Develop the Student Result Management System
Theme: Trees, Hashing, and Sorting

GitHub:

https://github.com/mayank24000/University_Assignments/tree/main/Ds_%20Assignments/Theory_Assignments/Theory_Assignment_3

Code:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <queue>
#include <stack>
#include <cmath>

using namespace std;

class Student {
public:
    int rollNumber;
    string name;
    float marks[5];  // Marks for 5 subjects
```

```
float totalMarks;
float percentage;
string grade;

Student() {
    rollNumber = 0;
    name = "";
    totalMarks = 0;
    percentage = 0;
    grade = "";
    for(int i = 0; i < 5; i++) {
        marks[i] = 0;
    }
}

Student(int roll, string n) {
    rollNumber = roll;
    name = n;
    totalMarks = 0;
    percentage = 0;
    grade = "";
    for(int i = 0; i < 5; i++) {
        marks[i] = 0;
    }
}

void calculateResult() {
    totalMarks = 0;
    for(int i = 0; i < 5; i++) {
```

```
            totalMarks += marks[i];
        }
        percentage = (totalMarks / 500.0) * 100;

        // Assign grade based on percentage
        if(percentage >= 90) grade = "A+";
        else if(percentage >= 80) grade = "A";
        else if(percentage >= 70) grade = "B+";
        else if(percentage >= 60) grade = "B";
        else if(percentage >= 50) grade = "C";
        else if(percentage >= 40) grade = "D";
        else grade = "F";
    }

    void display() {
        cout << "Roll No: " << rollNumber << " | Name: " << setw(20)
<< left << name
             << " | Total: " << setw(6) << totalMarks
             << " | Percentage: " << setw(6) << fixed <<
setprecision(2) << percentage
             << "% | Grade: " << grade << endl;
    }

    void displayDetailed() {
        cout << "\n----- Student Details -----" << endl;
        cout << "Roll Number: " << rollNumber << endl;
        cout << "Name: " << name << endl;
        cout << "Subject Marks:" << endl;
        string subjects[5] = {"Math", "Science", "English",
"History", "Computer"};
```

```
        for(int i = 0; i < 5; i++) {
            cout << "  " << subjects[i] << ": " << marks[i] <<
"/100" << endl;
        }
        cout << "Total Marks: " << totalMarks << "/500" << endl;
        cout << "Percentage: " << fixed << setprecision(2) <<
percentage << "%" << endl;
        cout << "Grade: " << grade << endl;
    }
};

class BSTNode {
public:
    Student data;
    BSTNode* left;
    BSTNode* right;

    BSTNode(Student s) {
        data = s;
        left = nullptr;
        right = nullptr;
    }
};

class StudentBST {
private:
    BSTNode* root;

    BSTNode* insertHelper(BSTNode* node, Student s) {
```

```
    if(node == nullptr) {
        return new BSTNode(s);
    }

    if(s.rollNumber < node->data.rollNumber) {
        node->left = insertHelper(node->left, s);
    }
    else if(s.rollNumber > node->data.rollNumber) {
        node->right = insertHelper(node->right, s);
    }

    return node;
}

BSTNode* searchHelper(BSTNode* node, int roll) {
    if(node == nullptr || node->data.rollNumber == roll) {
        return node;
    }

    if(roll < node->data.rollNumber) {
        return searchHelper(node->left, roll);
    }

    return searchHelper(node->right, roll);
}

void inorderHelper(BSTNode* node, vector<Student>& result) {
    if(node != nullptr) {
        inorderHelper(node->left, result);
```

```
        result.push_back(node->data);
        inorderHelper(node->right, result);
    }
}

BSTNode* findMin(BSTNode* node) {
    while(node->left != nullptr) {
        node = node->left;
    }
    return node;
}

BSTNode* deleteHelper(BSTNode* node, int roll) {
    if(node == nullptr) {
        return node;
    }

    if(roll < node->data.rollNumber) {
        node->left = deleteHelper(node->left, roll);
    }
    else if(roll > node->data.rollNumber) {
        node->right = deleteHelper(node->right, roll);
    }
    else {
        // Node with only one child or no child
        if(node->left == nullptr) {
            BSTNode* temp = node->right;
            delete node;
            return temp;
```

