Student Grade Monitoring System

PS5

Program Architecture and Design Document

I. Project Overview

The *Student Grade Monitoring System* is a command-line application designed to manage and monitor student grades efficiently. It enables adding, updating, searching, deleting, and displaying student grade records in a fast and structured way.

This system is implemented in C++, and it uses AVL Tree as its core data structure. The AVL Tree ensures that all operations — insert, delete, update and search — are performed in O(log n) time by maintaining a balanced height after every modification while the display function has the time complexity of O(n).

Problem / Needs Addressed:
Manually managing grades for multiple students can be inefficient and error prone. This project simplifies the process by maintaining student records in an automatically balanced tree that supports fast lookups, updates, and displays in sorted order by student ID.

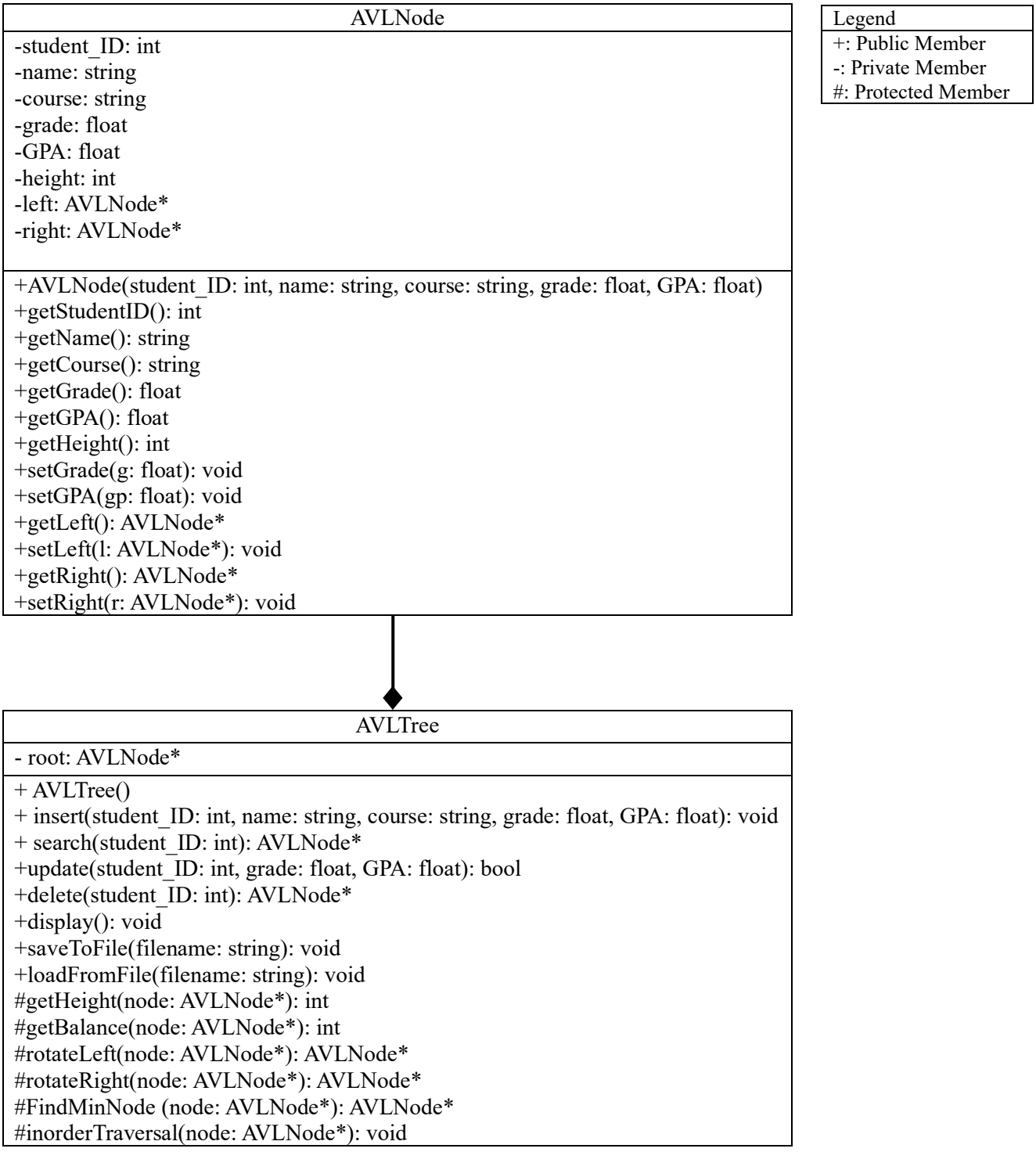
Technical Approach:

- **Primary Data Structure:** AVL Tree (self-balancing binary search tree)
- **Interface:** Command-line interface (menu-based)
- **Persistent Storage:** CSV file storing (student_ID, name, course, grade, GPA)
- **Implementation Language:** C++

Reason for Choosing AVL Tree:
A Binary Search Tree (BST) can degrade to O(n) performance when unbalanced. The AVL Tree guarantees a balanced structure through rotations after insertions and deletions, ensuring consistent logarithmic-time performance. This makes it ideal for a project where fast and reliable grade record access is essential.

II. Design of Primary Data Structure

UML diagram: -



III. Description of Critical Data Structure Operations

DSA1: Insert operation (insert)

Parameters:

- Student_ID: int – Stores unique identifier for student records (key)
- name: string – Stores Student's name
- course: string – Stores course name
- grade: float – stores student's grade
- GPA: float – stores student's GPA

Return value: void

Description: Inserts a new student record into the AVL Tree, maintaining the Binary Search Tree property. The operation is recursive and involves three main steps:

1) Standard BST Insertion.
2) Height and Balance Update.
3) Self-Balancing Rotation.

After insertion, the balance factor of the ancestors is checked. If it exceeds the AVL threshold (i.e., ±), one or two rotations (Left-Left, Right-Right, Left-Right, or Right-Left) are performed to restore balance and guarantee an O(log n) worst-case time complexity.

DSA2: Search operation (search)

Parameters:

- Student_ID: int – Stores unique identifier for student records (key) in this case is used to locate the student record.

Return Value: AVLNode* (Pointer to the found node, or null if not found)

Description: Searches for a student record by traversing the AVL Tree starting from the root. The algorithm compares the target student_ID with the current node's ID: if the target is smaller, it moves to the left subtree; otherwise, it moves to the right subtree. Due to the AVL property, the search path is logarithmic, guaranteeing an O(log n) worst-case time complexity.

DSA3: Update Operation (update)

Parameters:

- Student_ID: int – Stores unique identifier for student records (key) in this case helps to locate the student record that needs updating.
- Grade: float – Stores's student's grade, in this case carries the updated grade.
- GPA: float – Stores student's GPA, in this case carries the updated GPA.

Return Value: bool

Description: Updates the grade and GPA for an existing student record. The operation first calls the Search function (DSA2) to locate the target AVLNode using the Student_ID. If the node is found, its grade and gpa fields are modified directly, and the method returns true. If the node is not found, it returns false. This operation's efficiency is dominated by the search, making it O(log n).

DSA4: Delete operation (delete):

Parameters:

- Student_ID: int – Stores unique identifier for student records (key) in this case helps locate the to be deleted record.

Return Value: AVLNode*

Description: Deletes a student record by performing a recursive standard BST deletion. It handles all three BST cases (0, 1, or 2 children). After removing the node, the algorithm returns up the recursion stack, updating the height and checking the balance factor of all ancestor nodes. If an imbalance is found, it performs the necessary single or double rotations to ensure the AVL property is maintained. The worst-case time complexity is O(log n).

DSA5: Display Operation(display)

Parameters:

- N/A

Return value: void

Description: Displays all student records in sorted order. This is achieved by implementing the In-Order Traversal algorithm. The method recursively visits the nodes in the sequence:

Left Subtree → Current Node → Right Subtree.

Since every node must be visited and printed, this operation has a time complexity of O(N).

DSA6: Save To File Operation (saveToFile)

Parameters:

- filename: string – The name or path of the CSV file where the student grade data will be written.

Return value: void

Description: Saves all student records from the AVL tree to the designated CSV file for persistent storage. The operation uses an in-order traversal of the tree, visiting nodes in ascending order by student_ID (left subtree → current node → right subtree). Each node's data (student_ID, name, course, grade, GPA) is written as a single line in CSV format. Only actual student data is saved; tree structure information such as pointers, heights, and null references are not stored, as the tree will be automatically reconstructed and rebalanced when loaded. This approach creates a simple, human-readable CSV file with records sorted by student ID. The time complexity is O(n) since every node in the tree is visited exactly once.

DSA7: Load From File Operation (LoadFromFile)

Parameters:

- filename: string – The name or path of the CSV file containing the saved student grade data.

