Algorithms:

Q1)

1. Define a struct Node with members data, left, right, and height.
2. Define a private member root of type Node\* in the AVLTree class.
3. Implement a private member function getHeight that returns the height of a given node. If the node is nullptr, return 0.
4. Implement a private member function getBalanceFactor that returns the balance factor of a given node. If the node is nullptr, return 0.
5. Implement a private member function rotateRight that performs a right rotation on a given node and returns the new root of the subtree.
6. Implement a private member function rotateLeft that performs a left rotation on a given node and returns the new root of the subtree.
7. Implement a private member function insert that inserts a new node with the given data into the AVL tree and returns the root of the modified tree.
8. Implement a private member function minValueNode that returns the node with the minimum value in a given subtree.
9. Implement a private member function deleteNode that deletes a node with the given data from the AVL tree and returns the root of the modified tree.
10. Implement a private member function inOrderTraversal that performs an in-order traversal of the tree and prints the data of each node.
11. Implement a private member function preOrderTraversal that performs a pre-order traversal of the tree and prints the data of each node.
12. Implement a private member function postOrderTraversal that performs a post-order traversal of the tree and prints the data of each node.
13. Define public member functions insert, remove, traverseInOrder, traversePreOrder, and traversePostOrder that call the corresponding private member functions.
14. In the main function, create an instance of the AVLTree class.
15. Insert some values into the AVL tree using the insert function.
16. Print the in-order traversal of the tree using the traverseInOrder function.
17. Print the pre-order traversal of the tree using the traversePreOrder function.
18. Print the post-order traversal of the tree using the traversePostOrder function.
19. Remove a value from the AVL tree using the remove function.
20. Print the in-order traversal of the tree again to verify the deletion.

Q2)

1. Define the structure for the student record and AVL tree node.

2. Implement functions to get the height and balance factor of a node, update the height of a node, perform right and left rotations.

3. Implement the function to insert a new student record into the AVL tree.

4. Implement the function to perform in-order traversal of the AVL tree.

5. Implement the function to delete a student record from the AVL tree.

6. Implement the function to free memory allocated for AVL tree nodes.

7. In the main function:

   a. Create an AVL tree root node.

   b. Read student information from a text file and build the initial AVL tree.

   c. Perform operations on the AVL tree, such as inserting a new student record.

   d. Display the in-order traversal of the AVL tree.

   e. Perform the delete operation on a specific student record.

   f. Display the in-order traversal of the AVL tree after deletion.

   g. Free memory allocated for AVL tree nodes.

AVL Tree Implementation:

Q1.

#include <iostream>

using *namespace* std;

*class* AVLTree {

*private:*

*struct* Node {

*int* data;

        Node\* left;

        Node\* right;

*int* height;

    };

    Node\* root;

*int* getHeight(Node\* *node*) {

        if (*node* == nullptr)

            return 0;

        return *node*->height;

    }

*int* getBalanceFactor(Node\* *node*) {

        if (*node* == nullptr)

            return 0;

        return getHeight(*node*->left) - getHeight(*node*->right);

    }

    Node\* rotateRight(Node\* *y*) {

        Node\* x = *y*->left;

        Node\* T2 = x->right;

        x->right = *y*;

*y*->left = T2;

*y*->height = std::max(getHeight(*y*->left), getHeight(*y*->right)) + 1;

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

        return x;

    }

    Node\* rotateLeft(Node\* *x*) {

        Node\* y = *x*->right;

        Node\* T2 = y->left;

        y->left = *x*;

*x*->right = T2;

*x*->height = std::max(getHeight(*x*->left), getHeight(*x*->right)) + 1;

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

        return y;

    }

    Node\* insert(Node\* *node*, *int* *data*) {

        if (*node* == nullptr) {

            Node\* newNode = new Node;

            newNode->data = *data*;

            newNode->left = nullptr;

            newNode->right = nullptr;

            newNode->height = 1;

            return newNode;

        }

        if (*data* < *node*->data)

*node*->left = insert(*node*->left, *data*);

        else if (*data* > *node*->data)

*node*->right = insert(*node*->right, *data*);

        else

            return *node*;

*node*->height = std::max(getHeight(*node*->left), getHeight(*node*->right)) + 1;

