

11/14/2023

Graphs

1

Prof. Shweta Dhawan Chachra

Graph

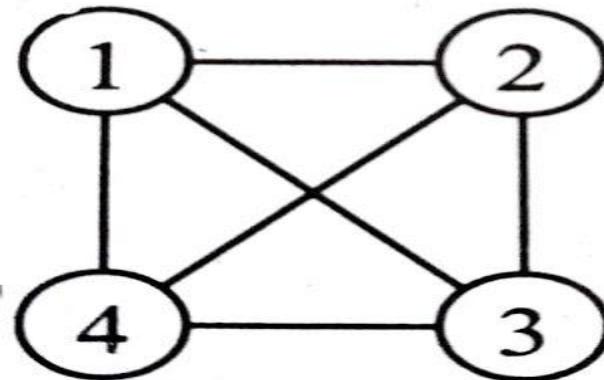
- Collection of 2 sets V & E
- V=Set of Nodes=(v₁,v₁,v₂.....v_n)
- E=Set of Edges=(e₁,e₂,e₃.....e_n)
- Edge=an arc that connects 2 nodes

Graph

- 2 Types
 - Undirected
 - Directed

Undirected Graph

- A graph, which has **unordered pair of vertices**, is called undirected graph.
- Suppose there is an edge between v_0 & v_1 then it can be represented as (v_0, v_1) or (v_1, v_0) also
- $V(G) = \{1, 2, 3, 4\}$
- $E(G) = \{ (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4) \}$



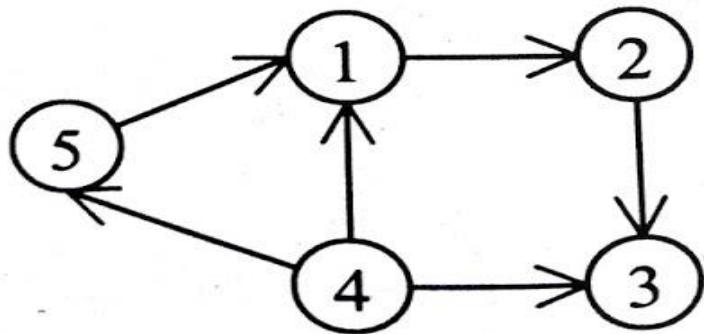
Directed Graph

- A graph which has **ordered pair of vertices $\langle v1, v2 \rangle$ where edge starts from $v1$ and ends at $v2$**
- Each edge has direction, means $\langle v1, v2 \rangle$ and $\langle v2, v1 \rangle$ **will represent different edges.**
- Directed means that a direction will be associated with that edge.
- Also known as **digraph**

Directed Graph

$$V(G) = \{1, 2, 3, 4, 5\}$$

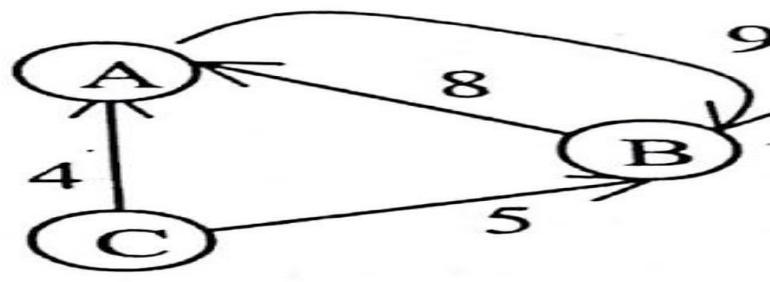
$$E(G) = \{ <1, 2>, <2, 3>, <4, 3>, <4, 1>, <4, 5>, <5, 1> \}$$



Graph Terminology

Weighted Graph

- A graph is said to be weighted if it's edges have been assigned some **non negative value as weight**.
- A weighted graph is also known as network.
- Graph G9 is a weighted graph.



G9

Graph Terminology

Adjacent nodes

- A node u is adjacent to another node or is a neighbor of another node v **if there is an edge from node u to node v .**
- In undirected graph if (v_0, v_1) is an edge then v_0 is adjacent to v_1 and v_1 is adjacent to v_0 .
- In a digraph if $\langle v_0, v_1 \rangle$ is an edge then v_0 is adjacent to v_1 and v_1 is adjacent from v_0

Graph Terminology

Incidence :-

- In an undirected graph the **edge (v_0, v_1) is incident on nodes v_0 and v_1 .**
- In a digraph the **edge $\langle v_0, v_1 \rangle$ is incident from node v_0 and is incident to node v_1 .**

Path :-

- A path from node u_0 to node u_n is a sequence of nodes $u_0, u_1, u_2, u_3, \dots, u_{n-1}, u_n$ such that **u_0 is adjacent to u_1 , u_1 is adjacent to u_2, \dots, u_{n-1} is adjacent to u_n .**

Length of path :-

- Length of a path is the **total number of edges included in the path.**

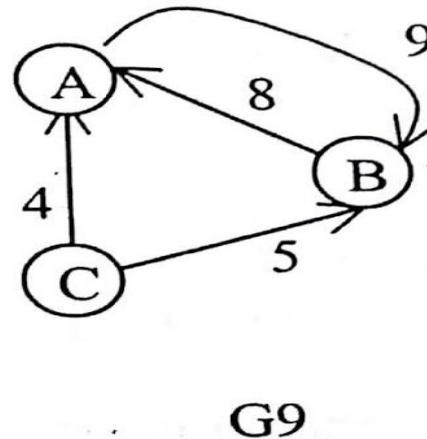
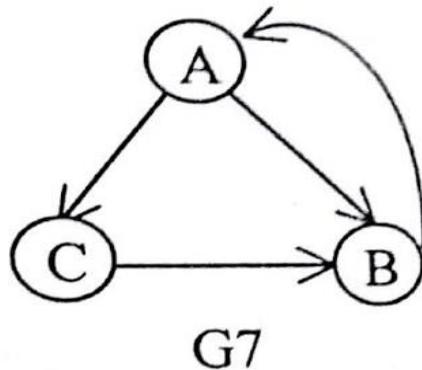
Graph Terminology

- **Closed path :-**
- A path is said to be closed **if first and last nodes of the path are same.**

- **Simple path :-**
- Simple path is a path in which **all the nodes are distinct with an exception that the first and last nodes of the path can be same.**

Graph Terminology

- **Cycle :-**
- Cycle is a simple path in which **first and last nodes are the same** or we can say that **a closed simple path** is a cycle.
 - In graph G7, path ACBA is a cycle
 - In graph G9 path ABA is a cycle.



Prof. Shweta Dhawan Chachra

Graph Terminology

Cyclic graph :-

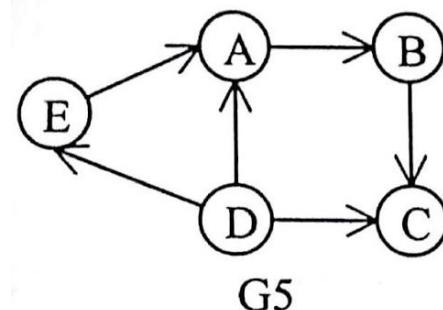
- A graph that has cycles is called a cyclic graph.

Acyclic graph :-

- A graph that has no cycles is called an acyclic graph.

Dag :-

- A directed acyclic graph is named as dag after its acronym.
- Graph G5 is an example of a dag.

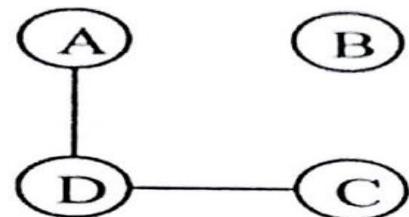


Prof. Shweta Dhawan Chachra

Graph Terminology

Degree :-

- In an undirected graph,
- **the number of edges connected to a node** is called the degree of that node,
or
- we can say that degree of a node is the number of edges incident on it.
- In graph G2 degree of node A is 1, degree of node B is zero.



G2

Graph Terminology

- In a digraph, there are two degrees for every node known as indegree and outdegree.

Indegree

Outdegree

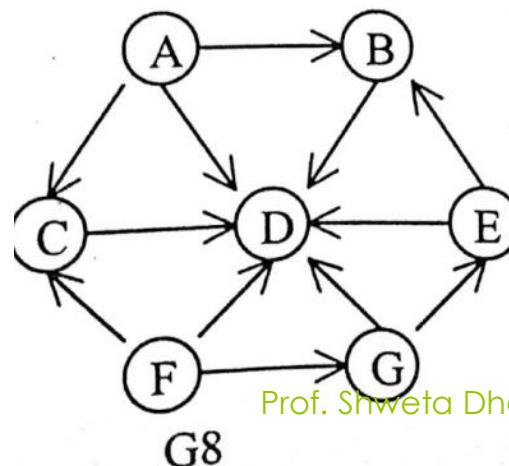
Graph Terminology

Indegree :-

- The indegree of a node is the number of edges coming to that node or in other words edges incident to it.
- In graph G8, the indegree of nodes A, B, D and G are 0, 2, 6 and 1 respectively.

Outdegree :-

- The outdegree of node is the number of edges going outside from that node, or in other words the edges incident from it.
- In graph G8, outdegrees of nodes A, B, D, F, and G are 3, 1, 0, 3, and 2 respectively.



Prof. Shweta Dhawan Chachra

G8

Representation of Graph

- ***There are two ways for representing the graph in computer memory.***
 - ***sequential representation and***
 - ***linked list representation.***

Representation of Graph

Adjacency Matrix :-

- Keeps the information of adjacent nodes.
 - that whether this node is adjacent to any other node or not.

Representation of Graph

Adjacency Matrix :-

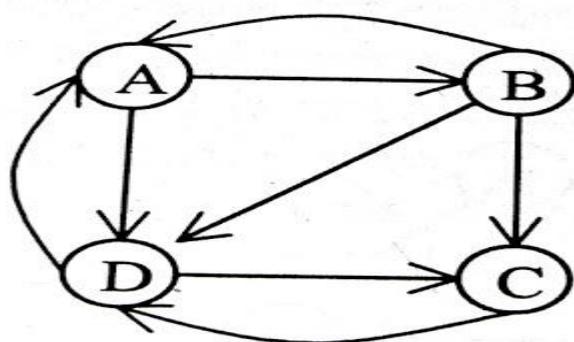
- Represent a matrix in two dimensional array
 - array[n][n]
 - Suppose there are 4 nodes in graph then row1 represents the node1, row2 represents the node2 and so on.
 - Similarly column1 represents node1, column2 represents node2 and so on.
- The entry of this matrix will be as-

**Arr[i][j] = 1 If there is an edge from node i to node j
= 0 If there is no edge from node i to node j**

- Hence, all the entries of this matrix will be either 1 or 0.

Representation of Graph

Adjacency Matrix :-



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

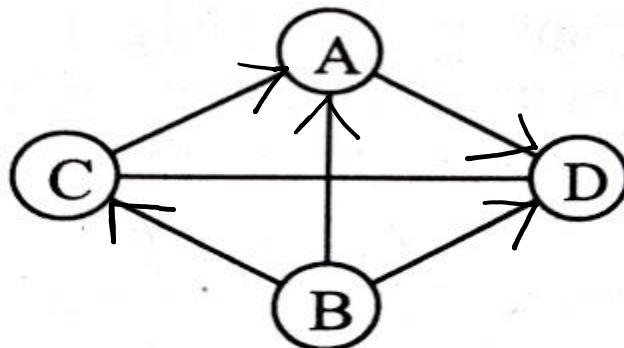
Adjacency Matrix A =

arr[0][1] = 1, which represents there is an edge in the graph from node A to node B.

Prof. Shweta Dhawan Chachra

Representation of Graph

Adjacency Matrix :-



Adjacency Matrix A =

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

arr[0][1] = 1, which represents there is an edge in the graph from node A to node B.

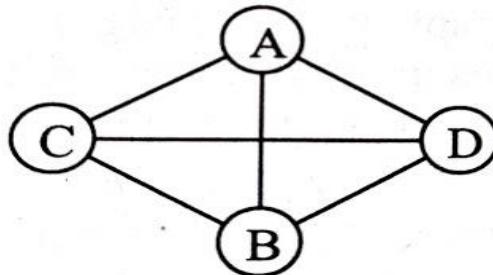
Prof. Shweta Dhawan Chachra

Representation of Graph

Adjacency Matrix :-

In an undirected graph

- rowsum and columnsum for a node is same and represents the degree of that node and



Adjacency Matrix A =

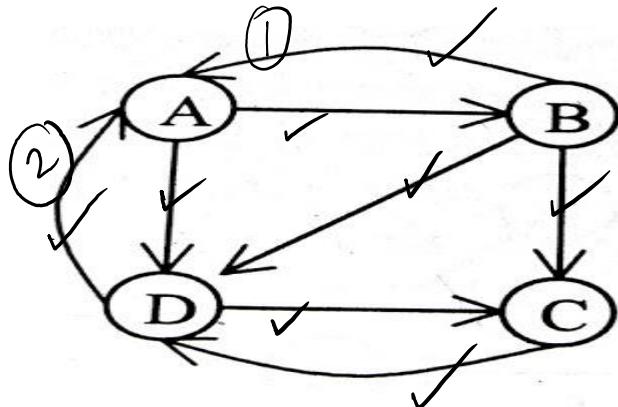
	A	B	C	D	
A	0	1	1	1	= 3
B	1	0	1	1	= 3
C	1	1	0	1	= 3
D	1	1	1	0	= 3
	3	3	3	3	

Representation of Graph

Adjacency Matrix :-

In directed graph,

- rowsum represents the outdegree
- columnsum represents the indegree of that node.