Return value: void

Description: Rebuilds the AVL Tree in memory from the saved student data in the CSV file. The operation reads the file sequentially, processing each line as a complete student record. For each valid record read from the file, the data fields (student_ID, name, course, grade, GPA) are parsed and passed to the Insert operation (DSA1). Since the Insert operation automatically maintains AVL balance properties through rotations after each insertion, the reconstructed tree is guaranteed to be properly balanced regardless of the order in which records appear in the file. The operation processes n student records, and each insertion takes O(log n) time due to the balanced nature of the AVL tree. Therefore, the total time complexity is O(n log n).

IV. Design of Critical Operations

DSA1: Insert operation for AVL Tree

PSEUDOCODE:

```
function Insert(root, student_ID, name, course, grade, GPA):
  // 1. Perform standard BST insertion
  if root is null:
    return new AVLNode(student_ID, name, course, grade, GPA)
  if student_ID < root.studentID:
    root.left = Insert(root.left, student_ID, name, course, grade, GPA)
  else if student_ID > root.studentID:
    root.right = Insert(root.right, student_ID, name, course, grade, GPA)
  else:
    // Key already exists (duplicates not allowed)
    return root

  // 2. Update height of the current node
  root.height = 1 + max(getHeight(root.left), getHeight(root.right))

  // 3. Get the balance factor
  balance = getBalance(root)

  // 4. Perform Rotations if unbalanced (4 Cases)

  // Left Left Case: New node inserted in left subtree of left child
  if balance > 1 AND student_ID < root.left.studentID:
    return rotateRight(root)

  // Right Right Case: New node inserted in right subtree of right child
```

```
    if balance < -1 AND student_ID > root.right.studentID:
        return rotateLeft(root)

    // Left Right Case: New node inserted in right subtree of left child
    if balance > 1 AND student_ID > root.left.studentID:
        root.left = rotateLeft(root.left)
        return rotateRight(root)

    // Right Left Case: New node inserted in left subtree of right child
    if balance < -1 AND student_ID < root.right.studentID:
        root.right = rotateRight(root.right)
        return rotateLeft(root)

    return root


// Helper Rotation Functions

function rotateRight(y):
    x = y.left
    T2 = x.right

    // Perform rotation
    x.right = y
    y.left = T2

    // Update heights
    y.height = 1 + max(getHeight(y.left), getHeight(y.right))
    x.height = 1 + max(getHeight(x.left), getHeight(x.right))
```

```
    return x

function rotateLeft(x):
    y = x.right
    T2 = y.left

    // Perform rotation
    y.left = x
    x.right = T2

    // Update heights
    x.height = 1 + max(getHeight(x.left), getHeight(x.right))
    y.height = 1 + max(getHeight(y.left), getHeight(y.right))

    return y
```

Source: "Insertion in an AVL Tree", *kartik,* ***GeeksforGeeks***, July 23, 2025, URL: https://www.geeksforgeeks.org/dsa/insertion-in-an-avl-tree/

DSA2: Search Operation for AVL Tree

PSEUDOCODE:

```
function Search(root, student_ID):
    if root is null OR root.studentID == student_ID:
        return root // Return node or null if not found

    if student_ID < root.studentID:
        // Search the left subtree
```

```
        return Search(root.left, student_ID)
    else:
        // Search the right subtree
        return Search(root.right, student_ID)
```

Source: "Search in an AVL Tree", *akshatsachan,* ***GeeksforGeeks***, July 23, 2025,

URL: https://www.geeksforgeeks.org/dsa/avl-trees-containing-a-parent-node-pointer/

DSA3: Update Operation for AVL Tree

PSEUDOCODE:

```
function Update(root, student_ID, newGrade, newGPA):
    // 1. Search for the node
    nodeToUpdate = Search(root, student_ID)

    // 2. Check if the node was found
    if nodeToUpdate is NOT null:
        // 3. Update the fields using setter methods
        nodeToUpdate.setGrade(newGrade)
        nodeToUpdate.setGPA(newGPA)
        return true // Update successful
    else:
        return false // Node not found
```

