11) Implement binary search tree and perform following operations:

a) Insert (Handle insertion of duplicate entry)

b) Delete :

```
#include <iostream>

class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) {
        data = value;
        left = right = nullptr;
    }
};

class BinarySearchTree {
private:
    TreeNode* root;

    TreeNode* insertRecursive(TreeNode* root, int key) {
        if (root == nullptr) {
            return new TreeNode(key);
        }

        if (key < root->data) {
            root->left = insertRecursive(root->left, key);
        } else if (key > root->data) {
            root->right = insertRecursive(root->right, key);
        } else {
```

```
        // Duplicate entry handling (optional)
        std::cout << "Duplicate entry ignored: " << key << std::endl;
    }

    return root;
}

TreeNode* findMinValueNode(TreeNode* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

TreeNode* deleteRecursive(TreeNode* root, int key) {
    if (root == nullptr) {
        return root;
    }

    if (key < root->data) {
        root->left = deleteRecursive(root->left, key);
    } else if (key > root->data) {
        root->right = deleteRecursive(root->right, key);
    } else {
        // Node with only one child or no child
        if (root->left == nullptr) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            TreeNode* temp = root->left;
```

```
            delete root;
            return temp;
        }

        // Node with two children, get the inorder successor (smallest in the right subtree)
        TreeNode* temp = findMinValueNode(root->right);

        // Copy the inorder successor's data to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteRecursive(root->right, temp->data);
    }

    return root;
}

void inOrderTraversal(TreeNode* root) {
    if (root != nullptr) {
        inOrderTraversal(root->left);
        std::cout << root->data << " ";
        inOrderTraversal(root->right);
    }
}

public:
BinarySearchTree() : root(nullptr) {}

void insert(int key) {
    root = insertRecursive(root, key);
}
```

```
    void remove(int key) {
        root = deleteRecursive(root, key);
    }

    void inOrderTraversal() {
        inOrderTraversal(root);
        std::cout << std::endl;
    }
};

int main() {
    BinarySearchTree bst;

    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    std::cout << "Inorder traversal before deletion: ";
    bst.inOrderTraversal();

    bst.remove(20);
    std::cout << "Inorder traversal after deleting 20: ";
    bst.inOrderTraversal();

    bst.remove(30);
    std::cout << "Inorder traversal after deleting 30: ";
```

```
    bst.inOrderTraversal();

    bst.remove(50);
    std::cout << "Inorder traversal after deleting 50: ";
    bst.inOrderTraversal();

    return 0;
}
```

12. Implement binary search tree and perform following operations:
a) Insert (Handle insertion of duplicate entry)
b) Search:

```
#include <iostream>

class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) {
        data = value;
        left = right = nullptr;
    }
};

class BinarySearchTree {
private:
    TreeNode* root;

    TreeNode* insertRecursive(TreeNode* root, int key) {
        if (root == nullptr) {
```

```
        return new TreeNode(key);
    }

    if (key < root->data) {
        root->left = insertRecursive(root->left, key);
    } else if (key > root->data) {
        root->right = insertRecursive(root->right, key);
    } else {
        // Duplicate entry handling (optional)
        std::cout << "Duplicate entry ignored: " << key << std::endl;
    }

    return root;
}

TreeNode* searchRecursive(TreeNode* root, int key) {
    if (root == nullptr || root->data == key) {
        return root;
    }

    if (key < root->data) {
        return searchRecursive(root->left, key);
    } else {
        return searchRecursive(root->right, key);
    }
}

void inOrderTraversal(TreeNode* root) {
    if (root != nullptr) {
        inOrderTraversal(root->left);
        std::cout << root->data << " ";
```

```
            inOrderTraversal(root->right);
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int key) {
        root = insertRecursive(root, key);
    }

    bool search(int key) {
        return searchRecursive(root, key) != nullptr;
    }

    void inOrderTraversal() {
        inOrderTraversal(root);
        std::cout << std::endl;
    }
};

int main() {
    BinarySearchTree bst;

    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);
```

```
    std::cout << "Inorder traversal: ";
    bst.inOrderTraversal();

    int keyToSearch = 40;
    if (bst.search(keyToSearch)) {
        std::cout << keyToSearch << " found in the BST." << std::endl;
    } else {
        std::cout << keyToSearch << " not found in the BST." << std::endl;
    }

    keyToSearch = 90;
    if (bst.search(keyToSearch)) {
        std::cout << keyToSearch << " found in the BST." << std::endl;
    } else {
        std::cout << keyToSearch << " not found in the BST." << std::endl;
    }

    return 0;
}
```

