,key\,entries}{table\,size}$$

When the *occupancy* >than some fixed level, double the size of the table and rehash the data. Describe your re-sizing method in the hash table section of the README file.

Input/Output & Basic Functionality

The program should read a series of commands from std::cin (STDIN) and write responses to std::cout (STDOUT). Sample input and output files have been provided. You can redirect the input and output to your program using the instructions in the section **Redirecting Input & Output** found at `http://www.cs.rpi.edu/academics/courses/spring22/csci1200/other_information.php`

Your program should accept the following commands:

- *movies filename* - Read movie data from *filename*.
- *actors filename* - Read actor data from *filename*.
- *table_size N* - this is an optional command. *N* is an integer. It is the initial hash table size. If it does not appear in the command file, the initial table size should be set to 100.
- *occupancy f* - this is an optional command. *f* is a float. When the occupancy goes above this level, the table should be resized. If it does not appear in the command file, the initial level should be set to 0.5.
- *query query_data* - Search the movie data for any matches to *query_data*.
- *quit* - Exit the program.

You may assume that if `table_size` or `occupancy` appear, they will appear before `movies`. You can also assume that `movies` and `actors` will appear before any `query` commands.

For output to queries, if matches are found, the program will print the number of matches and the corresponding movie data. Otherwise, the program will print "`No results for query.`" Since the ordering will depend on your hash function, you are not required to print the results of a query in any particular order. One subtle detail is that when outputting movie data you should use the actual names of actors instead of their nconsts.

You are not explicitly required to create any new classes when completing this assignment, but please do so as it will improve your program design. We expect you to use `const` and pass by reference/alias as appropriate throughout your assignment.

Submission

Use good coding style and detailed comments when you design and implement your program. Please use the provided template `README.txt` file for any notes you want the grader to read. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity". If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.