NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL

I B. Tech AI & DS

Data Structure and Algorithms Lab

LAB Assignment-4

Student Records Management System using double linked list

In this programming assignment, you will develop a student records management system using doubly linked lists. The system will store and manage student data including roll numbers, names, CGPAs, and their enrolled courses with respective marks. You are required to implement this system using C++ with a focus on modular programming concepts. Your program must handle a series of operations through file input, such as adding or removing students, updating their academic records, and managing course enrollments. Each operation should maintain the structural integrity of the doubly linked list while allowing efficient data manipulation. The implementation must be divided into four distinct modules to ensure code organization and maintainability. Each module will handle specific aspects of the system's functionality as detailed in the following sections.

Module Specifications

Module 1.1 CourseInfo Implementation (courseinfo.cpp, courseinfo.h)

Implement the basic structure for storing course information:

```cpp

class CourseInfo {
private:
    int courseId;       // Mutable
    float marks;        // Mutable

public:
    // Constructor
    CourseInfo(int courseId = 0, float marks = 0.0);

    // Getters
    int getCourseId() ;
    float getMarks();

    // Setters
    void setCourseId(int newCourseId);
    void setMarks(float newMarks);

};
```

Module 1.2 Node Implementation (node.cpp, node.h)

Implement the basic structure for Node information:

```cpp

#include "courseinfo.h"
#include <string>

class Node {
private:
    int rollNo;             // immutable
    std::string name;       // immutable
    float cgpa;             // Mutable
    int numSubjects;        // Mutable
    CourseInfo* courses;    // Array of CourseInfo objects
    Node* previous;         // Pointer to previous node
    Node* next;           // Pointer to next node

public:
    // Constructor
    Node(int rollNo, const std::string& name, float cgpa, int numSubjects,
        CourseInfo* courses);



    // Getters
    int getRollNo() ;
    std::string getName() ;
    float getCGPA() ;
    CourseInfo* getCourses() ;
    int getNumSubjects() ;
    Node* getPrevious() ;
    Node* getNext() ;

    // Setters
    void setCGPA(float newCGPA);
    void setNumSubjects(int newNumSubjects);
    void setCourseId(int index, int newCourseId);
    void setPrevious(Node* prev);
    void setNext(Node* nxt);
};
```

Module 2: Node Operations (node_operations.cpp, node_operations.h)

Implement all operations on the linked list:

- Insert new student node
- Delete existing student node
- Modify student CGPA
- Add new course to student record
- Modify course marks
- Delete course from student record
- Search operations

Essential operations for managing student records:

```cpp

#include "node.h"

class NodeOperations {
private:
    static Node* findStudent(Node* head, int rollNo);

public:
    // Student Operations
    static bool insertStudent(Node*& head, int rollNo, const std::string& name,
                                float cgpa, int numSubjects, CourseInfo* courses);
    static bool deleteStudent(Node*& head, int rollNo);
    static bool modifyStudentCGPA(Node* head, int rollNo, float newCGPA);
    static bool modifyStudentSubjects(Node* head, int rollNo, int
        newNumSubjects);

    // Course Operations
    static bool addCourse(Node* head, int rollNo, int courseId, float marks);
    static bool modifyCourse(Node* head, int rollNo, int courseId, float
        newMarks);
    static bool modifyCourseId(Node* head, int rollNo, int oldCourseId, int
        newCourseId);
    static bool deleteCourse(Node* head, int rollNo, int courseId);

    // Display Function
    static void displayAll(Node* head);
};
```

Module 3: File Handler (filehandler.cpp, filehandler.h)

Implement file reading and command processing operations:

- Read input file
- Parse different sections of the input
- Execute appropriate operations based on commands
- Handle file format validation

```cpp
#include "node_operations.h"
#include <string>
#include <vector>

class FileHandler {
private:
    Node*& head;

    // Helper functions for parsing data
    std::vector<std::string> splitAndTrim(const std::string& str);
    bool parseStudentData(const std::string& line, int& rollNo, std::string&
        name, float& cgpa, int& numSubjects);
    bool parseCourseData(const std::string& line, int& courseId, float& marks);
    bool parseModifyData(const std::string& line, int& id, float& value);

    // Helper functions for each operation type
    void handleInitialList(const std::vector<std::string>& lines);
    void handleAddStudent(const std::string& line);
    void handleAddCourse(const std::string& line);
    void handleModifyStudent(const std::string& line);
    void handleModifyCourse(const std::string& line);
    void handleDeleteStudent(const std::string& line);
    void handleDeleteCourse(const std::string& line);

public:
     FileHandler(Node*& head) : head(head) {}
    bool processFile(const std::string& filename);
};
```

Module 4: Main Program (main.cpp)

```
#include "filehandler.h"
#include "node_operations.h"
#include <iostream>
#include <string>

using namespace std;


int main(int argc, char* argv[]) {
    // Check command line arguments
    if (argc != 2) {
        cout<<"check␣command␣line␣arg"<<endl;
        return 1;
    }

    // Initialize head pointer
    Node* head = nullptr;

    // Create file handler with head pointer
    FileHandler fileHandler(head);


    if (!fileHandler.processFile(argv[1])) {
        cout << "Error" << argv[1] << endl;

        return 1;
    }

    cout << "File␣processed␣successfully." << endl;
    return 0;
}
```

