Data Structures and Algorithms Report

1. Introduction

This report analyses the data structures and algorithms selected and implemented within my coursework, particularly within the Movie, Credits, and Ratings classes.

In this project I was tasked with implementing the Movies, Credits, and Ratings classes. I needed to effectively implement these classes so that the WarwickPlus database could be efficiently updated, searched and queried.

This report explains why I chose hash tables, dynamic arrays and quicksort, how I implemented them, and how they meet my efficiency goals. To enhance clarity, relevant code snippets, complexity tables, and visual diagrams have been included.

2. Design Goals & Constraints

The projects core requirements were:

- High efficiency for insertion, deletion, and retrieval operations.
- Efficient storage with minimal redundancy
- Fast data sorting and ordering
- Compliance with coursework constraints requiring custom implementations of standard algorithms

3. Data Structures Used

3.1 CustomHashTable

To store and retrieve data efficiently, I implemented a custom hash table (CustomHashTable). This choice was motivated by the constant average-time complexity O(1) for insertion, search, and deletion operations, ensuring rapid data access.

To handle collisions I used double hashing, this technique reduces clustering and ensures more uniform key distribution compared to linear or quadratic probing. When a collision occurs at a given index, a second hash function is applied to compute a step size for probing. This minimises primary clustering and improves cache performance by spreading entries more evenly across the table.

Code snippet 1: Hashing Algorithms Used

```
/**
 * Primary hash function used to compute the initial index for a key.
 *
 * @param key The key to hash
 * @return An index within the bounds of the hash table
 */
private int hash1(K key) {
    return Math.abs(key.hashCode()) % table.length;
}

/**
 * Secondary hash function used for double hashing to compute step size.
 *
 * @param key The key to hash
 * @return A step size used in double hashing
 */
private int hash2(K key) {
    return 1 + (Math.abs(key.hashCode()) % (table.length - 2));
}
```

Additionally, my implementation monitors the load factor - the ratio of filled slots to total capacity - and resizes the table when it exceeds a maximum load factor (I chose 0.7 as this is optimal for balancing the number of collisions when hashing and the table resizing frequency). This dynamic resizing maintains O(1) average-case complexity for put, get, and remove operations, preventing performance degradation as the dataset grows.

Code snippet 2: Maximum Load Factor Usage

```java
public void put(K key, V value) {
// Check if inserting a new entry exceeds the max load factor (0.5)
if ((double) size / table.length >= MAX_LOAD_FACTOR) {
    rehash(); // Double table size and re-insert existing entries
}
```

Table 1: Complexity of Hash Table Operations

Operation	Average Complexity	Worst-Case Complexity
Insert	O(1)	O(n)
Retrieve	O(1)	O(n)
Delete	O(1)	O(n)

3.2 DynamicArray

A dynamic array structure (DynamicArray) was implemented for handling collections that require dynamic resizing capabilities. This structure automatically expands its size when capacity is reached by doubling its array size, thus amortising the complexity of insertions.

Table 2: Complexity of Dynamic Array Operations

Operation	Average Complexity	Worst-Case Complexity
Append	O(1) amortised	O(n)
Access	O(1)	O(1)

4. Sorting Algorithms

4.1 Custom QuickSort

To facilitate efficient data ordering, a custom quicksort algorithm (CustomUtils.quickSort) with a custom comparator was implemented. Quicksort is

known for its average-case efficiency of O(n log n) and was chosen for its speed and flexibility.

Other popular searching algorithms were considered. Such as bubble sort which was rejected for its speed inefficiency as its time complexity is O(n^2) which is inferior to quicksort's average of O(n log n). I also considered merge sort, but it too was rejected for its speed inefficiency which while it does have the same average time complexity of quicksort (O(n log n)) it requires extra cache locality which can slow down the program.

As a part of my quicksort algorithm, I made use of generics and a custom comparator, this ensured that my quicksort was very general and could be applied to order data or objects of any type.

