**ASSIGNMENT NO. 9**

**Name : Payal Bhagvan Charvande**

**Class: S.Y DIV: IT A**

**PRN NO: 125B2F002**

---

**CODE:**

```
#include <iostream>
#include <queue>
using namespace std;

// Node structure
struct Node {
   int data;
   Node* left;
   Node* right;
   Node(int val) {
      data = val;
      left = right = nullptr;
   }
};
// Class for BST
class BST {
private:
   Node* root;

   // Helper function for insertion
   Node* insert(Node* node, int val) {
      if (node == nullptr) {
```

```
        return new Node(val);
    }
    if (val < node->data)
        node->left = insert(node->left, val);
    else if (val > node->data)
        node->right = insert(node->right, val);
    else
        cout << "Duplicate entry " << val << " not allowed.\n";
    return node;
}
// Helper function to find minimum value in right subtree
Node* findMin(Node* node) {
    while (node && node->left != nullptr)
        node = node->left;
    return node;
}
// Helper function for deletion
Node* remove(Node* node, int val) {
    if (!node) return nullptr;
    if (val < node->data)
        node->left = remove(node->left, val);
    else if (val > node->data)
        node->right = remove(node->right, val);
    else {
        // Node with one child or no child
        if (!node->left) {
            Node* temp = node->right;
            delete node;
```

```
            return temp;
        }
        else if (!node->right) {
            Node* temp = node->left;
            delete node;
            return temp;
        }
        // Node with two children
        Node* temp = findMin(node->right);
        node->data = temp->data;
        node->right = remove(node->right, temp->data);
    }
    return node;
}
// Helper function for searching
bool search(Node* node, int val) {
    if (!node) return false;
    if (node->data == val) return true;
    if (val < node->data)
        return search(node->left, val);
    else
        return search(node->right, val);
}
// Helper functions for traversals
void inorder(Node* node) {
    if (node) {
        inorder(node->left);
        cout << node->data << " ";
```

```
        inorder(node->right);
    }
}
void preorder(Node* node) {
    if (node) {
        cout << node->data << " ";
        preorder(node->left);
        preorder(node->right);
    }
}
void postorder(Node* node) {
    if (node) {
        postorder(node->left);
        postorder(node->right);
        cout << node->data << " ";
    }
}
// Helper function to calculate depth
int depth(Node* node) {
    if (!node) return 0;
    int lDepth = depth(node->left);
    int rDepth = depth(node->right);
    return max(lDepth, rDepth) + 1;
}
// Helper function to create mirror
Node* mirror(Node* node) {
    if (!node) return nullptr;
    Node* mirrored = new Node(node->data);
```

```
    mirrored->left = mirror(node->right);
    mirrored->right = mirror(node->left);
    return mirrored;
}
// Helper function to copy tree
Node* copy(Node* node) {
    if (!node) return nullptr;
    Node* newNode = new Node(node->data);
    newNode->left = copy(node->left);
    newNode->right = copy(node->right);
    return newNode;
}
 // Helper function to display parent nodes
void displayParents(Node* node) {
    if (!node) return;
    if (node->left || node->right) {
        cout << "Parent: " << node->data;
        if (node->left) cout << ", Left Child: " << node->left->data;
        if (node->right) cout << ", Right Child: " << node->right->data;
        cout << endl;
    }
    displayParents(node->left);
    displayParents(node->right);
}
// Helper function to display leaf nodes
void displayLeaves(Node* node) {
    if (!node) return;
    if (!node->left && !node->right)
```

```
        cout << node->data << " ";
    displayLeaves(node->left);
    displayLeaves(node->right);
  }
  // Helper function for level-wise display
  void levelWise(Node* node) {
    if (!node) return;
    queue<Node*> q;
    q.push(node);
    while (!q.empty()) {
      Node* temp = q.front();
      q.pop();
      cout << temp->data << " ";
      if (temp->left) q.push(temp->left);
      if (temp->right) q.push(temp->right);
    }
    cout << endl;
  }
public:
  BST() { root = nullptr; }

  void insert(int val) { root = insert(root, val); }
  void remove(int val) { root = remove(root, val); }
  bool search(int val) { return search(root, val); }

  void displayInorder() { inorder(root); cout << endl; }
  void displayPreorder() { preorder(root); cout << endl; }
  void displayPostorder() { postorder(root); cout << endl; }
```

```
    void displayDepth() { cout << "Depth of tree: " << depth(root) << endl; }

    void displayMirror() {
        Node* mirroredTree = mirror(root);
        cout << "Mirror Inorder: ";
        inorder(mirroredTree);
        cout << endl;
    }
    void createCopy() {
        Node* copiedTree = copy(root);
        cout << "Copied Tree Inorder: ";
        inorder(copiedTree);
        cout << endl;
    }
    void displayParents() { displayParents(root); }
    void displayLeaves() {
        cout << "Leaf Nodes: ";
        displayLeaves(root);
        cout << endl;
    }
    void displayLevelWise() {
        cout << "Level-wise: ";
        levelWise(root);
    }
};
// Main function to test BST
int main() {
```

```
BST tree;


tree.insert(50);
tree.insert(30);
tree.insert(20);
tree.insert(40);
tree.insert(70);
tree.insert(60);
tree.insert(80);
tree.insert(30); // duplicate test

cout << "Inorder Traversal: ";
tree.displayInorder();

cout << "Preorder Traversal: ";
tree.displayPreorder();
cout << "Postorder Traversal: ";
tree.displayPostorder();
tree.displayDepth();
tree.displayMirror();
tree.createCopy();
tree.displayParents();
tree.displayLeaves();
tree.displayLevelWise();
cout << "Searching 40: " << (tree.search(40) ? "Found" : "Not Found") <<      endl;
cout << "Deleting 20\n";
tree.remove(20);
cout << "Inorder Traversal after deletion: ";
```

```
    tree.displayInorder();

    return 0;
}
```

**OUTPUT:**