Graph :- Non-linear data structures

V → set of vertices

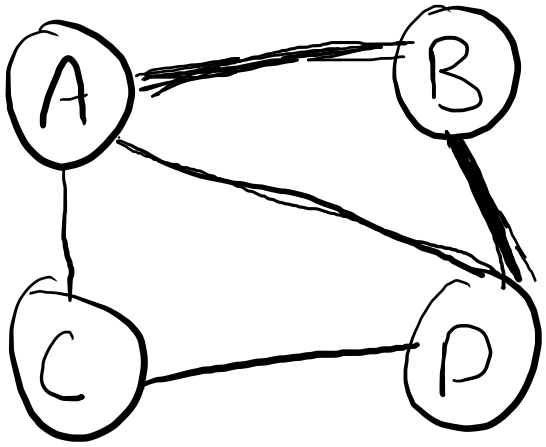
E → set of Edges

G = (V, E)

e = [A, B]

A → B

B → A



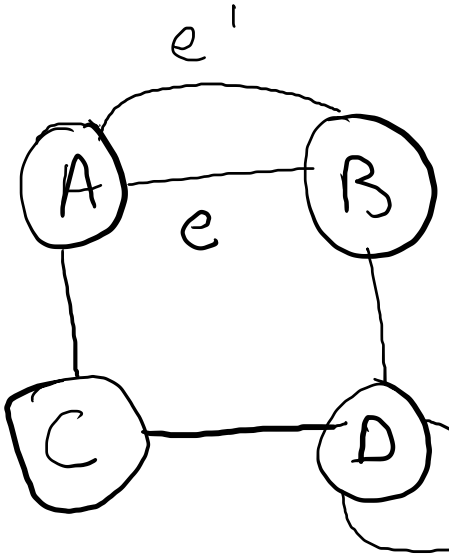
Path = (A, B, D)

A path of length n has n+1 nodes

Connected

Complete Graph :- N nodes

$N(N-1)/2$ Edges.

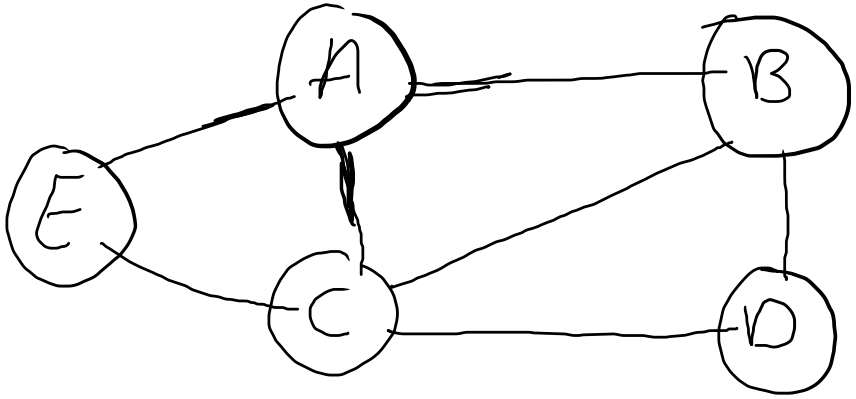


$e = [u, v]$

$e' = [u, v]$

$e'' = [u, u]$

Multigraph :-

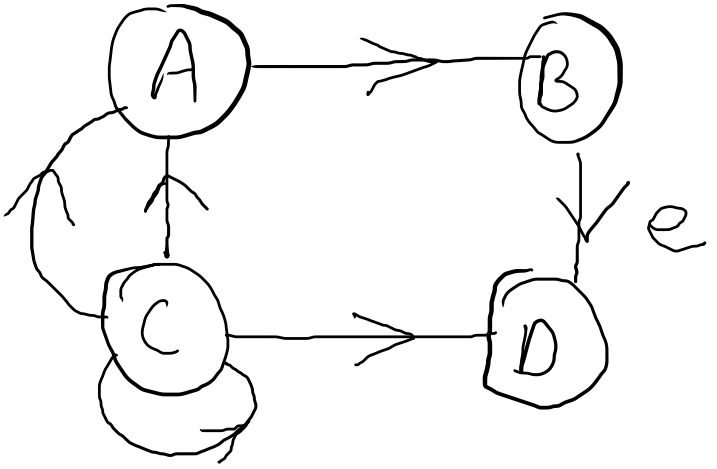


$\deg(A) = 3$

$\deg(D) = 2$

Directed Graph :-

(Digraph)

