


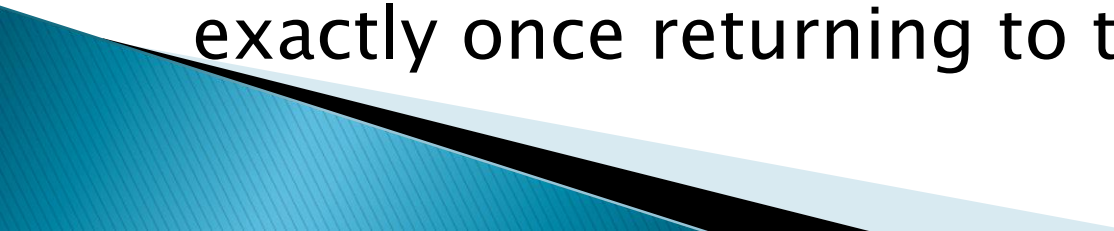
# Unit III: Graphs

Prof. Pujashree Vidap

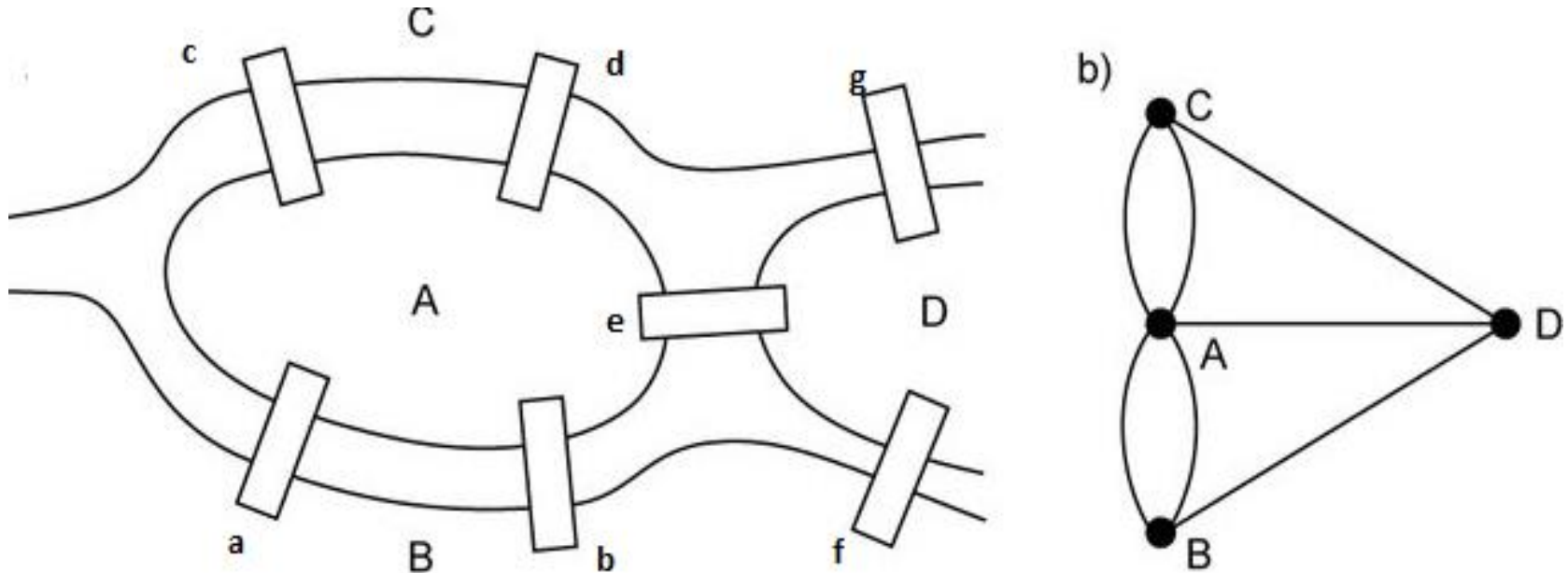
# Contents:

- ▶ Basic Concepts related to graphs
  - ▶ Storage representation,
    - Adjacency matrix,
    - adjacency list,
    - adjacency multi list,
    - inverse adjacency list.
  - ▶ Traversals–depth first and breadth first
  - ▶ Minimum spanning Tree:
    - Greedy algorithms for computing minimum spanning tree– Prims and Kruskal Algorithms
    - Dijkstra's Single source shortest path
  - ▶ All pairs shortest paths– Floyd–Warshall Algorithm
  - ▶ Topological ordering
- 

# Introduction

- ▶ The first recorded evidence of the use of graphs dates back to 1736 when Euler used them to solve the now classical Koenigsberg bridge problem.
  - ▶ The land areas themselves are labeled  $A-D$ . These land areas are interconnected by means of seven bridges  $a-g$ .
  - ▶ The Koenigsberg bridge problem is to determine whether starting at some land area it is possible to walk across all the bridges exactly once returning to the starting land area.
- 

# Koenigsberg bridge problem

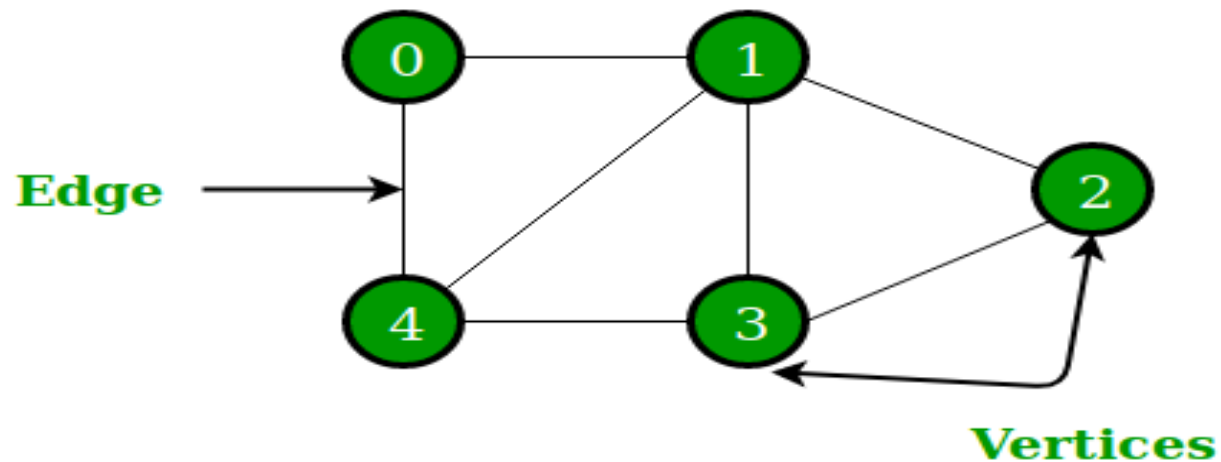


Since this first application of graphs, they have been used in a wide variety of applications. Some of these applications are: analysis of electrical circuits, finding shortest routes, analysis of project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences etc.

# Definitions and Terminology

- ▶ Graph is a non linear data structure consisting of vertices(nodes) and links between vertices(edges).
- ▶ A graph,  $G$ , consists of two sets  $V$  and  $E$ .
  - $V$  is a finite non-empty set of *vertices*.
  - $E$  is a set of pairs of vertices, these pairs are called *edges*.
  - $V(G)$  and  $E(G)$  will represent the sets of vertices and edges of graph  $G$ .
  - We also write  $G = (V, E)$  to represent a graph.
- ▶ The set of edges describes relationship between vertices.

# Example



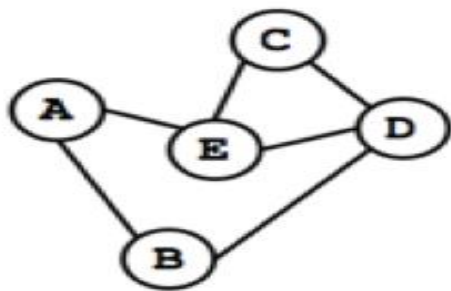
$V = \{0,1,2,3,4\}$  and  $E = \{(0,1), (1,2), (2,3), (3,4), (0,4), (1,4), (1,3)\}$ .

# Types of Graphs

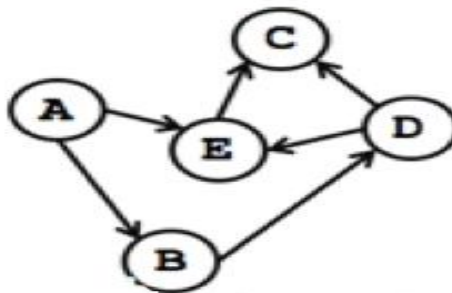
- ▶ There are several common kinds of graphs
  - Directed or undirected
  - Weighted or unweighted
  - Cyclic or acyclic
  - Connected or disconnected

# Directed and Undirected Graph

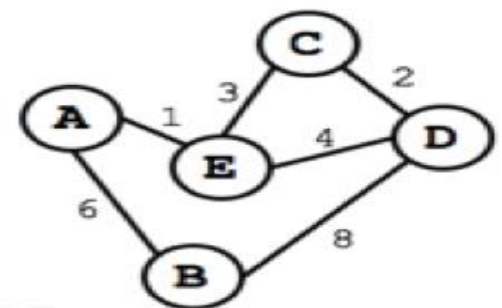
- Undirected graph : In this graph the pair of vertices representing any edge is unordered . Thus, the pairs  $(v_1, v_2)$  and  $(v_2, v_1)$  represent the same edge(edges have no direction).
- Directed graph(Digraph): In this each edge is represented by a directed pair  $(v_1, v_2)$ .  $v_1$  is the tail and  $v_2$  the head of the edge. Therefore  $\langle v_2, v_1 \rangle$  and  $\langle v_1, v_2 \rangle$  represent two different edges.



(A) Undirected Graph



(B) Directed Graph

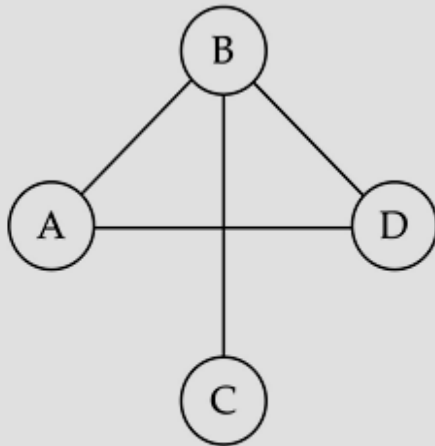


(C) Weighted Graph



# Examples:

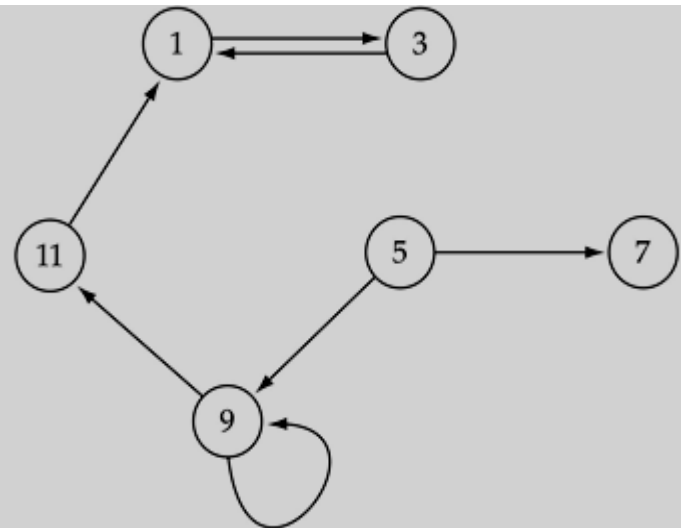
Graph1



$V(\text{Graph1}) = \{ A, B, C, D \}$

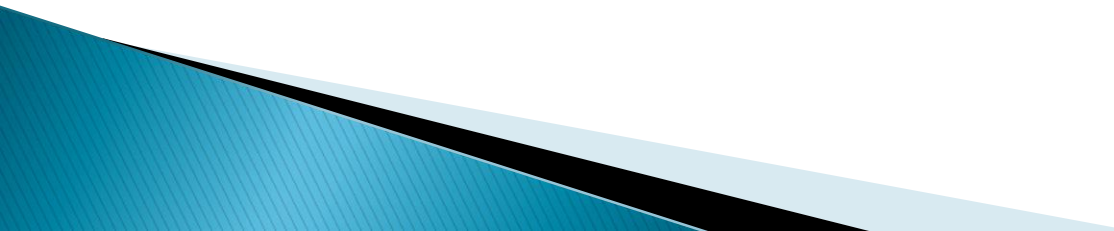
$E(\text{Graph1}) = \{ (A, B), (A, D), (B, C), (B, D) \}$

Graph2



$V(\text{Graph2}) = \{ 1, 3, 5, 7, 9, 11 \}$

# Weighted and Un weighted Graph:

- ▶ Weighted graph: edges have a weight
    - Weight typically shows cost of traversing
    - Example: weights are distances between cities
  - ▶ Unweighted graph: edges have no weight
    - Edges simply show connections
    - Example: course prereqs
- 

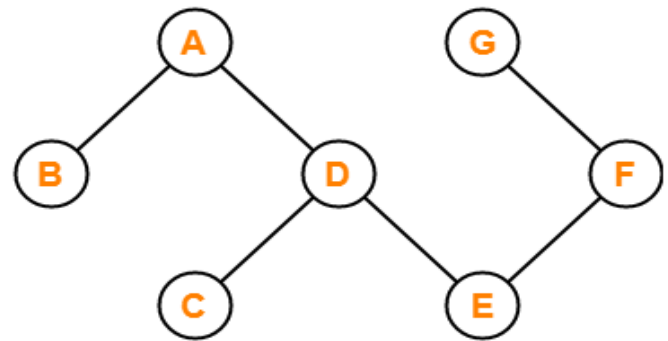
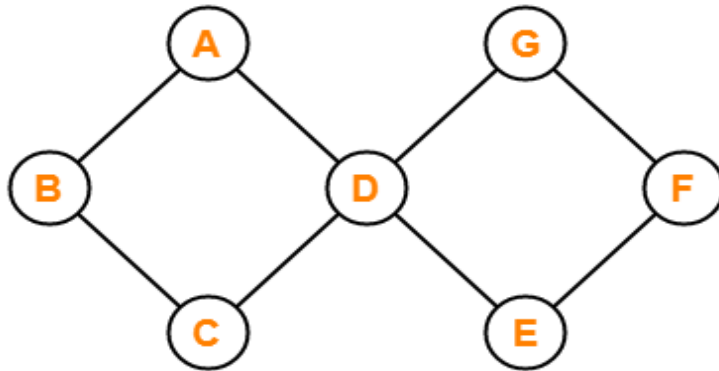
# Cyclic and Acyclic Graph

## ▶ Cyclic Graph:

- Graph contains a path from at least one node and back to itself .
- Graphs contain cycle/s.

## ▶ Acyclic Graph:

- Graph doesnot contain any cycle/s.



# Graph Terminologies

- ▶ Adjacent nodes: two nodes are adjacent if they are connected by an edge



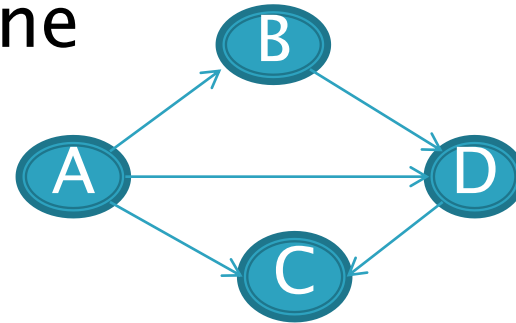
A is adjacent to B

B is adjacent from A

- ▶ Path: a sequence of vertices that connect two nodes in a graph. A sequence of vertices,  $p_0, p_1, \dots, p_m$ , such that each adjacent pair of vertices  $p_i$  and  $p_{i+1}$  are connected by an edge.
- ▶ The length of a path is the number of edges on it.
- ▶ Simple path: A path in which all vertices except possibly the first and last are distinct (No vertex is repeated and no cycle)
- ▶ A cycle is a simple path in which the first and last vertices are the same

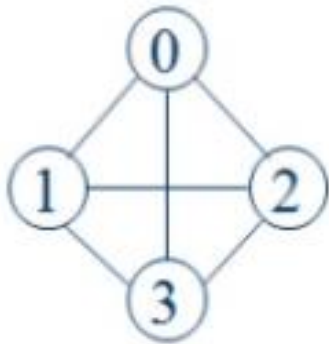
# Degree:

- ▶ The degree of a node is the number of edges the node is used to define
- ▶ In the example:
  - Degree 2: B and C
  - Degree 3: A and D
- ▶ A and D have odd degree, and B and C have even degree
- ▶ Can also define in-degree and out-degree
  - In-degree: Number of edges pointing to a node
  - Out-degree: Number of edges pointing from a node

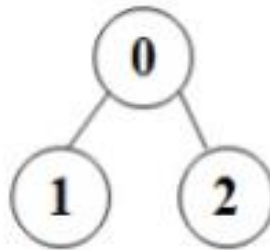


# Subgraph

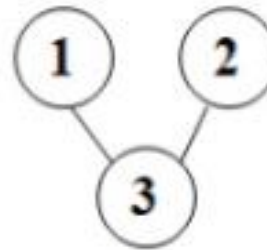
- ▶ If graph  $G=(V, E)$ 
  - Then Graph  $G'=(V',E')$  is a subgraph of  $G$  if  $V' \subseteq V$  and  $E' \subseteq E$  and



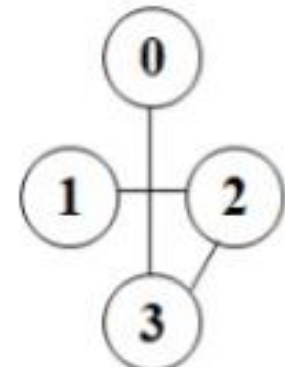
$G_1$



(i)



