ASSIGNMENT 2

AIM:
Implement a multiplayer game system that uses an AVL tree data structure to organize and manage player data efficiently. The multiplayer game supports multiple players participating simultaneously, and the AVL tree is used to store player information such as player_id and scores (key, value pair). The system should provide following operation:
1. Player Registration
2. Leaderboard Display
3. Remove player from game

SOURCE CODE:

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

class Player {
public:
    int player_id;
    int score;
    Player* left;
    Player* right;
    int height;
```

```
    Player(int id, int sc){
        player_id = id;
        score = sc;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

class AVLTree {
private:
    Player* root;

    int height(Player* node) {
        return node ? node->height : 0;
    }

    int getBalance(Player* node) {
        return node ? height(node->left) - height(node->right) : 0;
    }

    Player* rightRotate(Player* y) {
        Player* x = y->left;
        Player* T2 = x->right;

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

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

        return x;
    }

    Player* leftRotate(Player* x) {
        Player* y = x->right;
        Player* T2 = y->left;

        y->left = x;
```

```
    x->right = T2;

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

    return y;
}

Player* insert(Player* node, int player_id, int score) {
    if (!node) return new Player(player_id, score);

    if (player_id < node->player_id)
        node->left = insert(node->left, player_id, score);
    else if (player_id > node->player_id)
        node->right = insert(node->right, player_id, score);
    else
        return node; // Duplicate ID not allowed

    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);

    // Balancing cases
    if (balance > 1 && player_id < node->left->player_id)
        return rightRotate(node);
    if (balance < -1 && player_id > node->right->player_id)
        return leftRotate(node);
    if (balance > 1 && player_id > node->left->player_id) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && player_id < node->right->player_id) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

Player* minValueNode(Player* node) {
    Player* current = node;
```

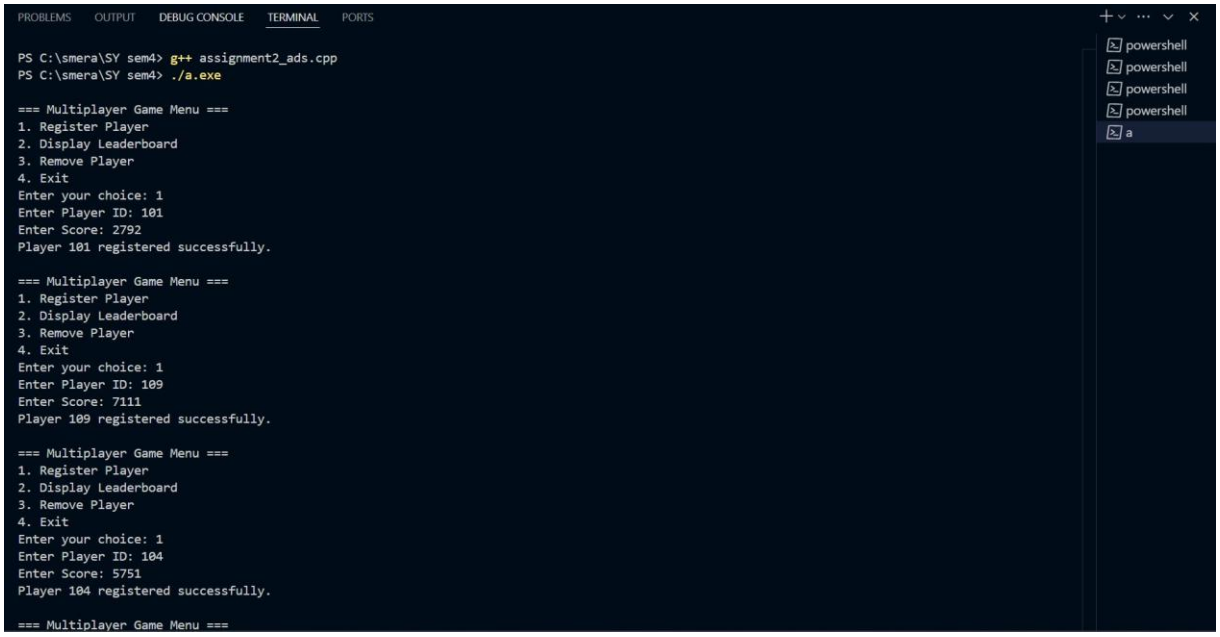
```
    while (current && current->left)
        current = current->left;
    return current;
}