*int* balanceFactor = getBalanceFactor(*node*);

        if (balanceFactor > 1 && *data* < *node*->left->data)

            return rotateRight(*node*);

        if (balanceFactor < -1 && *data* > *node*->right->data)

            return rotateLeft(*node*);

        if (balanceFactor > 1 && *data* > *node*->left->data) {

*node*->left = rotateLeft(*node*->left);

            return rotateRight(*node*);

        }

        if (balanceFactor < -1 && *data* < *node*->right->data) {

*node*->right = rotateRight(*node*->right);

            return rotateLeft(*node*);

        }

        return *node*;

    }

    Node\* minValueNode(Node\* *node*) {

        Node\* current = *node*;

        while (current->left != nullptr)

            current = current->left;

        return current;

    }

    Node\* deleteNode(Node\* *root*, *int* *data*) {

        if (*root* == nullptr)

            return *root*;

        if (*data* < *root*->data)

*root*->left = deleteNode(*root*->left, *data*);

        else if (*data* > *root*->data)

*root*->right = deleteNode(*root*->right, *data*);

        else {

            if (*root*->left == nullptr || *root*->right == nullptr) {

                Node\* temp = *root*->left ? *root*->left : *root*->right;

                if (temp == nullptr) {

                    temp = *root*;

*root* = nullptr;

                } else

                    \**root* = \*temp;

                delete temp;

            } else {

                Node\* temp = minValueNode(*root*->right);

*root*->data = temp->data;

*root*->right = deleteNode(*root*->right, temp->data);

            }

        }

        if (*root* == nullptr)

            return *root*;

*root*->height = std::max(getHeight(*root*->left), getHeight(*root*->right)) + 1;

*int* balanceFactor = getBalanceFactor(*root*);

        if (balanceFactor > 1 && getBalanceFactor(*root*->left) >= 0)

            return rotateRight(*root*);

        if (balanceFactor > 1 && getBalanceFactor(*root*->left) < 0) {

*root*->left = rotateLeft(*root*->left);

            return rotateRight(*root*);

        }

        if (balanceFactor < -1 && getBalanceFactor(*root*->right) <= 0)

            return rotateLeft(*root*);

        if (balanceFactor < -1 && getBalanceFactor(*root*->right) > 0) {

*root*->right = rotateRight(*root*->right);

            return rotateLeft(*root*);

        }

        return *root*;

    }

*void* inOrderTraversal(Node\* *root*) {

        if (*root* != nullptr) {

            inOrderTraversal(*root*->left);

            std::cout << *root*->data << " ";

            inOrderTraversal(*root*->right);

        }

    }

*void* preOrderTraversal(Node\* *root*) {

        if (*root* != nullptr) {

            std::cout << *root*->data << " ";

            preOrderTraversal(*root*->left);

            preOrderTraversal(*root*->right);

        }

    }

*void* postOrderTraversal(Node\* *root*) {

        if (*root* != nullptr) {

            postOrderTraversal(*root*->left);

            postOrderTraversal(*root*->right);

            std::cout << *root*->data << " ";

        }

    }

*public:*

    AVLTree() {

        root = nullptr;

    }

*void* insert(*int* *data*) {

        root = insert(root, *data*);

    }

*void* remove(*int* *data*) {

        root = deleteNode(root, *data*);

    }

*void* traverseInOrder() {

        inOrderTraversal(root);

        std::cout << std::endl;

    }

*void* traversePreOrder() {

        preOrderTraversal(root);

        std::cout << std::endl;

    }

*void* traversePostOrder() {

        postOrderTraversal(root);

        std::cout << std::endl;

    }

};

*int* main() {

    AVLTree avl;

    avl.insert(10);

    avl.insert(20);

    avl.insert(30);

    avl.insert(40);

    avl.insert(50);

    avl.insert(25);

    std::cout << "In-order traversal: ";

    avl.traverseInOrder();

    std::cout << "Pre-order traversal: ";

    avl.traversePreOrder();

    std::cout << "Post-order traversal: ";

    avl.traversePostOrder();

    avl.remove(30);

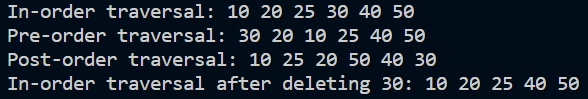
    std::cout << "In-order traversal after deleting 30: ";

    avl.traverseInOrder();

    return 0;

}

Output:



Q2.

