CS126 Coursework Report

Introduction

In this report, I will discuss the design and implementation of the three store classes (*Ratings*, *Movies*, and *Credits*) developed for the Warwick+ coursework project. I will explain the data structures I used, the reasoning behind my choices, how they helped me meet the coursework requirements efficiently, and reflect on possible improvements for the future.
The report also reflects on alternative approaches I considered and how my implementations evolved through testing and debugging.

Data Structures Overview

Across all three classes, I mainly used two custom-built data structures:

- **MyHashTable**: a simple hash table based on chaining with linked lists.
- **MyDynamicArray**: a manually implemented resizable array, similar to Java's built-in `ArrayList`.

These structures gave me fast access times, simplicity of implementation, and flexibility while staying within the coursework restrictions (i.e., not using Java's standard Collections framework).

They were lightweight but powerful enough to handle all the basic operations the Warwick+ project required, such as inserting, removing, finding, and listing many entries efficiently.

Ratings	**Movies**	**Credits**
movieID -> Ratings[]	**movieID -> Movie**	**filmID -> CreditRecord**
userID -> Ratings[]	**movieIDs[] Dynamic**	**uniqueCast[] Dynamic**

This summarizes the internal structure of the three classes and how they use hash tables and dynamic arrays to store the relationships between movies, users, cast, and crew efficiently.

Ratings Class

Structures Used

- `MyHashTable<Integer, MyDynamicArray<Rating>>` **movieRatings**
- `MyHashTable<Integer, MyDynamicArray<Rating>>` **userRatings**

Reasoning

The *Ratings* class needed to efficiently support two types of access:

- Find all ratings **for a movie**.
- Find all ratings **made by a user**.

A hash table was the best fit because it allowed **constant time O(1)** access to either movieID or userID. A dynamic array was used for storing multiple ratings associated with each movie or user because the number of ratings is unpredictable and could grow without an initial fixed limit.

Choosing these structures made operations like adding a rating, removing a rating, or computing averages simple and efficient.

Implementation

When a new rating is added, it is inserted into both `movieRatings` and `userRatings`.
The `set` method updates an existing rating if it finds one, otherwise it adds a new one.

Finding the most-rated movies or users involves counting entries in the hash table and sorting them with **bubble sort** — which, while not the most efficient, is a **stable** sort.

Strengths

- **Fast lookup**: Accessing ratings for a movie or user is extremely quick.
- **Low memory overhead**: Only non-empty entries are stored.
- **Simplicity**: Very little code duplication when adding, updating, or querying ratings.

Challenges

- **Consistency**: I had to ensure that any addition or removal updated both the movie-based and user-based hash tables.
- **Sorting**: Using bubble sort was straightforward but would not scale well for large datasets.

Movies Class

Structures Used

- `MyHashTable<Integer, Movie>` **movieTable**
- `MyDynamicArray<Integer>` **movieIDs**

Reasoning

I needed to:

- Quickly **find a movie** by its ID (for example, when setting ratings or popularity).
- **List all movies** for tasks like searching by title, checking collections, etc.

Using a hash table for movie lookup made sense because IDs were unique integers and the hash table provided **constant time access**.

Keeping a separate dynamic array of movieIDs made iteration (e.g., getting all movies, searching by title) easier and efficient without scanning the whole hash table.

Implementation

Each movie added to the store is stored in `movieTable`, and it's ID is recorded in `movieIDs`.

To avoid repeating similar code for each movie field (like title, overview, revenue, etc.), I created a generic `getField` helper that takes a lambda extractor.
This design kept the class **concise**, **clear**, and **less error-prone**.

Searching for films by a term (like title or overview) simply loops through the `movieIDs` array.

Strengths

- **Efficient movie lookup** using ID.
- **Order-preserving** listing of all movie IDs.
- **Clear separation** between random access (via hash table) and sequential access (via dynamic array).

Challenges

- Some operations like search involve a **linear scan** through all IDs, which could become slower if the database became extremely large.
- Movie objects contained many fields, increasing memory usage slightly.

