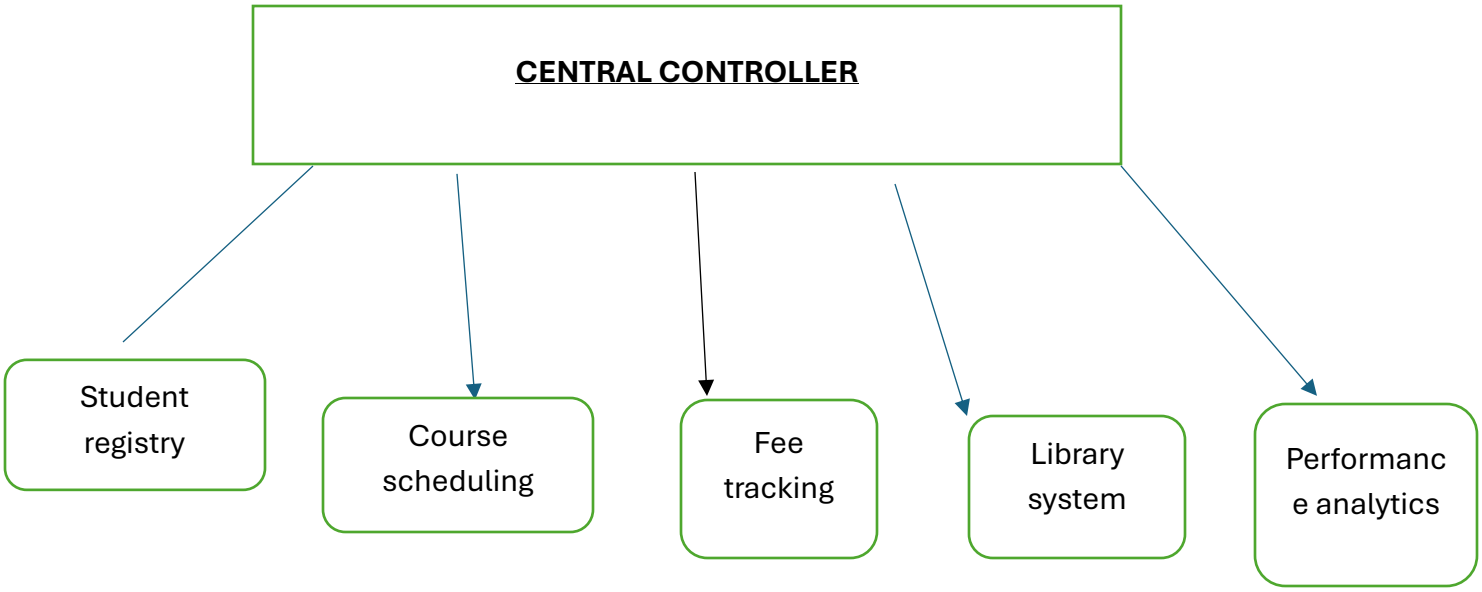



The system adopts a modular architecture with a **Central Controller** that manages communication between five key modules: Student Registry, Course Scheduling, Fee Tracking, Library Management, and Performance Analytics.
Each module maintains its data independently but exchanges data through a shared interface.

## 3. Module Design and Data Structure Justification

Each module section includes:

1. **Module Name**
2. **Data Structure(s) Used**
3. **Justification (Why chosen)**
4. **Key Operations (and Complexity)**

### 3.1 Student Registry

5. Data Structure:
   **Hash Table + Linked List**
   Why: Hash tables allow O(1) lookup for student IDs; linked lists handle sequential traversal for reporting.
   Operations:
6. Add student → O(1)
7. Delete student → O(1) average

8. Search student → O(1)

**3.2 Course Scheduling**

9. **Data Structure:** Queue + Circular Array
   **Why:** A queue ensures fairness in registration order; circular array avoids overflow and efficiently reuses slots.
   **Operations:**
10. Enqueue student → O(1)
11. Dequeue student → O(1)
12. View waitlist → O(n)