DSA4: Delete Operation for AVL Tree

PSEUDOCODE:

```
function Delete(root, student_ID):
   // 1. Standard BST Deletion (Find the node)
   if root is null:
      return root

   if student_ID < root.studentID:
      root.left = Delete(root.left, student_ID)
   else if student_ID > root.studentID:
      root.right = Delete(root.right, student_ID)
   else:
      // Found the node to be deleted

// Case 1 & 2: Node with 0 or 1 child
if root.left is null:
   return root.right
if root.right is null:
   return root.left

      // Case 3: Node with 2 children
      // Get the in-order successor (smallest in the right subtree)
      temp = FindMinNode(root.right)

      // Copy ALL data fields from the successor
      root.studentID = temp.studentID
      root.name = temp.name
      root.course = temp.course
      root.grade = temp.grade
```

```
    root.gpa = temp.gpa

    // Delete the in-order successor
    root.right  = Delete(root.right, temp.studentID)

// If the tree only had one node, return (already handled)
if root is null:
    return root

// 2. Update height and get balance
root.height  = 1 + max(getHeight(root.left), getHeight(root.right))
balance  = getBalance(root)

// 3. Perform Rotations if unbalanced (4 Cases)

// Left Left Case (Node becomes heavy on the left)
if balance > 1 AND getBalance(root.left) > 0:
    return rotateRight(root)

// Left Right Case
if balance > 1 AND getBalance(root.left) < 0:
    root.left =  rotateLeft(root.left)
    return rotateRight(root)

// Right Right Case (Node becomes heavy on the right)
if balance < -1 AND getBalance(root.right) < 0:
    return rotateLeft(root)

// Right Left Case
if balance < -1 AND getBalance(root.right) > 0:
```

```
        root.right  = rotateRight(root.right)
        return rotateLeft(root)

    return root

function FindMinNode(node):
    current = node
    while current.left is NOT null:
        current = current.left
    return current
```

Source: "Deletion in an AVL Tree", *kartik,* ***GeeksforGeeks***, June 19, 2025, URL: https://www.geeksforgeeks.org/dsa/deletion-in-an-avl-tree/

DSA5: Display Operation (In-Order Traversal) for AVL Tree

PSEUDOCODE:

```
function InOrderTraversal(node):
    if node is NOT null:
        // 1. Recurse on Left Subtree (Smaller Keys)
        InOrderTraversal(node.left)

        // 2. Visit Current Node (Print Data)
        Print "ID:", node.studentID,
            " Name:", node.name,
            " Course:", node.course,
            " Grade:", node.grade,
            " GPA:", node.gpa
```

```
    // 3. Recurse on Right Subtree (Larger Keys)
    InOrderTraversal(node.right)
```

Source: "Inorder Traversal of Binary Tree", *animeshdey,* ***GeeksforGeeks***, March 3, 2023, URL: https://www.geeksforgeeks.org/dsa/inorder-traversal-of-binary-tree/

HELPER FUNCTIONS

PSEUDOCODE:

```
function getHeight(node):
  if node is null:
    return 0
  return node.height


function getBalance(node):
  if node is null:
    return 0
  return getHeight(node.left) - getHeight(node.right)
```

Source: "Insertion in an AVL Tree", *kartik,* ***GeeksforGeeks***, July 23, 2025, URL: https://www.geeksforgeeks.org/dsa/insertion-in-an-avl-tree/

ADDITIONAL OPERATIONS (File I/O)

PSEUDOCODE:

```
// Save tree to file using in-order traversal (sorted by student ID)
function SaveToFile(root, file):
    if root is null:
        return

    // Recursively save left subtree first
    SaveToFile(root.left, file)

    // Write current node data as one CSV line
    Write root.studentID, root.name, root.course, root.grade, root.gpa to file

    // Recursively save right subtree
    SaveToFile(root.right, file)



// Load tree from file
function LoadFromFile(filename):
    root = null  // Initialize empty tree
    file = open(filename)

    while NOT end of file:
        // Read one complete CSV line (one student record)
        line = read next line from file

        if line is empty OR line is null:
            continue  // Skip empty lines

        // Parse the CSV line into fields
```

```
    student_ID, name, course, grade, gpa = parse CSV line

    // Use existing Insert function (guarantees balance)
    root = Insert(root, student_ID, name, course, grade, gpa)

close(file)
return root
```

V. Member Contributions

All three of us discussed the way we should go about this project. We finalized what all components we're going to have and laid all of them out. Once we had a vision of what and how we were going to complete the project, we documented the things we discussed as follows:

Project Overview and Design of Primary Data Structure: Prashant Chand

Description of Critical data structure operations: Sambhav Pyakurel

Design of Critical Operations: Sujal Maharjan