# 1.Use-cases

## Course Registration System - Use Cases

### 1. Primary Use Cases

**UC-1**: Register for Course
**Actor**:Student
**Preconditions**:Student is logged in and has completed prerequisites
**Flow**:
1. Student views available courses
2. Student selects desired course and section
3. System validates prerequisites using DAG traversal
4. System checks for timetable conflicts
5. System checks seat availability
6. If seats available: System enrolls student
7. If section full: System adds student to waitlist
8. System confirms registration or waitlist placement

**Postconditions**:Student is enrolled or waitlisted
**Alternative Flows:**
- 3a. Prerequisites not met then system will display an error with missing courses
- 4a. Timetable conflict then system will display conflicting course details

**UC-2**: Drop Course
**Actor**: Student
**Preconditions**:Student is enrolled in the course
**Flow**:
1. Student requests to drop course
2. System removes student from enrolled list
3. System checks if waitlist exists for that section
4. If waitlist not empty: System promotes first student from queue
5. System sends notification to promoted student
6. System updates seat count

**Postconditions**: Student removed, seat reallocated if waitlist exists

**UC-3**: Validate Prerequisites
**Actor**: System (Registrar Module)
**Preconditions**: Student attempts course registration
**Flow**:
1. System receives courseID from registration request
2. CourseGraph module performs DFT/BFT traversal on prerequisite DAG
3. System retrieves student's completed courses from hash map
4. System compares required prerequisites with completed courses

5. System returns validation result (pass/fail with missing courses)
**Postconditions**: Prerequisites validated or list of missing courses provided

**UC-4**: Check Timetable Conflicts
**Actor**: System (Registrar Module)
**Preconditions**: Student has existing schedule
**Flow**:
1. System retrieves student's current enrolled sections
2. System gets time slots for each enrolled section from array grid
3. System retrieves time slot for requested section
4. System compares time slots for overlaps
5. System returns conflict status with conflicting course if found
**Postconditions**: Conflict detected or registration cleared for timing

**UC-5**: Manage Waitlist
**Actor**: System (SectionManager Module)
**Trigger**: Student drops course
**Flow**:
1. SectionManager detects seat availability in section
2. System checks section's FIFO waitlist queue
3. If queue not empty: System dequeues first student
4. System validates student's current prerequisites (recheck)
5. System checks for timetable conflicts with student's current schedule
6. If valid: System will enroll student
7. If invalid: System moves to next student in queue
8. System updates seat count and waitlist
**Postconditions**: Seat filled or remains available if no valid waitlist student

**UC-6**: Balance Section Load
**Actor**: System (SectionManager Module)
**Trigger**: Registration request or periodic balancing
**Flow**:
1. System retrieves all sections for requested course
2. System calculates load factor for each section using heap
3. System identifies sections below capacity threshold
4. System suggests least loaded section to student
5. If student accepts: System registers in suggested section
6. System updates heap structure with new load
**Postconditions**: Student placed in balanced section

**UC-7**: Generate Enrollment Report
**Actor**:Admin/Registrar
**Preconditions**: System has registration data
**Flow**:
1. Admin requests enrollment report for specific course
2. Reports module queries hash map for course sections
3. System retrieves enrolled students for each section
4. System formats report with student details and seat counts
5. System outputs report
**Postconditions**: Report generated showing enrollment status

**UC-8**: Generate Waitlist Report
**Actor**: Admin/Registrar
**Preconditions**: Waitlists exist for sections
**Flow**:
1. Admin requests waitlist report
2. Reports module iterates through section queues
3. System retrieves student details for each waitlisted student
4. System includes position in queue and timestamp
5. System outputs report
**Postconditions**: Waitlist report generated with queue positions

**2. Secondary Use Cases**

**UC-9**: Add Course Prerequisites
**Actor**: Admin
**Flow**:
1. Admin specifies course and prerequisite course
2. System adds directed edge in prerequisite DAG
3. System validates no cycles created using topological sort
4. If cycle detected: Reject and display error
5. If valid: Confirm prerequisite added

**UC-10**: View Available Seats
**Actor**:Student
**Flow**:
1. Student requests seat availability for course
2. System queries SectionManager for all sections
3. System calculates available seats (capacity - enrolled)
4. System displays section details with seat counts

**UC-11**: Check Student Eligibility
**Actor**: System
**Trigger**: Before any registration attempt
**Flow**:
1. System checks student account status
2. System verifies student not already enrolled in course
3. System validates maximum credit hours not exceeded
4. System returns eligibility status

## 3. Use Case Relationships

**Include Relationships**:
- UC-1 includes UC-3 (Register includes Validate Prerequisites)
- UC-1 includes UC-4 (Register includes Check Conflicts)
- UC-2 includes UC-5 (Drop includes Waitlist Promotion)
- UC-6 included by UC-1 (Balancing during registration)

**Extend Relationships**:
- UC-5 extends UC-2 (Promote student only if waitlist exists)
- UC-6 extends UC-1 (Balancing suggested as alternative)

