Smart Library Management & Recommendation System

Course Project (Project 1)
Language: C++ (custom data structures, CLI, file I/O persistence)
Report date: 2025-12-31

This report explains what the system does, how it is designed, how the data structures work, how files are stored/loaded, and how to demonstrate the project during viva.

1. Executive summary

We built a campus library CLI application with two roles (Admin and User). The core catalog uses a custom open-addressing hash table keyed by ISBN for fast lookup. Users can search books by ISBN or by text (title/author), borrow/return books, maintain history, and receive recommendations (Popularity top-k via MaxHeap, Co-borrow BFS graph, and Genre-based). Data persists across runs using text files.

2. Features (what the system can do)

- Admin login (single admin account) with add/update books and add users.
- User login by user ID.
- Catalog viewing (alphabetical listing) with limited details for users and full inventory for admin.
- Search by ISBN (hash lookup) or Search by Title/Author (string processing + suggestions).
- Borrow and Return with per-book waitlist (FIFO) using a custom queue.
- Borrow limits: one user cannot borrow the same book twice; a user cannot be added to the same waitlist twice.
- Recommendations when a book is unavailable: Popularity (heap), Co-borrow BFS, Genre-based list.
- File persistence: books, users, borrowed list, history log, and waitlists.

3. Architecture and module responsibilities

The project is split into small header modules. The main.cpp acts as the CLI controller: it loads files at startup, runs login/menu loops, calls the engines/managers, and saves changes.

Module / File	Responsibility (high level)
book.h	Book record (ISBN, title, author, genre, copies, popularity) + per-book waitlist
hashtable.h	Catalog: custom open-addressing hash table mapping ISBN → Book.
linkedlist.h	Generic singly linked list used for history and current borrowed list.
queue.h	Custom FIFO queue used for each book’s waitlist.

user.h	User record (userId, name) + history list + currently-borrowed list.
usermanager.h	Stores users in a fixed array; add/find user; helpers for history/borrowed upda
borrowengine.h	Borrow/return rules, waitlist handling, popularity increment, borrowed-list enfo
maxheap.h	Custom max-heap storing pointers to books for popularity top-k retrieval.
PBrecommendationengine.h	Popularity recommendations (build heap, extract top-k available books).
BFSrecommendationengine.h	Co-borrow graph + BFS traversal for recommendations (no STL).
GenreRecommendation.h	Genre → ISBN list (array-based table) for genre recommendations.
searchengine.h	Text search by title/author with normalization/tokenization + “Did you mean”
main.cpp	CLI + file I/O (load/save), login menus, and calling all modules.

4. Data structures and algorithms

4.1 Catalog hash table (open addressing + linear probing)

The catalog uses an array of Book records. A book’s ISBN is hashed to an index. If a collision happens, the table linearly probes the next cells until an empty slot is found. Search also probes linearly but starts from the hashed index; it stops when an empty cell is hit or the table cycles.

Resizing: when load factor (size/capacity) exceeds 0.7, capacity doubles and all books are reinserted into the new array. This keeps average operations close to O(1).

4.2 Linked lists (history and currently-borrowed)

We use a singly linked list for two user-related lists: (1) history: every borrowed book ISBN is appended/inserted so we can display what the user borrowed; (2) borrowed: active books currently borrowed by the user. The borrowed list is used to enforce the rule “a user can’t borrow the same book twice.”

4.3 Custom queue (per-book waitlist)

Each Book has a FIFO queue of user IDs. If no copies are available, the requesting user is added to the queue. When a copy is returned, the first user in the queue automatically gets the book. A helper check prevents the same user from being added to the same waitlist multiple times.

4.4 Popularity recommendations (MaxHeap)

To recommend the top-k most popular books, we push Book pointers into a max-heap ordered by popularityCount. Then we extractMax() k times. We also filter out books that are currently unavailable so the user sees borrowable suggestions.

Complexity: building heap O(n), extracting k books O(k log n).

4.5 Co-borrow recommendations (BFS graph)

We store a simple adjacency list using fixed arrays: isbnList[] maps ISBN → index; neighbors[index][] stores up to a small number of related ISBNs. Edges are added when books are borrowed together (from history sequences). BFS prints a traversal order of related books starting from a chosen ISBN.

Complexity: O(V+E) over the explored part of the graph.

4.6 Genre recommendations (array table)

GenreRecommendation keeps an array of genres and for each genre an array of ISBNs belonging to that genre. When a book is inserted/loaded, its genre bucket is updated. For recommendations, we print all books in the same genre.

4.7 Search by title/author with string processing

SearchEngine accepts a user query string. It normalizes it (lowercase, trims spaces), tokenizes by spaces, and compares against normalized book titles/authors. It supports: • Exact match (normalized) • Substring match (e.g., 'kill' matches 'To Kill a Mockingbird') • Token overlap scoring (more matching tokens = higher score) If no direct match is found, it prints “Did you mean” suggestions. The user can then select which suggested book to borrow/return.

5. File persistence (File I/O)

All program data is stored in simple text files so the system can close and reopen without losing state. Loaders are defensive: they create missing files, trim whitespace, and use safeStoi to avoid crashing on bad lines.

