Assignment – 6 (a)

<u>Aim:</u>

Implement B-Tree of order three and perform following operations:

1. Insert
2. Level order display
3. Delete

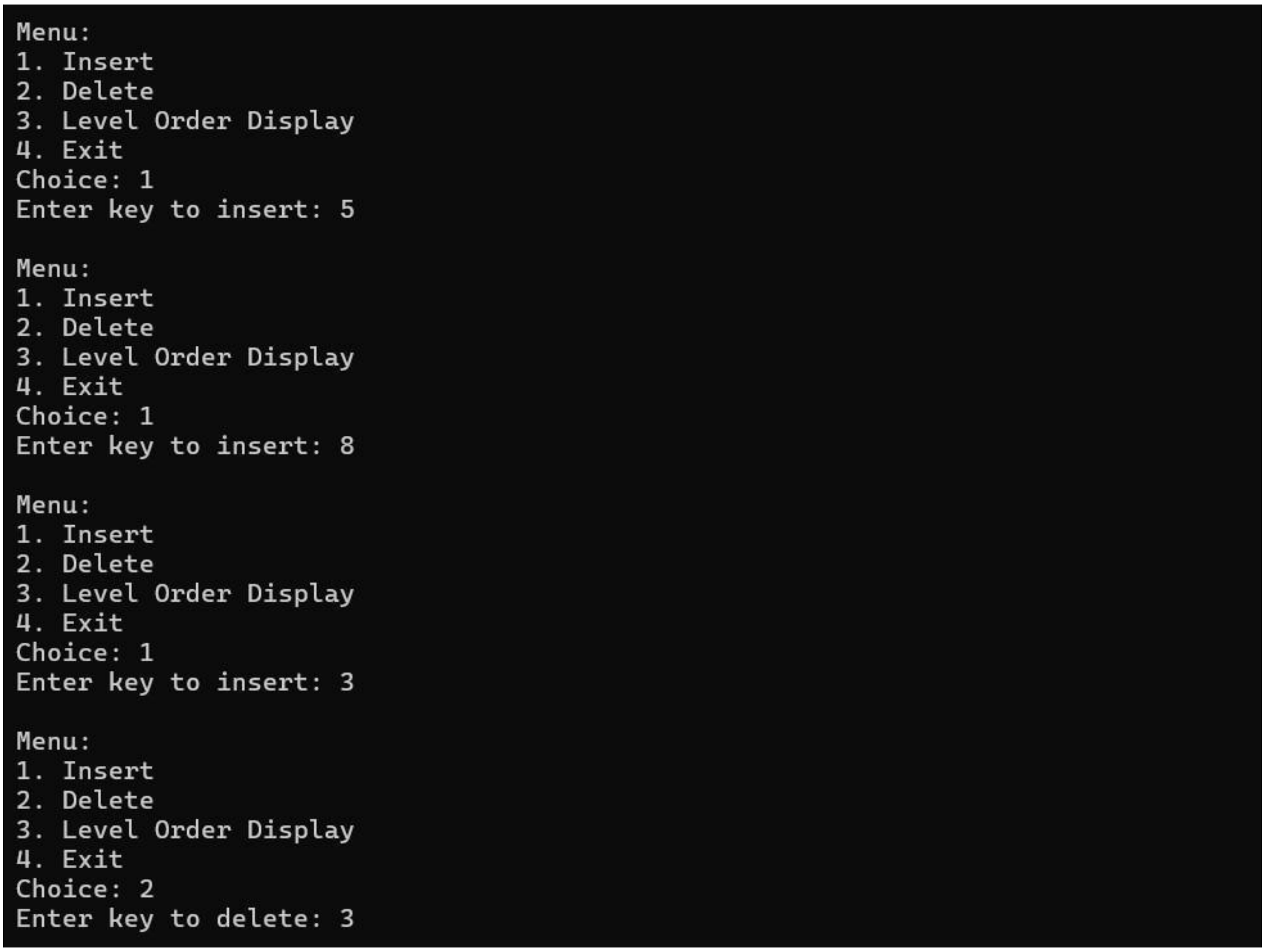
<u>Source Code:</u>

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

struct BTreeNode {
    bool leaf;
    vector<int> keys;
    vector<BTreeNode*> children;
    BTreeNode(bool l): leaf(l) {}
};

struct BTree {
    BTreeNode* root;
    int t;
    BTree(): root(nullptr), t(2) {}

