```
────────────────────────────────────────────────────────
│                School Manager (Controller)
│               "Central Coordination Layer"                        │
├──────────────────────────────────────────────────────────
──┤
│  ┌──────────────┐  ┌───────────────┐  ┌────────────────────┐
│  │ Student       │  │       Course    │  │           Fee        │  │
│  │ Registry      │  │       Scheduler  │  │          Tracker      │  │
│  │ (HashMap)     │  │         (Queue)    │  │       (BST)        │  │
│  └──────────────┘  └───────────────┘  └────────────────────┘
│                          │
│  ┌──────────────┐  ┌───────────────┐
│  │ Library                │  │ Analytics  │
│  │ System                │  │ Engine     │
│  │ (Stack)                │  │ (Graph)    │
│  └──────────────┘  └───────────────┘
└──────────────────────────────────────────────────────────
──┘
```

Module Interaction & Data Sharing

Central Controller Pattern: School Manager orchestrates all inter-module communication

Data Flow: Modules don't communicate directly; all requests routed through controller

Data Sharing: Shared data models (Student, Course, Transaction) ensure consistency

Event Coordination: Enrollment triggers analytics updates; payments trigger fee tracking

Data Flow Example: Complete Student Journey

text

1. REGISTRATION:

User → SchoolManager → StudentRegistry.addStudent() → HashMap.put()

2. COURSE ENROLLMENT:

User → SchoolManager → CourseScheduler.enrollStudent()

→ IF seats available: enroll & notify Analytics

→ IF full: add to PriorityQueue waitlist

3. FEE PAYMENT:

User → SchoolManager → FeeTracker.recordPayment()

→ BST insertion + HashMap update

4. LIBRARY ACCESS:

User → SchoolManager → LibrarySystem.borrowBook()

→ Stack push + HashMap update

5. PERFORMANCE TRACKING:

Automated → AnalyticsEngine.recordGrade()

→ Graph edge addition

2. DATA STRUCTURE JUSTIFICATION

MODULE 1: Student Registry - HashMap

Why Chosen:

- O(1) average case for insert, lookup, delete operations
- Perfect for frequent student lookups by unique ID
- Automatic collision handling with chaining
- Excellent scalability for large student populations

Alternative Considered: Linked List

- Rejected because: O(n) search time unacceptable for institutional use

MODULE 2: Course Scheduler - Priority Queue

Why Chosen:

- O(log n) enrollment operations with automatic ordering
- Fair allocation based on priority + timestamp
- Efficient waitlist processing
- Natural modeling of "first-come, first-served" with priority override

Alternative Considered: ArrayList

- Rejected because: O(n) insertion in middle, no automatic ordering

MODULE 3: Fee Tracker - Binary Search Tree

Why Chosen:

- O(log n) average insertion with natural date ordering
- O(n) in-order traversal for chronological financial reports
- Efficient range queries for date-based reporting
- Maintains sorted order without explicit sorting

Alternative Considered: PriorityQueue

- Rejected because: O(n log n) sorting required for reports vs O(n) traversal

MODULE 4: Library System - Stack + HashMap

Why Chosen:

- O(1) operations for all critical functions
- LIFO behavior perfect for tracking most recent activities
- ISBN-based lookup with HashMap provides instant access
- Stack naturally models borrow/return sequence

Alternative Considered: ArrayList for history

- Rejected because: O(n) insertion at beginning vs O(1) stack push

MODULE 5: Analytics Engine - Graph (Adjacency List)

Why Chosen:

- O(1) edge addition for grade recording

- Natural modeling of student-course relationships
- Efficient traversal for complex analytics
- Flexible queries for various reporting needs

Alternative Considered: Separate HashMaps

- Rejected because: Would require maintaining multiple synchronized data structures

