DAY 7

Student Name: Suman Kumar
UID: 22BCS15488
Branch: BE-CSE
Section/Group: 620 - B
Date of Performance:27/12/24

Problem 1

1. Aim: WAP to find the degree of given vertex in the graph.
2. Code:

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

int findDegree(const vector<vector<int>>& graph, int vertex) {
    int degree = 0;
    for (int neighbor : graph[vertex]) {
        degree++;
    }
    return degree;
}
int main() {
    vector<vector<int>> graph = {
        {1, 2},
        {0, 3},
        {0},
        {1}
    };
    int vertex = 1;
    int degree = findDegree(graph, vertex);
    cout << "Degree of vertex " << vertex << " is: " << degree << endl;
    return 0;
}
```

3. Output:

```
Degree of vertex 1 is: 2
```

Problem 2

1. Aim: Write a program for DFS.

2. Code:

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

class Graph {
private:
    int V;
    list<int> *adj;

public:
    Graph(int V);
    void addEdge(int v, int w);
    void DFSUtil(int v, vector<bool>& visited);
    void DFS(int v);
};
Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}
void Graph::addEdge(int v, int w) {
    adj[v].push_back(w); // Add w to v's list
}
void Graph::DFSUtil(int v, vector<bool>& visited) {
    visited[v] = true;
    cout << v << " ";
```

```
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}
void Graph::DFS(int v) {
    vector<bool> visited(V, false);
    DFSUtil(v, visited);
}

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
    cout << "Following is Depth First Traversal (starting from vertex 2) \n";
    g.DFS(2);
    return 0;
}
```

3. Output:

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

Problem 3

1. Aim: WAP to detect a cycle in an unidirected graph.

2. Code:

```
#include <iostream>
```

```
#include <list>
#include <vector>

using namespace std;

class Graph {
private:
    int V;
    list<int> *adj;
public:
    Graph(int V);
    void addEdge(int v, int w);
    bool isCyclicUtil(int v, bool visited[], int parent);
    bool isCyclic();
};
Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}
void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
    adj[w].push_back(v);
}
bool Graph::isCyclicUtil(int v, bool visited[], int parent) {
    visited[v] = true;
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i) {
        if (!visited[*i]) {
            if (isCyclicUtil(*i, visited, v))
                return true;
        }
        else if (*i != parent)
            return true;
```

```
    }
    return false;
}
bool Graph::isCyclic() {
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclicUtil(u, visited, -1))
                return true;
    return false;
}
int main() {
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    if (g1.isCyclic())
        cout << "Graph contains cycle\n";
    else
        cout << "Graph doesn't contain cycle\n";
    Graph g2(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    if (g2.isCyclic())
        cout << "Graph contains cycle\n";
    else
        cout << "Graph doesn't contain cycle\n";
    return 0;
}
```

3. Output:

```
Graph contains cycle
Graph doesn't contain cycle
```

Problem 4

1. Aim: Given the root of complete binary tree return the number of nodes in the tree.

2. Code:

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


struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};
int countNodes(Node* root) {
    if (root == NULL) {
```

```
        return 0;
    } else {
        return 1 + countNodes(root->left) + countNodes(root->right);
    }
}
int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    int nodeCount = countNodes(root);
    cout << "Number of nodes in the tree: " << nodeCount << endl;
    return 0;
}
```

3. Output:

```
Number of nodes in the tree: 6
```

Problem 5

1. Aim: WAP to find the maximum depth of binary tree.

2. Code:

```
#include <iostream>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

int maxDepth(Node* root) {
    if (root == NULL) {
        return 0;
    } else {
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        if (leftDepth > rightDepth) {
            return leftDepth + 1;
        } else {
            return rightDepth + 1;
        }
    }
}
int main() {
```

```
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    int treeDepth = maxDepth(root);
    cout << "Maximum depth of the binary tree: " << treeDepth << endl;
    return 0;
}
```

3. Output:

```
Maximum depth of the binary tree: 3
```

Problem 6

1. Aim: To return the preorder traverse of its nodes value.
2. Code:

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

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};
void preorderTraversal(Node* root, vector<int>& result) {
    if (root == NULL) {
```

```
        return;
    }
    result.push_back(root->data);
    preorderTraversal(root->left, result);
    preorderTraversal(root->right, result);
}
vector<int> preorder(Node* root) {
    vector<int> result;
    preorderTraversal(root, result);
    return result;
}
int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    vector<int> preOrderResult = preorder(root);
    cout << "Preorder traversal: ";
    for (int i = 0; i < preOrderResult.size(); ++i) {
        cout << preOrderResult[i] << " ";
    }
    cout << endl;
    return 0;
}
```

3. Output:

Problem 7

1. Aim: Given the root of binary tree, find the sum of all nodes in the binary tree.
2. Code:

```
#include <iostream>


using namespace std;


struct Node {
    int data;
    Node* left;
    Node* right;
};
Node*newNode(int data) {
    Node* node = new Node;
    node->data = data;
    node->left = nullptr;
    node->right = nullptr;
    return node;
}
int treeSum(Node* root) {
    if (root == nullptr) {
        return 0;
```

```
    }
    return root->data + treeSum(root->left) + treeSum(root->right);
}
int main() {
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(NULL);
    root->right->right = newNode(6);
    int sum = treeSum(root);
    cout << "Sum of all nodes in the binary tree: " << sum << endl;
    return 0;
}
```

3. Output:

```
Sum of all nodes in the binary tree: 21
```

Problem 8

1. Aim: WAP for given a binary tree, the task is to count the leaf of tree. A node is the leaf node of both left and right child is null.
2. Code:

```
#include <iostream>
```

```
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};
int countLeaves(Node* root) {
    if (root == NULL) {
        return 0;
    }
    if (root->left == NULL && root->right == NULL) {
        return 1;
    }
    return countLeaves(root->left) + countLeaves(root->right);
}
int main() {
    Node* root = new Node(1);
```

```
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    int leafCount = countLeaves(root);
    cout << "Number of leaves in the tree: " << leafCount << endl;
    return 0;
}
```

3. Output:

```
Number of leaves in the tree: 3
```

Problem 9

1. Aim: Implementation of Cyclic graph.

2. Code:

```
#include <iostream>
#include <vector>
#include <list>

using namespace std;

class Graph {
private:
    int V;
```

```
    list<int>* adj;
public:
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }
    void printGraph() {
        for (int v = 0; v < V; ++v) {
            cout << "Adjacency list of vertex " << v << ":\n head ";
            for (auto x : adj[v]) {
                cout << "-> " << x;
            }
            cout << endl;
        }
    }
};
int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2, 3);
    g.addEdge(3, 0);
    g.printGraph();
    return 0;
}
```

3. Output:

```
 head -> 3
Adjacency list of vertex 3:
 head -> 0
```

Problem 10

1. Aim: WAP to find the center of Star graph.
2. Code:

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

using namespace std;

int findCenter(vector<vector<int>>& edges) {
    return edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1] ?
edges[0][0] : edges[0][1];
}

int main() {
    vector<vector<int>> edges = {{1, 2}, {2, 3}, {4, 2}};
    int center = findCenter(edges);
    cout << "Center of the star graph: " << center << endl;
    return 0;
}
```

3. Output:

```
Center of the star graph: 2
```

Problem 11

1. Aim: WAP to detect the cycle in the directed graph by using DFS.

2. Code:

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;
class Graph {
private:
    int V; // Number of vertices
    list<int>* adj; // Adjacency list
public:
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }
    bool isCyclicUtil(int v, vector<bool>& visited, vector<bool>&
recStack) {
```

```
        if (recStack[v])
            return true;
        if (!visited[v]) {
            visited[v] = true;
            recStack[v] = true;
            list<int>::iterator i;
            for (i = adj[v].begin(); i != adj[v].end(); ++i)
                if (isCyclicUtil(*i, visited, recStack))
                    return true;
        }
        recStack[v] = false;
        return false;
    }
    bool isCyclic() {
        vector<bool> visited(V, false);
        vector<bool> recStack(V, false);
        for (int i = 0; i < V; i++)
            if (isCyclicUtil(i, visited, recStack))
                return true;
        return false;
    }
};
int main() {
```

```
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
    if (g.isCyclic())
        cout << "Graph contains cycle\n";
    else
        cout << "Graph doesn't contain cycle\n";
    return 0;
}
```

3. Output:

```
Graph contains cycle
```