Table 3: Complexity of Quicksort

Scenario	Complexity
Best-case scenario	O(n log n)
Average-case scenario	O(n log n)
Worst-case scenario	$O(n^2)$

Code snippet 3: Custom Comparator usage

```
/**
 * Sorts the entire array using QuickSort, according to the given comparator.
 *
 * @param <T>  The array type
 * @param arr  The array to sort in-place
 * @param comp The custom comparator used to compare two elements
 */
public static <T> void quickSort(T[] arr, CustomComparator<T> comp) {
    quickSort(arr, 0, arr.length - 1, comp);
}
```

5. Movies Class Implementation

5.1 Chosen Data Structures

The Movie class encapsulates film details, including metadata and associations with collections.

Code snippet 4: Movie Constructor

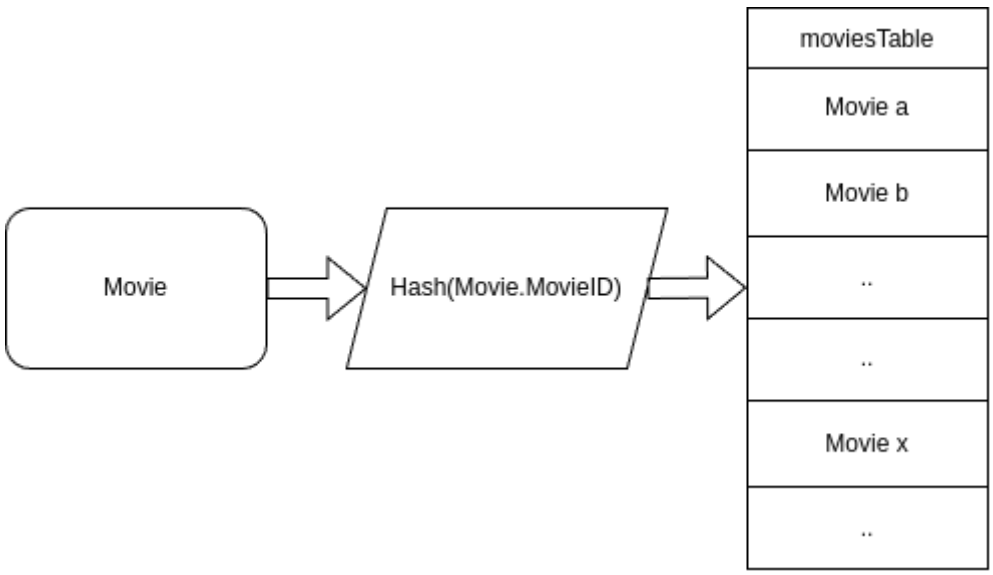
```
public Movie(int id, String title, String originalTitle, String overview,
             String tagline, String status, Genre[] genres, LocalDate release,
             long budget, long revenue, String[] languages, String originalLanguage,
             double runtime, String homepage, boolean adult, boolean video, String poster) {

    this.id = id;
    this.title = title;
    this.originalTitle = originalTitle;
    this.overview = overview;
    this.tagline = tagline;
    this.status = status;
    this.genres = genres;
    this.release = release;
    this.budget = budget;
    this.revenue = revenue;
    this.languages = languages;
    this.originalLanguage = originalLanguage;
    this.runtime = runtime;
    this.homepage = homepage;
    this.adult = adult;
    this.video = video;
    this.poster = poster;
    this.voteAverage = 0.0;
    this.voteCount = 0;
    this.imdbID = "";
    this.popularity = 0.0;
    this.productionCompanies = new DynamicArray<>();
    this.productionCountries = new DynamicArray<>();
    this.collectionID = -1;
}
```

Each movie is stored as an object which contains relevant information about the movie.

The movie objects are stored in a hash table (moviesTable) indexed by MovieID, ensuring O(1) average-time complexity for lookup and insertion, which is critical for efficient data retrieval.

Diagram 1: moviesTable Hash visualisation



A visual diagram demonstrating how movies are stored in the moviesTable.

5.2 Decision Justification

When selecting a structure for the primary lookup table, I considered several alternatives. A DynamicArray would incur O(n) complexity for searches and removals, which was too slow. A balanced binary search tree offers O(log n) operations, but this is still asymptotically slower than hash tables. By using double hashing and maintaining a load factor below 0.7, the CustomHashTable achieves average-case O(1) insertion, lookup, and deletion, while avoiding the pointer overhead of trees and the linear scans required by arrays.