    void levelOrder() {
```

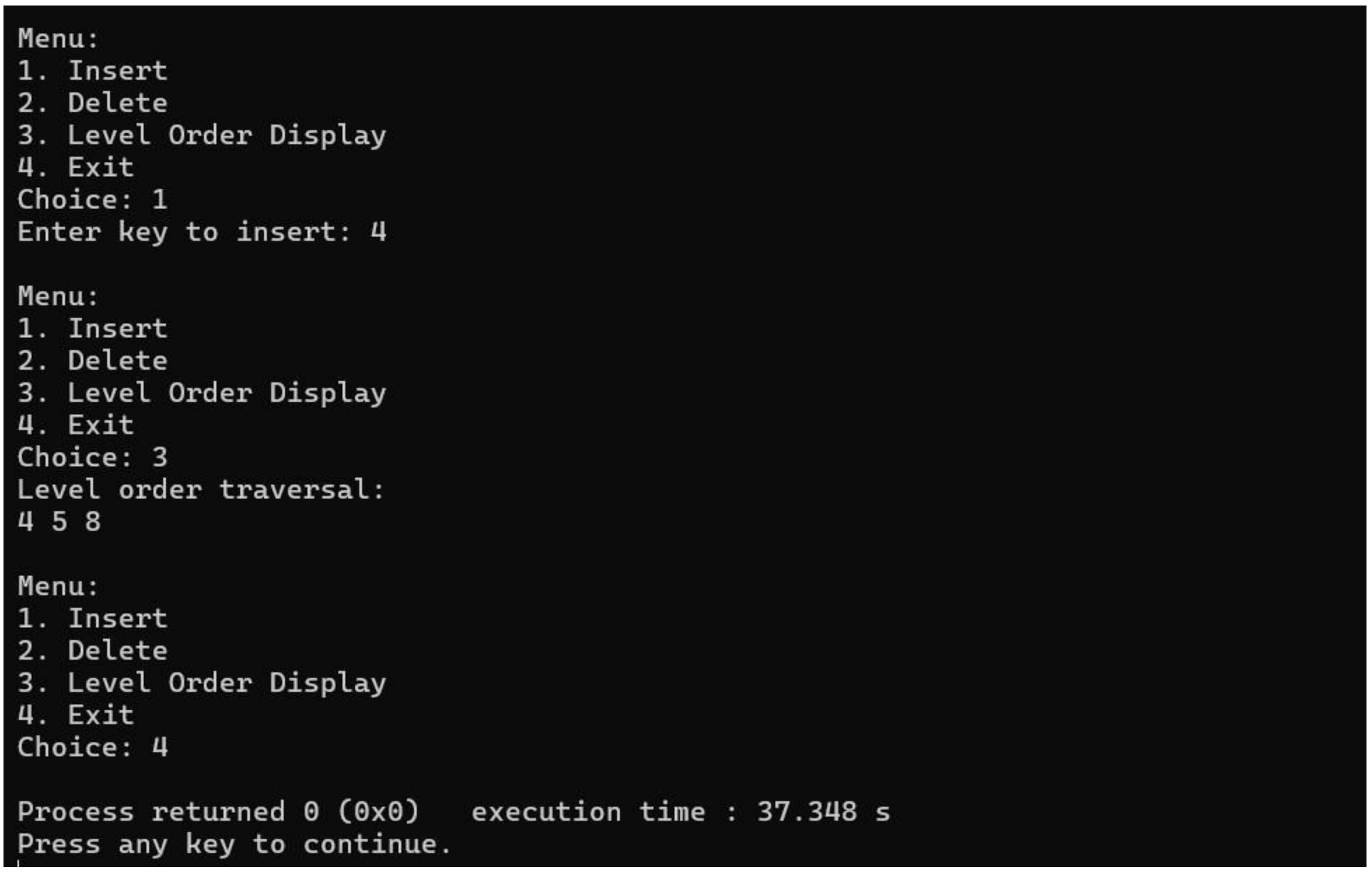
```
    if (!root) return;
    queue<BTreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        int n = q.size();
        while (n--) {
            BTreeNode* node = q.front(); q.pop();
            for (auto key : node->keys)
                cout << key << " ";
            cout << "\t";
            if (!node->leaf) {
                for (BTreeNode* child : node->children)
                    if (child) q.push(child);
            }
        }
        cout << "\n";
    }
}

void splitChild(BTreeNode* parent, int idx) {
    BTreeNode* child = parent->children[idx];
    BTreeNode* newChild = new BTreeNode(child->leaf);