13. Implement binary search tree and perform following operations:
a) Insert (Handle insertion of duplicate entry)
b) Display - Depth of tree

```
#include <iostream>
#include <algorithm>

class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;
```

```
    TreeNode(int value) {
        data = value;
        left = right = nullptr;
    }
};

class BinarySearchTree {
private:
    TreeNode* root;

    TreeNode* insertRecursive(TreeNode* root, int key) {
        if (root == nullptr) {
            return new TreeNode(key);
        }

        if (key < root->data) {
            root->left = insertRecursive(root->left, key);
        } else if (key > root->data) {
            root->right = insertRecursive(root->right, key);
        } else {
            // Duplicate entry handling (optional)
            std::cout << "Duplicate entry ignored: " << key << std::endl;
        }

        return root;
    }

    int depthRecursive(TreeNode* root) {
        if (root == nullptr) {
            return 0;
```

```
        }

        int leftDepth = depthRecursive(root->left);
        int rightDepth = depthRecursive(root->right);

        return std::max(leftDepth, rightDepth) + 1;
    }

    void inOrderTraversal(TreeNode* root) {
        if (root != nullptr) {
            inOrderTraversal(root->left);
            std::cout << root->data << " ";
            inOrderTraversal(root->right);
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int key) {
        root = insertRecursive(root, key);
    }

    int depth() {
        return depthRecursive(root);
    }

    void display() {
        std::cout << "Inorder traversal: ";
        inOrderTraversal(root);
        std::cout << std::endl;
```

```
    }
};

int main() {
    BinarySearchTree bst;

    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    bst.display();

    int treeDepth = bst.depth();
    std::cout << "Depth of the tree: " << treeDepth << std::endl;

    return 0;
}
```

14. Implement binary search tree and perform following operations:
a) Insert (Handle insertion of duplicate entry)
b) Display - Mirror image

```
#include <iostream>

class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;
```

```
    TreeNode(int value) {
        data = value;
        left = right = nullptr;
    }
};

class BinarySearchTree {
private:
    TreeNode* root;

    TreeNode* insertRecursive(TreeNode* root, int key) {
        if (root == nullptr) {
            return new TreeNode(key);
        }

        if (key < root->data) {
            root->left = insertRecursive(root->left, key);
        } else if (key > root->data) {
            root->right = insertRecursive(root->right, key);
        } else {
            // Duplicate entry handling (optional)
            std::cout << "Duplicate entry ignored: " << key << std::endl;
        }

        return root;
    }

    void mirrorRecursive(TreeNode* root) {
        if (root == nullptr) {
            return;
```

```
        }

        // Swap left and right subtrees
        std::swap(root->left, root->right);

        // Recursively mirror the left and right subtrees
        mirrorRecursive(root->left);
        mirrorRecursive(root->right);
    }

    void inOrderTraversal(TreeNode* root) {
        if (root != nullptr) {
            inOrderTraversal(root->left);
            std::cout << root->data << " ";
            inOrderTraversal(root->right);
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int key) {
        root = insertRecursive(root, key);
    }

    void mirror() {
        mirrorRecursive(root);
    }

    void display() {
        std::cout << "Inorder traversal: ";
```

```
        inOrderTraversal(root);
        std::cout << std::endl;
    }
};

int main() {
    BinarySearchTree bst;

    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    bst.display();

    std::cout << "Mirror image of the tree: ";
    bst.mirror();
    bst.display();

    return 0;
}
```

15. Implement binary search tree and perform following operations:
a) Insert (Handle insertion of duplicate entry)
b) Create a copy

```
#include <iostream>

class TreeNode {
public:
```

```
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) {
        data = value;
        left = right = nullptr;
    }
};

class BinarySearchTree {
private:
    TreeNode* root;

    TreeNode* insertRecursive(TreeNode* root, int key) {
        if (root == nullptr) {
            return new TreeNode(key);
        }

        if (key < root->data) {
            root->left = insertRecursive(root->left, key);
        } else if (key > root->data) {
            root->right = insertRecursive(root->right, key);
        } else {
            // Duplicate entry handling (optional)
            std::cout << "Duplicate entry ignored: " << key << std::endl;
        }

        return root;
    }
```

```
    TreeNode* copyRecursive(TreeNode* root) {
        if (root == nullptr) {
            return nullptr;
        }

        // Create a new node with the same data
        TreeNode* newNode = new TreeNode(root->data);

        // Recursively copy the left and right subtrees
        newNode->left = copyRecursive(root->left);
        newNode->right = copyRecursive(root->right);

        return newNode;
    }

    void inOrderTraversal(TreeNode* root) {
        if (root != nullptr) {
            inOrderTraversal(root->left);
            std::cout << root->data << " ";
            inOrderTraversal(root->right);
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int key) {
        root = insertRecursive(root, key);
    }

    BinarySearchTree* copy() {
```

```
        BinarySearchTree* copiedTree = new BinarySearchTree();
        copiedTree->root = copyRecursive(root);
        return copiedTree;
    }

    void display() {
        std::cout << "Inorder traversal: ";
        inOrderTraversal(root);
        std::cout << std::endl;
    }
};

int main() {
    BinarySearchTree bst;

    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    std::cout << "Original tree: ";
    bst.display();

    BinarySearchTree* copiedTree = bst.copy();
    std::cout << "Copied tree: ";
    copiedTree->display();

    // Clean up allocated memory for copied tree
```

```
    delete copiedTree;


    return 0;
}
```