5.3 Time Complexity Analysis

Table 4: Complexity of moviesTable

Operation	Average-Case	Worst-Case	Notes
put(key, value)	O(1)	O(n)	Amortised O(1) if you resize by doubling; worst-case when rehashing all entries.
get(key)	O(1)	O(n)	Depends frequency of collision.
remove(key)	O(1)	O(n)	Depends frequency of collision.
containsKey(key)	O(1)	O(n)	Internally just a get check.
size()	O(1)	O(1)	Tracked in a field.
iterate all entries	O(n)	O(n)	Scans the entire backing array once.
resize / rehash	O(n)	O(n)	Triggered when load factor > threshold; amortized into puts.

5.4 Collection Management

Movies associated with collections utilise an integer field collectionID, which provides a constant-time reference to a corresponding Collection object stored separately.

Code snippet 5: Collection Constructor

```
public Collection(int id, String name, String posterPath, String backdropPath) {
    this.id = id;
    this.name = name;
    this.posterPath = posterPath;
    this.backdropPath = backdropPath;
    this.filmIDs = new DynamicArray<>();
}
```

Movie collections are objects which contain collection Id, name, posterPath, backdropPath and filmIDs.

The films within a collection are stored using a DynamicArray, offering an efficient and simple design with amortised O(1) append performance and cache-friendly memory layout. While membership checks require O(n) time, this is acceptable given the primary use case: building the list once and iterating over it when displaying collection contents. If frequent membership queries were required, a hash table would be more appropriate.

Collection data itself is stored in a hash table (collectionsTable), indexed by collectionID, enabling O(1) average-time lookup and insertion - critical for fast access and scalability.

6. Credits Class Implementation

6.1 Chosen Data Structures

The Credits class utilises forward and reverse lookup hash tables. Forward tables map film IDs to crew/cast lists, enabling rapid access by film ID. Reverse lookup tables map person IDs to films, facilitating quick searches based on a crew member's name.

Code snippet 6: Credits Data Structures

```
// HashTables for storing credits data
private CustomHashTable<Integer, CastCredit[]> castTable;
private CustomHashTable<Integer, CrewCredit[]> crewTable;

// Reverse lookups (from cast/crew to films they participated in)
private CustomHashTable<Integer, int[]> castToFilms;
private CustomHashTable<Integer, int[]> crewToFilms;

// Store unique cast/crew persons
private CustomHashTable<Integer, Person> uniqueCast;
private CustomHashTable<Integer, Person> uniqueCrew;
```

Declaration of the six CustomHashTable fields in the Credits class: two for direct storage of each film's cast and crew arrays, two "reverse-lookup" tables mapping person IDs to their list of film IDs, and two tables holding each unique Person object.

