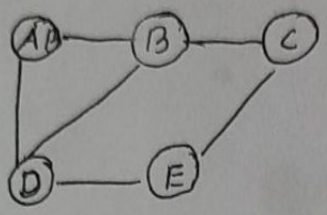
UNIT IV

(1) 57 copies

GRAPHS:

A graph 'G' is defined as an ordered set (V, E) where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.



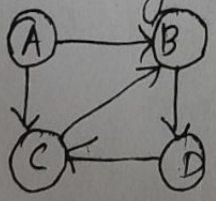
undirected graph.

$V(G) = \{A, B, C, D, E\}$

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

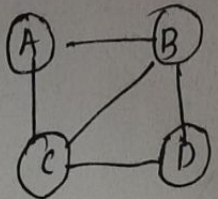
Terminologies:

Directed graph: In a directed graph, all the edges have direction associated with it. It is also called as digraph.



Directed graph.

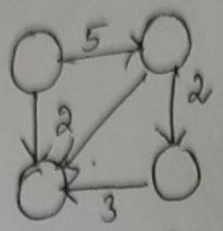
Undirected graph: In an undirected graph, edges do not have any direction associated with them.



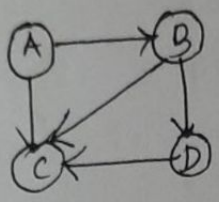
(29)

Weighted graph: All the edges in a graph have a weight or cost associated with it.

Eg:



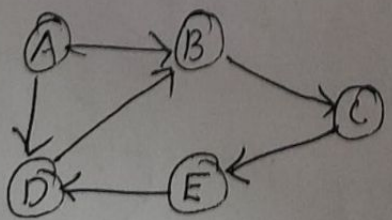
Unweighted graph: Edges of a graph will not have any cost or weight.



Path: A path in a graph is a sequence of vertices from source 'u' to destination 'v'. It is written as $P = \{v_0, v_1, v_2 \ldots v_n\}$.

Length of a path: Number of edges on a path.

Degree of a node: Degree of a node u, is the total number of incoming and outgoing edges.



Path (A, D) = A, B, C, E, D.

Length = 5

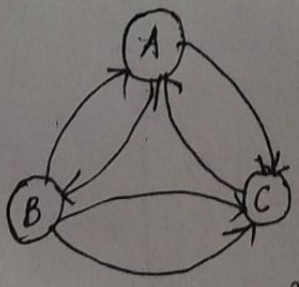
Degree (B) = 3.

outdegree: Number of outgoing edges of a node.
Indegree: Number of incoming edges of a node.

Connected graph: An undirected graph is connected if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected graph. If not it is weakly connected graph.

Complete graph: A complete graph is a graph in which there is an edge between every pair of vertices.

Strongly connected digraph: A digraph is said to be strongly connected if and only if there exist a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.



Cycle: A path, in which the first and the last vertices are the same.

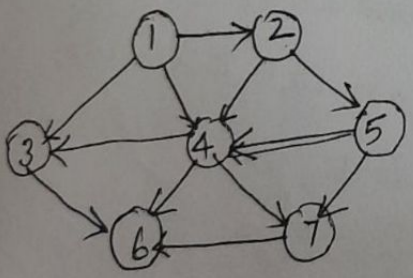
REPRESENTATION OF GRAPHS

2 ways.

* Adjacency matrix representation.
* Adjacency list representation.

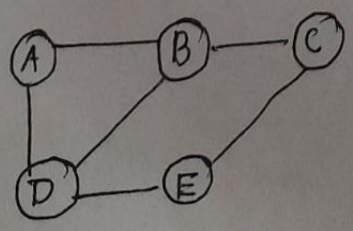
Adjacency matrix representation.

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry a_{ij} in the adjacency matrix will contain 1, if vertices V_i and V_j are adjacent to each other. If the nodes are not adjacent a_{ij} will be set to zero. Since an adjacency matrix contains only 0s and 1s it is called a bit matrix or a Boolean matrix



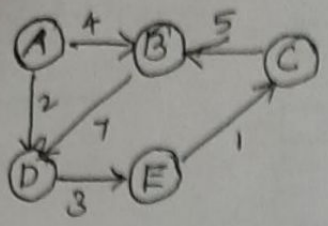
Directed Graph.