```
            }
            else if(node->right == nullptr) {
                BSTNode* temp = node->left;
                delete node;
                return temp;
            }

            // Node with two children
            BSTNode* temp = findMin(node->right);
            node->data = temp->data;
            node->right = deleteHelper(node->right, temp-
>data.rollNumber);
        }

        return node;
    }

public:
    StudentBST() {
        root = nullptr;
    }

    void insert(Student s) {
        root = insertHelper(root, s);
    }

    Student* search(int roll) {
        BSTNode* result = searchHelper(root, roll);
        if(result != nullptr) {
```

```
            return &(result->data);
        }
        return nullptr;
    }

    void deleteStudent(int roll) {
        root = deleteHelper(root, roll);
    }

    vector<Student> getAllStudents() {
        vector<Student> result;
        inorderHelper(root, result);
        return result;
    }
};

// ==================== HASH TABLE FOR QUICK LOOKUP
====================

class HashTable {
private:
    static const int TABLE_SIZE = 100;
    vector<Student>* table;

    int hashFunction(int key) {
        return key % TABLE_SIZE;
    }

public:
```

```
HashTable() {
    table = new vector<Student>[TABLE_SIZE];
}

void insert(Student s) {
    int index = hashFunction(s.rollNumber);

    // Check if student already exists
    for(int i = 0; i < table[index].size(); i++) {
        if(table[index][i].rollNumber == s.rollNumber) {
            table[index][i] = s;  // Update existing
            return;
        }
    }

    // Add new student
    table[index].push_back(s);
}

Student* search(int roll) {
    int index = hashFunction(roll);

    for(int i = 0; i < table[index].size(); i++) {
        if(table[index][i].rollNumber == roll) {
            return &table[index][i];
        }
    }

    return nullptr;
```

```
    }

    void remove(int roll) {
        int index = hashFunction(roll);

        for(int i = 0; i < table[index].size(); i++) {
            if(table[index][i].rollNumber == roll) {
                table[index].erase(table[index].begin() + i);
                return;
            }
        }
    }

    void displayAll() {
        cout << "\n--- All Students in Hash Table ---" << endl;
        for(int i = 0; i < TABLE_SIZE; i++) {
            for(int j = 0; j < table[i].size(); j++) {
                table[i][j].display();
            }
        }
    }
};

class SortingAlgorithms {
public:
    // Bubble Sort by Roll Number
    static void bubbleSortByRoll(vector<Student>& students) {
        int n = students.size();
        for(int i = 0; i < n-1; i++) {
```

```
            for(int j = 0; j < n-i-1; j++) {
                if(students[j].rollNumber > 
students[j+1].rollNumber) {
                    swap(students[j], students[j+1]);
                }
            }
        }
    }

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

            while(j >= 0 && students[j].totalMarks < key.totalMarks) 
{
                students[j+1] = students[j];
                j--;
            }
            students[j+1] = key;
        }
    }

    // Quick Sort by Percentage
    static int partition(vector<Student>& students, int low, int 
high) {
        float pivot = students[high].percentage;
        int i = low - 1;
```

```
        for(int j = low; j < high; j++) {
            if(students[j].percentage >= pivot) {
                i++;
                swap(students[i], students[j]);
            }
        }
        swap(students[i+1], students[high]);
        return i + 1;
    }

    static void quickSortByPercentage(vector<Student>& students, int
low, int high) {
        if(low < high) {
            int pi = partition(students, low, high);
            quickSortByPercentage(students, low, pi - 1);
            quickSortByPercentage(students, pi + 1, high);
        }
    }

    // Merge Sort by Name
    static void merge(vector<Student>& students, 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] = students[left + i];
}
for(int j = 0; j < n2; j++) {
    rightArr[j] = students[mid + 1 + j];
}

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

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

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

while(j < n2) {
    students[k] = rightArr[j];
    j++;
    k++;
```

```
        }
    }

    static void mergeSortByName(vector<Student>& students, int left,
int right) {
        if(left < right) {
            int mid = left + (right - left) / 2;
            mergeSortByName(students, left, mid);
            mergeSortByName(students, mid + 1, right);
            merge(students, left, mid, right);
        }
    }

