

# 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)

```
1. #include <iostream>
2. using namespace std;
3.
4. struct Node {
5.     int key;
6.     Node* left;
7.     Node* right;
8.     int height;
9.
10.    Node(int value) {
11.        key = value;
12.        left = right = NULL;
13.        height = 1;
14.    }
15. };
16.
17. int getHeight(Node* node) {
18.     if (node == NULL) return 0;
19.     return max(getHeight(node->left), getHeight(node->right)) + 1;
20. }
21.
22. Node* rightRotate(Node* y) {
23.     Node* x = y->left;
24.     Node* T2 = x->right;
25.
26.     x->right = y;
27.     y->left = T2;
28.
29.     y->height = getHeight(y);
30.     x->height = getHeight(x);
31.
32.     return x;
33. }
34.
35. Node* leftRotate(Node* x) {
36.     Node* y = x->right;
37.     Node* T2 = y->left;
38.
39.     y->left = x;
40.     x->right = T2;
41.
42.     x->height = getHeight(x);
43.     y->height = getHeight(y);
44.
45.     return y;
46. }
47.
48. Node* insert(Node* root, int key) {
49.     if (root == NULL) return new Node(key);
50.
51.     if (key < root->key)
52.         root->left = insert(root->left, key);
53.     else if (key > root->key)
54.         root->right = insert(root->right, key);
55.     else
56.         return root;
57.
58.     root->height = getHeight(root);
59.     int balance = getHeight(root->left) - getHeight(root->right);
60. }
```

```

61.     if (balance > 1 && key < root->left->key)
62.         return rightRotate(root);
63.
64.     if (balance < -1 && key > root->right->key)
65.         return leftRotate(root);
66.
67.     if (balance > 1 && key > root->left->key) {
68.         root->left = leftRotate(root->left);
69.         return rightRotate(root);
70.     }
71.
72.     if (balance < -1 && key < root->right->key) {
73.         root->right = rightRotate(root->right);
74.         return leftRotate(root);
75.     }
76.
77.     return root;
78. }
79.
80. void inorder(Node* root) {
81.     if (root == NULL) return;
82.     inorder(root->left);
83.     cout << root->key << " ";
84.     inorder(root->right);
85. }
86.
87. int main() {
88.     Node* root = NULL;
89.     root = insert(root, 10);
90.     root = insert(root, 20);
91.     root = insert(root, 30);
92.     root = insert(root, 40);
93.     root = insert(root, 50);
94.     root = insert(root, 25);
95.
96.     cout << "Inorder traversal of balanced AVL tree: ";
97.     inorder(root);
98.     cout << endl;
99.
100.    return 0;
101. }
102.
103.
104.
105.
106.

```

```

PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB> cd "c:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB"
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:

```

1. #include <iostream>
2. #include <queue>
3. #include <vector>

```

```

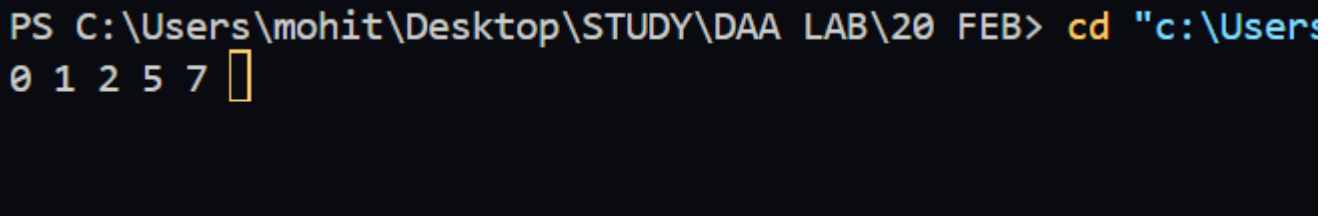
4.
5. using namespace std;
6.
7. void bfs(vector<vector<int>> &graph, int startNode, vector<bool> &visited) {
8.     queue<int> q;
9.     q.push(startNode);
10.    visited[startNode] = true;
11.
12.    while (!q.empty()) {
13.        int current = q.front();
14.        q.pop();
15.        cout << current << " ";
16.
17.        for (int neighbor : graph[current]) {
18.            if (!visited[neighbor]) {
19.                visited[neighbor] = true;
20.                q.push(neighbor);
21.            }
22.        }
23.    }
24. }
25.
26. int main() {
27.     int totalNodes = 6;
28.     vector<vector<int>> graph(totalNodes);
29.
30.     graph[0] = {1, 2};
31.     graph[1] = {0, 5, 7};
32.     graph[2] = {0, 8};
33.     graph[3] = {5};
34.     graph[4] = {5};

```

