**WEEK 8 LAB B**

1.

#include <iostream>

using namespace std;

struct Node {

    int key;

    Node\* left;

    Node\* right;

    int height;

};

int height(Node\* n) {

    if (n == NULL)

        return 0;

    return n->height;

}

Node\* newNode(int key) {

    Node\* node = new Node();

    node->key = key;

    node->left = NULL;

    node->right = NULL;

    node->height = 1;

    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), height(x->right)) + 1;

    return x;

}

Node\* leftRotate(Node\* x) {

    Node\* y = x->right;

    Node\* T2 = y->left;

    y->left = x;

    x->right = T2;

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

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

    return y;

}

int getBalance(Node\* n) {

    if (n == NULL)

        return 0;

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

}

Node\* insert(Node\* node, int key) {

    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

        return node;

    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;

}

Node\* minValueNode(Node\* node) {

    Node\* current = node;

    while (current->left != NULL)

        current = current->left;

    return current;

}

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

    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)) {

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

            if (temp == NULL) {

                temp = root;

                root = NULL;

            } else

                \*root = \*temp;

            delete temp;

        } else {

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

            root->key = temp->key;

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

        }

    }

    if (root == NULL)

        return root;

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

    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0)

        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;

}

void preOrder(Node\* root) {

    if (root != NULL) {

        cout << root->key << " ";

        preOrder(root->left);

        preOrder(root->right);

    }

}

int main() {

    Node\* root = NULL;

    // Insert the values from the AVL tree image

    root = insert(root, 18);

    root = insert(root, 15);

    root = insert(root, 24);

    root = insert(root, 8);

    root = insert(root, 17);

    root = insert(root, 22);

    root = insert(root, 28);

    root = insert(root, 31);

    cout << "Preorder traversal before deletion: ";

    preOrder(root);

    cout << endl;

    // Delete the specified nodes

    root = deleteNode(root, 24);

    root = deleteNode(root, 15);

    root = deleteNode(root, 18);

    root = deleteNode(root, 22);

    root = deleteNode(root, 17);

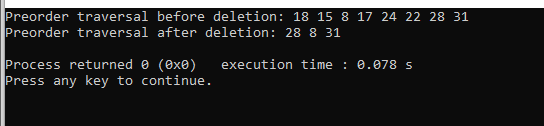
    cout << "Preorder traversal after deletion: ";

    preOrder(root);

    cout << endl;

    return 0;

}



2.

a)

#include <iostream>

using namespace std;

class Node {

public:

int key;

Node\* left;

Node\* right;

Node(int value) {

key = value;

left = NULL;

right = NULL;

}

};

int height(Node\* node) {

if (!node) {

return -1;

}

int leftHeight = height(node->left);

int rightHeight = height(node->right);

return 1 + max(leftHeight, rightHeight);

}

Node\* insert(Node\* node, int value) {

if (!node) {

return new Node(value);

}

if (value < node->key) {

node->left = insert(node->left, value);

} else {

node->right = insert(node->right, value);

}

return node;

}

int main() {

Node\* root = NULL;

int n, value;

cout << "Enter the number of nodes: ";

cin >> n;

cout << "Enter the node values:\n";

for (int i = 0; i < n; i++) {

cin >> value;

root = insert(root, value);

}

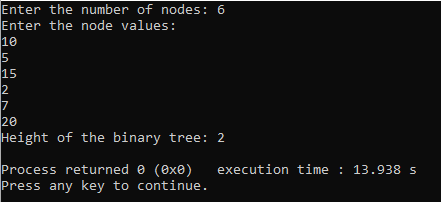
int treeHeight = height(root);

cout << "Height of the binary tree: " << treeHeight << endl;

delete root;

return 0;

}



b) #include <iostream>

#include <climits>

using namespace std;

struct node {

    int key;

    node\* left;

    node\* right;

};

node\* newnode(int key) {

    node\* n = new node();

    n->key = key;

    n->left = NULL;

    n->right = NULL;

    return n;

}

node\* constructbst(int preorder[], int\* preorderindex, int key, int min, int max, int n) {

    if (\*preorderindex >= n)

        return NULL;

    node\* root = NULL;

    if (key > min && key < max) {

        root = newnode(key);

        \*preorderindex = \*preorderindex + 1;

        if (\*preorderindex < n) {

            root->left = constructbst(preorder, preorderindex, preorder[\*preorderindex], min, key, n);

        }

        if (\*preorderindex < n) {

            root->right = constructbst(preorder, preorderindex, preorder[\*preorderindex], key, max, n);

        }

    }

    return root;

}

int height(node\* n) {

    if (n == NULL)

        return -1;

    int leftheight = height(n->left);

    int rightheight = height(n->right);

    return 1 + max(leftheight, rightheight);

}

bool isavl(node\* root) {

    if (root == NULL)

        return true;

    int leftheight = height(root->left);

    int rightheight = height(root->right);

    int balancefactor = leftheight - rightheight;

    if (balancefactor > 1 || balancefactor < -1)

        return false;

    return isavl(root->left) && isavl(root->right);