**13. 3.3 Fee Tracking**

14. **Data Structure:** Binary Search Tree (BST) / AVL Tree
    **Why:** Keeps student payments sorted by ID or date for quick searching and reporting. AVL ensures balance for consistent O(log n) time.
    **Operations:**
15. Add payment → O(log n)
16. Search by student ID → O(log n)
17. Generate fee report → O(n)

**18. 3.4 Library Management**

19. **Data Structure:** Stack + Hash Map
    **Why:** Stack manages borrowing/return order (LIFO). Hash map gives O(1) book lookup by ISBN.
    **Operations:**
20. Borrow book → Push stack (O(1))
21. Return book → Pop stack (O(1))

Check availability → Hash lookup (O(1))

**Performance Analytics**

**Data Structure:** Graph + Matrix + Heap
**Why:** Graph models relationships between students and subjects; matrix stores grades; heap finds top performers efficiently.
**Operations:**

1. Insert grade → O(1)
2. Analyze top students → O(log n)
3. Generate performance graph → O(V + E)

**DATA STRUCTURE JUSTIFACTION**
**Student Registry**
**Chosen Data Structures:**

**Hash Table** – for quick student lookup by ID

**Linked List** – for sequential traversal and record maintenance
**Justification:**
The Student Registry handles frequent **insertions**, **deletions**, and **searches** of student records.
A **hash table** provides **O(1)** average time complexity for lookups, ensuring fast retrieval using student IDs.
A **linked list** is used alongside the hash table to maintain an ordered list of students for operations like generating reports or processing all records sequentially.

**Course Scheduling**

**Chosen Data Structures:**

**Queue** – for managing student registration order

**Circular Array** – for efficient memory use and continuous scheduling

**Justification:**
Course allocation must treat all students fairly, following the **first-come, first-served** principle.
A **queue** perfectly models this behavior, as it enqueues students in order of registration and dequeues them for course assignment.
A **circular array** is used to implement the queue efficiently — allowing reuse of array slots once a student is processed, preventing overflow and saving memory.
Operations like enqueue and dequeue both run in **O(1)** time.

**Fee Tracking**

**Chosen Data Structures:**

**Binary Search Tree (BST)** – for maintaining sorted payment records

**AVL Tree** – for keeping the tree balanced

**Justification:**
The fee module requires frequent **searches** (to verify payments), **insertions** (new payments), and **sorted output** (for reports).
A **Binary Search Tree** allows sorted storage and efficient range queries.
However, unbalanced trees degrade to O(n) performance; hence an **AVL Tree** is used to maintain balance automatically, ensuring **O(log n)** time complexity for all major operations (insert, delete, search).
This design provides fast access and accurate reporting.

**Library Management**

**Chosen Data Structures:**

**Stack** – for managing borrowing and returning order

**Hash Map** – for fast ISBN lookups

**Justification:**
The library system tracks book loans and returns, which naturally follow a **Last-In-First-Out (LIFO)** pattern — the most recently borrowed book is the next to be returned.
A **stack** effectively models this behavior.
A **hash map** is used to quickly verify whether a book is available by **ISBN** or to check who borrowed it.
Hash maps provide **O(1)** average lookup time, ensuring quick responses to user queries.
Together, these structures make library operations efficient and reliable.

**Performance Analytics**

**Chosen Data Structures:**

**Graph** – for representing relationships between students, courses, and grades

**Matrix** – for storing numerical grades efficiently

**Heap (Priority Queue)** – for ranking top performers

**Justification:**
Performance analysis involves multiple entities (students, subjects, scores) and

relationships between them, making a **graph** a natural choice for representing these connections.
A **matrix** (2D array) stores grades in tabular form for quick access and computations (O(1) access time).
A **heap** or **priority queue** efficiently retrieves top-performing students in **O(log n)** time.
This combination supports both large-scale analysis and efficient ranking

## FLOWDIAGRAM AND PSEUDOCODE

**Student Registration Process**

🧠 ***Logic / Description:***

Registers new students by collecting their details, generating a unique ID, and inserting their record into a hash table and linked list.