#include <iostream>

#include <fstream>

#include <string>

using *namespace* std;

// Define structure for student record

*struct* StudentRecord {

*int* rollNumber;

    string name;

*float* cgpa;

};

// Define structure for AVL tree node

*struct* AVLNode {

    StudentRecord data;

    AVLNode\* left;

    AVLNode\* right;

*int* height;

};

// Function to get height of a node

*int* getHeight(AVLNode\* *node*) {

    if (*node* == nullptr)

        return 0;

    return *node*->height;

}

// Function to get balance factor of a node

*int* getBalanceFactor(AVLNode\* *node*) {

    if (*node* == nullptr)

        return 0;

    return getHeight(*node*->left) - getHeight(*node*->right);

}

// Function to update height of a node

*void* updateHeight(AVLNode\* *node*) {

    if (*node* != nullptr)

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

}

// Function to perform right rotation

AVLNode\* rightRotate(AVLNode\* *y*) {

    AVLNode\* x = *y*->left;

    AVLNode\* T2 = x->right;

    x->right = *y*;

*y*->left = T2;

    updateHeight(*y*);

    updateHeight(x);

    return x;

}

// Function to perform left rotation

AVLNode\* leftRotate(AVLNode\* *x*) {

    AVLNode\* y = *x*->right;

    AVLNode\* T2 = y->left;

    y->left = *x*;

*x*->right = T2;

    updateHeight(*x*);

    updateHeight(y);

    return y;

}

// Function to insert a new student record into AVL tree

AVLNode\* insertAVL(AVLNode\* *root*, StudentRecord *newStudent*) {

    if (*root* == nullptr) {

        AVLNode\* newNode = new AVLNode;

        newNode->data = *newStudent*;

        newNode->left = newNode->right = nullptr;

        newNode->height = 1;

        return newNode;

    }

    if (*newStudent*.rollNumber < *root*->data.rollNumber)

*root*->left = insertAVL(*root*->left, *newStudent*);

    else if (*newStudent*.rollNumber > *root*->data.rollNumber)

*root*->right = insertAVL(*root*->right, *newStudent*);

    else // Duplicate roll numbers are not allowed

        return *root*;

    // Update height of current node

    updateHeight(*root*);

    // Check balance factor and perform rotations if necessary

*int* balance = getBalanceFactor(*root*);

    // Left Left Case

    if (balance > 1 && *newStudent*.rollNumber < *root*->left->data.rollNumber)

        return rightRotate(*root*);

    // Right Right Case

    if (balance < -1 && *newStudent*.rollNumber > *root*->right->data.rollNumber)

        return leftRotate(*root*);

    // Left Right Case

    if (balance > 1 && *newStudent*.rollNumber > *root*->left->data.rollNumber) {

*root*->left = leftRotate(*root*->left);

        return rightRotate(*root*);

    }

    // Right Left Case

    if (balance < -1 && *newStudent*.rollNumber < *root*->right->data.rollNumber) {

*root*->right = rightRotate(*root*->right);

        return leftRotate(*root*);

    }

    return *root*;

}

// Function to perform in-order traversal of AVL tree

*void* inOrderTraversal(AVLNode\* *root*) {

    if (*root* != nullptr) {

        inOrderTraversal(*root*->left);

        cout << "Roll Number: " << *root*->data.rollNumber << ", Name: " << *root*->data.name << ", CGPA: " << *root*->data.cgpa << endl;

        inOrderTraversal(*root*->right);

    }

}

// Function to delete a student record from AVL tree

AVLNode\* deleteNodeAVL(AVLNode\* *root*, *int* *rollNumber*) {

    if (*root* == nullptr)

        return *root*;

    if (*rollNumber* < *root*->data.rollNumber)

*root*->left = deleteNodeAVL(*root*->left, *rollNumber*);

    else if (*rollNumber* > *root*->data.rollNumber)

*root*->right = deleteNodeAVL(*root*->right, *rollNumber*);

    else {

        // Node with only one child or no child

        if (*root*->left == nullptr || *root*->right == nullptr) {

            AVLNode\* temp = *root*->left ? *root*->left : *root*->right;

            // No child case

            if (temp == nullptr) {

                temp = *root*;

*root* = nullptr;

            } else // One child case

                \**root* = \*temp;

            delete temp;

        } else {

            // Node with two children, get the in-order successor (smallest in the right subtree)

            AVLNode\* temp = *root*->right;

            while (temp->left != nullptr)

                temp = temp->left;

            // Copy the in-order successor's data to this node

*root*->data = temp->data;