(ii)

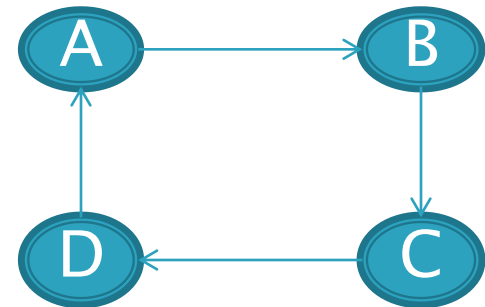


(iii)

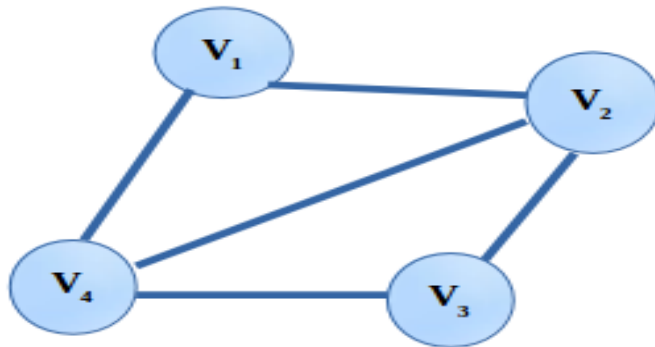
(a) Some of the subgraph of  $G_1$

# Connected and Unconnected Graphs and Connected Components

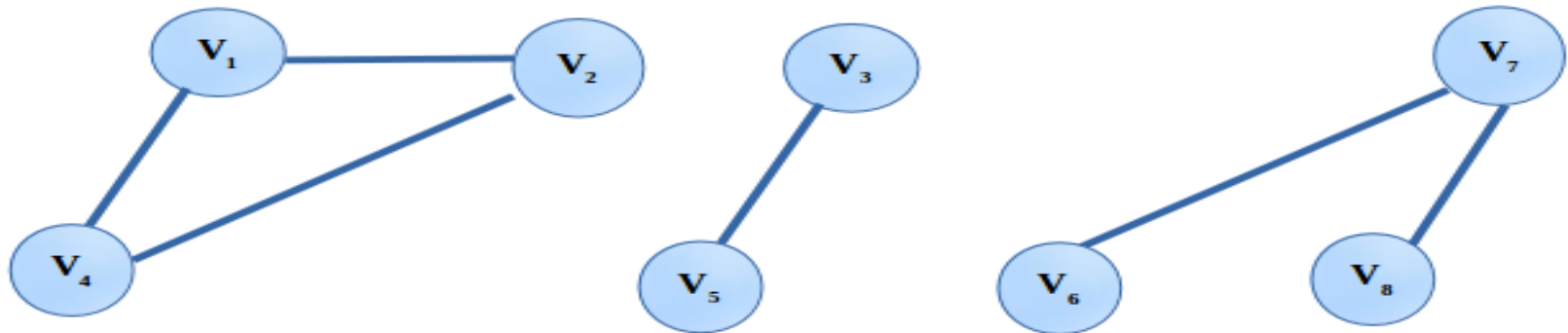
- ▶ An *undirected* graph is **connected** if every pair of vertices has a path between it
  - Otherwise it is unconnected
- ▶ An unconnected graph can be broken in to **connected components**
- ▶ A *directed* graph is **strongly connected** if every pair of vertices has a path between them, in **both directions**



# Connected and Unconnected Graphs and Connected Components



**Fig(i):  
Connected  
Graph**



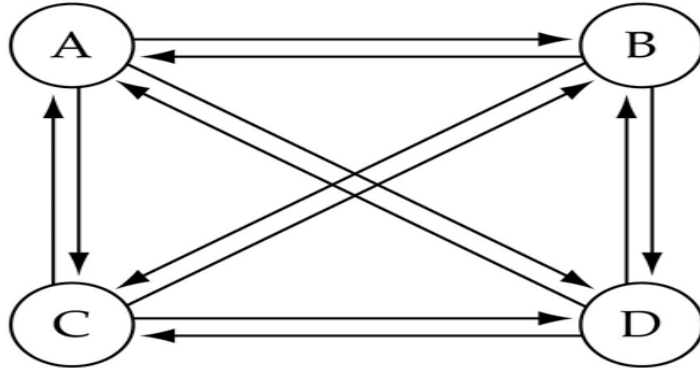
**Fig(ii):  
Unconnected Graph**

There are three component of above unconnected graph



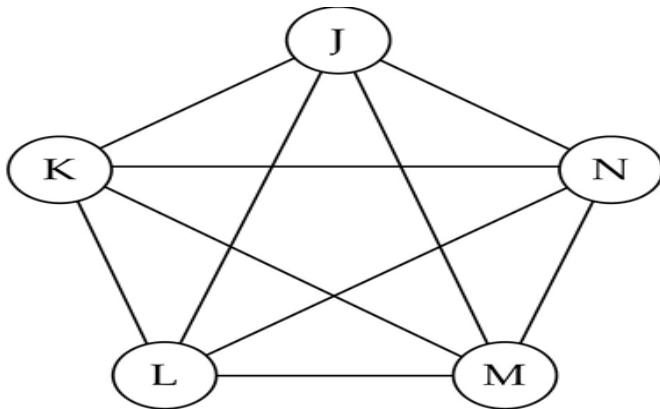
# Complete Graph: A graph in which every vertex is **directly connected** to **every other vertex**

- ▶ Complete directed graph:



$$\text{No of edges} = N * (N - 1)$$

- ▶ Complete undirected graph:

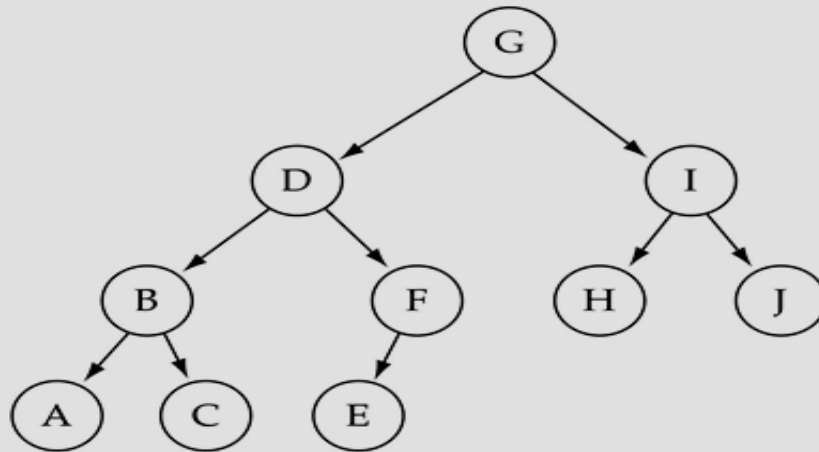


$$\text{No of edges} = N * (N - 1) / 2$$

# Tree Vs Graph

- ▶ Trees are special cases of graphs.
- ▶ Tree: **undirected/directed**, connected graph with **no cycles**

(c) Graph3 is a directed graph.



$V(\text{Graph3}) = \{ A, B, C, D, E, F, G, H, I, J \}$

$E(\text{Graph3}) = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$

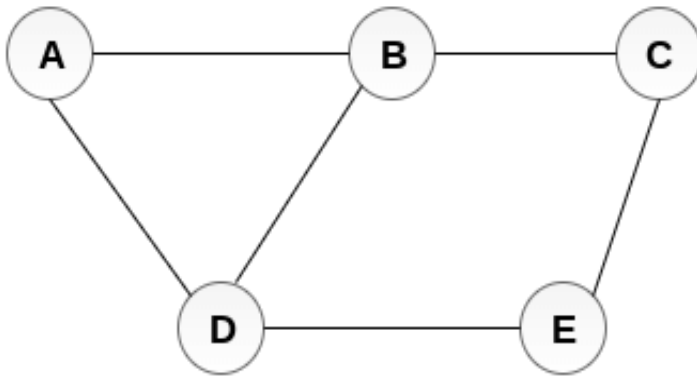
# Storage representations

- ▶ Several representations for graphs are possible.
  - Adjacency matrix,
  - adjacency list,
  - adjacency multi list,
  - inverse adjacency list
- ▶ The choice of a particular representation will depend upon the **application/situation** one has in mind and the **functions** one expects to perform on the graph.

# Adjacency Matrix

- ▶ Let  $G=(V,E)$  be a graph with  $n$  vertices,  $n \geq 1$ .
- ▶ The adjacency matrix of  $G$  is a 2-dimensional  $n \times n$  array, say  $A$ , with the property that  $A(i,j) = 1$  iff the edge  $(v_i,v_j)$  ( $\langle v_i,v_j \rangle$  for a directed graph) is in  $E(G)$ .
- ▶  $A(i,j) = 0$  if there is no such edge in  $G$ .
- ▶ The adjacency matrix for an undirected graph is symmetric as the edge  $(v_i,v_j)$  is in  $E(G)$  iff the edge  $(v_j,v_i)$  is also in  $E(G)$ .
- ▶ The adjacency matrix for a directed graph need not be symmetric.
- ▶ Undirected graphs can be store using upper or lower triangle of the matrix.

# Example:



Undirected Graph

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

Adjacency Matrix

Using adjacency matrices algorithm will require at least  $O(n^2)$  time as  $n^2 - n$  entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse then time can be reduce to  $O(e + n)$  where  $e$  is the number of edges using linked list.

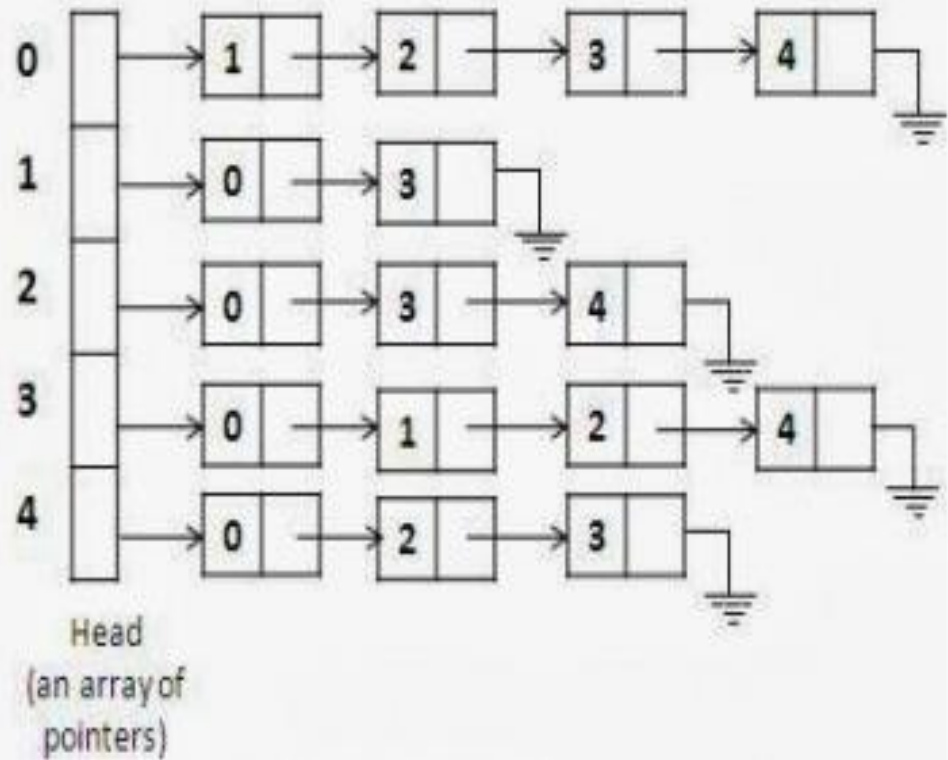
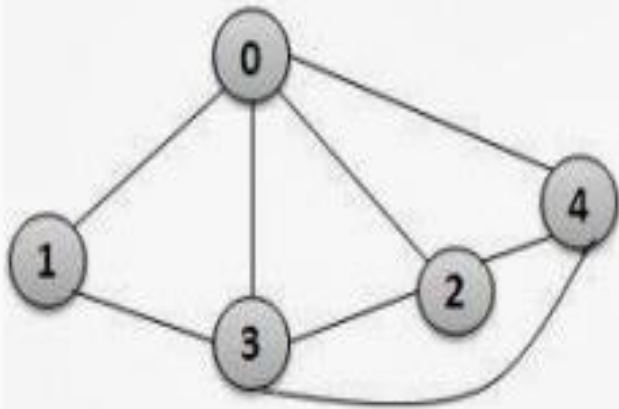
# Adjacency Lists

- ▶ For a sparse graph, we can use adjacency list for storage
  - If there are few edges, the adjacency matrix will contains many zero elements which consume much space and time
  - Complexity can be reduce to  $O(e+n)$  instead of  $O(n^2)$

# Adjacency Lists

- ▶ In this representation  $n$  rows of the adjacency matrix are represented as  $n$  linked lists. (One list for each vertex in  $G$ )
- ▶ The nodes in list  $i$  represent the vertices that are adjacent from vertex  $i$ .
- ▶ Each node has at least two fields: VERTEX and LINK. The VERTEX fields contain the indices of the vertices adjacent to vertex  $i$ .
- ▶ Each list has a headnode. The headnodes are sequential providing easy random access to the adjacency list for any particular vertex.
- ▶ Can be used for directed and undirected graphs

# Example





# Some points to observe

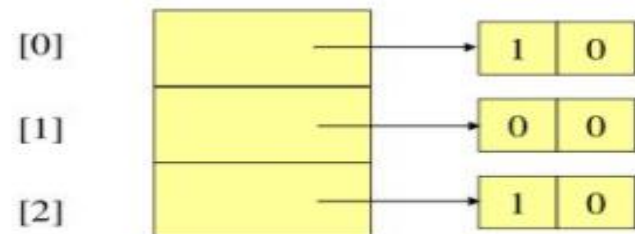
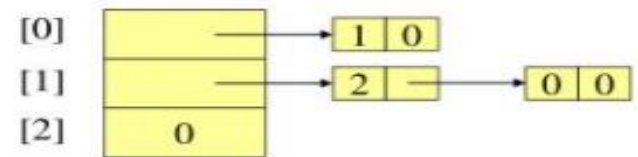
- ▶ The total number of edges in  $G$  may, therefore, be determined in time  $O(n + e)$
- ▶ In directed graph, number of nodes are  $e$  only.
- ▶ The degree of any vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list.
  - The out-degree of any vertex may be determined by counting the number of nodes on its adjacency list.
  - Determining the in-degree of a vertex is a little more complex, make it fast by keeping another set of lists called inverse adjacency lists.

# Storage units

- ▶ Undirected graph with  $n$  vertices and  $e$  edges
  - Need  $(n + 2e)$  storage units.
- ▶ Digraph with  $n$  vertices and  $e$  edges.
  - Need  $(n + e)$  storage units.

# Inverse Adjacency Lists

- ▶ Useful for finding in-degree of a vertex in directed graph
- ▶ Contain one list of each vertex
- ▶ Each list contains a node for each vertex adjacent to the vertex that the list represents



# Adjacency Multi List

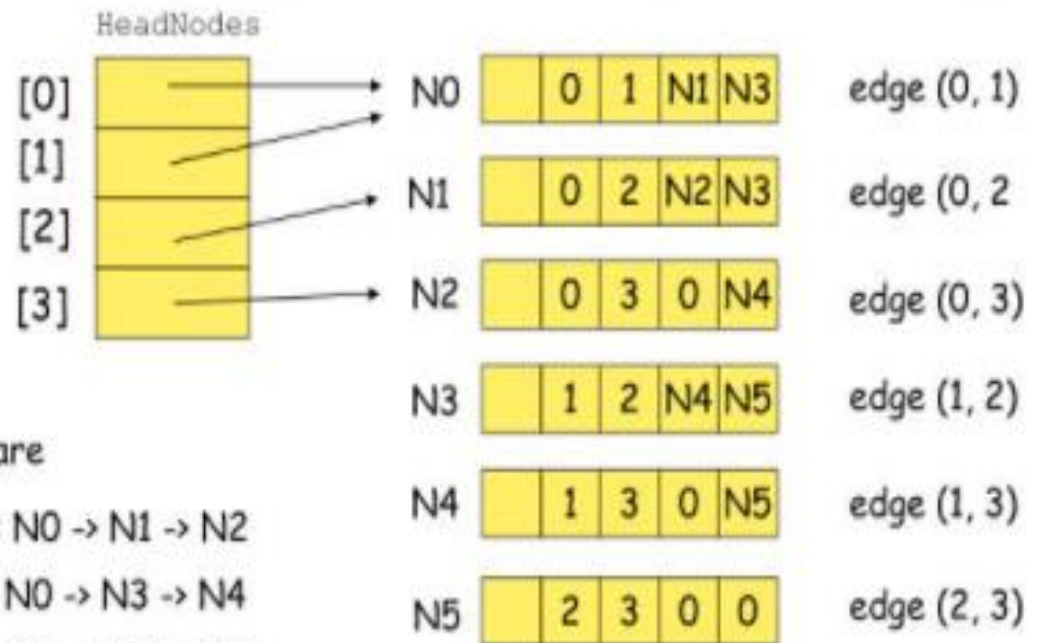
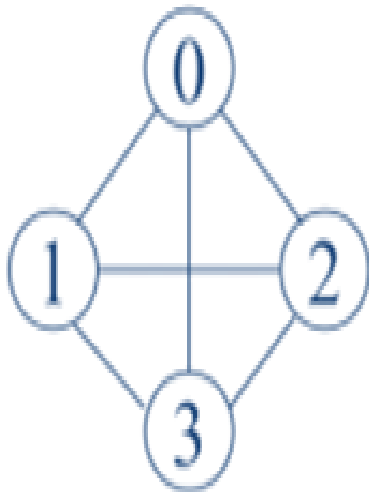
- ▶ Edge based representation than vertex based.
- ▶ An edge  $(V_i, V_j)$  in an undirected graph is represented by two nodes in adjacency list representation of  $V_i$  and  $V_j$ .
- ▶ In some situation, it is necessary to determine second entry of that particular edge and mark it as that edge is already examined.
- ▶ This is accomplished by Adjacency Multi lists in which nodes may be shared among several lists
- ▶ For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident).