**Pseudocode**

START

Input: student_name, student_ID, department, contact_info

Compute hash_index = hash(student_ID)

IF hash_table[hash_index] is empty THEN

  Insert new student record at hash_table[hash_index]

ELSE

  Append record to linked list at hash_table[hash_index]

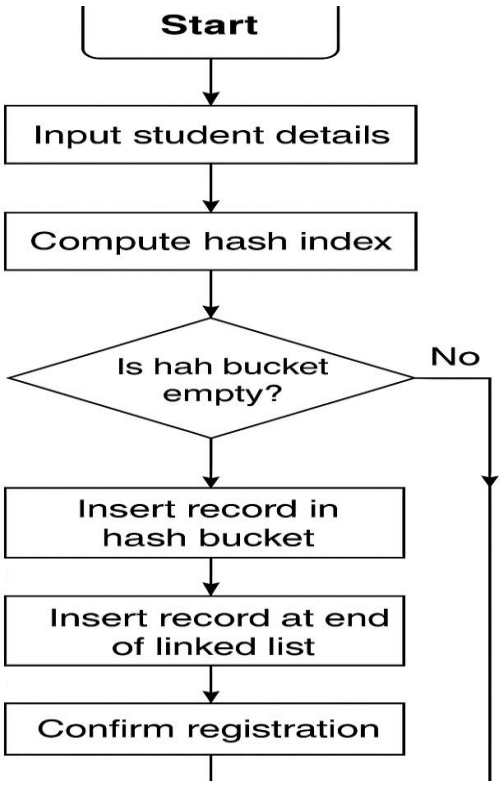
END IF

Add record to sequential linked list for reporting

Display "Student Registered Successfully"

END

**FLOWDIAGRAM**




2**.course allocation**

**pseudocode**

**START**

Input: student_ID, course_ID

IF course.capacity > current_enrolled THEN

  ENQUEUE(student_ID) in course queue

  Allocate course to student

ELSE

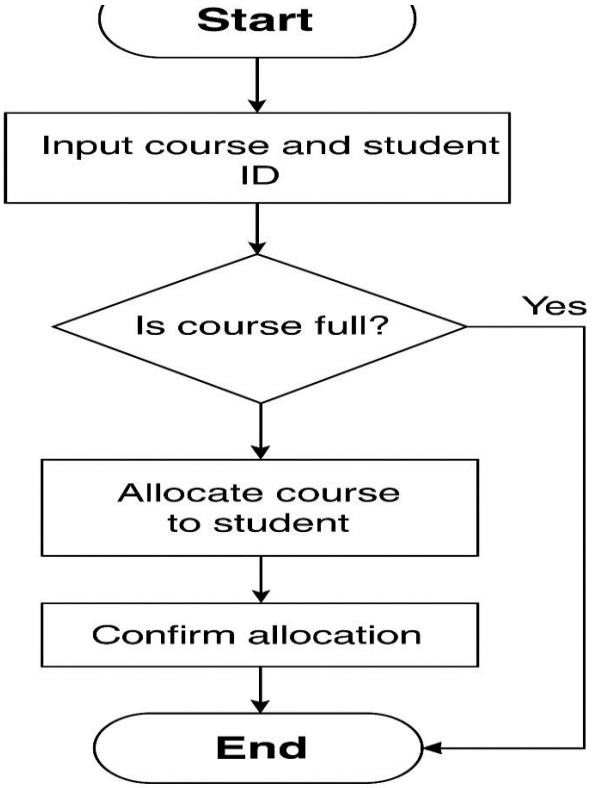
  Add student_ID to waitlist

END IF

Display "Registration Successful" or "Waitlisted"

END

**flowdiagram**




**Fee tracking and reporting**

**pseudocode**

START

Input: student_ID, amount, date

Create payment_node(student_ID, amount)