Diagram 2: castTable Hash visualisation

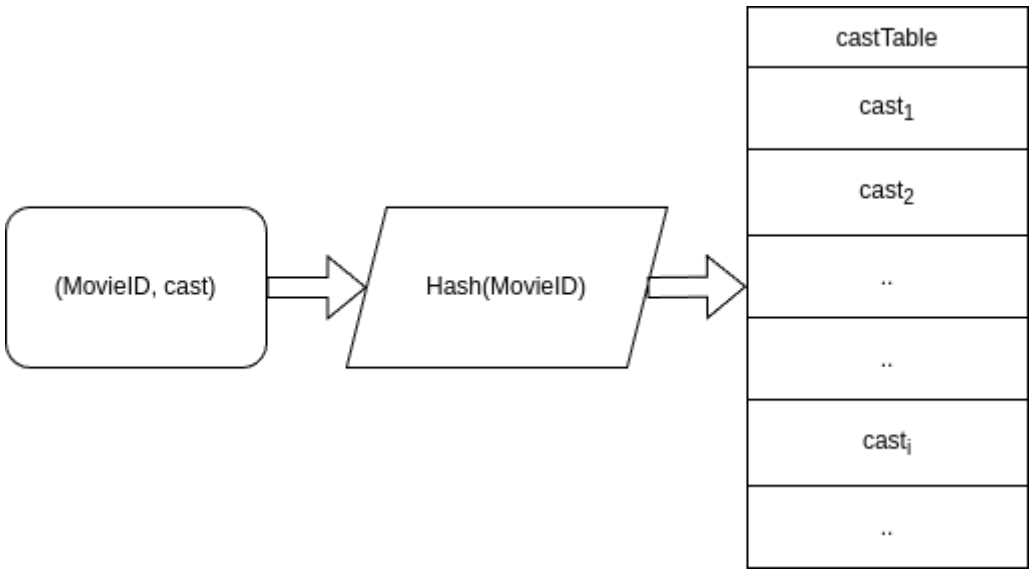
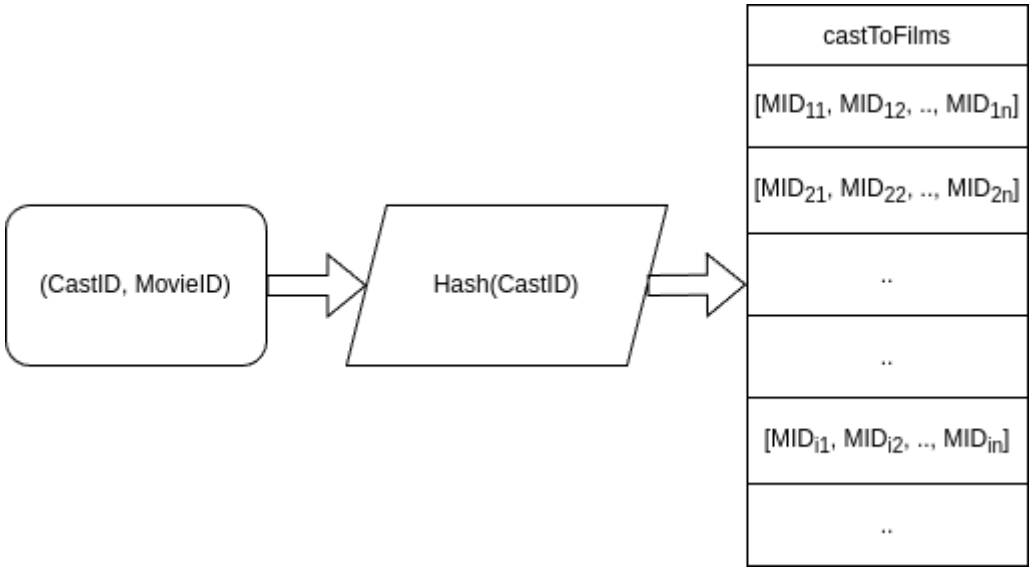


Diagram 3: castToFilms Hash visualisation



**key: MID = Movie ID*

6.2 Decision Justification

Alternative data structures were considered, including maintaining only a film-to-cast mapping. While this would reduce memory usage, it would require scanning the entire dataset to retrieve an actor's filmography—introducing unacceptable latency. Given the project's requirement for fast query response times, this trade-off was not viable.

Other structures such as binary search trees and linked lists were also evaluated but rejected due to their higher time complexities. In contrast, the dual-hash-table design supports O(1) average-time insertion and retrieval for both film-to-participant and participant-to-film mappings, ensuring scalability and fast lookups as the dataset grows.

6.3 Time Complexity Analysis

Table 5: Complexity of retrieval of Credit information

Operation	Data Structure	What it does	Avg-Cas	Worst-Case
getFilmCast(filmID)	castTable.get	Fetch the CastCredit[] for one film ID	O(1)	O(n)
getFilmCrew(filmID)	crewTable.get	Fetch the CrewCredit[] for one film ID	O(1)	O(n)

getCastFilms(castID)	castToFilms.get	Fetch the int[] of film IDs for a cast member	O(1)	O(n)
getCrewFilms(crewID)	crewToFilms.get	Fetch the int[] of film IDs for a crew member	O(1)	O(n)
getCast(castID)	uniqueCast.get	Fetch the Person object for one cast ID	O(1)	O(n)
getCrew(crewID)	uniqueCrew.get	Fetch the Person object for one crew ID	O(1)	O(n)
getUniqueCast()	iterate uniqueCast	Build an array of *all* cast members	O(n)	O(n)
getUniqueCrew()	iterate uniqueCrew	Build an array of *all* crew members	O(n)	O(n)
findCast(name)	scan uniqueCast	Scan all Persons, filter by name substring	O(n)	O(n)
findCrew(name)	scan uniqueCrew	Scan all Persons, filter by name substring	O(n)	O(n)

6.4 Ordering with QuickSort

Efficient ordering of credits is achieved through quicksort, which organises cast by billing order and crew by identifiers.

