Project 4: Course Registration System

1. Introduction

In this assignment, you will implement five modules in order to represent a course registration system: Student, StudentDatabase (+ BSTNode), Course, CourseManager, and EnrollmentManager.

The code skeleton provides you with seven files (**Student.hpp, StudentDatabase.hpp, BSTNode.hpp, Course.hpp, CourseManager.hpp, EnrollmentManager.hpp**, **test.cpp**) and is under **/comp/15/files/p4** on the CS server.

1. Log in to the CS homework server.
2. Move to your **comp15** directory that you created in Lab 1.
3. Create a directory named **project4** under the comp15 directory.
4. Move to the project directory.
5. Copy the code skeleton to the current working directory.
6. Leverage the features of Git to manage your progress toward writing the program.

As the first step, you are encouraged to take a look at the contents of the code skeleton and try to identify what each file is for. Then, spend enough time to think about how to represent a course registration system with your program modules. The following list contains some requirements for your program specification:

- A student database is a binary search tree of student data (pointers).
- A course holds an array of student IDs.
- A course manager holds an array of course data (pointers).
- An enrollment manager has access to the student database and the course manager.
- All instances of Student class sent to a student database live in the heap memory space; plus, after they are added to the student database, the student database is responsible for them.
- All instances of Course class sent to a course manager live in the heap memory space; plus, after they are held by the course manager, the course manager is responsible for them.

Suggestion: The size of this project is bigger than any of the previous projects. **Consider writing test cases for each of your modules as you make progress**.

You will at least need to create **Student.cpp**, **StudentDatabase.cpp**, **BSTNode.cpp**, **Course.cpp**, **CourseManager.cpp** and **EnrollmentManager.cpp** files by yourself. Note that the file names matter for the grading purpose and are case-sensitive. Note also that the public section of each .hpp files as well as the entire BSTNode.hpp **cannot** be modified. Finally, you are expected to write your own tests in **test.cpp**.

2. Specifications

Part 1: Student class

`Student();`

This student is initialized to be one that holds (id) **0**, (first name, case-sensitive) **FIRST**, and (last name, case-sensitive) **LAST**.

`Student(int id, std::string firstName, std::string lastName);`

This student is initialized to hold the given data.
If the given "id" is a negative integer and/or the size of the given "firstName" is zero and/or the size of the given "lastName" is zero, then throw a `std::runtime_error` exception with the message (case-sensitive): **Not Valid**

`Student(const Student& other);`
`Student& operator=(const Student& other);`

This student is initialized to be one that holds the same information that the given student holds.

`~Student();`

You are supposed to release (memory) resources used for this student, if necessary.

`int getID() const;`

Return the id that this student holds.

`std::string getFirstName() const;`

Return the first name that this student holds.

`std::string getLastName() const;`

Return the last name that this student holds.

`std::string toString() const;`

Return the string representation of this student (case-sensitive). The representation should have the form such that: **id first last**

- where **id** is this student's id,
- followed by a space,
- followed by **first** that is this student's first name,
- followed by a space,
- followed by **last** that is this student's last name.

For example, the string representation of a student holding (id) **1**, (first name) **John**, and (last name) **Smith** is: **1 John Smith**

Part 2: StudentDatabase class

`StudentDatabase();`

This student database is initialized to be one that holds no student data at this moment.

`StudentDatabase(const StudentDatabase& other);`
`StudentDatabase& operator=(const StudentDatabase& other);`

Throw a `std::runtime_error` exception with the message (case-sensitive):
Not Implemented

`~StudentDatabase();`

You are supposed to release (memory) resources used for this student database, if necessary.

`const Student* searchBy(int studentID) const;`

Return (the pointer to) the student data that is associated with the given "studentID" in this student database. Return `nullptr` if there is no such student data in this student database.

`bool insert(Student* student);`

Insert the given student data to this student database in which the search key is the student id of the student data. Return `true` if this insertion succeeds; `false`, otherwise.
If the given data is `nullptr`, then do not add it and return `false`.
If this student database already has data that holds the same information that the given data holds, then do not add the given data and return `false`.
Please assume the instance of the Student class pointed by the given "student" is in the heap space.

`bool deleteBy(int studentID);`

Delete the student data that is associated with the given "studentID" from this student database. Return `true` if this deletion succeeds; `false`, otherwise.

`void balance();`

Balance the current tree.

`bool isEmpty() const;`

Return `true` if this student database holds no student data; `false`, otherwise.

`int getNumberOfStudents() const;`

Return the number of student data that this student database holds.

