GRAPHS

- Combination of nodes without any rule for connection
- A non-linear data structure.
- Useful when data have some relation.
  for example:
→ Social Network (Fb) → unweighted undirected
→ WWW → unweighted directed
→ Intercity roadmap → weighted undirected
→ Intracity roadmap → weighted directed
- Unweighted graph is just a graph with equal weight usually equals to 1.
- Undirected graph can be transformed to directed graph but not vice-versa.
  → reloading a webpage on www
- SELF-LOOP is when source and destination node is same
- MULTI EDGE or PARALLEL EDGE having same connection twice.
  ↳ flight network
- SIMPLE GRAPH if no self loop and multiedge

Number of edges,
of simple graph

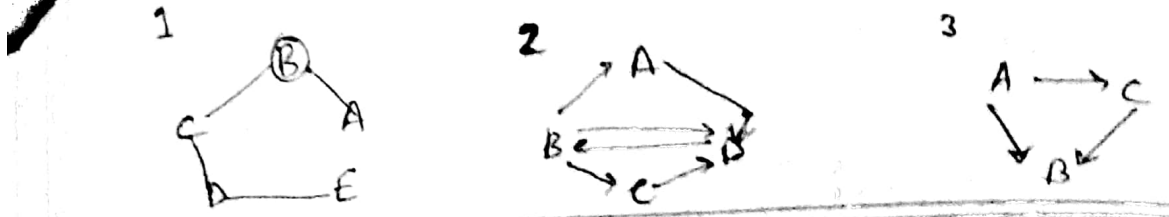
DIRECTED

$$0 \leq |E| \leq |V|(|V|-1)$$

UNDIRECTED

$$0 \leq |E| \leq |V|(|V|-1)/2$$

Max. edges will be almost $|V|^2$
DENSE if close to $|V|^2$
SPARSE if close to $|V|$



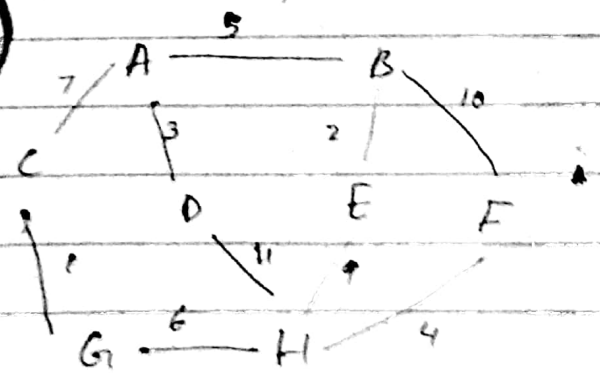
1 CONNECTED undirected graph have connection between every node.

2 STRONGLY CONNECTED digraph have connection between every node.

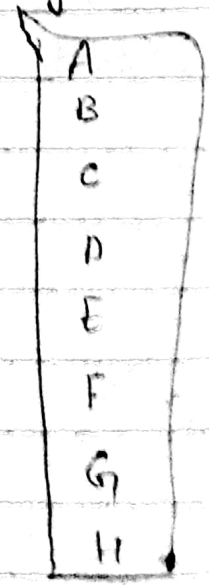
3 WEAKLY CONNECTED digraph have connection between nodes but not directly.

REPRESENTATION

- To implement GRAPHS we can make two lists. One for vertices/nodes another for edges/~~connections~~ links.



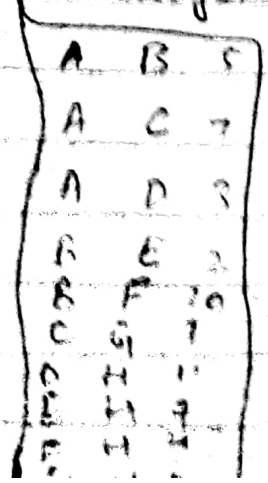
↳ vector<string> vertex;



for edges

↳ class Edge
{
string startVertex;
string endVertex;
int weight
};

↳ vector<Edge> edges;



FOR BETTER SPACE COMPLEXITY

Instead we can do in edge obj is to store indices of respective vertex in it list ie.

```
class Edge {
        int startVertex;
        int endVertex;
        int weight;
};
```

0	1	5
0	2	7
0	3	3
1	4	2
1	5	10

TIME ANALYSIS

* Finding Nodes adjacent to given node
  to we have to traverse Edge list
  so $O(|E|) = O(V^2)$
* Check if given nodes are connected
  to We have to traverse Edge list
  so $O(|E|) = O(V^2)$

} costly

BETTER APPROACH

- As time complexity is a issue, what we can do is to implement a 2D matrix of integers to represent edges. If there is a edge from i to j then A[i][j] will equal to 1 Else 0. start end

	0	1	2	3	4	5	6	7
0	0	1	1	1	0	0	0	0
1	1	0	0	0	1	1	0	0
2	1	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	1
4	0	1	0	0	0	0	0	1
5	0	1	0	0	0	0	0	1
6	0	0	1	0	0	0	0	1
7	0	0	0	1	1	1	1	0

- For storing weighted graph we will set weight to A[i][j] instead of 1. And instead of 0 we can use any constant value which will never be weight like ∞ (INT_MAX).