Adjacency Matrix A =

	A	B	C	D	
A	0	1	0	1	2 ✓
B	1	0	1	1	3 ✓
C	0	0	0	1	1 ✓
D	1	0	1	0	2 ✓

Rowsum \Rightarrow Out-degree

Columnsum
Indegree

Prof. Snweta Dhawan Chachra

Representation of Graph

Adjacency Matrix :-

Weighted Graph

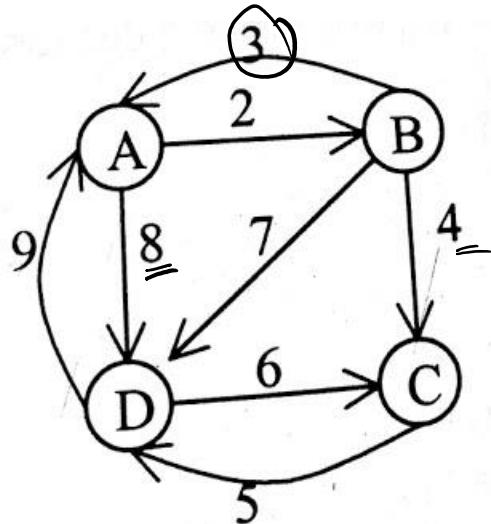
= Weight on edge If there is an edge from node i to node j.

$Arr[i][j]$

= 0 Otherwise

Representation of Graph

Adjacency Matrix :-



$$\text{Weighted Adjacency Matrix } W = \begin{bmatrix} A & B & C & D \\ \hline A & 0 & 2 & 0 & 8 \\ B & 3 & 0 & 4 & 7 \\ C & 0 & 0 & 0 & 5 \\ D & 9 & 0 & 6 & 0 \end{bmatrix}$$

arr[0][1] = weight, which represents there is an weighted edge in the graph from node A to node B.

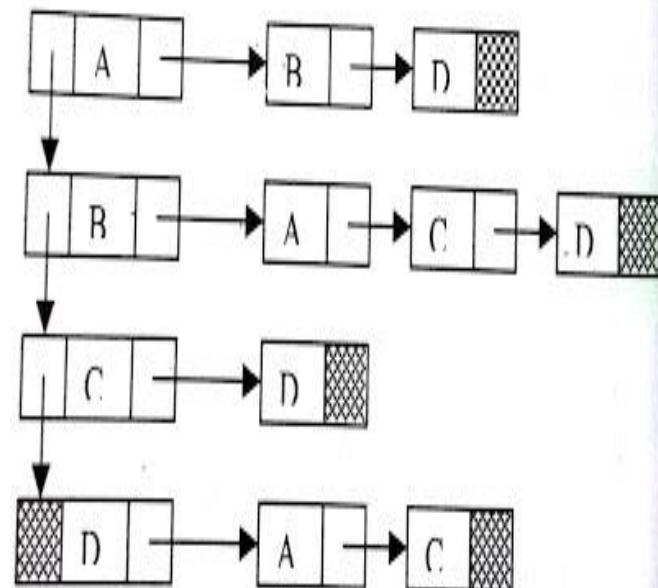
Prof. Shweta Dhawan Chachra

Adjacency List :-

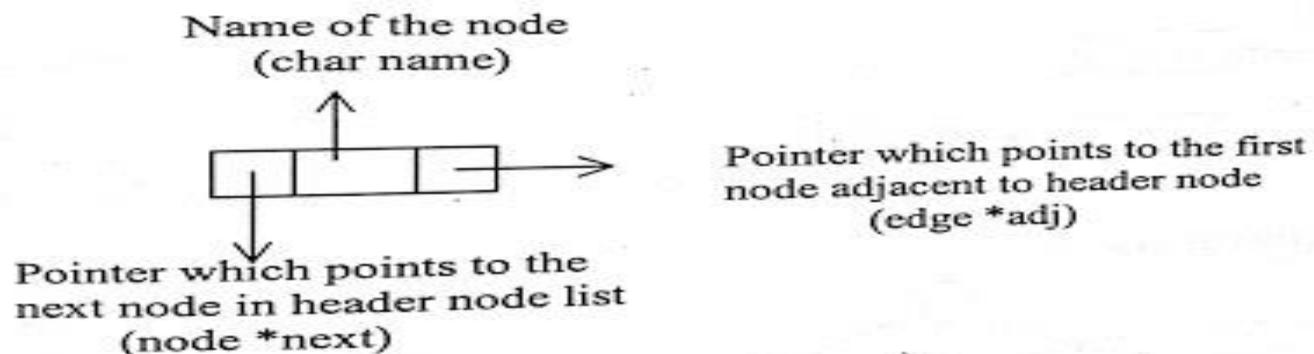
- If the adjacency matrix of the graph is sparse then
 - it is more efficient to represent the graph through adjacency list.
- We will maintain two lists.
- First list will keep **track of all the nodes** in the graph
- Second list will maintain **a list of adjacent nodes for each node**.

Adjacency List :-

- Suppose there are n nodes then we will create one list which will keep information of all nodes in the graph
- After that we will create n lists, where each list will keep information of all adjacent nodes of that particular node.
- Each list has a header node, which will be the corresponding node in the first list.



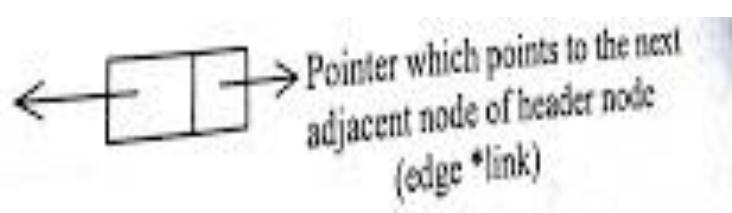
Structure of header node :



```
struct node{  
    struct node *next;  
    char name;  
    struct edge *adj;  
};
```

Structure of edge :

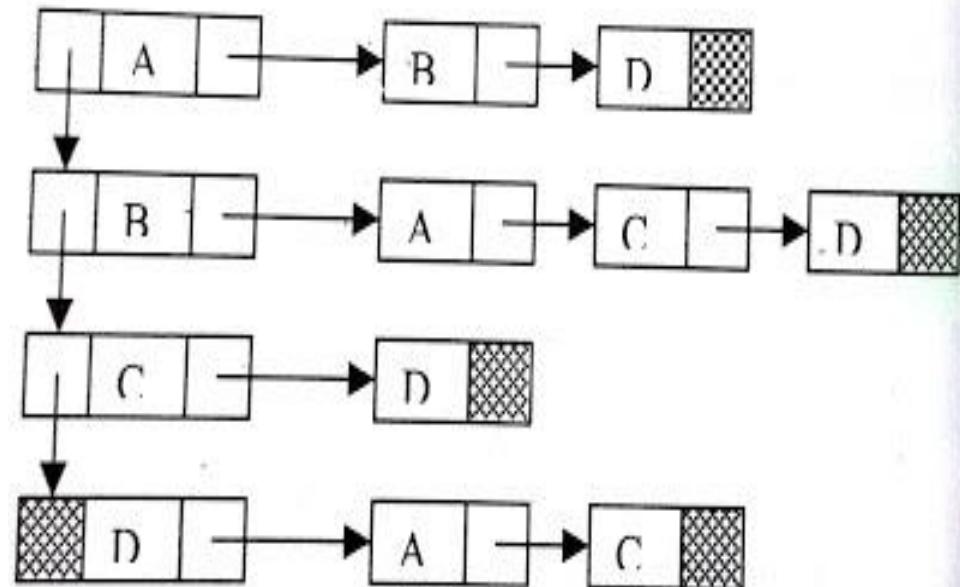
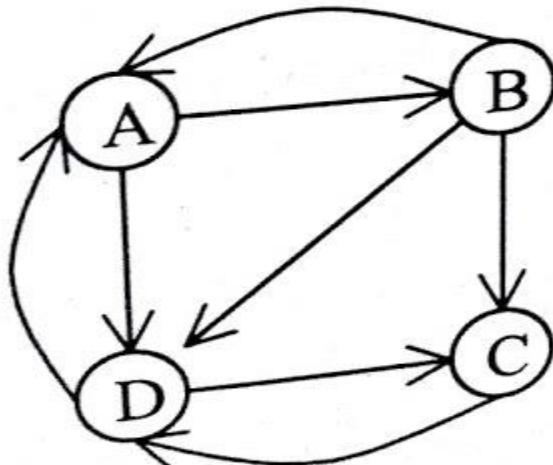
Name of the destination node
of the edge (char dest)



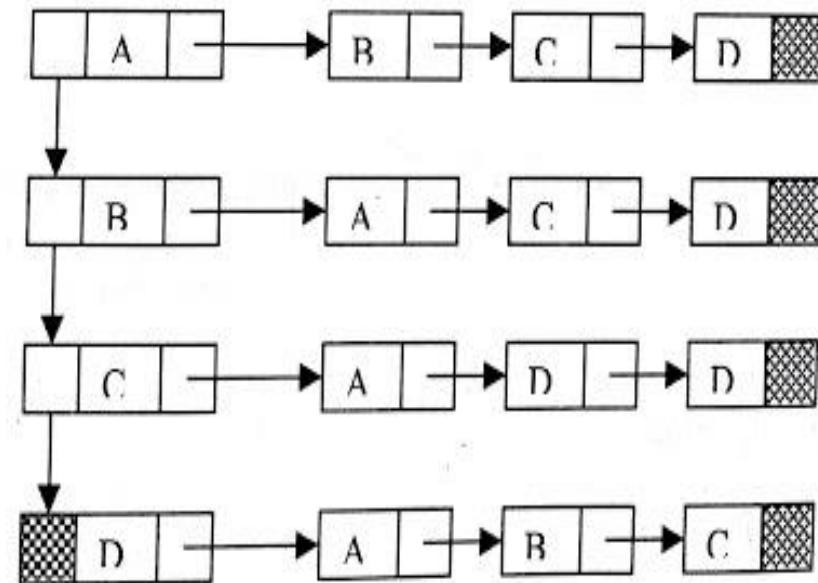
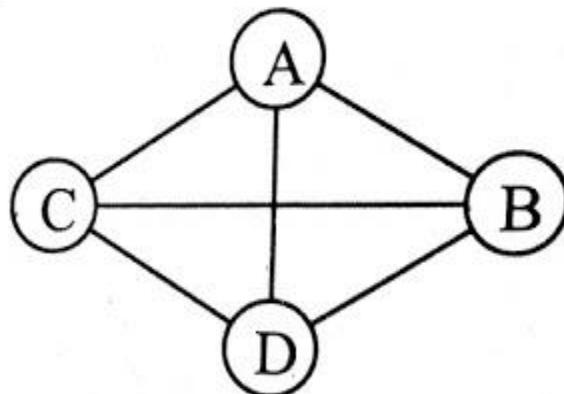
struct edge {
 char dest;
 struct edge *link;
};



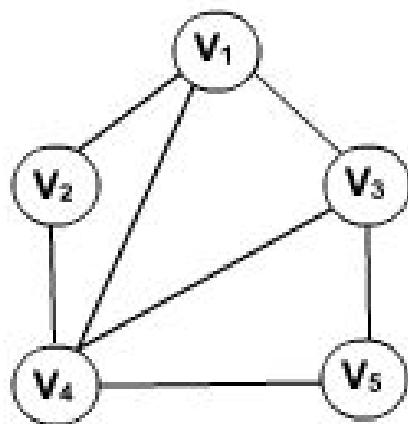
Adjacency list:



Adjacency list:

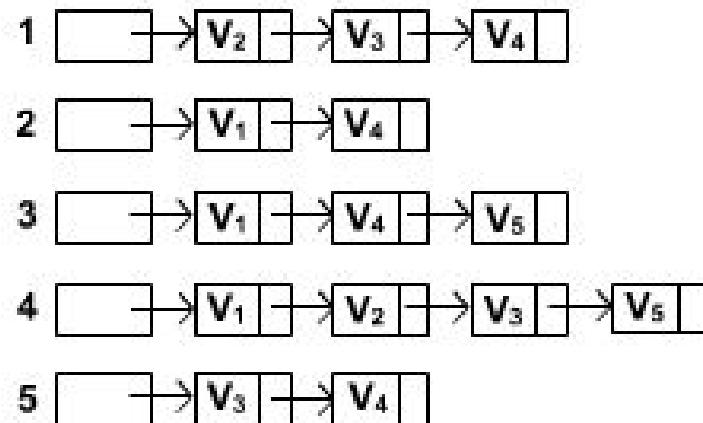
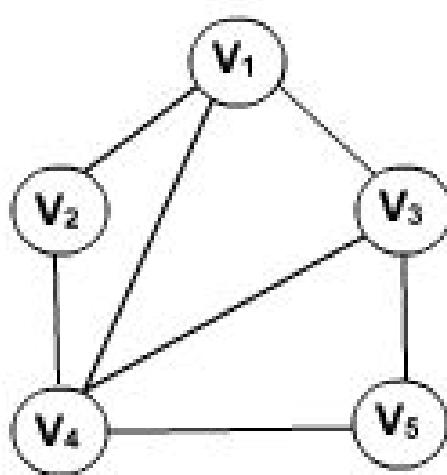


Adjacency Matrix Representation:



	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	0
3	1	0	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

Adjacency Matrix Representation:



Operations on graph

The two main operations on graph will be-

- **Insertion**
- **Deletion**

Here insertion and deletion will be also on two things-

- **On node**
- **On edge**

Insertion in Adjacency Matrix :

- **Node insertion :-**
- Insertion of node requires only addition of one row and one column with zero entries in that row and column.

