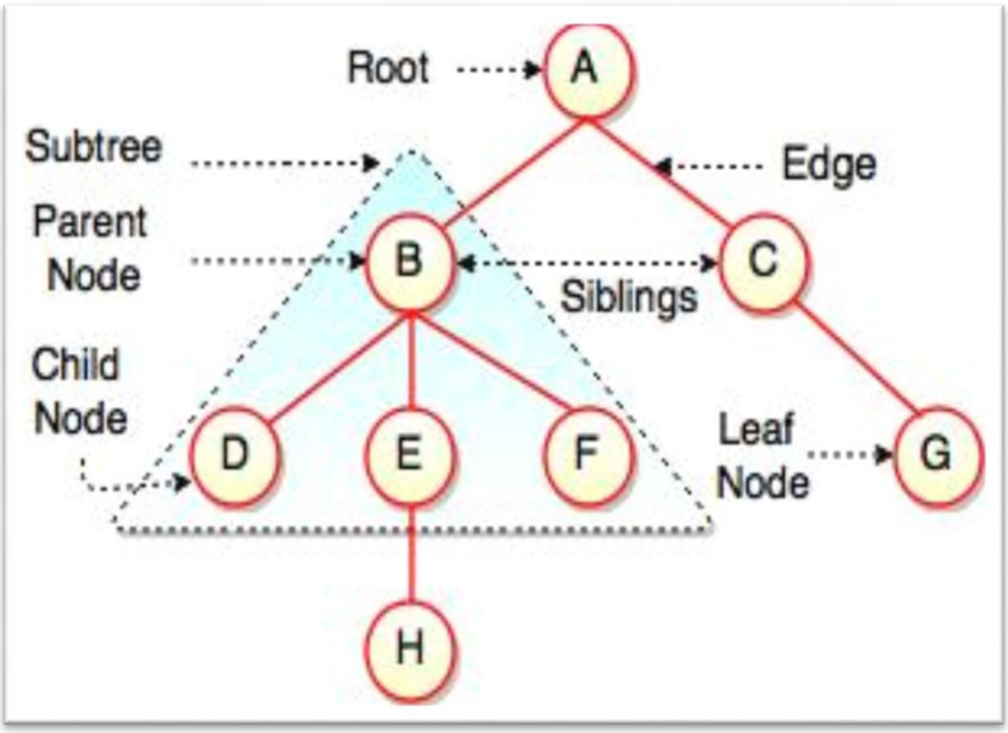
TREE DATA STRUCTURE

What is a tree?

→ Tree is a nonlinear hierarchical data structure consisting of nodes. It allows user quicker access to data thus reducing time complexity.

Basic Terminologies:

- Root- It is the topmost node of a tree.
- Node- It is an entity that contains some data and pointer as a reference to child node they are interconnected by Edge. (A node that doesn't have a child node is called leaf node).
- Depth of the node- It is the total number of edges needed to connect that particular node to the root.
- Height of the tree- It is the depth of the deepest node of the tree.
- Siblings- Nodes having same parents are called siblings.
- Forest- Different disjoint tree constitutes a forest.



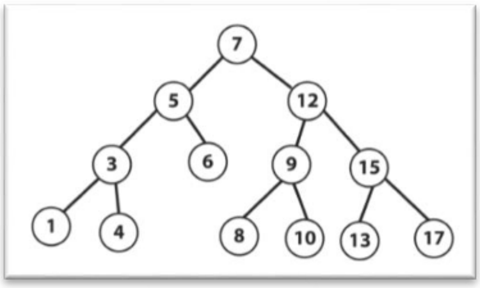
Types of tree data structure:

- Binary Tree – Each node can have utmost 2 child nodes.
- Binary Search Tree
- AVL Tree
- B-Tree

Binary Search Tree:

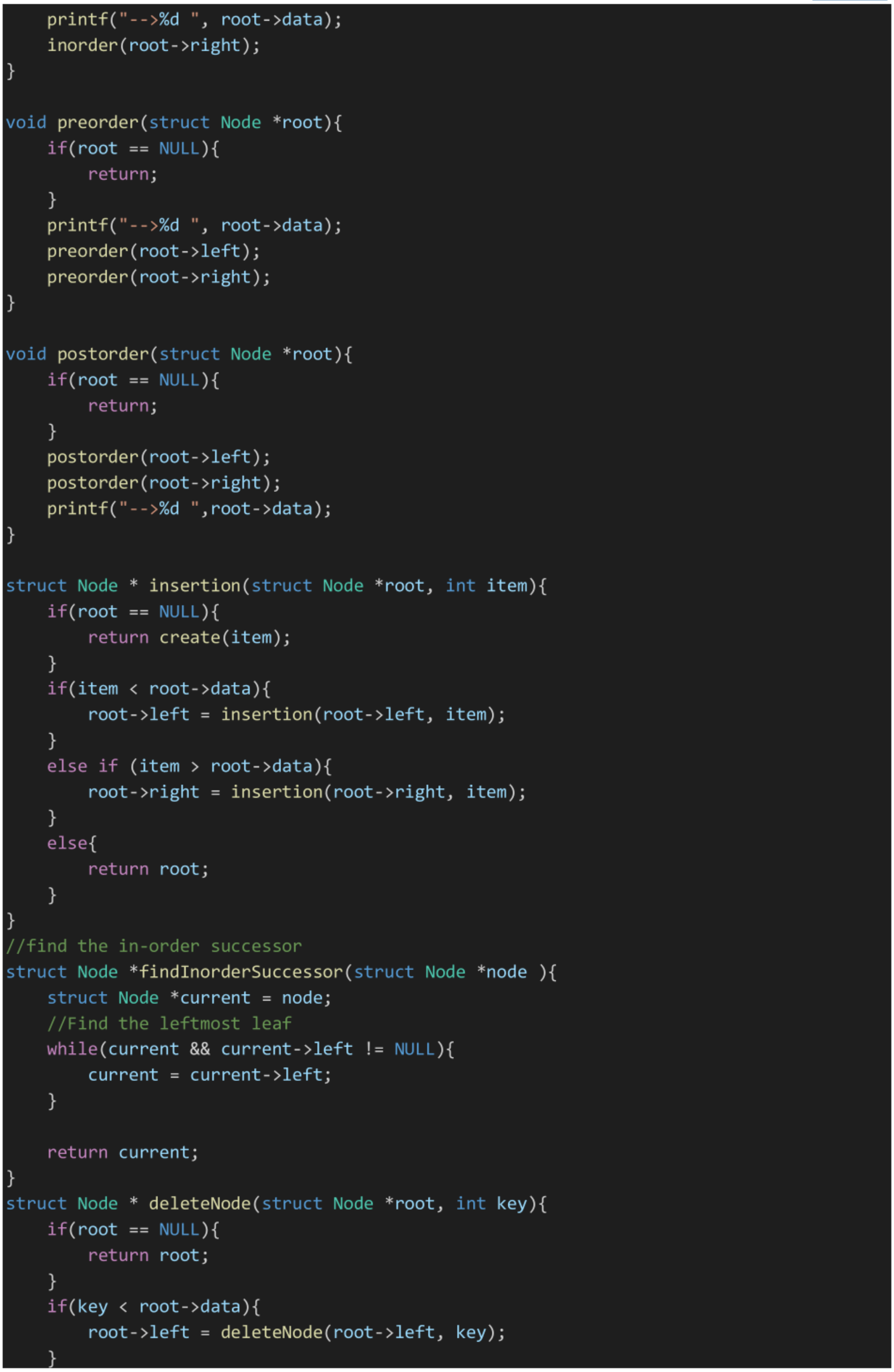
Properties-

- Each node has maximum 2 child node.
- Nodes of left subtree are less than the root node; In case of right subtree it’s just the opposite.

