***NAME : Himanshu Dixit***

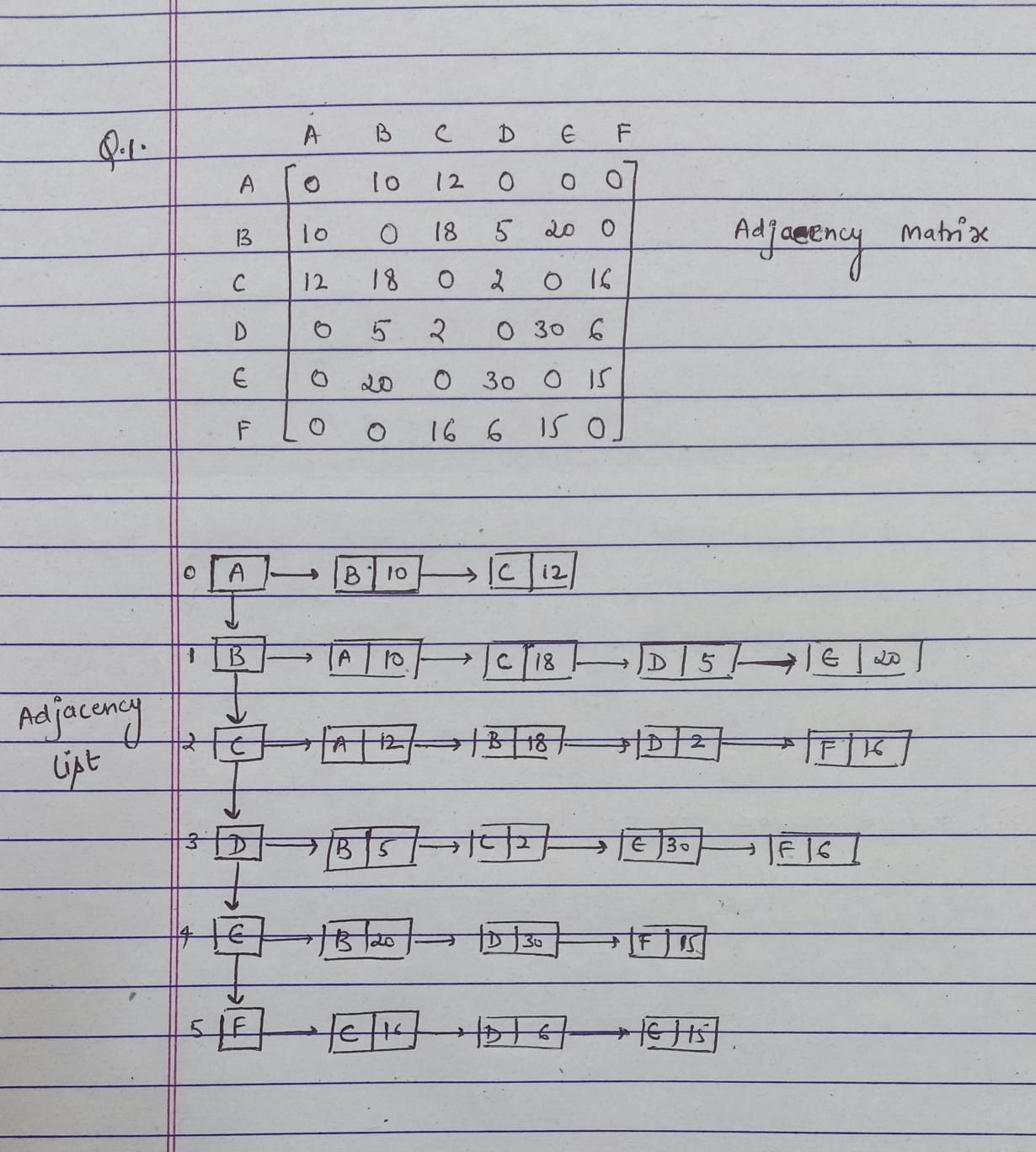
***ENROLL NO. : 21103262***

***BATCH : B11***

***Data Structure [15B11CI311]***

***Tutorial Sheet 14***

**Solution 1 :**



**Solution 2 :**

#include<bits/stdc++.h>

using namespace std;

int minEdgeBFS(vector <int> edges[], int u, int v, int n)

