Graphs:

* Similar to trees but we can have closed loops in graphs.

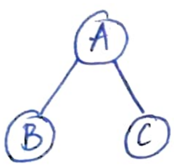
Set of (V,E) pair

V → set of vertices
E → set of edges

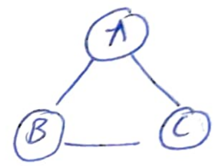
Vertices → Represented as circles also known as nodes

Edge —— Represented as lines connecting two vertices / nodes.

eg)



A, B, C → nodes, Vertices



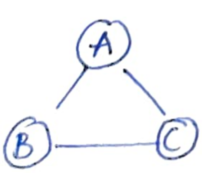
Only graph.

Tree or Graph.

Graph Terminology

1. Node
2. Edge
3. Adjacent Nodes
4. Degree of Node → Numbers of edges connected to that node
5. Size of graph → Total number of edges in graph
6. path → sequence of vertices from source node to destination node.

eg)



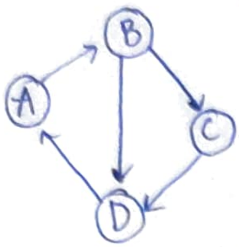
Degree of A = 2
B = 2
C = 2

Let
A → source
C → destination

path ⇒ A–B–C (or) A–C

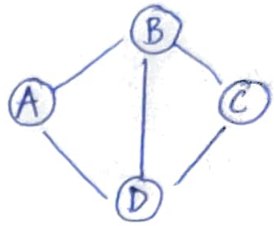
Types of graphs :

1. DIRECTED GRAPH



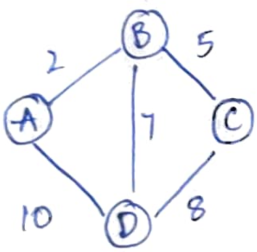
here direction
is specified
(A, B) ≠ (B, A)
unidirectional

2. UNDIRECTED GRAPH



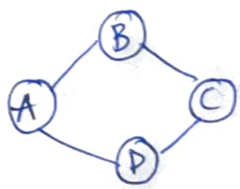
here we can travel
A to B or B to A
C to D or D to C
~~Bidect~~
Bidirectional.

3. WEIGHTED GRAPH



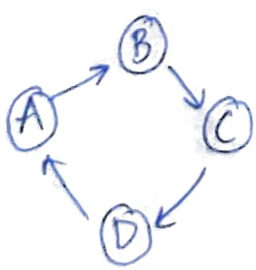
Here we specify
weightage for
every edge

4. UNWEIGHTED GRAPH



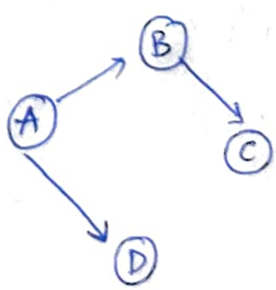
No weight is specified.

5. CYCLIC GRAPH



It forms a
cycle
A → B → C → D

6. ACYCLIC GRAPH



non-cyclic.

Representation of Graphs :

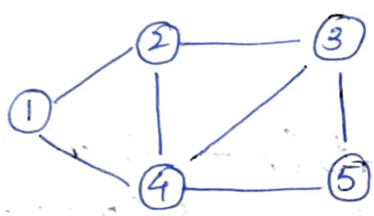
1. Adjacency Matrix
2. Adjacency list

1. Adjacency Matrix

It is a $n \times n$ Matrix where n is number of vertices ~~edges~~ in given graph.

If edge is available b/w nodes we fill it with 1 or 0. we fill 1 when an edge have a self loop.

eg) 1.



$\longrightarrow$ For directed graph.

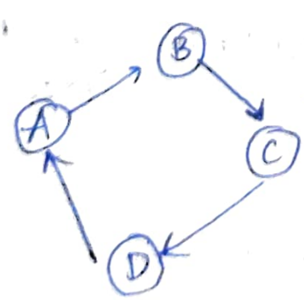
① — ②
we are having edge and ~~so~~ directed we fill it with 1.

$$\begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}_{5 \times 5} \end{array}$$

① — ③
No edge $\rightarrow$ fill 0

[if i & j are adjacent
$a[i][j] = 1$
or else 0]

2.

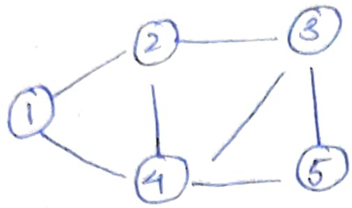


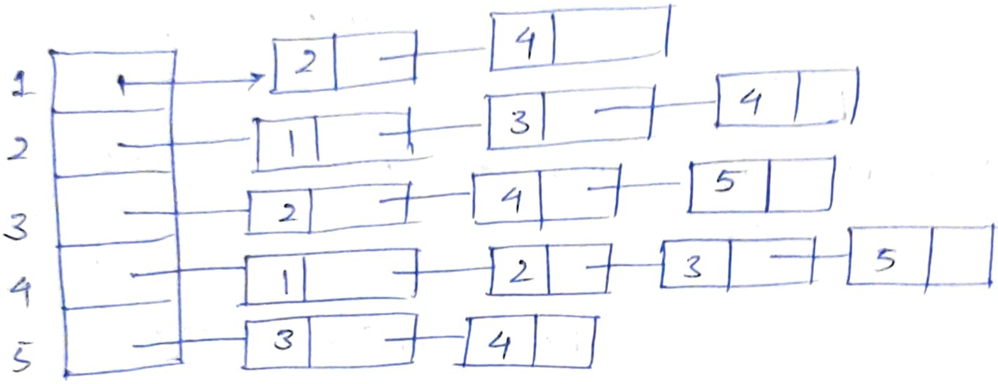
$\longrightarrow$ ~~dt~~ Undirected graph

$A \rightarrow B = 1$
$B \rightarrow A = 0$

$$\begin{array}{c|c} & \begin{matrix} A & B & C & D \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

2. Adjacency list

eg)  → undirected graph.



space complexity using Adjacency Matrix – $O(n^2)$

Adjacency List – $O(n+2e)$

For dense graph better to use Adjacency Matrix

For sparse graph better to use Adjacency List.

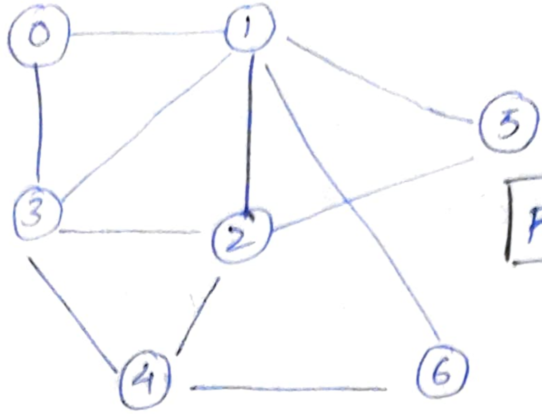
Graph Traversals

1) BFS (Breadth-First search Algorithm) → Levelorder on Binary Tree.
2) DFS (Depth First search Algorithm) → Preorder on Binary Tree

1) BFS

- Data structure used for BFS is Queue

eg



Queue: ~~0~~ ~~1~~ ~~3~~ ~~2~~ ~~5~~ ~~6~~ ~~4~~

Result : 0 1 3 2 5 6 4

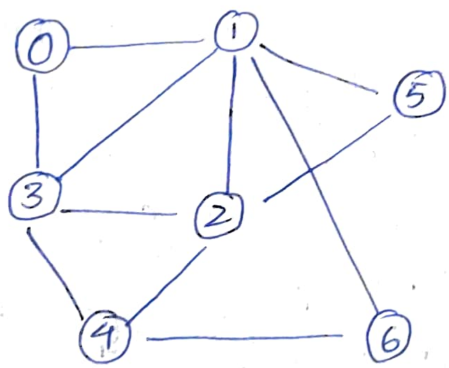
(There are numerous Valid BFS for a graph)

First complete visiting the adjacent nodes of a node and then continue with other nodes

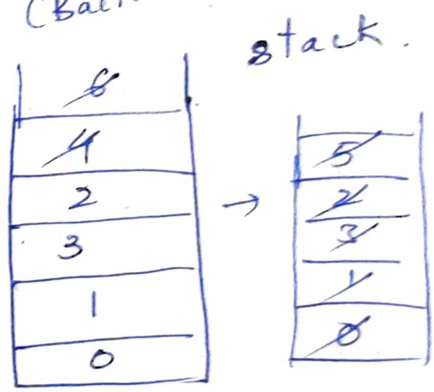
2) DFS

- Data structure used for DFS is **stack**.

eg



(Backtracking 6, 4)



Result : 0, 1, 3, 2, 4, 6, 5

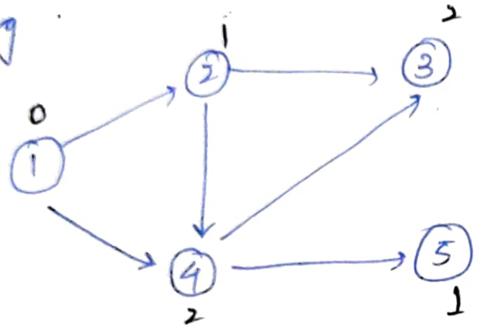
(After 6 there will be no adj vertices of 6 then we have to backtrack poping out in stack)

exploring the adjacent nodes

Topological sorting :

- It is a linear ordering of its vertices such that for every directed edge uv for vertex u to v, u comes before vertex v in ordering.
- Graph should be DAG (Directed Acyclic graph)
- Every DAG will have atleast one topological sorting.

eg)



~~Indeg~~

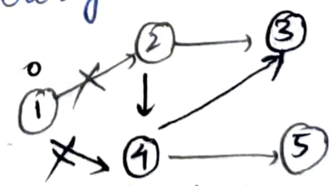
Indegree of a node is the numbers of edges coming towards that node.

Indegree of 1 is 0
2 is 1
3 is 2
4 is 2
5 is 1

We will start writing the topological ordering with node having least Indegree & then we delete that node & also delete all edges outgoing through it

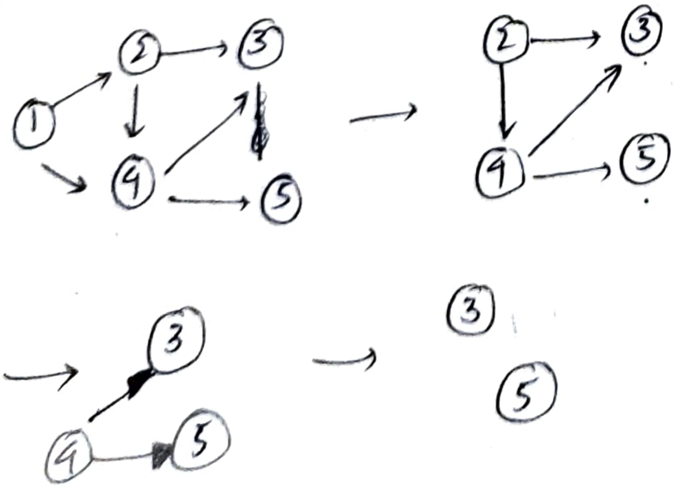
Result : 1 2 4 3 5
or
1 2 4 5 3

1.



1 is deleted

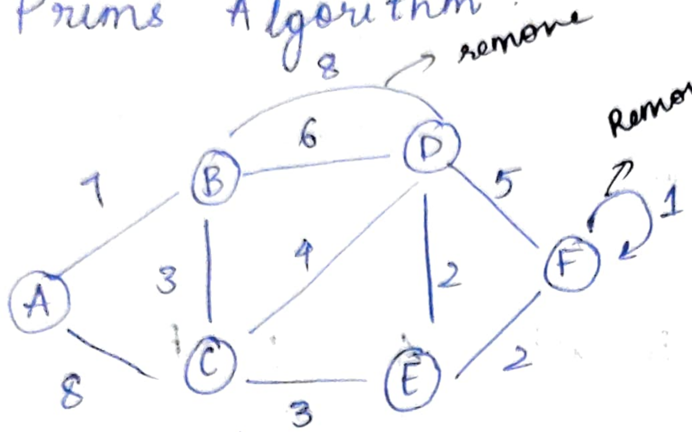
Now indegree is
2 → 0
3 → 2
4 → 1
5 → 1



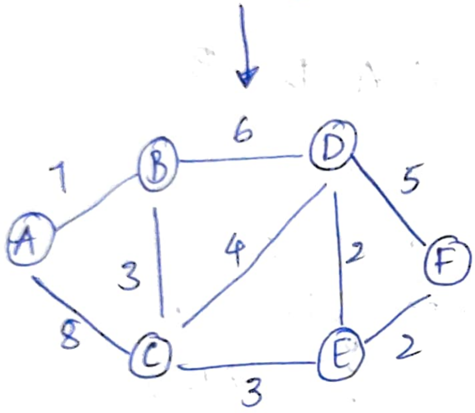
When the node is deleted indegree of other nodes changes. change the Indegree & continue the steps with least Indegree

Minimam spanning Tree for graph

1. Prim's Algorithm

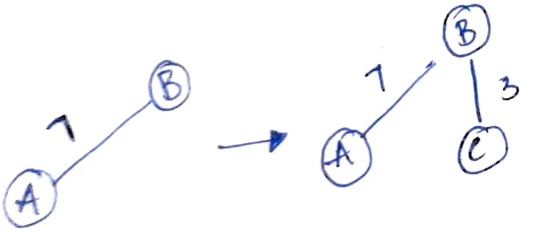


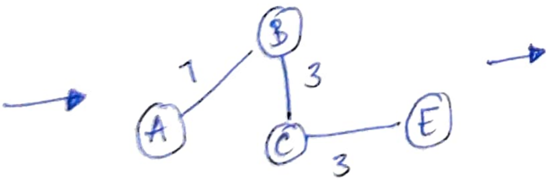
Remove 1. First Remove self loops
2. Remove parallel edges (delete edge having maximum weight)



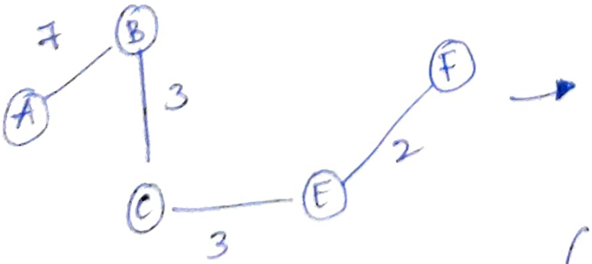
We can select any node as a root node in Prim's Algo.

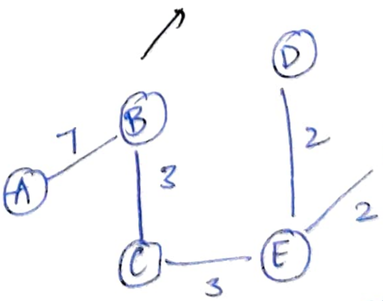
Let choose A as root node
Now check the incoming or outgoing edges of A and select the minimum edge. Now check B and A continue the same (minimum edge).





Minimum spanning tree.

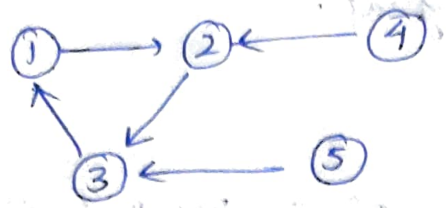




edges = |V| - 1 in spanning Tree

(Now when All ~~edges~~ nodes are present ~~in~~ then we stop)
(Total is minimum)

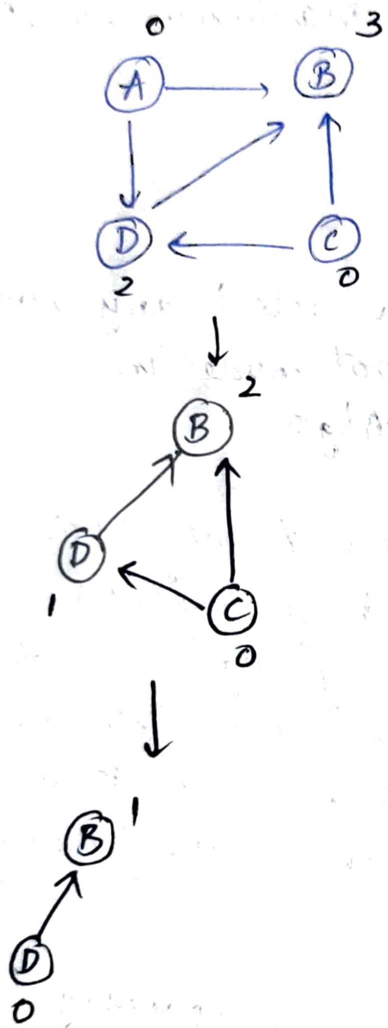
eg 2.



1 ⟶ 2 ⟶ 3
cycle
(It is not Acyclic)

We cannot ~~fing~~ find Topological sort

eg 3.



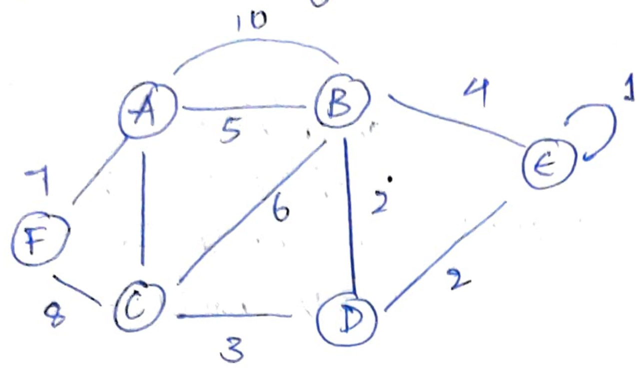
Result A C D B

or

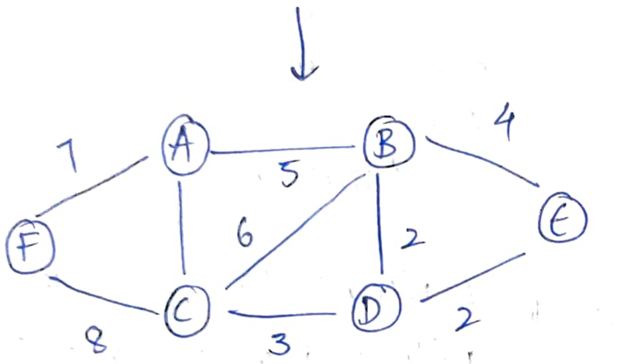
C A D B.

(more than one Topological sort is possible)

2. Kruskal's Algorithm

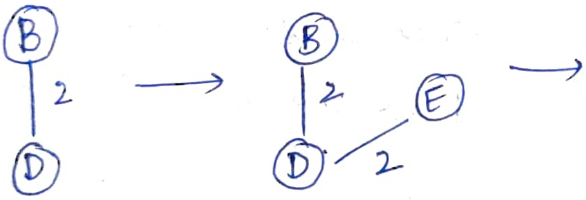


1. First Remove self loops
2. Remove parallel edges (delete edge having max ~~wi~~ weight)

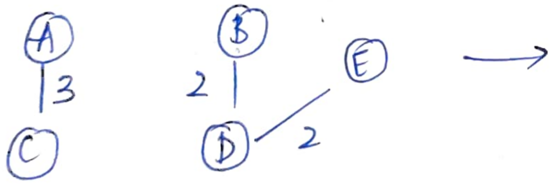


BD = 2
DE = 2
AC = 3
CD = 3
BE = 4
AB = 5
BC = 6
AF = 7
FC = 8

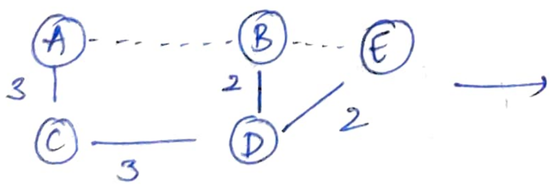
Increasing edge weight



We select minimum edge weight in Kruskal's Algo. (BD here)