With Adjancy matrix approach we tackled time complexity but now space complexity is a tradeoff here i.e. $O(|V|^2)$.

- AM is good for dense graph but usually graphs are sparse ie. edges will pretty less than vertices.

→ Time analysis for checking adjacent node, will be O(V).

→ And for checking link between two nodes will be simply O(1).

- Each row is representing links to one node with all other nodes.

↳

	0	1	2	3	4	5	6	7
A[0][j] = [	0	1	1	1	0	0	0	0]

- Redundancy is that we are also storing that A node do not have links with A(selfloop), E, F, G, H.

Here comes in picture Adjancy list where we store that to which nodes an node is connected

↳ A[3] = [1 2 3]

Here indexes are just indices and value is the index of vertices list.

To implement this list we have multiple options like

- Array (jagged)
- LL
- T

List will look like

0	1 2 3
1	0 4 5
2	0 6
3	0 7
4	1 7
5	1 7
6	2 7
7	3 6 4 5

A naive implementation of above can be array of ~~size~~ ptr with size = vertices.

↳ int *A[8].

```
A[0] = new int[3];
A[1] = new int[2];
  .
  .
  .
A[7] = new int[7];
```

space = $O(e)$ | $e \ll V^2$

time for checking connection

$O(v)$ → linear search

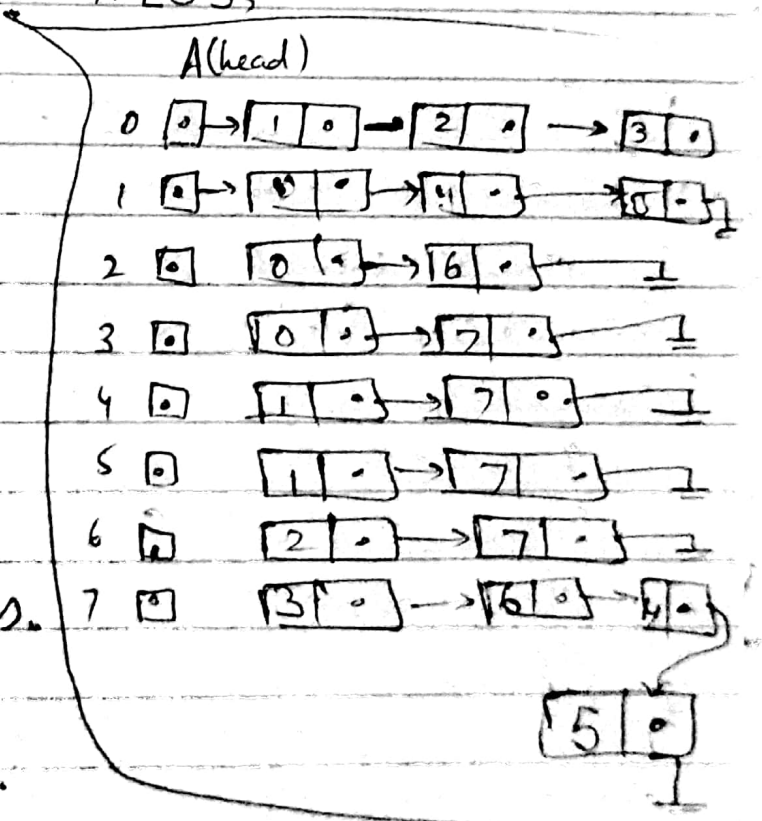
time for finding all neighbours

$O(v)$

* In an array insertion is costly, so we can simply use a linked list to implement adjacency list. We can make array of node ptr with size = vertices.

↳

```
class Node { int data; Node *next; };
Node *A[8];
```



- In a LL also many operations are O(n)
- We may use a balanced BST (AVL) for this list which cost O(log n) for almost all operations.