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



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



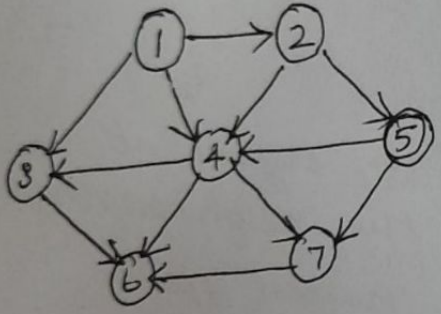
Weighted Graph

	A	B	C	D	E
A	0	4	0	2	0
B	0	0	0	7	0
C	0	5	0	0	0
D	0	0	0	0	7
E	0	0	1	0	0

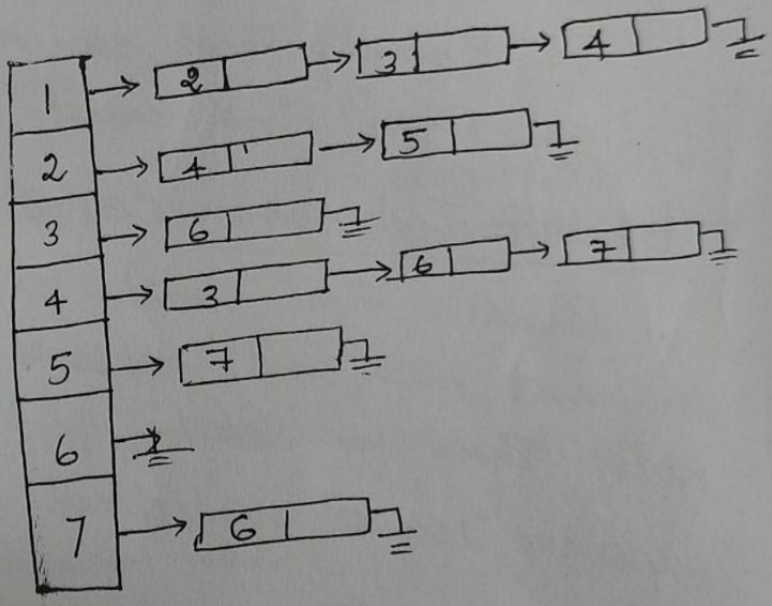
(5)

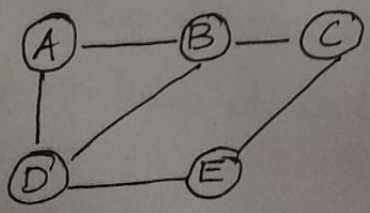
Adjacency list representation:

An adjacency list is another way in which graphs can be represented in the computer memory. This ~~case~~ structure consists of a list of all nodes in G. Thus every node is linked to its own list that contains the names of all other nodes that are adjacent to it.