- ▶ The node structure become

M	V1	V2	Link 1 for V1	Link 2 for V2
---	----	----	------------------	------------------

- ▶  $M$  is a one bit mark field that may be used to indicate whether or not the edge has been examined.
- ▶ The storage requirements are the same as for normal adjacency lists except for the addition of the mark bit  $M$ .

# Example



The lists are

Vertex 0: N0 → N1 → N2

Vertex 1: N0 → N3 → N4


Vertex 2: N1 → N3 → N5

Vertex 3: N2 → N4 → N5

# Operations on Graph

- ▶ The most common graph operations are:
  - Graph Traversal (visiting nodes in some order)
  - Add and delete elements(vertex, edges) to graph
  - Finding the path from one vertex to another (specially shortest path)
- ▶ Most common operation is traversal.
- ▶ If all vertices are connected then we are interested in visiting all vertices in  $G$  that are reachable from  $v$ .
  - *DFS*
  - *BFS*

# DFS

- ▶ The start vertex  $v$  is visited.
  - ▶ Next an unvisited vertex  $w$  adjacent to  $v$  is selected and a depth first search from  $w$  initiated.
  - ▶ When a vertex  $u$  is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex  $w$  adjacent to it and initiate a depth first search from  $w$ .
  - ▶ The search terminates when no unvisited vertex can be reached from any of the visited one
  - ▶ This procedure is best described recursively
- 



# DFS Algorithm

**procedure** *DFS*(*v*)

// Given an undirected graph  $G = (V, E)$  with  $n$  vertices and an array *VISITED*( $n$ ) initially set to zero, this algorithm visits all vertices reachable from *v*. *G* and *VISITED* are global. //

*VISITED*(*v*) := 1 // print the *v*

**for** *each vertex w adjacent to v* **do**

**if** *VISITED*(*w*) = 0 **then** call *DFS*(*w*)

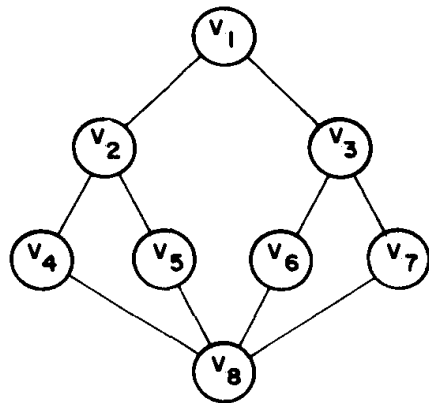
**end**

**end** *DFS*



Algorithm DFS-iterative (G, s) //Where G is graph and s is source vertex

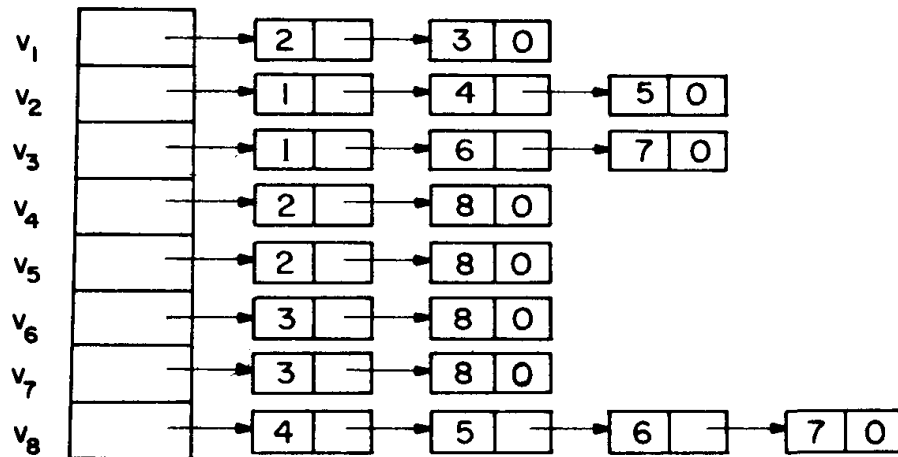
- ▶ S.push( s ) // Let S be the stack and Inserting s in stack
- ▶ mark s as visited.
- ▶ while ( S is not empty):
  - ▶ v = S.top( ) //Pop a vertex from stack to visit next , print v
  - ▶ S.pop( )
  - ▶ //Push all the neighbours of v in stack that are not visited
  - ▶ for all neighbours w of v in Graph G:
    - ▶ if w is not visited :
      - ▶ S.push( w )
      - ▶ mark w as visited



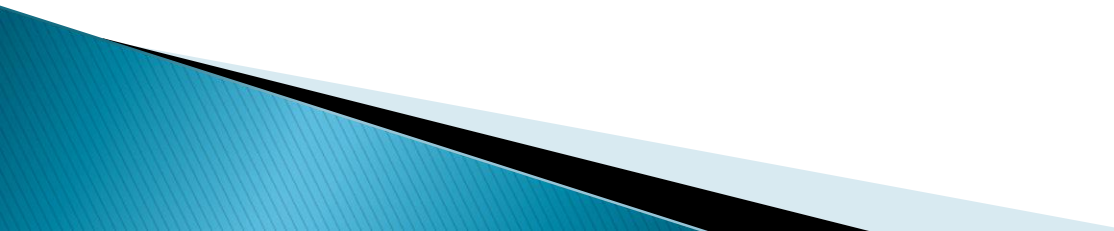
(a)

$v_1, v_2, v_4, v_8, v_5, v_6, v_3, v_7$ .

One may easily verify that DFS ( $v_1$ ) visits all vertices connected to  $v_1$ . So, all the vertices visited, together with all edges in  $G$  incident to these vertices form a connected component of  $G$



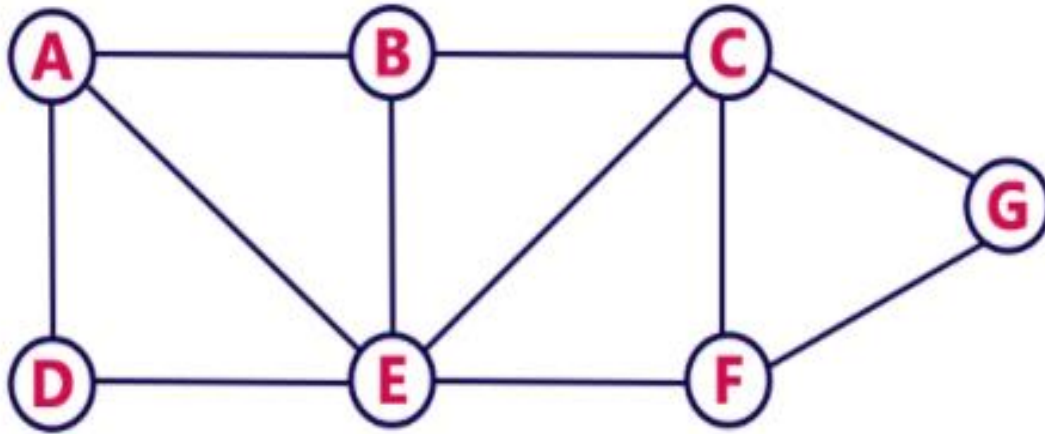
# BFS

- ▶ Starting at vertex  $v$  and marking it as visited,
  - ▶ All unvisited vertices adjacent to  $v$  are visited next.
  - ▶ Then unvisited vertices adjacent to these vertices are visited and so on.
- 

# BFS

- ▶ **procedure** *BFS*(*v*)
- ▶ //A breadth first search of *G* is carried out beginning at vertex *v*. All vertices visited are marked as *VISITED*(*i*)= 1. The graph *G* and array *VISITED* are global and *VISITED* is initialized to zero.//
- ▶ *VISITED* (*v*) :=1
- ▶ *initialize Q to be empty* // *Q* is a queue//
- ▶ **loop**
- ▶ **for** all vertices *w* adjacent to *v* **do**
- ▶ **if** *VISITED*(*w*) = 0 //add *w* to queue//
- ▶ **then** *ADDQ*(*w*,*Q*); *VISITED*(*w*) :=1 //mark *w* as *VISITED*//
- ▶ **end**
- ▶ **if** *Q is empty* **then return**
- ▶ **call** *DELETEQ*(*v*,*Q*)
- ▶ **forever**
- ▶ **end** *BFS*

# Example



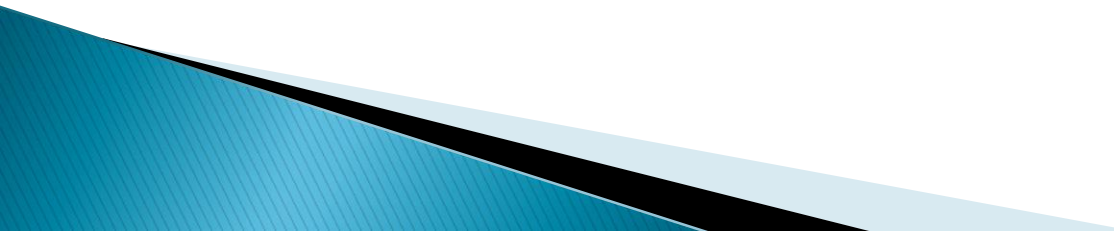
Answer: A, B, D, E, C, F, G

# DFS Vs BFS

- ▶ In DFS, we can determine whether path exists between two nodes  $x$  and  $y$  by looking at children of starting node and recursively determine if a path exists.
- ▶ In BFS, we traversal through a graph one level of children at a time.
- ▶ In DFS, we travel down a single path, one child node at a time and stick to one path
- ▶ BFS is good to find shortest path as we evaluate all possible node from a given node.
- ▶ It's possible to run BFS recursively without any data structures, but with higher complexity. DFS, as opposed to BFS, uses a stack instead of a queue, and so it can be implemented recursively

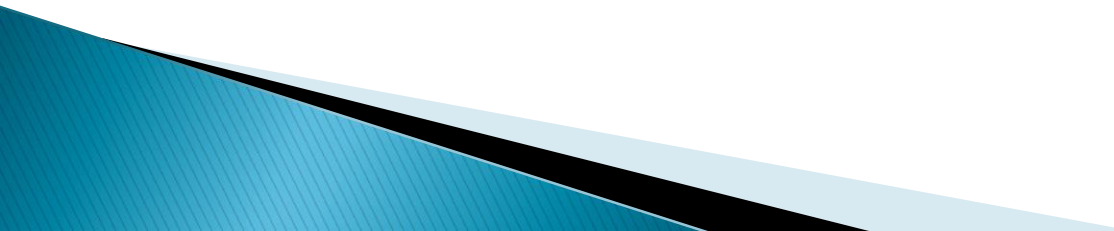
# Greedy Strategy

## Basic idea

- ▶ Find the optimal solution piece by piece.
  - ▶ Iterative make decision one by one and hope everything works out well at the end.
  - ▶ Once the decision is made then choice should not get changed for the subsequent steps.
- 



# Optimization Problems

- ▶ For most optimization problems you want to find, not just *a* solution, but the *best* solution.
  - ▶ A *greedy algorithm* sometimes works well for optimization problems. It works in phases. At each phase:
    - You take the best you can get right now, without regard for future consequences.
    - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.
- 

# General principal

- ▶ The choice that seems best at the moment is the one we go with.

# Example: Counting Money

- ▶ Suppose you want to count out a certain amount of money, using the **fewest possible bills and coins**
- ▶ A greedy algorithm to do this would be:  
At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make \$6.39, you can choose:
    - a \$5 bill
    - a \$1 bill, to make \$6
    - a 25¢ coin, to make \$6.25
    - A 10¢ coin, to make \$6.35
    - four 1¢ coins, to make \$6.39
- ▶ For US money, the greedy algorithm always gives the optimum solution

# Greedy Algorithm Failure

- ▶ In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- ▶ Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- ▶ A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- ▶ The greedy algorithm results in a solution, but not in an optimal solution

# Greedy Algorithms

- The most straight forward design technique; can be applied to a wide range of problems
- Most of these problems have  $n$  inputs and require us to obtain a subset that satisfies some constraints
- Any subset that satisfies these constraints is called a **feasible solution**
- we need to find a feasible solution that either maximizes or minimizes a given **objective function**
- A feasible solution that does this is called an **optimal solution**

## Greedy Algorithms contd..

- The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time
- At each stage, a decision is made regarding whether a particular input is in an optimal solution
- This is done by considering inputs in an order determined by some selection procedure
- If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution; otherwise it is added
- The selection procedure itself is based on some optimization measure (objective function)



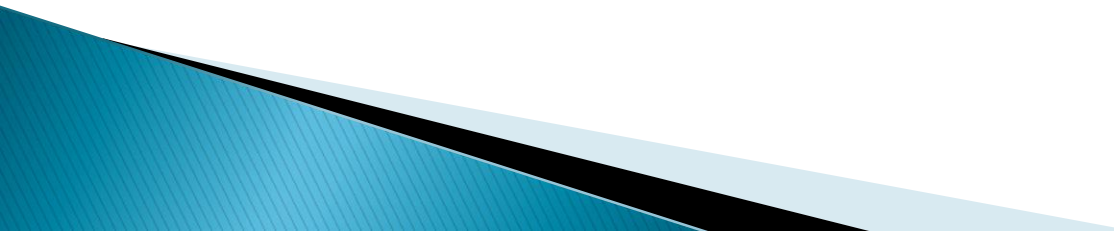
# Control Abstraction

Greedy method control abstraction – subset paradigm

```
Algorithm Greedy(a,n) //a[1..n] contains n inputs
  solution =  $\Phi$  ; //initialize the solution
  for i=1 to n do
  {
    x=Select(a); //selecting an input from a
                  and removing it
    if Feasible(solution, x) then
      solution=Union(solution, x); //updates the
  }
  return solution;
```

Feasible fun: Used to determine whether a candidate can be used to contribute to the solution.

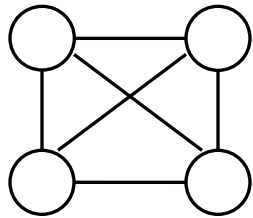
# Important terms

- ▶ Note terms :-
  - ▶ Feasible solution
  - ▶ Optimal solution
  - ▶ Objective function
- 

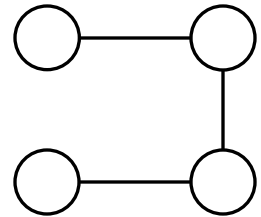
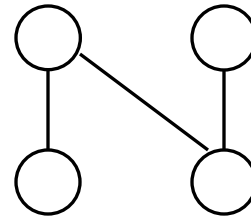
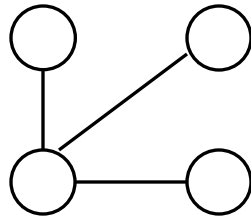
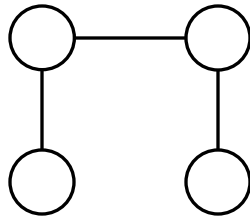


# Spanning trees

- ▶ Suppose you have a connected undirected graph
- ▶ Then a spanning tree of the graph is a **connected** (if there is at least one path between every pair of vertices in a graph) **subgraph** in which there are no cycle.