Player* remove(Player* root, int player_id) {
    if (!root) return root;

    if (player_id < root->player_id)
        root->left = remove(root->left, player_id);
    else if (player_id > root->player_id)
        root->right = remove(root->right, player_id);
    else {
        if (!root->left || !root->right) {
            Player* temp = root->left ? root->left : root->right;
            if (!temp) {
                temp = root;
                root = nullptr;
            } else {
                *root = *temp;
            }
            delete temp;
        } else {
            Player* temp = minValueNode(root->right);
            root->player_id = temp->player_id;
            root->score = temp->score;
            root->right = remove(root->right, temp->player_id);
        }
    }

    if (!root) return root;

    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
```

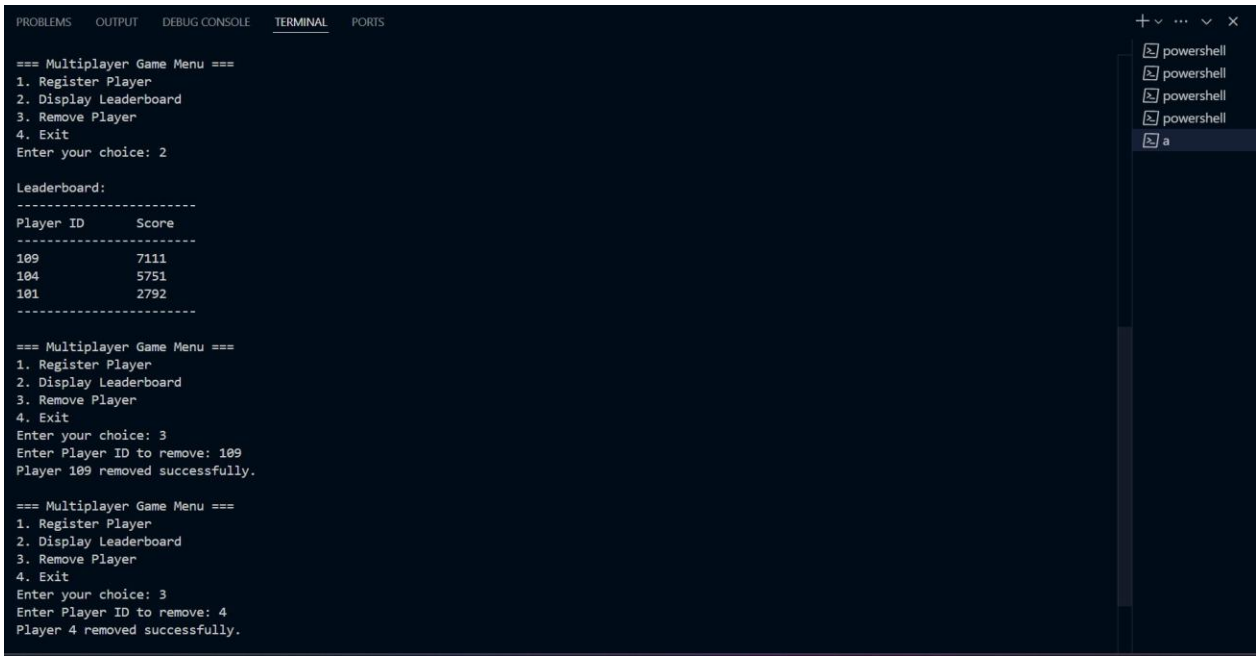
```
        }
        if (balance < -1 && getBalance(root->right) <= 0)
            return leftRotate(root);
        if (balance < -1 && getBalance(root->right) > 0) {
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }

        return root;
    }

    void inOrderTraversal(Player* node, vector<Player*>& players) {
        if (!node) return;
        inOrderTraversal(node->left, players);
        players.push_back(node);
        inOrderTraversal(node->right, players);
    }

    static bool comparePlayers(Player* a, Player* b) {
        if (a->score != b->score)
            return a->score > b->score; // Higher score first
        return a->player_id < b->player_id; // Tie-breaker: lower ID first
    }

public:
    AVLTree() : root(nullptr) {}

    void registerPlayer(int player_id, int score) {
        root = insert(root, player_id, score);
        cout << "Player " << player_id << " registered successfully.\n";
    }

    void removePlayer(int player_id) {
        root = remove(root, player_id);
        cout << "Player " << player_id << " removed successfully.\n";
    }

    void displayLeaderboard() {
        vector<Player*> players;
        inOrderTraversal(root, players);
```

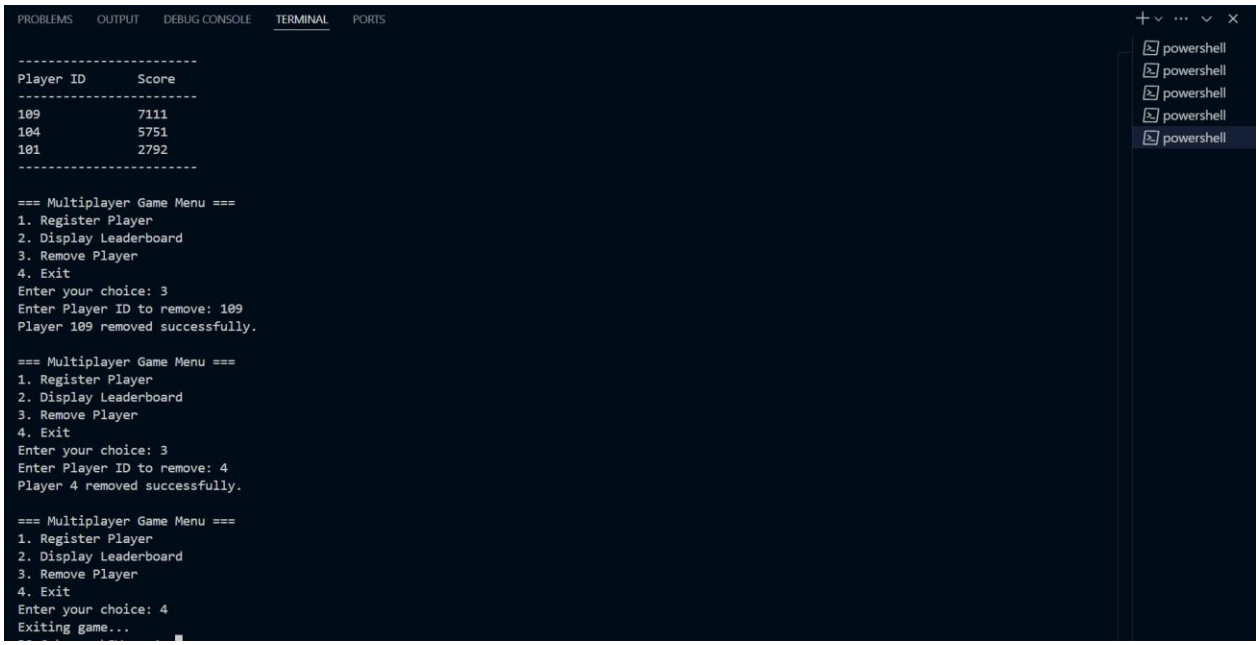
```
        sort(players.begin(), players.end(), comparePlayers);

        cout << "\nLeaderboard:\n";
        cout << "----------------------- \n";
        cout << "Player ID\tScore\n";
        cout << "----------------------- \n";
        for (Player* p : players) {
            cout << p->player_id << "\t\t" << p->score << endl;
        }
        cout << "----------------------- \n";
    }
};

int main() {
    AVLTree game;
    int choice, id, score;

    do {
        cout << "\n=== Multiplayer Game Menu ===\n";
        cout << "1. Register Player\n";
        cout << "2. Display Leaderboard\n";
        cout << "3. Remove Player\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter Player ID: ";
                cin >> id;
                cout << "Enter Score: ";
                cin >> score;
                game.registerPlayer(id, score);
                break;

            case 2:
                game.displayLeaderboard();
                break;

            case 3:
```

```
            cout << "Enter Player ID to remove: ";
            cin >> id;
            game.removePlayer(id);
            break;

        case 4:
            cout << "Exiting game...\n";
            break;

        default:
            cout << "Invalid choice. Please try again.\n";
      }

   } while (choice != 4);

   return 0;
}
```

OUTPUT:







CONCLUSION:

The implementation of a multiplayer game system using an AVL tree effectively ensures balanced and efficient management of player data. By storing player information as (player_id, score) pairs, the AVL tree maintains its height-balanced property, enabling quick operations such as insertion, deletion, and traversal in O(log n) time. This guarantees seamless **player** registration, dynamic leaderboard updates, and swift player removal, which are critical in real-time multiplayer environments. The use of AVL trees enhances the system's responsiveness

and scalability, making it well-suited for competitive gaming platforms where performance and real-time data handling are essential. Overall, this approach offers a robust and optimized solution for managing player data in multiplayer game applications.