            // Delete the in-order successor

*root*->right = deleteNodeAVL(*root*->right, temp->data.rollNumber);

        }

    }

    // If the tree had only one node then return

    if (*root* == nullptr)

        return *root*;

    // Update height of current node

    updateHeight(*root*);

    // Check balance factor and perform rotations if necessary

*int* balance = getBalanceFactor(*root*);

    // Left Left Case

    if (balance > 1 && getBalanceFactor(*root*->left) >= 0)

        return rightRotate(*root*);

    // Left Right Case

    if (balance > 1 && getBalanceFactor(*root*->left) < 0) {

*root*->left = leftRotate(*root*->left);

        return rightRotate(*root*);

    }

    // Right Right Case

    if (balance < -1 && getBalanceFactor(*root*->right) <= 0)

        return leftRotate(*root*);

    // Right Left Case

    if (balance < -1 && getBalanceFactor(*root*->right) > 0) {

*root*->right = rightRotate(*root*->right);

        return leftRotate(*root*);

    }

    return *root*;

}

// Function to free memory allocated for AVL tree nodes

*void* deleteAVLTree(AVLNode\* *root*) {

    if (*root* != nullptr) {

        deleteAVLTree(*root*->left);

        deleteAVLTree(*root*->right);

        delete *root*;

    }

}

*int* main() {

    AVLNode\* avlRoot = nullptr;

    // Read student information from text file and build initial AVL tree

    ifstream inputFile("student\_data.txt");

*int* rollNumber;

    string name;

*float* cgpa;

    while (inputFile >> rollNumber >> name >> cgpa) {

        StudentRecord newStudent;

        newStudent.rollNumber = rollNumber;

        newStudent.name = name;

        newStudent.cgpa = cgpa;

        avlRoot = insertAVL(avlRoot, newStudent);

    }

    inputFile.close();

    // Perform operations on AVL tree

    // For example, to insert a new student:

    StudentRecord newStudent;

    newStudent.rollNumber = 102015200;

    newStudent.name = "John Doe";

    newStudent.cgpa = 7.8;

    avlRoot = insertAVL(avlRoot, newStudent);

    // Display in-order traversal of AVL tree

    cout << "In-Order Traversal of AVL Tree:" << endl;

    inOrderTraversal(avlRoot);

    // Perform delete operation

*int* rollToDelete = 102015070;

    avlRoot = deleteNodeAVL(avlRoot, rollToDelete);

    // Display in-order traversal after deletion

    cout << "\nIn-Order Traversal after Deletion:" << endl;

    inOrderTraversal(avlRoot);

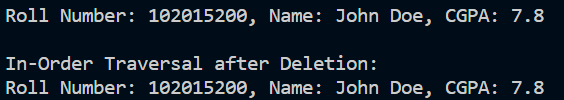
    // Free memory allocated for AVL tree nodes

    deleteAVLTree(avlRoot);

    return 0;

}

Output:



1. Red Black Tree Implementation:

Q1.

#include <iostream>

using *namespace* std;

*class* RBTree {

*private:*

*enum* Color { RED, BLACK };

*struct* Node {

*int* data;

        Node\* parent;

        Node\* left;

        Node\* right;

        Color color;

    };

    Node\* root;

    Node\* rotateLeft(Node\* *x*) {

        Node\* y = *x*->right;

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

        if (y->left != nullptr)

            y->left->parent = *x*;

        y->parent = *x*->parent;

        if (*x*->parent == nullptr)

            root = y;

        else if (*x* == *x*->parent->left)

*x*->parent->left = y;

        else

*x*->parent->right = y;

        y->left = *x*;

*x*->parent = y;

        return root;

    }

    Node\* rotateRight(Node\* *y*) {

        Node\* x = *y*->left;

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

        if (x->right != nullptr)

            x->right->parent = *y*;

        x->parent = *y*->parent;

        if (*y*->parent == nullptr)

            root = x;

        else if (*y* == *y*->parent->left)

*y*->parent->left = x;

        else

*y*->parent->right = x;

        x->right = *y*;

*y*->parent = x;

        return root;

    }

*void* insertFixup(Node\* *z*) {

        while (*z* != nullptr && *z*->parent != nullptr && *z*->parent->color == RED) {

            if (*z*->parent == *z*->parent->parent->left) {

                Node\* y = *z*->parent->parent->right;

                if (y != nullptr && y->color == RED) {

*z*->parent->color = BLACK;

                    y->color = BLACK;

