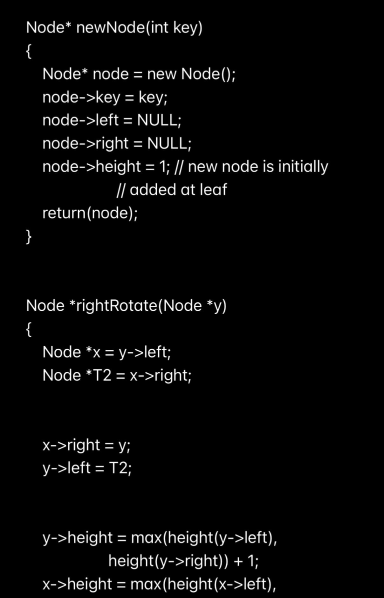
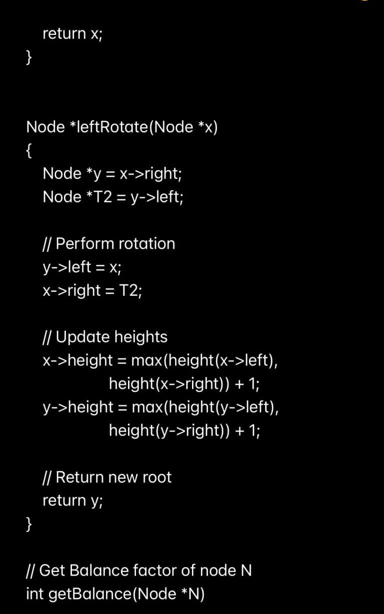
```
#include<bits/stdc++.h>
using namespace std;

class Node
{
   public:
   int key;
   Node *left;
   Node *right;
   int height;
};

int max(int a, int b);


int height(Node *N)
{
   if (N == NULL)
      return 0;
   return N->height;
}

int max(int a, int b)
{
   return (a > b)? a : b;
```

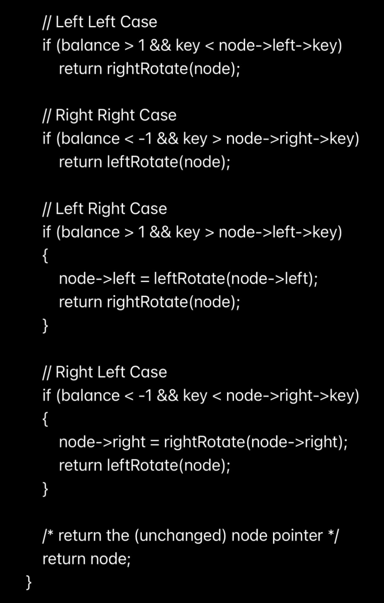


```
Node* newNode(int key)
{
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially
                      // added at leaf
    return(node);
}


Node *rightRotate(Node *y)
{
    Node *x = y->left;
    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),
```



```
    return x;
}


Node *leftRotate(Node *x)
{
    Node *y = x->right;
    Node *T2 = y->left;

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

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

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(Node *N)
```

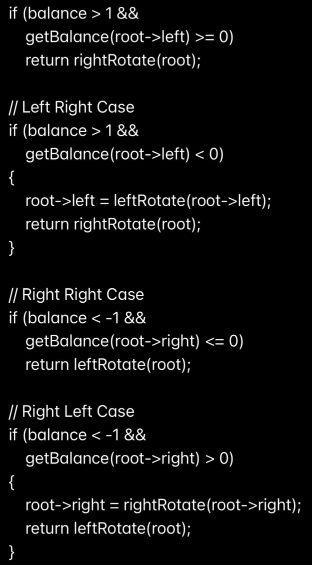
```
    if (N == NULL)
        return 0;
    return height(N->left) -
           height(N->right);
}

Node* insert(Node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                        height(node->right));


    int balance = getBalance(node);
```



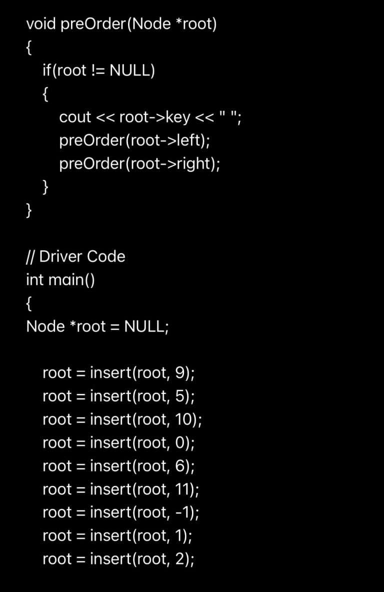
```
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

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

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

    /* return the (unchanged) node pointer */
    return node;
}
```

```
Node * minValueNode(Node* node)
{
   Node* current = node;

   /* loop down to find the leftmost leaf */
   while (current->left != NULL)
      current = current->left;

   return current;
}


Node* deleteNode(Node* root, int key)
{

   // STEP 1: PERFORM STANDARD BST
DELETE
   if (root == NULL)
      return root;


   if ( key < root->key )
      root->left = deleteNode(root->left, key);
```