{

vector<bool> visited(n, 0);

vector<int> distance(n, 0);

queue <int> Q;

distance[u] = 0;

Q.push(u);

visited[u] = true;

while (!Q.empty())

{

int x = Q.front();

Q.pop();

for (int i=0; i<edges[x].size(); i++)

{

if (visited[edges[x][i]])

continue;

distance[edges[x][i]] = distance[x] + 1;

Q.push(edges[x][i]);

visited[edges[x][i]] = 1;

}

}

return distance[v];

}

void addEdge(vector <int> edges[], int u, int v)

{

edges[u].push\_back(v);

edges[v].push\_back(u);

}

int main()

{

int n = 9;

vector <int> edges[9];

addEdge(edges, 0, 1);

addEdge(edges, 0, 7);

addEdge(edges, 1, 7);

addEdge(edges, 1, 2);

addEdge(edges, 2, 3);

addEdge(edges, 2, 5);

addEdge(edges, 2, 8);

addEdge(edges, 3, 4);

addEdge(edges, 3, 5);

addEdge(edges, 4, 5);

addEdge(edges, 5, 6);

addEdge(edges, 6, 7);

addEdge(edges, 7, 8);

int u = 0;

int v = 5;

cout << minEdgeBFS(edges, u, v, n);

return 0;

}

Queue and array is used for performing this task..

**Solution 3 :**

#include <iostream>

#include <list>

using namespace std;

class Graph {

int V;

list<int>\* adj;

void printAllPathsUtil(int, int, bool[], int[], int&);

public:

Graph(int V);

void addEdge(int u, int v);

void printAllPaths(int s, int d);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int u, int v)

{

adj[u].push\_back(v); // Add v to u’s list.

}

void Graph::printAllPaths(int s, int d)

{

bool\* visited = new bool[V];

int\* path = new int[V];

int path\_index = 0; // Initialize path[] as empty

for (int i = 0; i < V; i++)

visited[i] = false;

printAllPathsUtil(s, d, visited, path, path\_index);

}

void Graph::printAllPathsUtil(int u, int d, bool visited[],int path[], int& path\_index)

{

visited[u] = true;

path[path\_index] = u;

path\_index++;

if (u == d) {

for (int i = 0; i < path\_index; i++)

cout << path[i] << " ";

cout << endl;

}

else

{

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

if (!visited[\*i])

printAllPathsUtil(\*i, d, visited, path,

path\_index);

}

path\_index--;

visited[u] = false;

}

int main()

{

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(0, 3);

g.addEdge(2, 0);

g.addEdge(2, 1);

g.addEdge(1, 3);

int s = 2, d = 3;

cout << "Following are all different paths from " << s

<< " to " << d << endl;

g.printAllPaths(s, d);

return 0;

}

Graph is used to perform this task..

**Solution 4 :**

#include <bits/stdc++.h>

using namespace std;

void add\_edge(vector<int> adj[], int src, int dest)

{

adj[src].push\_back(dest);

adj[dest].push\_back(src);

}

bool BFS(vector<int> adj[], int src, int dest, int v, int pred[], int dist[])

{

list<int> queue;

bool visited[v];

for (int i = 0; i < v; i++) {

visited[i] = false;

dist[i] = INT\_MAX;

pred[i] = -1;

}

visited[src] = true;

dist[src] = 0;

queue.push\_back(src);

while (!queue.empty()) {

int u = queue.front();

queue.pop\_front();

for (int i = 0; i < adj[u].size(); i++) {

if (visited[adj[u][i]] == false) {

visited[adj[u][i]] = true;

dist[adj[u][i]] = dist[u] + 1;

pred[adj[u][i]] = u;

queue.push\_back(adj[u][i]);

if (adj[u][i] == dest)

return true;

