

**Data Structures And Algorithms**

**Lab Task 08**

**SUBMITTED BY:**

Hasaan Ahmad SP22-BSE-017

**SUBMITTED TO: Sir Ahmad Qasim**

**All Activities and Graded tasks are Included in the code.**

**Code:**

// Hasaan Ahmad(SP22-BSE-017)

// LAB 09

// AVL TREES

// Insertion, Deletion, Level Order Traversal

// 1. Insertion

// 2. Deletion

// 3. Level Order Traversal

#include <iostream>

#include <queue>

using namespace std;

// AVL TREES

class Node

{

public:

    int data;

    Node \*left;

    Node \*right;

    int height;

    Node(int data)

    {

        this->data = data;

        this->left = NULL;

        this->right = NULL;

        this->height = 1;

    }

};

// Height of a node. Height is treated as

int height(Node \*root)

{

    if (root == NULL)

    {

        return 0;

    }

    return root->height;

}

Node \*rotateRight(Node \*root)

{

    Node \*newRoot = root->left;

    Node \*temp = newRoot->right;

    newRoot->right = root;

    root->left = temp;

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

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

    return newRoot;

}

Node \*rotateLeft(Node \*root)

{

    Node \*newRoot = root->right;

    Node \*temp = newRoot->left;

    newRoot->left = root;

    root->right = temp;

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

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

    return newRoot;

}

Node \*leftRight(Node \*root)

{

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

    return rotateRight(root);

}

Node \*rightLeft(Node \*root)

{

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

    return rotateLeft(root);

}

int balanceFactor(Node \*root)

{

    if (root == NULL)

    {

        return 0;

    }

    return height(root->left) - height(root->right);

}

Node \*insertAVL(Node \*root, int data)

{

    if (root == NULL)

    {

        return new Node(data);

    }

    if (data < root->data)

    {

        root->left = insertAVL(root->left, data);

    }

    else if (data > root->data)

    {

        root->right = insertAVL(root->right, data);

    }

    else

    {

        return root;

    }

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

    int bf = balanceFactor(root);

    if (bf > 1 && data < root->left->data)

    {

        return rotateRight(root);

    }

    if (bf < -1 && data > root->right->data)

    {

        return rotateLeft(root);

    }

    if (bf > 1 && data > root->left->data)

    {

        return leftRight(root);

    }

    if (bf < -1 && data < root->right->data)

    {

        return rightLeft(root);

    }

    return root;

}

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

{

    if (root == NULL)

    {

        return root;

    }

    if (key < root->data)

    {

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

    }

    else if (key > root->data)

    {

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

    }

    else

    {

        if (root->left == NULL)

        {

            Node \*temp = root->right;

            delete root;

            return temp;

        }

        else if (root->right == NULL)

        {

            Node \*temp = root->left;

            delete root;

            return temp;

        }

        else

        {

            Node \*temp = root->right;

            while (temp->left != NULL)

            {

                temp = temp->left;

            }

            root->data = temp->data;

            root->right = deleteFromAVL(root->right, temp->data);

        }

    }

    if (root == NULL)

    {

        return root;

    }

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

    int bf = balanceFactor(root);

    if (bf > 1 && balanceFactor(root->left) >= 0)

    {

        return rotateRight(root);

    }

    if (bf < -1 && balanceFactor(root->right) <= 0)

    {

        return rotateLeft(root);

    }

    if (bf > 1 && balanceFactor(root->left) < 0)

    {

        return leftRight(root);

    }

    if (bf < -1 && balanceFactor(root->right) > 0)

    {

        return rightLeft(root);

    }

    return root;

}

// Level Order Traversal

Node\* levelOrderTraversal(Node\* root, int level)

{

    if (root == NULL)

    {

        return NULL;

    }

    if (level == 1)

    {

        cout << root->data << " ";

    }

    else if (level > 1)

    {

        levelOrderTraversal(root->left, level - 1);

        levelOrderTraversal(root->right, level - 1);

    }

    return root;

}

Node\* levelOrder(Node\* root)

{

    int h = height(root);

    for (int i = 1; i <= h; i++)

    {

        levelOrderTraversal(root, i);

    }

    return root;

}

int main()

{

    Node \*root = NULL;

    root = insertAVL(root, 10);

    root = insertAVL(root, 20);

    root = insertAVL(root, 30);

    root = insertAVL(root, 40);

    root = insertAVL(root, 50);

    root = insertAVL(root, 25);

    root = insertAVL(root, 5);

    root = insertAVL(root, 15);

    root = insertAVL(root, 35);

    root = insertAVL(root, 45);

    root = insertAVL(root, 55);

    root = insertAVL(root, 60);

    root = insertAVL(root, 65);

    root = insertAVL(root, 70);

    root = insertAVL(root, 75);

    root = insertAVL(root, 80);

    root = insertAVL(root, 85);

    root = insertAVL(root, 90);

    root = insertAVL(root, 95);

    root = insertAVL(root, 100);

    // Level Order Traversal

    for (int i = 1; i <= height(root); i++)

    {

        levelOrderTraversal(root, i);

    }

    cout << endl;

    root = deleteFromAVL(root, 10);

    root = deleteFromAVL(root, 20);

    root = deleteFromAVL(root, 30);

    root = deleteFromAVL(root, 40);

    root = deleteFromAVL(root, 50);

    // Level Order Traversal

    levelOrder(root);

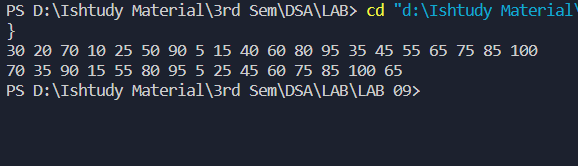
    cout << endl;

    return 0;

}

// This is Hasaan Signing Off. See you in the next one.

**Output**

****