AVL Tree Deletion Code

```
#include <iostream>
using namespace std;

class Node {
public:
    int data, height;
    Node *left, *right;

    Node(int value) {
        data = value;
        height = 1;
        left = right = nullptr;
    }
};

// Utility to get height
int getHeight(Node *root) {
    if (!root) return 0;
    return root->height;
}

// Utility to get balance factor
int getBalance(Node *root) {
    if (!root) return 0;
    return getHeight(root->left) - getHeight(root->right);
}

// Left Rotation
Node* leftRotation(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;

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

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

```
    return y;
}

// Right Rotation
Node* rightRotation(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;

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

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

    return x;
}

// Find node with minimum value
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left)
        current = current->left;
    return current;
}

// AVL Delete
Node* deleteNode(Node* root, int key) {
    if (!root)
        return root;

    // Step 1: BST delete
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node with 0 or 1 child
        if (!root->left || !root->right) {
            Node* temp = root->left ? root->left : root->right;
```

```
        if (!temp) {
            temp = root;
            root = nullptr;
        } else
            *root = *temp;
        delete temp;
    } else {
        // Node with 2 children
        Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
}

// If tree had 1 node
if (!root)
    return root;

// Step 2: Update height
root->height = 1 + max(getHeight(root->left), getHeight(root->right));

// Step 3: Get balance
int balance = getBalance(root);

// Step 4: Rebalance cases

// LL
if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotation(root);

// LR
if (balance > 1 && getBalance(root->left) < 0) {
    root->left = leftRotation(root->left);
    return rightRotation(root);
}

// RR
if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotation(root);
```

```
    // RL
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotation(root->right);
        return leftRotation(root);
    }

    return root;
}

// Inorder traversal
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

int main() {
    Node *root = nullptr;

    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);
    root = insert(root, 10);
    root = insert(root, 100);
    root = insert(root, 95); // Triggers RL rotation

    cout << "Inorder before deletion: ";
    inorder(root);
    cout << endl;

    root = deleteNode(root, 100); // Delete and check balance
    root = deleteNode(root, 20);  // More deletions

    cout << "Inorder after deletion: ";
    inorder(root);
    cout << endl;
```

```
    return 0;
}
```