*z*->parent->parent->color = RED;

*z* = *z*->parent->parent;

                } else {

                    if (*z* == *z*->parent->right) {

*z* = *z*->parent;

                        root = rotateLeft(*z*);

                    }

*z*->parent->color = BLACK;

*z*->parent->parent->color = RED;

                    root = rotateRight(*z*->parent->parent);

                }

            } else {

                Node\* y = *z*->parent->parent->left;

                if (y != nullptr && y->color == RED) {

*z*->parent->color = BLACK;

                    y->color = BLACK;

*z*->parent->parent->color = RED;

*z* = *z*->parent->parent;

                } else {

                    if (*z* == *z*->parent->left) {

*z* = *z*->parent;

                        root = rotateRight(*z*);

                    }

*z*->parent->color = BLACK;

*z*->parent->parent->color = RED;

                    root = rotateLeft(*z*->parent->parent);

                }

            }

        }

        if (*z* != nullptr)

*z*->color = BLACK;

    }

    Node\* insert(Node\* *root*, *int* *data*) {

        Node\* z = new Node;

        z->data = *data*;

        z->left = z->right = nullptr;

        z->color = RED;

        Node\* y = nullptr;

        Node\* x = *root*;

        while (x != nullptr) {

            y = x;

            if (z->data < x->data)

                x = x->left;

            else if (z->data > x->data)

                x = x->right;

            else {

                delete z;

                return *root*;

            }

        }

        z->parent = y;

        if (y == nullptr)

*root* = z;

        else if (z->data < y->data)

            y->left = z;

        else

            y->right = z;

        insertFixup(z);

        return *root*;

    }

    Node\* minValueNode(Node\* *node*) {

        Node\* current = *node*;

        while (current->left != nullptr)

            current = current->left;

        return current;

    }

    Node\* deleteNode(Node\* *root*, *int* *data*) {

        Node\* z = nullptr;

        Node\* x, \* y;

        z = *root*;

        while (z != nullptr) {

            if (*data* < z->data)

                z = z->left;

            else if (*data* > z->data)

                z = z->right;

            else

                break;

        }

        if (z == nullptr)

            return *root*;

        Color originalColor = z->color;

        if (z->left == nullptr) {

            x = z->right;

*root* = transplant(*root*, z, z->right);

        } else if (z->right == nullptr) {

            x = z->left;

*root* = transplant(*root*, z, z->left);

        } else {

            y = minValueNode(z->right);

            originalColor = y->color;

            x = y->right;

            if (y->parent == z)

                x->parent = y;

            else {

*root* = transplant(*root*, y, y->right);

                y->right = z->right;

                y->right->parent = y;

            }

*root* = transplant(*root*, z, y);

            y->left = z->left;

            y->left->parent = y;

            y->color = z->color;

        }

        delete z;

        if (originalColor == BLACK)

            deleteFixup(*root*, x);

        return *root*;

    }

    Node\* transplant(Node\* *root*, Node\* *u*, Node\* *v*) {

        if (*u*->parent == nullptr)

*root* = *v*;

        else if (*u* == *u*->parent->left)

*u*->parent->left = *v*;

        else

*u*->parent->right = *v*;

        if (*v* != nullptr)

*v*->parent = *u*->parent;

        return *root*;

    }

*void* deleteFixup(Node\*& *root*, Node\*& *x*) {

        while (*x* != *root* && (*x* == nullptr || *x*->color == BLACK)) {

            if (*x* == *x*->parent->left) {

                Node\* w = *x*->parent->right;

                if (w->color == RED) {

                    w->color = BLACK;

*x*->parent->color = RED;

*root* = rotateLeft(*x*->parent);

                    w = *x*->parent->right;

                }

                if ((w->left == nullptr || w->left->color == BLACK) &&

                    (w->right == nullptr || w->right->color == BLACK)) {

                    w->color = RED;

*x* = *x*->parent;

                } else {

                    if (w->right == nullptr || w->right->color == BLACK) {

                        if (w->left != nullptr)

                            w->left->color = BLACK;

                        w->color = RED;

*root* = rotateRight(w);

                        w = *x*->parent->right;

                    }

                    w->color = *x*->parent->color;

*x*->parent->color = BLACK;

                    if (w->right != nullptr)

                        w->right->color = BLACK;

*root* = rotateLeft(*x*->parent);

