

**DSA Lab(Data Structured and Algorithms Lab)**

**Assignment # 12**

**Semester**: 3ndSemester

**Section**: C

**Submitted To:**

**Mr. Abdullah Shahrose**

**Submitted By:**

**Name**: Abdul Ahad

**Roll No**: 22-CS-071

**Task:**

#include <iostream>

using namespace std;

class Node

{

public:

    int key;

    Node \*left;

    Node \*right;

    int height;

    Node(int key)

    {

        this->key = key;

        left = nullptr;

        right = nullptr;

        height = 1;

    }

};

// AVL Tree Class

class AVLTree

{

private:

    Node \*root;

    int getHeight(Node \*node)

    {

        if (node == nullptr)

            return 0;

        return node->height;

    }

    int getBalanceFactor(Node \*node)

    {

        if (node == nullptr)

            return 0;

        return getHeight(node->left) - getHeight(node->right);

    }

    Node \*rotateLeft(Node \*node)

    {

        Node \*newRoot = node->right;

        Node \*temp = newRoot->left;

        newRoot->left = node;

        node->right = temp;

        node->height = 1 + max(getHeight(node->left), getHeight(node->right));

        newRoot->height = 1 + max(getHeight(newRoot->left), getHeight(newRoot->right));

        return newRoot;

    }

    Node \*rotateRight(Node \*node)

    {

        Node \*newRoot = node->left;

        Node \*temp = newRoot->right;

        newRoot->right = node;

        node->left = temp;

        node->height = 1 + max(getHeight(node->left), getHeight(node->right));

        newRoot->height = 1 + max(getHeight(newRoot->left), getHeight(newRoot->right));

        return newRoot;

    }

    Node \*insertNode(Node \*node, int key)

    {

        if (node == nullptr)

            return new Node(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;

        node->height = 1 + max(getHeight(node->left), getHeight(node->right));

        int balanceFactor = getBalanceFactor(node);

        if (balanceFactor > 1)

        {

            if (key < node->left->key)

                return rotateRight(node);

            else

            {

                node->left = rotateLeft(node->left);

                return rotateRight(node);

            }

        }

        if (balanceFactor < -1)

        {

            if (key > node->right->key)

                return rotateLeft(node);

            else

            {

                node->right = rotateRight(node->right);

                return rotateLeft(node);

            }

        }

        return node;

    }

    Node \*findMinNode(Node \*node)

    {

        Node \*current = node;

        while (current->left != nullptr)

            current = current->left;

        return current;

    }

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

    {

        if (node == nullptr)

            return node;

        if (key < node->key)

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

        else if (key > node->key)

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

        else

        {

            if (node->left == nullptr || node->right == nullptr)

            {

                Node \*temp = node->left ? node->left : node->right;

                if (temp == nullptr)

                {

                    temp = node;

                    node = nullptr;

                }

                else

                    \*node = \*temp;

                delete temp;

            }

            else

            {

                Node \*temp = findMinNode(node->right);

                node->key = temp->key;

                node->right = deleteNode(node->right, temp->key);

            }

        }

        if (node == nullptr)

            return node;

        node->height = 1 + max(getHeight(node->left), getHeight(node->right));

        int balanceFactor = getBalanceFactor(node);

        if (balanceFactor > 1)

        {

            if (getBalanceFactor(node->left) >= 0)

                return rotateRight(node);

            else

            {

                node->left = rotateLeft(node->left);

                return rotateRight(node);

            }

        }

        if (balanceFactor < -1)

        {

            if (getBalanceFactor(node->right) <= 0)

                return rotateLeft(node);

            else

            {

                node->right = rotateRight(node->right);

                return rotateLeft(node);

            }

        }

        return node;

    }

    void inorderTraversal(Node \*node)

    {

        if (node != nullptr)

        {

            inorderTraversal(node->left);

            cout << node->key << " ";

            inorderTraversal(node->right);

        }

    }

public:

    AVLTree()

    {

        root = nullptr;

    }

    void insert(int key)

    {

        root = insertNode(root, key);

    }

    void remove(int key)

    {

        root = deleteNode(root, key);

    }

    bool search(int key)

    {

        Node \*current = root;

        while (current != nullptr)

        {

            if (key == current->key)

                return true;

            else if (key < current->key)

                current = current->left;

            else

                current = current->right;

        }

        return false;

    }

    void printSorted()

    {

        inorderTraversal(root);

        cout << endl;

    }

};

int main()

{

    AVLTree avlTree;

    avlTree.insert(50);

    avlTree.printSorted();

    avlTree.insert(30);

    avlTree.printSorted();

    avlTree.insert(70);

    avlTree.printSorted();

    avlTree.insert(20);

    avlTree.printSorted();

    avlTree.insert(40);

    avlTree.printSorted();

    avlTree.insert(60);

    avlTree.printSorted();

    avlTree.insert(50);

    avlTree.printSorted();

    avlTree.remove(40);

    avlTree.printSorted();

    cout << "Key 30 found: " << (avlTree.search(30) ? "Yes" : "No") << endl;

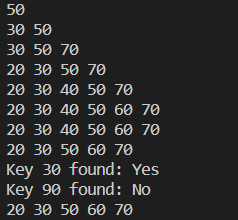
    cout << "Key 90 found: " << (avlTree.search(90) ? "Yes" : "No") << endl;

    avlTree.printSorted();

    return 0;

}

**Output:**

****