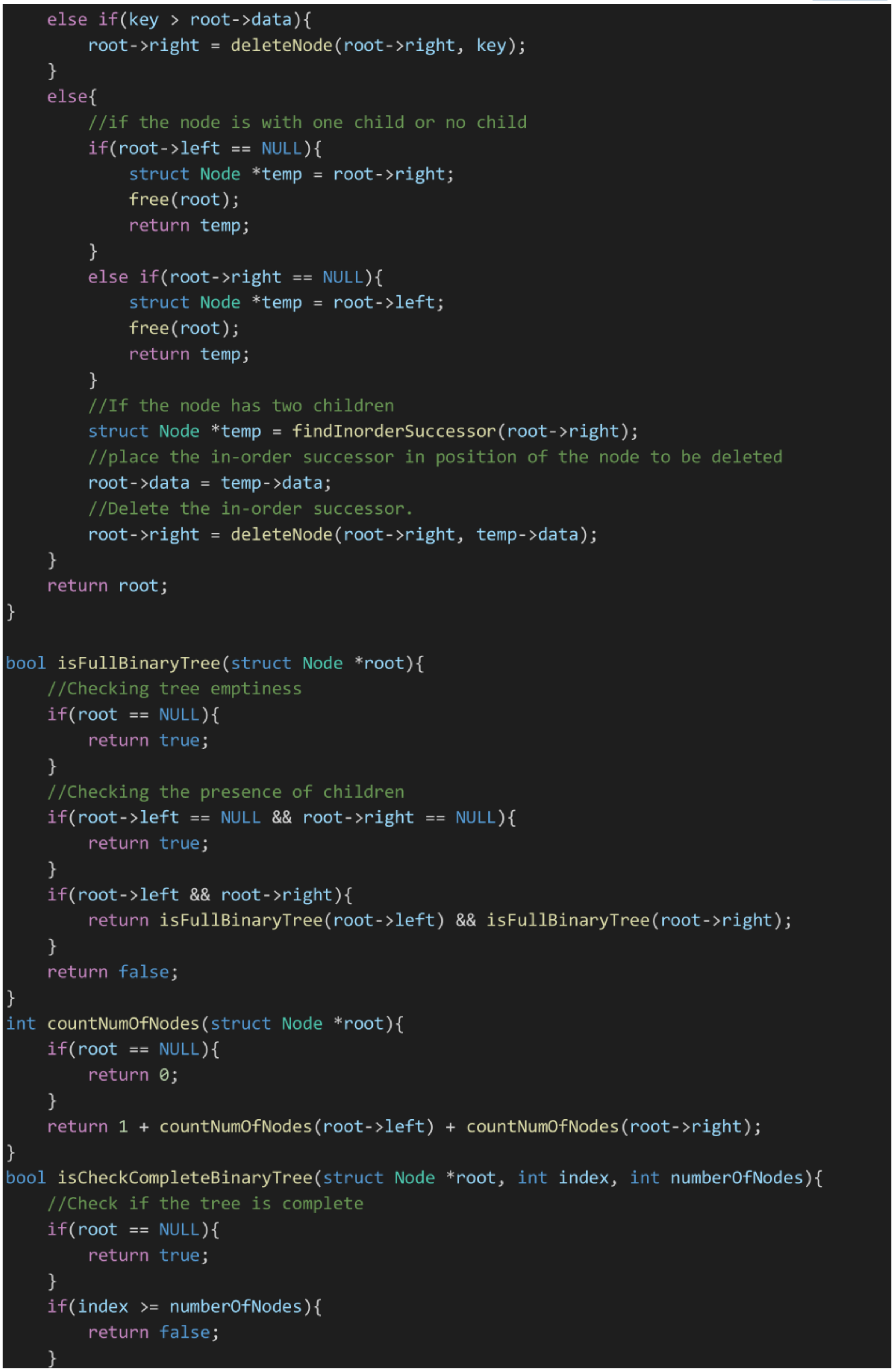


Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

struct Node{
    int data;
    struct Node *left;
    struct Node *right;
};
struct Node * create(int item){
    struct Node * node = (struct Node*) malloc(sizeof(struct Node*));
    node->data = item;
    node->left = node->right = NULL;
    return node;
}