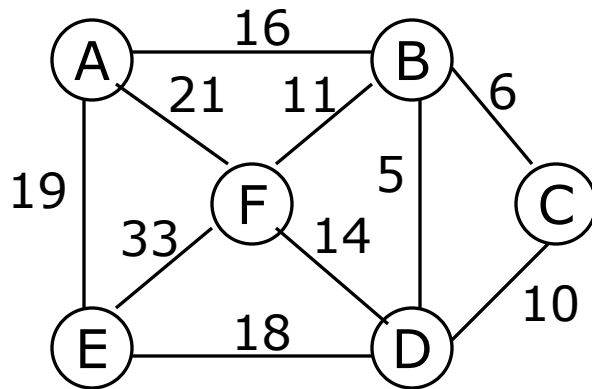
A connected,  
undirected graph



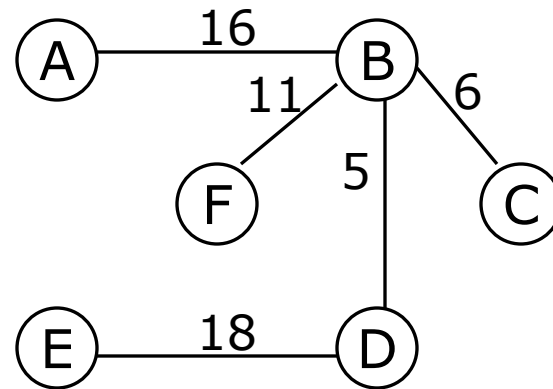
Four of the spanning trees of the graph

# Minimum-cost spanning trees

- ▶ Suppose you have a connected undirected graph with a weight (or cost) associated with each edge
- ▶ The cost of a spanning tree would be the sum of the costs of its edges
- ▶ A minimum-cost spanning tree is a spanning tree that has the lowest cost



A connected, undirected graph



A minimum-cost spanning tree

# Finding spanning trees

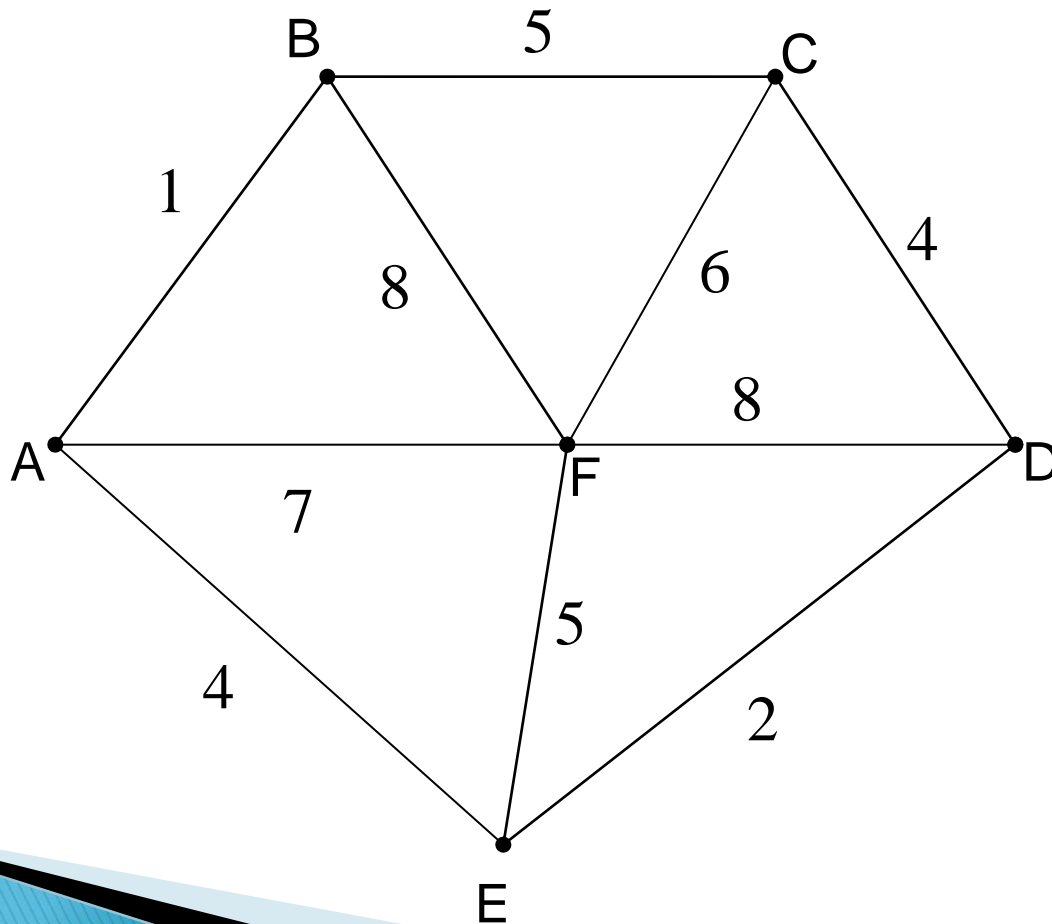
- ▶ There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms
- ▶ **Kruskal's algorithm:** Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle
- ▶ **Prim's algorithm:** Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.

# Minimum Connector Algorithms

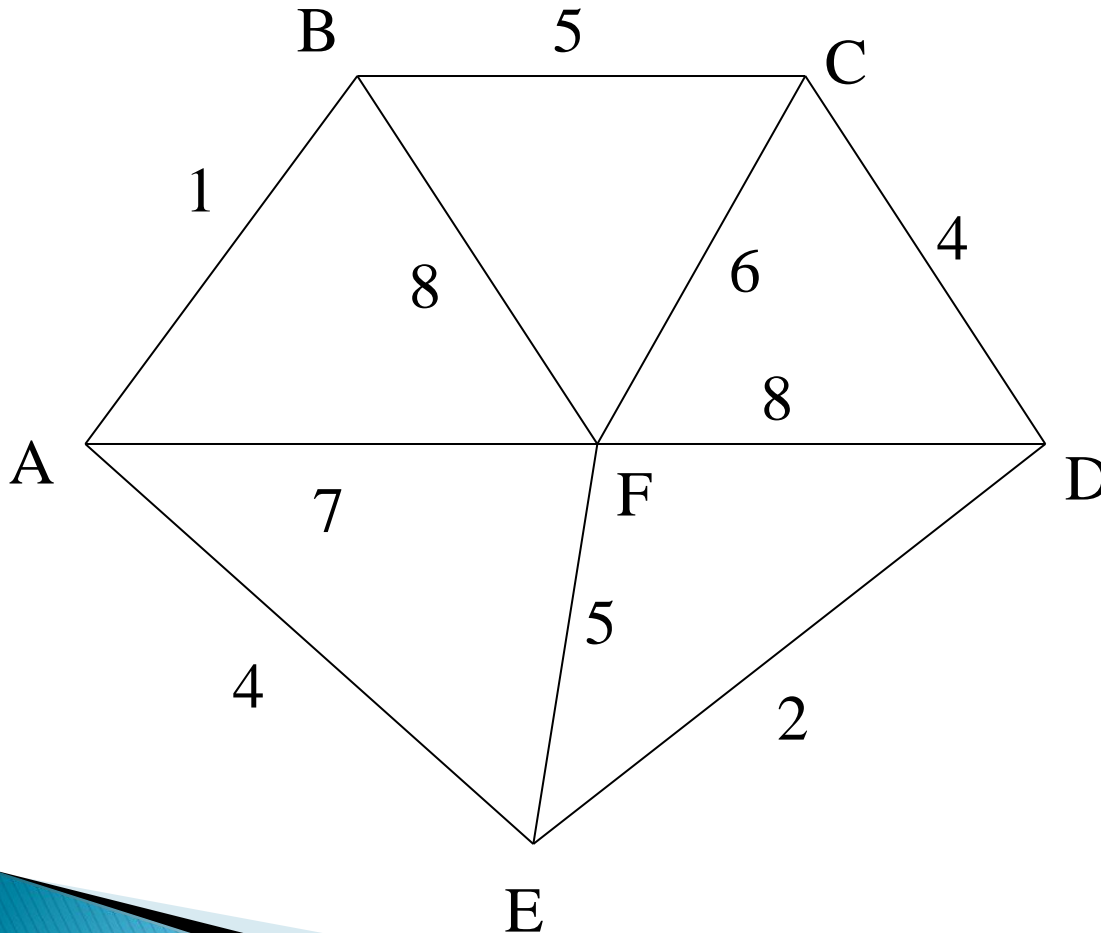
## Kruskal's algorithm

1. Select the shortest edge in a network
2. Select the next shortest edge which does not create a cycle
3. Repeat step 2 until spanning tree has  $n-1$  edges.

We model the situation as a network, then the problem is to find the minimum connector for the network



# Kruskal's Algorithm



List the edges in  
order of size:

AB 1

ED 2

AE 4

CD 4

BC 5

EF 5

CF 6

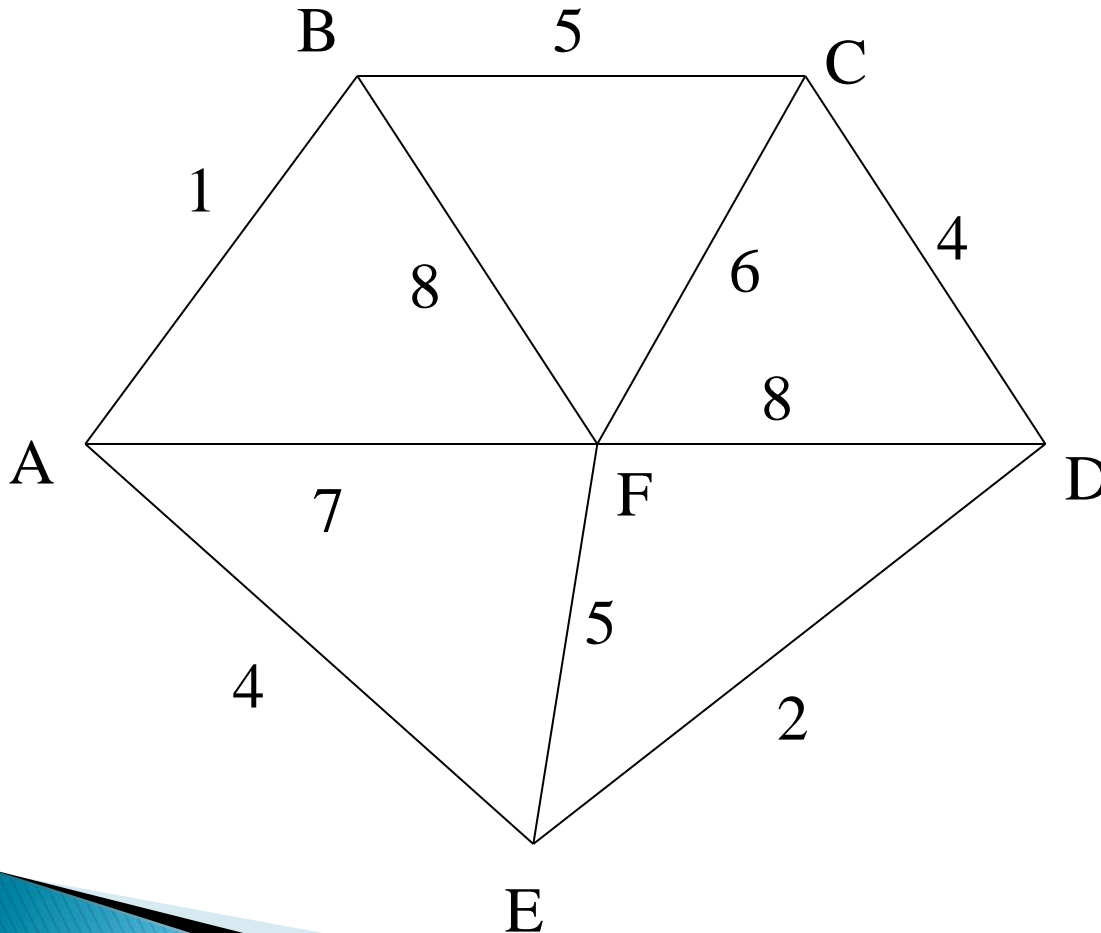
AF 7

BF 8

CF 8

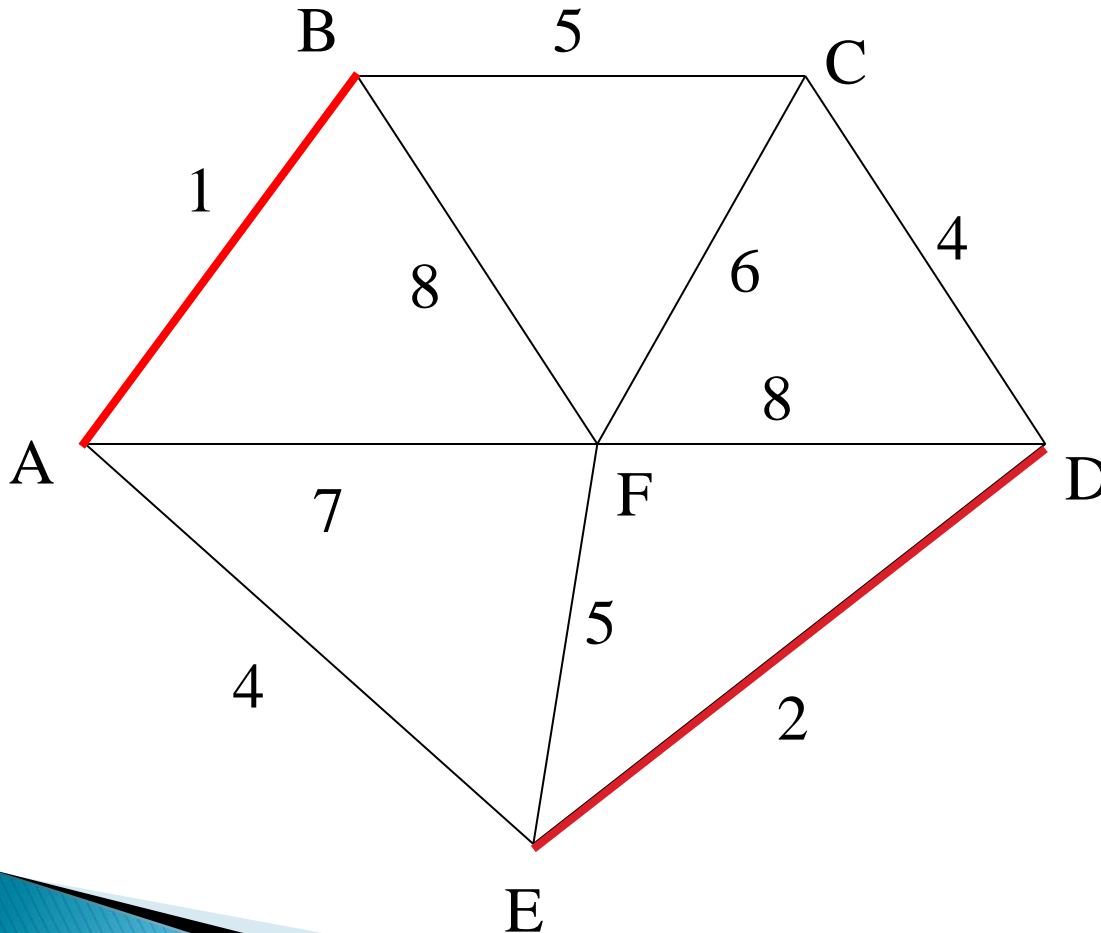
# Kruskal's Algorithm

Select the shortest edge in the network



**AB 1**

# Kruskal's Algorithm



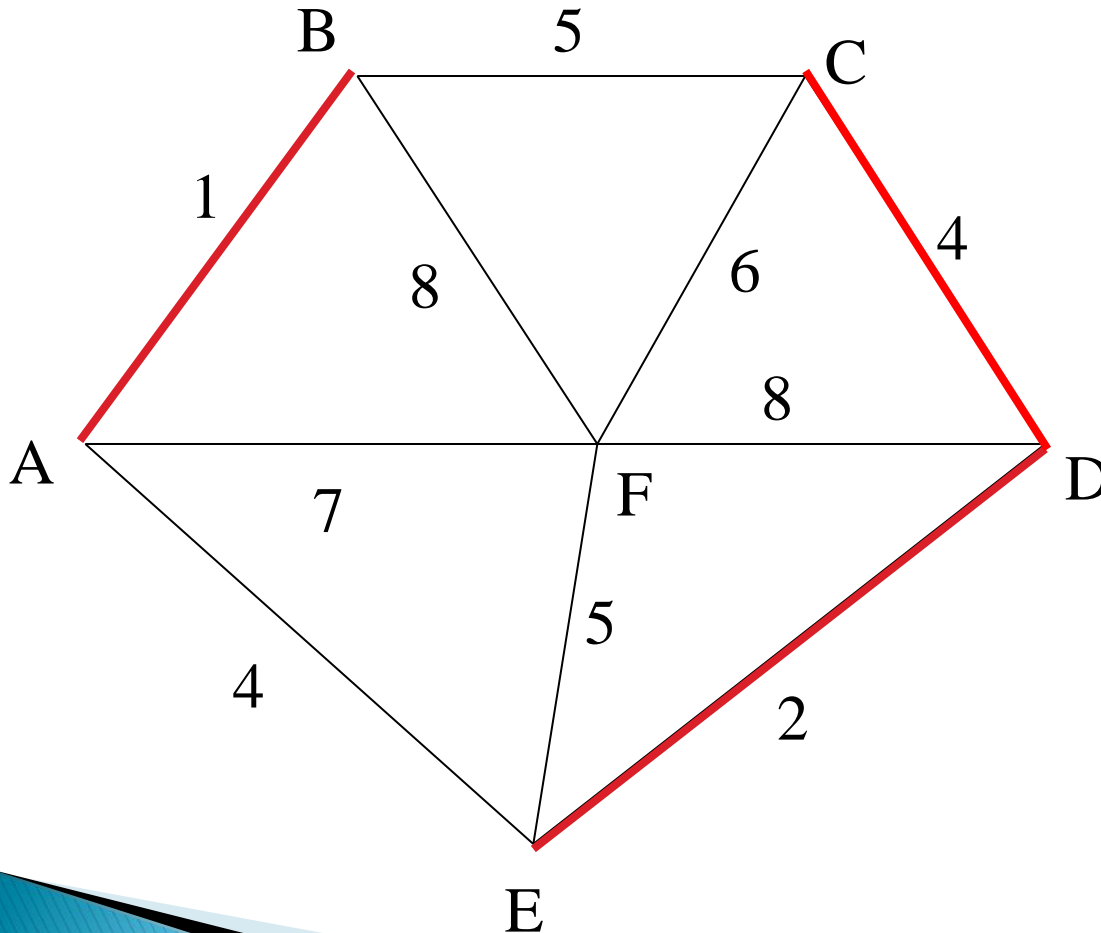
Select the next shortest edge which does not create a cycle