    // Heap Sort by Total Marks
    static void heapify(vector<Student>& students, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if(left < n && students[left].totalMarks >
students[largest].totalMarks) {
            largest = left;
        }

        if(right < n && students[right].totalMarks >
students[largest].totalMarks) {
            largest = right;
        }

        if(largest != i) {
```

```
            swap(students[i], students[largest]);
            heapify(students, n, largest);
        }
    }

    static void heapSortByMarks(vector<Student>& students) {
        int n = students.size();

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

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

// ==================== AVL TREE FOR BALANCED OPERATIONS
====================

class AVLNode {
public:
    Student data;
    AVLNode* left;
    AVLNode* right;
```

```
    int height;

    AVLNode(Student s) {
        data = s;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

class AVLTree {
private:
    AVLNode* root;

    int getHeight(AVLNode* node) {
        if(node == nullptr) return 0;
        return node->height;
    }

    int getBalance(AVLNode* node) {
        if(node == nullptr) return 0;
        return getHeight(node->left) - getHeight(node->right);
    }

    AVLNode* rotateRight(AVLNode* y) {
        AVLNode* x = y->left;
        AVLNode* T2 = x->right;

        x->right = y;
```

```
        y->left = T2;

        y->height = max(getHeight(y->left), getHeight(y->right)) +
1;
        x->height = max(getHeight(x->left), getHeight(x->right)) +
1;

        return x;
    }

    AVLNode* rotateLeft(AVLNode* x) {
        AVLNode* y = x->right;
        AVLNode* T2 = y->left;

        y->left = x;
        x->right = T2;

        x->height = max(getHeight(x->left), getHeight(x->right)) +
1;
        y->height = max(getHeight(y->left), getHeight(y->right)) +
1;

        return y;
    }

    AVLNode* insertHelper(AVLNode* node, Student s) {
        // Standard BST insertion
        if(node == nullptr) {
            return new AVLNode(s);
        }
```

```
        if(s.rollNumber < node->data.rollNumber) {
            node->left = insertHelper(node->left, s);
        } else if(s.rollNumber > node->data.rollNumber) {
            node->right = insertHelper(node->right, s);
        } else {
            return node;  // No duplicates allowed
        }

        // Update height
        node->height = 1 + max(getHeight(node->left),
getHeight(node->right));

        // Get balance factor
        int balance = getBalance(node);

        // Left Left Case
        if(balance > 1 && s.rollNumber < node->left-
>data.rollNumber) {
            return rotateRight(node);
        }

        // Right Right Case
        if(balance < -1 && s.rollNumber > node->right-
>data.rollNumber) {
            return rotateLeft(node);
        }

        // Left Right Case
```

```
        if(balance > 1 && s.rollNumber > node->left-
>data.rollNumber) {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }

        // Right Left Case
        if(balance < -1 && s.rollNumber < node->right-
>data.rollNumber) {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }

        return node;
    }

    AVLNode* searchHelper(AVLNode* node, int roll) {
        if(node == nullptr || node->data.rollNumber == roll) {
            return node;
        }

        if(roll < node->data.rollNumber) {
            return searchHelper(node->left, roll);
        }

        return searchHelper(node->right, roll);
    }

    void inorderHelper(AVLNode* node, vector<Student>& result) {
```

```
        if(node != nullptr) {
            inorderHelper(node->left, result);
            result.push_back(node->data);
            inorderHelper(node->right, result);
        }
    }

public:
    AVLTree() {
        root = nullptr;
    }

    void insert(Student s) {
        root = insertHelper(root, s);
    }

    Student* search(int roll) {
        AVLNode* result = searchHelper(root, roll);
        if(result != nullptr) {
            return &(result->data);
        }
        return nullptr;
    }

    vector<Student> getAllStudents() {
        vector<Student> result;
        inorderHelper(root, result);
        return result;
    }
```

```
};

class StudentResultManagementSystem {
private:
    StudentBST bst;
    AVLTree avl;
    HashTable hashTable;
    vector<Student> studentList;

public:
    // Add new student
    void addStudent() {
        int roll;
        string name;

        cout << "\n--- Add New Student ---" << endl;
        cout << "Enter Roll Number: ";
        cin >> roll;
        cin.ignore();