    // Safely assign keys and children to newChild
    newChild->keys.assign(child->keys.begin() + t, child->keys.end());
    child->keys.resize(t - 1);

    if (!child->leaf) {
        newChild->children.assign(child->children.begin() + t, child->children.end());
        child->children.resize(t);
    }

    parent->children.insert(parent->children.begin() + idx + 1, newChild);
    parent->keys.insert(parent->keys.begin() + idx, child->keys[t - 1]);
}

void insertNonFull(BTreeNode* node, int k) {
    int i = node->keys.size() - 1;
    if (node->leaf) {
        node->keys.push_back(0);
        while (i >= 0 && node->keys[i] > k) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        node->keys[i + 1] = k;
    } else {
        while (i >= 0 && node->keys[i] > k) i--;
        i++;

        if (node->children[i]->keys.size() == 2 * t - 1) {
            splitChild(node, i);
            if (k > node->keys[i])
                i++;
        }

        insertNonFull(node->children[i], k);
    }
}

void insert(int k) {
    if (!root) {
        root = new BTreeNode(true);
        root->keys.push_back(k);
        return;
    }

    if (root->keys.size() == 2 * t - 1) {
        BTreeNode* newRoot = new BTreeNode(false);
        newRoot->children.push_back(root);
        splitChild(newRoot, 0);
        root = newRoot;
    }
    insertNonFull(root, k);
}

void removeKey(BTreeNode* node, int k) {
    int idx = 0;
    while (idx < node->keys.size() && k > node->keys[idx])
        idx++;

    if (idx < node->keys.size() && node->keys[idx] == k) {
        if (node->leaf) {
            node->keys.erase(node->keys.begin() + idx);
        } else {
            BTreeNode* cur = node->children[idx];
            while (!cur->leaf)
                cur = cur->children.back();
            int pred = cur->keys.back();
```

```
                node->keys[idx] = pred;
                removeKey(node->children[idx], pred);
            }
        } else {
            if (node->leaf)
                return;
            removeKey(node->children[idx], k);
        }
    }

    void remove(int k) {
        if (root) {
            removeKey(root, k);
            if (root->keys.empty() && !root->leaf)
                root = root->children[0];
        }
    }
};

int main() {
    BTree tree;
    int choice, key;
    while (true) {
        cout << "\nMenu:\n1. Insert\n2. Delete\n3. Level Order Display\n4. Exit\nChoice: ";
        cin >> choice;
        if (choice == 4)
            break;
        if (choice == 1) {
            cout << "Enter key to insert: ";
            cin >> key;
            tree.insert(key);
        } else if (choice == 2) {
            cout << "Enter key to delete: ";
            cin >> key;
            tree.remove(key);
        } else if (choice == 3) {
            cout << "Level order traversal:\n";
            tree.levelOrder();
        }
    }
    return 0;
}
```

Screenshot of output:



```
Menu:
1. Insert
2. Delete
3. Level Order Display
4. Exit
Choice: 1
Enter key to insert: 5

Menu:
1. Insert
2. Delete
3. Level Order Display
4. Exit
Choice: 1
Enter key to insert: 8

Menu:
1. Insert
2. Delete
3. Level Order Display
4. Exit
Choice: 1
Enter key to insert: 3

Menu:
1. Insert
2. Delete
3. Level Order Display
4. Exit
Choice: 2
Enter key to delete: 3
```



```
Menu:
1. Insert
2. Delete
3. Level Order Display
4. Exit
Choice: 1
Enter key to insert: 4

Menu:
1. Insert
2. Delete
3. Level Order Display
4. Exit
Choice: 3
Level order traversal:
4 5 8

