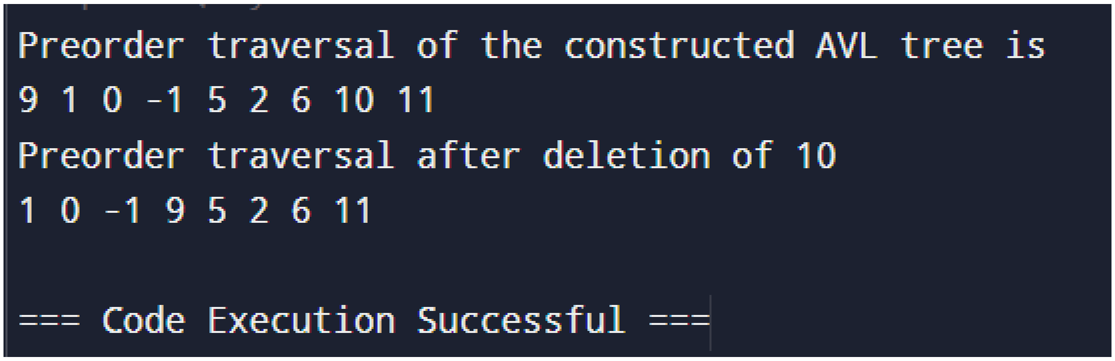
AVL Code Insertion and Deletion

```
#include <iostream>


 using namespace std;

 class Node {
 private:
     int key;
     Node* left;
     Node* right;
     int height;

 public:
     Node(int k) : key(k), left(nullptr), right(nullptr), height(1) {}

     int getKey() const { return key; }
     Node* getLeft() const { return left; }
     Node* getRight() const { return right; }
     int getHeight() const { return height; }

     void setKey(int k) { key = k; }
     void setLeft(Node* node) { left = node; }
     void setRight(Node* node) { right = node; }
     void setHeight(int h) { height = h; }
 };


 int height(Node* n) {
     if (n == nullptr) return 0;
     return n->getHeight();
 }


 int getBalanceFactor(Node* n) {
     if (n == nullptr) return 0;
     return height(n->getLeft()) - height(n->getRight());
 }


 Node* rightRotate(Node* y) {
     Node* x = y->getLeft();
     Node* T2 = x->getRight();

     // Perform rotation
     x->setRight(y);
     y->setLeft(T2);

     // Update heights
     y->setHeight(1 + max(height(y->getLeft()), height(y->getRight())));
     x->setHeight(1 + max(height(x->getLeft()), height(x->getRight())));

     // Return new root
     return x;
 }


 Node* leftRotate(Node* x) {
     Node* y = x->getRight();
     Node* T2 = y->getLeft();
```

```
    // Perform rotation
    y->setLeft(x);
    x->setRight(T2);

    // Update heights
    x->setHeight(1 + max(height(x->getLeft()), height(x->getRight())));
    y->setHeight(1 + max(height(y->getLeft()), height(y->getRight())));

    // Return new root
    return y;
}


Node* insert(Node* node, int key) {
    if (node == nullptr) return new Node(key);

    if (key < node->getKey()) {
        node->setLeft(insert(node->getLeft(), key));
    }
    else if (key > node->getKey()) {
        node->setRight(insert(node->getRight(), key));
    }
    else {
        return node; // Duplicate keys not allowed
    }

    // Update height of current node
    node->setHeight(1 + max(height(node->getLeft()), height(node->getRight())));


    int balance = getBalanceFactor(node);



    // Left Left Case
    if (balance > 1 && key < node->getLeft()->getKey()) {
        return rightRotate(node);
    }

    // Right Right Case
    if (balance < -1 && key > node->getRight()->getKey()) {
         return leftRotate(node);
     }

     // Left Right Case
     if (balance > 1 && key > node->getLeft()->getKey()) {
         node->setLeft(leftRotate(node->getLeft()));
         return rightRotate(node);
     }

     // Right Left Case
     if (balance < -1 && key < node->getRight()->getKey()) {
         node->setRight(rightRotate(node->getRight()));
         return leftRotate(node);
     }

     return node;
 }

 // Utility function to find the node with the minimum key value
 Node* minValueNode(Node* node) {
     Node* current = node;
     while (current->getLeft() != nullptr)
         current = current->getLeft();
     return current;
```

```
}


Node* deleteNode(Node* root, int key) {
    if (root == nullptr)
        return root;

    // Perform standard BST deletion
    if (key < root->getKey()) {
        root->setLeft(deleteNode(root->getLeft(), key));
    }
    else if (key > root->getKey()) {
        root->setRight(deleteNode(root->getRight(), key));
    }
     else {
        if ((root->getLeft() == nullptr) || (root->getRight() == nullptr)) {
            Node* temp = root->getLeft() ? root->getLeft() : root->getRight();
            if (temp == nullptr) {
                temp = root;
                root = nullptr;
            } else {
                *root = *temp;
            }
            delete temp;
        }
     else {
            Node* temp = minValueNode(root->getRight());
            root->setKey(temp->getKey());  // Use the setter function here
            root->setRight(deleteNode(root->getRight(), temp->getKey()));
        }
    }

    if (root == nullptr)
        return root;

    // Update height
    root->setHeight(1 + max(height(root->getLeft()), height(root->getRight())));

    // Get balance factor
    int balance = getBalanceFactor(root);

    // Balancing tree
    if (balance > 1 && getBalanceFactor(root->getLeft()) >= 0)
        return rightRotate(root);
    if (balance > 1 && getBalanceFactor(root->getLeft()) < 0) {
        root->setLeft(leftRotate(root->getLeft()));
        return rightRotate(root);
    }
    if (balance < -1 && getBalanceFactor(root->getRight()) <= 0)
        return leftRotate(root);
    if (balance < -1 && getBalanceFactor(root->getRight()) > 0) {
        root->setRight(rightRotate(root->getRight()));
        return leftRotate(root);
    }

    return root;
}


void preOrder(Node* root) {
    if (root != nullptr) {
        cout << root->getKey() << " ";
        preOrder(root->getLeft());
        preOrder(root->getRight());
    }
```

```
}


int main() {
    Node* root = nullptr;

    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    cout << "Preorder traversal of the "
            "constructed AVL tree is \n";
    preOrder(root);

    root = deleteNode(root, 10);

    cout << "\nPreorder traversal after"
            " deletion of 10 \n";
    preOrder(root);
    return 0;
}
```



```
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11

=== Code Execution Successful ===
```