        // Check if student already exists
        if(hashTable.search(roll) != nullptr) {
            cout << "Student with roll number " << roll << " already
exists!" << endl;
            return;
        }

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

```
        Student newStudent(roll, name);

        cout << "Enter marks for 5 subjects (out of 100):" << endl;
        string subjects[5] = {"Math", "Science", "English",
"History", "Computer"};

        for(int i = 0; i < 5; i++) {
            cout << subjects[i] << ": ";
            cin >> newStudent.marks[i];

            // Validate marks
            if(newStudent.marks[i] < 0 || newStudent.marks[i] > 100)
{
                cout << "Invalid marks! Please enter between 0 and
100." << endl;
                i--;
            }
        }

        newStudent.calculateResult();

        // Add to all data structures
        bst.insert(newStudent);
        avl.insert(newStudent);
        hashTable.insert(newStudent);
        studentList.push_back(newStudent);

        cout << "Student added successfully!" << endl;
        newStudent.display();
```

```
}

// Search student by roll number
void searchStudent() {
    int roll;
    cout << "\n--- Search Student ---" << endl;
    cout << "Enter Roll Number: ";
    cin >> roll;

    cout << "\nSearching using different methods:" << endl;

    // Search in Hash Table (Fastest - O(1) average)
    auto start = clock();
    Student* result1 = hashTable.search(roll);
    auto end = clock();

    if(result1 != nullptr) {
        cout << "\n1. Hash Table Search (Time: "
             << double(end - start) / CLOCKS_PER_SEC * 1000
             << " ms):" << endl;
        result1->displayDetailed();
    }

    // Search in AVL Tree (O(log n))
    start = clock();
    Student* result2 = avl.search(roll);
    end = clock();

    if(result2 != nullptr) {
```

```
        cout << "\n2. AVL Tree Search (Time: "
             << double(end - start) / CLOCKS_PER_SEC * 1000
             << " ms):" << endl;
    }

    // Search in BST (O(log n) average, O(n) worst)
    start = clock();
    Student* result3 = bst.search(roll);
    end = clock();

    if(result3 != nullptr) {
        cout << "\n3. BST Search (Time: "
             << double(end - start) / CLOCKS_PER_SEC * 1000
             << " ms):" << endl;
    }

    if(result1 == nullptr) {
        cout << "Student not found!" << endl;
    }
}

// Update student marks
void updateStudentMarks() {
    int roll;
    cout << "\n--- Update Student Marks ---" << endl;
    cout << "Enter Roll Number: ";
    cin >> roll;

    Student* student = hashTable.search(roll);
```

```
        if(student == nullptr) {
            cout << "Student not found!" << endl;
            return;
        }

        cout << "Current marks:" << endl;
        string subjects[5] = {"Math", "Science", "English",
"History", "Computer"};
        for(int i = 0; i < 5; i++) {
            cout << subjects[i] << ": " << student->marks[i] <<
endl;
        }

        cout << "\nEnter new marks (enter -1 to keep current mark):"
<< endl;
        for(int i = 0; i < 5; i++) {
            cout << subjects[i] << ": ";
            float newMark;
            cin >> newMark;

            if(newMark != -1) {
                if(newMark >= 0 && newMark <= 100) {
                    student->marks[i] = newMark;
                } else {
                    cout << "Invalid mark! Keeping current value."
<< endl;
                }
            }
        }
```

```
    student->calculateResult();
    cout << "\nMarks updated successfully!" << endl;
    student->displayDetailed();
}

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

    cout << "\n--- Display Options ---" << endl;
    cout << "1. Sort by Roll Number (Bubble Sort)" << endl;
    cout << "2. Sort by Total Marks (Insertion Sort)" << endl;
    cout << "3. Sort by Percentage (Quick Sort)" << endl;
    cout << "4. Sort by Name (Merge Sort)" << endl;
    cout << "5. Sort by Total Marks (Heap Sort)" << endl;
    cout << "6. No Sorting" << endl;
    cout << "Enter choice: ";

    int choice;
    cin >> choice;

    vector<Student> displayList = studentList;

    switch(choice) {