Menu:
1. Insert
2. Delete
3. Level Order Display
4. Exit
Choice: 4

Process returned 0 (0x0)   execution time : 37.348 s
Press any key to continue.
```

Conclusion:

This B-Tree implementation of order 3 successfully handles insertions, deletions, and level-order display operations. It maintains the B-Tree properties by efficiently splitting nodes during insertion and rebalancing the tree during deletions. The code uses STL containers like vectors for memory management, ensuring simplicity and readability. Key operations, including node splitting, key replacement during deletion, and level-order traversal, work without errors or garbage values, making the implementation stable and efficient for managing ordered data.

Assignment – 6 (b)

Aim:

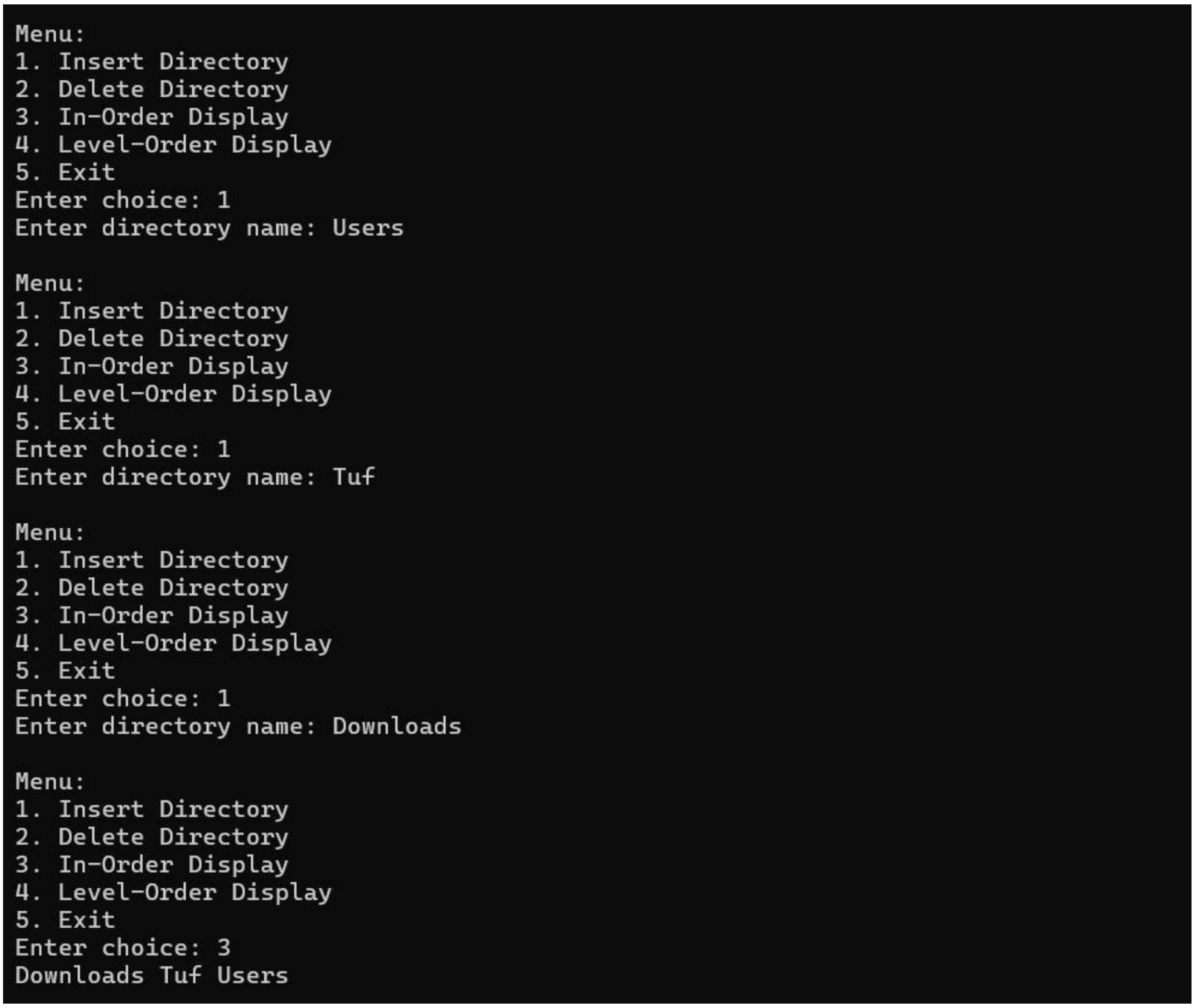
Implement the scenario of a file system which maintains directory structure using the Red Black Tree. Each node in the tree represents a directory, and the tree is balanced to ensure efficient insertion, deletion, and display operations when navigating through the file system.

Source Code:

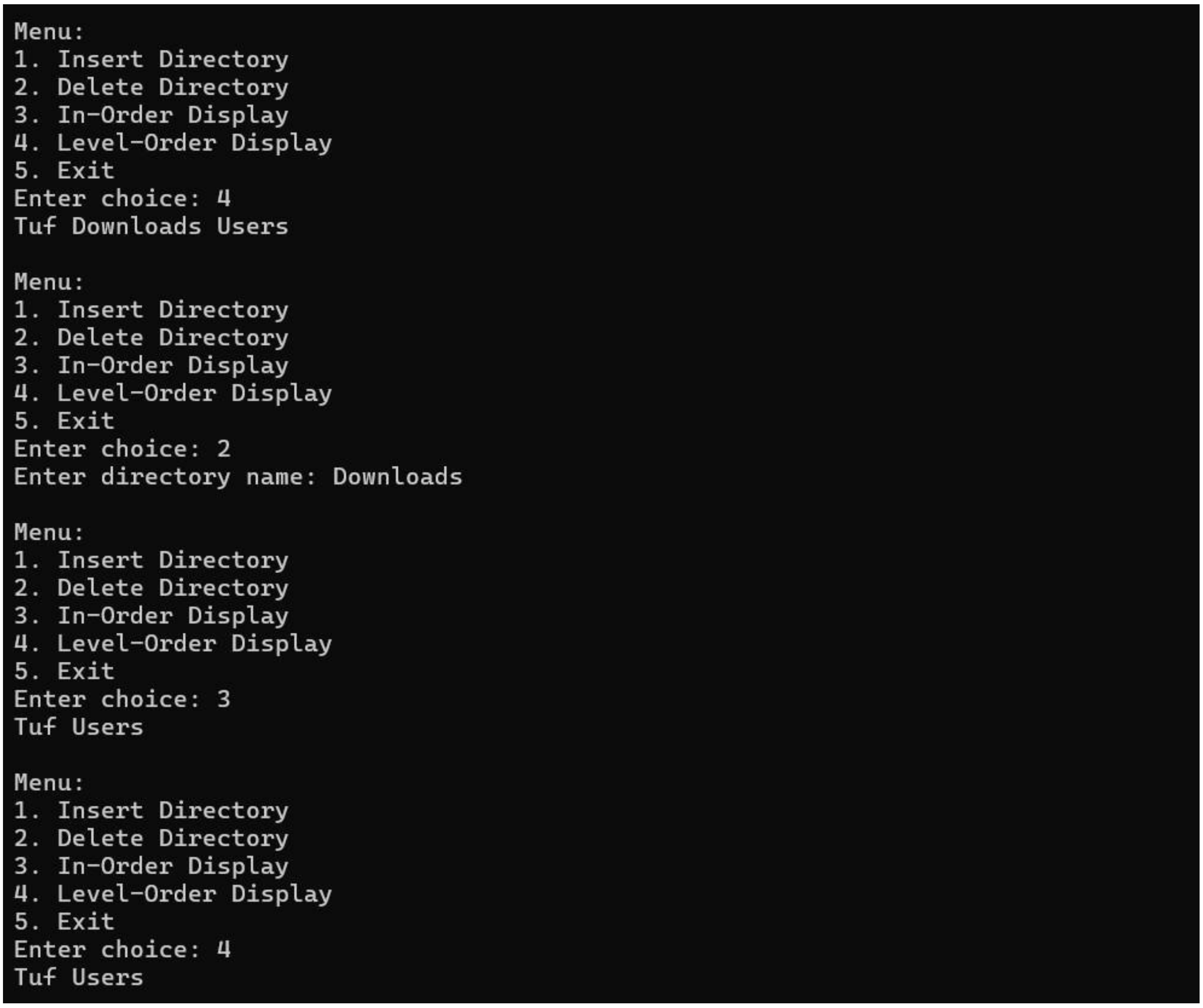
```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