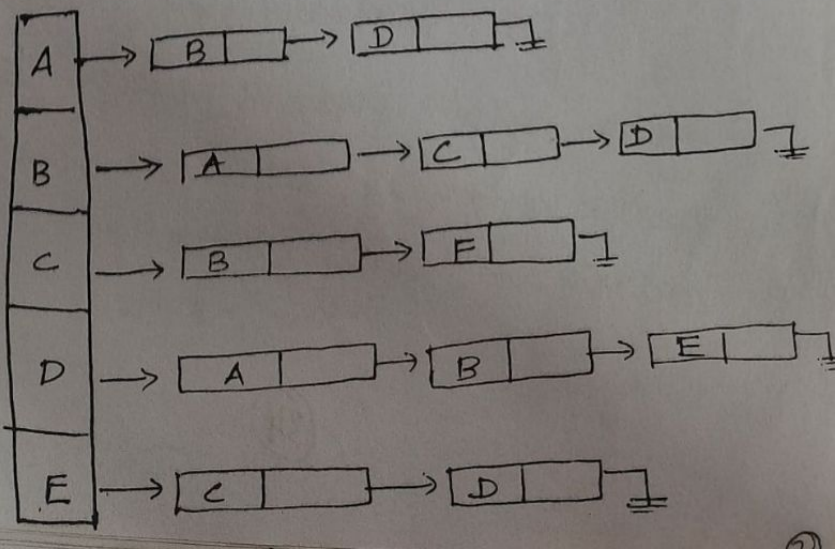


Directed Graph

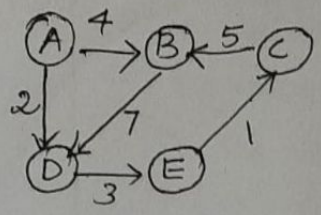




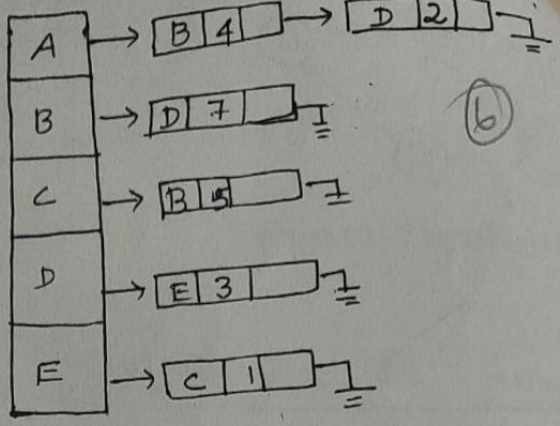
undirected graph



(33)



Weighted Graph.



(6)

GRAPH TRAVERSAL

Graph traversal is the process of examining the nodes and edges of the graph. There are Two standard methods of graph traversal.

1. Breadth First search (BFS)
2. Depth First search (DFS)

1. Breadth First Search (BFS).

Breadth First search algorithm begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, explores their unexplored neighbouring nodes. and so on.

BFS traverses a graph in a breadth-wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

(34)

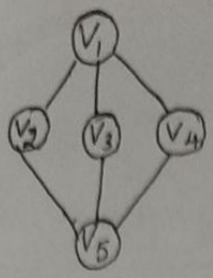
Algorithm:

1. Create an empty queue
2. Start with any node 'v' and enqueue it in the queue. and mark it as visited.
3. Dequeue a node from the queue and enqueue its adjacents to the queue and Mark them as visited. if not visited.
4. Repeat step 3 until the queue is empty.

Routine:

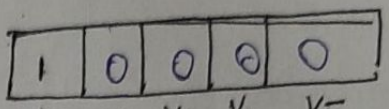
```
void BFS (Vertex v)
{
   Initialize (Q);                    v → source vertex
   visited [u] = 1;                   u → adjacent vertex
   Enqueue (u, Q);
   while ( ! IsEmpty (Q))
    {
      u = Dequeue (Q);
      print u;
      for all vertices v adjacent to u do
         if (visited [v] == 0) then
          {
             Enqueue (v, Q);
             visited [v] = 1;
          }
    }
}
```

Eg:

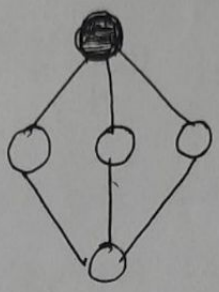


Iteration 1: Enqueue (V_1), mark it as visited.

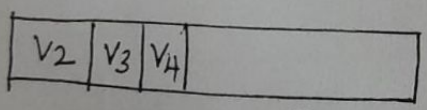
Queue Q



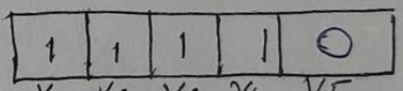
Visited.



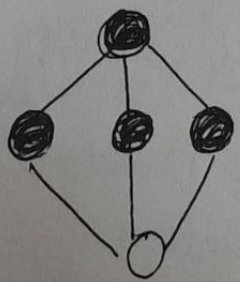
Iteration 2: Enqueue (V_2), Enqueue (V_3), Enqueue (V_4) and dequeue (V_1) and print it



Queue Q

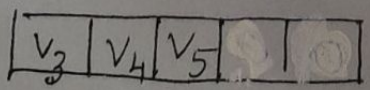


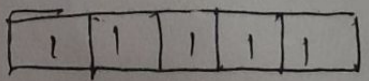
Visited.

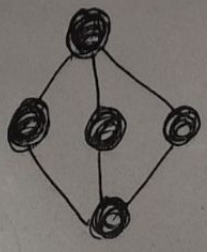


V_1

Iteration 3: Enqueue (V_5) and dequeue (V_2) and display.







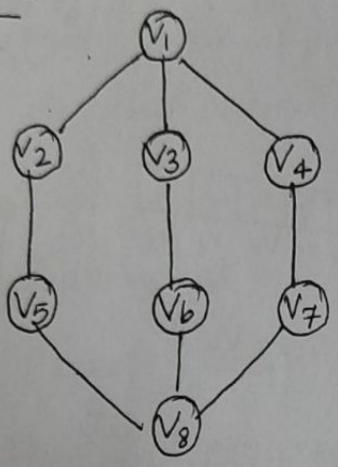
$V_1 \rightarrow V_2$

36

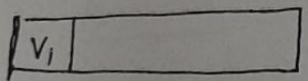
Iteration 4 : since all the nodes are visited dequeue the ~~elements~~ nodes and print all the nodes.