        case 1:
            SortingAlgorithms::bubbleSortByRoll(displayList);
```

```
                cout << "\n--- Students Sorted by Roll Number ---"
<< endl;
                break;
            case 2:
                SortingAlgorithms::insertionSortByMarks(displayList)
;
                cout << "\n--- Students Sorted by Total Marks
(Highest First) ---" << endl;
                break;
            case 3:
                SortingAlgorithms::quickSortByPercentage(displayList
, 0, displayList.size()-1);
                cout << "\n--- Students Sorted by Percentage
(Highest First) ---" << endl;
                break;
            case 4:
                SortingAlgorithms::mergeSortByName(displayList, 0,
displayList.size()-1);
                cout << "\n--- Students Sorted by Name ---" << endl;
                break;
            case 5:
                SortingAlgorithms::heapSortByMarks(displayList);
                cout << "\n--- Students Sorted by Total Marks (Heap
Sort) ---" << endl;
                break;
            case 6:
                cout << "\n--- All Students (Unsorted) ---" << endl;
                break;
            default:
                cout << "Invalid choice!" << endl;
                return;
```

```
    }

    cout << string(100, '-') << endl;
    for(int i = 0; i < displayList.size(); i++) {
        displayList[i].display();
    }
    cout << string(100, '-') << endl;
    cout << "Total Students: " << displayList.size() << endl;
}

// Generate result statistics
void generateStatistics() {
    if(studentList.empty()) {
        cout << "No students in the system!" << endl;
        return;
    }

    cout << "\n--- Result Statistics ---" << endl;

    // Calculate statistics
    float totalPercentage = 0;
    float maxPercentage = 0;
    float minPercentage = 100;
    Student* topper = nullptr;
    Student* lowest = nullptr;

    int gradeCount[7] = {0}; // A+, A, B+, B, C, D, F

    for(int i = 0; i < studentList.size(); i++) {
```

```
            totalPercentage += studentList[i].percentage;

            if(studentList[i].percentage > maxPercentage) {
                maxPercentage = studentList[i].percentage;
                topper = &studentList[i];
            }

            if(studentList[i].percentage < minPercentage) {
                minPercentage = studentList[i].percentage;
                lowest = &studentList[i];
            }

            // Count grades
            if(studentList[i].grade == "A+") gradeCount[0]++;
            else if(studentList[i].grade == "A") gradeCount[1]++;
            else if(studentList[i].grade == "B+") gradeCount[2]++;
            else if(studentList[i].grade == "B") gradeCount[3]++;
            else if(studentList[i].grade == "C") gradeCount[4]++;
            else if(studentList[i].grade == "D") gradeCount[5]++;
            else gradeCount[6]++;
        }

        float avgPercentage = totalPercentage / studentList.size();

        cout << "Total Students: " << studentList.size() << endl;
        cout << "Average Percentage: " << fixed << setprecision(2)
<< avgPercentage << "%" << endl;
        cout << "Highest Percentage: " << maxPercentage << "% (";
        if(topper) cout << topper->name;
```

```
        cout << ")" << endl;
        cout << "Lowest Percentage: " << minPercentage << "% (";
        if(lowest) cout << lowest->name;
        cout << ")" << endl;

        cout << "\nGrade Distribution:" << endl;
        string grades[7] = {"A+", "A", "B+", "B", "C", "D", "F"};
        for(int i = 0; i < 7; i++) {
            cout << grades[i] << ": " << gradeCount[i] << " students
";

            // Display bar chart
            cout << "[";
            int barLength = (gradeCount[i] * 20) /
studentList.size();
            for(int j = 0; j < barLength; j++) cout << "=";
            cout << "]" << endl;
        }

        // Pass/Fail statistics
        int passed = studentList.size() - gradeCount[6];
        float passPercentage = (passed * 100.0) /
studentList.size();

        cout << "\nPass Percentage: " << passPercentage << "%" <<
endl;
        cout << "Failed Students: " << gradeCount[6] << endl;
    }

    // Generate merit list
```

```