Credits Class

Structures Used

- `MyHashTable<Integer, CreditRecord>` **creditRecords**
- `MyDynamicArray<CastCredit>` **uniqueCast**
- `MyDynamicArray<CrewCredit>` **uniqueCrew**

Reasoning

Credits management needed to:

- Quickly **find the cast and crew for a film**.
- **Find all films** for a cast/crew member.
- **Search people** globally (across all movies).

Using a hash table for mapping filmID to `CreditRecord` gave fast access to a movie's credits.
Using dynamic arrays for `uniqueCast` and `uniqueCrew` allowed flexible global searching and listing.

Implementation

Each film's cast and crew are stored together in a `CreditRecord` object linked to its filmID.

When adding credits, I made sure that no duplicate cast or crew members were inserted into the unique lists.
Insertion sort was used for tasks like:

- Sorting cast by **billing order**.
- Sorting crew members by **ID**.

When finding "stars" (cast members with billing order 1, 2, or 3), I scanned cast lists for matches.

Strengths

- **Efficient management** of movie-people relationships.
- **Clear division** between film-specific data and global data (unique people).
- **Lightweight structure** for searching and listing people.

Challenges

- **Searching** for cast or crew by name was linear time — which is acceptable for coursework but would be slow at real-world scale.
- **Maintaining uniqueness** required extra checks when adding cast/crew.

Why I Chose These Data Structures

MyHashTable

Hash tables were essential because:

- They offer **O(1) average lookup** time, compared to O(n) for arrays or lists.
- They allowed me to directly jump to the movie or user I was interested in without scanning.
- Implementing my own chaining-based hash table gave me flexibility over collision handling and load control.

In all three stores, IDs (integers) made hashing very simple and efficient.

MyDynamicArray

Dynamic arrays were chosen because:

- They **grow automatically** when needed without worrying about capacity.
- They offer **O(1)** random access by index.
- For operations like listing all movies, listing cast/crew, and sorting results, they were the simplest solution.
- Writing my own array allowed me to understand and control resizing policies.

Compared to linked lists, dynamic arrays have **better cache performance** and **faster access**, which suited this coursework better.

Possible Improvements

1. Hash Table Resizing

Currently, the hash table has a fixed capacity (2048).
If the number of stored entries grows too large, the hash table could become inefficient because of collisions.

An improvement would be to:

- **Track load factor** (e.g., size / capacity).
- **Resize** (double capacity) and **rehash** all entries once a threshold is exceeded.

This would maintain near O(1) access even for very large datasets.

2. Better Sorting Algorithms

Bubble sort is simple but inefficient ($O(n^2)$).

For production-scale data:

- A stable and efficient sort like **merge sort** (O(n log n)) would perform much better.
- For numeric sorts, **quick sort** could be used where stability is not required.

3. Indexing by Name

Searching by cast/crew name or movie title currently requires linear scans.

A better version would involve:

- Building a second hash table mapping **names** -> **IDs**.
- This would make searching for names **instant** (O(1)) instead of O(n).

This could especially help if the database contains thousands of movies and people.

4. Memory Efficiency

MyDynamicArray doubles its size during resizing.

This is fast, but wastes memory when only slightly over capacity.
Strategies like **increasing size by 1.5x** would save memory while still keeping resizing cost low.

5. Error Handling

Currently, invalid IDs return `null` or `-1`.

A more robust system would:

- Throw custom exceptions (e.g., `MovieNotFoundException`).
- Provide clearer API behavior for error cases.

This would make the store classes safer and more predictable to use.

Reflections

When designing these classes, my main goals were simplicity, correctness. I wanted to:

- Minimize unnecessary complexity.
- Write code that is easy to read, debug, and explain.
- Focus on functionality first, then minor optimizations.

I also realized during testing that good internal helpers (like getField) saved me a lot of repetitive work.
This project gave me a lot of hands-on experience with core data structure concepts like hashing, dynamic arrays, and sorting algorithms.

Conclusion

Overall, I am very happy with my design and implementation. The combination of hash tables and dynamic arrays allowed me to meet all the coursework functionality requirements, keep my code manageable, and achieve good performance within the limits of simple data structures.

If this project was expanded to a real-world application, I would definitely invest time into optimizing sort performance, improving lookup speeds, and reducing memory usage.
However, for Warwick+ coursework, I believe this balance of performance, simplicity, and correctness was the right choice.