LAB 4

Question 1

Implement an AVL Tree that supports insertion while maintaining the balanced property of the tree. The tree should automatically balance itself using rotations (left, right, left-right, or right-left)

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

struct Node {
    int key;
    Node* left;
    Node* right;
    int height;

    Node(int value) {
        key = value;
        left = right = NULL;
        height = 1;
    }
};

int getHeight(Node* node) {
    if (node == NULL) return 0;
    return max(getHeight(node->left), getHeight(node->right)) + 1;
}

Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = getHeight(y);
    x->height = getHeight(x);

    return x;
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = getHeight(x);
    y->height = getHeight(y);

    return y;
}

Node* insert(Node* root, int key) {
    if (root == NULL) return new Node(key);

    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    else
        return root;

    root->height = getHeight(root);
    int balance = getHeight(root->left) - getHeight(root->right);

```

```cpp
    if (balance > 1 && key < root->left->key)
        return rightRotate(root);

    if (balance < -1 && key > root->right->key)
        return leftRotate(root);

    if (balance > 1 && key > root->left->key) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if (balance < -1 && key < root->right->key) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

int main() {
    Node* root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    cout << "Inorder traversal of balanced AVL tree: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

```
PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB> cd "c:\Users\mol
Insertion in AVL TREE
10 20 25 30 40 50
PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB>
```

2. Implement BFS using both iterative and recursive method.

BFS is a traversal technique that visits all adjacent nodes of a level before moving to the next level. It uses a **queue** and is best for **finding the shortest path and networking** applications.

Iterative:

```cpp
#include <iostream>
#include <queue>
#include <vector>
```

```cpp

using namespace std;

void bfs(vector<vector<int>> &graph, int startNode, vector<bool> &visited) {
    queue<int> q;
    q.push(startNode);
    visited[startNode] = true;

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " ";

        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    int totalNodes = 6;
    vector<vector<int>> graph(totalNodes);

    graph[0] = {1, 2};
    graph[1] = {0, 5, 7};
    graph[2] = {0, 8};
    graph[3] = {5};
    graph[4] = {5};
```

```cpp
    graph[5] = {5};

    vector<bool> visited(totalNodes, false);

    bfs(graph, 0, visited);

    return 0;

}
```

```
PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB> cd "c:\Users
0 1 2 5 7
```

Recursive

```cpp
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

void bfs_recursive(vector<vector<int>> &adj, queue<int> &q, vector<bool> &visited) {
    if (q.empty()) return;

    int node = q.front();
    q.pop();
    cout << node << " ";

    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            visited[neighbor] = true;
            q.push(neighbor);
        }
    }

    bfs_recursive(adj, q, visited);
```

```cpp
}

int main() {
    int n = 6;
    vector<vector<int>> adj(n);
    adj[0] = {1, 2};
    adj[1] = {0, 3, 4};
    adj[2] = {0, 5};
    adj[3] = {1};
    adj[4] = {1};
    adj[5] = {2};

    vector<bool> visited(n, false);
    queue<int> q;
    q.push(0);
    visited[0] = true;

    bfs_recursive(adj, q, visited);

    return 0;
}
```

```
PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB> cd
nerFile }
0 1 2 3 4 5
PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB>
```

DFS

DFS is a traversal technique that explores as deep as possible along a branch before backtracking. It uses a **stack (recursion or explicit)** and is useful for **pathfinding and maze-solving**.

Iterative

```cpp
#include <iostream>
#include <stack>
#include <vector>

using namespace std;

void dfs_iterative(vector<vector<int>> &adj, int start, vector<bool> &visited) {
    stack<int> s;
    s.push(start);

    while (!s.empty()) {
        int node = s.top();
        s.pop();

        if (!visited[node]) {
            visited[node] = true;
            cout << node << " ";

            for (int i = adj[node].size() - 1; i >= 0; i--) {
                if (!visited[adj[node][i]]) {
                    s.push(adj[node][i]);
                }
            }
        }
    }
}

int main() {
    int n = 6;
```

```cpp
    vector<vector<int>> adj(n);
    adj[0] = {1, 2};
    adj[1] = {0, 3, 4};
    adj[2] = {0, 5};
    adj[3] = {5};
    adj[4] = {8};
    adj[5] = {2};

    vector<bool> visited(n, false);
    dfs_iterative(adj, 0, visited);

    return 0;
}
```

```
PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20
nerFile }
0 1 3 5 2 4 8
```

Recursive

```cpp
#include <iostream>
#include <vector>

using namespace std;

void dfs_recursive(vector<vector<int>> &adj, int node, vector<bool> &visited) {
    visited[node] = true;
    cout << node << " ";

    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs_recursive(adj, neighbor, visited);
        }
    }
}

int main() {
    int n = 6;
    vector<vector<int>> adj(n);
    adj[0] = {1, 2};
    adj[1] = {0, 3, 4};
    adj[2] = {0, 5};
    adj[3] = {1};
    adj[4] = {1};
    adj[5] = {2};

    vector<bool> visited(n, false);
    dfs_recursive(adj, 0, visited);

    return 0;
}
```

```
PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB> c
nerFile }
0 1 3 4 2 5
```