**AB 1**

**ED 2**



# Kruskal's Algorithm



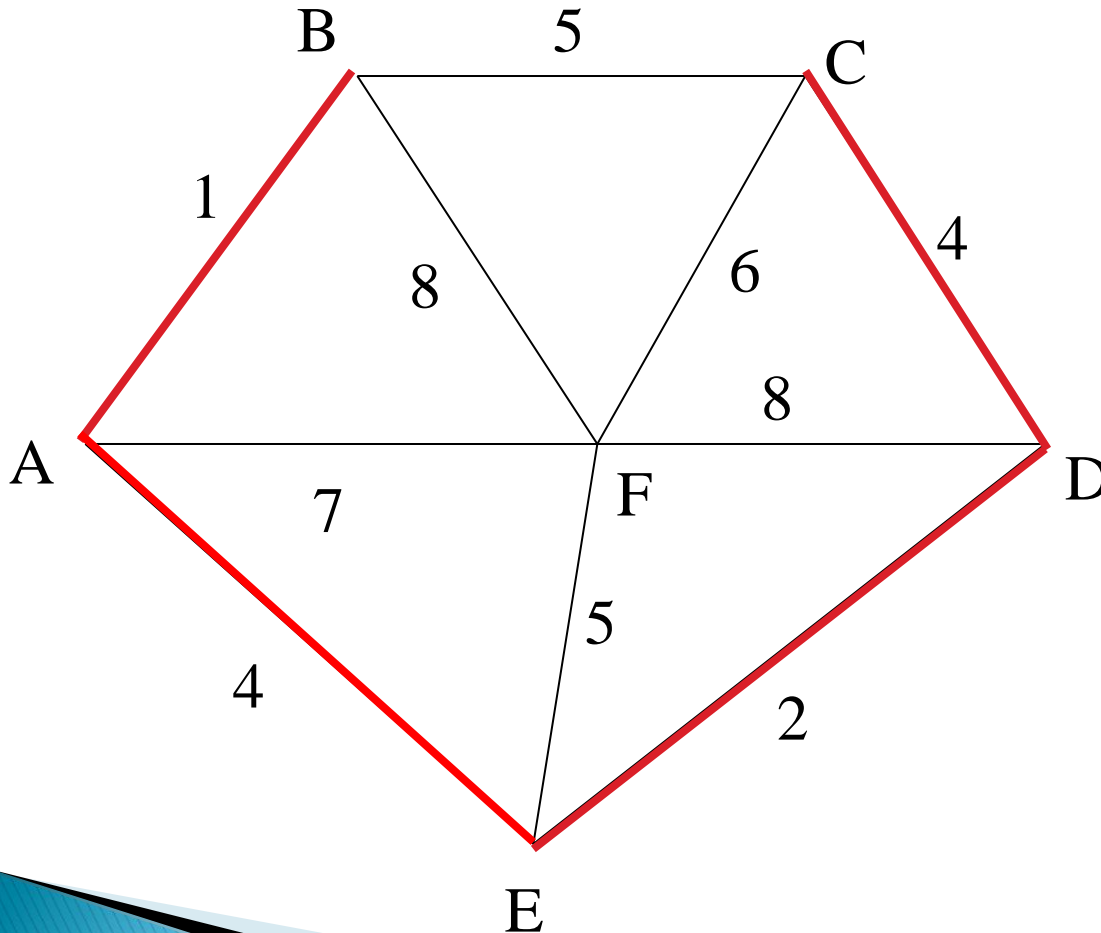
Select the next shortest edge which does not create a cycle

**AB 1**

**ED 2**

**CD 4 (or AE 4)**

# Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

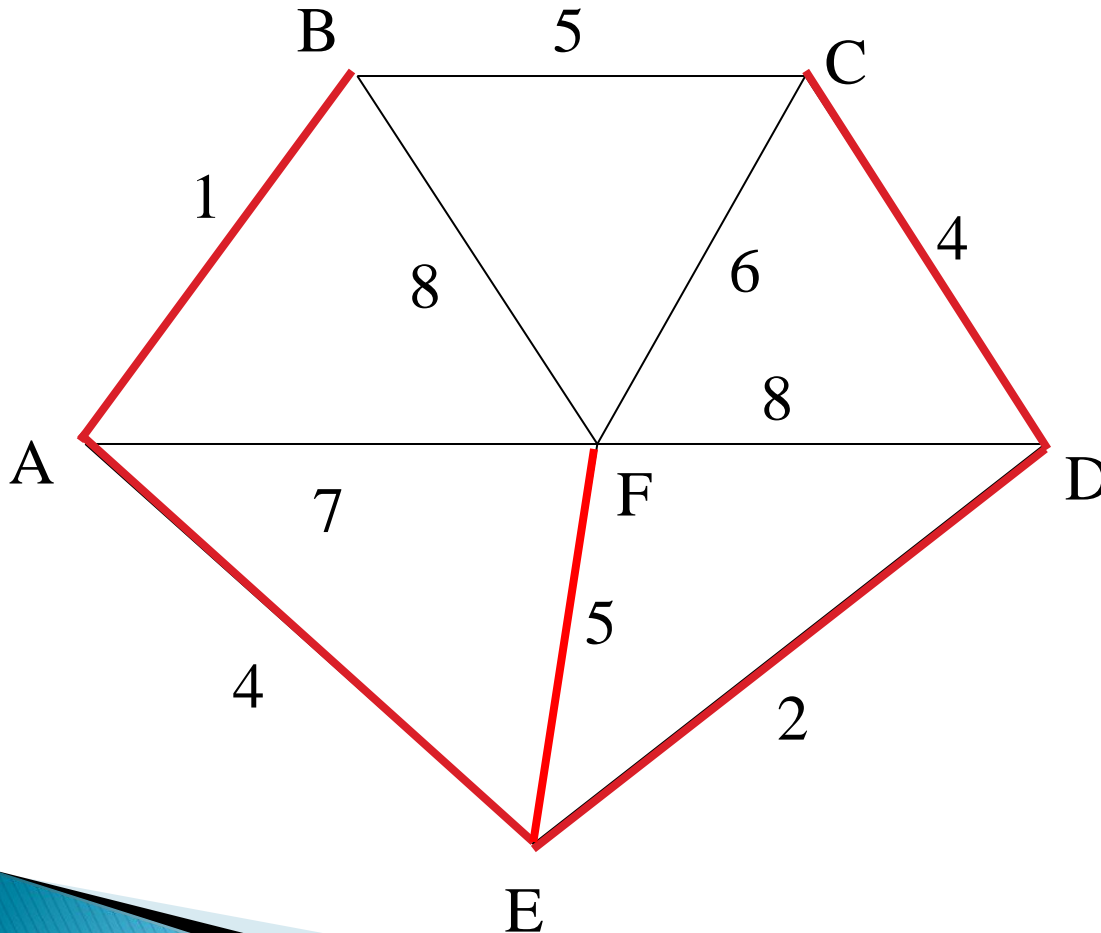
**AB 1**

**ED 2**

**CD 4**

**AE 4**

# Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

**AB 1**

**ED 2**

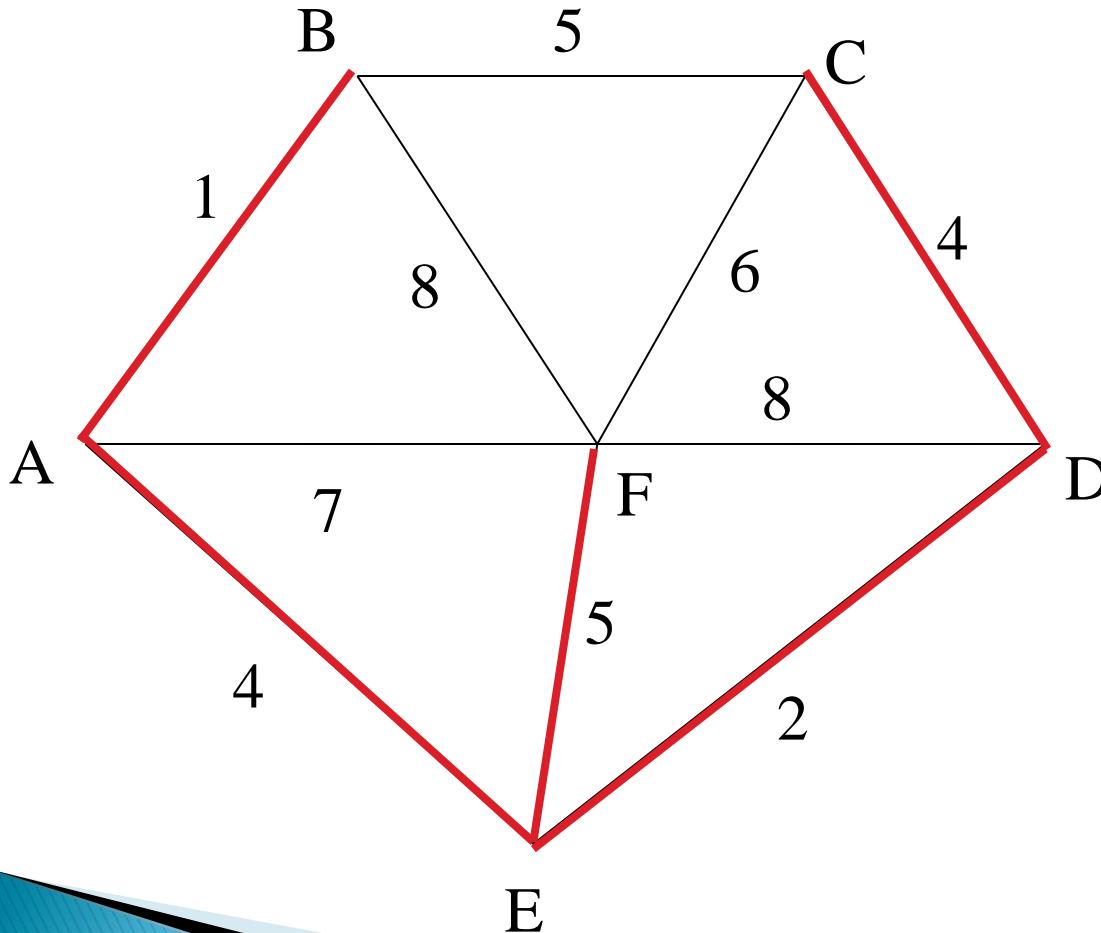
**CD 4**

**AE 4**

**BC 5 – forms a cycle**

**EF 5**

# Kruskal's Algorithm



All vertices have been connected.

The solution is

**AB 1**

**ED 2**

**CD 4**

**AE 4**

**EF 5**

Total weight of tree: 16

# KRUSKAL'S

Algorithm  $\text{kruskal}(G, V, E, T)$

{

1. Sort  $E$  in increasing order of weight

2. let  $G=(V, E)$  and  $T=(A, B)$ ,  $A=V$ ,  $B$  is null set and let  $n = \text{count}(V)$

3. Initialize  $n$  set, each containing a different element of  $v$ .  
for each  $v \in V$   
make\_disjoint\_set( $v$ )

4. while( $|B| < n-1$ ) do

begin

$e = \langle u, v \rangle$  the shortest edge  $\in E$  not yet considered

$U = \text{find\_set}(u)$

$V = \text{find\_set}(v)$

if ( $U \neq V$ ) {

$B = B \cup \{ (u, v) \}$  // update the edge in  $B$  and add the cost  
 $\text{union}(u, v)$  // merge two sets

