Develop the Student Performance Tracker
Theme: Arrays and Complexity Analysis

GitHub:

https://github.com/mayank24000/University_Assignments/tree/main/Ds_%20Assignments/Lab_Assignments/Lab_Assignment_3

Code:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <chrono>
#include <iomanip>

using namespace std;

// Student Record ADT
class Student {
public:
    int studentID;
    string studentName;
    float grade;
    string courseDetails;
```

```
    // Constructor
    Student() {
        studentID = -1;
        studentName = "";
        grade = 0.0;
        courseDetails = "";
    }

    Student(int id, string name, float g, string course) {
        studentID = id;
        studentName = name;
        grade = g;
        courseDetails = course;
    }

    // Display student information
    void display() {
        cout << "ID: " << studentID << ", Name: " << studentName
             << ", Grade: " << fixed << setprecision(2) << grade <<
", Course: " << courseDetails << endl;
    }
};

// Hash Table Node for collision handling (using chaining)
class HashNode {
public:
    Student data;
    HashNode* next;
```

```
    HashNode(Student s) {
        data = s;
        next = nullptr;
    }
};

// Hash Table Class
class HashTable {
private:
    static const int TABLE_SIZE = 100;
    HashNode* table[TABLE_SIZE];

    // Hash function
    int hashFunction(int key) {
        return key % TABLE_SIZE;
    }

public:
    // Constructor
    HashTable() {
        for (int i = 0; i < TABLE_SIZE; i++) {
            table[i] = nullptr;
        }
    }

    // Insert student into hash table
    void insertStudent(Student student) {
        int index = hashFunction(student.studentID);
```

```
    HashNode* newNode = new HashNode(student);

    // If no collision
    if (table[index] == nullptr) {
        table[index] = newNode;
    }
    // Handle collision using chaining
    else {
        HashNode* current = table[index];
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
    }
    cout << "Student inserted successfully!" << endl;
}

// Search student by ID
Student* searchByID(int id) {
    int index = hashFunction(id);
    HashNode* current = table[index];

    while (current != nullptr) {
        if (current->data.studentID == id) {
            return &(current->data);
        }
        current = current->next;
    }
    return nullptr;
```

```
    }

    // Display all students
    void displayAll() {
        cout << "\n--- All Students in Hash Table ---" << endl;
        for (int i = 0; i < TABLE_SIZE; i++) {
            HashNode* current = table[i];
            while (current != nullptr) {
                current->data.display();
                current = current->next;
            }
        }
    }
};

// Student Performance Tracker System
class StudentPerformanceTracker {
private:
    HashTable hashTable;
    vector<Student> studentArray;

public:
    // Add a new student record
    void addStudentRecord() {
        int id;
        string name, course;
        float grade;

        cout << "\nEnter Student ID: ";
```

```
        while (!(cin >> id)) {
            cin.clear();
            cin.ignore(10000, '\n');
            cout << "Invalid input! Enter a numeric ID: ";
        }
        cin.ignore();

        cout << "Enter Student Name: ";
        getline(cin, name);

        cout << "Enter Grade (0-100): ";
        while (!(cin >> grade) || grade < 0 || grade > 100) {
            cin.clear();
            cin.ignore(10000, '\n');
            cout << "Invalid input! Enter a numeric grade (0-100):
";
        }
        cin.ignore();

        cout << "Enter Course Details: ";
        getline(cin, course);

        Student newStudent(id, name, grade, course);
        hashTable.insertStudent(newStudent);
        studentArray.push_back(newStudent);
    }

    // Sequential Search by name
    void sequentialSearchByName(string name) {
```

```
        bool found = false;
        cout << "\n--- Sequential Search Results ---" << endl;

        for (int i = 0; i < studentArray.size(); i++) {
            if (studentArray[i].studentName == name) {
                studentArray[i].display();
                found = true;
            }
        }

        if (!found) {
            cout << "Student not found!" << endl;
        }
    }

    // Binary Search by ID (requires sorted array)
    void binarySearchByID(int id) {
        // First, sort by ID
        vector<Student> sortedArray = studentArray;

        // Simple bubble sort by ID for sorting
        for (int i = 0; i < sortedArray.size() - 1; i++) {
            for (int j = 0; j < sortedArray.size() - i - 1; j++) {
                if (sortedArray[j].studentID > sortedArray[j +
1].studentID) {
                    swap(sortedArray[j], sortedArray[j + 1]);
                }
            }
        }
```

```
    // Binary search
    int left = 0;
    int right = sortedArray.size() - 1;
    bool found = false;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (sortedArray[mid].studentID == id) {
            cout << "\n--- Binary Search Result ---" << endl;
            sortedArray[mid].display();
            found = true;
            break;
        }

        if (sortedArray[mid].studentID < id) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    if (!found) {
        cout << "Student not found!" << endl;
    }
}

// Bubble Sort by grades
```

```
void bubbleSort(vector<Student>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j].grade > arr[j + 1].grade) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

// Insertion Sort by grades
void insertionSort(vector<Student>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        Student key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j].grade > key.grade) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Merge Sort helper function
void merge(vector<Student>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
```