BFS : $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5$

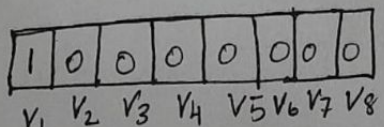
Eg 2 :



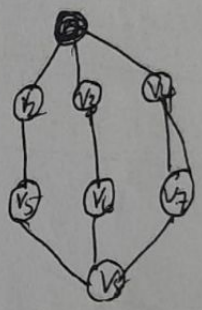
Iteration 1 :



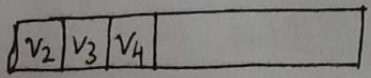
Queue



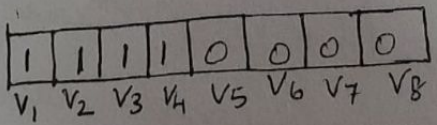
visited.

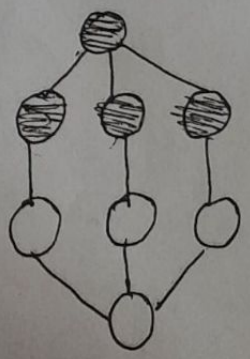


Iteration 2 : (V_1 dequeued, V_2, V_3, V_4 enqueued).



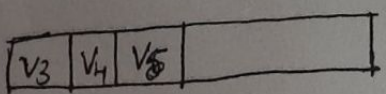
Queue

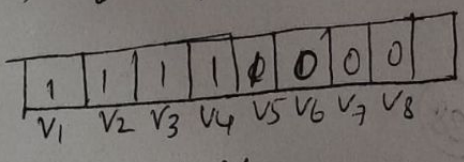


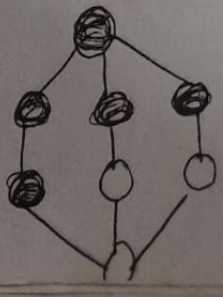


print : V_1

Iteration 3 : V_2 dequeued, ~~V_5~~ enqueued.



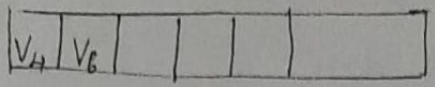




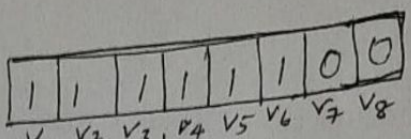
print : $V_1 \rightarrow V_2$

(37)

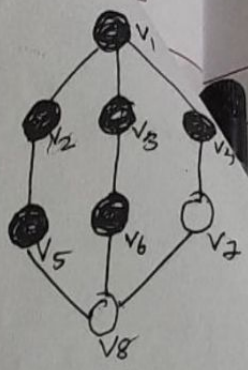
Iteration 4: V_3 dequeued, V_6 enqueued.



Queue

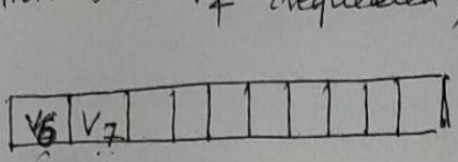


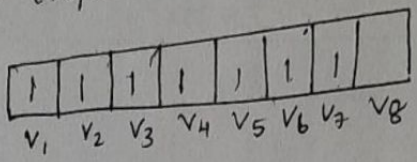
visited

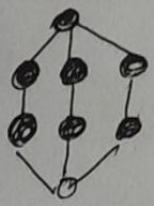


print : $V_1 \rightarrow V_2 \rightarrow V_3$

Iteration 5: V_4 dequeued, V_7 enqueued.

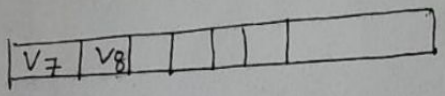


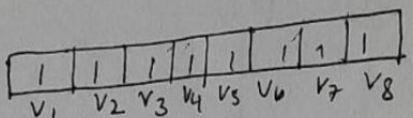


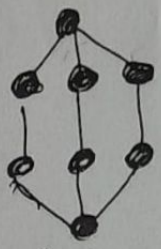


Print : $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4$

Iteration 6: V_6 dequeued, V_8 enqueued.







print : $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6$

Iteration 7:

Dequeue all the nodes and print it

BFS : $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8$

2. Depth First search: (DFS).

DFS starts with a node and visits one of its adjacent and further moves one by one until it fonds a dead end. then it come back to previous ~~node~~ node and repeats the same until visiting all the nodes.

