```
/* Platform Crowd Management Code*/
#include <iostream>
#include <queue>
#include <vector>
#include <cmath>
#include <climits>

using namespace std;

// Platform node for AVL Tree and Priority Queue
struct Platform {
    int platform_id;
    int crowd_density;
    vector<int> train_ids; // List of train IDs

    Platform(int id, int density, vector<int> trains)
        : platform_id(id), crowd_density(density), train_ids(trains) {}

    // Define comparison operator for max-heap (priority queue)
    bool operator<(const Platform& other) const {
        return crowd_density < other.crowd_density; // Max-heap by crowd density
    }
};

// AVL Tree Node
struct AVLNode {
    Platform platform;
    AVLNode* left;
```

```
    AVLNode* right;
    int height;

    AVLNode(Platform p) : platform(p), left(nullptr), right(nullptr), height(1) {}
};

// AVL Tree Class
class AVLTree {
private:
    AVLNode* root;

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

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

    AVLNode* rightRotate(AVLNode* node) {
        AVLNode* new_root = node->left;
        node->left = new_root->right;
        new_root->right = node;
        node->height = max(height(node->left), height(node->right)) + 1;
        new_root->height = max(height(new_root->left), height(new_root->right)) + 1;
        return new_root;
    }
```

```
AVLNode* leftRotate(AVLNode* node) {
    AVLNode* new_root = node->right;
    node->right = new_root->left;
    new_root->left = node;
    node->height = max(height(node->left), height(node->right)) + 1;
    new_root->height = max(height(new_root->left), height(new_root->right)) + 1;
    return new_root;
}

AVLNode* insert(AVLNode* node, Platform p) {
    if (node == nullptr) return new AVLNode(p);

    if (p.crowd_density < node->platform.crowd_density)
        node->left = insert(node->left, p);
    else if (p.crowd_density > node->platform.crowd_density)
        node->right = insert(node->right, p);
    else return node;

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

    int balance = getBalance(node);

    if (balance > 1 && p.crowd_density < node->left->platform.crowd_density)
        return rightRotate(node);
    if (balance < -1 && p.crowd_density > node->right->platform.crowd_density)
        return leftRotate(node);
    if (balance > 1 && p.crowd_density > node->left->platform.crowd_density) {
        node->left = leftRotate(node->left);
```

```
            return rightRotate(node);
        }
        if (balance < -1 && p.crowd_density < node->right->platform.crowd_density) {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }

        return node;
    }

    void inOrder(AVLNode* root) {
        if (root != nullptr) {
            inOrder(root->left);
            cout << "Platform " << root->platform.platform_id
                 << " with crowd density " << root->platform.crowd_density
                 << " and trains: ";
            for (int train_id : root->platform.train_ids) {
                cout << train_id << " ";
            }
            cout << endl;
            inOrder(root->right);
        }
    }

    void detectOvercrowdedPlatformsHelper(AVLNode* root, priority_queue<Platform>& pq)
{
        if (root != nullptr) {
            detectOvercrowdedPlatformsHelper(root->left, pq);
```

```
            if (root->platform.crowd_density >= 500) {
                pq.push(root->platform); // Add overcrowded platform to the priority queue
            }
            detectOvercrowdedPlatformsHelper(root->right, pq);
        }
    }

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

    void insertPlatform(Platform p) {
        root = insert(root, p);
    }

    void displayPlatforms() {
        inOrder(root);
    }

    void detectOvercrowdedPlatforms(priority_queue<Platform>& pq) {
        detectOvercrowdedPlatformsHelper(root, pq);
    }
};

// Main program
int main() {
    AVLTree tree;
    priority_queue<Platform> pq;
    int n;
```

```
cout << "Enter the number of platforms: ";
cin >> n;

for (int i = 0; i < n; ++i) {
    int platform_id, crowd_density, num_trains;
    cout<<"--------------------------------------------------";
    cout << "\nEnter platform ID and crowd density for platform " << i + 1 << ": ";
    cin >> platform_id >> crowd_density;

    cout << "Enter the number of trains on this platform: ";
    cin >> num_trains;

    vector<int> train_ids(num_trains);
    cout << "Enter train IDs for platform" << platform_id << ":\n";
    for (int j = 0; j < num_trains; ++j) {
        cin >> train_ids[j];
    }

    tree.insertPlatform(Platform(platform_id, crowd_density, train_ids));
}

cout << "\nPlatforms in AVL Tree (sorted by crowd density):" << endl;
tree.displayPlatforms();

cout << "\nDetecting overcrowded platforms and adding to priority queue:" << endl;
tree.detectOvercrowdedPlatforms(pq);
```

```
    cout << "Overcrowded platforms detected:" << endl;
    while (!pq.empty()) {
        Platform p = pq.top();
        pq.pop();
        cout << "Platform " << p.platform_id << " with crowd density " << p.crowd_density
             << " and trains : ";
        for (int train_id : p.train_ids) {
            cout << train_id << " ";
        }
        cout << " will be RESCHEDULED";
        cout << endl;
    }
    if (pq.empty()){
        cout << "No platforms are overcrowded\n";
    }

    return 0;
}
```