ASSIGNMENT NO.10

NAME: PAYAL BHAGVAN CHARVANDE

PRN NO: 125B2F002

CLASS: SY DIV: IT A

CODE:

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


// Structure of a Node
struct Node {
   int data;
   Node* left;
   Node* right;
};
// Function to create a new node
Node* createNode(int value) {
   Node* newNode = new Node;
   newNode->data = value;
   newNode->left = newNode->right = NULL;
   return newNode;
}
// Function to insert a node (Handles duplicate)
Node* insert(Node* root, int value) {
   if (root == NULL) {
      return createNode(value);
```

```
    }
    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
    else
        cout << "Duplicate value not allowed!\n";
    return root;
}
// Function to find minimum value node (used in delete)
Node* findMin(Node* root) {
    while (root->left != NULL)
        root = root->left;
    return root;
}
// Function to delete a node
Node* deleteNode(Node* root, int value) {
    if (root == NULL) return root;

    if (value < root->data)
        root->left = deleteNode(root->left, value);
    else if (value > root->data)
        root->right = deleteNode(root->right, value);
    else {
        // Node found
        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 = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
// Search function
bool search(Node* root, int value) {
    if (root == NULL)
        return false;
    if (root->data == value)
        return true;
    else if (value < root->data)
        return search(root->left, value);
    else
        return search(root->right, value);
}
// Inorder traversal
void inorder(Node* root) {
    if (root != NULL) {
```

```
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}
// Preorder traversal
void preorder(Node* root) {
    if (root != NULL) {
        cout << root->data << " ";
        preorder(root->left);
        preorder(root->right);
    }
}
// Postorder traversal
void postorder(Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        cout << root->data << " ";
    }
}
// Function to calculate depth of tree
int depth(Node* root) {
    if (root == NULL)
        return 0;
    int leftDepth = depth(root->left);
    int rightDepth = depth(root->right);
    return (leftDepth > rightDepth ? leftDepth : rightDepth) + 1;
```

```
}
// Function to create mirror image
Node* mirror(Node* root) {
   if (root == NULL)
      return NULL;
   Node* temp = new Node;
   temp->data = root->data;
   temp->left = mirror(root->right);
   temp->right = mirror(root->left);
   return temp;
}
// Function to copy the tree
Node* copyTree(Node* root) {
   if (root == NULL)
      return NULL;
   Node* newNode = new Node;
   newNode->data = root->data;
   newNode->left = copyTree(root->left);
   newNode->right = copyTree(root->right);
   return newNode;
}
// Display all parent nodes with their child nodes
void displayParents(Node* root) {
   if (root == NULL)
      return;
   if (root->left != NULL || root->right != NULL) {
      cout << "Parent: " << root->data;
      if (root->left)
```

```
        cout << " | Left Child: " << root->left->data;
    if (root->right)
        cout << " | Right Child: " << root->right->data;
    cout << endl;
  }
  displayParents(root->left);
  displayParents(root->right);
}
// Display all leaf nodes
void displayLeaves(Node* root) {
  if (root == NULL)
    return;
  if (root->left == NULL && root->right == NULL)
    cout << root->data << " ";
  displayLeaves(root->left);
  displayLeaves(root->right);
}
// Display level wise (simple recursive method)
void printLevel(Node* root, int level) {
  if (root == NULL) return;
  if (level == 1)
    cout << root->data << " ";
  else if (level > 1) {
    printLevel(root->left, level - 1);
    printLevel(root->right, level - 1);
  }
}
void levelOrder(Node* root) {
```

```
    int h = depth(root);
    for (int i = 1; i <= h; i++) {
        printLevel(root, i);
        cout << endl;
    }
}
// Main Function
int main() {
    Node* root = NULL;
    int choice, value;
    do {
        cout << "\n=== Binary Search Tree Operations ===\n";
        cout << "1. Insert\n2. Delete\n3. Search\n4. Display (Inorder, Preorder, Postorder)\n";
        cout << "5. Display Depth\n6. Display Mirror Image\n7. Create Copy\n";
        cout << "8. Display Parent & Child\n9. Display Leaf Nodes\n10. Display Level Wise\n0. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter value to insert: ";
            cin >> value;
            root = insert(root, value);
            break;
        case 2:
            cout << "Enter value to delete: ";
            cin >> value;
```

```
            root = deleteNode(root, value);
            break;
        case 3:
            cout << "Enter value to search: ";
            cin >> value;
            if (search(root, value))
                cout << "Value found!\n";
            else
                cout << "Value not found!\n";
            break;
        case 4:
            cout << "Inorder: ";
            inorder(root);
            cout << "\nPreorder: ";
            preorder(root);
            cout << "\nPostorder: ";
            postorder(root);
            cout << endl;
            break;
        case 5:
            cout << "Depth of tree: " << depth(root) << endl;
            break;
        case 6:
            cout << "Mirror Image (Inorder): ";
            inorder(mirror(root));
            cout << endl;
            break;
        case 7:
```

```
            cout << "Copied Tree (Inorder): ";
            inorder(copyTree(root));
            cout << endl;
            break;
        case 8:
            cout << "Parent and Child Nodes:\n";
            displayParents(root);
            break;
        case 9:
            cout << "Leaf Nodes: ";
            displayLeaves(root);
            cout << endl;
            break;
        case 10:
            cout << "Tree Level Wise:\n";
            levelOrder(root);
            break;
        case 0:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid choice!\n";
        }
    } while (choice != 0);
    return 0;
}
```

OUTPUT:

```
=== Binary Search Tree Operations ===
1. Insert
2. Delete
Enter your choice: 1
Enter value to insert: 40

=== Binary Search Tree Operations ===
1. Insert
2. Delete
Enter value to insert: 40

=== Binary Search Tree Operations ===
1. Insert
2. Delete
=== Binary Search Tree Operations ===
1. Insert
2. Delete
3. Search
4. Display (Inorder, Preorder, Postorder)
5. Display Depth
6. Display Mirror Image
7. Create Copy
8. Display Parent & Child
9. Display Leaf Nodes
10. Display Level Wise
8. Display Parent & Child
9. Display Leaf Nodes
10. Display Level Wise
10. Display Level Wise
0. Exit
Enter your choice: 4
Inorder: 10 20 30 40
Preorder: 10 20 30 40
Postorder: 40 30 20 10
```