DFS traverse a graph in a depth wand motion and uses a stack to remember to get the next vertex to re-start the search when a dead end occurs.

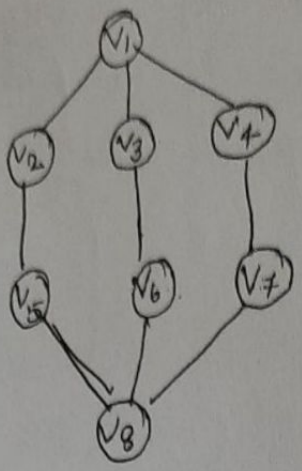
Algorithm:

1. Choose any one of the node as starting node and push it in the stack, ~~and~~ Display the same in the output and mark it visited.
2. Push the adjacent nodes in the stack and display it.
3. Repeat step 2 until finds a dead end.
4. If no adjacent node, perform pop operation. Again repeat step 2.

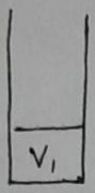
Routine:

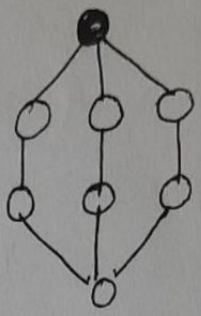
```
void DFS (Vertex V)
{
    visited[v] = True;
    for each w adjacent to v
        if (!visited[w])
            DFS(w);
}
```

Eg:



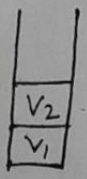
Iteration 1:

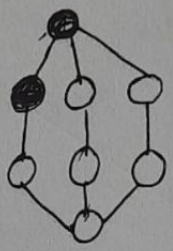




V_1

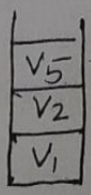
Iteration 2:

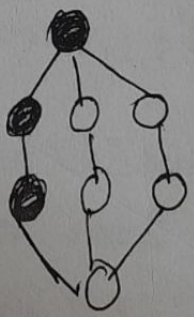




$V_1 \rightarrow V_2$

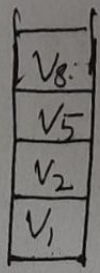
Iteration 3:

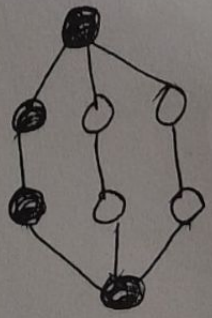




$V_1 \rightarrow V_2 \rightarrow V_5$

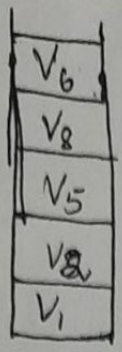
Iteration 4:

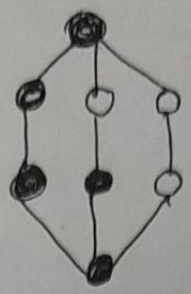




$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8$

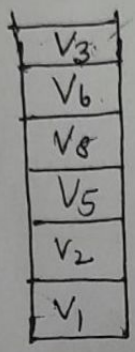
Iteration 5

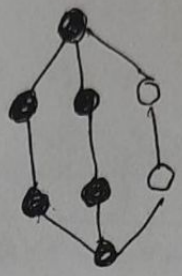




$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8 \rightarrow V_6$

Iteration 6

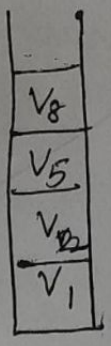




$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8 \rightarrow V_6 \rightarrow V_3$

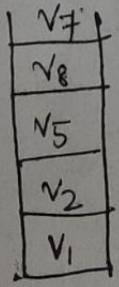
Iteration 7:

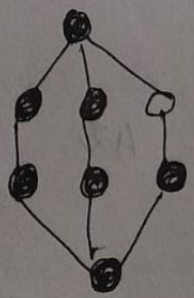
Reached dead end hence pop out V_3, and check whether V_6 has any unvisited adjacent. Since it does not has any unvisited adjacent, it is also poped out.



Iteration 8:

Since V_8 has an unvisited adjacent ie V_7 it is pushed.