void inorder(struct Node *root){
    if(root == NULL){
        return;
    }
    inorder(root->left);
```

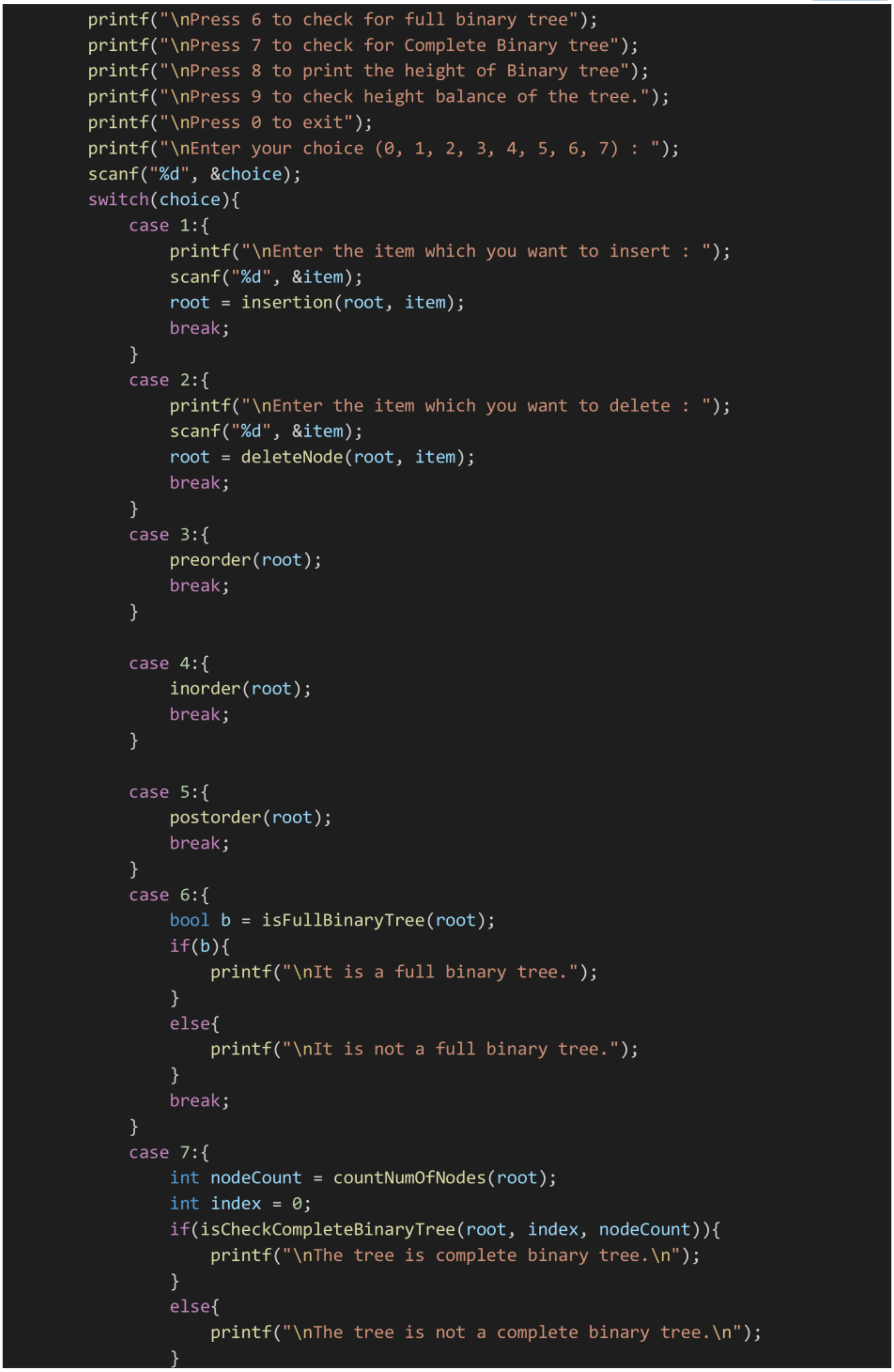


```
    printf("-->%d ", root->data);
    inorder(root->right);
}

void preorder(struct Node *root){
    if(root == NULL){
        return;
    }
    printf("-->%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

void postorder(struct Node *root){
    if(root == NULL){
        return;
    }
    postorder(root->left);
    postorder(root->right);
    printf("-->%d ",root->data);
}

struct Node * insertion(struct Node *root, int item){
    if(root == NULL){
        return create(item);
    }
    if(item < root->data){
        root->left = insertion(root->left, item);
    }
    else if (item > root->data){
        root->right = insertion(root->right, item);
    }
    else{
        return root;
    }
}
//find the in-order successor
struct Node *findInorderSuccessor(struct Node *node ){
    struct Node *current = node;
    //Find the leftmost leaf
    while(current && current->left != NULL){
        current = current->left;
    }