Insertion in Adjacency Matrix :

- **Node insertion :-**
- Let us take an adjacency matrix-

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

- Suppose we want to add one node E in the graph then we have a need to **add one row and one column with all zero entries for node E.**

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

- **Edge insertion :-**
- ***Insertion of edge requires changing the value 0 into 1 for those particular nodes.***

- **Edge insertion :-**
- Let us take an adjacency matrix-

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

- There is no edge between D to B.
- Suppose we want to insert an edge between D to B, then we have a need to change the **0 entry into 1 at the position 4th row 2nd column.**
- Now the adjacency matrix will be-

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

Deletion in Adjacency Matrix :

- ***Node deletion :-***
- ***Deletion of node requires deletion of that **particular row and column in adjacency matrix for node to be deleted*****
- ***As node deletion requires deletion of all the edges which are connected to that particular node.***

- **Deletion in Adjacency Matrix :**
- **Node deletion :-**
- Let us take an adjacency matrix-

$$\begin{array}{cc} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \left[\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{array} \right] \end{array}$$

- Suppose we want to delete the node *D*, **then 4th row and 4th column of adjacency matrix will be deleted.**
Now the adjacency matrix will be-

$$\begin{array}{cc} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} A \\ B \\ C \end{array} & \left[\begin{array}{ccc} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{array} \right] \end{array}$$

Edge deletion :-

- Deletion of an edge requires **changing the value 1 to 0 for those particular nodes.**

- **Edge deletion :-**
- Let us take an adjacency matrix-

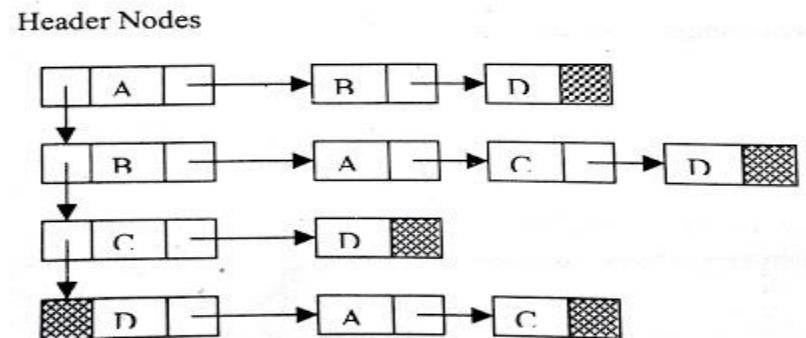
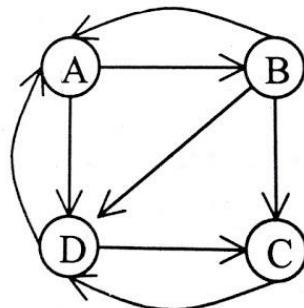
$$\begin{array}{ccccc} & \text{A} & \text{B} & \text{C} & \text{D} \\ \text{A} & \left[\begin{matrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{matrix} \right] \\ \text{B} & & & & \\ \text{C} & & & & \\ \text{D} & & & & \end{array}$$

- To **delete the edge which is in between B and C,**
- Change the **entry 1 to 0 at the position 2nd row, 3rd column.**
- Adjacency matrix will be-

$$\begin{array}{ccccc} & \text{A} & \text{B} & \text{C} & \text{D} \\ \text{A} & \left[\begin{matrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{matrix} \right] \\ \text{B} & & & & \\ \text{C} & & & & \\ \text{D} & & & & \end{array}$$

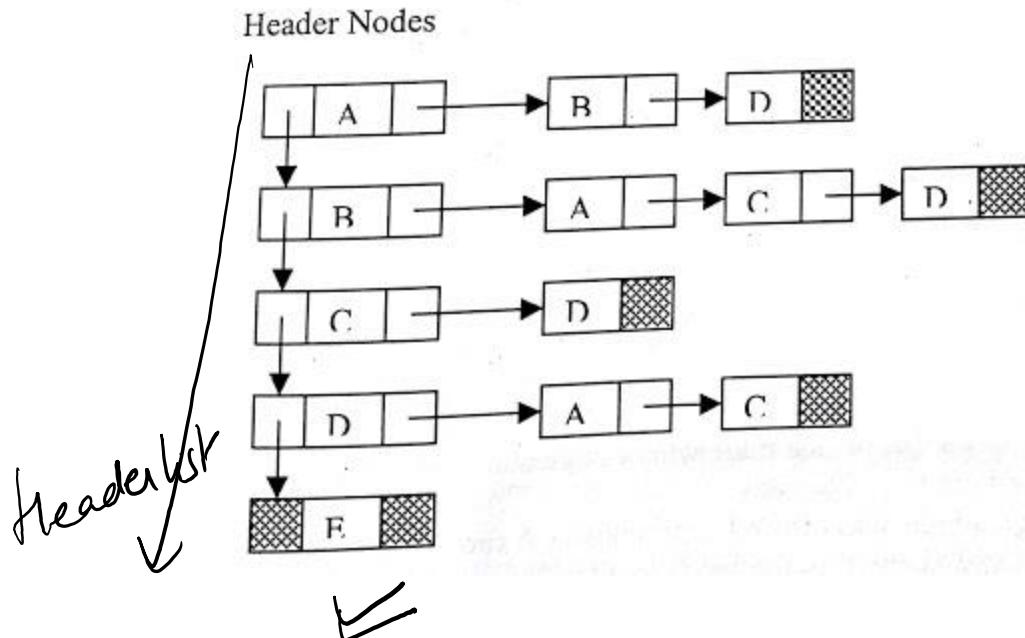
Insertion in Adjacency list

- **Node insertion :-**
- Insertion of node in adjacency list requires only **insertion of that node in header nodes** of adjacency list.
- Let us take a graph G-
- **The adjacency list for this graph will be as-**



Insertion in Adjacency list

- Suppose we want to **insert one node E**
- Addition of node E in header node only.**
- Now the adjacency list will be-

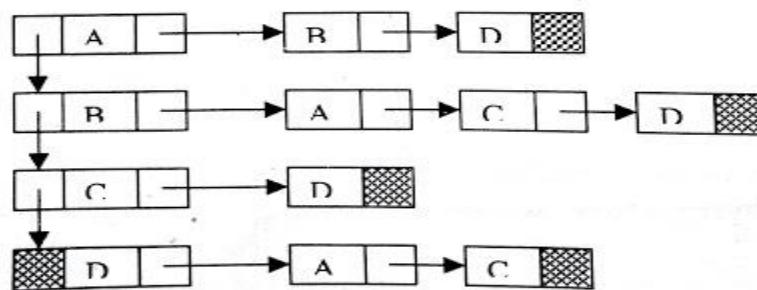


Edge insertion in Adjacency list

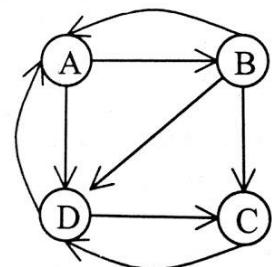
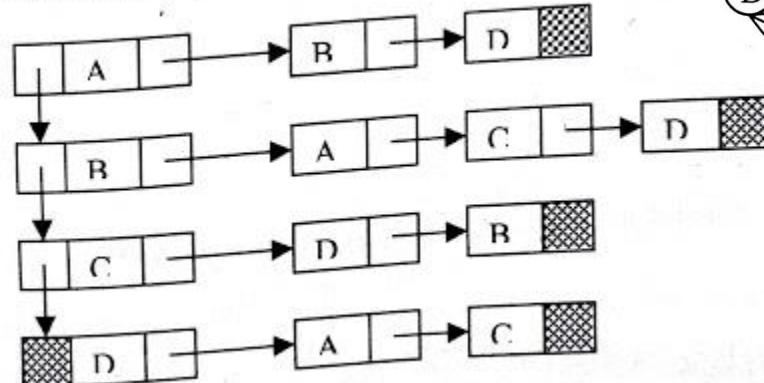
Insertion of an edge

- Requires add operation in the list of the starting node of edge.
- Remember here graph is directed graph.
- In undirected graph, it will be added in the list of both nodes.
- Add the edge which **starts from node C and ends at node B**, then add operation is needed in the list if C node.

Header Nodes



Header Nodes

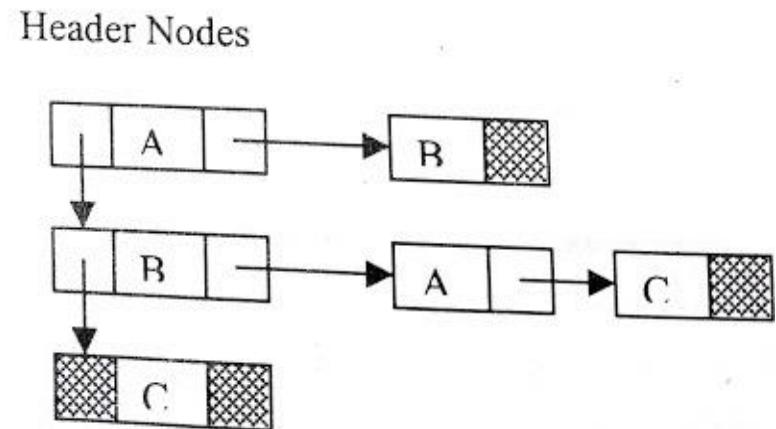
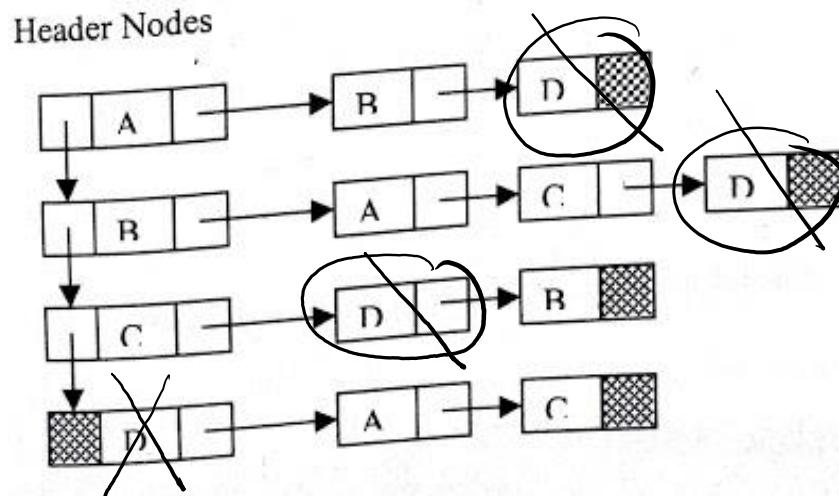


Deletion in Adjacency list :

- **Node deletion :-**
- Deletion of node requires **deletion of that particular node from header node and from the entire list wherever it's coming.**
- Deletion of node from header **will automatically free the list attached to that node.**

Deletion in Adjacency list :

- **Node deletion :-**
- **Delete the node D from graph**
- **Delete node D from header node and from the list of A, B and C nodes.**



Deletion in Adjacency list :

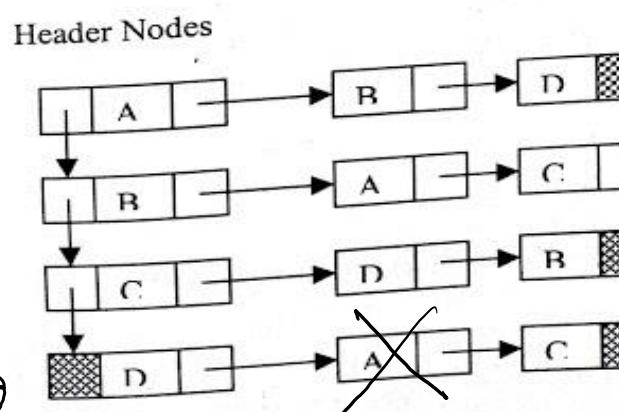
Edge deletion :-

- *Deletion in the list of that node where edge starts, and that element of the list will be deleted where the edge ends.*