`std::string toStringInOrder() const;`

Return the string representation of this student database (case-sensitive). The representation should have the form such that:

- A line contains the IDs of all student data in this student database.
- Each ID is followed by a comma, except for the last one.
- The IDs are ordered by using in-order traversal.

For example, assume a student database is holding (**1**, **John**, **Smith**), (**2**, **David**, **Smith**), and (**3**, **Mary**, **Smith**). The string representation of the student database is: **1,2,3**

`std::string toTreeString() const;`

Return the tree string representation of this student database (case-sensitive). Examples are as follows:

- Assume a student database is holding no student data. The tree string representation of the student database is: **[]**
- Assume a student database is holding one student data, (**1**, **John**, **Smith**). The tree string representation of the student database is: **[[](1 John Smith)[]]**
- Assume a student database is holding (**1**, **John**, **Smith**) and (**2**, **David**, **Smith**).
    - If #1 is the root, #2 is the right of #1, then the tree string representation of the student database is: **[[](1 John Smith)[[](2 David Smith)[]]]**
    - If #2 is the root, #1 is the left of #2, then the tree string representation of the student database is: **[[[](1 John Smith)[]](2 David Smith)[]]**
- Assume a student database is holding (**1**, **John**, **Smith**), (**2**, **David**, **Smith**), and (**3**, **Mary**, **Smith**).
    - If #1 is the root, #2 is the right of #1, and #3 is the right of #2, then the tree string representation of the student database is: **[[](1 John Smith)[[](2 David Smith)[[](3 Mary Smith)[]]]]**
    - If #1 is the root, #3 is the right of #1, and #2 is the left of #3, then the tree string representation of the student database is: **[[](1 John Smith)[[[](2 David Smith)[]](3 Mary Smith)[]]]**
    - If #2 is the root, #1 is the left of #2, and #3 is the right of #2, then the tree string representation of the student database is: **[[[](1 John Smith)[]](2 David Smith)[[](3 Mary Smith)[]]]**

Part 3: BSTNode class

`BSTNode();`
`BSTNode(const BSTNode& other);`
`BSTNode& operator=(const BSTNode& other);`

Throw a `std::runtime_error` exception with the message (case-sensitive):
Not Implemented

`BSTNode(Student* data);`

This node is initialized to be one that holds the given data.

`~BSTNode();`

You are supposed to release (memory) resources used for this node, if necessary.

`Student* getData() const;`

Return the data that this node holds.

`BSTNode* getLeft() const;`

Return the left node of this node.

`void setLeft(BSTNode* left);`

Set the given node to be the left node of this node.

`BSTNode* getRight() const;`

Return the right node of this node.

`void setRight(BSTNode* right);`

Set the given node to be the right node of this node.

Part 4: Course class

`Course();`

This course is initialized to be one that holds (id, case-sensitive) **COMP-000**, and (capacity) **1**.

`Course(std::string id, int capacity);`

This course is initialized to hold the given data.
If the size of the given "id" is zero and/or the given "capacity" is less than or equal to zero, then throw a `std::runtime_error` exception with the message (case-sensitive): **Not Valid**

`Course(const Course& other);`
`Course& operator=(const Course& other);`

This course is initialized to be one that holds the same information (deep copy) that the given course holds.

`~Course();`

You are supposed to release (memory) resources used for this course, if necessary.

`bool isFull() const;`

Return `true` if this course is already full; `false`, otherwise.

`bool enroll(int studentID);`

Add the given "studentID" to the enrolled students list of this course, in which the added ID becomes the last ID in the list. Return `true` if the enrollment succeeds; `false`, otherwise.
If this course is already full, then do not add the given "studentID" and return `false`.
If the given "studentID" is already in the enrolled students list of this course, then do not add the given "studentID" and return `false`.

`bool drop(int studentID);`

Remove the given "studentID" from the enrolled students list of this course. Return `true` if the drop action succeeds; `false`, otherwise. If the enrolled students list of this course does not contain the given "studentID", then return `false`.

`bool isHaving(int studentID);`

Return `true` if the given "studentID" is already in the enrolled students list of this course; `false`, otherwise.

`std::string getID() const;`

Return the id that this course holds.

`int getCapacity() const;`

Return the capacity that this course holds.

`int getNumberOfEnrolledStudents() const;`

Return the number of enrolled students of this course.