}

end

5. return  $T$ ; //  $T$  is the minimum spanning tree

}

# complexity

- ▶  $O(|E|\log|E|)$

# Prim's algorithm

## Prim's algorithm

1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

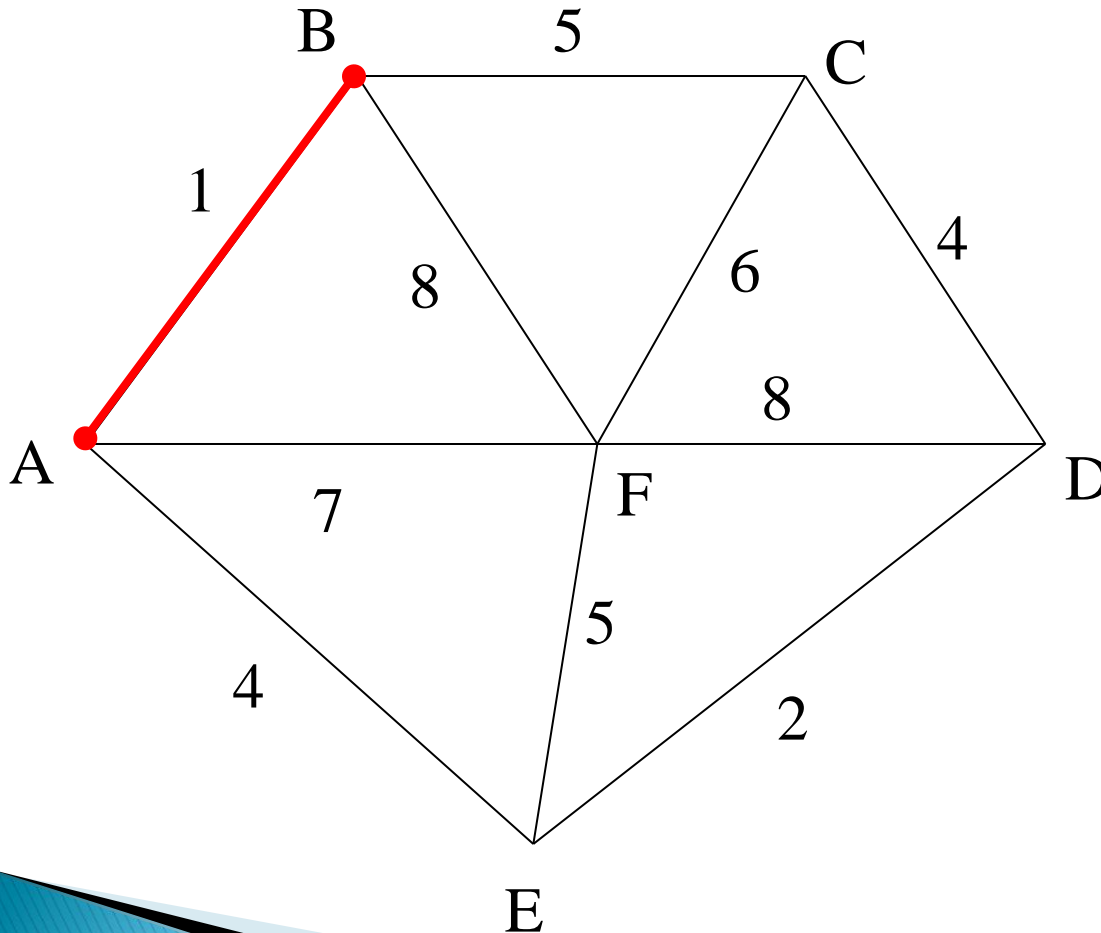
# Prim's Algorithm

Select any vertex

A

Select the shortest edge connected to that vertex

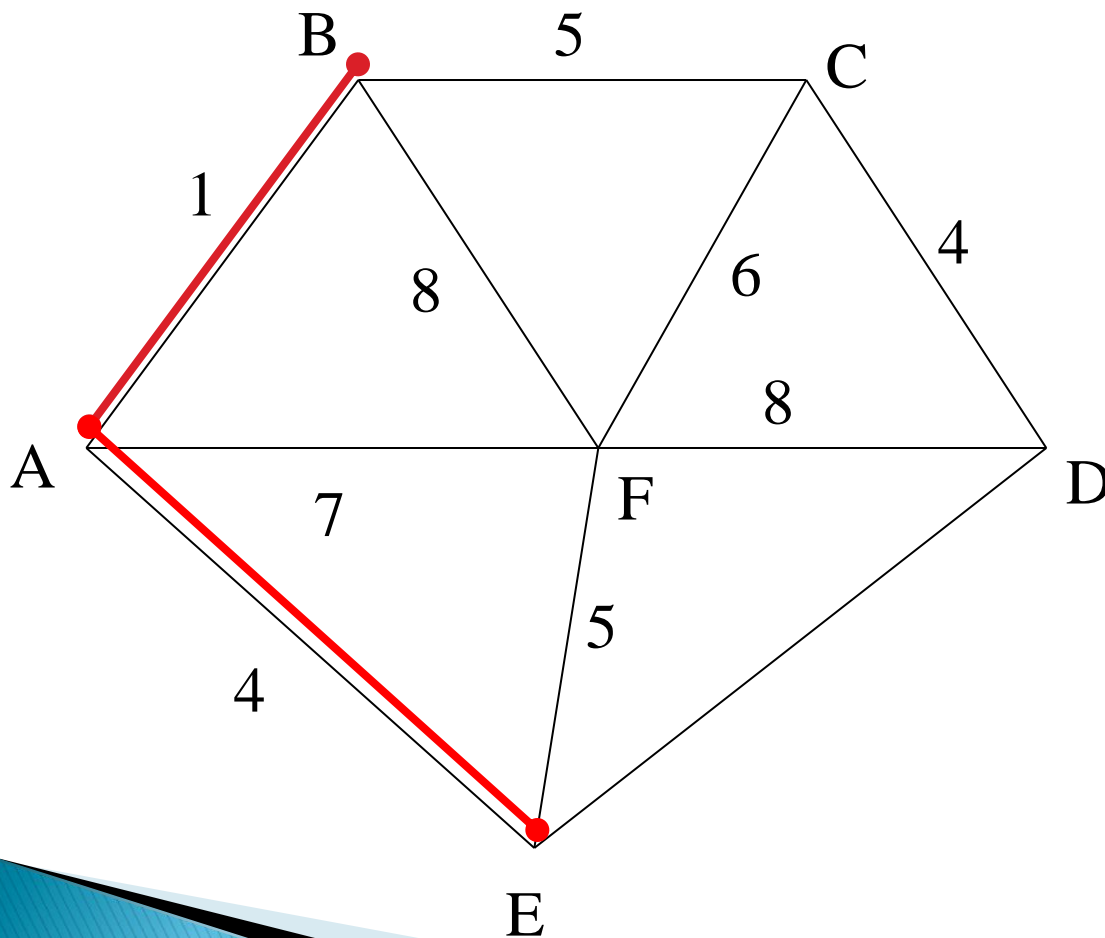
AB 1





# Prim's Algorithm

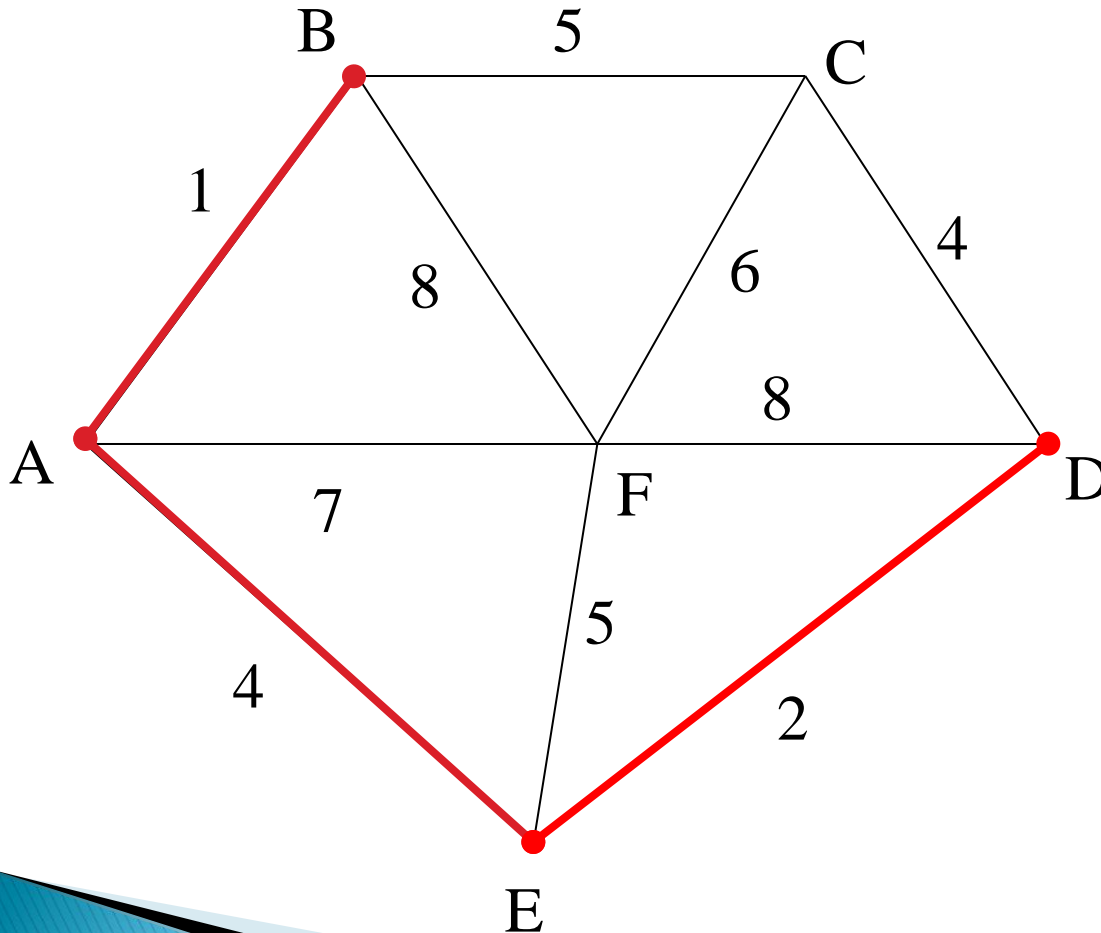
Select the shortest edge connected to any vertex already connected.



AE 4

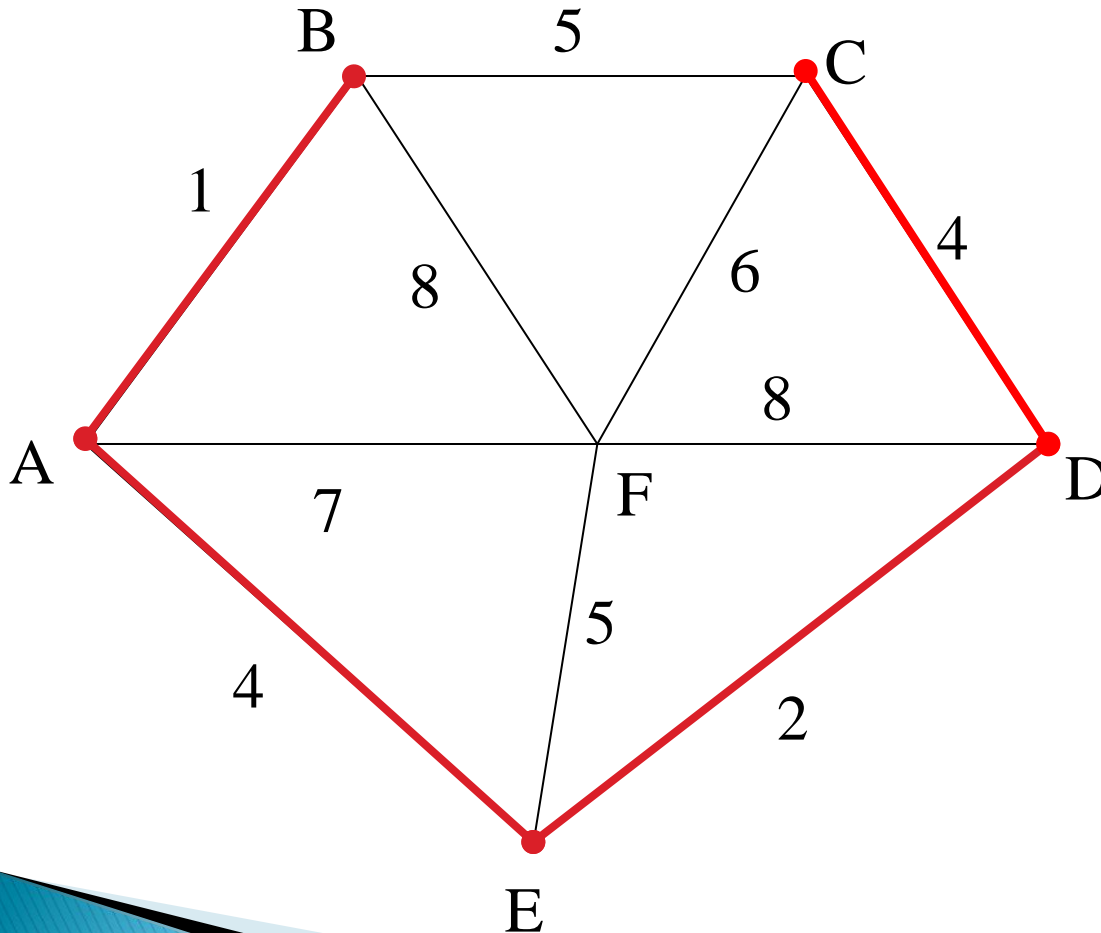
# Prim's Algorithm

Select the shortest edge connected to any vertex already connected.



ED 2

# Prim's Algorithm

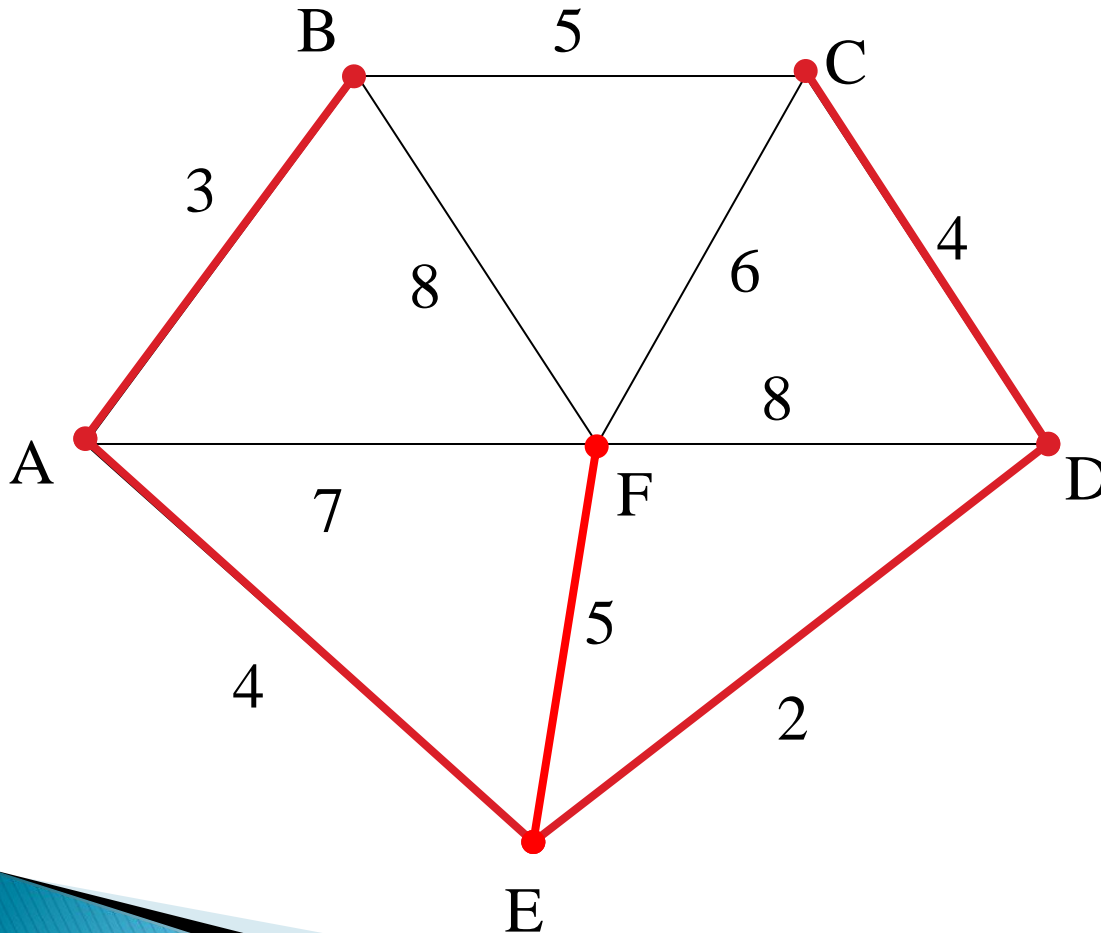


Select the shortest edge connected to any vertex already connected.

DC 4

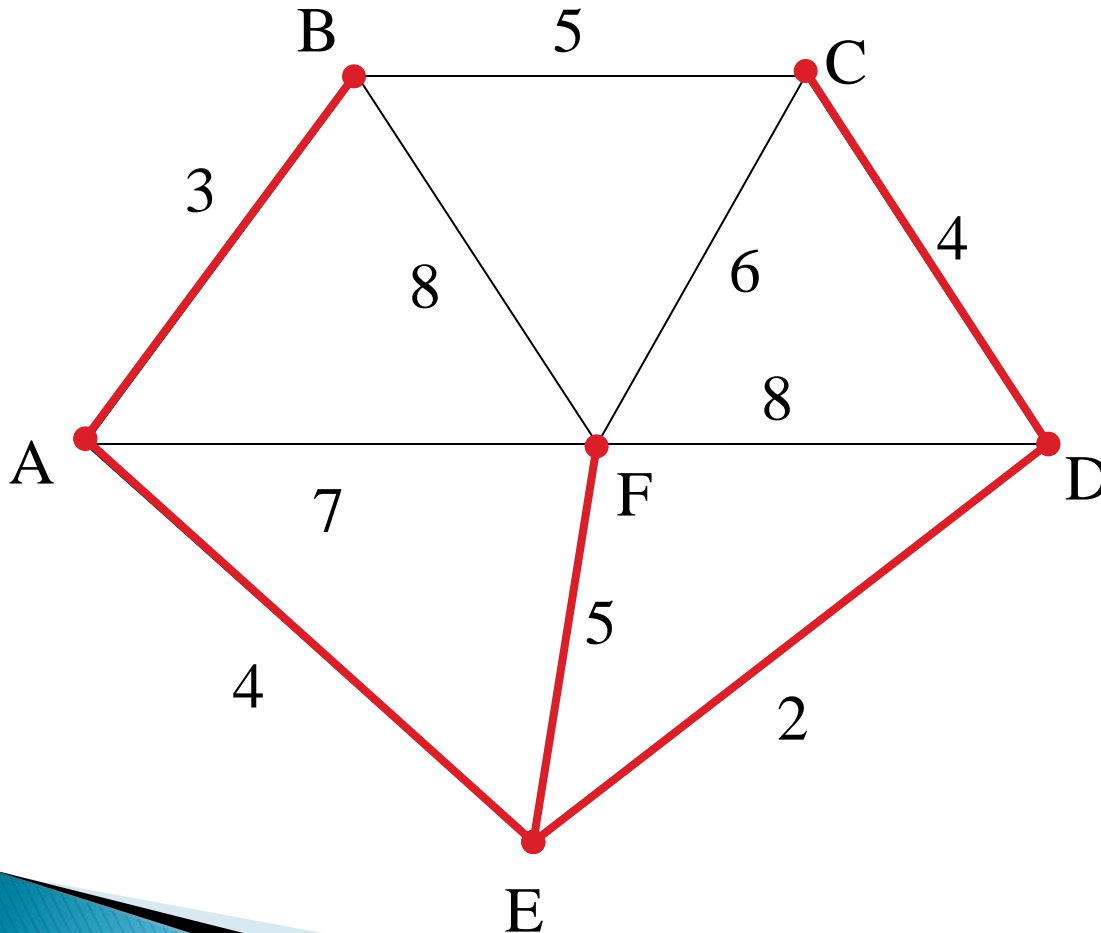
# Prim's Algorithm

Select the shortest edge connected to any vertex already connected.



EF 5

# Prim's Algorithm



All vertices have been connected.

The solution is

**AB 1**

**AE 4**

**ED 2**

**DC 4**

**EF 5**

Total weight of tree: 16

# Prims Algorithm:

**ReachSet = {0};**

**UnReachSet = {1, 2, ..., N-1};**

**SpanningTree = {};**

**while ( UnReachSet  $\neq$  empty )**

**{**

**Find edge  $e = (x, y)$  such that:**

**1.  $x \in \text{ReachSet}$**

**2.  $y \in \text{UnReachSet}$**

**3.  $e$  has smallest cost**

**SpanningTree = SpanningTree  $\cup$  { $e$ };**

**ReachSet = ReachSet  $\cup$  { $y$ };**

**UnReachSet = UnReachSet - { $y$ };**

**}**

# Prims Algorithm

Procedure prims(int v1)

int count1 = 1, count = 0, A[20], min = 9999, father[20] = {-1,-1,-1....-1}, wt = 0;

Int Res[20[20];

A[count1] = v1; count1++; //reached set

while(count < n-1)

Min = 9999

for(v1 = 1; v1 < count1; v1++)

for(v2 = 1; v2 < n; v2++)

if( G[A[v1]][v2] != 0)

min = G[A[v1]][v2];

T1 = A[v1]; t2 = v2

Endif

End for

End for

Temp1 = t1 ; temp2 = t2;

# Prims continued.....

```
G[t1][t2] = G[t2][t1] = 0;
```

```
while(t1>=0)
```

```
root_temp1 = t1; t1 = father[t1]
```

```
while(t2>=0)
```

```
root_temp2 = t2; t2 = father[t2]
```

```
if(root_temp1!=root_temp2)
```

```
    Result[temp1][temp2] = Result[temp2][temp1] = min;
```

```
    Wt = wt+ Result[temp1][temp2]
```

```
    father[root_temp2] = root_temp1;
```

```
    Count++;
```

```
    A[count1] = t2;
```

```
    Count1++
```

```
Endif
```

```
endwhile
```

```
End prims
```



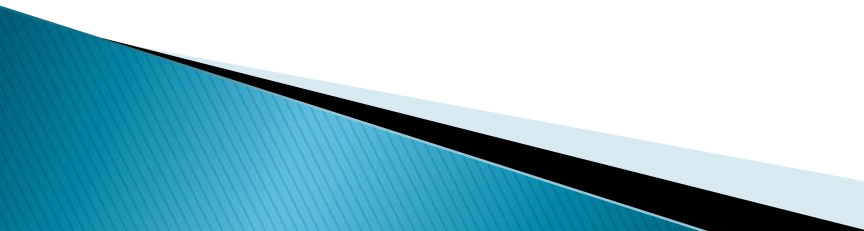
# COMPLEXITY

- ▶  $O(V^2)$

# Kruskal Algorithm

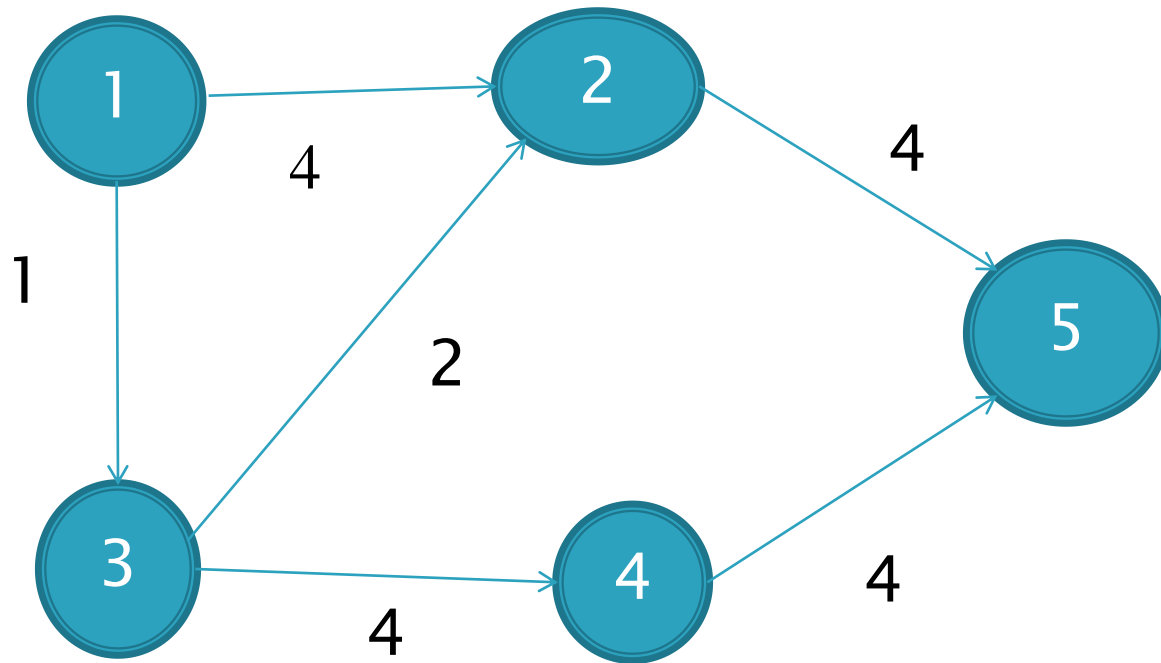
```
Procedure kruskal(int v1)
int count = 0, min = 9999, father[20] = {-1,-1,-1....-1}, wt = 0;
Int Res[20][20];
while(count < n-1)
Min = 9999
for(v1 = 0; v1 < n; v1++)
    for(v2 = 0; v2 < n; v2++)
        if( G[v1][v2] != 0 and G[v1][v2] < min)
            min = G[v1][v2];
            T1 = v1; t2 = v2;
        Endif
    End for
End for
Temp1 = t1 ; temp2 = t2;
G[t1][t2] = G[t2][t1] = 0;
while(t1 >= 0)
    root_temp1 = t1; t1 = father[t1]
while(t2 >= 0)
    root_temp2 = t2; t2 = father[t2]
```

```
if(root_temp1!=root_temp2)
    Result[temp1][temp2] = Result[temp2][temp1] = min;
    Wt = wt+ Result[temp1][temp2]
    father[root_temp2] = root_temp1;
    Count++;
Endif
end while
End kruskal
```



# Dijkstra's Algorithm

- ▶ To find shortest path
- ▶ Let  $G=(V,E,W)$  be a directed weighted graph with  $N$  vertices  $v_1, v_2, \dots, v_n$ .
- ▶ Consider the starting vertex , dijkstra's Algorithm will find the minimum length of each vertex from the source vertex.