Input File Format

The input file contains different sections marked by headers:

```
# initial list
Roll No., Name, CGPA, Number of Subjects
Course Code1, Marks
Course Code2, Marks
...
Course CodeN, Marks

# add student
Roll No., Name, CGPA, Number of subjects

# add course
Roll No., Course Code, Marks

# modify student
Roll No., CGPA

# modify course
Roll No., Course Code, Marks

# delete student
Roll No.

# delete course
Roll No., Course Code
```

Sample Input

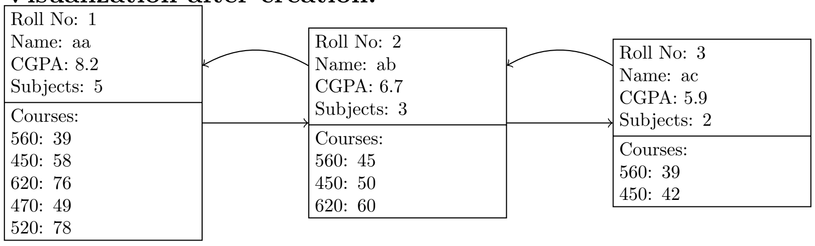
```
# initial list
1, aa, 8.2, 5
560, 39
450, 58
620, 76
470, 49
520, 78
2, ab, 6.7, 3
560, 45
450, 50
620, 60
3, ac, 5.9, 2
560, 39
450, 42
# add student
4, ad, 7.9, 3
5, ae, 7.1, 3
# add course
4, 520, 81
5, 560, 70
# modify student
1, 8.1
# modify course
5, 560, 76
# delete student
2
# delete course
1, 520
```

Student Database Operations and Visualizations

1. Initial Database Creation

```
# initial list
1, aa, 8.2, 5
560, 39
450, 58
620, 76
470, 49
520, 78
2, ab, 6.7, 3
560, 45
450, 50
620, 60
3, ac, 5.9, 2
560, 39
450, 42
```

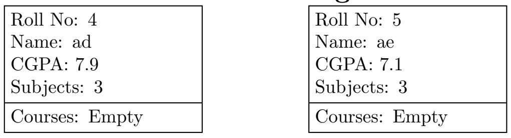
Visualization after creation:



2. Add Students

```
# add student
4, ad, 7.9, 3
5, ae, 7.1, 3
```

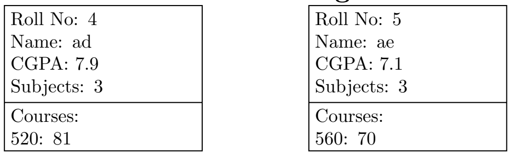
Visualization after adding students:



3. Add Courses

```
# add course
4, 520, 81
5, 560, 70
```

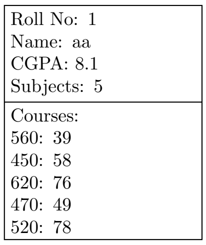
Visualization after adding courses:



4. Modify Student

```
# modify student
1, 8.1
```

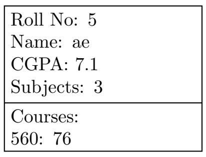
Visualization after modifying student:



5. Modify Course

```
# modify course
5, 560, 76
```

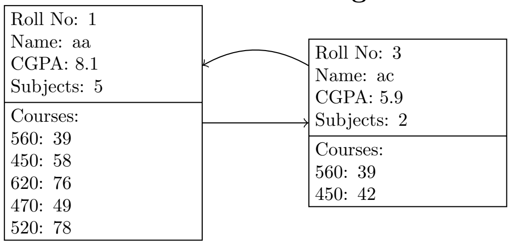
Visualization after modifying course:



6. Delete Student

```
# delete student
2
```

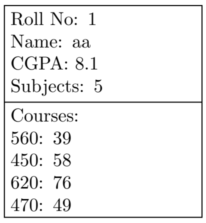
Visualization after deleting student:



7. Delete Course

```
# delete course
1, 520
```

Visualization after deleting course:



Additional Notes

1. All necessary starter files are provided in the zip format, which includes:
   - Header files (.h) with class definitions
   - Implementation files (.cpp) with detailed comments
   - Sample input file and Makefile
2. Students have the freedom to:
   - Add new methods to any module
   - Modify existing method signatures
   - Remove provided methods
   - Restructure internal implementations

   However, any modifications must:
   - Maintain the modular programming approach
   - Ensure all required functionalities are implemented
   - Follow good coding practices with proper documentation
3. The program must be compiled and run using the provided Makefile
   Command usage:

   ```
   make        # to compile the program
   make clean  # to remove compiled files
   make run  # to run the program
   ```