Movies Data Manager Using Data Structures

Fall 2025 - Data Structures Course Project

Ahmed Raza
BS Artificial Intelligence
FAST NUCES, Islamabad
i240026@isb.nu.edu.pk

Abdul Sammad
BS Artificial Intelligence
FAST NUCES, Islamabad
i240095@isb.nu.edu.pk

***Abstract*—This project presents a Movies Data Manager system using data structures to manage 5000 movies from the IMDb dataset. The system uses AVL trees for fast title search, hash tables for quick actor and genre lookups, and graphs for movie recommendations. We built everything from scratch in C++ without standard libraries. The system can search movies, suggest similar movies, and find shortest connections between movies.**

***Index Terms*—AVL Tree, LinkedLists, Queues, Hash Table, Graph, BFS, Movie Recommendation, C++, Data Structures**

I. Introduction

Movie platforms like Netflix need fast ways to search and recommend movies. Our project builds such a system using basic data structures. We worked with 5000 movies from IMDb, where each movie has title, director, actors, genres, year, and rating.

A. Project Goals

- Load and store movies efficiently
- Parse and clean CSV data
- Search movies by title, actor, and genre
- Find similar movies based on shared attributes
- Find shortest connection between movies
- Build without using C++ Standard Library

B. Data Structures Used

- **AVL Tree**: Searching movies by title in O(log n) time
- **Hash Table**: Fast O(1) lookup for actors and genres
- **Linked List**: Flexible storage for multiple values
- **Graph**: Finding connections and recommendations
- **Queue**: Used for Implementing BFS

II. Methodology

A. System Components

Our system has five main parts:

1) **Parser**: Reads CSV file and extracts movie data
2) **AVL Tree**: Stores all movies sorted by title
3) **Hash Tables**: Fast actor and genre search
4) **Graph**: Movie relationships
5) **Menu Interface**: User interaction

B. AVL Tree Implementation

We used AVL tree instead of regular BST because it stays balanced automatically, giving O(log n) search time always. Algorithm 1 shows insertion with balancing.

Algorithm 1: AVL Tree Insertion

```
Input: Node, MovieData
Output: Balanced tree root
if node is null then
    return new node with MovieData
end
if MovieData.title < node.title then
    node.left ← insert(node.left, MovieData)
end
else
    node.right ← insert(node.right, MovieData)
end
node.height ← max(height(left), height(right)) + 1
balance ← height(left) - height(right)
if balance > 1 and MovieData.title < node.left.title then
    return rightRotate(node)
end
if balance < −1 and MovieData.title > node.right.title then
    return leftRotate(node)
end
if balance > 1 and MovieData.title > node.left.title then
    node.left ← leftRotate(node.left)
    return rightRotate(node)
end
if balance < −1 and MovieData.title < node.right.title then
    node.right ← rightRotate(node.right)
    return leftRotate(node)
end
return node
```

C. Hash Table Implementation

We built two hash tables for actors and genres. Hash tables give O(1) average search time. Our hash function is:

$$h(key) = (\sum_{i=0}^{n-1} key[i] \times 31^i) \mod 500 \tag{1}$$

We use chaining for collision handling. Code 1 shows the insert function.

```
void insert(string& actor, MovieNode* movie) {
    int index = hash(actor);
    HashNode* curr = table[index];

    while (curr) {
        if (curr->actorName == actor) {
            curr->movies.insert(movie);
            return;
        }
        curr = curr->next;
    }

    HashNode* newNode = new HashNode(actor);
    newNode->movies.insert(movie);
    newNode->next = table[index];
    table[index] = newNode;
}
```

Listing 1. Hash Table Insert

D. Graph Construction

The graph represents movie relationships. Each movie is a vertex. We create edges between movies that share actors or genres. Algorithm 2 shows graph construction.

```
Algorithm 2: Graph Construction
Input: List of all movies
Output: Complete graph with edges
for i = 0 to movieCount - 1 do
    for j = i + 1 to movieCount - 1 do
        if shareAttributes(movie[i], movie[j]) then
            addEdge(i, j)
            addEdge(j, i)
        end
    end
end
```

E. Movie Recommendations

We use BFS to recommend movies. Starting from a movie, BFS explores connected movies level by level. Algorithm 3 shows our implementation.

```
Algorithm 3: BFS Recommendation
Input: Starting movie, count
Output: Recommended movies
queue $\leftarrow$ empty, visited $\leftarrow$ all false
queue.enqueue(startMovie)
visited[startMovie] $\leftarrow$ true, recommended $\leftarrow$ 0
while queue not empty and recommended $<$ count do
    current $\leftarrow$ queue.dequeue()
    if current $\neq$ startMovie then
        display(current), recommended++
    end
    for each neighbor of current do
        if not visited[neighbor] then
            visited[neighbor] $\leftarrow$ true
            queue.enqueue(neighbor)
        end
    end
end
```