```
int n2 = right - mid;

vector<Student> leftArr(n1), rightArr(n2);

for (int i = 0; i < n1; i++)
    leftArr[i] = arr[left + i];
for (int j = 0; j < n2; j++)
    rightArr[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {
    if (leftArr[i].grade <= rightArr[j].grade) {
        arr[k] = leftArr[i];
        i++;
    } else {
        arr[k] = rightArr[j];
        j++;
    }
    k++;
}

while (i < n1) {
    arr[k] = leftArr[i];
    i++;
    k++;
}

while (j < n2) {
```

```
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

// Merge Sort by grades
void mergeSort(vector<Student>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Quick Sort partition
int partition(vector<Student>& arr, int low, int high) {
    float pivot = arr[high].grade;
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j].grade < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
```

```
    return i + 1;
}

// Quick Sort by grades
void quickSort(vector<Student>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Heapify for heap sort
void heapify(vector<Student>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left].grade > arr[largest].grade)
        largest = left;

    if (right < n && arr[right].grade > arr[largest].grade)
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
```

```
}

// Heap Sort by grades (for ranking)
void heapSort(vector<Student>& arr) {
    int n = arr.size();

    // Build heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

// Sort by grades using selected algorithm
void sortByGrades(int choice) {
    if (studentArray.empty()) {
        cout << "No students to sort!" << endl;
        return;
    }

    vector<Student> sortedArray = studentArray;

    auto start = chrono::high_resolution_clock::now();

    switch (choice) {
```

```
            case 1:
                bubbleSort(sortedArray);
                cout << "Sorted using Bubble Sort" << endl;
                break;
            case 2:
                insertionSort(sortedArray);
                cout << "Sorted using Insertion Sort" << endl;
                break;
            case 3:
                mergeSort(sortedArray, 0, sortedArray.size() - 1);
                cout << "Sorted using Merge Sort" << endl;
                break;
            case 4:
                quickSort(sortedArray, 0, sortedArray.size() - 1);
                cout << "Sorted using Quick Sort" << endl;
                break;
            case 5:
                heapSort(sortedArray);
                cout << "Sorted using Heap Sort" << endl;
                break;
            default:
                cout << "Invalid sorting choice!" << endl;
                return;
        }

        auto end = chrono::high_resolution_clock::now();
        auto duration =
chrono::duration_cast<chrono::microseconds>(end - start);
```

```
        cout << "\n--- Sorted Students by Grade ---" << endl;
        for (int i = 0; i < sortedArray.size(); i++) {
            sortedArray[i].display();
        }

        cout << "\nTime taken: " << duration.count() << "
microseconds" << endl;
    }

    // Rank students by performance using heap sort
    void rankByPerformance() {
        if (studentArray.empty()) {
            cout << "No students to rank!" << endl;
            return;
        }

        vector<Student> rankedArray = studentArray;
        heapSort(rankedArray);

        cout << "\n--- Student Rankings (Highest to Lowest) ---" <<
endl;
        for (int i = rankedArray.size() - 1; i >= 0; i--) {
            cout << "Rank " << (rankedArray.size() - i) << ": ";
            rankedArray[i].display();
        }
    }

    // Compare sorting algorithms complexity
    void compareSortingComplexity() {
```

```
        if (studentArray.size() < 2) {
            cout << "Need at least 2 students to compare sorting
algorithms!" << endl;
            return;
        }

        cout << "\n--- Sorting Algorithm Time Complexity Comparison
---" << endl;
        cout << setw(20) << "Algorithm" << setw(20) << "Time
(microseconds)" << endl;
        cout << string(40, '-') << endl;

        vector<Student> testArray;

        // Bubble Sort
        testArray = studentArray;
        auto start = chrono::high_resolution_clock::now();
        bubbleSort(testArray);
        auto end = chrono::high_resolution_clock::now();
        auto duration =
chrono::duration_cast<chrono::microseconds>(end - start);
        cout << setw(20) << "Bubble Sort" << setw(20) <<
duration.count() << endl;

        // Insertion Sort
        testArray = studentArray;
        start = chrono::high_resolution_clock::now();
        insertionSort(testArray);
        end = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::microseconds>(end -
start);
```

```
        cout << setw(20) << "Insertion Sort" << setw(20) <<
duration.count() << endl;


        // Merge Sort
        testArray = studentArray;
        start = chrono::high_resolution_clock::now();
        mergeSort(testArray, 0, testArray.size() - 1);
        end = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::microseconds>(end -
start);
        cout << setw(20) << "Merge Sort" << setw(20) <<
duration.count() << endl;


        // Quick Sort
        testArray = studentArray;
        start = chrono::high_resolution_clock::now();
        quickSort(testArray, 0, testArray.size() - 1);
        end = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::microseconds>(end -
start);
        cout << setw(20) << "Quick Sort" << setw(20) <<
duration.count() << endl;


        // Heap Sort
        testArray = studentArray;
        start = chrono::high_resolution_clock::now();
        heapSort(testArray);
        end = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::microseconds>(end -
start);
```