    void generateMeritList() {
        if(studentList.empty()) {
            cout << "No students in the system!" << endl;
            return;
        }

        vector<Student> meritList = studentList;
        SortingAlgorithms::quickSortByPercentage(meritList, 0,
meritList.size()-1);

        cout << "\n==================== MERIT LIST
====================" << endl;
        cout << "Rank | Roll No | Name                 | Total |
Percentage | Grade" << endl;
        cout << string(70, '-') << endl;

        for(int i = 0; i < min(10, (int)meritList.size()); i++) {
            cout << setw(4) << (i+1) << " | "
                 << setw(7) << meritList[i].rollNumber << " | "
                 << setw(20) << left << meritList[i].name << " | "
                 << setw(5) << right << meritList[i].totalMarks << "
| "
                 << setw(9) << fixed << setprecision(2) <<
meritList[i].percentage << "% | "
                 << meritList[i].grade << endl;
        }
        cout << string(70, '-') << endl;
    }

    // Delete student
```

```
    void deleteStudent() {
        int roll;
        cout << "\n--- Delete Student ---" << endl;
        cout << "Enter Roll Number: ";
        cin >> roll;

        Student* student = hashTable.search(roll);
        if(student == nullptr) {
            cout << "Student not found!" << endl;
            return;
        }

        cout << "Are you sure you want to delete this student?
(y/n): ";
        char confirm;
        cin >> confirm;

        if(confirm == 'y' || confirm == 'Y') {
            // Remove from all data structures
            bst.deleteStudent(roll);
            hashTable.remove(roll);

            // Remove from vector
            for(int i = 0; i < studentList.size(); i++) {
                if(studentList[i].rollNumber == roll) {
                    studentList.erase(studentList.begin() + i);
                    break;
                }
            }
```

```
        cout << "Student deleted successfully!" << endl;
    } else {
        cout << "Deletion cancelled." << endl;
    }
}

// Add sample data for testing
void addSampleData() {
    vector<Student> samples;

    // Create sample students
    Student s1(101, "Alice Johnson");
    s1.marks[0] = 85; s1.marks[1] = 90; s1.marks[2] = 78;
    s1.marks[3] = 92; s1.marks[4] = 88;
    s1.calculateResult();
    samples.push_back(s1);

    Student s2(102, "Bob Smith");
    s2.marks[0] = 75; s2.marks[1] = 82; s2.marks[2] = 88;
    s2.marks[3] = 79; s2.marks[4] = 85;
    s2.calculateResult();
    samples.push_back(s2);

    Student s3(103, "Charlie Brown");
    s3.marks[0] = 95; s3.marks[1] = 98; s3.marks[2] = 92;
    s3.marks[3] = 96; s3.marks[4] = 94;
    s3.calculateResult();
    samples.push_back(s3);
```

```
Student s4(104, "Diana Prince");
s4.marks[0] = 67; s4.marks[1] = 72; s4.marks[2] = 65;
s4.marks[3] = 70; s4.marks[4] = 68;
s4.calculateResult();
samples.push_back(s4);

Student s5(105, "Eva Green");
s5.marks[0] = 55; s5.marks[1] = 60; s5.marks[2] = 52;
s5.marks[3] = 58; s5.marks[4] = 50;
s5.calculateResult();
samples.push_back(s5);

Student s6(106, "Frank Miller");
s6.marks[0] = 88; s6.marks[1] = 85; s6.marks[2] = 90;
s6.marks[3] = 87; s6.marks[4] = 92;
s6.calculateResult();
samples.push_back(s6);

Student s7(107, "Grace Kelly");
s7.marks[0] = 78; s7.marks[1] = 75; s7.marks[2] = 80;
s7.marks[3] = 77; s7.marks[4] = 82;
s7.calculateResult();
samples.push_back(s7);

Student s8(108, "Henry Ford");
s8.marks[0] = 45; s8.marks[1] = 48; s8.marks[2] = 42;
s8.marks[3] = 50; s8.marks[4] = 40;
s8.calculateResult();
```

```
        samples.push_back(s8);

        // Add all samples to data structures
        for(int i = 0; i < samples.size(); i++) {
            bst.insert(samples[i]);
            avl.insert(samples[i]);
            hashTable.insert(samples[i]);
            studentList.push_back(samples[i]);
        }

        cout << "Sample data added successfully! " << samples.size()
<< " students added." << endl;
    }

    // Compare search performance
    void compareSearchPerformance() {
        if(studentList.empty()) {
            cout << "No students in the system!" << endl;
            return;
        }