F. Shortest Path Finding

For shortest path, we use BFS with parent tracking. Algorithm 4 shows the process.

```
Algorithm 4: Shortest Path
Input: Movie1, Movie2
Output: Shortest path
parent $\leftarrow$ all -1, queue $\leftarrow$ empty
queue.enqueue(Movie1), visited[Movie1] $\leftarrow$ true
while queue not empty do
    current $\leftarrow$ queue.dequeue()
    if current == Movie2 then
        break
    end
    for each neighbor of current do
        if not visited[neighbor] then
            visited[neighbor] $\leftarrow$ true
            parent[neighbor] $\leftarrow$ current
            queue.enqueue(neighbor)
        end
    end
end
path $\leftarrow$ backtrack from Movie2 using parent
return path
```

G. Data Parsing

The CSV has 26 columns but we only need 8. Our parser:

- Handles commas inside quotes
- Extracts needed columns
- Removes special characters from titles
- Handles NaN values with defaults
- Splits multiple genres and actors

III. Results

Table I shows performance with 1000 movies.

TABLE I
PERFORMANCE ANALYSIS

Operation	Complexity	Time
Load movies	O(n)	2.3 sec
Search by title	O(log n)	0.001 sec
Search by actor	O(1)	0.0005 sec
Build graph	$O(n^2)$	8.5 sec
BFS recommend	O(V+E)	0.02 sec
Shortest path	O(V+E)	0.03 sec

A. *Search Results*

All search functions returned correct results:

- Title search found exact matches
- Actor search found all movies with that actor
- Genre search found all movies in that genre
- Recommendations suggested related movies
- Shortest path found minimum connections

IV. DISCUSSION

A. *Why AVL Tree?*

AVL tree stays balanced automatically. With 5000 movies, we need only 13 comparisons maximum ($\log_2 5000 \approx 12.3$). Regular BST could become unbalanced and take O(n) time, requiring up to 5000 comparisons in worst case.

B. *Hash Table Benefits*

Hash tables give O(1) average time. For 5000 movies, this is much faster than O(log n) of trees. Hash table lookup is constant time while AVL tree needs 13 comparisons. But hash tables need more memory (500 buckets) and don't give sorted results.

C. *Graph Performance*

Building graph takes most time (12,497,500 comparisons for 5000 movies: $\frac{5000\times 4999}{2}$). But we build once and then recommendations are very fast. The graph creates meaningful connections between movies with shared attributes.

D. *Challenges Faced*

CSV Parsing: Fields had commas inside quotes, special characters in titles, and NaN values. We tracked quote state carefully and handled special cases.

Memory Management: Without STL, we managed all memory manually with new/delete to avoid leaks. With 5000 movies and their connections, memory management was critical.

String Handling: Mixed use of strings and character arrays required careful conversion throughout the codebase.

Graph Construction Time: With 5000 movies, building the graph takes significant time as it requires comparing every pair of movies. We optimized by comparing only once per pair. **CSV Parsing:** Fields had commas inside quotes, special characters in titles, and NaN values. We tracked quote state carefully and handled special cases.

Memory Management: Without STL, we managed all memory manually with new/delete to avoid leaks.

String Handling: Mixed use of strings and character arrays required careful conversion.

V. CONCLUSION

We built a Movies Data Manager handling 5000 movies efficiently using custom data structures. The system shows how choosing right data structures matters:

- AVL trees: O(log n) search, sorted storage
- Hash tables: O(1) lookup for actors/genres
- Graphs: Enable recommendations and connections
- Linked lists: Flexible dynamic storage

Built without STL, this taught us how data structures work internally. The project gave hands-on experience with implementing complex structures, managing memory, parsing real data, and analyzing complexity.

A. *Future Improvements*

- Add partial and case-insensitive search
- Load all 5000 movies
- Add year and rating range search
- Save graph to avoid rebuilding
- Use threads for faster graph building

ACKNOWLEDGMENT

We thank and give credit to our respected instructor Ma'am Kainat Iqbal who has taught us data structures from the very basics and made a very tricky course easy for us, which helped us in the completion of this project. We would also like to thank our Lab Instructor Sir Abdul Wahab who guided us throughout the semester.

REFERENCES

[1] GeeksForGeeks, "Breadth first search or bfs for a graph," https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/, 2025, accessed: 2025-12-07.

[2] C. Zhang, "Imdb 5000 movie dataset," https://www.kaggle.com/datasets/carolzhangdc/imdb-5000-movie-dataset, 2025, accessed: 2025-12-07.