`int getStudentIDAt(int index) const;`

Return the student ID at the given "index" in the enrolled student list of this course.
If the given "index" is not in the acceptable range, then throw a `std::range_error` exception with the message (case-sensitive): **Out of Range**

`std::string toString() const;`

Return the string representation of this course (case-sensitive). The representation should have the form such that: **id (enrolled/capacity)**

- where **id** is the id of this course,
- followed by a space,
- followed by an open parenthesis,
- followed by **enrolled** that is the number of the enrolled students,
- followed by a forward slash,
- followed by **capacity** that is the capacity of this course,
- followed by a close parenthesis.

For example, the string representation of a course holding (id) **COMP-015**, (enrolled) **14**, and (capacity) **20** is: **COMP-015 (14/20)**

Part 5: CourseManager class

`CourseManager();`

This course manager is initialized to be one that can hold an arbitrary number of (pointers to) courses.

`CourseManager(const CourseManager& other);`
`CourseManager& operator=(const CourseManager& other);`

Throw a `std::runtime_error` exception with the message (case-sensitive):
Not Implemented

`~CourseManager();`

You are supposed to release (memory) resources used for this course manager, if necessary.

`const Course* searchBy(std::string courseID) const;`

Return (the pointer to) the course data that is associated with the given "courseID" that this course manager holds. Return `nullptr` if there is no such course data.

`bool add(Course* course);`

Add the given course data to this course manager. Return `true` if this addition succeeds; `false`, otherwise.
If the given data is `nullptr`, then do not add it and return `false`.
If this course manager already has data that has the same information that the given data holds, then do not add the given data and return `false`.
Please assume the instance of the Course class pointed by the given "course" is in the heap space.

`bool cancel(std::string courseID);`

Remove the course data that is associated with the given "courseID" from this course manager. Return `true` if this removal succeeds; `false`, otherwise.

`bool enroll(int studentID, std::string courseID);`

Add the given "studentID" to the course data that is associated with the given "courseID". Return `true` if this enrollment succeeds; `false`, otherwise.

`bool drop(int studentID, std::string courseID);`

Remove the given "studentID" from the course data that is associated with the given "courseID". Return `true` if this drop action succeeds; `false`, otherwise.

`void dropFromAllCourses(int studentID);`

Remove the given "studentID" from all the course data that this course manager holds.

`std::string getAllEnrolledCoursesStringOf(int studentID);`

Return the string representation of all courses that the student associated with the given "studentID" is enrolled in (case-sensitive). The representation should have the form such that:

- A line contains the IDs of courses that the student is enrolled in.
- Each course ID is followed by a comma, except for the last one.
- The order of the course IDs is the same as the order of the courses that this course manager holds.

For example, assume a course manager is holding (**COMP-000**), (**COMP-015**), and (**COMP-010**) in this order and the student is enrolled in (**COMP-015**) and (**COMP-010**). The string representation that this method returns is: **COMP-015,COMP-010**

`int getNumberOfCourses() const;`

Return the number of courses that this course manager holds.

`std::string getCourseListString() const;`

Return the string representation of courses that this course manager holds (case-sensitive). The representation should have the form such that:

- A line contains the ID of each course that this course manager holds.
- Each ID is followed by a comma, except for the last one.
- The order of the course IDs is the same as the order of the courses that this course manager holds.

For example, assume a course manager is holding (**COMP-000**), (**COMP-015**), and (**COMP-010**) in this order. The string representation that this method returns is: **COMP-000,COMP-015,COMP-010**

Part 6: EnrollmentManager class

`EnrollmentManager();`

This enrollment manager is initialized to be one that holds (year) **2019**, and (semester) `Semester::SUMMER`.

`EnrollmentManager(int year, Semester semester);`

This enrollment manager is initialized to hold the given data.
If the given "year" is a negative integer, then throw a `std::runtime_error` exception with the message (case-sensitive): **Not Valid**

`EnrollmentManager(const EnrollmentManager& other);`
`EnrollmentManager& operator=(const EnrollmentManager& other);`

Throw a `std::runtime_error` exception with the message (case-sensitive):
Not Implemented

`~EnrollmentManager();`

You are supposed to release (memory) resources used for this enrollment manager, if necessary.

`bool registerStudent(int studentID, std::string firstName, std::string lastName);`

Add the student data of the given information to the student database that this enrollment manager holds. The student data should be in the heap memory space.
Return `true` if this registration succeeds; `false` otherwise.
After each registration, if the number of student data in the student database becomes a multiple of 100, (call the) balance (method of) the student database.

