// C++ program to delete a node from AVL Tree

#include<bits/stdc++.h>

using namespace std;

// An AVL tree node

class Node

{

    public:

    int key;

    Node \*left;

    Node \*right;

    int height;

};

// A utility function to get maximum

// of two integers

int max(int a, int b);

// A utility function to get height

// of the tree

int height(Node \*N)

{

    if (N == NULL)

        return 0;

    return N->height;

}

// A utility function to get maximum

// of two integers

int max(int a, int b)

{

    return (a > b)? a : b;

}

/\* Helper function that allocates a

   new node with the given key and

   NULL left and right pointers. \*/

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);

}

// A utility function to right

// rotate subtree rooted with y

// See the diagram given above.

Node \*rightRotate(Node \*y)

{

    Node \*x = y->left;

    Node \*T2 = x->right;

    // Perform rotation

    x->right = y;

    y->left = T2;

    // Update heights

    y->height = max(height(y->left),

                    height(y->right)) + 1;

    x->height = max(height(x->left),

                    height(x->right)) + 1;

    // Return new root

    return x;

}

// A utility function to left

// rotate subtree rooted with x

// See the diagram given above.

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));

    /\* 3. Get the balance factor of this

        ancestor node to check whether

        this node became unbalanced \*/

    int balance = getBalance(node);

    // If this node becomes unbalanced,

    // then there are 4 cases

    // 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;

}

/\* Given a non-empty binary search tree,

return the node with minimum key value

found in that tree. Note that the entire

tree does not need to be searched. \*/

Node \* minValueNode(Node\* node)

{

    Node\* current = node;

    /\* loop down to find the leftmost leaf \*/

    while (current->left != NULL)

        current = current->left;

    return current;

}

// Recursive function to delete a node

// with given key from subtree with

// given root. It returns root of the

// modified subtree.

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

{

    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)

        return root;

    // If the key to be deleted is smaller

    // than the root's key, then it lies

    // in left subtree

    if ( key < root->key )

        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater

    // than the root's key, then it lies

    // in right subtree

    else if( key > root->key )

        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then

    // This is the node to be deleted

    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 with two children: Get the inorder

            // successor (smallest in the right subtree)

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

            // Copy the inorder successor's

            // data to this node

            root->key = temp->key;

            // Delete the inorder successor

            root->right = deleteNode(root->right,

                                     temp->key);

        }

    }

    // If the tree had only one node

    // then return

    if (root == NULL)

    return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE

    root->height = 1 + max(height(root->left),

                           height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF

    // THIS NODE (to check whether this

    // node became unbalanced)

    int balance = getBalance(root);

    // If this node becomes unbalanced,

    // then there are 4 cases

    // Left Left Case

    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);

    }

    return root;

}

// A utility function to print preorder

// traversal of the tree.

// The function also prints height

// of every node

void preOrder(Node \*root)

{

    if(root != NULL)

    {

        cout << root->key << " ";

        preOrder(root->left);

        preOrder(root->right);

    }

}

// Driver Code

int main()

{

Node \*root = NULL;

    /\* Constructing tree given in

    the above figure \*/

    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;

}