## 4. Use Case Priorities

**Critical (Must Have)**:
- UC-1: Register for Course
- UC-2: Drop Course
- UC-3: Validate Prerequisites
- UC-4: Check Timetable Conflicts

**Important (Should Have)**:
- UC-5: Manage Waitlist
- UC-6: Balance Section Load
- UC-7: Generate Enrollment Report

**Nice to Have**:
- UC-8: Generate Waitlist Report
- UC-10: View Available Seats
- UC-11: Check Student Eligibility

# 2.Class diagram/pseudocode

**Class Diagram:**

https://www.canva.com/design/DAG7I9Aewno/QzWL0b02HlEu8MdL5Frnsg/view?utm_content=DAG7I9Aewno&utm_campaign=designshare&utm_medium=link2&utm_source=uniquelinks&utlId=h93cf0d180b

# 3.Data structure specs

## Students

### Attributes:

- `studentIDs[ ]` (int) : Unique ID for each student
- `studentNames[ ]` (char* array) : Names of students
- `completedCourses[ ][ ]` (2D char* array) : Courses already completed
- `enrolledCourses[ ][ ]` (2D char* array) : Courses registered this semester

## Courses

### Attributes:

- `courseCodes[ ]` (char* array) : Unique course IDs
- `courseTitles[ ]` (char* array) : Names of courses
- `prerequisites[ ][ ]` (2D char* array) : Prerequisite course codes for each course

### Sections

**Attributes:**

- `sectionID` (char*) : Unique section ID
- `courseCode` (char*) : Which course this section belongs to
- `capacity` (int) : Max seats
- `enrolledStudents[]` (int array) : StudentIDs currently enrolled
- `enrolledCount` (int) : Number of students enrolled
- `waitlist[]` (int array or linked list) : FIFO queue for waitlisted students

### Timetable

**Attributes:**

`timetable[][] (2D char* array) : Each student's registered timeslots`

### Reports

**Attributes:**

- `reportID` (int) : Unique identifier for the report. Example: 1, 2, 3.
- type (char*) : Type of report. Values: `"enrolled"`, `"waitlisted"`, `"available seats"`.
- `"enrolled"` will show which students are enrolled in a section
- `"waitlisted"` will show which students are in the waitlist queue
- `"available seats"` will show remaining seat count per section

- `data` *(char* array)* : Report content, depending on the report type:
- **Enrolled Report:** studentIDs in a section, e.g., `{"S1:101", "S1:102"}`
- **Waitlisted Report:** studentIDs in waitlist queue, e.g., `{"S1:105"}`
- **Available Seats Report:** sectionID and remaining seats, e.g., `{"S1:38"}`

# 4.File formats

This project uses multiple text-based files to store and manage data.
All files are simple `.txt` formats so they can easily be read using `fstream` in C++.

### 1. `courses.txt`

**Stores:** Course information
**Includes:**

- Course code
- Course name
- Total seats
- Seats already filled

### 2. `students.txt`

**Stores:** Student information
**Includes:**

- Student ID
- Student Name
- Program / Department

### 3. `prereq.txt`

**Stores:** Course prerequisite pairs (prerequisite graph)
**Includes:**

- Course
- Its prerequisite course

### `4.completed.txt`

**Stores:** Courses that each student has completed (for prerequisite checking)
**Includes:**

- Student ID
- Completed course

### 5. `registrations.txt`

**Stores:** Registration output results
**Includes:**

- Student ID
- Course ID
- Status (Registered / Waitlisted)

### 6. `waitlist.txt`

**Stores:** Students waiting for seats in courses
**Includes:**

- Course ID
- Student IDs in waiting queue

## 5.Sample datasets

### Dataset 1: Student IDs (1D array)

```
int studentIDs[] = {101, 103, 102, 105, 104};

int studentCount = 5;
```

### Dataset 2: Student Names (parallel pointer array)

```
char* studentNames[] = {"Zeenat", "Kashaf", "Meryam", "Amna", "Bisma"};
```

### Dataset 3: Completed Courses (2D array of strings)

Each student may have completed multiple courses (empty string if none):

```
char* completedCourses[5][3] = {

    {"OOP", "Math101", ""},

    {"OOP", "Math101", "DSA"},

    {"Math101", "", ""},

    {"OOP", "", ""},

    {"", "", ""}

};
```

### Dataset 4: Enrolled Courses (2D array of strings)

Courses student is currently registered for (empty string if none):

```
char* enrolledCourses[5][3] = {
    {"", "", ""},
    {"", "", ""},
    {"", "", ""},
    {"", "", ""},
    {"", "", ""}
};
```

**Dataset 5: Course Codes (1D array)**

```
char* courseCodes[] = {"CS101", "CS201", "MA101", "CS301"};
int courseCount = 4;
```

**Dataset 6: Course Titles (parallel pointer array)**

```
char* courseTitles[] = {"OOP", "DSA", "Math101", "Algorithms"};
```

**Dataset 7: Prerequisites (2D array of strings)**

```
char* prerequisites[4][2] = {
    {"PF", ""},
    {"OOP", ""},
    {"", ""},
    {"DSA", ""}
};
```