**Qstn 1**

#include <iostream>

#include <queue>

using namespace std;

// Node structure for the Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = right = NULL;

}

};

// Function to insert a value into the BST

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

if (root == NULL) {

return new Node(value);

}

if (value < root->data) {

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

} else {

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

}

return root;

}

// In-Order traversal

void inOrderTraversal(Node\* root) {

if (root != NULL) {

inOrderTraversal(root->left);

cout << root->data << " ";

inOrderTraversal(root->right);

}

}

// Pre-order traversal

void preOrderTraversal(Node\* root) {

if (root != NULL) {

cout << root->data << " ";

preOrderTraversal(root->left);

preOrderTraversal(root->right);

}

}

// Post-order traversal

void postOrderTraversal(Node\* root) {

if (root != NULL) {

postOrderTraversal(root->left);

postOrderTraversal(root->right);

cout << root->data << " ";

}

}

// BFS Level-Order traversal

void levelOrderTraversal(Node\* root) {

if (root == NULL) {

return;

}

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* current = q.front();

q.pop();

cout << current->data << " ";

if (current->left != NULL) {

q.push(current->left);

}

if (current->right != NULL) {

q.push(current->right);

}

}

}

// Function to delete a node with a given value from the BST

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

if (root == NULL) {

return root;

}

if (key < root->data) {

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

} else if (key > root->data) {

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

} else {

// Node with only one child or no child

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;

}

// Node with two children

Node\* temp = root->right;

while (temp->left != NULL) {

temp = temp->left;

}

root->data = temp->data;

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

}

return root;

}

int main() {

// Constructing the provided BST

Node\* root = new Node(10);

root = insert(root, 5);

root = insert(root, 15);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 12);

root = insert(root, 18);

root = insert(root, 2);

root = insert(root, 4);

root = insert(root, 6);

root = insert(root, 8);

root = insert(root, 11);

root = insert(root, 14);

root = insert(root, 13);

root = insert(root, 19);

root = insert(root, 20);

root = insert(root, 25);

// Part a) Insertion

root = insert(root, 25);

root = insert(root, 12);

// Part b) In-Order Traversal

cout << "In-Order Traversal: ";

inOrderTraversal(root);

cout << endl;

// Part c) Pre-Order Traversal

cout << "Pre-Order Traversal: ";

preOrderTraversal(root);

cout << endl;

// Part d) Post-Order Traversal

cout << "Post-Order Traversal: ";

postOrderTraversal(root);

cout << endl;

// Part e) BFS Level-Order Traversal

cout << "BFS Level-Order Traversal: ";

levelOrderTraversal(root);

cout << endl;

// Part f) Deletion

root = deleteNode(root, 30);

// In-Order Traversal after deletion

cout << "In-Order Traversal after deletion: ";

inOrderTraversal(root);

cout << endl;

return 0;

}

**Output:**

**In-Order Traversal:** 2 3 4 5 6 7 8 10 11 12 12 13 14 15 18 19 20 25 25

**Pre-Order Traversal**: 10 5 3 2 4 7 6 8 15 12 11 14 13 12 18 19 20 25 25

**Post-Order Traversal:** 2 4 3 6 8 7 5 11 12 13 14 12 25 25 20 19 18 15 10

**BFS Level-Order Traversal:** 10 5 15 3 7 12 18 2 4 6 8 11 14 19 13 20 12 25 25

**In-Order Traversal after deletion:** 2 3 4 5 6 7 8 10 11 12 12 13 14 15 18 19 20 25 25

**Qstn 3**

#include <iostream>

using namespace std;

// Node structure for a linked list

struct Node {

int data;

Node\* next;

Node(int value) {

data = value;

next = NULL;

}

};

// Function to remove duplicate nodes from a sorted linked list

void RemoveDuplicateNode(Node\* head) {

if (head == NULL) {

return;

}

Node\* current = head;

while (current != NULL && current->next != NULL) {

if (current->data == current->next->data) {

// Duplicate node found, remove it

Node\* duplicate = current->next;

current->next = current->next->next;

delete duplicate;