`bool unregisterStudent(int studentID);`

Remove the student data that is associated with the given "studentID" from the student database that this enrollment manager holds and ask the course manager to remove the given "studentID" from all courses that the student is enrolled in. Return `true` if this removal succeeds; `false` otherwise.

`bool addCourse(std::string courseID, int capacity);`

Ask the course manager that this enrollment manager holds to add the course of the given data. Return `true` if this addition succeeds; `false` otherwise.

`bool cancelCourse(std::string courseID);`

Ask the course manager that this enrollment manager holds to remove the course that is associated with the given "courseID". Return `true` if this removal succeeds; `false` otherwise.

`bool enroll(int studentID, std::string courseID);`

Ask the course manager that this enrollment manager holds to enroll the given "studentID" in the course that is associated with the given "courseID". Return `true` if this enrollment succeeds; `false` otherwise.
Return `false`, if there is no student data that is associated with the given "studentID" in the student database that this enrollment manager holds.

`bool drop(int studentID, std::string courseID);`

Ask the course manager that this enrollment manager holds to remove the given "studentID" from the course that is associated with the given "courseID". Return `true` if this drop action succeeds; `false` otherwise.
Return `false`, if there is no student data that is associated with the given "studentID" in the student database that this enrollment manager holds.

`std::string reportSummary() const;`

Return the summary of information that this enrollment manager holds (case-sensitive). The summary should have the form such that:

- Beginning with the string representation of the semester that this enrollment manager holds (case-sensitive), i.e. one of: **Spring**, **Summer**, or **Fall**
- followed by a space,
- followed by the year that this enrollment manager holds,
- followed by a newline control: **\n**
- followed by the word: **Students:**
- followed by the number of students that the student database holds,
- followed by a newline control: **\n**
- followed by the string that the "toStringInOrder()" method of StudentDatabase class returns,
- followed by a newline control: **\n**
- followed by the word: **Courses:**
- followed by the number of courses that the course manager holds,
- followed by a newline control: **\n**
- followed by the string that the "getCourseListString()" of CourseManager class returns.

`std::string report(int studentID) const;`

Return the report of the student data that is associated with the given "studentID" (case-sensitive). The report should have the form such that:

- Beginning with the string representation of the student data,
- followed by a newline control: **\n**
- followed by the string that the "getAllEnrolledCoursesStringOf()" of CourseManager class returns.

If there is no such student data in the student database that this enrollment manager holds, then return (case-sensitive): **Not Found**

`std::string report(std::string courseID) const;`

Return the report of the course data that is associated with the given "courseID" (case-sensitive). The report should have the form such that:

- Beginning with the string representation of the course data,
- followed by a newline control: **\n**

- followed by the list of students enrolled in the course. The list should have the form such that:
    - Each line begins with the string representation of the student data which is followed by a newline control: **\n** , except for the last line.
    - The order of students in this list is the same as the order of studentIDs that the course holds.

If the course manager that this enrollment manager holds does not hold such course data, then return (case-sensitive): **Not Found**

`int getYear() const;`

Return the year that this enrollment manager holds.

`Semester getSemester() const;`

Return the semester that this enrollment manager holds.

(Optional)

Create a **main.cpp** file and write the main() function, in which a user can type in and issue commands and your EnrollmentManager module responds to the user's commands.

3. Testing

Modify the **test.cpp** by adding your own test cases to ensure that your module function correctly. The given **test.cpp** includes two example test cases using assert(). You are encouraged to add as many test cases as necessary to make you feel confident about your modules.

4. README

Create the README file that includes the following categories with appropriate section headers.

1. **Name**: The programmer's name.
2. **Date**: The last updated date of the program code, including README.
3. **Summary**: A brief summary of the project. (e.g. What does the program do? How do you use the modules? and so on.)
4. **Files**: A list of files that are necessary to build and test the program.
5. **Instructions**: A sequence of commands to compile and test the program. Note that you are expected to report procedures without using the make command.
6. **References**: A list of citations to information used to complete the project.
7. **Post evaluation on planning**: Did your project go smoothly, as planned? If not, which parts were difficult? What were the causes? How will you improve your planning for future assignments?

5. Submission

Submit your files listed below using Gradescope.

Files:	**test.cpp**	**README**	**(main.cpp)**
	BSTNode.cpp	**BSTNode.hpp**	
	Course.cpp	**Course.hpp**	
	CourseManager.cpp	**CourseManager.hpp**	
	EnrollmentManager.cpp	**EnrollmentManager.hpp**	
	Student.cpp	**Student.hpp**	
	StudentDatabase.cpp	**StudentDatabase.hpp**	