}

int main() {

    int preorder1[] = {20, 10, 15, 18, 30, 25, 40};

    int preorder2[] = {20, 15, 18, 30, 25, 40};

    int n1 = sizeof(preorder1) / sizeof(preorder1[0]);

    int n2 = sizeof(preorder2) / sizeof(preorder2[0]);

    int preorderindex1 = 0;

    int preorderindex2 = 0;

    node\* root1 = constructbst(preorder1, &preorderindex1, preorder1[0], INT\_MIN, INT\_MAX, n1);

    node\* root2 = constructbst(preorder2, &preorderindex2, preorder2[0], INT\_MIN, INT\_MAX, n2);

    if (isavl(root1))

        cout << "preorder1 forms a valid avl tree" << endl;

    else

        cout << "preorder1 does not form a valid avl tree" << endl;

    if (isavl(root2))

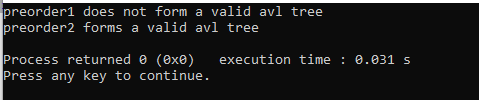
        cout << "preorder2 forms a valid avl tree" << endl;

    else

        cout << "preorder2 does not form a valid avl tree" << endl;

    return 0;

}



c)

#include <iostream>

#include <vector>

using namespace std;

class Node {

public:

int key;

Node\* left;

Node\* right;

int height;

Node(int value) {

key = value;

left = NULL;

right = NULL;

height = 1;

}

};

int getHeight(Node\* node) {

return node ? node->height : 0;

}

int getBalance(Node\* node) {

return node ? getHeight(node->left) - getHeight(node->right) : 0;

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

Node\* insert(Node\* node, int key) {

if (!node) return new Node(key);

if (key < node->key)

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

else if (key > node->key)

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

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

}

void inorder(Node\* node, vector<int>& nodes) {

if (!node) return;

inorder(node->left, nodes);

nodes.push\_back(node->key);

inorder(node->right, nodes);

}

vector<int> mergeSortedArrays(const vector<int>& arr1, const vector<int>& arr2) {

vector<int> merged(arr1.size() + arr2.size());

int i = 0, j = 0, k = 0;

while (i < arr1.size() && j < arr2.size()) {

if (arr1[i] < arr2[j]) {

merged[k++] = arr1[i++];

} else {

merged[k++] = arr2[j++];

}

}

while (i < arr1.size()) merged[k++] = arr1[i++];

while (j < arr2.size()) merged[k++] = arr2[j++];

return merged;

}

Node\* buildAVLTreeFromSortedArray(const vector<int>& sortedArray, int start, int end) {

if (start > end) return NULL;

int mid = (start + end) / 2;

Node\* root = new Node(sortedArray[mid]);

root->left = buildAVLTreeFromSortedArray(sortedArray, start, mid - 1);

root->right = buildAVLTreeFromSortedArray(sortedArray, mid + 1, end);

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

return root;

}

Node\* mergeAVLTrees(Node\* rootA, Node\* rootB) {

vector<int> nodesA;

vector<int> nodesB;

inorder(rootA, nodesA);

inorder(rootB, nodesB);

vector<int> mergedNodes = mergeSortedArrays(nodesA, nodesB);

return buildAVLTreeFromSortedArray(mergedNodes, 0, mergedNodes.size() - 1);

}

void printInOrder(Node\* node) {

if (!node) return;

printInOrder(node->left);

cout << node->key << " ";

printInOrder(node->right);

}

int main() {

Node\* rootA = NULL;

Node\* rootB = NULL;

rootA = insert(rootA, 30);

rootA = insert(rootA, 20);

rootA = insert(rootA, 40);

rootA = insert(rootA, 10);

rootA = insert(rootA, 25);

rootB = insert(rootB, 35);

rootB = insert(rootB, 15);

rootB = insert(rootB, 45);

rootB = insert(rootB, 5);

rootB = insert(rootB, 22);

Node\* mergedRoot = mergeAVLTrees(rootA, rootB);

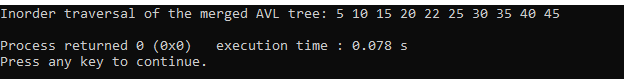
cout << "Inorder traversal of the merged AVL tree: ";

printInOrder(mergedRoot);

cout << endl;

return 0;

}



d)

#include <iostream>

#include <cmath>

using namespace std;

class Node {

public:

int key;

Node\* left;

Node\* right;

int height;

Node(int value) {

key = value;

left = NULL;

right = NULL;

height = 1;

}

};

int getHeight(Node\* node) {

return node ? node->height : 0;

}

int getBalance(Node\* node) {

return node ? getHeight(node->left) - getHeight(node->right) : 0;

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

Node\* rebalance(Node\* node) {

if (!node) return NULL;

int balance = getBalance(node);

if (balance > 1) {

if (getBalance(node->left) < 0)