Algorithm dijkstra(g,s)

1.Create priority queue pq

2.Enqueue(pq,s)

3.For( $i=1; i \leq g-v; i++$ )

Distance[i]=-1

4.Distance[s]=0

5.while(!isemptyqueue(pq))

{5.1 v=deletemin(pq);

5.2 for all adjacent vertices w to v

{

Compute new distance  $d = \text{distance}[v] + \text{weight}[v][w]$ ;

If( $\text{Distance}[w] == -1$ )

{Distance[w]=new distance d;

Insert w in priorityqueue with priority d

Path[w]=v}

If( $\text{Distance}[w] > \text{newdistance } d$ )

{distance[w]=new distance d;

Update priority of vertex w to be d;

Path[w]=v;

}

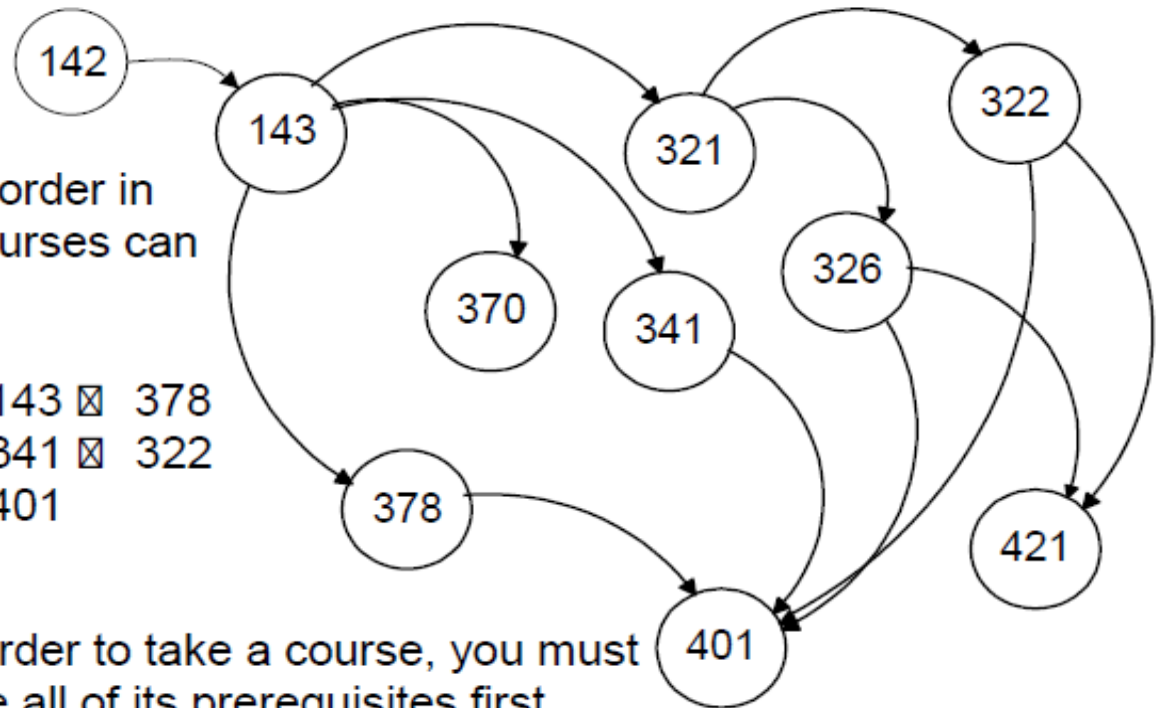
}}}

# Topological Sort

## Topological Sort

Problem: Find an order in which all these courses can be taken.

Example: 142 ☒ 143 ☒ 378  
☒ 370 ☒ 321 ☒ 341 ☒ 322  
☒ 326 ☒ 421 ☒ 401



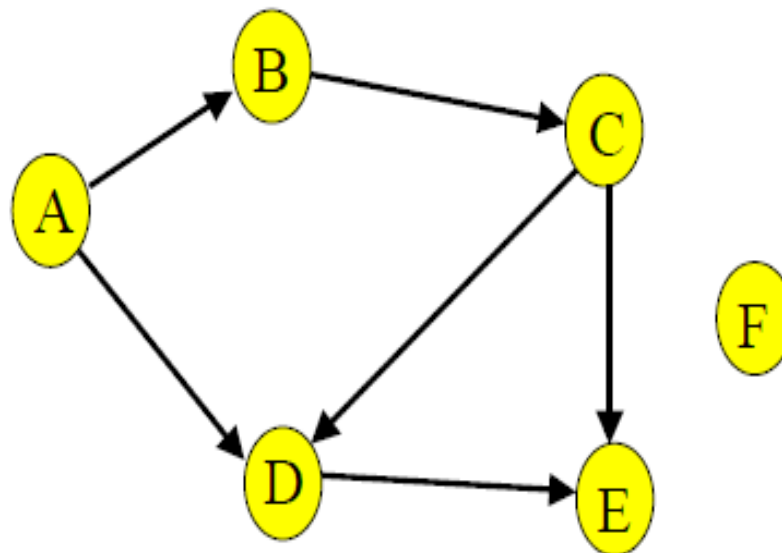
In order to take a course, you must take all of its prerequisites first

# Topological Sort

---

Given a digraph  $G = (V, E)$ , find a linear ordering of its vertices such that:

for any edge  $(v, w)$  in  $E$ ,  $v$  precedes  $w$  in the ordering

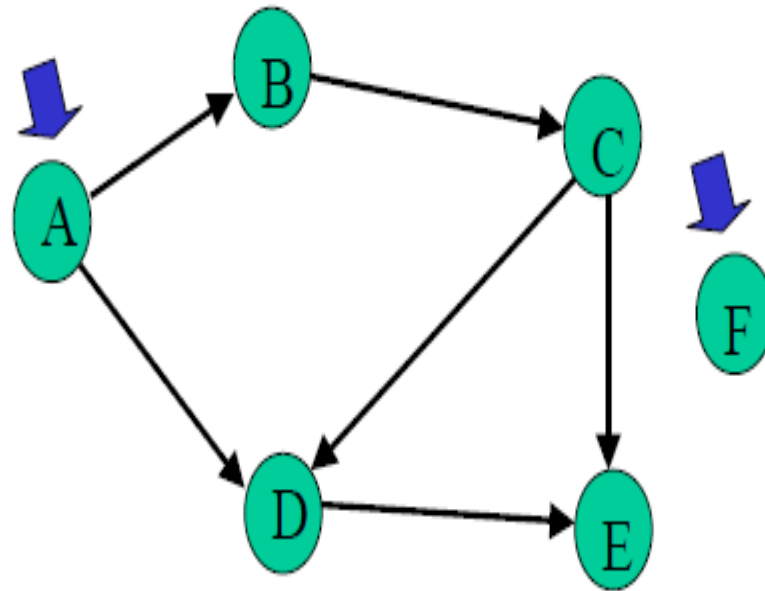




---

Step 1: Identify vertices that have no incoming edges

- The “in-degree” of these vertices is zero

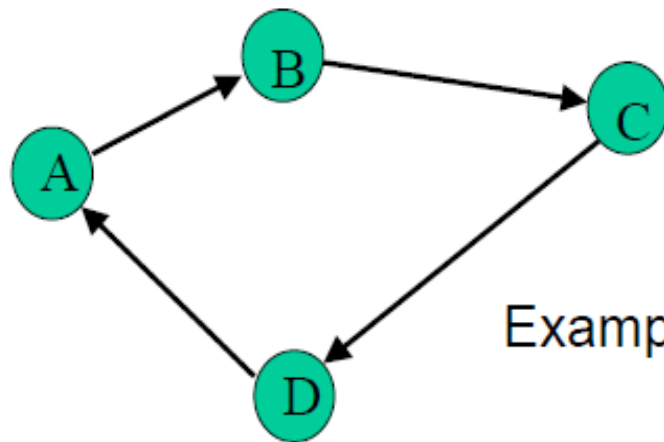


# Topo sort algorithm - 1a

---

Step 1: Identify vertices that have no incoming edges

- If *no such vertices*, graph has only cycle(s) (cyclic graph)
- Topological sort not possible – Halt.



Example of a cyclic graph

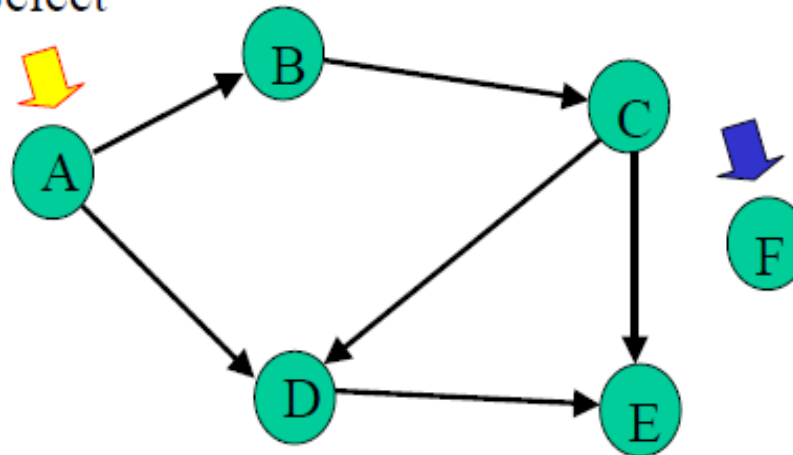
# Topo sort algorithm - 1b

---

Step 1: Identify vertices that have no incoming edges

- Select one such vertex

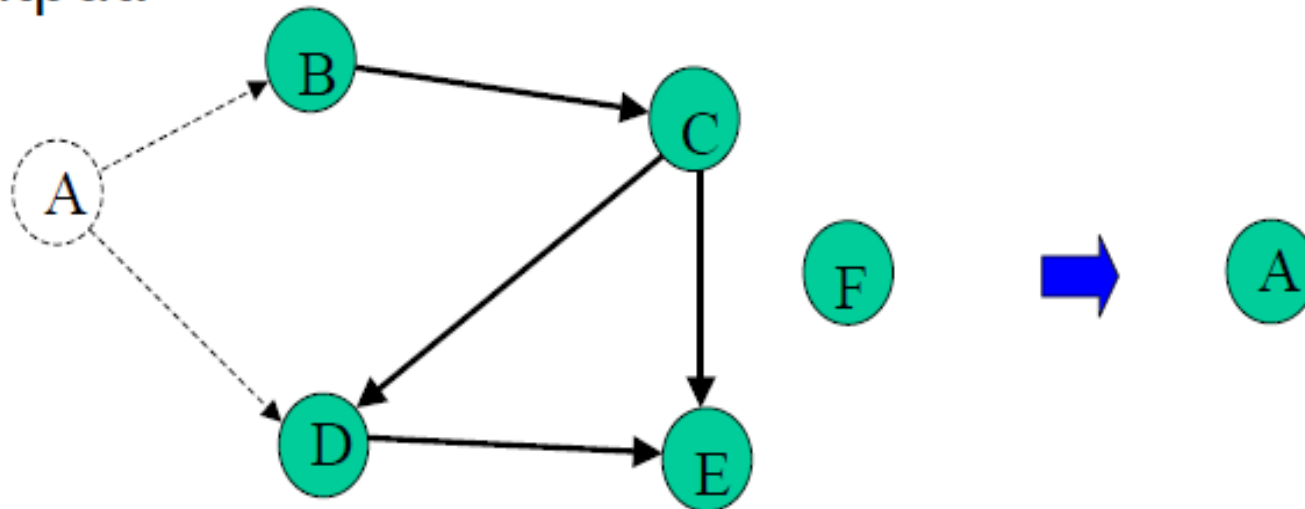
Select



# Topo sort algorithm - 2

---

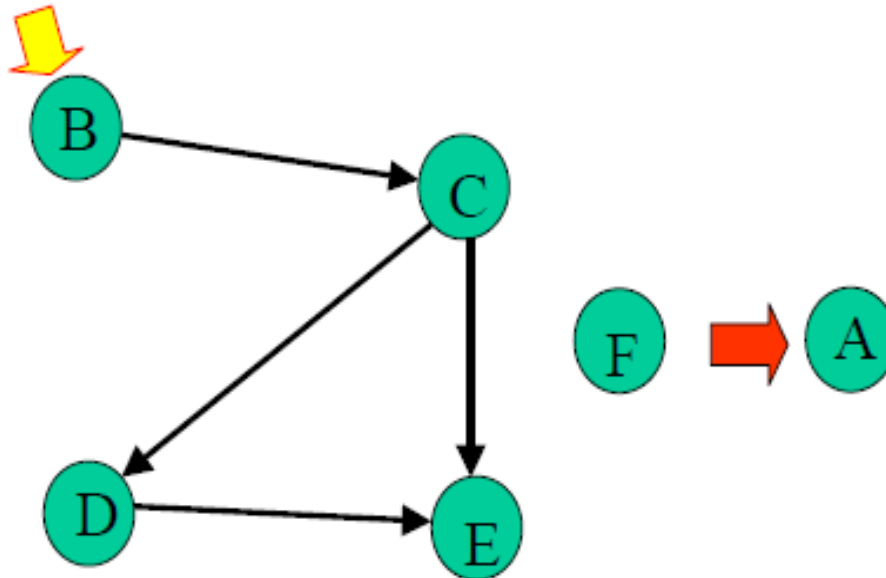
Step 2: Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.



# Continue until done

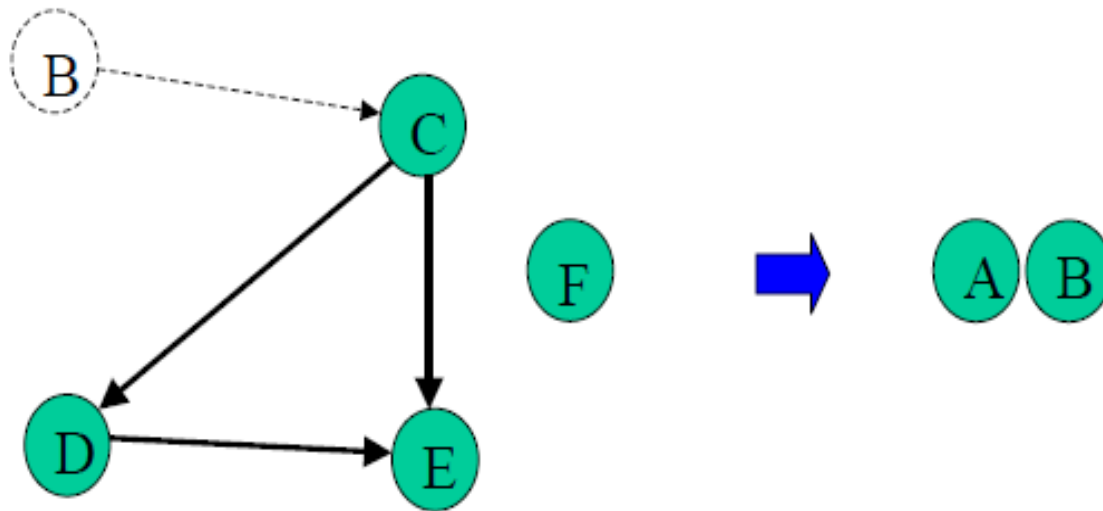
Repeat Step 1 and Step 2 until graph is empty

Select



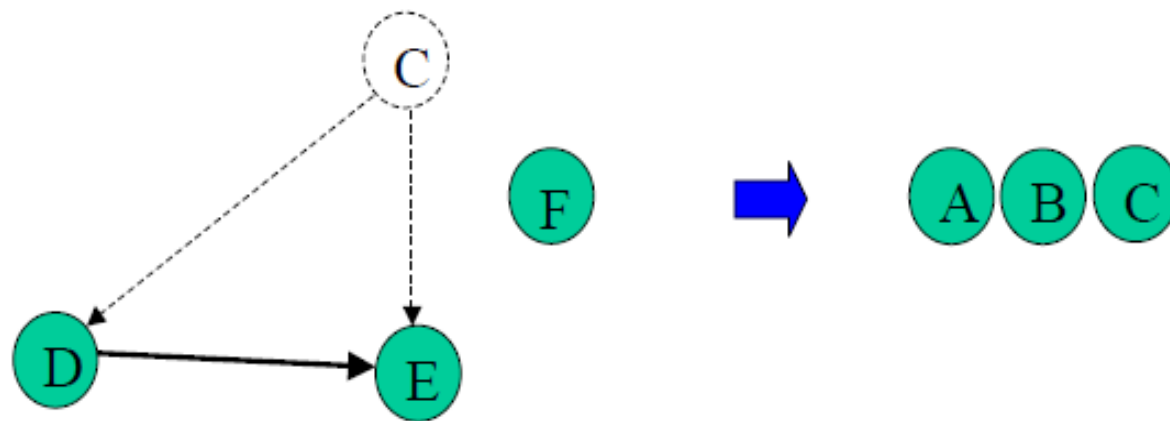
# B

Select B. Copy to sorted list. Delete B and its edges.