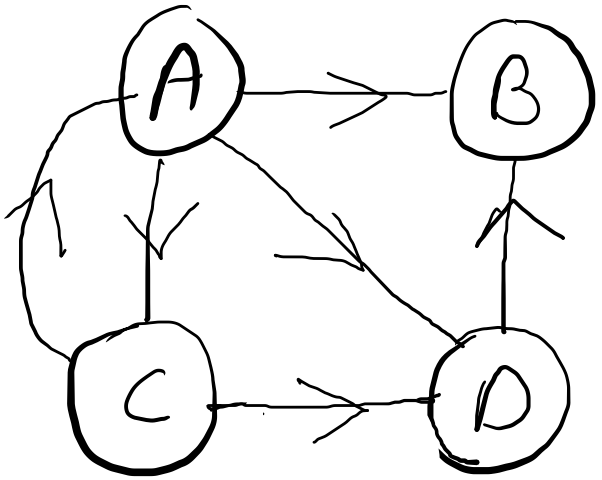


$e = (B, D)$ ordered pair

$\text{Indeg}(A) = 2$

$\text{outdeg}(A) = 1$

Sequential Representation :- Adjacency Matrix



Sink

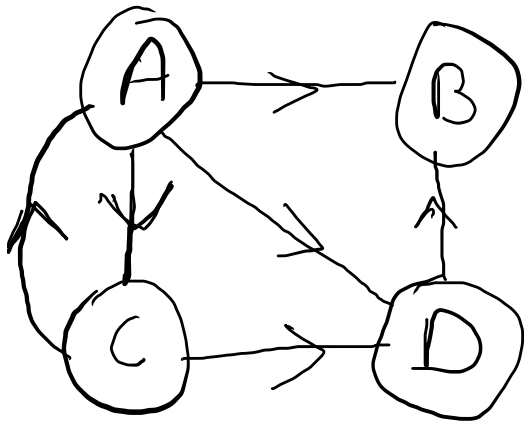
$A =$

	A	B	C	D
A	0	1	1	1
B	0	0	0	0
C				
D				

Source

Sink

Path Matrix :-



Sink

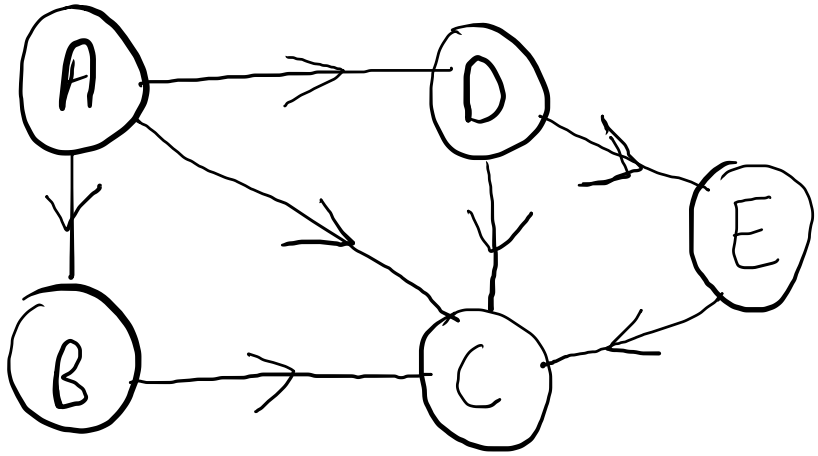
$$P_{ij} = \begin{cases} 1 & \text{if there is a path} \\ 0 & \text{otherwise} \end{cases}$$

$$P = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ & & & \\ & & & \end{bmatrix} \end{array}$$