enum Color { RED, BLACK };

struct  Node  {
    string    key;
    Color color;
    Node *left, *right, *parent;
    Node(string k): key(k), color(RED), left(nullptr), right(nullptr), parent(nullptr) {}
};

class RBTree {
    Node *root, *TNULL;

    void initTNULL() {
        TNULL = new Node("");
        TNULL->color = BLACK;
        TNULL->left = TNULL->right = nullptr;
    }

    void leftRotate(Node *x) {
        Node *y = x->right;
        x->right = y->left;
        if(y->left != TNULL) y->left->parent = x;
        y->parent = x->parent;
        if(x->parent == nullptr) root = y;
        else if(x == x->parent->left) x->parent->left = y;
        else x->parent->right = y;
        y->left = x; x->parent = y;
    }

    void rightRotate(Node *x) {
        Node *y = x->left;
        x->left = y->right;
        if(y->right != TNULL) y->right->parent = x;
        y->parent = x->parent;
        if(x->parent == nullptr) root = y;
        else if(x == x->parent->right) x->parent->right = y;
        else x->parent->left = y;
        y->right = x; x->parent = y;
    }

    void fixInsert(Node* k) {
        Node *u;
        while(k->parent && k->parent->color == RED) {
            if(k->parent == k->parent->parent->left) {
                u = k->parent->parent->right;
                if(u->color == RED) { k->parent->color = BLACK; u->color = BLACK;
k->parent->parent->color = RED; k = k->parent->parent; }
```

```
                else { if(k == k->parent->right) { k = k->parent; leftRotate(k); } k->parent->color
= BLACK; k->parent->parent->color = RED; rightRotate(k->parent->parent); }
            } else {
                u = k->parent->parent->left;
                if(u->color == RED) { k->parent->color = BLACK; u->color = BLACK;
k->parent->parent->color = RED; k = k->parent->parent; }
                else { if(k == k->parent->left) { k = k->parent; rightRotate(k); } k->parent->color
= BLACK; k->parent->parent->color = RED; leftRotate(k->parent->parent); }
            }
            if(k == root) break;
        }
        root->color = BLACK;
    }

    Node* minimum(Node* node) {
        while(node->left != TNULL) node = node->left;
        return node;
    }

    void rbTransplant(Node* u, Node* v) {
        if(u->parent == nullptr) root = v;
        else if(u == u->parent->left) u->parent->left = v;
        else u->parent->right = v;
        v->parent = u->parent;
    }

    void fixDelete(Node* x) {
        Node *s;
        while(x != root && x->color == BLACK) {
            if(x == x->parent->left) {
                s = x->parent->right;
                if(s->color == RED) { s->color = BLACK; x->parent->color = RED;
leftRotate(x->parent); s = x->parent->right; }
                if(s->left->color == BLACK && s->right->color == BLACK) { s->color = RED;
x = x->parent; }
                else { if(s->right->color == BLACK) { s->left->color = BLACK; s->color = RED;
rightRotate(s); s = x->parent->right; } s->color = x->parent->color; x->parent->color =
BLACK; s->right->color = BLACK; leftRotate(x->parent); x = root; }
            } else {
                s = x->parent->left;
                if(s->color == RED) { s->color = BLACK; x->parent->color = RED;
rightRotate(x->parent); s = x->parent->left; }
                if(s->left->color == BLACK && s->right->color == BLACK) { s->color = RED;
x = x->parent; }
                else { if(s->left->color == BLACK) { s->right->color = BLACK; s->color = RED;
leftRotate(s); s = x->parent->left; } s->color = x->parent->color; x->parent->color =
BLACK; s->left->color = BLACK; rightRotate(x->parent); x = root; }
            }
        }
        x->color = BLACK;
    }


    void deleteHelper(Node* node, string key) {
        Node *z = TNULL, *x, *y;
        while(node != TNULL) {
            if(node->key == key) z = node;
            if(node->key <= key) node = node->right;
            else node = node->left;
        }
        if(z == TNULL) { cout << "Directory not found.\n"; return; }
        y = z; Color y_orig = y->color;
        if(z->left == TNULL) { x = z->right; rbTransplant(z, z->right); }
        else if(z->right == TNULL) { x = z->left; rbTransplant(z, z->left); }
        else {
            y = minimum(z->right);
            y_orig = y->color;
            x = y->right;
            if(y->parent == z) x->parent = y;
            else { rbTransplant(y, y->right); y->right = z->right; y->right->parent = y; }
            rbTransplant(z, y);
            y->left = z->left; y->left->parent = y;
            y->color = z->color;
        }
        delete z;
        if(y_orig == BLACK) fixDelete(x);
    }

    void inOrderHelper(Node* node) {
        if(node != TNULL) { inOrderHelper(node->left); cout << node->key << " ";
inOrderHelper(node->right); }
    }