```

1. 35.     graph[5] = {5};
2. 36.
3. 37.     vector<bool> visited(totalNodes, false);
4. 38.
5.     bfs(graph, 0, visited);
6.
7.     return 0;
8.
}

```



```

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

```

## Recursive

```

1. #include <iostream>
2. #include <queue>
3. #include <vector>
4.
5. using namespace std;
6.
7. void bfs_recursive(vector<vector<int>> &adj, queue<int> &q, vector<bool> &visited) {
8.     if (q.empty()) return;
9.
10.    int node = q.front();
11.    q.pop();
12.    cout << node << " ";
13.
14.    for (int neighbor : adj[node]) {
15.        if (!visited[neighbor]) {
16.            visited[neighbor] = true;
17.            q.push(neighbor);
18.        }
19.    }
20.
21.    bfs_recursive(adj, q, visited);

```

```

22. }
23.
24. int main() {
25.     int n = 6;
26.     vector<vector<int>> adj(n);
27.     adj[0] = {1, 2};
28.     adj[1] = {0, 3, 4};
29.     adj[2] = {0, 5};
30.     adj[3] = {1};
31.     adj[4] = {1};
32.     adj[5] = {2};
33.
34.     vector<bool> visited(n, false);
35.     queue<int> q;
36.     q.push(0);
37.     visited[0] = true;
38.
39.     bfs_recursive(adj, q, visited);
40.
41.     return 0;
42. }
43.
44.
45.

```

```

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

```

1. #include <iostream>
2. #include <stack>
3. #include <vector>
4.
5. using namespace std;
6.
7. void dfs_iterative(vector<vector<int>> &adj, int start, vector<bool> &visited) {
8.     stack<int> s;
9.     s.push(start);
10.
11.     while (!s.empty()) {
12.         int node = s.top();
13.         s.pop();
14.
15.         if (!visited[node]) {
16.             visited[node] = true;
17.             cout << node << " ";
18.
19.             for (int i = adj[node].size() - 1; i >= 0; i--) {
20.                 if (!visited[adj[node][i]]) {
21.                     s.push(adj[node][i]);
22.                 }
23.             }
24.         }
25.     }
26. }
27.
28. int main() {
29.     int n = 6;

```

```

30.     vector<vector<int>> adj(n);
31.     adj[0] = {1, 2};
32.     adj[1] = {0, 3, 4};
33.     adj[2] = {0, 5};
34.     adj[3] = {5};
35.     adj[4] = {8};
36.     adj[5] = {2};
37.
38.     vector<bool> visited(n, false);
39.     dfs_iterative(adj, 0, visited);
40.
41.     return 0;
42. }
43.
44.
45.

```

```

PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB> g++ dfs_iterative.cpp -o dfs_iterative.exe
nerFile }
0 1 3 5 2 4 8

```

## Recursive

```

1. #include <iostream>
2. #include <vector>
3.
4. using namespace std;
5.
6. void dfs_recursive(vector<vector<int>> &adj, int node, vector<bool> &visited) {
7.     visited[node] = true;
8.     cout << node << " ";
9.
10.    for (int neighbor : adj[node]) {
11.        if (!visited[neighbor]) {
12.            dfs_recursive(adj, neighbor, visited);
13.        }
14.    }
15. }
16.
17. int main() {
18.     int n = 6;
19.     vector<vector<int>> adj(n);
20.     adj[0] = {1, 2};
21.     adj[1] = {0, 3, 4};
22.     adj[2] = {0, 5};
23.     adj[3] = {1};
24.     adj[4] = {1};
25.     adj[5] = {2};
26.
27.     vector<bool> visited(n, false);
28.     dfs_recursive(adj, 0, visited);
29.
30.     return 0;
31. }
32.
33.
34.

```

```

PS C:\Users\mohit\Desktop\STUDY\DAA LAB\20 FEB> g++ dfs_recursive.cpp -o dfs_recursive.exe
nerFile }
0 1 3 4 2 5

```