Final Report: Student Grade Management System

1. Summary of Features Implemented

- **Student Record Management**:
  - Add new students with 7-digit ID validation.
  - Remove students by ID.
  - Search for students by ID.

- **Grade Calculations**:

  - Automatic GPA calculation using a comprehensive grading scale
  - Update student grades either by ID or name
  - Support for 5 subjects per student

- **Data Persistence**:
  - Save and load student records to/from CSV files
  - Automatic header handling in CSV files

- **Sorting Functionality**:
  - Sort students by ID
  - Sort students by name
  - Sort students by GPA

- **Visualization**:
  - Interactive GPA bar charts for all students
  - Subject grade bar charts for individual students

- **Memory Management**:
  - Proper allocation and deallocation of linked list nodes
  - Cleanup of all memory on program exit

  User interface:
  - Ask the user to make input providing different choices

  can be selected for different operations stated above.

2. Use of Key Programming Concepts

A **`struct`** is like a custom data container that holds different types of related information together. For example, in a student record, you can store ID (int), name (string), grades (array), and GPA (float) in one single unit. It makes managing complex data easier. Instead of handling separate variables, you deal with one grouped object.

```
typedef struct Student {
    int id;
    char name[50];
    float grades[NUM_SUBJECTS];
    float gpa;
} Student;
```

A **`linked list`** is a dynamic chain of nodes where each node stores data and a pointer to the next node. In this case, each node holds a student struct. It allows flexible addition or removal of students without moving other elements. Linked lists are ideal when the number of items changes frequently.

```
void removeStudent(StudentNode** head, int id) {
    StudentNode *temp = *head, *prev = NULL;

    while (temp && temp->student.id != id) {
        prev = temp;
        temp = temp->next;
    }

    if (!temp) {
        printf("ID not found.\n");
        return;
    }

    if (!prev) {
        *head = temp->next;
    } else {
        prev->next = temp->next;
    }

    free(temp);
    printf("Student removed.\n");
}
```

An **`array`** stores a fixed number of elements of the same type, like a row of lockers. In the student struct, an array holds grades for 5 subjects. You can access each grade using an index (e.g., `grades[0]`, `grades[1]`). Arrays are fast and simple when you know the number of items ahead of time.

All operations (add, remove, search) traverse this linked list. For example, the add function:

```c
void addStudent(StudentNode** head) {
    Student newStudent;

    while (1) {
        printf("Enter student ID (7-digit): ");
        if (scanf("%d", &newStudent.id) != 1 || newStudent.id < 1000000 || newStudent.id > 9999999) {
            printf("Invalid ID. Must be a 7-digit number.\n");
            while (getchar() != '\n');
            continue;
        }

        StudentNode* temp = *head;
        int exists = 0;
        while (temp) {
            if (temp->student.id == newStudent.id) {
                exists = 1;
                break;
            }
            temp = temp->next;
        }
        if (exists) {
            printf("Student ID already exists. Try again.\n");
        } else {
            break;
        }
    }
```

We implemented CSV file handling for data persistence:

The `saveToCSV` function uses **file handling** to write student data into a `.csv` (Comma-Separated Values) file. It loops through the linked list and prints each student's ID, name, grades, and GPA in a structured, comma-separated format. This allows the data to be saved externally and opened later in programs like Excel.

```c
void saveToCSV(StudentNode* head, const char* filename) {
    FILE* f = fopen(filename, "w");
    if (!f) {
        printf("Error saving file.\n");
        return;
    }

    fprintf(f, "ID,Name,Grade1,Grade2,Grade3,Grade4,Grade5,GPA\n");
    while (head) {
        fprintf(f, "%d,%s", head->student.id, head->student.name);
        for (int i = 0; i < NUM_SUBJECTS; i++) fprintf(f, ",%.2f", head->student.grades[i]);
        fprintf(f, ",%.2f\n", head->student.gpa);
        head = head->next;
    }

    fclose(f);
    printf("Data saved to %s.\n", filename);
}
```

We implemented a quick sort for linked lists with different comparison criteria:

The **Quick Sort** implementation sorts students in a linked list based on ID, name, or GPA. It uses a **divide-and-conquer** approach by choosing a pivot, partitioning nodes into smaller (before) and greater (after) sublists, and recursively sorting them. This allows efficient sorting without converting the linked list to an array.

Memory Allocation:

The program uses **dynamic memory allocation** to create space for each new student record during runtime. It uses the `malloc()` function to allocate memory for a new `StudentNode` whenever a student is added. After all operations are done, the `freeStudents()` function is used to release the allocated memory, preventing memory leaks. Use of valgrind to verify memory safety.

4. Testing & Validation

We tested the system with various scenarios:

- **Input Validation**:
    - invalid student IDs (non-7-digit numbers)
    - Rejects duplicate student IDs
    - validates grade inputs (0-100 range)
- **Edge Cases**:
    - Empty student list handling
    - CSV file corruption handling
    - Grade boundary cases
- **Functionality Tests**:
    - Add/remove sequence verification
    - Sorting consistency checks
    - GPA calculation accuracy

5. Challenges Faced and Lessons Learned

Technical Challenges

1. **Linked List Sorting**: Implementing efficient sorting for linked lists was more complex than array sorting. Initially, we worked with array based sorting but as we learned about the new concepts such as linked list, it was bit complex to convert the method with regards to complete working of the linked list.
2. **Memory Management**: Ensuring all nodes were properly freed required careful attention, especially after removal operations. At the beginning, we observed memory leak several times but after working on that problems and methods to ensure the memory is freed and it can pass the valgrind test.
3. **CSV Parsing**: Handling edge cases in CSV file parsing (quoted strings, missing fields) was more complex than initially anticipated.

Conclusion

Despite the challenges, we successfully implemented a functional student grade management system showing core C programming concepts. The system provides extensive student record management with data deletion, search and visualization functions. The project provided us with valuable experience in system design, memory management and teamwork during obstacles.