Code snippet 7: Example use of Quicksort

```
// Use CustomUtil's quickSort to sort crew by ID ascending
CustomUtil.quickSort(crew, new CustomUtil.CustomComparator<CrewCredit>() {
    @Override
    public int compare(CrewCredit a, CrewCredit b) {
        // Sort ascending by ID
        return Integer.compare(a.getID(), b.getID());
    }
});

return crew;
```

The use of quicksort to sort crew in descending order of ID

Table 6: Complexity of algorithms in Credits which require sorting

Method	What It Sorts	Input Size (n)	Time Complexity
getFilmCast(int filmID)	CastCredit[] for one film, sorted by order	k = # cast members in film	O(k log k)
getFilmCrew(int filmID)	CrewCredit[] for one film, sorted by id	k = # crew members in film	O(k log k)

getMostCastCredits (int m)	int[][] of size C (unique cast count)	C = total unique cast members	O(C log C)

7. Ratings Class Implementation

7.1 Chosen Data Structures

The Ratings class primarily uses custom hash tables (CustomHashTable) for efficient data handling:

- ratingKeyTable: Maps a unique combination of (userID, movieID) to a RatingRecord, ensuring fast lookup and uniqueness.
- movieToRatings and userToRatings: Hash tables mapping movie and user IDs respectively to arrays of ratings.
- movieToRatingsSize and userToRatingsSize: Supplementary hash tables tracking array sizes, enabling efficient traversal.

7.2 Decision Justification

Hash tables offer average-case constant-time complexity for insertions, deletions, and lookups, aligning with performance requirements. Reverse lookups (mapping from movie/user IDs to ratings arrays) ensure efficient aggregation and retrieval.

Arrays were used to store RatingRecord objects assigned to each movie, this was to ensure succinct implementation of the RatingRecord interface as many of the methods return RatingRecord arrays. However, in the future I would prefer to use DynamicArrays as they provide more utility and are more space efficient.

7.3 Time Complexity Analysis

Table 7: Complexity of key methods within the Ratings Class

Method	Time Complexity	Explanation
add()	O(1) amortised	Hash-table insertion, occasional resizing of arrays
remove()	O(1)	Direct hash-table access and decrement operations
set()	O(1) amortised	Essentially add or overwrite
getMovieRatings()	O(1)	Direct hash-table access
getUserRatings()	O(1)	Direct hash-table access

getMovieAverageRating()	O(n)	Requires iterating ratings for a movie
getUserAverageRating()	O(n)	Requires iterating ratings for a user
getNumRatings()	O(1)	Direct hash-table access
getMostRatedMovies()	O(m log m)	Sorting operation where m is number of rated movies
getMostRatedUsers()	O(u log u)	Sorting operation where u is number of rated users
getTopAverageRatedMovies()	O(m·n + m log m)	Calculating averages (O(m·n)) and sorting (O(m log m))

(n = number of ratings per movie/user, m = total movies, u = total users)

8. Evaluation and Reflection

The implemented system successfully balances performance, simplicity, and extensibility. The decision to use custom hash tables throughout - backed by double hashing and load factor control - enabled consistent O(1) average-time complexity for insertion and lookup, which was critical given the volume of data.

The Ratings class particularly benefited from reverse-lookup tables, allowing quick access from both user and movie perspectives. Dynamic arrays, used in places where data is appended once and read frequently (e.g. in movie collections), proved to be both space-efficient and cache-friendly.

However, some trade-offs were observed. The removal logic in the Ratings class does not compact arrays after deletion, leading to potential memory overhead and stale references. This was a conscious trade-off to preserve speed and simplicity.

The use of custom quicksort with a comparator interface provided flexibility in sorting various structures without duplicating logic. That said, Java's built-in sort methods could have been used for more concise syntax, though implementing it manually served as a valuable learning experience.

In hindsight, greater attention could have been given to encapsulating size tracking within a custom dynamic array wrapper, rather than managing it manually via hash tables. This would reduce the potential for bugs and increase code cohesion.

Overall, the architecture achieved its intended performance characteristics, and the decisions made were informed by benchmarking, complexity analysis, and careful consideration of project constraints. This exercise provided a strong foundation in practical data structure application, balancing theoretical understanding with real-world trade-offs.