    return current;
}
struct Node * deleteNode(struct Node *root, int key){
    if(root == NULL){
        return root;
    }
    if(key < root->data){
        root->left = deleteNode(root->left, key);
    }
```

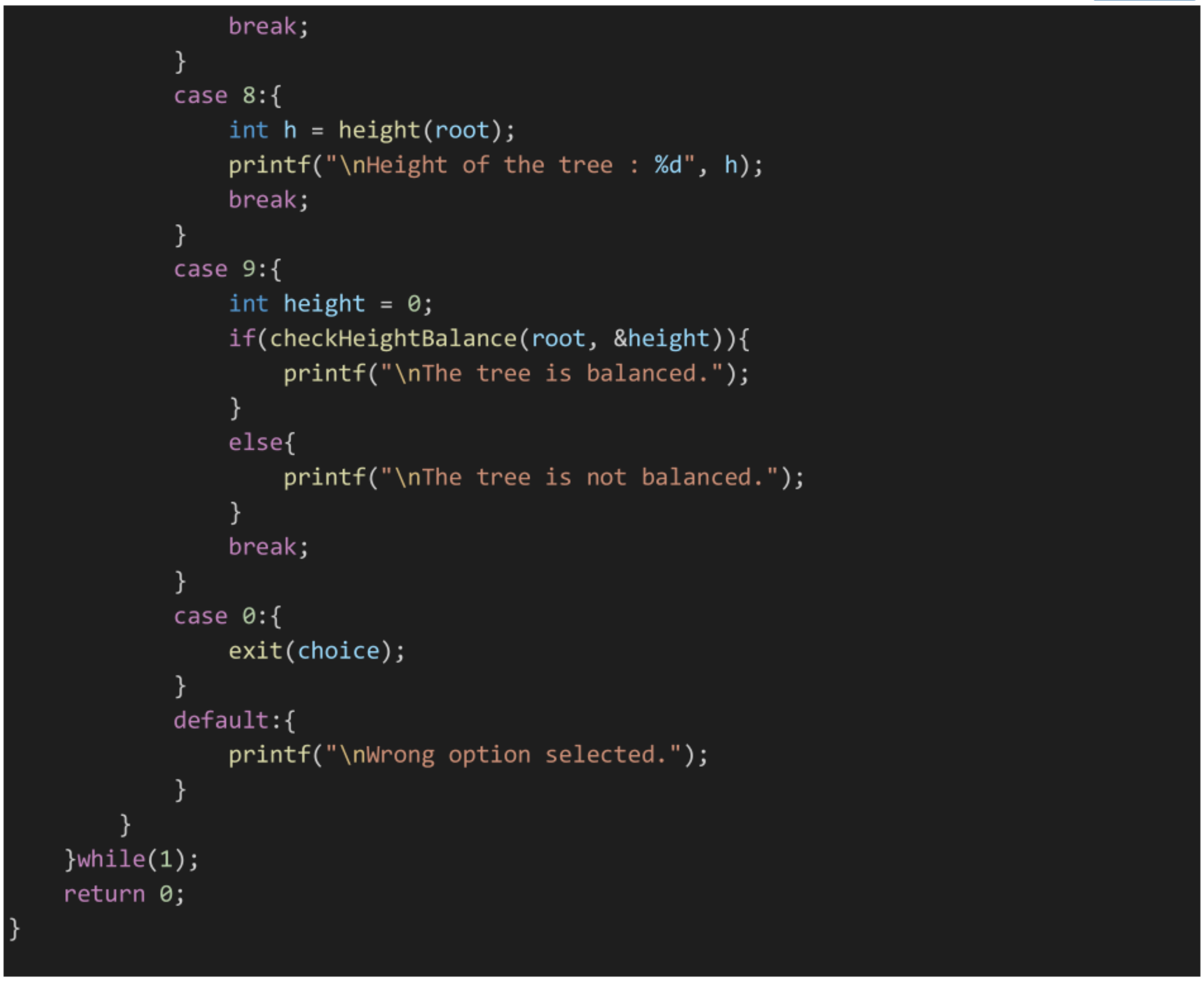


```
    else if(key > root->data){
        root->right = deleteNode(root->right, key);
    }
    else{
        //if the node is with one child or no child
        if(root->left == NULL){
            struct Node *temp = root->right;
            free(root);
            return temp;
        }
        else if(root->right == NULL){
            struct Node *temp = root->left;
            free(root);
            return temp;
        }
        //If the node has two children
        struct Node *temp = findInorderSuccessor(root->right);
        //place the in-order successor in position of the node to be deleted
        root->data = temp->data;
        //Delete the in-order successor.
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

bool isFullBinaryTree(struct Node *root){
    //Checking tree emptiness
    if(root == NULL){
        return true;
    }
    //Checking the presence of children
    if(root->left == NULL && root->right == NULL){
        return true;
    }
    if(root->left && root->right){
        return isFullBinaryTree(root->left) && isFullBinaryTree(root->right);
    }
    return false;
}
int countNumOfNodes(struct Node *root){
    if(root == NULL){
        return 0;
    }
    return 1 + countNumOfNodes(root->left) + countNumOfNodes(root->right);
}
bool isCheckCompleteBinaryTree(struct Node *root, int index, int numberOfNodes){
    //Check if the tree is complete
    if(root == NULL){
        return true;
    }
    if(index >= numberOfNodes){
        return false;
    }
```

```c
    return (isCheckCompleteBinaryTree(root->left, 2*index + 1, numberOfNodes) &&
            isCheckCompleteBinaryTree(root->right, 2*index + 2, numberOfNodes));
}
int height(struct Node* root){
    if(root == NULL)
        return 0;
    else{
        int leftDepth = height(root->left);
        int rightDepth = height(root->right);

        if(leftDepth > rightDepth){
            return leftDepth + 1;
        }
        else{
            return rightDepth + 1;
        }
    }
}
//Check for height balance
bool checkHeightBalance(struct Node *root, int *height){
    //Check for emptiness
    int leftHeight = 0, rightHeight = 0;
    int L = 0, R = 0;

    if(root == NULL){
        *height = 0;
        return 1;
    }

    L = checkHeightBalance(root->left, &leftHeight);
    R = checkHeightBalance(root->right, &rightHeight);