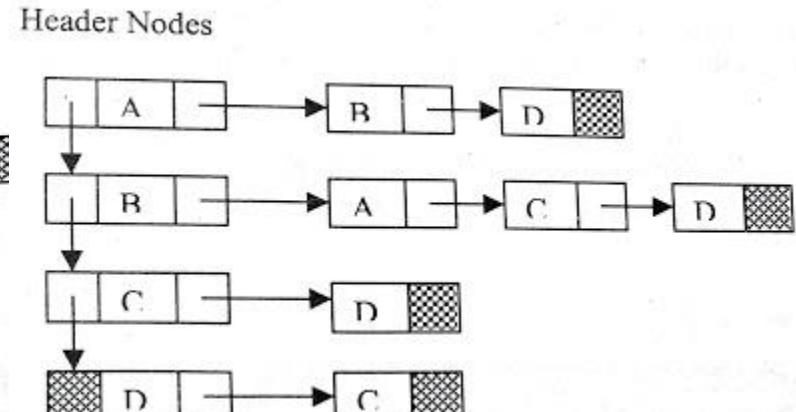
Deletion in Adjacency list :

Edge deletion :-

- Delete the edge which starts from D and ends at A
- Delete in the list of node D and element in the list deleted will be A.



adjacency



Traversal of Graph

The various graph traversals are

- Depth first search (DFS)
- Breadth first search (BFS)

Depth First Search

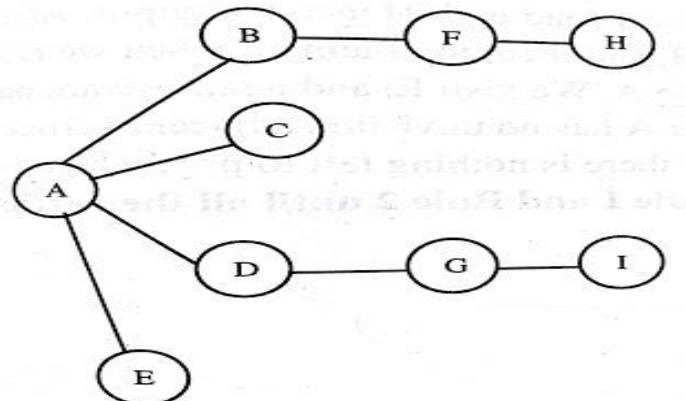
- This method is called depth first search since searching is done forward (deeper) from current node.
- The distance from start vertex is called depth

Depth First Search

- Choose a **starting node**, **mark it visited**, **traverse it** **Push it into the stack**
- Rule 1 : If possible, **visit an adjacent** unvisited vertex, **mark it visited**, **traverse it and push it** on the stack.
- Rule 2 : If Rule 1 fails, then **pop a vertex** off the stack **follow Rule 1** from it.
- i.e. For the Vertex on the top of the stack, If there is no unvisited adjacent node only then we pop it.
- Rule 3 : **Repeat Rule 1 and Rule 2 until the all the vertices are visited.**

○ Example of Depth First Search :

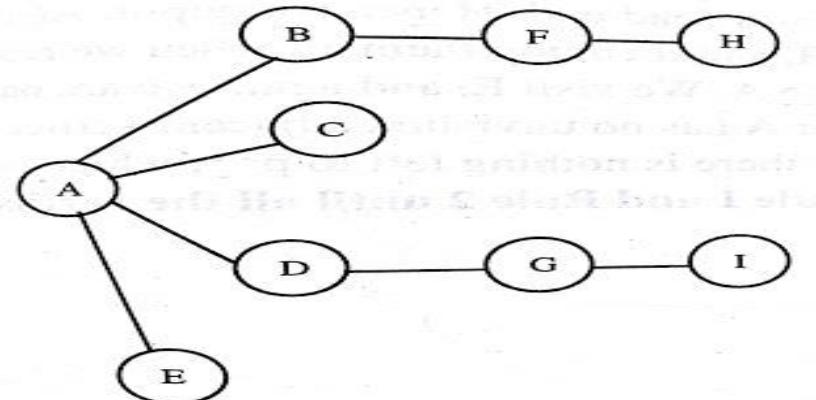
Let us consider the following graph-



- The depth-first search (DFS) uses **a stack to remember** where it should go when it reaches a dead end.

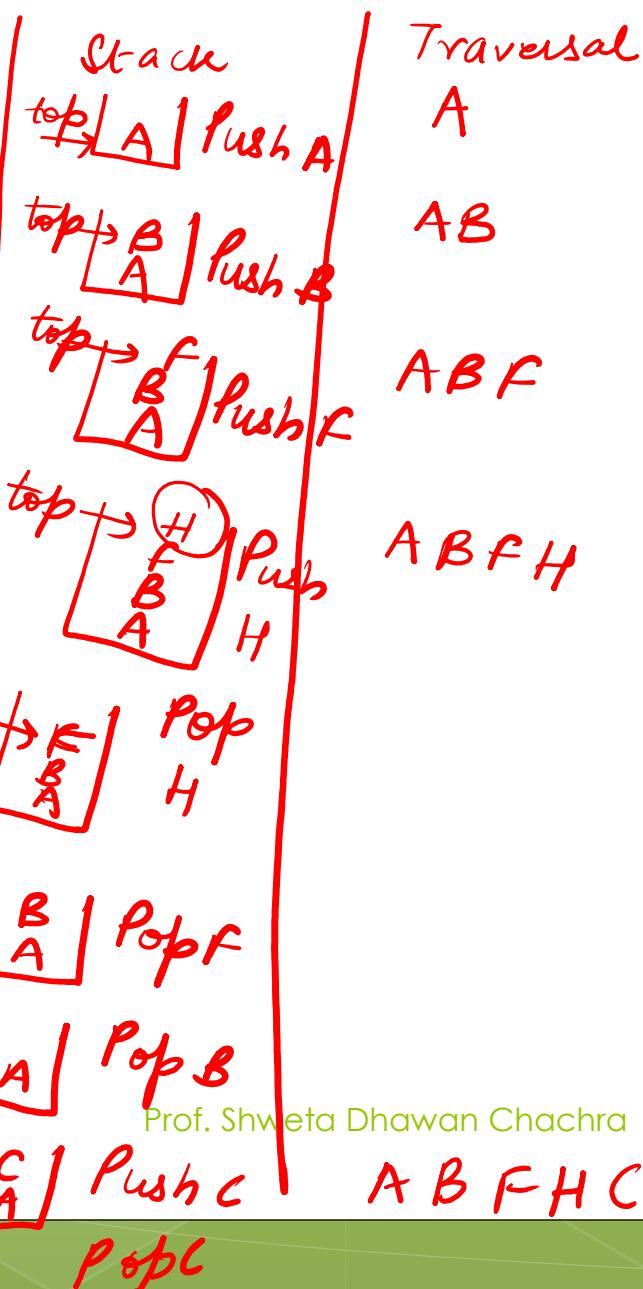
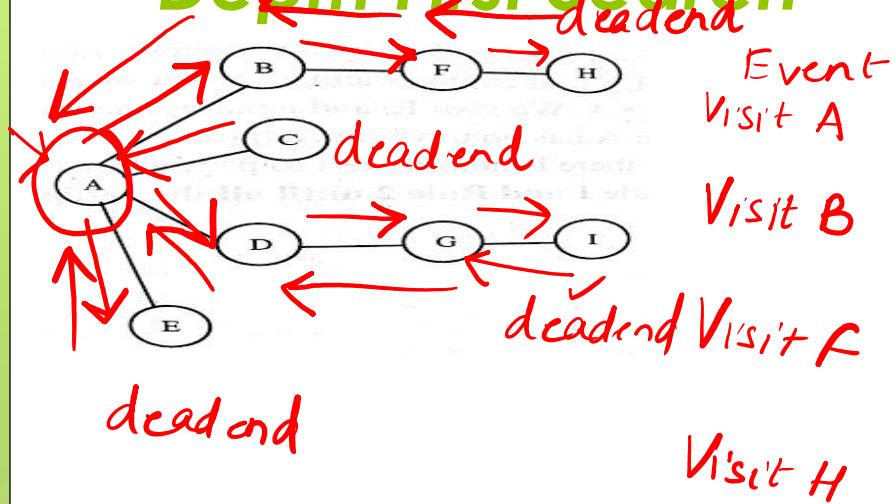
Depth First Search

- To carry out the depth-first search, we pick a starting point, vertex A.



- We can do three things:
- Visit the vertex
- Push it onto stack so we can remember it, and
- Mark it, so it is not visited again.

Depth First Search



Visit D

Visit G

Visit I

Visit E

Stack



Pop C



Push D



Push G



Push I



Pop I

Pop G

Pop D



Push E

Pop E

Pop A

Traversal

A B F H C

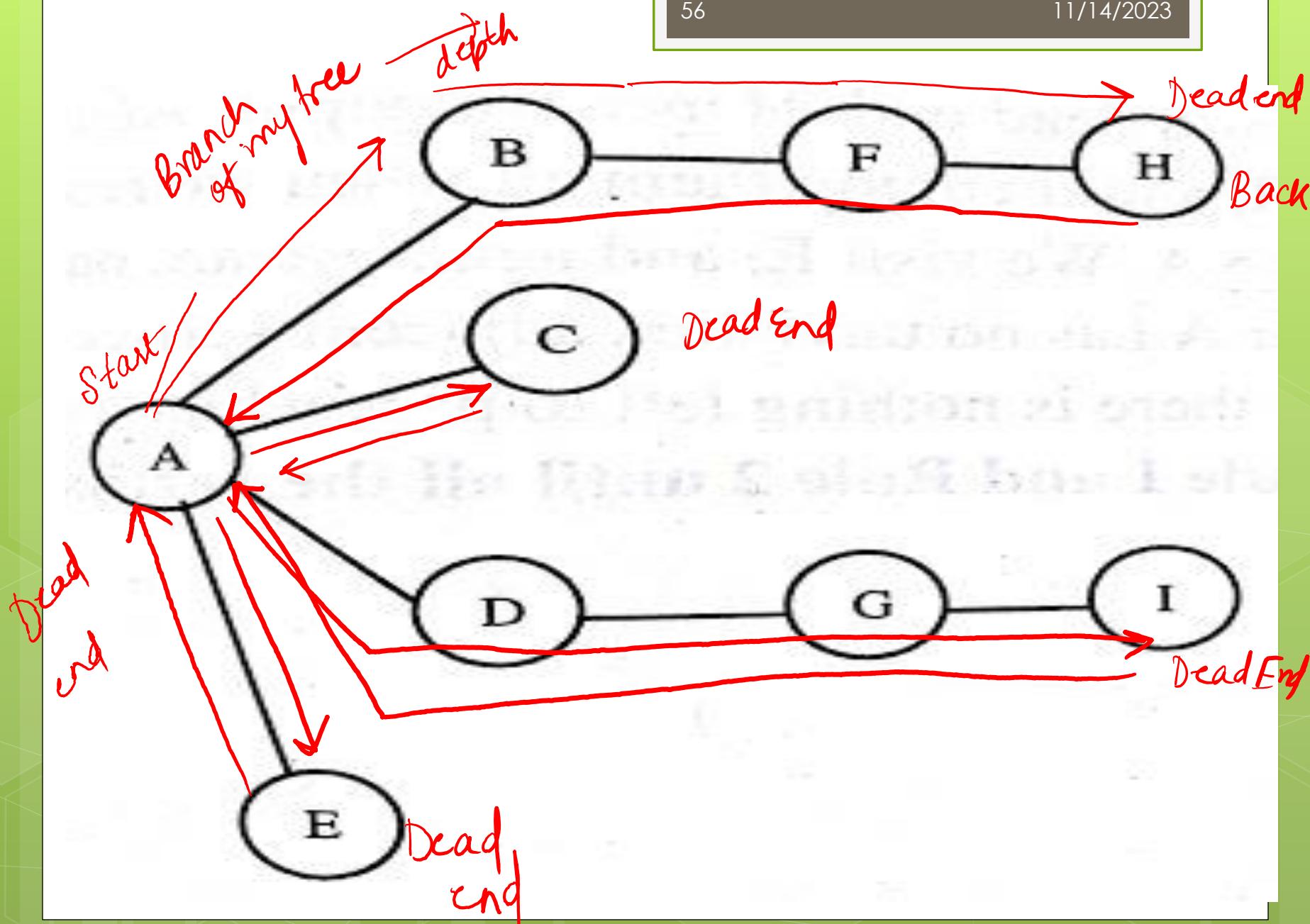
A B F H C D

A B F H C D G

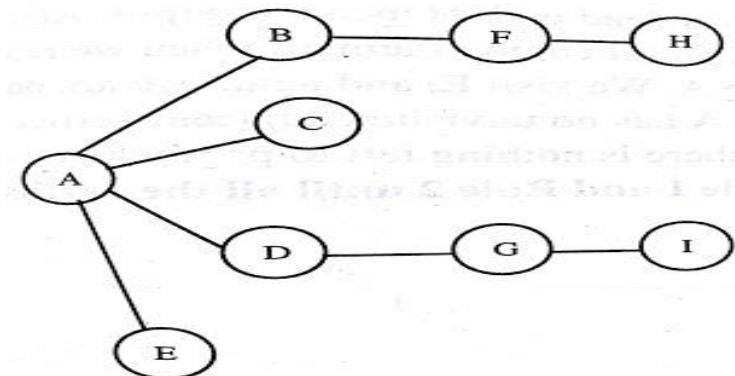
A B F H C D G I

Depth first traversal

A B F H C D G I E



Depth First Search



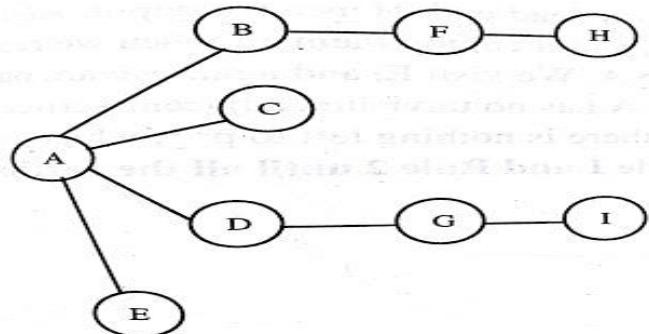
- Assuming the vertex are visited in alphabetical manner
- Visit the next vertex connected to A.
- It is B
- The vertex **B is pushed** into the stack and marked as visited, **traverse it**
We are now at B.
- The adjacent unvisited vertex of B i.e. **F is visited, traversed, pushed** into stack
- We call this process Rule 1