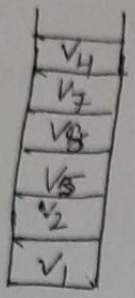


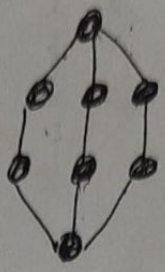


$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8 \rightarrow V_6 \rightarrow V_3 \rightarrow V_7$

Iteration 9:

since v_4 has adjacent, it is pushed





$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8 \rightarrow V_6 \rightarrow V_3 \rightarrow V_7 \rightarrow V_4$.

since all the nodes are visited stop the process and pop out all the nodes

DFS : $V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8 \rightarrow V_6 \rightarrow V_3 \rightarrow V_7 \rightarrow V_4$.

TOPOLOGICAL SORTING:

Topological sorting is the process of ordering the nodes of a directed acyclic graph in a linear manner. That is, if there is a path from v_i to v_j means v_j appears after v_i

Ordering is only possible for directed acyclic graph and not possible for the graph with cycle.

42 Kulkarni Sree Krishna yesbee

TOPOLOGICAL SORTING

- Topological sorting is the process of ordering the vertices in directed acyclic graph such that if there is a path from v_i to v_j, then v_j appears after v_i.
- Topological ordering is not possible if the graph has a cycle.
- Ordering is not necessarily unique; any legal ordering can be done.
- Directed Acyclic Graph (DAG) is a directed graph without cycles.

Algorithm:

step 1: Calculate the indegree of all the vertices

step 2: Identify the vertices whose indegree is zero and insert into a queue.

step 3: Dequeue an element from the queue (i.e a vertex from the queue) and remove all the edges connected to it.

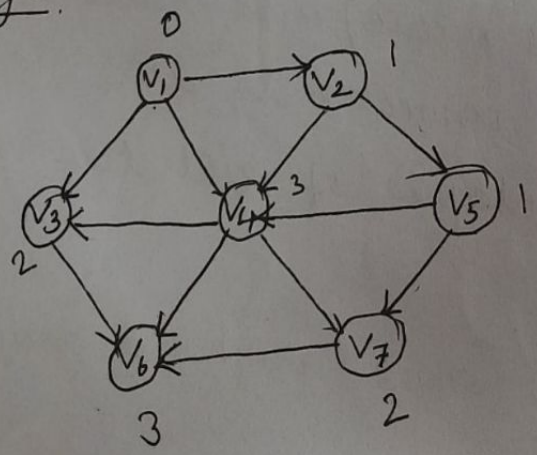
step 4: Update the indegree of all the vertices

step 5: Repeat step 2 to step 3 until the graph is empty.

Routine:

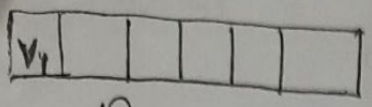
```
void Topsort ( Graph G)
{
  int counter;
  Vertex V, W;
  for(counter=0; counter < NumVertex ; counter++)
  {
    V=FindNewVertexofDegreeZero();
    if (V== NotAVertex)
    {
      Error ("Graph has a cycle");
      break;
    }
    TopNum[V]= counter ;
    for each w adjacent to V
      Indegree[W]--;
  }
}
```

Eg:



V_1	0
V_2	1
V_3	2
V_4	3
V_5	1
V_6	3
V_7	2

Step 1: Enqueue V_1 (since its indegree is 0)



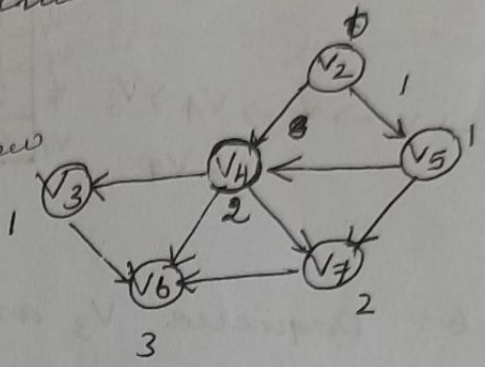
Queue.

V_1

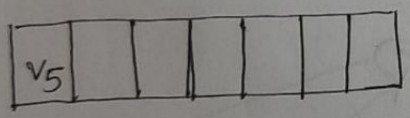
Step 2: Dequeue V_1 from queue and enqueue V_2