Transisbive Closure

= Adjacency Matrix

Linked Representation : Adjacency List



Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

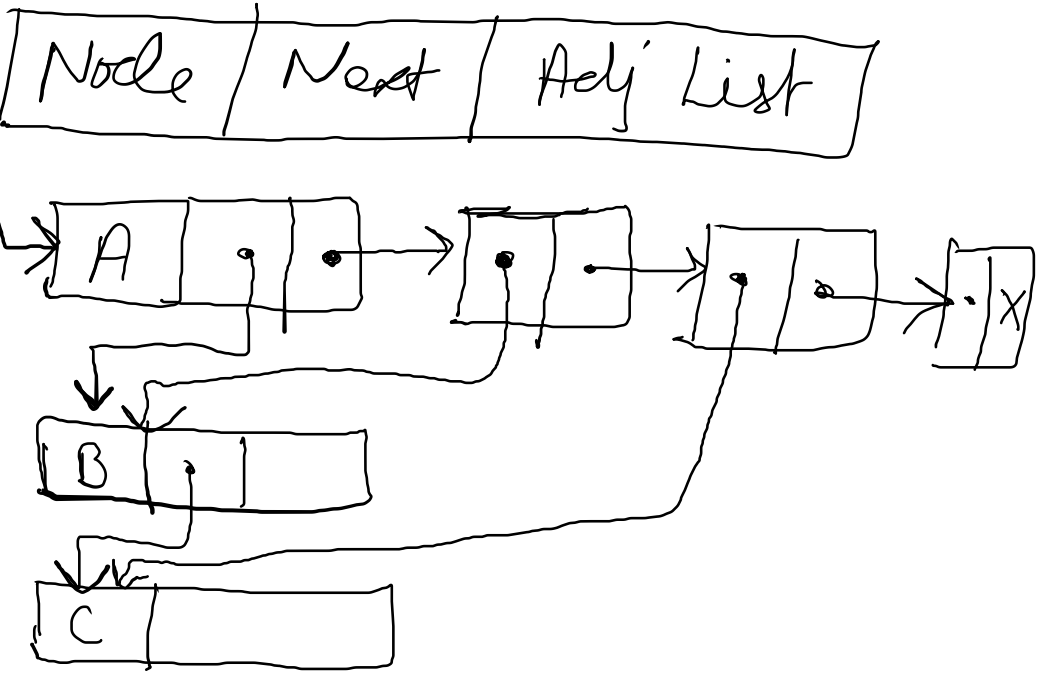
A =

	A	B	C	D
A	0	1	0	1
B				
C				
D				

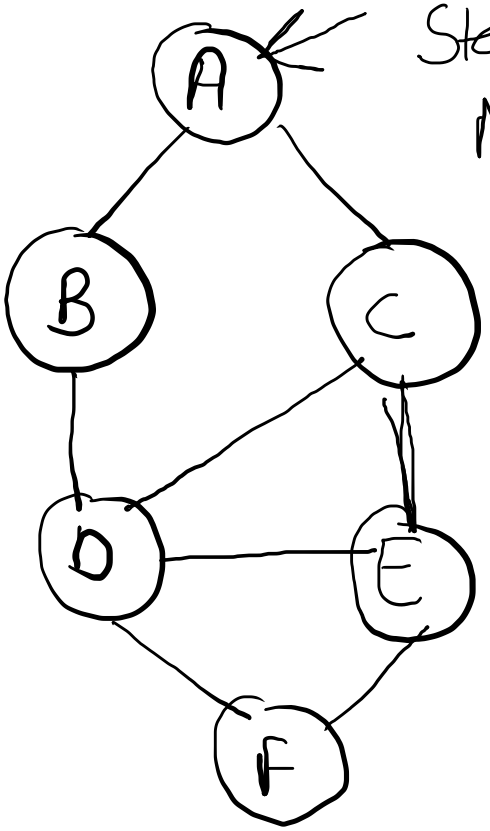
⇒ `int a[10][10];`

Node	Adjacency list
A	B, C, D
B	C
C	
D	

⇒