# C

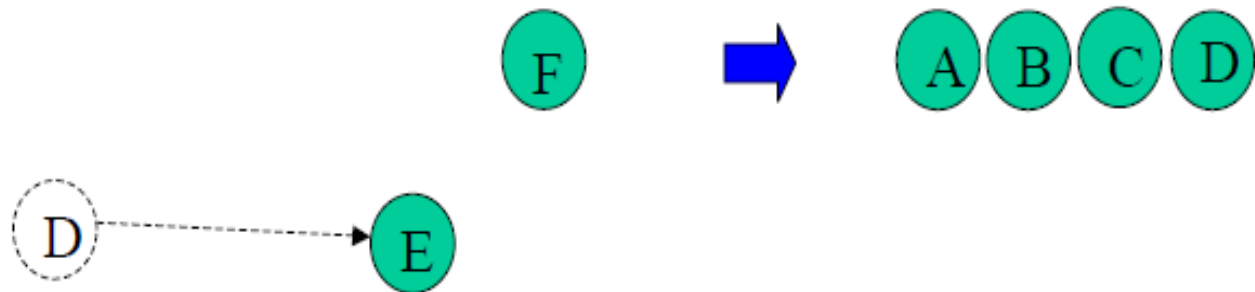
Select C. Copy to sorted list. Delete C and its edges.



# D

---

Select D. Copy to sorted list. Delete D and its edges.

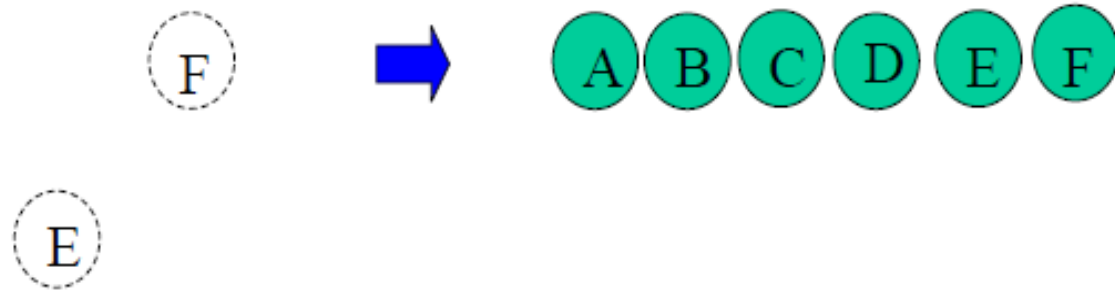




# E, F

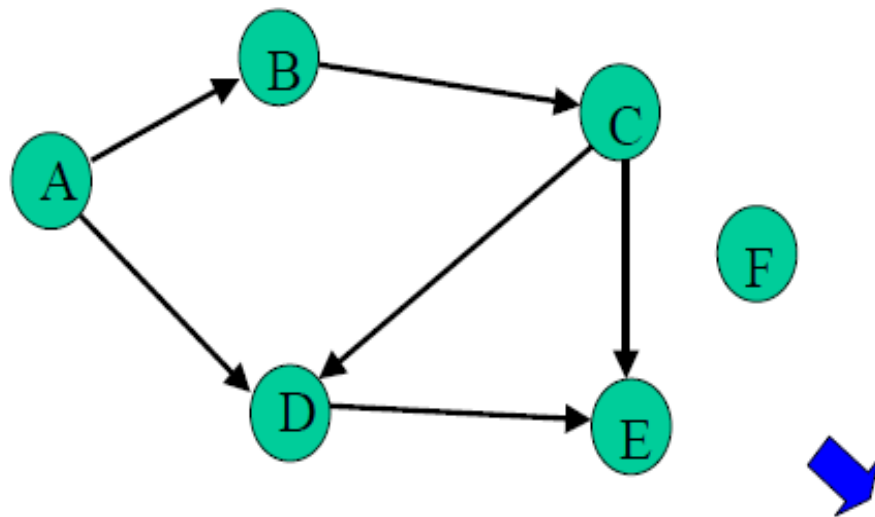
---

Select E. Copy to sorted list. Delete E and its edges.  
Select F. Copy to sorted list. Delete F and its edges.



# Done

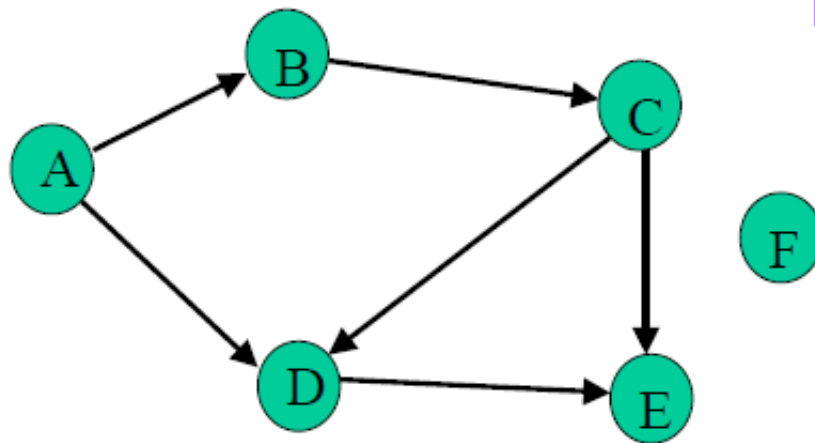
---



Remove from algorithm  
and serve.



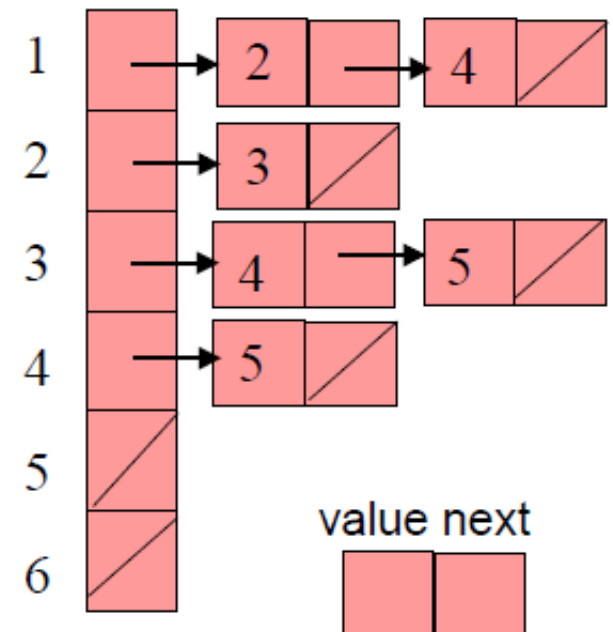
# Implementation



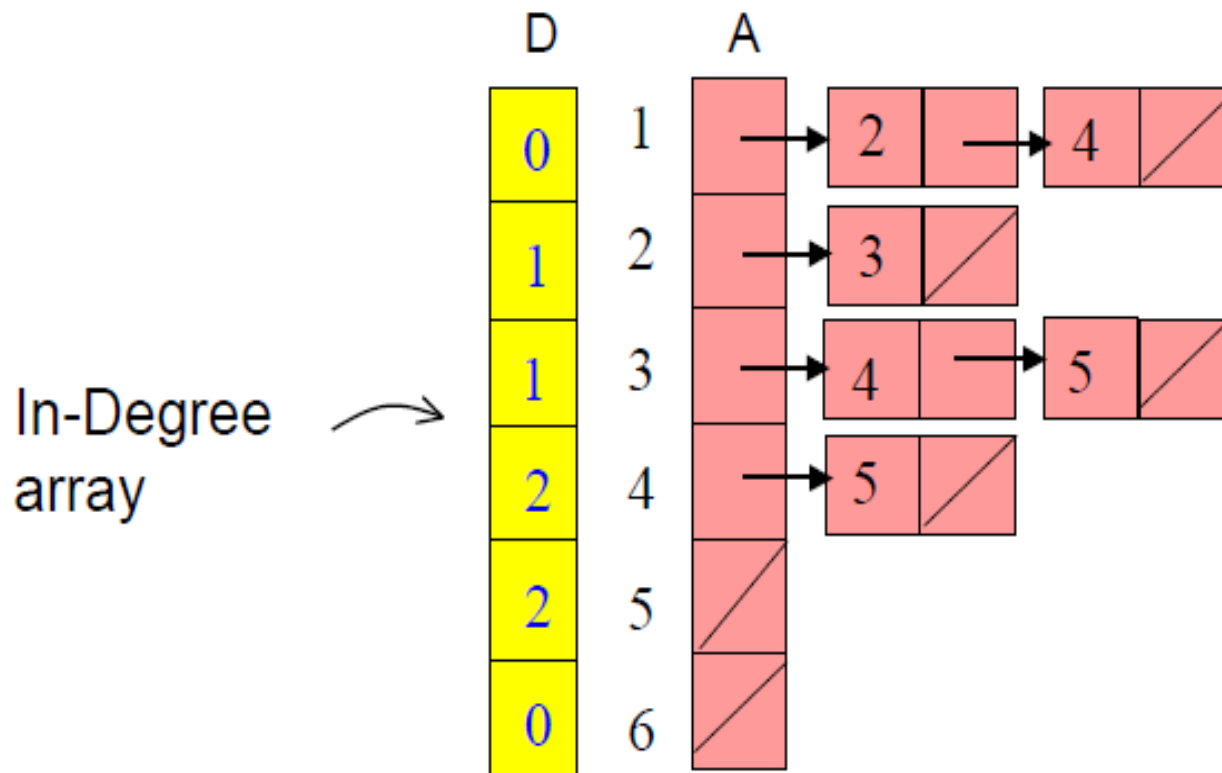
Translation array

1	2	3	4	5	6
A	B	C	D	E	F

Assume adjacency list representation



# Calculate In-degrees



Calculate In-degrees

for i = 1 to n do D[i] := 0; end for

for i = 1 to n do

  x := A[i];

  while x != null do

    D[x.value] := D[x.value] + 1;

    x := x.next;

  endwhile

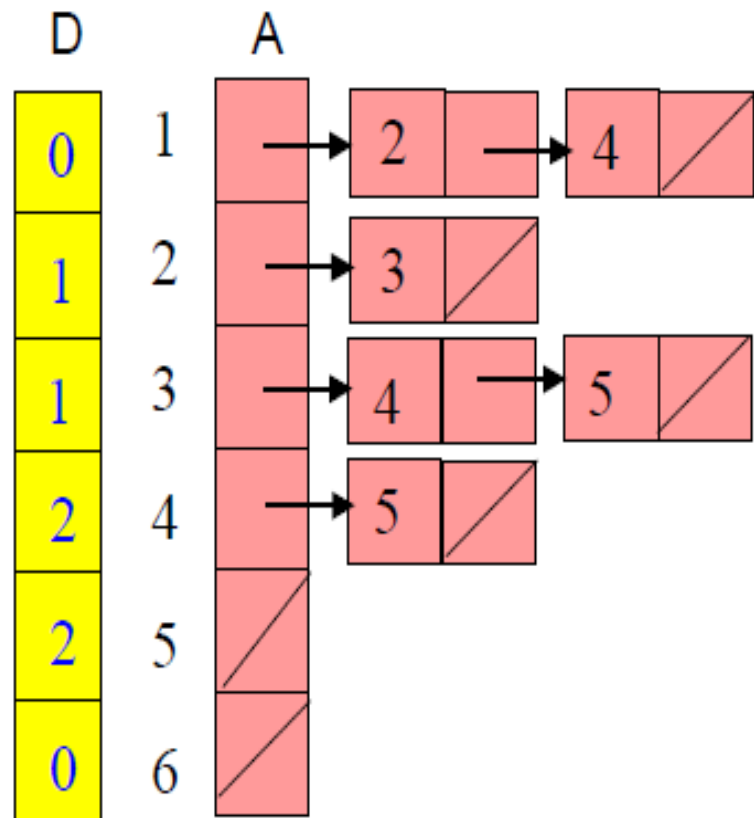
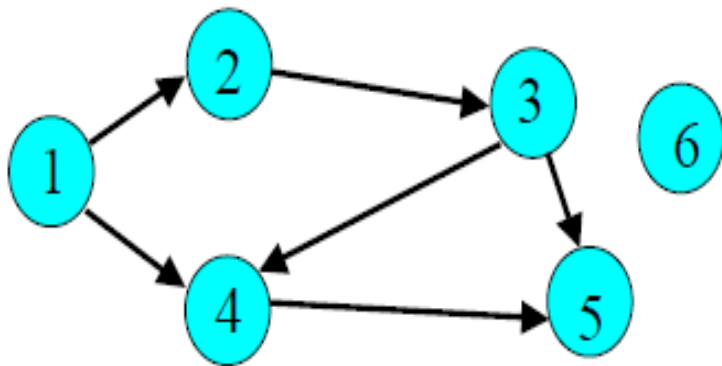
endfor



# Maintaining Degree 0 Vertices

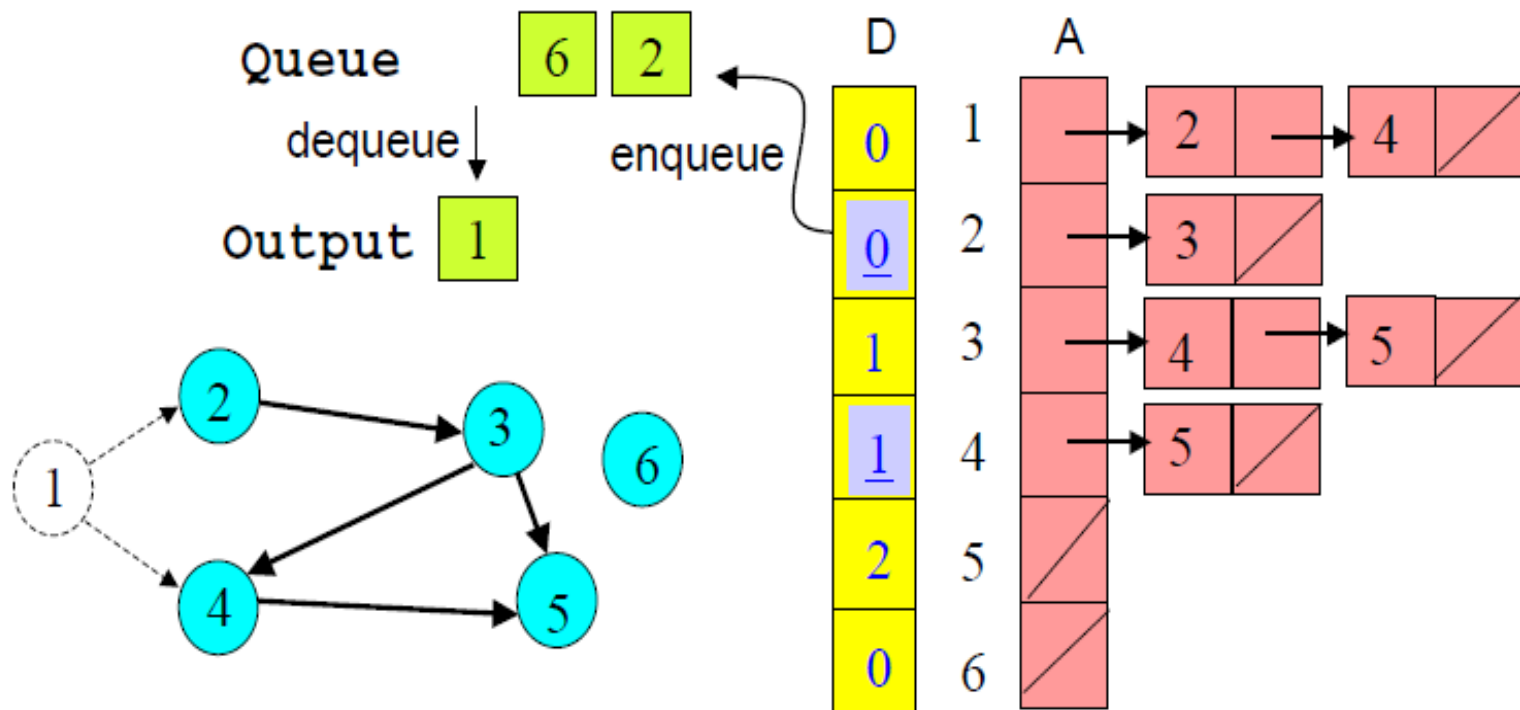
Key idea: Initialize and maintain a *queue* (or *stack*) of vertices with In-Degree 0

Queue 1 6



# Topo Sort using a Queue

After each vertex is output, when updating In-Degree array, *enqueue any vertex whose In-Degree becomes zero*



# Topological Sort Algorithm

---

1. Store each vertex's In-Degree in an array D
2. Initialize queue with all "in-degree=0" vertices
3. While there are vertices remaining in the queue:
  - (a) Dequeue and output a vertex
  - (b) Reduce In-Degree of all vertices adjacent to it by 1
  - (c) Enqueue any of these vertices whose In-Degree became zero
4. If all vertices are output then success, otherwise there is a cycle.



Thank you.....

Any Questions??