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


struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};


class BST {
private:
    Node* root;


    Node* insert(Node* node, int val) {
        if (!node) return new Node(val);
        if (val < node->data)
            node->left = insert(node->left, val);
        else
            node->right = insert(node->right, val);
        return node;
    }


    Node* deleteNode(Node* node, int val) {
        if (!node) return node;
        if (val < node->data)
            node->left = deleteNode(node->left, val);
        else if (val > node->data)
            node->right = deleteNode(node->right, val);
        else {
            if (!node->left) return node->right;
            if (!node->right) return node->left;
            Node* minNode = minValueNode(node->right);
            node->data = minNode->data;
            node->right = deleteNode(node->right, minNode->data);
        }
        return node;
    }


    Node* minValueNode(Node* node) {
        while (node && node->left)
            node = node->left;
        return node;
    }


    bool search(Node* node, int val) {
        if (!node) return false;
        if (node->data == val) return true;
        return val < node->data ? search(node->left, val) : search(node-
>right, val);
    }


    void display(Node* node) {
        if (node) {
            display(node->left);
            cout << node->data << " ";
            display(node->right);
        }
    }


    int depth(Node* node) {
        if (!node) return 0;
        int leftDepth = depth(node->left);
        int rightDepth = depth(node->right);
        return max(leftDepth, rightDepth) + 1;
    }


    void mirror(Node* node) {
        if (node) {
            swap(node->left, node->right);
            mirror(node->left);
            mirror(node->right);
        }
    }


    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;
    }


    void displayLeafNodes(Node* node) {
        if (node) {
            if (!node->left && !node->right) {
                cout << node->data << " ";
            }
            displayLeafNodes(node->left);
            displayLeafNodes(node->right);
        }
    }


    void displayParentNodes(Node* node) {
        if (node) {
            if (node->left || node->right) {
                cout << node->data << " ";
            }
            displayParentNodes(node->left);
            displayParentNodes(node->right);
        }
    }


    void levelOrder(Node* node) {
        if (!node) return;
        queue<Node*> q;
        q.push(node);
        while (!q.empty()) {
            Node* curr = q.front();
            q.pop();
            cout << curr->data << " ";
            if (curr->left) q.push(curr->left);
            if (curr->right) q.push(curr->right);
        }
    }

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

    void insert(int val) {
        root = insert(root, val);
    }

    void deleteNode(int val) {
        root = deleteNode(root, val);
    }

    bool search(int val) {
        return search(root, val);
    }

    void display() {
        display(root);
        cout << endl;
    }

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

    void mirror() {
        mirror(root);
        cout << "Tree mirrored." << endl;
    }

    BST copy() {
        BST newTree;
        newTree.root = copy(root);
        return newTree;
    }

    void displayLeafNodes() {
        displayLeafNodes(root);
        cout << endl;
    }

    void displayParentNodes() {
        displayParentNodes(root);
        cout << endl;
    }

    void levelOrder() {
        levelOrder(root);
        cout << endl;
    }
};

int main() {
    BST tree;
    int baseElements[] = {5, 3, 7, 2, 4, 6, 8};

    for (int val : baseElements) {
        tree.insert(val);
    }

    int choice, value;
    do {
        cout << "\nBinary Search Tree Operations Menu (Given Elements = 
5, 3, 7, 2, 4, 6, 8):\n";
        cout << "1. Insert\n";
        cout << "2. Delete\n";
        cout << "3. Search\n";
        cout << "4. Display (In-order)\n";
        cout << "5. Depth of Tree\n";
        cout << "6. Mirror the Tree\n";
        cout << "7. Create a Copy of the Tree\n";
        cout << "8. Display Leaf Nodes\n";
        cout << "9. Display Parent Nodes\n";
        cout << "10. Level Order Display\n";
        cout << "11. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter value to insert: ";
                cin >> value;
                tree.insert(value);
                break;
            case 2:
                cout << "Enter value to delete: ";
                cin >> value;
                tree.deleteNode(value);
                break;
            case 3:
                cout << "Enter value to search: ";
                cin >> value;
                cout << (tree.search(value) ? "Found" : "Not Found") <<
endl;
                break;
            case 4:
                cout << "In-order display: ";
                tree.display();
                break;
            case 5:
                cout << "Depth of the tree: " << tree.depth() << endl;
                break;
            case 6:
                tree.mirror();
                break;
            case 7: {
                BST copiedTree = tree.copy();
                cout << "Copied tree (In-order): ";
                copiedTree.display();
                break;
            }
            case 8:
                cout << "Leaf nodes: ";
                tree.displayLeafNodes();
                break;
            case 9:
                cout << "Parent nodes: ";
                tree.displayParentNodes();
                break;
            case 10:
                cout << "Level order display: ";
                tree.levelOrder();
                break;
            case 11:
                cout << "Exiting." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    } while (choice != 11);

    return 0;
}
```