V_1

V_1	0
V_2	0 ← New
V_3	1
V_4	2
V_5	1
V_6	2
V_7	2

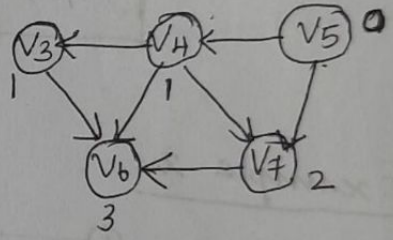


Step 3: Dequeue V_2 from queue and enqueue V_5

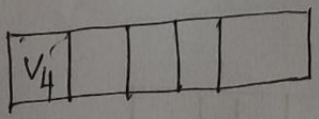


$V_1 \rightarrow V_2$

V_1	0
V_2	0
V_3	1
V_4	1
V_5	0 ← New.
V_6	3
V_7	2

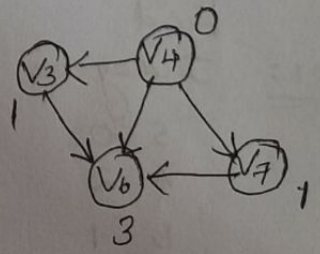


Step 4: Dequeue V_5 from queue and ~~deq~~ enqueue V_4

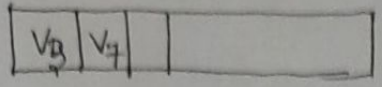


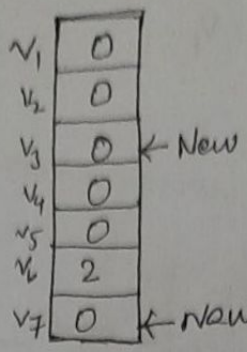
$V_1 \rightarrow V_2 \rightarrow V_5$

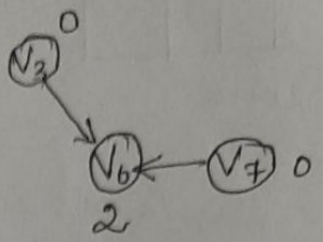
V_1	0
V_2	0
V_3	1
V_4	0 ← New
V_5	0
V_6	3
V_7	1



Step 5: Dequeue V_4, Enqueue V_3 & V_7







$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_4$

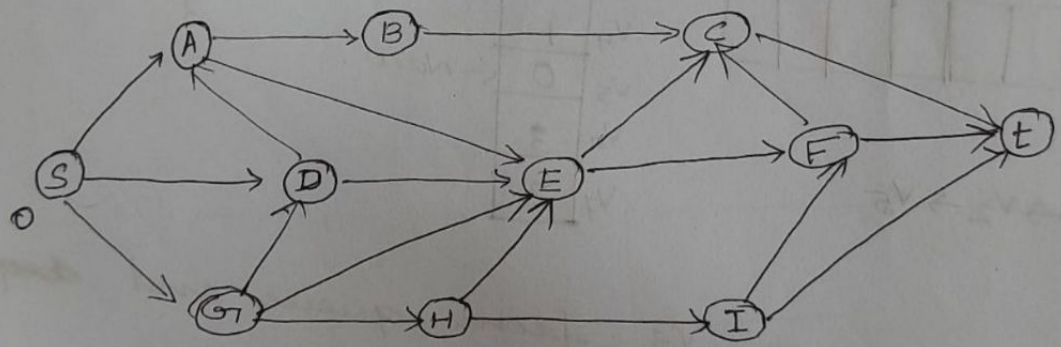
Step 6: Dequeue V_3 and V_7.

$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_4 \rightarrow V_3 \rightarrow V_7$

Topological sorting is

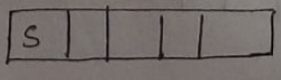
$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_4 \rightarrow V_3 \rightarrow V_7$

Example 2:

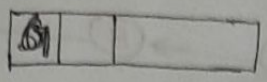


Step 1: Find Indegree of all vertices. and enqueue S.

S	0
A	2
B	1
C	3
D	2
E	4
F	2
G1	1
H	1
I	1
t	3

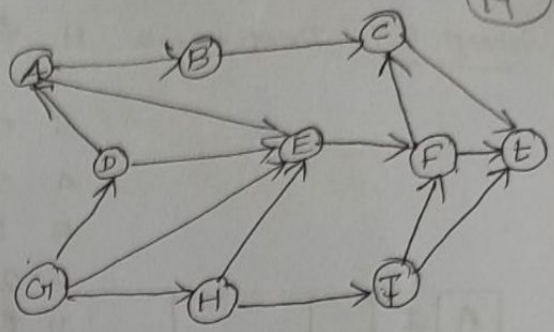


Step 2: Dequeue S and enqueue G1.