    *height = ((leftHeight > rightHeight)? leftHeight:rightHeight) + 1;

    if((leftHeight - rightHeight >= 2) || (rightHeight - leftHeight >= 2)){
        return 0;
    }
    else
        return L && R;
}

//main method
int main()
{
    struct Node *root = NULL;
    int choice, item;
    do{
        printf("\nBST Operations");
        printf("\nPress 1 to insert.");
        printf("\nPress 2 to delete.");
        printf("\nPress 3 to Pre-order traversal.");
        printf("\nPress 4 to  In-order traversal.");
        printf("\nPress 5 to Postorder traversal");
```



```c
        printf("\nPress 6 to check for full binary tree");
        printf("\nPress 7 to check for Complete Binary tree");
        printf("\nPress 8 to print the height of Binary tree");
        printf("\nPress 9 to check height balance of the tree.");
        printf("\nPress 0 to exit");
        printf("\nEnter your choice (0, 1, 2, 3, 4, 5, 6, 7) : ");
        scanf("%d", &choice);
        switch(choice){
            case 1:{
                printf("\nEnter the item which you want to insert : ");
                scanf("%d", &item);
                root = insertion(root, item);
                break;
            }
            case 2:{
                printf("\nEnter the item which you want to delete : ");
                scanf("%d", &item);
                root = deleteNode(root, item);
                break;
            }
            case 3:{
                preorder(root);
                break;
            }

