📘 System Design Document

**Title:** School Management System – Design Document

Architecture Overview

The system is composed of five independent modules that interact through shared identifiers:

* **Student ID**: Used across all modules to identify and link student records.
* **Course Code**: Used by CourseScheduler and AnalyticsModule to manage enrollment and grades.
* **ISBN**: Used by LibrarySystem to manage book inventory and borrowing.

Module Interactions

|  |  |  |
| --- | --- | --- |
| Module A | Shares data with | Shared data |
| studentregistry | All modules | Studentid,name |
| courseScheduler | Analytics module | Coursecode,studentid |
| feetracker | studentregistry | studentid |
| Library system | studentregistry | studentid |
| Analytics module | Studentregistry,course scheduler | Studentid,coursecode,grade |

Each module is self-contained but uses shared keys to maintain consistency and enable cross-module operations.

Data Structure Justification

**Why HashMap for StudentRegistry?**

HashMap Advantages

* **Constant-Time Lookup (O(1))**: A HashMap allows direct access to a student record using their unique ID. This is ideal for operations like findStudent(studentId) which are frequent and must be fast.
* **Efficient Updates**: Adding or updating a student record is also O(1), making the system responsive even with thousands of students.
* **Unique Keys**: HashMap enforces uniqueness of student IDs, preventing accidental duplicates.

❌ Why Not LinkedList?

* **Linear Search (O(n))**: A LinkedList requires scanning each node to find a student by ID, which becomes inefficient as the number of students grows.
* **No Key-Based Access**: LinkedLists are ordered collections, not key-value stores. They don’t support direct lookup by ID.
* **Higher Memory Overhead for Search**: Searching through a LinkedList consumes more CPU cycles and memory bandwidth compared to a HashMap.

**Why ArrayList and Queue for CourseScheduler?**

**Dynamic Sizing**: ArrayList and Queue in Java automatically resize as needed. This is ideal for a school system where the number of students per course may vary or grow unexpectedly.

* **Ease of Use**: Java’s built-in ArrayList and LinkedList (used for queues) provide simple APIs for adding, removing, and iterating over students.
* **Separation of Roles**:
* ArrayList is used for enrolled students (order matters).
* Queue is used for waitlisted students (FIFO behavior).

❌ Why Not Circular Array

* **Fixed Capacity**: Circular arrays require a predefined size. In a real-world school system, course enrollment can fluctuate, and fixed-size structures are too rigid.
* **Manual Management**: Implementing a circular array requires manual tracking of head and tail pointers, wrap-around logic, and overflow conditions — increasing complexity.
* **Limited Flexibility**: Circular arrays are ideal for low-level systems (e.g., buffering in embedded systems), not for high-level dynamic enrollment logic.

Why Binary Search Tree for FeeTracker?

Excellent question — and a great opportunity to show your understanding of data structure trade-offs in your system design document. Here's how you can write this section clearly and professionally:

💰 Why Binary Search Tree (BST) Instead of AVL Tree in Fee Tracking

In the **FeeTracker** module, we use a standard Binary Search Tree (BST) to store student payment records chronologically. Each node contains the student ID, payment amount, and date. While an AVL tree is a self-balancing variant of BST, we chose a regular BST for the following reasons:

✅ Why BST Was Chosen

* **Simplicity**: A regular BST is easier to implement and maintain, especially for a class project. It avoids the complexity of rotations and balance factors.
* **Chronological Ordering**: Payments are inserted based on date, and in-order traversal of the BST naturally yields payments in chronological order.
* **Expected Balanced Behavior**: In typical school scenarios, payment dates are spread out reasonably, so the tree remains relatively balanced without needing strict enforcement.
* **Low Overhead**: BSTs have lower memory and computational overhead compared to AVL trees, which store additional balance information and perform rotations.

❌ Why Not AVL Tree

* **Unnecessary Balancing**: AVL trees enforce strict balance after every insertion and deletion. For fee tracking, where insertions are mostly sequential by date and deletions are rare, this adds complexity without significant benefit.
* **Higher Maintenance Cost**: AVL trees require additional logic for rotations and balance factor updates, which increases code complexity and runtime overhead.
* **No Performance Gain for In-Order Traversal**: Since we only need sorted output by date, a regular BST already provides this efficiently.

Why Deque for LibrarySystem?

 **LIFO Behavior**: A stack naturally models the borrowing logic — the most recent borrower is the one expected to return the book first.

* **Simple Return Validation**: When a student returns a book, we check if they are at the top of the stack. If so, the return is valid.
* **Efficient Operations**: Stack operations (push, pop, peek) are O(1), making them fast and predictable.
* **Chronological Tracking**: The stack maintains the exact order of borrowers, which is useful for audit trails or resolving disputes.

Why Not HashMap

* **No Order Guarantee**: A HashMap stores key-value pairs but does not preserve the order of borrowing.
* **Complex Return Logic**: With a HashMap, validating who should return the book first would require additional logic and possibly scanning all entries.
* **Unnecessary Overhead**: HashMap is optimized for key-based lookup, which isn’t needed when tracking a simple borrow-return sequence.

Why heap for analytical model?

 **Efficient Top-K Ranking**: A min-heap allows us to maintain the top N performers in O(n log k) time, which is ideal when ranking students by grade.

* **Automatic Ordering**: The heap automatically keeps the lowest grade at the root, making it easy to discard lower performers as we iterate.
* **Memory Efficiency**: We only store the top N students in the heap, rather than sorting the entire dataset.
* **Built-in Support**: Java’s PriorityQueue provides a clean API for heap operations without manual implementation.

Why Not a Matrix

* **Sparse Data**: Not every student takes every course. A matrix (e.g., students × courses) would contain many empty cells, wasting memory.
* **Rigid Structure**: Matrices require fixed dimensions, making it harder to add new students or courses dynamically.
* **Poor Fit for Ranking**: Extracting top performers from a matrix would require scanning entire rows or columns and sorting — less efficient than a heap.

Why Not a Graph

* **No Relationships to Model**: Graphs are ideal for modeling connections (e.g., friendships, dependencies). Grades are independent values — there’s no edge or path between students or courses.
* **Unnecessary Complexity**: Implementing nodes and edges adds overhead without solving a relevant problem in this context.
* **No Traversal Needed**: We don’t need to find paths, cycles, or connectivity — just rank numeric values.

🔁 Flow Diagrams / Pseudocode

1. Student Enrollment Flow

[StudentRegistry] --> findStudent(studentId)

|

v

[CourseScheduler] --> enrollStudent(student, courseCode)

|

v

If capacity full --> add to waitlist

1. Fee Recording Flow

recordPayment(studentId, amount, date)

|

v

Insert into BST by date

|

v

generateReport(studentId)

|

v

In-order traversal of BST

1. Library Borrowing Flow

borrowBook(isbn, studentId)

|

v

Check availability

|

v

If available --> mark unavailable + push studentId to stack

Else --> show "already borrowed"

4. Performance Ranking Flow

recordGrade(studentId, courseCode, grade)

|

v

Store in HashMap<studentId, Map<courseCode, grade>>

getTopPerformers(courseCode, topN)

|

v

Use PriorityQueue to rank top N students