        cout << "\n--- Search Performance Comparison ---" << endl;
        cout << "Testing search for all " << studentList.size() << "
students..." << endl;

        // Test Hash Table
        auto start = clock();
        for(int i = 0; i < studentList.size(); i++) {
            hashTable.search(studentList[i].rollNumber);
```

```
        }
        auto end = clock();
        double hashTime = double(end - start) / CLOCKS_PER_SEC *
1000;

        // Test AVL Tree
        start = clock();
        for(int i = 0; i < studentList.size(); i++) {
            avl.search(studentList[i].rollNumber);
        }
        end = clock();
        double avlTime = double(end - start) / CLOCKS_PER_SEC *
1000;

        // Test BST
        start = clock();
        for(int i = 0; i < studentList.size(); i++) {
            bst.search(studentList[i].rollNumber);
        }
        end = clock();
        double bstTime = double(end - start) / CLOCKS_PER_SEC *
1000;

        cout << "\n--- Results ---" << endl;
        cout << "Hash Table: " << hashTime << " ms (O(1) average)"
<< endl;
        cout << "AVL Tree: " << avlTime << " ms (O(log n)
guaranteed)" << endl;
        cout << "BST: " << bstTime << " ms (O(log n) average, O(n)
worst)" << endl;
```

```
        cout << "\nFastest: ";
        if(hashTime <= avlTime && hashTime <= bstTime) {
            cout << "Hash Table" << endl;
        } else if(avlTime <= bstTime) {
            cout << "AVL Tree" << endl;
        } else {
            cout << "Binary Search Tree" << endl;
        }
    }
};

int main() {
    StudentResultManagementSystem system;
    int choice;

    cout << "========================================" << endl;
    cout << "   STUDENT RESULT MANAGEMENT SYSTEM    " << endl;
    cout << "========================================" << endl;

    while(true) {
        cout << "\n--- Main Menu ---" << endl;
        cout << "1.  Add New Student" << endl;
        cout << "2.  Search Student" << endl;
        cout << "3.  Update Student Marks" << endl;
        cout << "4.  Delete Student" << endl;
        cout << "5.  Display All Students" << endl;
        cout << "6.  Generate Merit List" << endl;
        cout << "7.  Generate Statistics" << endl;
        cout << "8.  Compare Search Performance" << endl;
```

```
cout << "9.  Add Sample Data (Testing)" << endl;
cout << "10. Exit" << endl;
cout << "Enter your choice: ";
cin >> choice;

switch(choice) {
    case 1:
        system.addStudent();
        break;
    case 2:
        system.searchStudent();
        break;
    case 3:
        system.updateStudentMarks();
        break;
    case 4:
        system.deleteStudent();
        break;
    case 5:
        system.displayAllStudents();
        break;
    case 6:
        system.generateMeritList();
        break;
    case 7:
        system.generateStatistics();
        break;
    case 8:
        system.compareSearchPerformance();
```

```
                break;
            case 9:
                system.addSampleData();
                break;
            case 10:
                cout << "\nThank you for using Student Result
Management System!" << endl;
                cout << "Goodbye!" << endl;
                return 0;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    }

    return 0;
}
```

OUTPUT:

```
========================================
   STUDENT RESULT MANAGEMENT SYSTEM
========================================

--- Main Menu ---
1.  Add New Student
2.  Search Student
3.  Update Student Marks
4.  Delete Student
5.  Display All Students
6.  Generate Merit List
7.  Generate Statistics
8.  Compare Search Performance
9.  Add Sample Data (Testing)
10. Exit
Enter your choice: 1

--- Add New Student ---
Enter Roll Number: 2405
Enter Name: Mayank Rawat
Enter marks for 5 subjects (out of 100):
Math: 90
Science: 86
English: 99
History: 97
Computer: 99
Student added successfully!
Roll No: 2405 | Name: Mayank Rawat         | Total: 471    | Percentage: 94.20 % | Grade: A+

--- Main Menu ---
1.  Add New Student
2.  Search Student
3.  Update Student Marks
4.  Delete Student
5.  Display All Students
6.  Generate Merit List
7.  Generate Statistics
8.  Compare Search Performance
```