```
        cout << setw(20) << "Heap Sort" << setw(20) <<
duration.count() << endl;


        cout << "\n--- Theoretical Time Complexities ---" << endl;
        cout << "Bubble Sort:    O(n²) average and worst case" <<
endl;
        cout << "Insertion Sort: O(n²) average and worst case, O(n)
best case" << endl;
        cout << "Merge Sort:     O(n log n) all cases" << endl;
        cout << "Quick Sort:     O(n log n) average, O(n²) worst
case" << endl;
        cout << "Heap Sort:      O(n log n) all cases" << endl;
    }


    // Search menu
    void searchMenu() {
        int choice;
        cout << "\n--- Search Menu ---" << endl;
        cout << "1. Sequential Search by Name" << endl;
        cout << "2. Binary Search by ID" << endl;
        cout << "3. Hash Table Search by ID" << endl;
        cout << "Enter choice: ";
        cin >> choice;


        switch (choice) {
            case 1: {
                string name;
                cin.ignore();
                cout << "Enter student name: ";
                getline(cin, name);
```

```
                sequentialSearchByName(name);
                break;
            }
            case 2: {
                int id;
                cout << "Enter student ID: ";
                cin >> id;
                binarySearchByID(id);
                break;
            }
            case 3: {
                int id;
                cout << "Enter student ID: ";
                cin >> id;
                Student* result = hashTable.searchByID(id);
                if (result != nullptr) {
                    cout << "\n--- Hash Table Search Result ---" <<
endl;
                    result->display();
                } else {
                    cout << "Student not found!" << endl;
                }
                break;
            }
            default:
                cout << "Invalid choice!" << endl;
        }
    }
```

```
    // Display all students
    void displayAllStudents() {
        if (studentArray.empty()) {
            cout << "No students in the system!" << endl;
            return;
        }

        cout << "\n--- All Students ---" << endl;
        for (int i = 0; i < studentArray.size(); i++) {
            studentArray[i].display();
        }
    }

    // Add sample data for testing
    void addSampleData() {
        studentArray.push_back(Student(101, "Alice Johnson", 85.5,
"Computer Science"));
        studentArray.push_back(Student(102, "Bob Smith", 92.3,
"Mathematics"));
        studentArray.push_back(Student(103, "Charlie Brown", 78.9,
"Physics"));
        studentArray.push_back(Student(104, "Diana Prince", 88.7,
"Chemistry"));
        studentArray.push_back(Student(105, "Eva Green", 95.2,
"Biology"));

        for (int i = 0; i < studentArray.size(); i++) {
            hashTable.insertStudent(studentArray[i]);
        }
```

```
        cout << "Sample data added successfully!" << endl;
    }
};

// Main function
int main() {
    StudentPerformanceTracker tracker;
    int choice;

    cout << "===== STUDENT PERFORMANCE TRACKER SYSTEM =====" <<
endl;

    while (true) {
        cout << "\n--- Main Menu ---" << endl;
        cout << "1. Add Student Record" << endl;
        cout << "2. Search Student" << endl;
        cout << "3. Sort Students by Grade" << endl;
        cout << "4. Rank Students by Performance" << endl;
        cout << "5. Compare Sorting Algorithms" << endl;
        cout << "6. Display All Students" << endl;
        cout << "7. Add Sample Data (for testing)" << endl;
        cout << "8. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                tracker.addStudentRecord();
                break;
```

```
case 2:
    tracker.searchMenu();
    break;

case 3: {
    cout << "\n--- Sorting Options ---" << endl;
    cout << "1. Bubble Sort" << endl;
    cout << "2. Insertion Sort" << endl;
    cout << "3. Merge Sort" << endl;
    cout << "4. Quick Sort" << endl;
    cout << "5. Heap Sort" << endl;
    cout << "Enter sorting choice: ";
    int sortChoice;
    cin >> sortChoice;
    tracker.sortByGrades(sortChoice);
    break;
}

case 4:
    tracker.rankByPerformance();
    break;

case 5:
    tracker.compareSortingComplexity();
    break;

case 6:
    tracker.displayAllStudents();
```

```
                break;

            case 7:
                tracker.addSampleData();
                break;

            case 8:
                cout << "Thank you for using Student Performance
Tracker!" << endl;
                return 0;

            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    }

    return 0;
}
```

OUTPUT:

```
===== STUDENT PERFORMANCE TRACKER SYSTEM =====

--- Main Menu ---
1. Add Student Record
2. Search Student
3. Sort Students by Grade
4. Rank Students by Performance
5. Compare Sorting Algorithms
6. Display All Students
7. Add Sample Data (for testing)
8. Exit
Enter your choice: 1

Enter Student ID: 2405
Enter Student Name: Mayank Rawat
Enter Grade (0-100): 99
Enter Course Details: Bsc(Hon)Cybersecurity
Student inserted successfully!

--- Main Menu ---
1. Add Student Record
2. Search Student
3. Sort Students by Grade
4. Rank Students by Performance
5. Compare Sorting Algorithms
6. Display All Students
7. Add Sample Data (for testing)
8. Exit
Enter your choice:
```