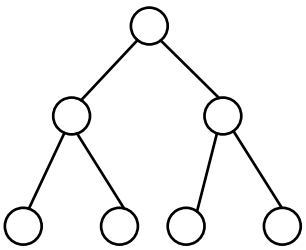
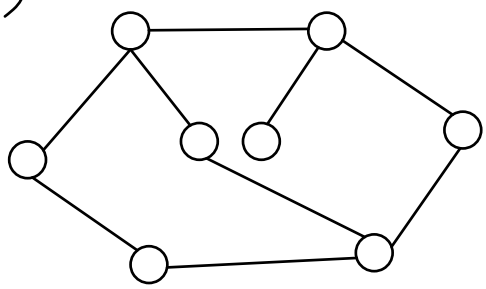
Graph : Bunch of nodes, connected via edges.

1)



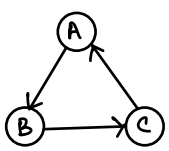
2)



Tree :- Hierarchical DS unlike Graphs.

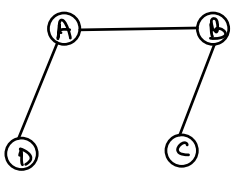
N nodes → (N-1) edges ⇒ Tree.

1)



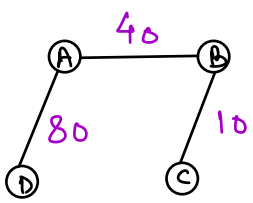
Directed graph.

2)



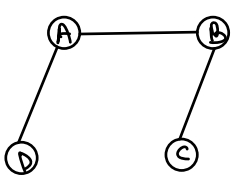
Undirected graph.

3)



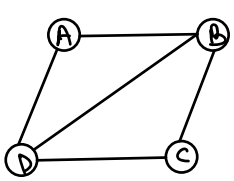
Weighted graph

4)



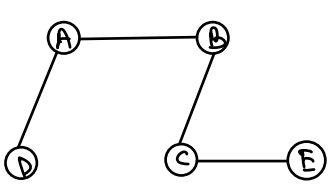
Unweighted graph

5)



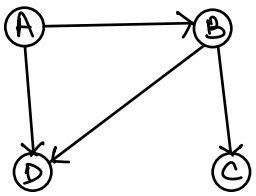
Undirected Cyclic Graph

6)



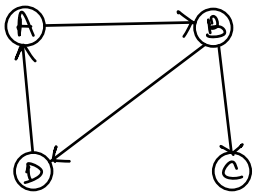
ACyclic Undirected Unweighted.

7)



Directed acyclic graph

8)



Directed cyclic graph

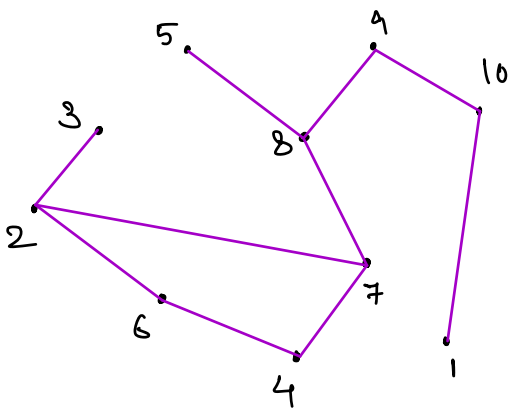
How to store a graph in code?

* Undirected Graph

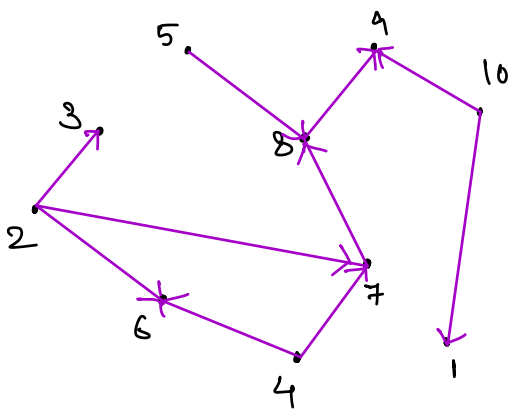
# of Nodes (N), # of Edges (E)

N = 10, E = 10

u	v
2	3
4	7
8	9
2	7
7	8
10	1
4	6
5	8
2	6
10	9



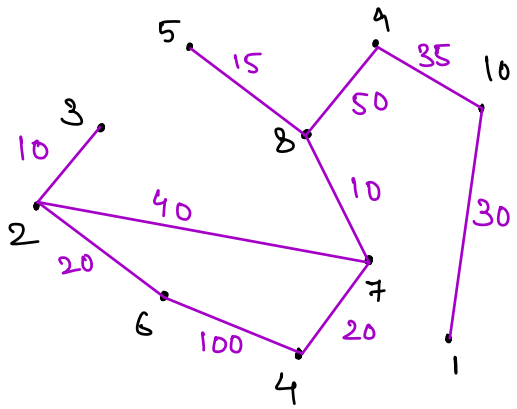
* Directed Graph



N = 10, E = 10

u	v	wt
2	3	10
4	7	20
8	9	50
2	7	40
7	8	10
10	1	30
4	6	100
5	8	15
2	6	20
10	9	35

Undirected Weighted graph.

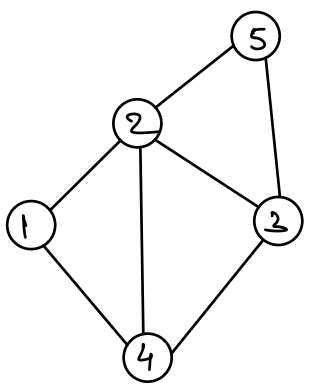


* 27 (Adjacency) Matrix

N	E
5	7
1	4
2	5
3	2
4	3
2	4
3	5
1	2

Undirected

u: int []
v: int []



int mat[6][6] ⇒ 1 based index.

	0	1	2	3	4	5
0	x	x	x	x	x	x
1	x	0	1	0	1	0
2	x	1	0	1	1	1
3	x	0	1	0	1	1
4	x	1	1	1	0	0
5	x	0	1	1	0	0

* `int mat[N+1][N+1] = {0}`

	Unweighted	Weighted
Undirected	`mat[u][v] = 1` `mat[v][u] = 1`	`mat[u][v] = w` `mat[v][u] = w`
Directed	`mat[u][v] = 1`	`mat[u][v] = w`

# of nodes = N

# of Edges = E

TC: O(E)

SC: $O(N^2)$ ⇒ Huge Space Wastage

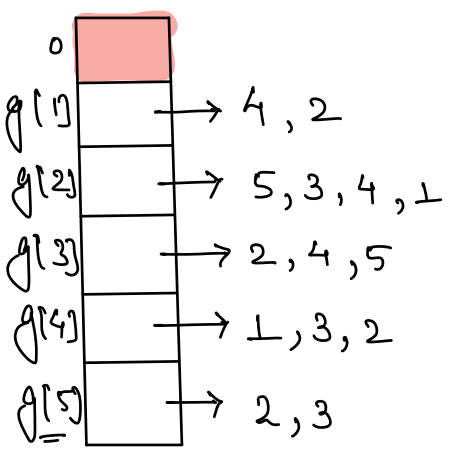
N = 1000, E = 5000 ⇒ Size of matrix = 10^6

Idea 2:- Adj. List

vector<int> in C++

```
list<int> g[6]; // Array of list.
                ↑ N+1
{ list< list<int> > g; }
```

N	E
5	7
1	4
2	5
3	2
4	3
2	4
3	5
1	2 ←



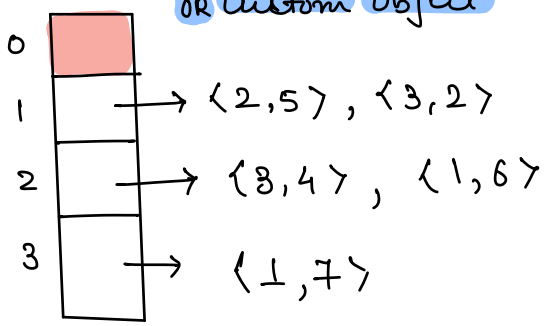
N E (Directed)

3 5

u	v	w
1	2	5
1	3	2
2	3	4
2	1	6
3	1	7

List<pair<int, int>> g[4]

OR Custom Object



u v w

	Unweighted	Weighted
Undirected	g[u].add(v) g[v].add(u)	g[u].add({v,w}) g[v].add({u,w})
Directed	g[u].add(v)	g[u].add({v,w})

TC : O(E)

SC : O(E)

Total Space :- g[0] + g[1] + g[2] + ... g[N]

g[0] ↓ 0

g[1] + g[2] + ... g[N] → 2E | E

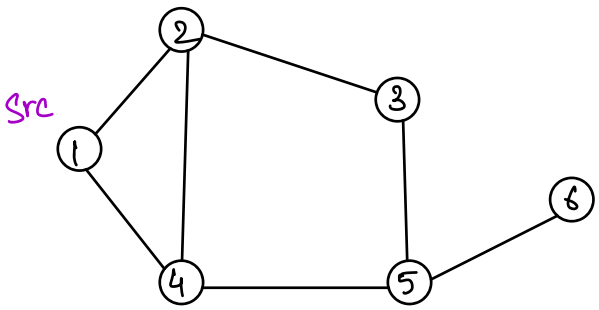
∴ 2E | E

⇒ O(E)

Unweighted graph: `list<int> g[N+1];`

Weighted graph:- `list<pair<int, int>> g[N+1]`

Q. Given an undirected graph, a source node & a destination node. Check if destination node can be visited from source node.



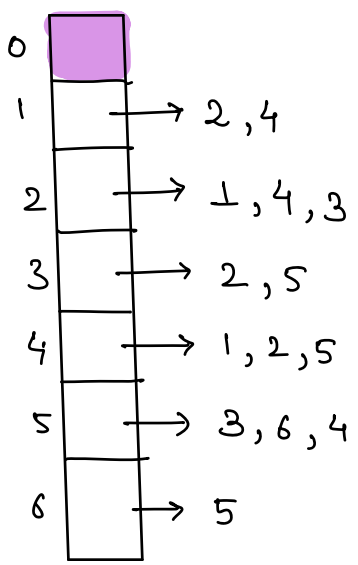
S = 1, D = 6

→ True

N = 6, E = 7

u[]	v[]
1	2
1	4
2	4
2	3
3	5
5	6
4	5

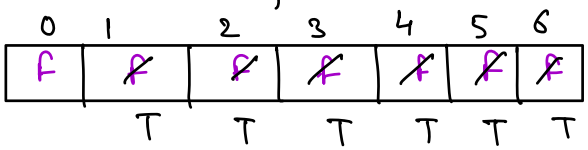
list<int> g[7]



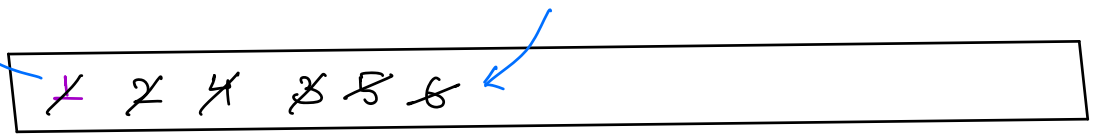
1, 2, 4, 1, 3, 4, 1, 2, 5

→ We are reaching the same node again

* bool visited[7] = {false};



DS



Queue < Delete from front
inserting from rear.

Idea :-

Step 1 :- Insert source node in the Queue & mark it as visited.

Step 2 :- Get the front node from the Queue & remove it.

Step 3 Go to adjacency list of node, and add all unvisited neighbours in the Queue & mark them as visited.

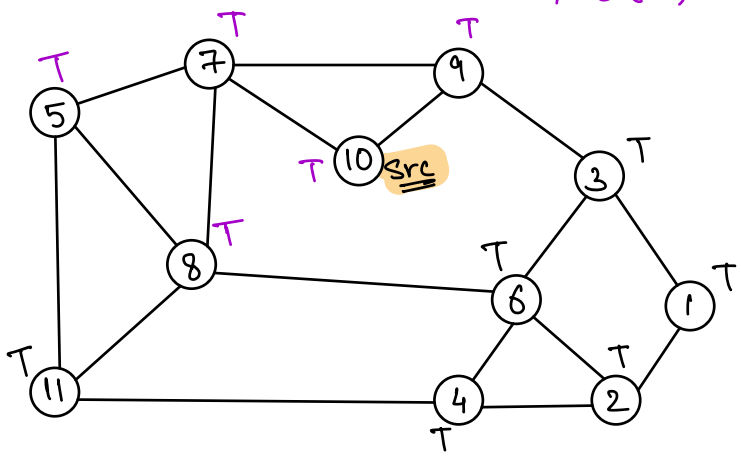
```
bool pathExist(N, E, u[], v[], src, dest){
List<int> g[N+1];
for(i = 0; i < E ; i++){
    // u[i], v[i] => Edge.
    g[u[i]].add(v[i]);
    g[v[i]].add(u[i]);
}
Queue<int> q;
q.insert(src);
bool vis[N+1] = {false}; => SC: O(N)
vis[src] = true;
int level[N+1] = {-1}; level[src] = 0
int par[N+1] = {-1}; par[src] = -1;

while( !Q.isEmpty() ){
    int cu = q.front();
    q.dequeue();
    // Adj. list of node cu => g[cu]
    for(i = 0; i < g[cu].size(); i++){
        cv = g[cu][i]
        if( !vis[cv] ){
            level[cv] = level[cu] + 1
            par[cv] = cu;
            q.enqueue(cv);
            vis[cv] = true;
        }
    }
}
return vis[dest];
}
```

TC: O(E)
SC: O(E)

TC: O(E)

TC : O(E)

SC : O(N+E) { if E >> N ⇒ O(E) }



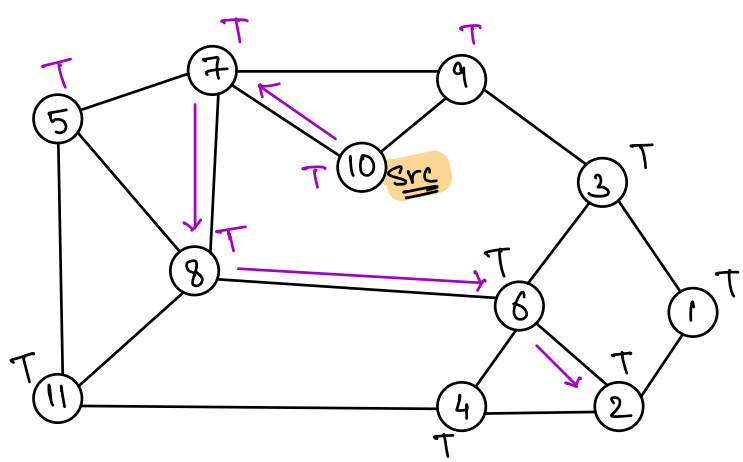
S = 10 , D = 2

Q

0	1	2	3	4
~~10~~	~~7~~ ~~9~~	~~5~~ ~~8~~ ~~3~~	~~11~~ ~~6~~ ~~1~~	~~4~~ ~~2~~

⇒ This traversal gives us the length of shortest path from source to all the Node.

Level	0	1	2	3	4	5	6	7	8	9	10	11
	-1	3	4	2	4	2	3	1	2	1	0	3



~~10~~, ~~7~~, ~~9~~, ~~5~~, ~~8~~, ~~3~~, ~~11~~, ~~6~~, ~~1~~, ~~4~~, ~~2~~

Par[12] :

0	1	2	3	4	5	6	7	8	9	10	11
-1	3	6	9	11	7	8	10	7	10	-1	5	x

Dest

2 → Par[2] → Par[6] → Par[8] → Par[7] → Par[10]

← 6 ← 8 ← 7 ← 10 -1