*x* = *root*;

                }

            } else {

                Node\* w = *x*->parent->left;

                if (w->color == RED) {

                    w->color = BLACK;

*x*->parent->color = RED;

*root* = rotateRight(*x*->parent);

                    w = *x*->parent->left;

                }

                if ((w->right == nullptr || w->right->color == BLACK) &&

                    (w->left == nullptr || w->left->color == BLACK)) {

                    w->color = RED;

*x* = *x*->parent;

                } else {

                    if (w->left == nullptr || w->left->color == BLACK) {

                        if (w->right != nullptr)

                            w->right->color = BLACK;

                        w->color = RED;

*root* = rotateLeft(w);

                        w = *x*->parent->left;

                    }

                    w->color = *x*->parent->color;

*x*->parent->color = BLACK;

                    if (w->left != nullptr)

                        w->left->color = BLACK;

*root* = rotateRight(*x*->parent);

*x* = *root*;

                }

            }

        }

        if (*x* != nullptr)

*x*->color = BLACK;

    }

*void* inOrderTraversal(Node\* *root*) {

        if (*root* != nullptr) {

            inOrderTraversal(*root*->left);

            std::cout << *root*->data << " ";

            inOrderTraversal(*root*->right);

        }

    }

*void* preOrderTraversal(Node\* *root*) {

        if (*root* != nullptr) {

            std::cout << *root*->data << " ";

            preOrderTraversal(*root*->left);

            preOrderTraversal(*root*->right);

        }

    }

*void* postOrderTraversal(Node\* *root*) {

        if (*root* != nullptr) {

            postOrderTraversal(*root*->left);

            postOrderTraversal(*root*->right);

            std::cout << *root*->data << " ";

        }

    }

*public:*

    RBTree() {

        root = nullptr;

    }

*void* insert(*int* *data*) {

        root = insert(root, *data*);

        if (root != nullptr)

            root->color = BLACK;

    }

*void* remove(*int* *data*) {

        root = deleteNode(root, *data*);

        if (root != nullptr)

            root->color = BLACK;

    }

*void* traverseInOrder() {

        inOrderTraversal(root);

        std::cout << std::endl;

    }

*void* traversePreOrder() {

        preOrderTraversal(root);

        std::cout << std::endl;

    }

*void* traversePostOrder() {

        postOrderTraversal(root);

        std::cout << std::endl;

    }

};

*int* main() {

    RBTree rb;

    rb.insert(10);

    rb.insert(20);

    rb.insert(30);

    rb.insert(40);

    rb.insert(50);

    rb.insert(25);

    std::cout << "In-order traversal: ";

    rb.traverseInOrder();

    std::cout << "Pre-order traversal: ";

    rb.traversePreOrder();

    std::cout << "Post-order traversal: ";

    rb.traversePostOrder();

    rb.remove(30);

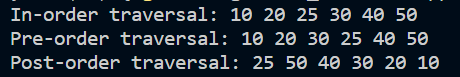
    std::cout << "In-order traversal after deleting 30: ";

    rb.traverseInOrder();

    return 0;

}

Output:



Q2.

// Red-Black Tree implementation can be quite extensive.

// Here is a simplified example for the given requirements:

#include <iostream>

#include <fstream>

#include <string>

using *namespace* std;

*enum* Color { RED, BLACK };

// Define structure for student record

*struct* StudentRecord {

*int* rollNumber;

    string name;

*float* cgpa;

};

// Define structure for Red-Black tree node

*struct* RBNode {

    StudentRecord data;

    RBNode\* parent;

    RBNode\* left;

    RBNode\* right;

    Color color;

};

// Function to perform left rotation

RBNode\* leftRotateRB(RBNode\* *root*, RBNode\* *x*) {

    RBNode\* y = *x*->right;

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

    if (y->left != nullptr)

        y->left->parent = *x*;

    y->parent = *x*->parent;

    if (*x*->parent == nullptr)

*root* = y;

    else if (*x* == *x*->parent->left)

*x*->parent->left = y;

    else

*x*->parent->right = y;

    y->left = *x*;

*x*->parent = y;

    return *root*;

}

// Function to perform right rotation

RBNode\* rightRotateRB(RBNode\* *root*, RBNode\* *y*) {

    RBNode\* x = *y*->left;

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

    if (x->right != nullptr)

        x->right->parent = *y*;

    x->parent = *y*->parent;

    if (*y*->parent == nullptr)