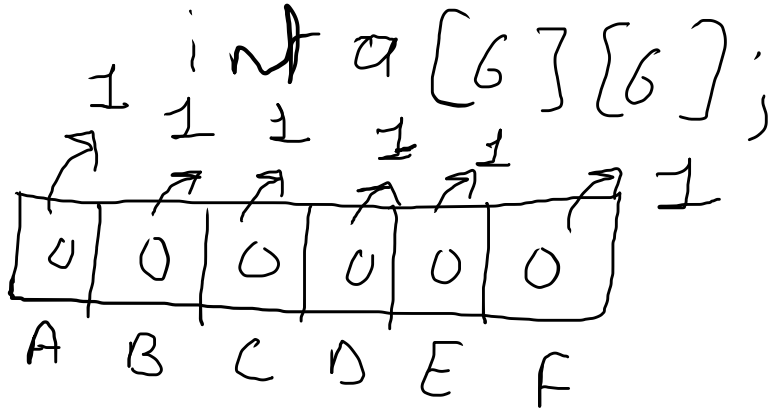


BFS: Breadth First Search



Starting Node

Visited



Queue : Initially Empty

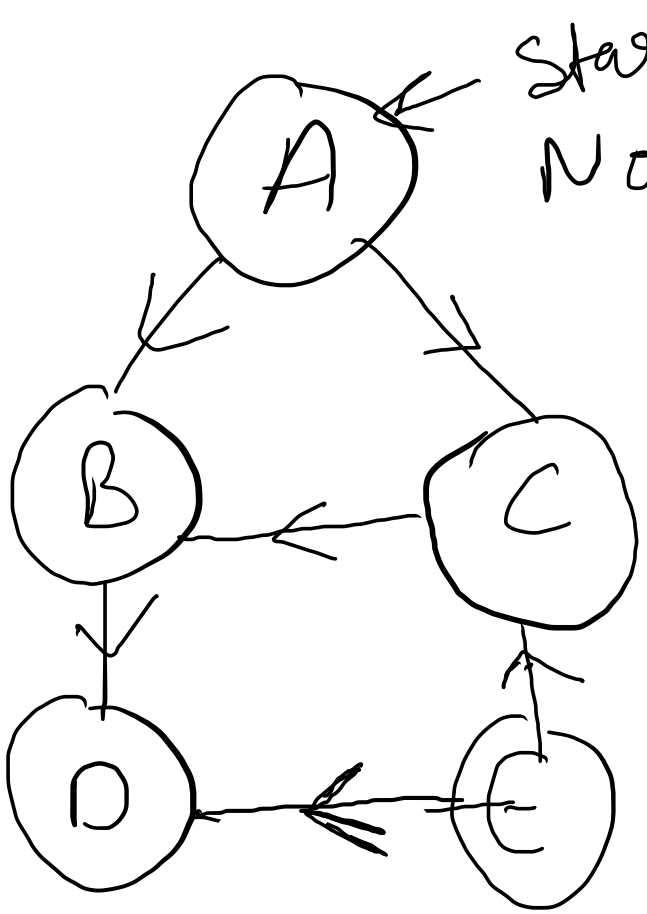
Queue : ~~A B C D E F~~

Print : A B C D E F

```
int a[6][6], q[6], visited[6];
for( )
Enqueue(q, A); visited[A] = 1;
while(q is not empty) {
S = dequeue(q);
print S;
Search adjacent for S
}
```