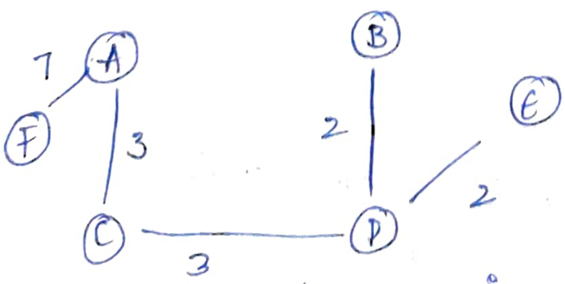
When we are connecting there should be no



(Now BE makes a closed graph so we dont use it)

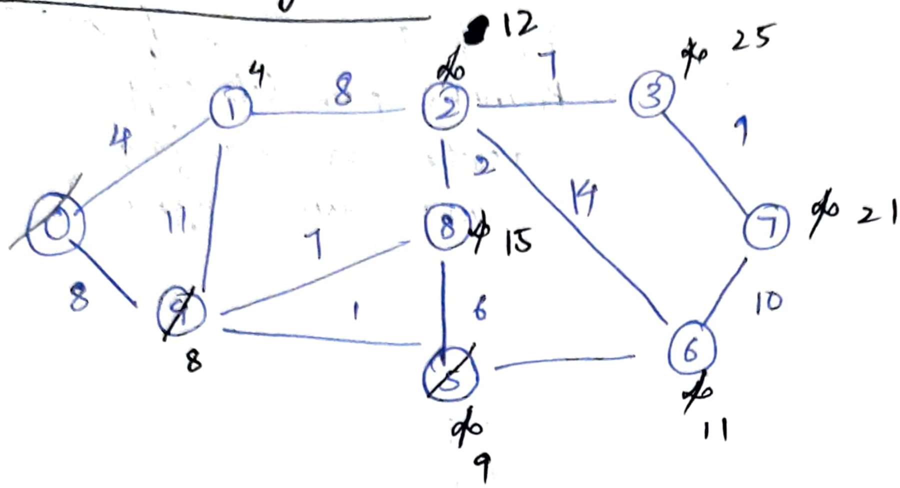
AB, BC cannot be connected

FC cannot be connected.



Min spanning tree

Dijkstra Algorithm: (single source)



Mention distances from a node to All nodes.

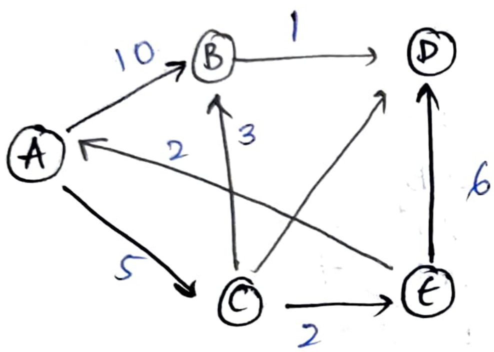
0 – 0 = 0
0 – 1 = 4
0 – 4 = 8
0 – Other nodes is ∞

$$\left\{ \begin{array}{l} \text{if } (d(u) + c(u,v) < d(v)) \\ \quad d(v) = d(u) + c(u,v) \end{array} \right\}$$

Now 1 is at least distance so select 1 & write distance

Minimum weight of node is selected.

eg)



Source	A	B	C	D	E
A	$\boxed{0}$	∞	∞	∞	∞
C		10	$\boxed{5}$	∞	∞
E		8		14	$\boxed{7}$
B		$\boxed{8}$		13	
D				$\boxed{9}$	

A to B = 8

shortest path A to D

$\boxed{ACBD = 9}$

5+3+1 = 9

(For negative edges Dijkstra does not exist)

We cannot use Dijkstra