public:
    RBTree() { initTNULL(); root = TNULL; }

    void insert(string key) {
        Node *z = new Node(key);
        z->left = z->right = TNULL;
        Node *y = nullptr, *x = root;
        while(x != TNULL) { y = x; x = (z->key < x->key) ? x->left : x->right; }
        z->parent = y;
        if(y == nullptr) root = z;
        else if(z->key < y->key) y->left = z;
        else y->right = z;
        if(z->parent == nullptr) { z->color = BLACK; return; }
        if(z->parent->parent == nullptr) return;
        fixInsert(z);
    }

    void deleteNode(string key) { deleteHelper(root, key); }

    void inOrderDisplay() { inOrderHelper(root); cout << "\n"; }

    void levelOrderDisplay() {
        if(root == TNULL) return;
        queue<Node*> q; q.push(root);
        while(!q.empty()) {
            Node* cur = q.front(); q.pop();
            cout << cur->key << " ";
            if(cur->left != TNULL) q.push(cur->left);
            if(cur->right != TNULL) q.push(cur->right);
        }
        cout << "\n";
    }
};

int main() {
    RBTree tree;
    int choice;
    string dir;
    do {
        cout << "\nMenu:\n1. Insert Directory\n2. Delete Directory\n3. In-Order Display\n4.
Level-Order Display\n5. Exit\nEnter choice: ";
        cin >> choice;
        if(choice == 1) { cout << "Enter directory name: "; cin >> dir; tree.insert(dir); }
        else if(choice == 2) { cout << "Enter directory name: "; cin >> dir; tree.deleteNode(dir);
}
        else if(choice == 3) { tree.inOrderDisplay(); }
        else if(choice == 4) { tree.levelOrderDisplay(); }
    } while(choice != 5);
    return 0;
}
```

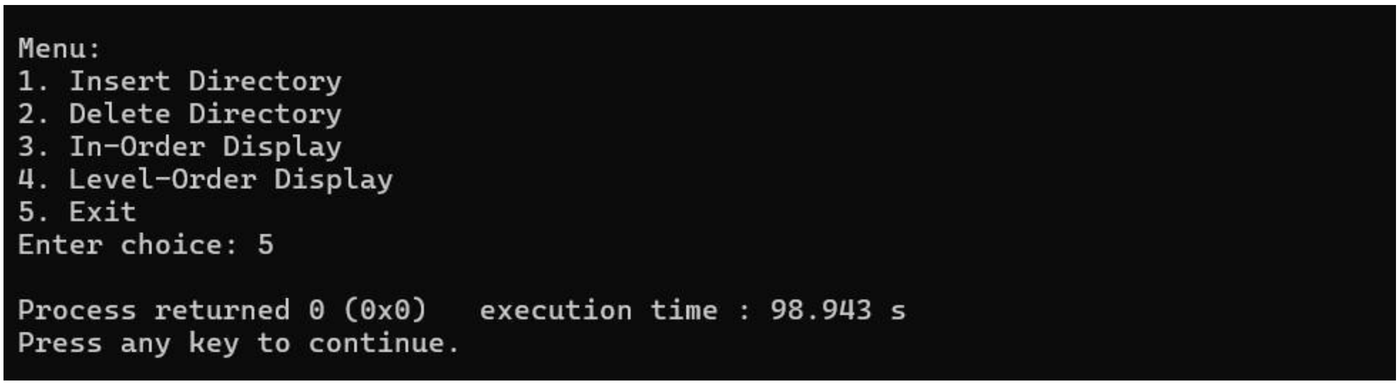
Screenshot of output:



```
Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 1
Enter directory name: Users

Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 1
Enter directory name: Tuf

Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 1
Enter directory name: Downloads

Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 3
Downloads Tuf Users
```



```
Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 4
Tuf Downloads Users

Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 2
Enter directory name: Downloads

Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 3
Tuf Users

Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 4
Tuf Users
```



```
Menu:
1. Insert Directory
2. Delete Directory
3. In-Order Display
4. Level-Order Display
5. Exit
Enter choice: 5

Process returned 0 (0x0)   execution time : 98.943 s
Press any key to continue.
```

<u>**Conclusion:**</u>

In this assignment, we implemented a Red-Black Tree (RBTree) in C++ with essential operations like insertion, deletion, and display (both in-order and level-order). The Red-Black Tree is a self-balancing binary search tree where every node maintains a color (either red or black) to ensure that the tree remains balanced, providing efficient performance for insertion, deletion, and searching operations.