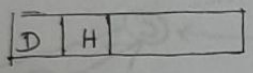


S

S	0
A	1
B	1
C	2
D	1
E	4
F	2
G	0 ← New
H	1
I	1
t	3

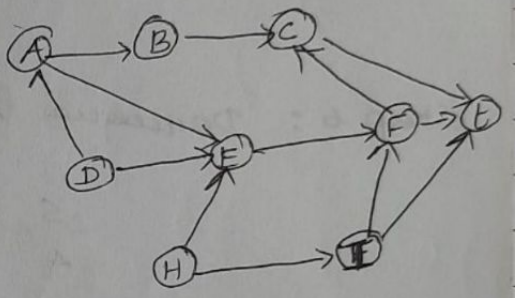


Step 3: Dequeue G1, Enqueue D & H

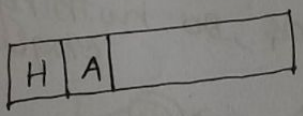


S → G1

S	0
A	1
B	1
C	2
D	0 ← New
E	3
F	2
G	0
H	0 ← New
I	1
t	3

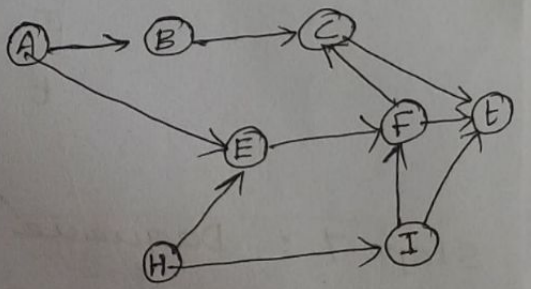


Step 4: Dequeue D, Enqueue A

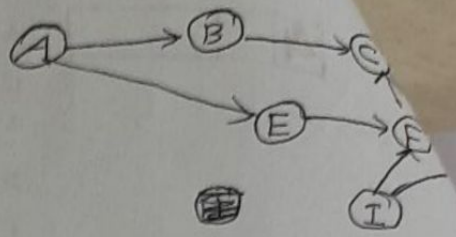


S → G1 → D

S	0
A	0 ← New
B	1
C	2
D	0
E	2
F	2
G	0
H	0
I	1
t	3



Step 5: Dequeue H, Enqueue I

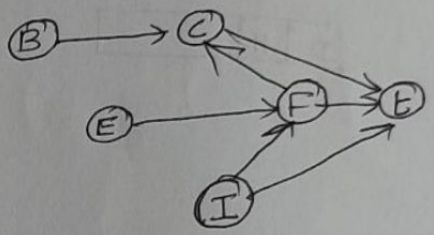


S	0
A	0
B	1
C	2
D	0
E	1
F	2
G	0
H	0 ~~Head~~
I	0 ← New
t	3

A	I		

S → G → D → H

Step 6: Dequeue A, Enqueue B & E

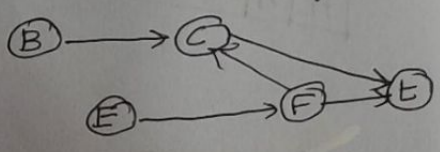


S	0
A	0
B	0 ← New
C	2
D	0
E	0 ← New
F	2
G	0
H	0
I	0
t	3

I	B	E	

S → G → D → H → A

Step 7: Dequeue I, no new zeros, so nothing enqueued

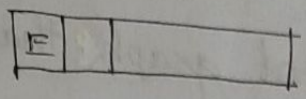


B	0
C	2
E	0
F	1
~~G~~	
t	2

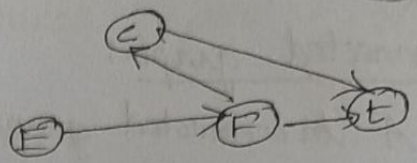
B	E		

S → G → D → H → A → I

step 8: Dequeue B

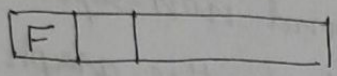


C 1
E 0
F 1
t 2

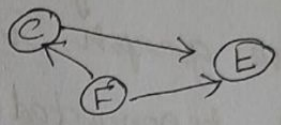


S → G → D → H → A → I → B

step 9: Dequeue E, Enqueue F

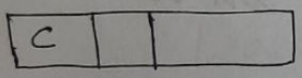


C 1
F 0 ← New
t 2



S → G → D → H → A → I → B → F

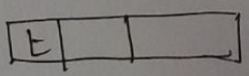
Step 10: Dequeue F, enqueue C



C 0
t 1

S → G → D → H → A → I → B → E → F

Step 11: Dequeue C, and enque t



t 0

S → G → D → H → A → I → B → E → F → C → t