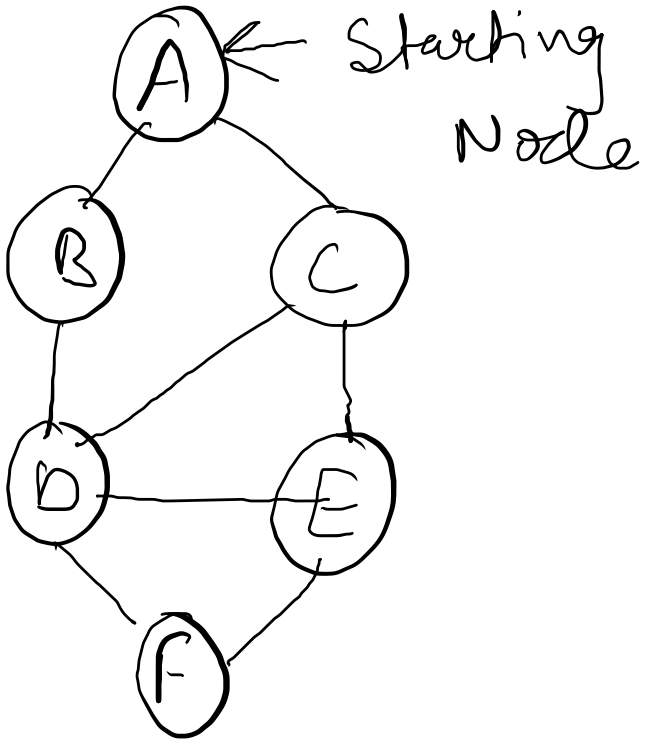
→

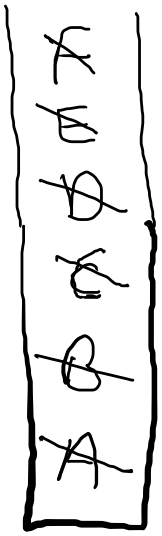
```
for(j=0; j<6; j++)
{
if(a[S][j]==1 &&
   visited[j]=0)
{
  enqueue(q, j);
  visited[j] = 1;
}
```



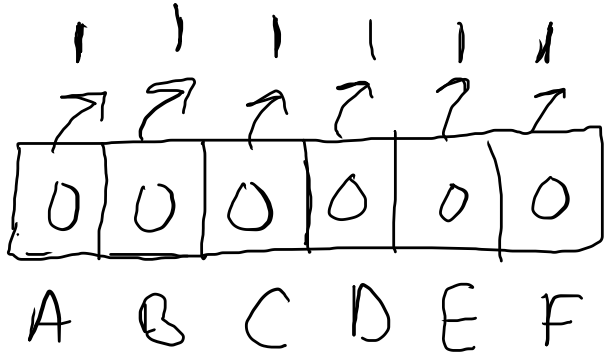
BFS ⇒ A B C D

DFS: Depth First Search





Visited:



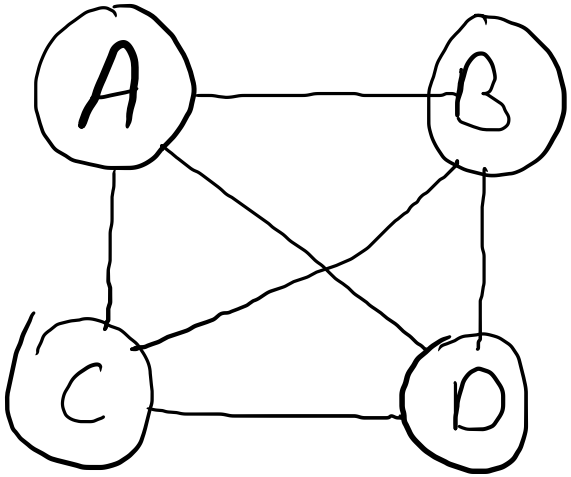
Stack: Initially empty

Stack: ~~A~~ B C D E ~~F~~

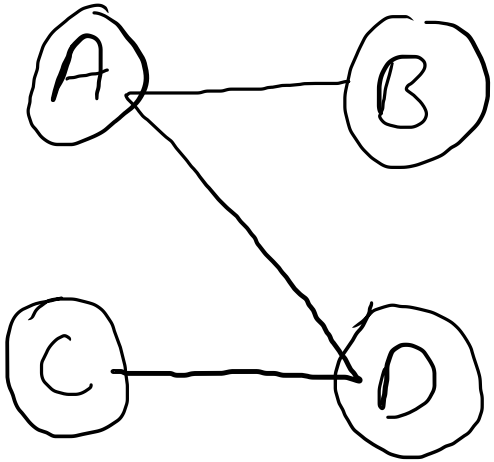
Print: A C E F D B

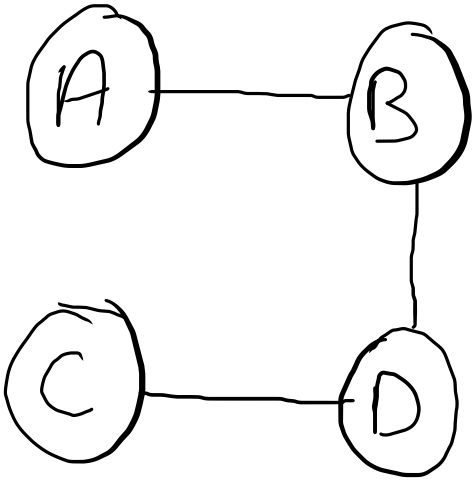
```
int a[6][6], s[6], visited[6];
push(S, A); visited[A] = 1;
while ( S is not empty ) {
    pop top element from stack & print it;
    element = pop(s);
    for( ) ← Search adjacent nodes of element
    push all not visited adjacent nodes into
    the stack and set their visited
    to 1;
}
```

MST: (Minimum Spanning Tree)