Event	Stack	DFS
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		

Depth First Search

Rule 1

If possible, visit an adjacent unvisited vertex, mark it visited, and push it on the stack.



- Applying Rule 1 the vertex **H** is visited.
- Now from here there is no further unvisited adjacent vertex that we can visit.
- This condition is called **dead end**. Now we apply Rule 2.

Event	Stack	DFS
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		

Depth First Search

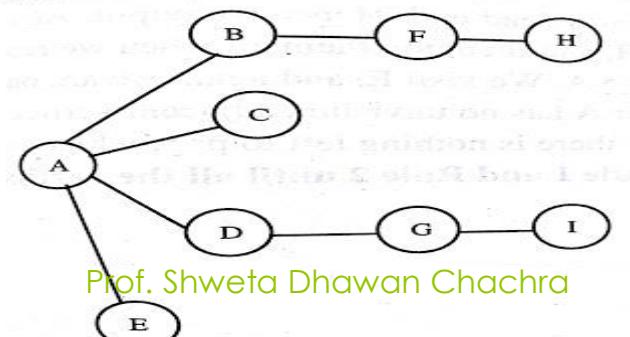
Rule 2

- If Rule 1 fails, then pop a vertex off the stack follow Rule 1 from it.

Depth First Search

- Following Rule 2, we **pop H** off the stack, which brings us back to F.
- F also has no unvisited adjacent vertices**, so we **pop it** also.
- It applies to **B** also.
- Now only A is left on the stack. A, however, **does have unvisited adjacent vertices**, so we now **visit C and push C**.
- But C also becomes a dead end.**
- Hence we **pop it**.
- Again we are **left with A**.

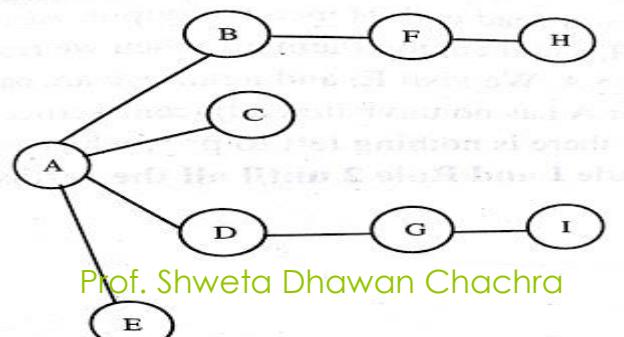
Event	Stack	DFS
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		



Depth First Search

- We visit D, G, I, and then **pop them all** when we reach a **dead end at I**
- Now we are **back to A**
- We visit E, and again we are **back to A**
This time, however **A has no unvisited adjacent vertices.**
- So we **pop it off the stack.**
- But now there is nothing left to pop, which brings us to Rule 3.
- So Stop**

Event	Stack	DFS
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		

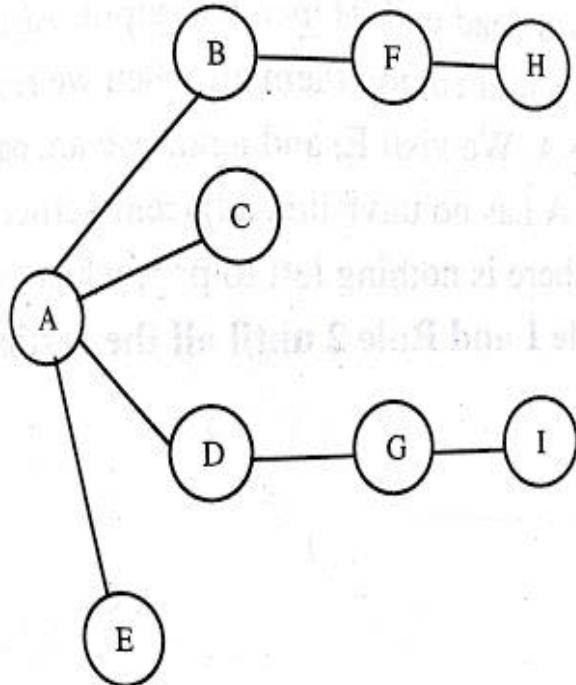


Depth First Search

- **Rule 3 : Repeat Rule 1 and Rule 2 until the all the vertices are visited.**

Depth First Search

- The table below shows the stack contents at each push and pop operations:



<u>Event</u>	<u>Stack</u>	<u>DFS</u>
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		

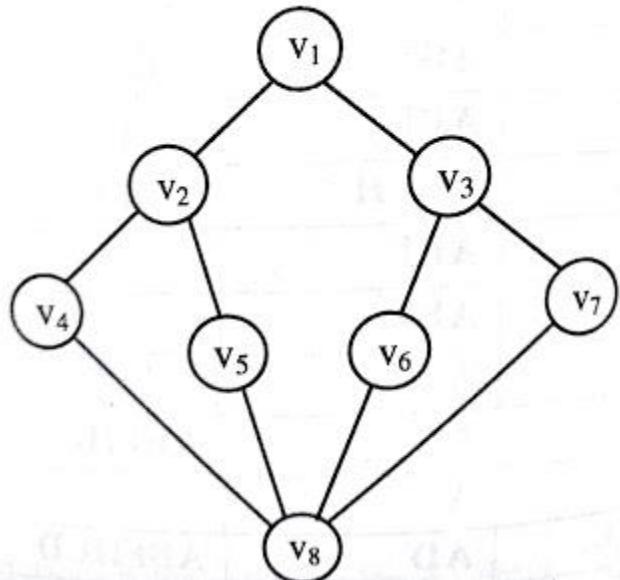
Required DFS is ABFHCDGIE

Prof. Shweta Dhawan Chachra

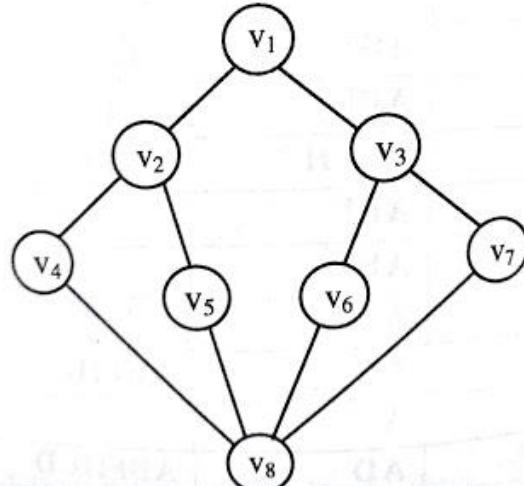
DFS-Procedure using Stack-

- Use array implementation of stack to keep the unvisited neighbors of the node
 - Initially stack is empty and top =-1
- Take a Boolean Array, visited[n]
- In which value ,
 - visited[i]=false, If node has not been visited
 - visited[i]=true, If node has been visited
- Initially visited[i]=false where i=1 to n, n is total no of nodes

Perform DFS



Perform DFS

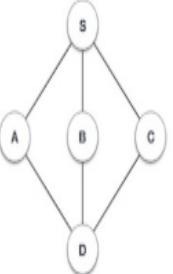
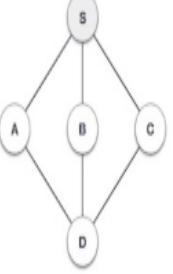
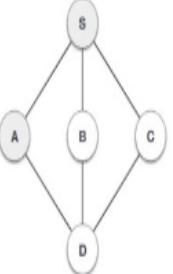
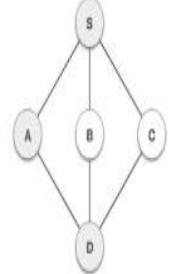
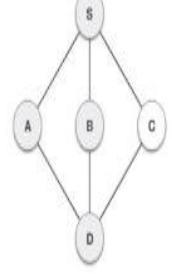
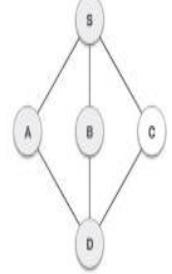
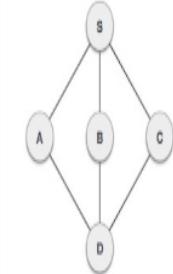


<u>Event</u>	<u>Stack</u>	<u>DFS</u>
Visit V_1	V_1	V_1
Visit V_2	$V_1 V_2$	$V_1 V_2$
Visit V_4	$V_1 V_2 V_4$	$V_1 V_2 V_4$
Visit V_8	$V_1 V_2 V_4 V_8$	$V_1 V_2 V_4 V_8$
Visit V_5	$V_1 V_2 V_4 V_8 V_5$	$V_1 V_2 V_4 V_8 V_5$
Pop V_5	$V_1 V_2 V_4 V_8$	
Visit V_6	$V_1 V_2 V_4 V_8 V_6$	$V_1 V_2 V_4 V_8 V_5 V_6$
Visit V_3	$V_1 V_2 V_4 V_8 V_6 V_3$	$V_1 V_2 V_4 V_8 V_5 V_6 V_3$
Visit V_7	$V_1 V_2 V_4 V_8 V_6 V_3 V_7$	$V_1 V_2 V_4 V_8 V_5 V_6 V_3 V_7$
Pop V_7	$V_1 V_2 V_4 V_8 V_6 V_3$	
Pop V_3	$V_1 V_2 V_4 V_8 V_6$	
Pop V_6	$V_1 V_2 V_4 V_8$	
Pop V_8	$V_1 V_2 V_4$	
Pop V_4	$V_1 V_2$	
Pop V_2	V_1	
Pop V_1		
Done		

The required DFS is $V_1 V_2 V_4 V_8 V_5 V_6 V_3 V_7$

Prof. Shweta Dhawan Chachra

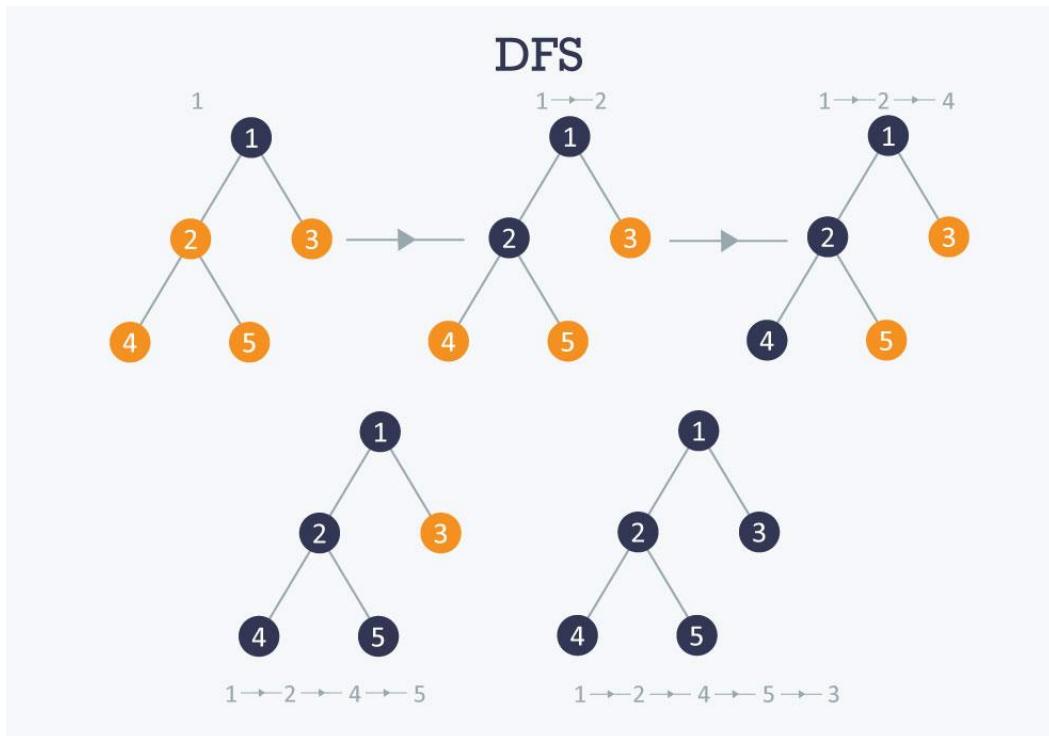
Depth-first search using Backtracking

Step	Traversal	Description
1	 	Initialize the stack.
2	 	Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3	 	Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4	 	Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5	 	We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6	 	We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7	 	Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

Prof. Shweta Dhawan Chachra

DFS

DFS of a graph is analogous to the pre-order traversal of an ordered tree.