INSERT payment_node into AVL_tree

IF tree unbalanced THEN

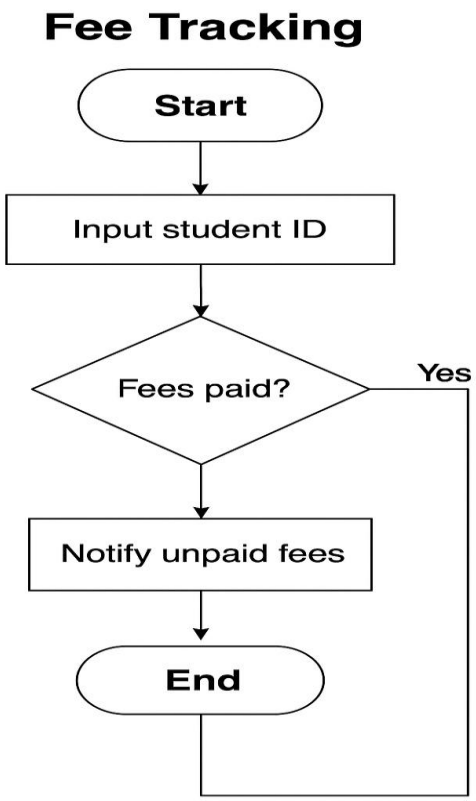
  PERFORM appropriate rotation

END IF

Display "Payment Recorded Successfully"

END

**flowdiagram**




**Library book management**

**pseudocode**

START

Input: ISBN, action (borrow/return)

IF action = borrow THEN

  IF book_available(ISBN) THEN

    PUSH(ISBN) to stack

    Update hash_map: mark borrowed

  ELSE

    Display "Book Not Available"

  END IF

ELSE IF action = return THEN

  POP(ISBN) from stack

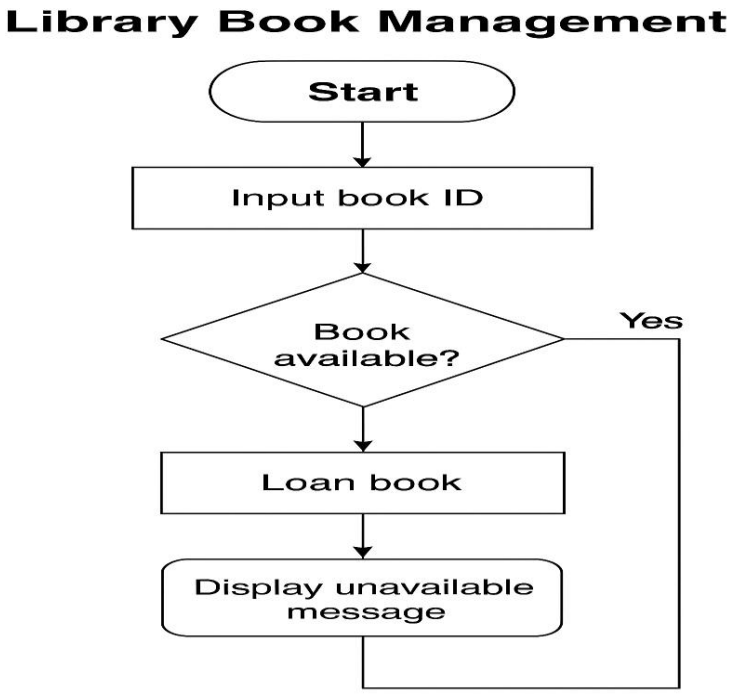
  Update hash_map: mark available

END IF

Display "Action Completed"

END

**Flowchart**




---

**performance analytics**

**pseudocode**

START

FOR each student IN course_list DO

  FOR each subject IN student_subjects DO

    Insert grade into matrix[student][subject]

  END FOR

END FOR

Build max_heap from student average grades

Display top N performers from heap

END

flowchart