3. FLOW DIAGRAMS & PSEUDOCODE

MODULE 1: Student Registry - HashMap

Flowchart: Student Registration

Pseudocode: Student Registry

text

```
MODULE StudentRegistry
    DATA:
        students: HashMap<String, Student>
        nextId: Integer = 1001

    FUNCTION registerStudent(name, email, year)
        studentId = "S" + nextId
        student = NEW Student(studentId, name, email, year)
        students.put(studentId, student)
        nextId = nextId + 1
        RETURN studentId
    END FUNCTION
    FUNCTION findStudent(studentId)
        RETURN students.get(studentId)
    END FUNCTION
    FUNCTION removeStudent(studentId)
```

```
        IF students.containsKey(studentId) THEN
            students.remove(studentId)
            RETURN true
        ELSE
            RETURN false
        END IF
    END FUNCTION
    FUNCTION displayAllStudents()
        FOR EACH student IN students.values()
            PRINT student.toString()
        END FOR
    END FUNCTION
END MODULE
```

MODULE 2: Course Scheduler - Priority Queue

Flowchart: Course Enrollment

Pseudocode: Course Scheduler

text

```
MODULE CourseScheduler
    DATA:
        courses: HashMap<String, Course>
        waitlists: HashMap<String, PriorityQueue<Registration>>
        enrollments: HashMap<String, Set<String>>

    FUNCTION enrollStudent(courseId, studentId, priority)
        IF NOT courses.containsKey(courseId) THEN
            RETURN EnrollmentResult(false, "Course not found", -1)
        END IF

        course = courses.get(courseId)
        enrolled = enrollments.get(courseId)
        waitlist = waitlists.get(courseId)

        IF enrolled.contains(studentId) THEN
            RETURN EnrollmentResult(false, "Already enrolled", -1)
        END IF

        IF enrolled.size() < course.getCapacity() THEN
            enrolled.add(studentId)
            RETURN EnrollmentResult(true, "Enrolled successfully", -1)
        ELSE
            registration = NEW Registration(studentId, priority, currentTime)
            waitlist.offer(registration)
            position = waitlist.size()
            RETURN EnrollmentResult(false, "Added to waitlist", position)
        END IF
    END FUNCTION

    FUNCTION processWaitlist(courseId)
        newlyEnrolled = NEW List<String>

        course = courses.get(courseId)
        enrolled = enrollments.get(courseId)
        waitlist = waitlists.get(courseId)


        WHILE waitlist NOT empty AND enrolled.size() < course.getCapacity()
          registration = waitlist.poll()
          enrolled.add(registration.studentId)
          newlyEnrolled.add(registration.studentId)
        END WHILE


        RETURN newlyEnrolled
      END FUNCTION
    END MODULE
```

MODULE 3: Fee Tracker - Binary Search Tree

Flowchart: BST Insertion

Flowchart: In-Order Traversal (Financial Report)

Pseudocode: Fee Tracker

text

```
MODULE FeeTracker
  DATA:
    root: Transaction
    transactionCount: Integer = 0
    studentTransactions: HashMap<String, List<Transaction>>


  FUNCTION recordPayment(studentId, amount, date)
    transactionId = "T" + (transactionCount + 1)
    newTransaction = NEW Transaction(transactionId, studentId, amount, date)
    root = insert(root, newTransaction)
    studentTransactions.getOrDefault(studentId, NEW List).add(newTransaction)
    transactionCount = transactionCount + 1

    RETURN transactionId
END FUNCTION

FUNCTION insert(node, newTransaction)
    IF node IS NULL THEN
        RETURN newTransaction
    END IF

    IF newTransaction.date < node.date THEN
        node.left = insert(node.left, newTransaction)
    ELSE
        node.right = insert(node.right, newTransaction)
    END IF

    RETURN node
END FUNCTION

FUNCTION generateFinancialReport()
    report = NEW List<Transaction>
    inOrderTraversal(root, report)
    RETURN report
END FUNCTION
```

```
    FUNCTION inOrderTraversal(node, report)
        IF node IS NOT NULL THEN
            inOrderTraversal(node.left, report)
            report.add(node)
            inOrderTraversal(node.right, report)
        END IF
    END FUNCTION
END MODULE
```