Recursive Defn of DFS

Algorithm DFS (G)

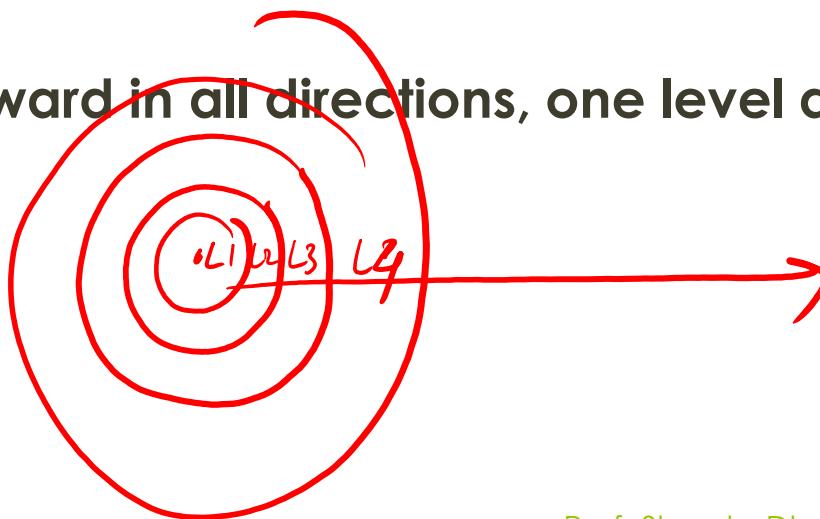
```
for i = 1 to n do // Initialize all vertices are unvisited  
    status[i] = unvisited  
    parent[i] = NULL  
for i = 1 to n do  
    if (status[i] == unvisited) // If there exits an unvisited vertex,  
        start traversal  
        DF-Travel(i)
```

Algorithm DF-Travel (v)

```
status[v] = visited  
for each vertex u adjacent to v do  
    if status[u] == unvisited then  
        parent[u] = v  
        DF-Travel ( u )
```

Breadth First Search

- BFS of a graph is analogous to level-by-level traversal of an ordered tree.
- This method is called breadth first search, since it **works outward from a center point, much like the ripples created when throwing a stone into a pond.**
- It moves **outward in all directions, one level at a time.**



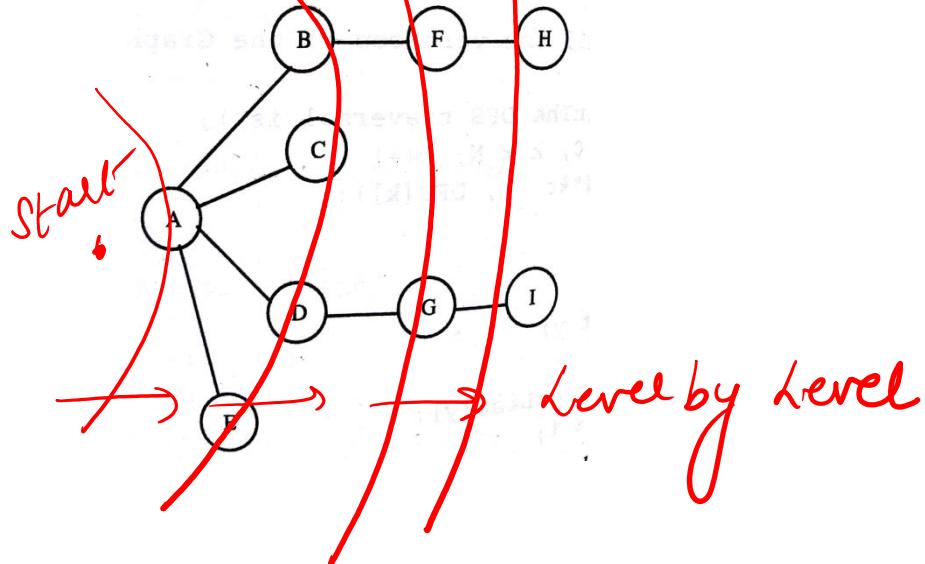
Prof. Shweta Dhawan Chachra

Breadth First Search

- Choose the **start node, mark it visited, traverse it, enqueue it**
- Rule 1 : Visit all the next unvisited vertices (if any) that is adjacent to the current vertex(**At Front end**), and **insert them** into a queue **one at a time on every visit.**
- Rule 2 : If there is **no unvisited vertex**, **remove a vertex** from the Queue and **make it the current vertex and then follow Rule 1.**
- Rule 3 : Repeat Rule 1 and Rule 2 until the all the vertices visited.

Breadth First Search

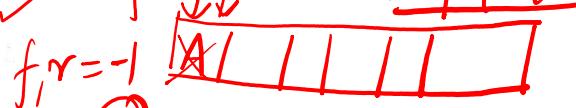
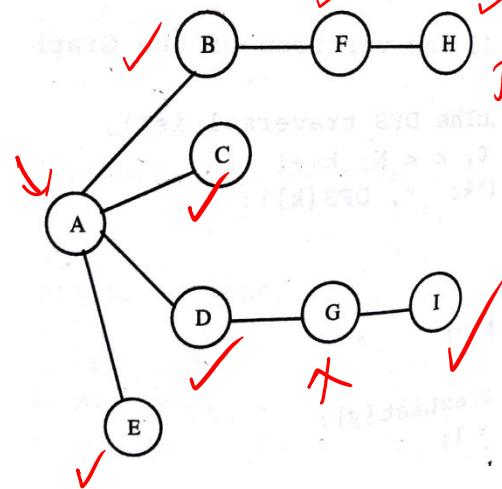
- Let us consider the following graph-



- In BFS, all the vertices adjacent to start vertex are visited, In first pass
- This kind of search is implemented **using a Queue**

Breadth First Search

Empty queue 73 $f, r = -1$

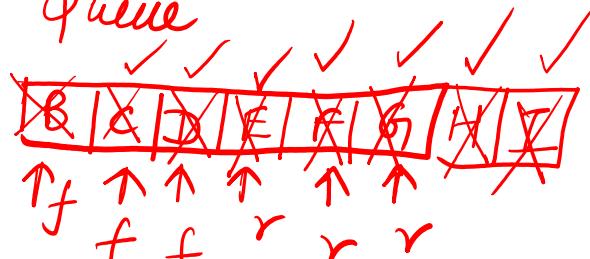


- ① Visit A
- ② Enqueue(A)
- ③ Dequeue(A)

check for any unvisited

adjacent nodes
→ visiting them
→ enqueue them in the

queue



Deg(B)

Check for unvisited adj nodes

Visit F, Eng(F)

Deg(C)

Deg(D), Eng(G), Visit G

Deg(E), Deg(F), Visit H, Eng(H)

Traversal

A B C D EFG H I

Deg(B)

Visit I

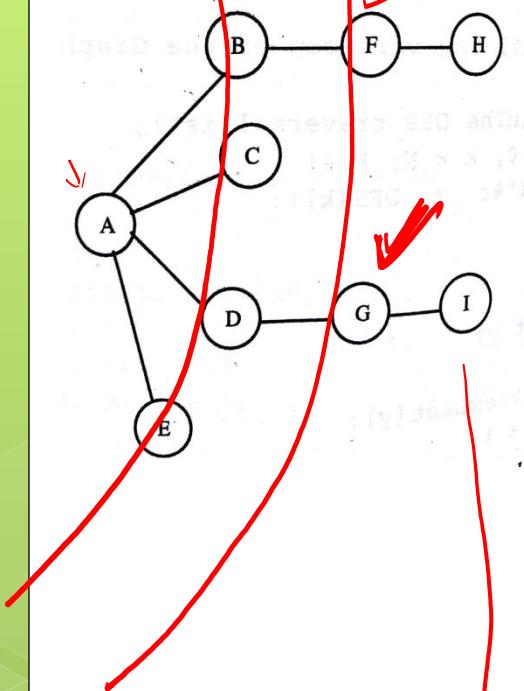
Eng(I)

Deg(H)

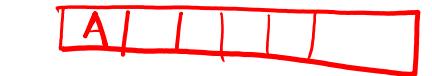
Deg(I)

Breadth First Search

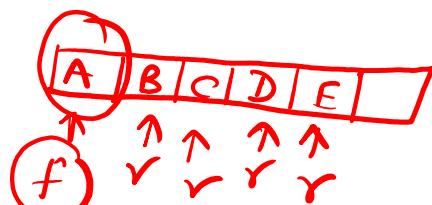
Traversal



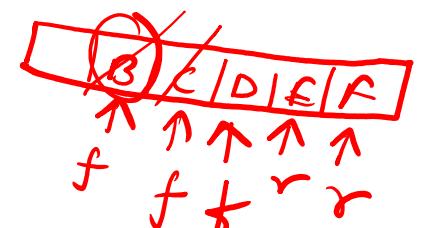
Deq(F)
 Test
 Visit I
 ABCDEF
 GH
 Deq G
 Deq H
 Deq(I)



f, r



f, r



f, f, r, r



f, f, r, r



f, f

Enqueue(A)

Test

Enq(B)

Enq(C)

Enq(D)

Enq(E)

Test

Deq(A)

Test

Test

Deq(B)

Test

Test

Deq(C)

Test

Enq(G)

Test

Deq(D)

Test

Deq(E)

Test

Test

Visit A

A

Visit B

AB

Visit C

ABC

Visit D

ABCD

Visit E

ABCDE

Visit F

ABCDEF

Visit G

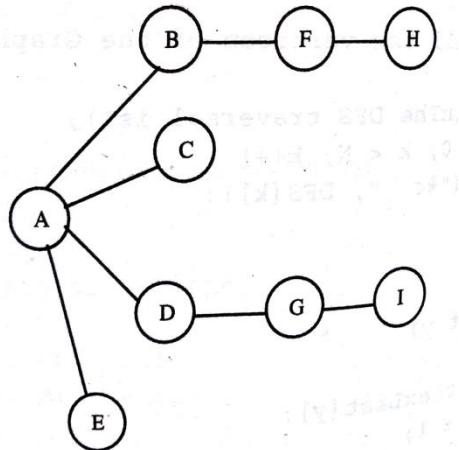
ABCDEFG

Visit H

ABCDEFGHI

Breadth First Search

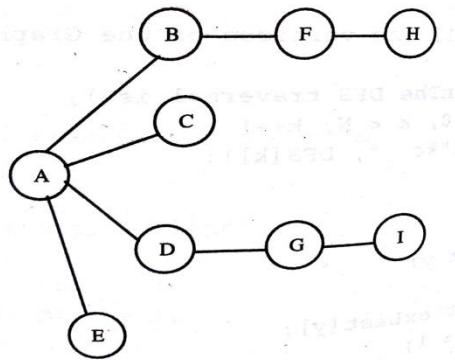
- Let us consider the following graph-



- Pick a starting point,
- Lets **start with vertex A**.
- We **mark it as visited, traverse it**
- Then rules are applied

Event	Queue (Front to Rear)	BFS
Visit A		A
Visit B	B	AB
Visit C	B C	ABC
Visit D	BCD	ABCD
Visit E	BCDE	ABCDE
Remove B	CDE	
Visit F	CDEF	ABCDEF
Remove C	DEF	
Remove D	EF	
Visit G	EFG	ABCDEFG
Remove E	FG	
Remove F	G	
Visit H	GH	ABCDEFGHI
Remove G	H	
Visit I	HI	
Remove H	I	
Remove I		
Done		

Breadth First Search



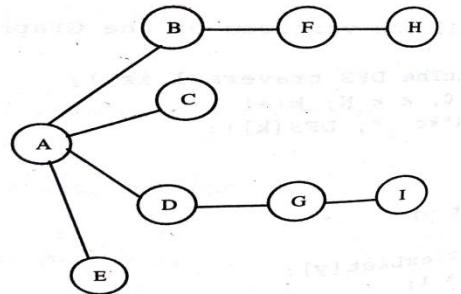
- Visit all the vertices adjacent to A, inserting each one into the queue during each visit.
- We visit A, B, C, D and E.
- At this point the queue (from front to rear) contains **B, C D and E**.
- There are **no more unvisited vertices adjacent to A**.

