Code for implementing pre-order, post-order, and in-order traversals for a binary tree:

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

// Definition of a tree node
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Pre-order traversal: Root -> Left -> Right
void preOrder(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->value << " ";
    preOrder(root->left);
    preOrder(root->right);
}

// In-order traversal: Left -> Root -> Right
void inOrder(TreeNode* root) {
    if (root == nullptr) return;
    inOrder(root->left);
```

```
    cout << root->value << " ";
    inOrder(root->right);
}

// Post-order traversal: Left -> Right -> Root
void postOrder(TreeNode* root) {
    if (root == nullptr) return;
    postOrder(root->left);
    postOrder(root->right);
    cout << root->value << " ";
}

int main() {
    // Create a sample tree:
    //         1
    //        / \
    //       2   3
    //      / \
    //     4   5

    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
```

```
    cout << "Pre-order Traversal: ";
    preOrder(root);
    cout << endl;

    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;

    cout << "Post-order Traversal: ";
    postOrder(root);
    cout << endl;

    // Clean up dynamically allocated memory
    delete root->left->left;
    delete root->left->right;
    delete root->left;
    delete root->right;
    delete root;

    return 0;
}
```

2. Create a binary tree

```
#include <iostream>
```

```
using namespace std;



struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

int main() {

    TreeNode* root = new TreeNode(1);



    root->left = new TreeNode(2);



    cout << "Root node value: " << root->value << endl;
    cout << "Left child value: " << root->left->value << endl;



    delete root->left;
    delete root;

    return 0;
```

```
}
```

3. Convert binary tree to binary search tree

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

// Definition of a tree node
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Function to insert a new value into the BST
TreeNode* insert(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val); // Create a new node if root is null
    }
    if (val < root->value) {
        root->left = insert(root->left, val); // Insert into the left subtree
    } else if (val > root->value) {
        root->right = insert(root->right, val); // Insert into the right subtree
    }
    return root; // Return the unchanged root
}
```

```
// In-order traversal to display the BST
void inOrder(TreeNode* root) {
    if (root == nullptr) return;
    inOrder(root->left);
    cout << root->value << " ";
    inOrder(root->right);
}

int main() {
    TreeNode* root = nullptr; // Start with an empty tree

    // Insert nodes into the BST
    root = insert(root, 5);
    root = insert(root, 3);
    root = insert(root, 7);
    root = insert(root, 2);
    root = insert(root, 4);
    root = insert(root, 6);
    root = insert(root, 8);

    // Display the BST using in-order traversal
    cout << "In-order traversal of the BST: ";
    inOrder(root);
    cout << endl;

    // Memory cleanup is omitted here for simplicity but should be handled in production code
```

```
    return 0;
}
```

4. convert the binary search tree into AVL tree

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

// Definition of a tree node
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;
    int height;

    TreeNode(int val) : value(val), left(nullptr), right(nullptr), height(1) {}
};

// Helper function to get the height of a node
int getHeight(TreeNode* node) {
    return node ? node->height : 0;
}

// Calculate the balance factor of a node
int getBalance(TreeNode* node) {
    return node ? getHeight(node->left) - getHeight(node->right) : 0;
}
```

```
// Right rotation
TreeNode* rightRotate(TreeNode* y) {
    TreeNode* x = y->left;
    TreeNode* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

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

    // Return the new root
    return x;
}

// Left rotation
TreeNode* leftRotate(TreeNode* x) {
    TreeNode* y = x->right;
    TreeNode* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
```

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

    // Return the new root
    return y;
}

// Insert a new value into the AVL tree
TreeNode* insert(TreeNode* root, int val) {
    // Perform normal BST insertion
    if (root == nullptr) {
        return new TreeNode(val);
    }
    if (val < root->value) {
        root->left = insert(root->left, val);
    } else if (val > root->value) {
        root->right = insert(root->right, val);
    } else {
        // Duplicate values are not allowed in BST/AVL
        return root;
    }

    // Update the height of this node
    root->height = 1 + max(getHeight(root->left), getHeight(root->right));

    // Get the balance factor
    int balance = getBalance(root);
```

```
    // Perform rotations to balance the tree
    // Left Left Case
    if (balance > 1 && val < root->left->value) {
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && val > root->right->value) {
        return leftRotate(root);
    }

    // Left Right Case
    if (balance > 1 && val > root->left->value) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Left Case
    if (balance < -1 && val < root->right->value) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    // Return the unchanged root
    return root;
}
```

```
// In-order traversal to display the AVL tree
void inOrder(TreeNode* root) {
    if (root == nullptr) return;
    inOrder(root->left);
    cout << root->value << " ";
    inOrder(root->right);
}

int main() {
    TreeNode* root = nullptr; // Start with an empty tree

    // Insert nodes into the AVL tree
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    // Display the AVL tree using in-order traversal
    cout << "In-order traversal of the AVL tree: ";
    inOrder(root);
    cout << endl;

    // Memory cleanup is omitted here for simplicity but should be handled in production code
```

```
    return 0;
}
```