Course Enrollment System

- Nikhitha D, CS23B103

I) Data Structures:

1. `map<string, student> sMap`:

- Stores the list of students, mapped by their student ID. This allows quick lookups for any given student ID.
- **Time Complexity**: **`O(log(N))`** for operations like insert and find, where `N` is the number of students.

2. `map<string, course> cMap`:

- Holds all course information, using the course code as the key.
- **Time Complexity**: **`O(log(M))`** for operations where `M` is the number of courses.

3. `map<string, set<string>> csList`:

- Maps each course to a set of enrolled student IDs. Using a `set` allows efficient insertion, deletion, and iteration while keeping the students sorted.
- **Time Complexity**: **`O(log(T))`** for insertion and deletion due to `set`.

4. `map<string, queue<string>> waitList`:

- Stores a queue of students waiting to enrol in each course if capacity is reached.
- **Time Complexity**: **`O(1)`** for enqueue and dequeue operations.

5. `unordered_map<char, int> slotMap` (in `student` class):

- Tracks the time slots to detect clashes for enrolled courses for a student.
- **Time Complexity**: **`O(1)`** for insert, delete, and find due to `unordered_map`.

II) PseudoCode:

1. **`add_student` Operation**
    - **Pseudocode**:
        - Read student ID, name, year, and completed courses.
        - Insert a new student object into `sMap` using the student ID.
        - Store completed courses in a set for each student.
    - **Complexity**: **`O(ncc * log(ncc))`** where `ncc` is the number of completed courses, due to `set` insertions.

2. **`add_course` Operation**
    - **Pseudocode**:
        - Read course code, name, credits, capacity, slot, and prerequisites.
        - Check if all prerequisites exist in `cMap`; if any prerequisite is missing, skip adding the course.
        - Check for cycles using `isCyclic()` before adding the course to prevent dependency issues.
    - **Complexity**: **`O(npre + E)`** where `npre` is the number of prerequisites and `E` is the number of dependencies for cycle checking.

3. **`enroll` Operation**
    - **Pseudocode**:
        - Verify if the student meets prerequisites.
        - Check for slot clashes using `slotMap`.
        - If the course capacity allows, enrol the student by adding the course to `enrolc` and updating `slotMap` and `csList`.
        - If the course is full, place the student in the `waitList`.
    - **Complexity**: **`O(log(T))`** for insertion into `csList` due to `set`.

4. **`drop` Operation**
    - **Pseudocode**:
        - Remove the course from `enrolc`, decrement capacity, and remove from `csList`.
        - Process the `waitList` to enrol the next student if a spot opens.
    - **Complexity**: **`O(log(T))`** for removing from `csList`.

5. **`print` Operation**
    - **Pseudocode**:
        - Check if the course exists in `csList`.
        - Print all enrolled students for the course.
    - **Complexity**: **`O(log(T) + K)`**, where K is the number of enrolled students for the course.

Time Complexity Summary Table:

Operation	Time Complexity
add_student	O(ncc * log(ncc))
add_course	O(npre + E)
enroll	O(log(T))
drop	O(log(T))
print	O(log(T) + K)

III) CYCLIC DEPENDENCY:

Yes, a cyclic dependency can be introduced when adding a course, even with the checks in the initial implementation. This happens if a course's prerequisite depends on another course that eventually loops back to itself, creating a cycle. To prevent this, the `add_course` function should include a cycle detection check. If a cycle is detected, the course should not be added.

Explanation of the Changes

- **Tentative Addition:** We add the course temporarily to cMap so we can check if adding it would introduce a cycle.
- **Cycle Detection:** We use the isCyclic method to check if there's a cyclic dependency.
- **Rejection of Cyclic Courses:** If a cycle is detected, the course is removed from cMap, and a message is printed indicating that the course was not added due to a cycle.

Justification

This approach ensures that cyclic dependencies are detected and rejected during the add_course operation. The cycle detection is implemented using the hasCycle and isCyclic methods, which traverse prerequisites to identify loops. This guarantees that every course added will not create any circular dependencies, maintaining the integrity of the course dependency graph.