File	Purpose	Format (each line)
books.txt	Catalog + inventory + popularity	isbn|title|author|genre|totalCopies|availableCopies|popularity
users.txt	Registered users	userId|name
history.txt	Borrowing history (log)	userId|isbn|isbn|isbn ...
borrowed.txt	Currently borrowed (active)	userId|isbn|isbn|isbn ...
waitlist.txt	Per-book FIFO waitlist	isbn|userId|userId|userId ...

Important design decision: history vs borrowed

history.txt is a log: it stores what the user borrowed over time and never removes entries. borrowed.txt stores the active state: what the user currently has checked-out. Return operations remove from borrowed but do NOT add duplicate entries to history.

6. CLI workflow (what to show in demo/viva)

The CLI uses a role-based flow. First you choose Admin or User. Admin can manage data; User can search and borrow/return.

6.1 Login menu

Admin login uses one fixed credential (username: admin, password: 1234). User login uses only userId.

6.2 Admin menu

- Add/Update Book (ISBN collision updates existing record).
- View Full Book Inventory (alphabetical list with copies and popularity).
- Add User (ID uniqueness enforced).
- View All Users.
- Logout.

6.3 User menu

- View All Books (alphabetical list, only title + ISBN).
- Search Book: choose ISBN search (fast hash) or Title/Author search (SearchEngine suggestions).
- Borrow Book: uses SearchEngine selection, then BorrowEngine rules.
- Return Book: user selects from their borrowed list and the system processes the waitlist.
- View Your History.
- Logout.

6.4 Borrow/Return rules to explain in viva

- If availableCopies > 0: borrow succeeds, availableCopies--, user history updated, popularityCount++ and borrowed list updated.
- If not available: user is added to the book waitlist (if not already queued) and recommendations are displayed.
- Return: if waitlist not empty, the first queued user gets the book (their borrowed list is updated). Otherwise availableCopies++.
- A user cannot borrow the same ISBN twice; they also cannot be queued twice for the same ISBN.

7. Complexity summary

Operation	Target / Actual complexity (typical)
Search by ISBN	O(1) average (hash), O(n) worst-case if many collisions
Insert/Update book	O(1) average; resize reinsertion O(n) when triggered
Search by title/author	O(n * L) over all books (string checks), where L is text length
Borrow/Return	O(1) average + waitlist check; duplicate checks depend on queue/borro
Top-k popularity	O(n) build heap + O(k log n) extract
BFS recommendation	O(V + E) over visited nodes/edges
Genre recommendation	O(g) where g = books in that genre bucket

8. Testing plan (what to mention)

Testing is a mix of unit-level checks and integration tests through the CLI. Even if you do not have a full automated test suite, you can explain that you tested the following cases:

- Insert and search many ISBNs; verify collisions still allow correct search.
- Borrow when copies exist; verify availableCopies decreases and borrowed/history updates.
- Borrow when no copies exist; verify user joins waitlist and is not duplicated.
- Return triggers waitlist: first queued user receives book automatically.
- Popularity recommendations return top-k available books and are stable under ties.
- SearchEngine: exact match, partial match, uppercase/lowercase, and suggestion flow.

9. How to run (demo commands)

In VS Code with MinGW g++ (Windows), open terminal in the project folder and run:

```
g++ -std=c++11 main.cpp -o library ./library
```

If your compiler outputs a.exe by default, you can also run: ./a.exe.

10. Sample dataset (clean reset)

If your files become corrupted during demo/testing, you can replace them with the following clean sample data.

books.txt

```
111|The Silent Patient|Alex Michaelides|Thriller|2|1|5
112|Atomic Habits|James Clear|SelfHelp|3|2|8
113|The Alchemist|Paulo Coelho|Fiction|2|2|6
114|1984|George Orwell|Classics|2|1|7
115|To Kill a Mockingbird|Harper Lee|Classics|1|0|10
116|Deep Work|Cal Newport|Productivity|2|2|4
117|The Hobbit|J.R.R. Tolkien|Fantasy|2|1|9
118|Harry Potter and the Sorcerer's Stone|J.K. Rowling|Fantasy|3|2|12
119|Sapiens|Yuval Noah Harari|NonFiction|2|2|3
120|The Great Gatsby|F. Scott Fitzgerald|Classics|2|2|2
```

users.txt

```
1|Zaid
2|Ibrahim
3|Fatima
4|Yousuf
5|Ali
```

borrowed.txt

```
1|112
2|117
3|116
4|118
5|115
```

history.txt

```
1|113|112|114
2|118|117
3|119|116|112
4|111|118|117
5|115|113
```

waitlist.txt

```
115|2|1
```

11. Viva cheat sheet (how to explain quickly)

If you get asked “why hash table if you still probe linearly?”, explain: the formula gives a starting index; probing only happens when collisions occur. Average search remains O(1) because most lookups find the key after very few probes.

- Start from requirements: fast search, borrow/return, waitlist, recommendations, persistence.
- Explain data structure mapping: hash table (catalog), queue (waitlist), linked list (history/borrowed), heap (top-k), graph+BFS (co-borrow), genre table (genre → ISBNs).
- Explain one end-to-end scenario: user searches title → selects book → borrow succeeds or joins waitlist → return triggers next user.
- Mention safety: safe file parsing, trimming, stoi guarding, preventing duplicate borrow / duplicate queue entries.