*root* = x;

    else if (*y* == *y*->parent->left)

*y*->parent->left = x;

    else

*y*->parent->right = x;

    x->right = *y*;

*y*->parent = x;

    return *root*;

}

// Function to fix the Red-Black tree after insertion

RBNode\* insertFixupRB(RBNode\* *root*, RBNode\* *z*) {

    while (*z* != nullptr && *z*->parent != nullptr && *z*->parent->color == RED) {

        if (*z*->parent == *z*->parent->parent->left) {

            RBNode\* y = *z*->parent->parent->right;

            if (y != nullptr && y->color == RED) {

*z*->parent->color = BLACK;

                y->color = BLACK;

*z*->parent->parent->color = RED;

*z* = *z*->parent->parent;

            } else {

                if (*z* == *z*->parent->right) {

*z* = *z*->parent;

*root* = leftRotateRB(*root*, *z*);

                }

*z*->parent->color = BLACK;

*z*->parent->parent->color = RED;

*root* = rightRotateRB(*root*, *z*->parent->parent);

            }

        } else {

            RBNode\* y = *z*->parent->parent->left;

            if (y != nullptr && y->color == RED) {

*z*->parent->color = BLACK;

                y->color = BLACK;

*z*->parent->parent->color = RED;

*z* = *z*->parent->parent;

            } else {

                if (*z* == *z*->parent->left) {

*z* = *z*->parent;

*root* = rightRotateRB(*root*, *z*);

                }

*z*->parent->color = BLACK;

*z*->parent->parent->color = RED;

*root* = leftRotateRB(*root*, *z*->parent->parent);

            }

        }

    }

*root*->color = BLACK;

    return *root*;

}

// Function to insert a new student record into Red-Black tree

RBNode\* insertRB(RBNode\* *root*, StudentRecord *newStudent*) {

    RBNode\* z = new RBNode;

    z->data = *newStudent*;

    z->left = z->right = z->parent = nullptr;

    z->color = RED;

    RBNode\* y = nullptr;

    RBNode\* x = *root*;

    while (x != nullptr) {

        y = x;

        if (z->data.rollNumber < x->data.rollNumber)

            x = x->left;

        else if (z->data.rollNumber > x->data.rollNumber)

            x = x->right;

        else {

            // Duplicate roll numbers are not allowed

            delete z;

            return *root*;

        }

    }

    z->parent = y;

    if (y == nullptr)

*root* = z;

    else if (z->data.rollNumber < y->data.rollNumber)

        y->left = z;

    else

        y->right = z;

    return insertFixupRB(*root*, z);

}

// Function to perform in-order traversal of Red-Black tree

*void* inOrderTraversalRB(RBNode\* *root*) {

    if (*root* != nullptr) {

        inOrderTraversalRB(*root*->left);

        cout << "Roll Number: " << *root*->data.rollNumber << ", Name: " << *root*->data.name << ", CGPA: " << *root*->data.cgpa << endl;

        inOrderTraversalRB(*root*->right);

    }

}

// Function to free memory allocated for Red-Black tree nodes

*void* deleteRBTree(RBNode\* *root*) {

    if (*root* != nullptr) {

        deleteRBTree(*root*->left);

        deleteRBTree(*root*->right);

        delete *root*;

    }

}

*int* main() {

    RBNode\* rbRoot = nullptr;

    // Read student information from text file and build initial Red-Black tree

    ifstream inputFile("student\_data.txt");

*int* rollNumber;

    string name;

*float* cgpa;

    while (inputFile >> rollNumber >> name >> cgpa) {

        StudentRecord newStudent;

        newStudent.rollNumber = rollNumber;

        newStudent.name = name;

        newStudent.cgpa = cgpa;

        rbRoot = insertRB(rbRoot, newStudent);

    }

    inputFile.close();

    // Perform operations on Red-Black tree

    // For example, to insert a new student:

    StudentRecord newStudent;

    newStudent.rollNumber = 102015200;

    newStudent.name = "John Doe";

    newStudent.cgpa = 7.8;

    rbRoot = insertRB(rbRoot, newStudent);

    // Display in-order traversal of Red-Black tree

    cout << "In-Order Traversal of Red-Black Tree:" << endl;

    inOrderTraversalRB(rbRoot);

    // Free memory allocated for Red-Black tree nodes

    deleteRBTree(rbRoot);

    return 0;

}

Output:

