Graph -> I

Edge -> Directed

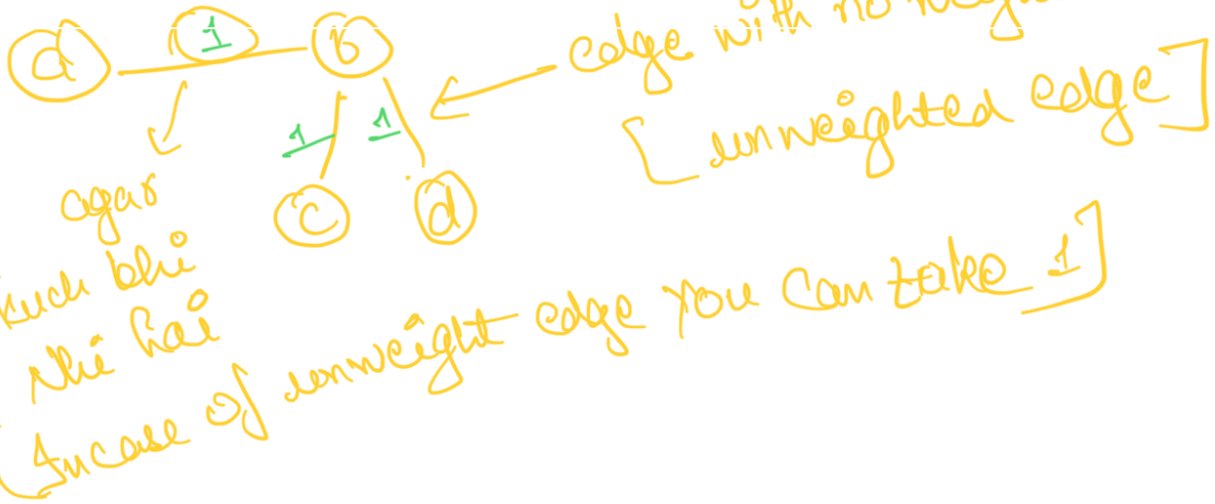
Undirected ->

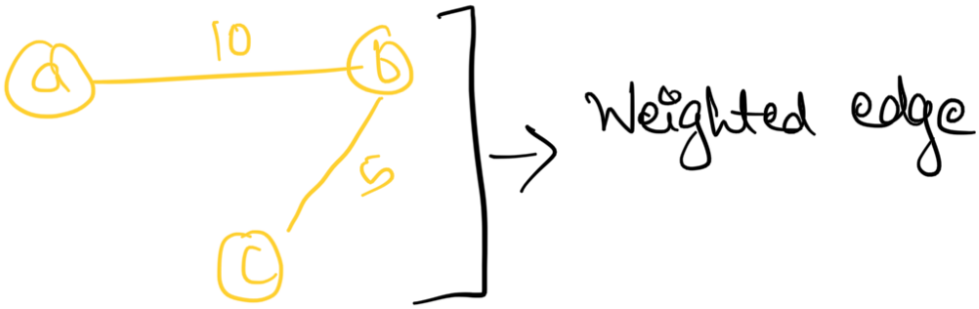
edge list

a-b b-c
b-a c-b
a-d
d-a
d-b
b-d

Convert undirected edge into Directed

Practical application
- ↳ google maps
- ↳ facebook





Cycles ->

Acyclic Graph

Degree →

↓

A Node/Vertex if connected to how many other Nodes/vertex.

degree (a) → 3
degree (b) → 2
degree (d) → 2
degree (e) → 3

In case of unweighted

In case of Weighted Graph

Indegree (a) = 0

Outdegree (a) = 3

Indegree (c) = 3

Outdegree (c) = 0

Graph
-> edge
-> Node
-> undirected
-> Directed
-> weighted
-> unweighted
-> Degree
-> Indegree

→ Outdegree

→ cycle/Acycle

Path :-

a - b - c - d → ✓

a - b - c - e - a → ✗ (Path cannot have cycle)

Components →

-> This is also a graph

(c is connected graph)

Note -> Every Tree is a graph but not vice-versa

Impt -> (1) -> Clone a graph

→ Adjancy Matrix

→ Adjancy List

① → Adjancy Matrix

↓

n x n

no of Node → 4

	0	1	2	3
0	0	1	0	0
1	0	0	1	1
2	0	0	0	1
3	0	0		0

edge list

0 - 1

1 → 2

2 → 3

1 → 3

```
void solve (vector<pair<int, int>> edgeList)
{
    int n = edgeList.size

    vector <vector<int>> adj (n, vector<int
                                  n,0));
```

```
for(auto i: edgeList){
    int u = i.first;
    int v = i.second;
    adj[u][v] = 1;
}
```

i -> pair<int, int>

u v

u ⟶ v

SC -> $O(n^2)$
PC -> $O(n^2)$

2 -> Adjancy List -> Neighbours

0 -> 1

0: { 1 }

1: { 2, 3 }

2: {3}
3: {4}
4: {5}
5: {3}

Adjacency List D.S

→ Map<int, list<int>>

list is like a vector

a → b
b → c
b → d

map <int, List<int>>

↓ Src

↓ Neighbour

Weighted →

map <int, List<pair<int, int>>>

↓ Src

↓ Neighbour

↓ weight

0 : { {1, 5}, {2, 12} }

↓ weight

↓ Neighbour

Graph Traversal

↓

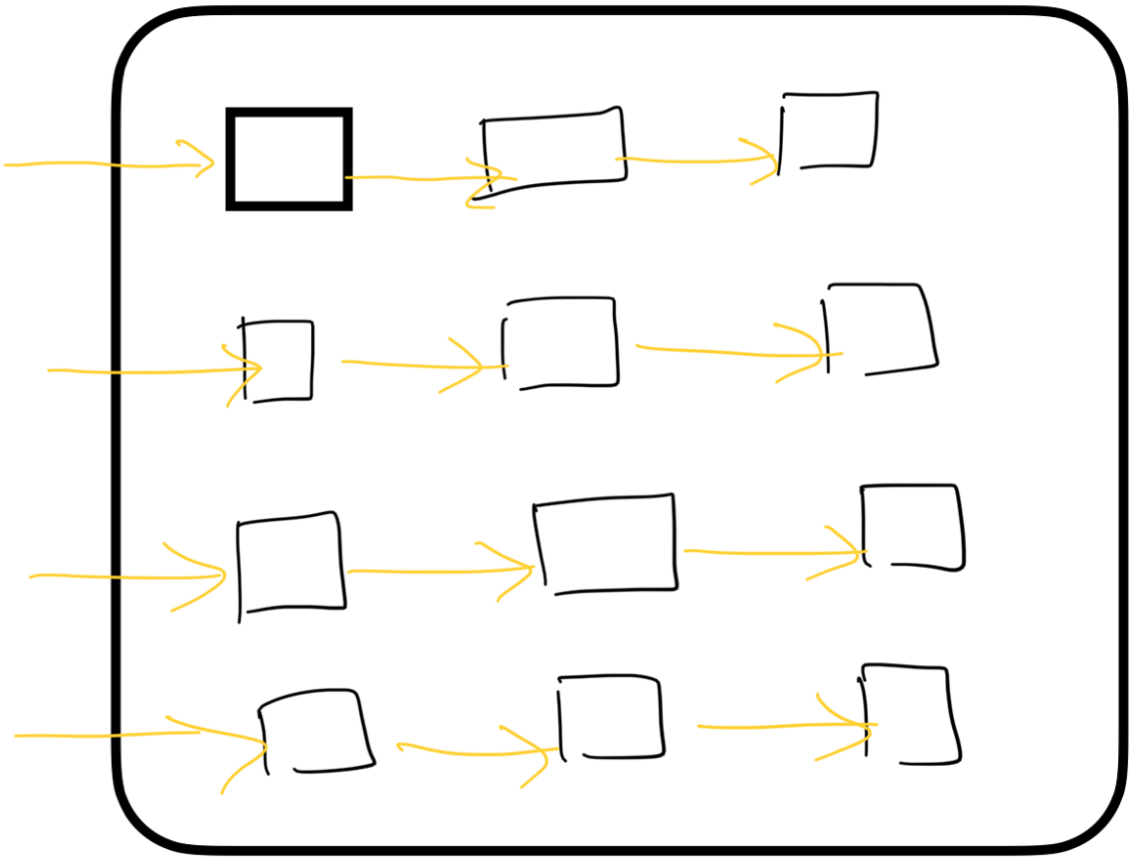
↓

BFS

DFS

1 → BFS (Breadth first Search)

↳ Queue (LIFO)



a

b c

d e f

track -> a.s (T/F)

node -> visited or not

visited

a -> F T

b -> F T

c -> F T

d -> F T

e -> F T

f -> F T

edge - list

a -> b

b -> a

a -> c

c -> a

b -> e

c -> b

b -> a

d -> b

c -> e

e -> c

e -> f

f -> e

c -> f

f -> c

Adj List

a -> { b, c }

b -> { a, e, d }

c -> { a, b, e, f }

d -> { b }

e -> { c, f }

f -> { e, c }

initial -> src -> 'a'

q.push(a)

vis[a] = T

queue

node = q.front()

node [a] -> print

q.pop()

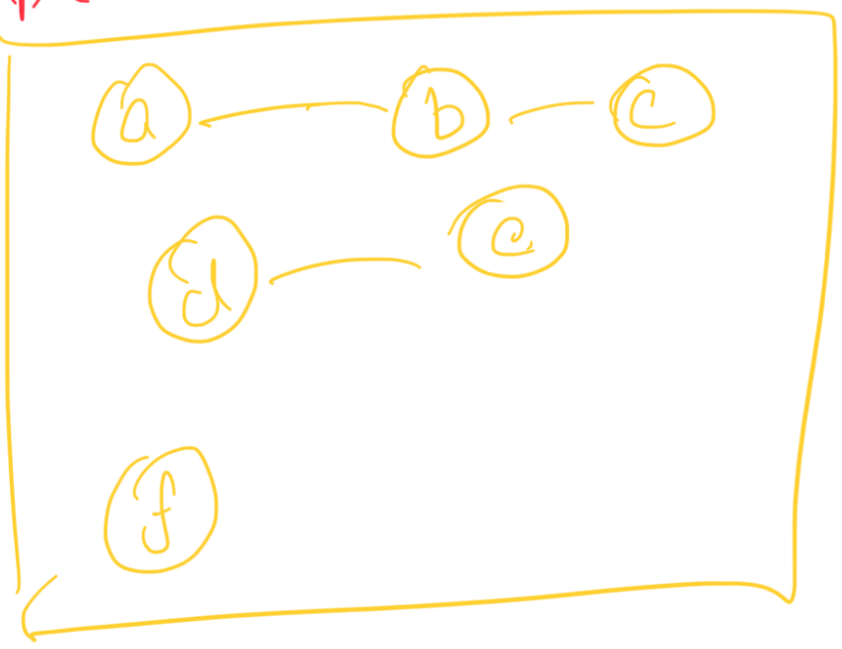
cout << a

o/p

a b c d

e f

Impt -> Yhi Pe glti hoti hai



only a, b, c -> will print, In case of Disconnected graph

→ Call function for every node

a d f

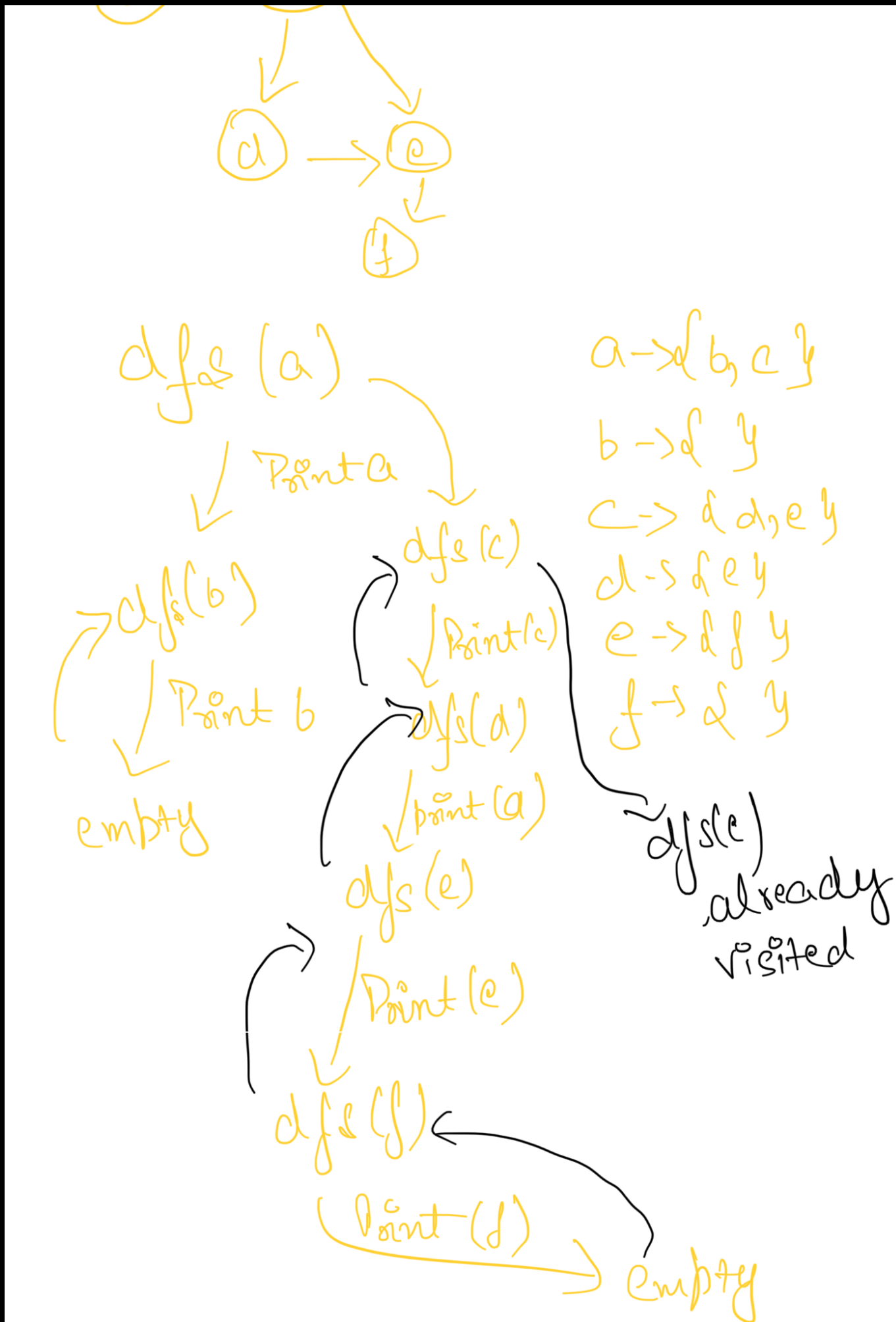
for These Nodes

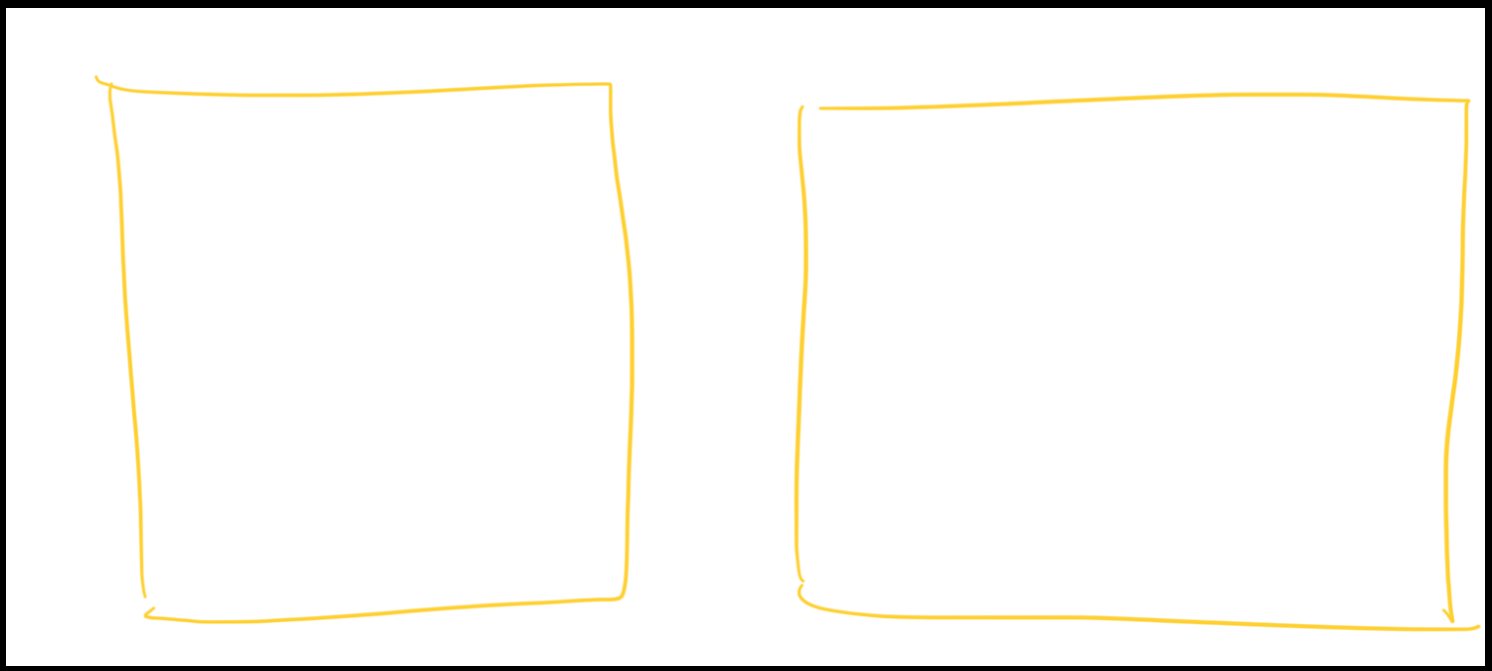
TC ->

SC ->

② -> DFS -> (Depth first Search)

↳ Recursion


a
e
f
dfs(a)
Print a
dfs(b)
Print b
empty
dfs(c)
Print(c)
dfs(d)
print(a)
dfs(e)
Print(e)
dfs(f)
Print(f)
empty
a -> { b, c }
b -> { }
c -> { d, e }
d -> { e }
e -> { f }
f -> { }
dfs(e)
already
visited



#DSA-Supreme-2.0