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

return rightRotate(node);

}

if (balance < -1) {

if (getBalance(node->right) > 0)

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

return leftRotate(node);

}

return node;

}

Node\* deleteSubtree(Node\* root, Node\* nodeToDelete) {

if (!root) return NULL;

if (root == nodeToDelete) {

nodeToDelete->left = NULL;

nodeToDelete->right = NULL;

return nodeToDelete;

}

root->left = deleteSubtree(root->left, nodeToDelete);

root->right = deleteSubtree(root->right, nodeToDelete);

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

return rebalance(root);

}

void printInOrder(Node\* node) {

if (!node) return;

printInOrder(node->left);

cout << node->key << " ";

printInOrder(node->right);

}

Node\* insert(Node\* node, int key) {

if (!node) return new Node(key);

if (key < node->key)

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

else if (key > node->key)

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

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

return rebalance(node);

}

int main() {

Node\* root = NULL;

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 10);

root = insert(root, 25);

cout << "Original AVL Tree Inorder: ";

printInOrder(root);

cout << endl;

Node\* nodeToDelete = root->left; // Assuming we want to delete subtree rooted at 20

deleteSubtree(root, nodeToDelete);

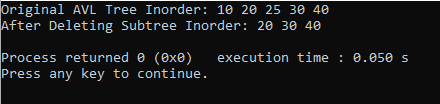
cout << "After Deleting Subtree Inorder: ";

printInOrder(root);

cout << endl;

return 0;

}



e)

#include <iostream>

using namespace std;

struct node {

    int key;

    node\* left;

    node\* right;

    int height;

};

node\* newnode(int key) {

    node\* n = new node();

    n->key = key;

    n->left = NULL;

    n->right = NULL;

    n->height = 1;

    return n;

}

int getheight(node\* n) {

    if (n == NULL)

        return 0;

    return n->height;

}

int updateheight(node\* n) {

    return 1 + max(getheight(n->left), getheight(n->right));

}

int getbalance(node\* n) {

    if (n == NULL)

        return 0;

    return getheight(n->left) - getheight(n->right);

}

node\* rightrotate(node\* y) {

    node\* x = y->left;

    node\* t2 = x->right;

    x->right = y;

    y->left = t2;

    y->height = updateheight(y);

    x->height = updateheight(x);

    return x;

}

node\* leftrotate(node\* x) {

    node\* y = x->right;

    node\* t2 = y->left;

    y->left = x;

    x->right = t2;

    x->height = updateheight(x);

    y->height = updateheight(y);

    return y;

}

node\* insert(node\* root, int key) {

    if (root == NULL)

        return newnode(key);

    if (key < root->key)

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

    else if (key > root->key)

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

    else

        return root;

    root->height = updateheight(root);

    int balance = getbalance(root);

    if (balance > 1 && key < root->left->key)

        return rightrotate(root);

    if (balance < -1 && key > root->right->key)

        return leftrotate(root);

    if (balance > 1 && key > root->left->key) {

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

        return rightrotate(root);

    }

    if (balance < -1 && key < root->right->key) {

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

        return leftrotate(root);

    }

    return root;

}

node\* deletekey(node\* root, int key) {

    if (root == NULL)

        return root;

    if (key < root->key)

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

    else if (key > root->key)

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

    else {

        if ((root->left == NULL) || (root->right == NULL)) {

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

            if (temp == NULL) {

                temp = root;

                root = NULL;

            } else

                \*root = \*temp;

            delete temp;

        } else {

            node\* temp = root->right;

            while (temp->left != NULL)

                temp = temp->left;

            root->key = temp->key;

            root->right = deletekey(root->right, temp->key);

        }

    }

    if (root == NULL)

        return root;

    root->height = updateheight(root);

    int balance = getbalance(root);

    if (balance > 1 && getbalance(root->left) >= 0)

        return rightrotate(root);

    if (balance < -1 && getbalance(root->right) <= 0)

        return leftrotate(root);

    if (balance > 1 && getbalance(root->left) < 0) {

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

        return rightrotate(root);

    }

    if (balance < -1 && getbalance(root->right) > 0) {

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

        return leftrotate(root);

    }

    return root;

}

node\* updatekey(node\* root, int oldkey, int newkey) {

    root = deletekey(root, oldkey);

    root = insert(root, newkey);

    return root;

}

void inorder(node\* root) {

    if (root == NULL)

        return;

    inorder(root->left);

    cout << root->key << " ";

    inorder(root->right);

}

int main() {

    node\* root = NULL;

    root = insert(root, 20);

    root = insert(root, 10);

    root = insert(root, 30);

    root = insert(root, 25);

    root = insert(root, 5);

    cout << "inorder before update: ";

    inorder(root);

    cout << endl;

    int oldkey = 10;

    int delta = 100;

    int newkey = oldkey + delta;  // update operation

    root = updatekey(root, oldkey, newkey);

    cout << "inorder after update: ";

    inorder(root);

    cout << endl;

    return 0;

}