MODULE 4: Library System - Stack + HashMap

Flowchart: Borrow Book Operation

Pseudocode: Library System

text

```
MODULE LibrarySystem
    DATA:
        catalog: HashMap<String, Book>
        borrowHistory: HashMap<String, Stack<BorrowRecord>>

    FUNCTION borrowBook(isbn, studentId)
        book = catalog.get(isbn)

        IF book IS NULL THEN
            PRINT "Book not found"
            RETURN false
        END IF
    IF book.availableCopies > 0 THEN
        book.availableCopies = book.availableCopies - 1
        record = NEW BorrowRecord(studentId, "BORROW", currentTime)
        borrowHistory.get(isbn).push(record)
        PRINT "Book borrowed successfully"
        RETURN true
    ELSE
        PRINT "No copies available"
        RETURN false
    END IF
END FUNCTION

FUNCTION returnBook(isbn, studentId)
    book = catalog.get(isbn)

    IF book IS NULL THEN
        PRINT "Book not found"
        RETURN false
    END IF

    IF book.availableCopies < book.totalCopies THEN
        book.availableCopies = book.availableCopies + 1
        record = NEW BorrowRecord(studentId, "RETURN", currentTime)
        borrowHistory.get(isbn).push(record)
        PRINT "Book returned successfully"
        RETURN true
    ELSE
      PRINT "All copies already available"
      RETURN false
    END IF
  END FUNCTION

  FUNCTION displayRecentActivity(isbn, count)
    history = borrowHistory.get(isbn)

    IF history IS NULL OR history.empty() THEN
      PRINT "No history available"
      RETURN
    END IF

    displayed = 0
    tempStack = COPY OF history

    WHILE NOT tempStack.empty() AND displayed < count
      record = tempStack.pop()
      PRINT record.toString()
      displayed = displayed + 1
    END WHILE
  END FUNCTION
END MODULE
```

MODULE 5: Analytics Engine - Graph

Flowchart: Course Analysis

Pseudocode: Analytics Engine

text

```
MODULE AnalyticsEngine
    DATA:
        graph: HashMap<String, List<Edge>>

    FUNCTION recordGrade(studentId, courseId, grade)
        // Create bidirectional edges
        edge1 = NEW Edge(courseId, grade, "STUDENT_TO_COURSE")
        edge2 = NEW Edge(studentId, grade, "COURSE_TO_STUDENT")

        // Add to student's adjacency list
        graph.getOrDefault(studentId, NEW List).add(edge1)
        // Add to course's adjacency list
        graph.getOrDefault(courseId, NEW List).add(edge2)
    END FUNCTION

    FUNCTION analyzeCourse(courseId)
        edges = graph.get(courseId)

        IF edges IS NULL THEN
            RETURN CourseAnalysis(courseId, 0, 0, 0, EMPTY_LIST)
        END IF

        sum = 0
        count = 0
        maxGrade = MIN_VALUE
        minGrade = MAX_VALUE
        studentGrades = NEW List<StudentGrade>()

        FOR EACH edge IN edges
            IF edge.type EQUALS "COURSE_TO_STUDENT" THEN
                grade = edge.weight
                studentId = edge.targetNode

                sum = sum + grade
                count = count + 1
                maxGrade = MAX(maxGrade, grade)
                minGrade = MIN(minGrade, grade)
                studentGrades.add(NEW StudentGrade(studentId, grade))
            END IF
        END FOR

        IF count > 0 THEN
            average = sum / count
        ELSE
            average = 0
        END IF

        SORT studentGrades DESCENDING BY grade
        RETURN CourseAnalysis(courseId, average, maxGrade, minGrade, studentGrades)
    END FUNCTION
```

```
FUNCTION getTopPerformers(topN)
    minHeap = NEW PriorityQueue<StudentGrade>(topN)
    studentAverages = NEW HashMap<String, Double>()

    // Calculate averages for all students
    FOR EACH studentId IN graph.keySet() WHERE studentId STARTS WITH "S"
        grades = getStudentGrades(studentId)
        IF grades NOT EMPTY THEN
            average = SUM(grades) / SIZE(grades)
            studentAverages.put(studentId, average)
        END IF
    END FOR

    // Maintain top N in min-heap
    FOR EACH studentId, average IN studentAverages
        minHeap.offer(NEW StudentGrade(studentId, average))
        IF minHeap.size() > topN THEN
            minHeap.poll()  // Remove smallest
        END IF
    END FOR

    // Convert to sorted list
    result = NEW List<StudentGrade>(minHeap)
    SORT result DESCENDING BY grade
    RETURN result
    END FUNCTION
END MODULE
```