}

}

}

return false;

}

void printShortestDistance(vector<int> adj[], int s, int dest, int v)

{

int pred[v], dist[v];

if (BFS(adj, s, dest, v, pred, dist) == false) {

cout << "Given source and destination"

<< " are not connected";

return;

}

vector<int> path;

int crawl = dest;

path.push\_back(crawl);

while (pred[crawl] != -1) {

path.push\_back(pred[crawl]);

crawl = pred[crawl];

}

cout << "Shortest path length is : "

<< dist[dest];

cout << "\nPath is::\n";

for (int i = path.size() - 1; i >= 0; i--)

cout << path[i] << " ";

}

int main()

{

int v = 8;

vector<int> adj[v];

add\_edge(adj, 0, 1);

add\_edge(adj, 0, 3);

add\_edge(adj, 1, 2);

add\_edge(adj, 3, 4);

add\_edge(adj, 3, 7);

add\_edge(adj, 4, 5);

add\_edge(adj, 4, 6);

add\_edge(adj, 4, 7);

add\_edge(adj, 5, 6);

add\_edge(adj, 6, 7);

int source = 0, dest = 7;

printShortestDistance(adj, source, dest, v);

return 0;

}

Graph DFS is used to perform this task..

**Solution 5 :**

Kruskal’s Algorithm

#include <bits/stdc++.h>

using namespace std;

class DSU {

int\* parent;

int\* rank;

public:

DSU(int n)

{

parent = new int[n];

rank = new int[n];

for (int i = 0; i < n; i++) {

parent[i] = -1;

rank[i] = 1;

}

}

int find(int i)

{

if (parent[i] == -1)

return i;

return parent[i] = find(parent[i]);

}

void unite(int x, int y)

{

int s1 = find(x);

int s2 = find(y);

if (s1 != s2) {

if (rank[s1] < rank[s2]) {

parent[s1] = s2;

}

else if (rank[s1] > rank[s2]) {

parent[s2] = s1;

}

else {

parent[s2] = s1;

rank[s1] += 1;

}

}

}

};

class Graph {

vector<vector<int> > edgelist;

int V;

public:

Graph(int V) { this->V = V; }

void addEdge(int x, int y, int w)

{

edgelist.push\_back({ w, x, y });

}

void kruskals\_mst()

{

sort(edgelist.begin(), edgelist.end());

DSU s(V);

int ans = 0;

cout << "Following are the edges in the "

"constructed MST"

<< endl;

for (auto edge : edgelist) {

int w = edge[0];

int x = edge[1];

int y = edge[2];

if (s.find(x) != s.find(y)) {

s.unite(x, y);

ans += w;

cout << x << " -- " << y << " == " << w

<< endl;

}

}

cout << "Minimum Cost Spanning Tree: " << ans;

}

};

int main()

{

Graph g(4);

g.addEdge(0, 1, 10);

g.addEdge(1, 3, 15);

g.addEdge(2, 3, 4);

g.addEdge(2, 0, 6);

g.addEdge(0, 3, 5);

g.kruskals\_mst();

return 0;

}

Prim’s Algorithm

#include <bits/stdc++.h>

using namespace std;

#define V 5

int minKey(int key[], bool mstSet[])

{

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

void printMST(int parent[], int graph[V][V])

{

cout << "Edge \tWeight\n";

for (int i = 1; i < V; i++)

cout << parent[i] << " - " << i << " \t"

<< graph[i][parent[i]] << " \n";

}

void primMST(int graph[V][V])

{

int parent[V];

int key[V];

bool mstSet[V];

for (int i = 0; i < V; i++)

key[i] = INT\_MAX, mstSet[i] = false;

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = true;

for (int v = 0; v < V; v++)

if (graph[u][v] && mstSet[v] == false

&& graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, graph);

}

int main()

{

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 } };

primMST(graph);

return 0;

}