<u>Event</u>	<u>Queue (Front to Rear)</u>	<u>BFS</u>
Visit A		A
Visit B	B	AB
Visit C	B C	ABC
Visit D	BCD	ABCD
Visit E	BCDE	ABCDE
Remove B	CDE	
Visit F	CDEF	ABCDEF
Remove C	DEF	
Remove D	EF	
Visit G	EFG	ABCDEFG
Remove E	FG	
Remove F	G	
Visit H	GH	ABCDEFGH
Remove G	H	
Visit I	HI	ABCDEFGHI
Remove H	I	
Remove I		
Done		

Rule 1 : Visit all the next unvisited vertices (if any) that is adjacent to the current vertex, and insert them into a queue one at a time on every visit.

Prof. Shweta Dhawan Chachra

Breadth First Search



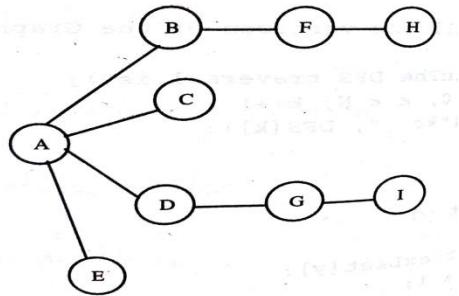
- So we **remove B** from the queue and **look for vertices adjacent to it**
- We find **F**, so we **insert it in the queue**.
- There are **no more unvisited vertices adjacent to B**,

<u>Event</u>	<u>Queue (Front to Rear)</u>	<u>BFS</u>
Visit A		A
Visit B	B	AB
Visit C	B C	ABC
Visit D	BCD	ABCD
Visit E	BCDE	ABCDE
Remove B	CDE	
Visit F	CDEF	ABCDEF
Remove C	DEF	
Remove D	EF	
Visit G	EFG	ABCDEFG
Remove E	FG	
Remove F	G	
Visit H	GH	ABCDEFGH
Remove G	H	
Visit I	HI	ABCDEFGHI
Remove H	I	
Remove I		
Done		

Rule 2 : If there is no unvisited vertex, remove a vertex from the Queue and make it the current vertex and then follow Rule 1.

Prof. Shweta Dhawan Chachra

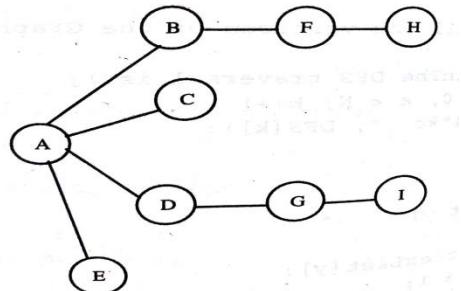
Breadth First Search



- Remove **C** from the queue, It has **no unvisited vertices**
- Remove **D** from the queue
- **D** has an **unvisited vertex G** adjacent to it, insert **G**
- Remove **E** as it has no adjacent vertices

Event	Queue (Front to Rear)	BFS
Visit A		A
Visit B	B	AB
Visit C	B C	ABC
Visit D	BCD	ABCD
Visit E	BCDE	ABCDE
Remove B	CDE	
Visit F	CDEF	ABCDEF
Remove C	DEF	
Remove D	EF	
Visit G	EFG	ABCDEFG
Remove E	FG	
Remove F	G	
Visit H	GH	ABCDEFGHI
Remove G	H	
Visit I	HI	
Remove H	I	
Remove I		
Done		

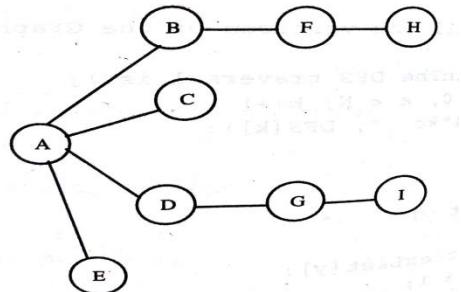
Breadth First Search



- Now the **queue is FG**.
- **Remove F**, F has an **unvisited node H**, **insert H** in Queue
- **Remove G**, G has an **unvisited node I**, **insert I** in queue

<u>Event</u>	<u>Queue (Front to Rear)</u>	<u>BFS</u>
Visit A		A
Visit B	B	AB
Visit C	B C	ABC
Visit D	BCD	ABCD
Visit E	BCDE	ABCDE
Remove B	CDE	
Visit F	CDEF	ABCDEF
Remove C	DEF	
Remove D	EF	
Visit G	EFG	ABCDEFG
Remove E	FG	
Remove F	G	
Visit H	GH	ABCDEFGHI
Remove G	H	
Visit I	HI	
Remove H	I	
Remove I		
Done		

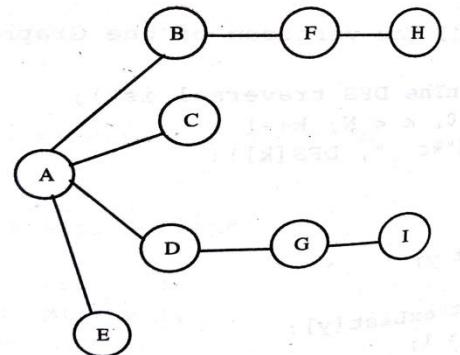
Breadth First Search



- Now the **queue is HI**
- **Visit H, Remove H**
- **Visit I, Remove I**

<u>Event</u>	<u>Queue (Front to Rear)</u>	<u>BFS</u>
Visit A		A
Visit B	B	AB
Visit C	B C	ABC
Visit D	BCD	ABCD
Visit E	BCDE	ABCDE
Remove B	CDE	
Visit F	CDEF	ABCDEF
Remove C	DEF	
Remove D	EF	
Visit G	EFG	ABCDEFG
Remove E	FG	
Remove F	G	
Visit H	GH	ABCDEFGHI
Remove G	H	
Visit I	HI	
Remove H	I	
Remove I		
Done		

Breadth First Search



- While removing each of these and found no adjacent unvisited vertices,
- The queue is empty.
- BFS terminates

Event	<u>Queue (Front to Rear)</u>	<u>BFS</u>
Visit A		A
Visit B	B	AB
Visit C	B C	ABC
Visit D	BCD	ABCD
Visit E	BCDE	ABCDE
Remove B	CDE	
Visit F	CDEF	ABCDEF
Remove C	DEF	
Remove D	EF	
Visit G	EFG	ABCDEFG
Remove E	FG	
Remove F	G	
Visit H	GH	ABCDEFGHI
Remove G	H	
Visit I	HI	
Remove H	I	
Remove I		
Done		

BFS-Procedure-using Queue

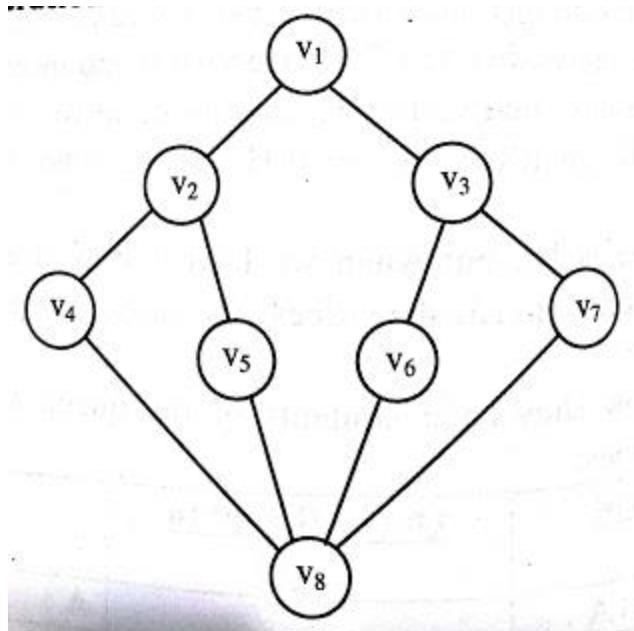
- Use array implementation of queue to keep the unvisited neighbors of the node
 - Initially Queue is empty, front=-1, rear=-1 $f, r = -1$
- Take a Boolean Array, visited[n]
- In which value ,
 - visited[i]=false, If node has not been visited
 - visited[i]=true, If node has been visited
- Initially visited[i]=false where i=1 to n, n is total no of nodes

for all nodes
not visited

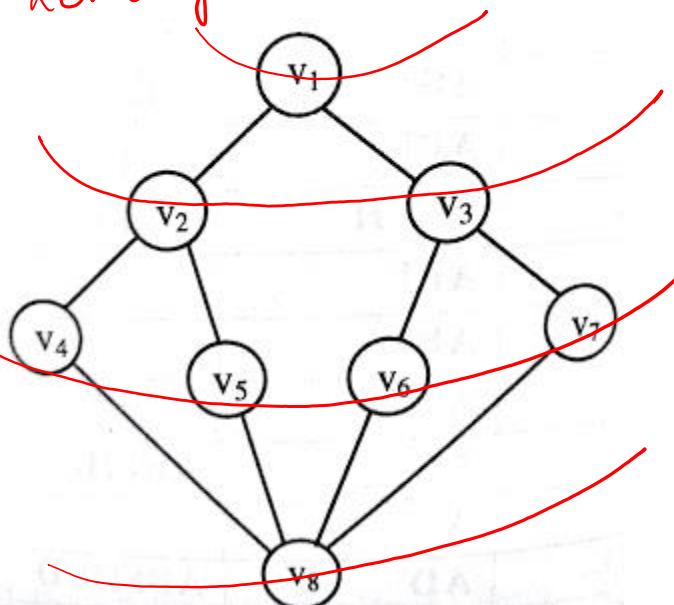
BFS- Procedure using Queue

- 1) Insert starting node into the queue and traverse it, make visited[i]=true
- 2) Delete front element from the queue and insert all its unvisited neighbors into the queue at the end, and traverse them. Also make the value of visited array true for these nodes
- 3) Repeat step 2 until the queue is empty

Perform BFS



Perform BFS
Level by Level



<u>Event</u>	<u>Queue (Front to Rear)</u>	<u>BFS</u>
Visit V ₁		V ₁
Visit V ₂	V ₂	V ₁ V ₂
Visit V ₃	V ₂ V ₃	V ₁ V ₂ V ₃
Remove V ₂	V ₃	
Visit V ₄	V ₃ V ₄	V ₁ V ₂ V ₃ V ₄
Visit V ₅	V ₃ V ₄ V ₅	V ₁ V ₂ V ₃ V ₄ V ₅
Remove V ₃	V ₄ V ₅	
Visit V ₆	V ₄ V ₅ V ₆	V ₁ V ₂ V ₃ V ₄ V ₅ V ₆
Visit V ₇	V ₄ V ₅ V ₆ V ₇	V ₁ V ₂ V ₃ V ₄ V ₅ V ₆ V ₇
Remove V ₄	V ₅ V ₆ V ₇	
Visit V ₈	V ₅ V ₆ V ₇ V ₈	V ₁ V ₂ V ₃ V ₄ V ₅ V ₆ V ₇ V ₈
Remove V ₅	V ₆ V ₇ V ₈	
Remove V ₆	V ₇ V ₈	
Remove V ₇	V ₈	
Remove V ₈		
Done		

The required BFS is V₁ V₂ V₃ V₄ V₅ V₆ V₇ V₈.