MINIMUM SPANNING TREE

For a graph $G=(V,E)$ the MST will be $G'=(V';E')$ where

$$V'=V$$

$$E'=|V|-1$$

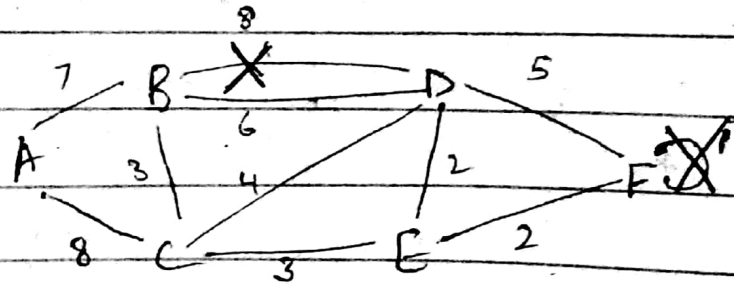
Removing an edge will make ST disconnected and adding an edge will create a loop.

- If weight of every edge is distinct then there exist one and only MST of a graph.
- A completed undirected graph contains n^{n-2} spanning tree; where n = no. of vertices
- Every connected and undirected graph has atleast one ST.
- Disconnected graph doesn't have ST.
- From a complete graph by removing max $(e-n+1)$ edges we can construct a ST.

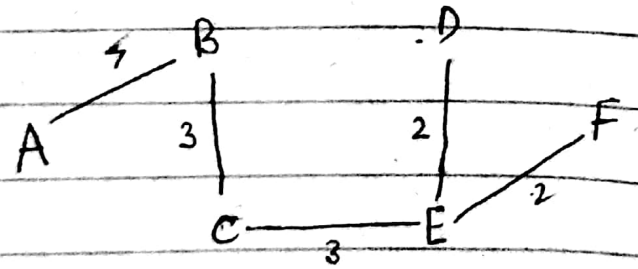
PRIM'S ALGO

-• Remove all loops and multiedges (the one with greater cost)
-• Select any vertex as starting point
1. Check all incident edges from that vertex and select the one with min. cost.
4. Now, check all incident edges of all vertices which aren't selected yet and select the min. cost edge.
5. Repeat step 4 till covered all vertices

↳



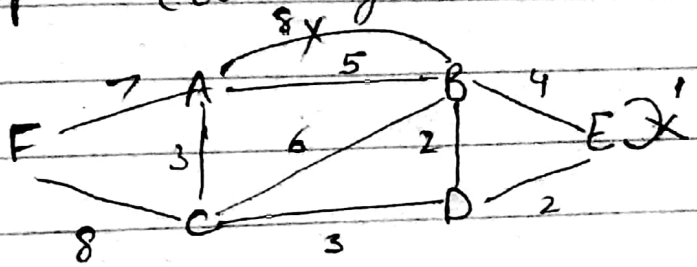
starting point = A



cost = 17

KRUSKAL ALGO

1. Remove all loops and multiedges (the one with greater cost).
2. Write down all edges in ascending order.
3. Start constructing MST with minimum cost edge.
4. Connect next edge while not making any cycle.
5. Repeat step 4 till edges ended.



✓ BD = 2

✓ DE = 2

✓ AC = 3

✓ CD = 3

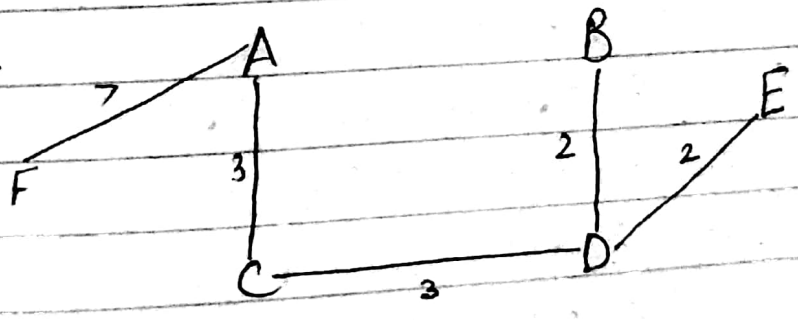
✗ BE = 4

✗ AB = 5

✗ BC = 6

✓ AF = 7

✗ FC = 8



cost = 17

DIJKSTRA ALGO

- This algo works to find single source shortest path for both directed and undirected graphs.

1. Select the given vertex and set its distance to zero i.e. d[v] = 0
2. Assume or set distances of all other vertices to ∞ or some invalid value.
3. Calculate distance of all adjacent vertices to given vertex.
   ↳ d[v] = d(u) + c(u, v)]
   where, u is startVertex and v is endVertex
   c is cost (weight) from u to v.
4. The condition here is to check if the calculated distance is less than already assigned distance which will be ∞ (INT_MAX) for the first time. If true then only update distance.
5. After exploration of a vertex, select the one with minimum distance and repeat above steps.