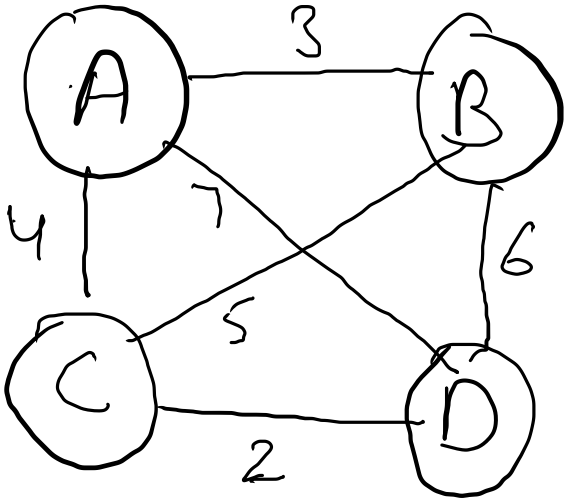


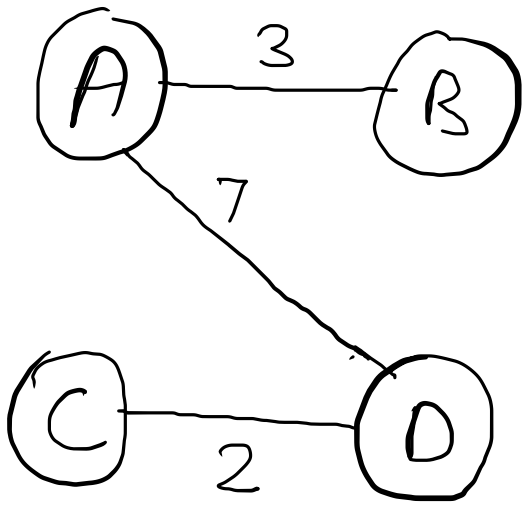
Graph



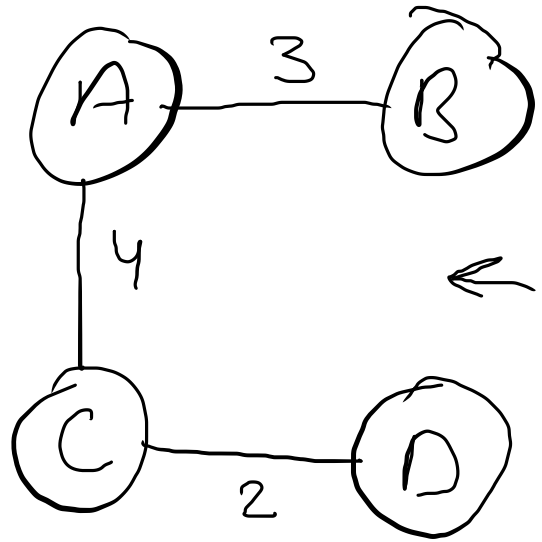


Spanning Trees





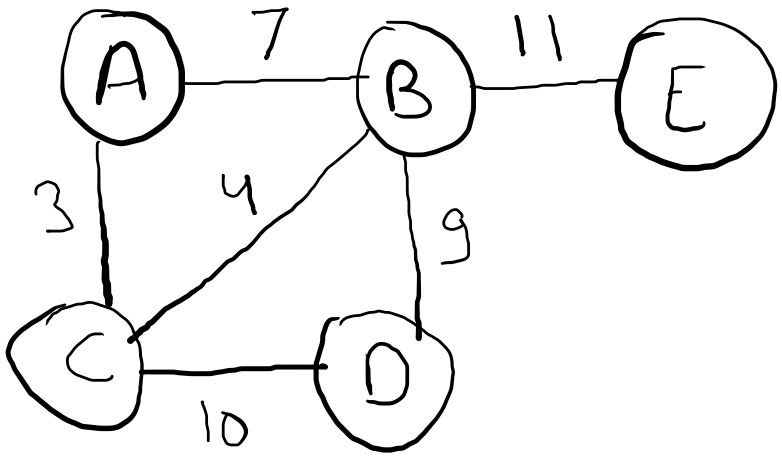
Total cost = 12

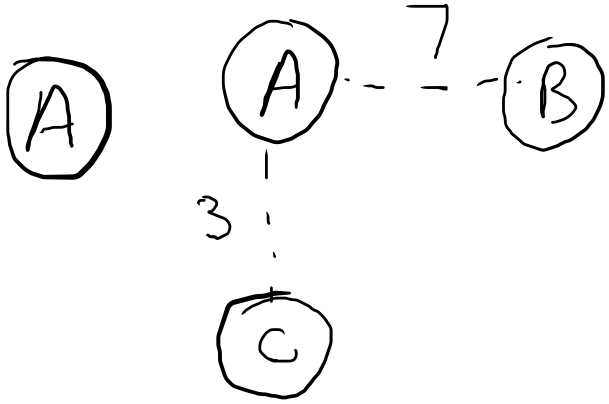


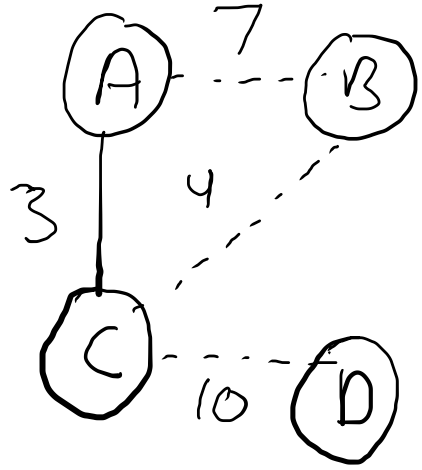
← MST

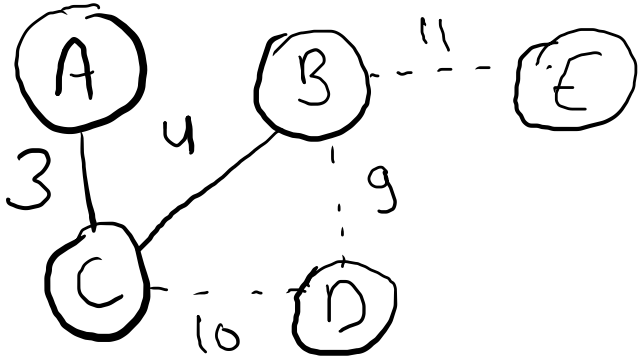
Total cost = 9

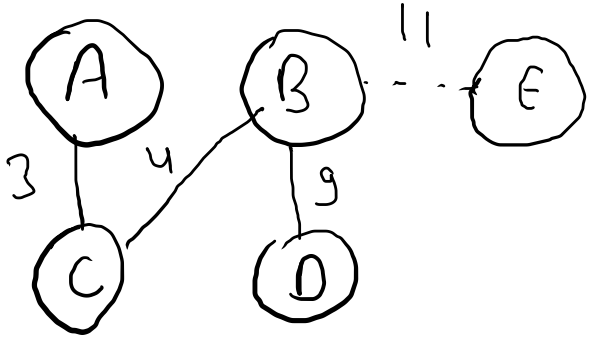
MST : Minimum Spanning Tree

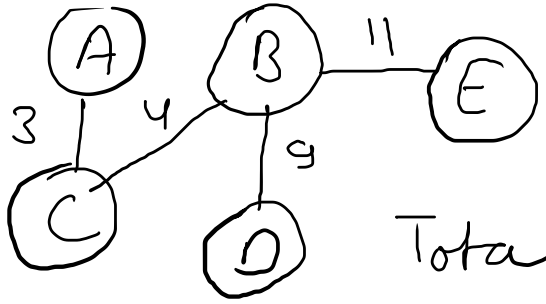










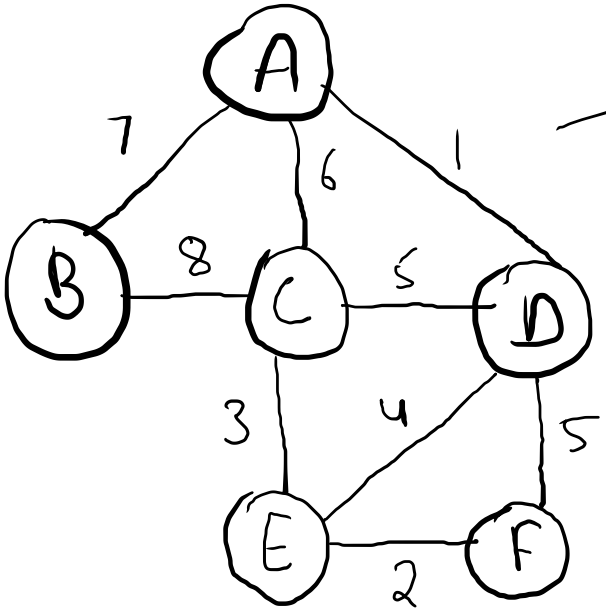


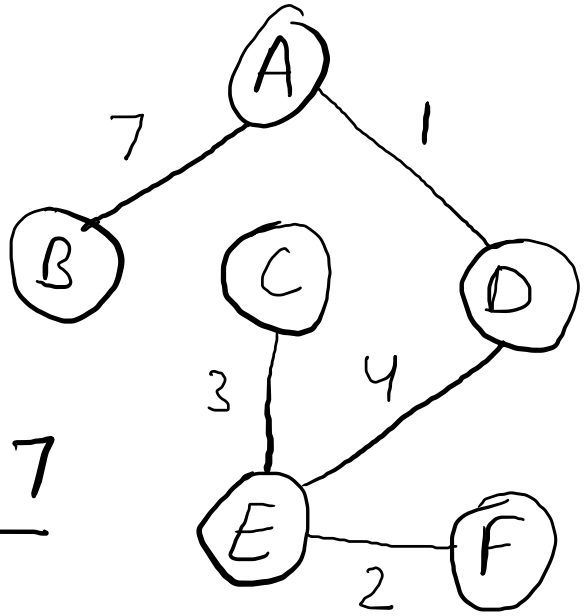
Total cost
= 27

Prim's Algorithm :-

1:- Select a starting vertex
2:- Repeat steps 3 & 4 until there are fringe vertices
3:- Select an edge containing the tree vertex & the fringe vertex that has min. weight
4:- Add the selected edge & the vertex to your tree.
5:- Exit

Kruskal's Algorithm :-





Total cost = 17

$Q = \{(A,D), (E,F), (C,E), (E,D), (C,D), (D,F), (A,C), (A,B), (B,C)\}$

1. Create a forest in such a way that each node can be seen as a separate tree
2. Create a priority queue Q that contains all the edges of the graph.
3. Repeat steps 4 & 5 while Q is not empty
4. Remove an edge from Q
5. If the edge connects two different trees then Add it else simply discard the edge.
6. Exit