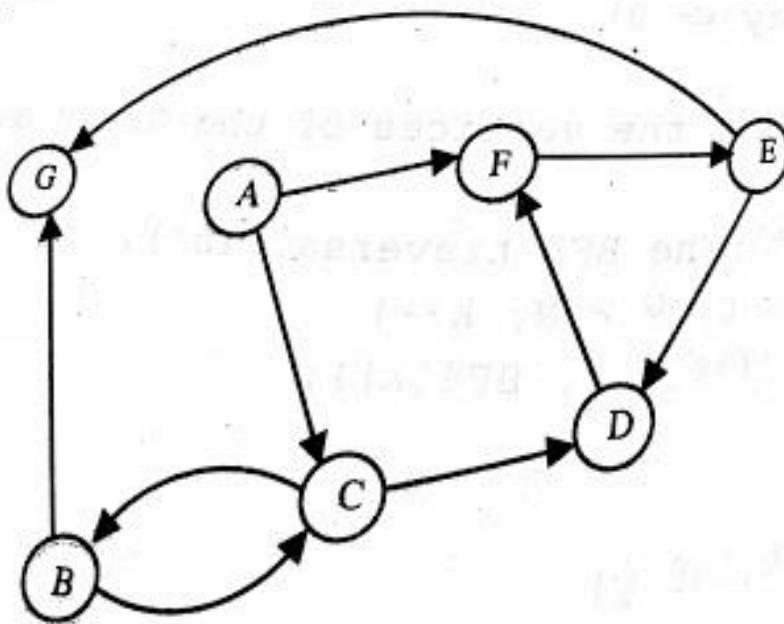
Prof. Shweta Dhawan Chachra

Find DFS in a di-graph

- The rules for a DFS traversal of a graph same as for undirected graph
- Difference-Adjacent nodes for a node A exists **only if there exists an edge starting from A and incident to any other node**

Find DFS & BFS in a di-graph

- Let us consider the following di-graph-

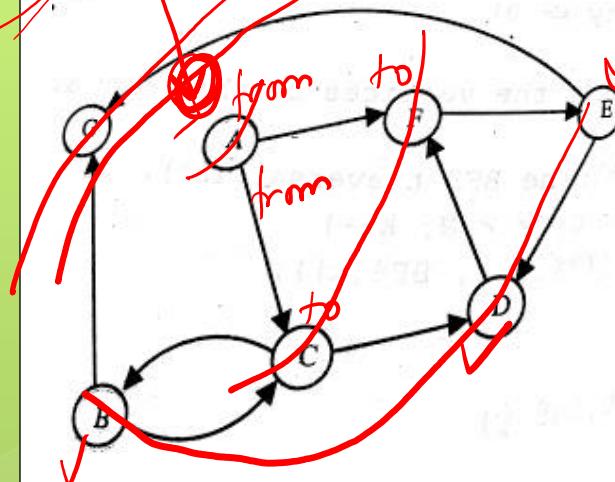


Covering nodes in Alphabetical Order

Prof. Shweta Dhawan Chachra

~~dead node~~

Find DFS & BFS in a di-graph



Operations

Eng(A)
visit

Deq(A)

Eng(E), visit
Eng(F), visit

Deq(C)

Eng(B), ✓

Eng(D), ✓

Deq(F)

Eng(E), ✓

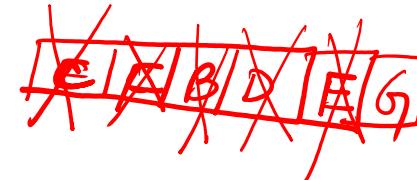
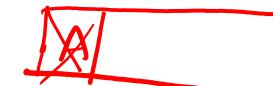
Deq(B)

Eng(G), visiting

Deq(D)

Deq(E) Deq(h) Queue is empty

Queue



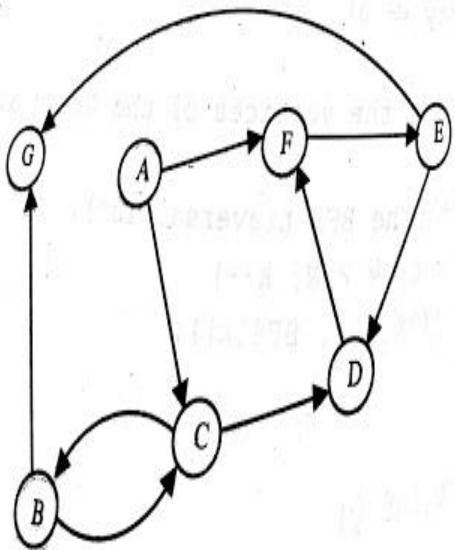
Traversal

ACF B DEG
✓

1

Find DFS in a di-graph

- The table below shows the stack contents at each push and pop



Event	Stack	DFS
Visit A	A	A
Visit C	AC	AC
Visit B	ACB	ACB
Visit G	ACBG	ACBG
Pop G	ACB	
Pop B	AC	
Visit D	ACD	ACBGD
Visit F	ACDF	ACBGDF
Visit E	ACDFE	ACBGDFE
Pop E	ACDF	
Pop F	ACD	
Pop D	AC	
Pop C	A	
Pop A	NULL	

- The required DFS is A C B G D F E.

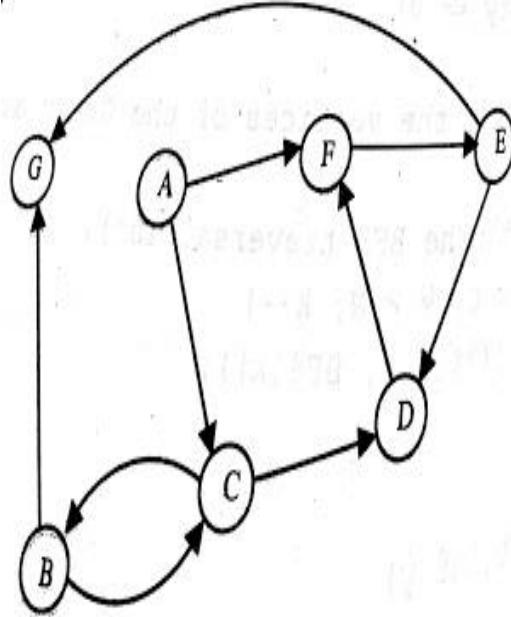
Prof. Shweta Dhawan Chachra

Find BFS in a di-graph

- The rules for a BFS traversal of a graph are: -
- Same as for Undirected Graph
- **Only concept of Directed Edges , Adjacency will differ**

Find BFS in a di-graph

- The table below shows the Queue contents at every insert and remove operations:



<u>Event</u>	<u>Queue (Front to Rear)</u>	<u>BFS</u>
Visit A		A
Visit C	C	AC
Visit F	CF	ACF
Remove C	F	
Visit B	FB	ACFB
Visit D	FBD	ACFB
Remove F	BD	
Visit E	BDE	ACFBDE
Remove B	DE	
Visit G	DEG	ACFBDEG
Remove D	EG	
Remove E	G	
Remove G		
Done		

- The required BFS is A C F B D E G.

Why Use Graphs?

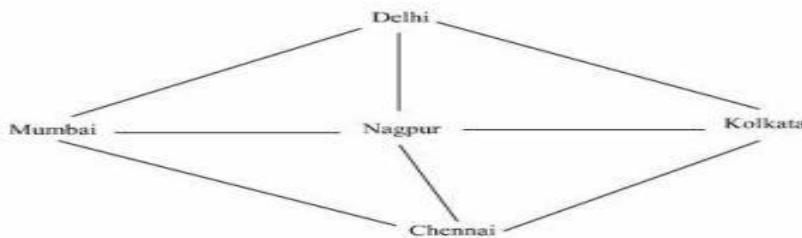
- Graphs serve as models of a wide range of objects:
 - A roadmap
 - A map of airline routes
 - A layout of an adventure game world
 - A schematic of the computers and connections that make up the Internet
 - The links between pages on the Web
 - The relationship between students and courses
 - A diagram of the flow capacities in a communications or transportation network

GRAPHS Representation :

- Graph representation of a **road network**
- A road network is a simple example of a graph,
- Vertices represents **cities** and road connecting them are correspond to edges.

$V = \{ \text{Delhi, Chenai, Kolkata, Mumbai, Nagpur} \}$

$E = \{ (\text{Delhi, Kolkata}), (\text{Delhi, Mumbai}), (\text{Delhi, Nagpur}), (\text{Chenai, Kolkata}), (\text{Chenai, Mumbai}), (\text{Chenai, Nagpur}), (\text{Kolkata, Nagpur}), (\text{Mumbai, Nagpur}) \}$



Prof. Shweta Dhawan Chachra

Applications of Graph Data Structure

- Google maps
- Facebook
- World Wide Web
- Operating System

- ??

Applications of Graph Data Structure

- In Computer science graphs are used to represent the flow of computation.
- Google maps –
 - uses graphs for building **transportation systems**,
 - where **intersection of two(or more) roads are considered to be a vertex** and
 - the **road connecting two vertices** is considered to be an edge,
 - thus their navigation system is based on the algorithm to calculate the **shortest path between two vertices**.

Structure

- In Facebook,
 - **users** are considered to be the **vertices** and
 - if they are **friends** then there is an **edge running between them**.
 - Facebook's **Friend suggestion algorithm uses graph theory**.
 - Facebook is an example of **undirected graph**.

Applications of Graph Data Structure

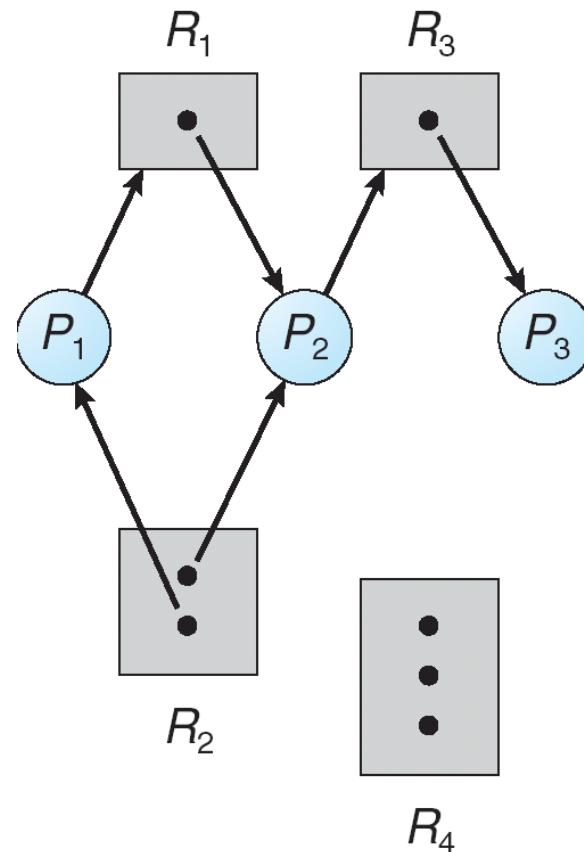
- In World Wide Web,
 - **web pages** = vertices.
 - There is an edge from a page u to other page v if there is **a link of page v on page u** .
 - This is an example of Directed graph.
 - It was the basic idea behind **Google Page Ranking Algorithm**.

Applications of Graph Data Structure

- In Operating System,
- we come across the **Resource Allocation Graph** where
 - **each process and resources are considered to be vertices.**

Applications of Graph Data Structure

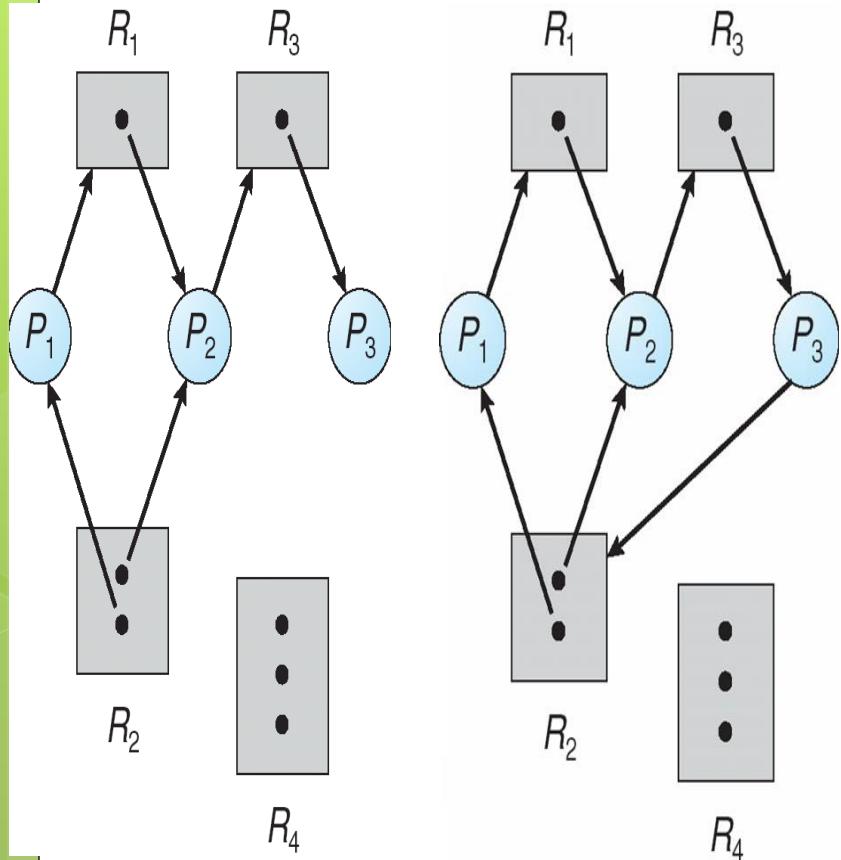
- Edges are drawn from
 - resources to the **allocated** process, or
 - from **requesting** process to the requested resource.
- If this leads to any **formation of a cycle** then a **deadlock may occur**.



Resource Allocation Graph With A Deadlock

100

11/14/2023



- Suppose P_3 requests for R_2 ,
- Since no resource instance is free, a request edge $P_3 \rightarrow R_2$ is added
- So, Now Two cycles exist –
 $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
 $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- P_1, P_2, P_3 are deadlocked