            case 4:{
                inorder(root);
                break;
            }

            case 5:{
                postorder(root);
                break;
            }
            case 6:{
                bool b = isFullBinaryTree(root);
                if(b){
                    printf("\nIt is a full binary tree.");
                }
                else{
                    printf("\nIt is not a full binary tree.");
                }
                break;
            }
            case 7:{
                int nodeCount = countNumOfNodes(root);
                int index = 0;
                if(isCheckCompleteBinaryTree(root, index, nodeCount)){
                    printf("\nThe tree is complete binary tree.\n");
                }
                else{
                    printf("\nThe tree is not a complete binary tree.\n");
                }
```



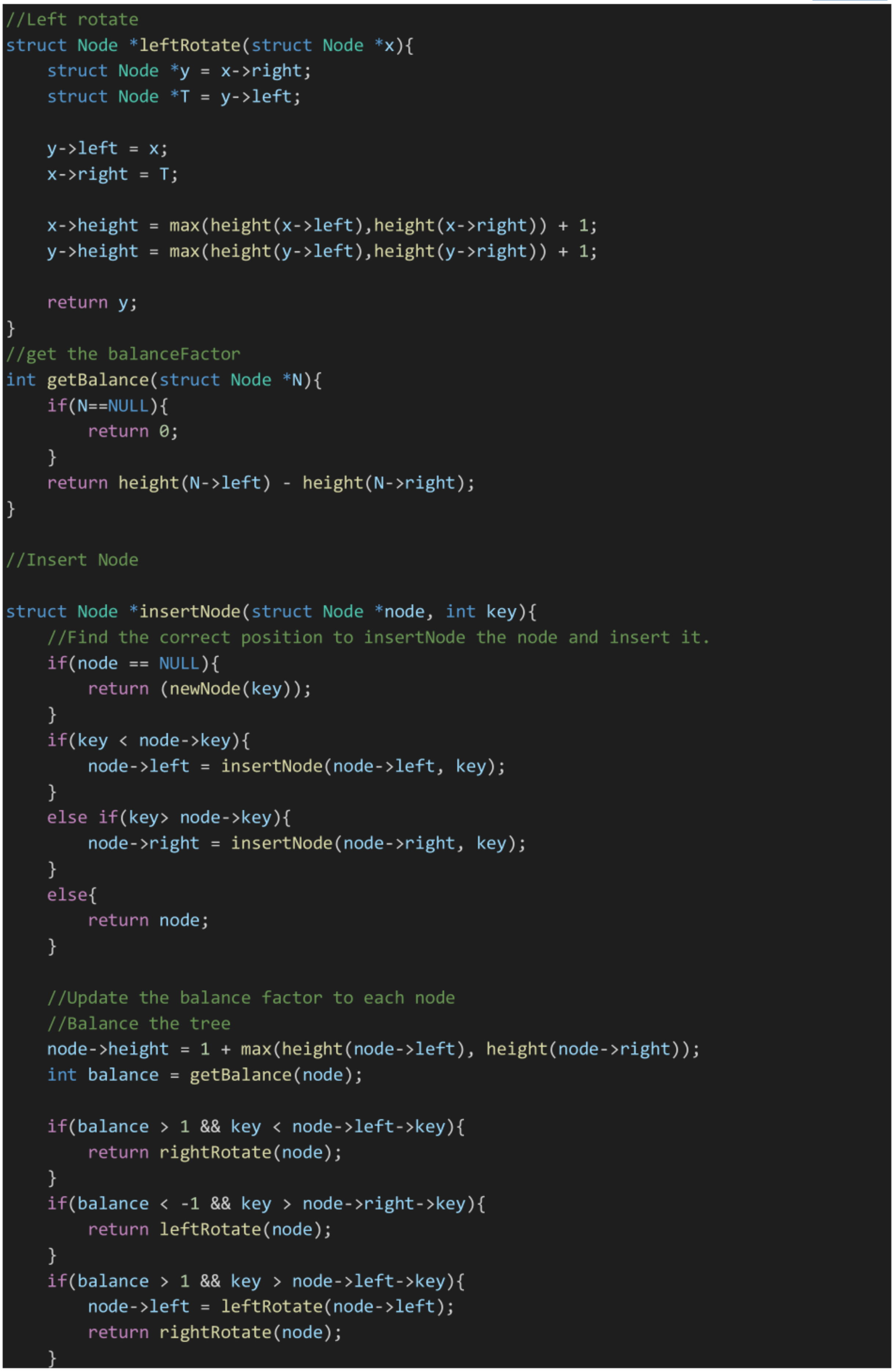
```c
                break;
            }
            case 8:{
                int h = height(root);
                printf("\nHeight of the tree : %d", h);
                break;
            }
            case 9:{
                int height = 0;
                if(checkHeightBalance(root, &height)){
                    printf("\nThe tree is balanced.");
                }
                else{
                    printf("\nThe tree is not balanced.");
                }
                break;
            }
            case 0:{
                exit(choice);
            }
            default:{
                printf("\nWrong option selected.");
            }
        }
    }while(1);
    return 0;
}
```

AVL Tree:

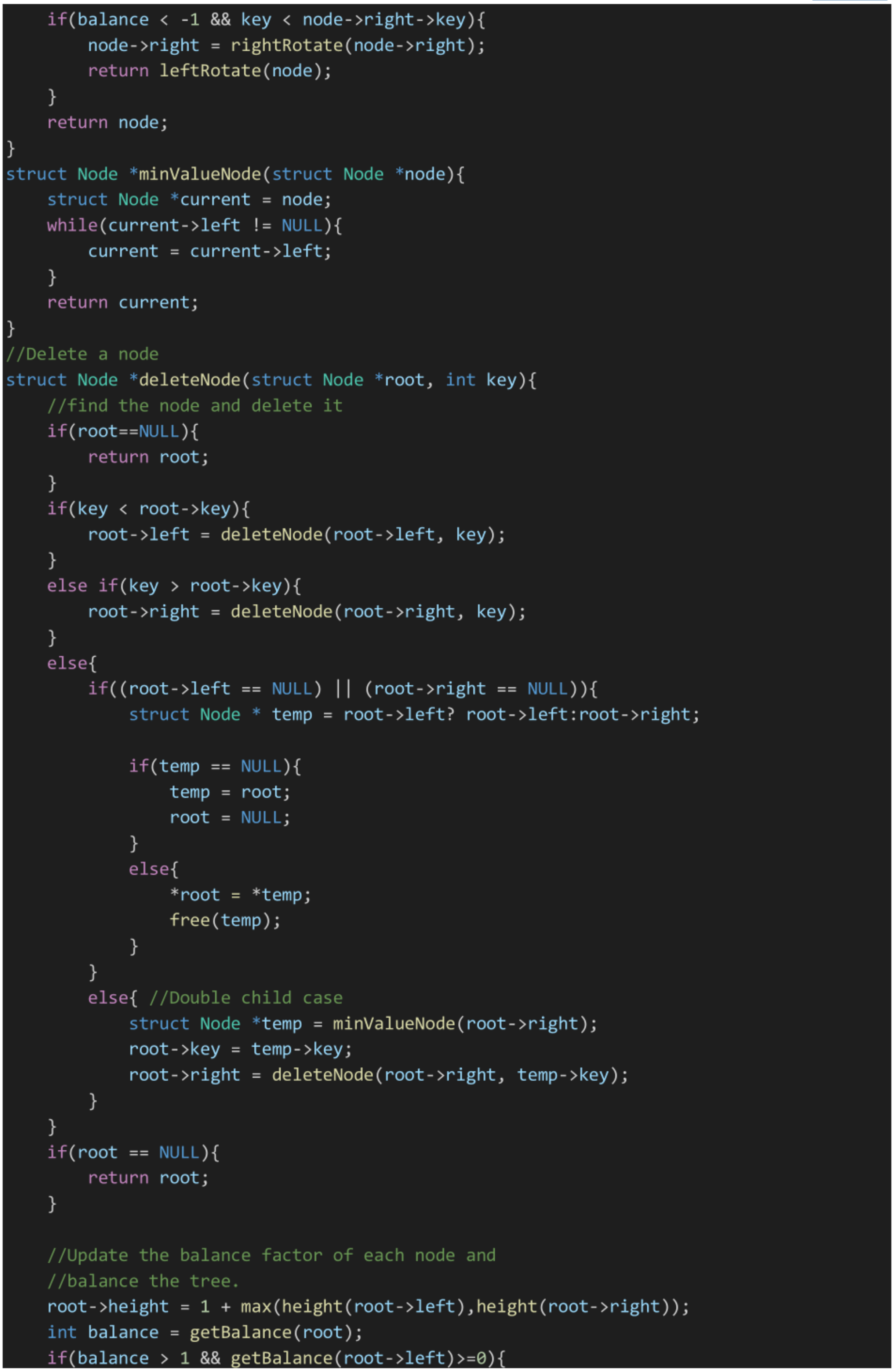
The AVL tree is a self-balancing binary search tree in which each node has additional information called the balance factor, whose value is 1, 0, or +1.

Implementation:

```c
#include<stdio.h>
#include<stdlib.h>
//Create Node
struct Node{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

int max(int a, int b)
{
    return (a > b)? a : b;
}
//calculate height
int height(struct Node *N){
    if(N==NULL){
        return 0;
    }
    return N->height;
}
//Create a node
struct Node* newNode(int key){
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = node->right = NULL;
    node->height = 1;
    return (node);
}
//Right rotate
struct Node *rightRotate(struct Node *Y){
    struct Node *x = Y->left;
    struct Node *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;
}
```



```c
//Left rotate
struct Node *leftRotate(struct Node *x){
    struct Node *y = x->right;
    struct Node *T = y->left;

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

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

    return y;
}
//get the balanceFactor
int getBalance(struct Node *N){
    if(N==NULL){
        return 0;
    }
    return height(N->left) - height(N->right);
}

//Insert Node

struct Node *insertNode(struct Node *node, int key){
    //Find the correct position to insertNode the node and insert it.
    if(node == NULL){
        return (newNode(key));
    }
    if(key < node->key){
        node->left = insertNode(node->left, key);
    }
    else if(key> node->key){
        node->right = insertNode(node->right, key);
    }
    else{
        return node;
    }

    //Update the balance factor to each node
    //Balance the tree
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);

    if(balance > 1 && key < node->left->key){
        return rightRotate(node);
    }
    if(balance < -1 && key > node->right->key){
        return leftRotate(node);
    }
    if(balance > 1 && key > node->left->key){
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
```



```
    if(balance < -1 && key < node->right->key){
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
struct Node *minValueNode(struct Node *node){
    struct Node *current = node;
    while(current->left != NULL){
        current = current->left;
    }
    return current;
}
//Delete a node
struct Node *deleteNode(struct Node *root, int key){
    //find the node and delete it
    if(root==NULL){
        return root;
    }
    if(key < root->key){
        root->left = deleteNode(root->left, key);
    }
    else if(key > root->key){
        root->right = deleteNode(root->right, key);
    }
    else{
        if((root->left == NULL) || (root->right == NULL)){
            struct Node * temp = root->left? root->left:root->right;

            if(temp == NULL){
                temp = root;
                root = NULL;
            }
            else{
                *root = *temp;
                free(temp);
            }
        }
        else{ //Double child case
            struct Node *temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }
    if(root == NULL){
        return root;
    }

    //Update the balance factor of each node and
    //balance the tree.
    root->height = 1 + max(height(root->left),height(root->right));
    int balance = getBalance(root);
    if(balance > 1 && getBalance(root->left)>=0){
```

```c
        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;
}
//Print the tree
void printPreOrder(struct Node * root){
    if(root!=NULL){
        printf("%d ", root->key);
        printPreOrder(root->left);
        printPreOrder(root->right);
    }
}
int main()
{
    struct Node *root = NULL;

    root = insertNode(root, 20);
    root = insertNode(root, 10);
    root = insertNode(root, 70);
    root = insertNode(root, 40);
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 90);
    root = insertNode(root, 80);
    root = insertNode(root, 60);
    printPreOrder(root);
    //root = deleteNode(root, 30);
    root = deleteNode(root, 70);
    printf("\nAfter deletion : ");
    printPreOrder(root);
    return 0;
}
```