**LibraryApp Project Report**

**1. Introduction and Project Description**

For my project, I developed **LibraryApp**, a simple console-based library management system using C# and .NET. The app stores information about resources like books and journals, including ID, title, author, publication year, genre, availability status, and due date. My goal was to practice core programming concepts—custom data structures, database integration with Entity Framework Core, and basic user interfaces—while meeting the assignment’s functional and technical requirements. I didn’t have much experience with these and C#, but I feel like doing this project, and being forced to learn how to use these concepts has made me much more confident now, and a better programmer in general.

**2. Design and Data Structure Choice**

To manage resources in memory, I implemented an **array-backed list** (a fixed-size Resource[] plus a count variable). I chose this structure because:

* **Simplicity**: As a beginner, it was easy to understand and implement without relying on built-in collections.
* **Predictable behavior**: Adding and removing items involves straightforward index operations and loops.
* **Educational value**: I could demonstrate manual memory management (tracking count) and simple algorithms.

**Basic Operations (Pseudo-code)**

Add(r): Remove(id): SearchByTitle(title):

if count < capacity for i from 0 to count-1 for i from 0 to count-1

items[count] = r if items[i].Id == id if items[i].Title == title

count++ items[i] = items[count-1] return items[i]

count-- return null

return false

**3. Time-Complexity Analysis**

* **Add**: O(1) — inserts the new resource at the end of the array in constant time.
* **Remove**: O(n) — in the worst case it scans every element to find the matching ID, then swaps and decrements count.
* **Search (by title/author/genre)**: O(n) — linear scan checking each element’s field.
* **Reports (overdue/genre)**: O(n) — iterates through all stored items.

These complexities are acceptable for small libraries; for larger datasets, a more advanced structure (like a tree or hash table) would improve performance.

**4. Testing Approach and Test Cases**

I tested the app both manually and with automated unit tests:

**Manual testing:**

* Added a large number of sample books via the console UI to the database and verified they appeared on restart.
* Removed several items by ID and confirmed they no longer loaded or appeared in search results.
* Searched by title, author, and genre to ensure accurate matches, including tests for non-matching queries.
* Checked out and returned items, verifying IsAvailable toggles and DueDate updates correctly.
* Ran the overdue report and genre report commands to confirm correct filtering.

**Automated unit tests (xUnit):**

* Add\_IncrementsCount: verifies that Add increases Count by 1.
* Remove\_ExistingId: ensures removing an existing ID returns true and decrements Count.
* Remove\_NonexistentId: checks removing a missing ID returns false without changing Count.
* SearchByTitle\_Found / SearchByTitle\_NotFound: tests successful and unsuccessful title lookups.

These combined tests caught edge cases early and documented the expected behavior of each operation.

**5. Conclusion and Reflection**

Overall, creating LibraryApp was a valuable learning experience. I’m proud that the console app now supports all the main operations: adding, removing, and searching for resources, checking items in and out, and generating simple reports for overdue items and by genre.

Designing my own array-backed list taught me a lot about how data structures work under the hood—tracking a count, shifting elements, and writing loops to search or modify the collection. Integrating with EF Core was a great introduction to object-relational mapping: I saw how my C# classes translate into database tables, and how easy it is to query and save data without writing raw SQL.

That said, the application isn’t perfect. Because most operations loop through every item, performance would slow down if the library grew very large. The console interface is also quite basic—there’s no graphical display or navigation beyond text prompts.

If I had more time, I’d explore:

* **A nicer UI:** A simple WinForms or WPF interface to make it more user-friendly and visually appealing.
* **Improved data structures:** Switching to a hash table or balanced tree for faster search and removal (O(log n) or even O(1) on average).
* **More robust reports:** Exporting to CSV or PDF, and adding filters like date ranges or categories.

Overall, this project solidified my understanding of C# fundamentals, data structures, and database integration, and gave me confidence to tackle more complex applications in the future.

## References

I relied on an AI assistant on some parts of this project, especially since it was my first time building a .NET console app with EF Core and a custom data structure. AI helped me in several ways:

1. **Understand core concepts**  
   – When I was unsure how EF Core migrations worked, I asked for step-by-step explanations and even coding puzzles to build my confidence.
2. **Write code snippets**  
   – I got guidance on implementing the array-backed list in LibraryCollection.cs, and later AI helped troubleshoot my LINQ-based OverdueReport when I ran into errors.
3. **Draft documentation**  
   – AI generated an initial README template that I then simplified and personalized.
4. **Debug issues**  
   – When Visual Studio’s “View Data” featured an error, AI suggested the fixes and explained how those features work.

Overall, AI saved me many hours of research and gave me examples to adapt, but I always wrote and integrated the final code myself.