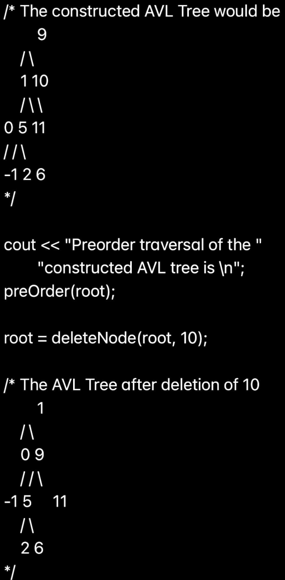
```
    else if( key > root->key )
        root->right = deleteNode(root->right,
key);


    else
    {
        // node with only one child or no child
        if( (root->left == NULL) ||
            (root->right == NULL) )
        {
            Node *temp = root->left ?
                         root->left :
                         root->right;

            // No child case
            if (temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
            *root = *temp; // Copy the contents of
                           // the non-empty child
            free(temp);
        else
        {

            Node* temp = minValueNode(root-
>right);


            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right,
                                     temp->key);
        }
    }


    if (root == NULL)
    return root;

    // STEP 2: UPDATE HEIGHT OF THE
CURRENT NODE
    root->height = 1 + max(height(root->left),
                           height(root->right));


    int balance = getBalance(root);
```



```
if (balance > 1 &&
    getBalance(root->left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 &&
    getBalance(root->left) < 0)
{
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 &&
    getBalance(root->right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 &&
    getBalance(root->right) > 0)
{
    root->right = rightRotate(root->right);
    return leftRotate(root);
}
```



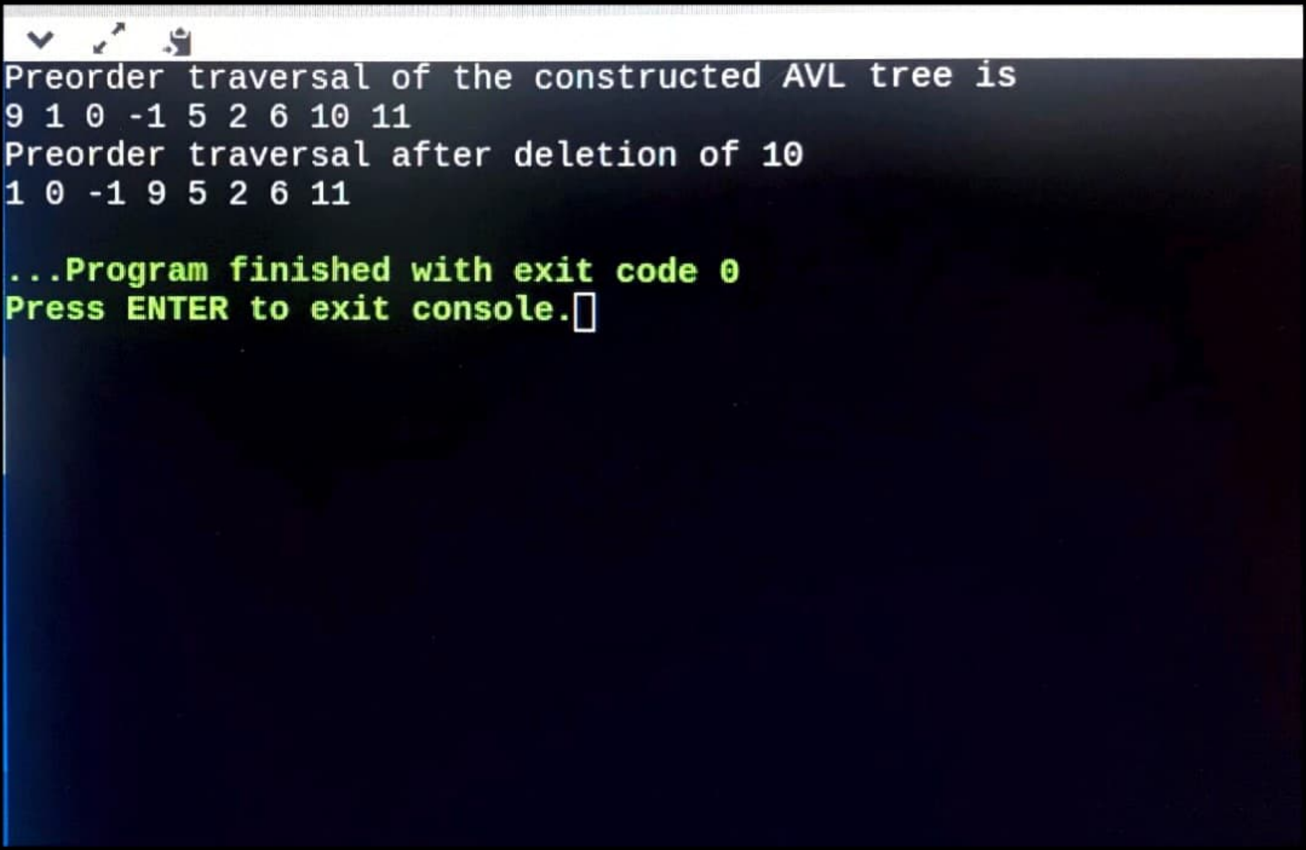
```
void preOrder(Node *root)
{
    if(root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver Code
int main()
{
Node *root = NULL;

    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);
```



```
/* The constructed AVL Tree would be
        9
    /\
    1 10
    /\\
0 5 11
//\
-1 2 6
*/

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

root = deleteNode(root, 10);

/* The AVL Tree after deletion of 10
        1
    /\
    0 9
    //\
-1 5     11
    /\
    2 6
*/
```

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

    return 0;
}
```

Output:



```
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

...Program finished with exit code 0
Press ENTER to exit console.
```