} else {

// Move to the next node

current = current->next;

}

}

}

// Function to display the linked list

void DisplayList(Node\* head) {

while (head != NULL) {

cout << head->data;

if (head->next != NULL) {

cout << " -> ";

}

head = head->next;

}

cout << endl;

}

int main() {

// Create a sorted linked list with duplicate nodes

Node\* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(3);

head->next->next->next->next = new Node(8);

cout << "Original List: ";

DisplayList(head);

// Remove duplicates

RemoveDuplicateNode(head);

cout << "List after removing duplicates: ";

DisplayList(head);

// Clean up memory

while (head != NULL) {

Node\* temp = head;

head = head->next;

delete temp;

}

return 0;

}

**Output:**

**Original List: 1 -> 2 -> 3 -> 3 -> 8**

**List after removing duplicates: 1 -> 2 -> 3 -> 8**

**QSTN 2**

#include <iostream>

#include <algorithm>

using namespace std;

// Node structure for AVL Tree

struct AVLNode {

int data;

AVLNode\* left;

AVLNode\* right;

int height;

AVLNode(int value) {

data = value;

left = right = NULL;

height = 1;

}

};

class AVLTree {

private:

AVLNode\* root;

int getHeight(AVLNode\* node) {

if (node == NULL) {

return 0;

}

return node->height;

}

int getBalanceFactor(AVLNode\* node) {

if (node == NULL) {

return 0;

}

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

}

AVLNode\* rightRotate(AVLNode\* y) {

AVLNode\* x = y->left;

AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

AVLNode\* leftRotate(AVLNode\* x) {

AVLNode\* y = x->right;

AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

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

// Perform standard BST insertion

if (node == NULL) {

return new AVLNode(value);

}

if (value < node->data) {

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

} else if (value > node->data) {

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

} else {

// Duplicate values are not allowed

return node;

}

// Update height of current node

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

// Get the balance factor to check if the node became unbalanced

int balance = getBalanceFactor(node);

// Left Left Case

if (balance > 1 && value < node->left->data) {

return rightRotate(node);

}

// Right Right Case

if (balance < -1 && value > node->right->data) {

return leftRotate(node);

}

// Left Right Case

if (balance > 1 && value > node->left->data) {

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

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && value < node->right->data) {

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

return leftRotate(node);

}

// No rotation needed

return node;

}

// Helper function to display AVL tree structure

void showTree(AVLNode\* root, int space) {

if (root == NULL) {

return;

}

space += 5;

showTree(root->right, space);

cout << endl;

for (int i = 5; i < space; i++) {

cout << " ";

}

cout << root->data << "\n";

showTree(root->left, space);

}

public:

AVLTree() {

root = NULL;

}

// Wrapper function for insertion

void insert(int value) {

root = insert(root, value);

}

// Wrapper function to display the AVL tree structure

void show() {

showTree(root, 0);

}

};

int main() {

AVLTree avlTree;

// Inserting nodes into AVL tree

avlTree.insert(25);

avlTree.insert(15);

avlTree.insert(35);

avlTree.insert(10);

avlTree.insert(20);

avlTree.insert(30);

avlTree.insert(40);

avlTree.insert(5);

avlTree.insert(12);

avlTree.insert(18);

avlTree.insert(22);

avlTree.insert(28);

avlTree.insert(32);

avlTree.insert(38);

avlTree.insert(45);

avlTree.insert(2);

avlTree.insert(8);

avlTree.insert(26);

avlTree.insert(36);

avlTree.insert(42);

avlTree.insert(24);

avlTree.insert(39);

avlTree.insert(23);

// Displaying the AVL tree structure

cout << "AVL Tree Structure:" << endl;

avlTree.show();

return 0;

}

**OUTPUT:**

**AVL Tree Structure:**

45

42

40

39

38

36

35

32

30

28

26

25

24

23

22

20

18

15

12

10

8

5

2