10/22/2024

Graphs

Graph

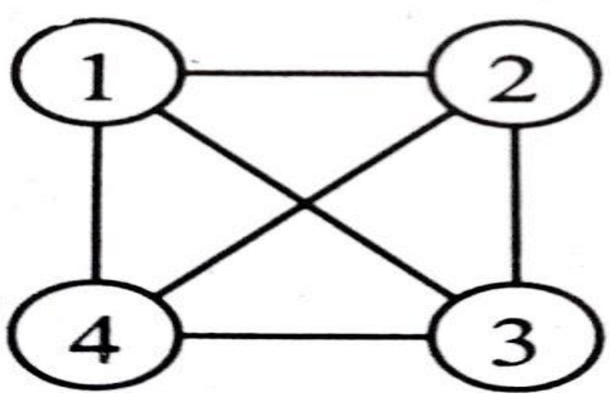
- Collection of 2 sets V &E
- V=Set of Nodes=(v1,v1,v2……vn)
- E=Set of Edges=(e1,e2,e3………en)
- Edge=an arc that connects 2 nodes

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) }***



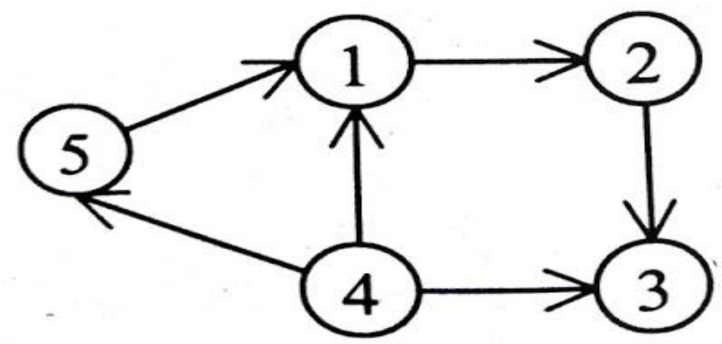
Directed Graph

- *A graph which has **ordered pair of vertices <v1,v2> where v1 is the head and v2 is the tail of the edge.***
- *Each edge has direction, means **<v1,v2> and <v2,v1> 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}

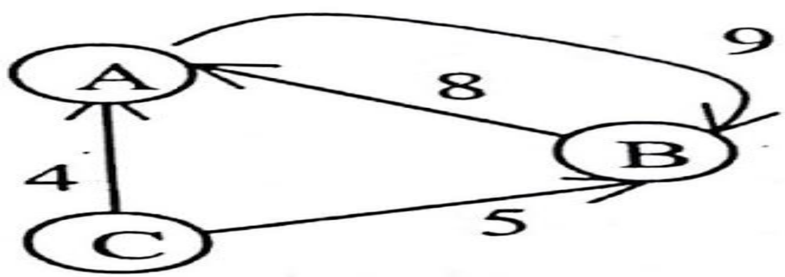
E(G) = { <1,2>, <2,3>, <4,3>, <4,1>, <4,5>, <5,1> }



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 **(v0,v1)** is an edge then v0 is adjacent to v1 and v1 is adjacent to v0.*
- *In a digraph if **<v0,v1>** is an edge then v0 is adjacent to v1 and v1 is adjacent from v0*

Graph Terminology

Incidence :-

- *In an undirected graph the **edge (v0,v1) is incident on nodes v0 and v1**.*
- *In a digraph the **edge <v0,v1> is incident from node v0 and is incident to node v1.***

Path :-

- *A path from node u0 to node un is a sequence or nodes $u_0, u_1, u_2, u_3 \ldots\ldots u_{n-1}, u_n$ such that **u_0 is adjacent to u_1, u_1 is adjacent to $u_2, \ldots\ldots\ldots, u_{n-1}$ is adjacent to u_n .***

Length of path :-

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

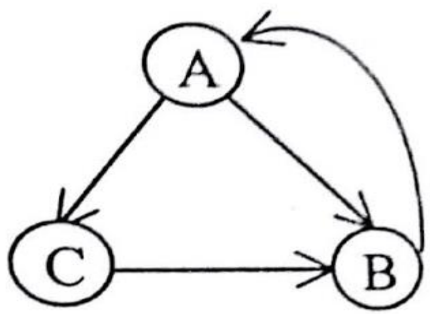
Graph Terminology

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

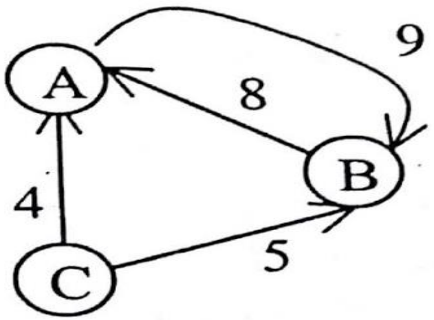
- ***Simple path :-***
- *Simple path is a path in which* ***all the nodes are distinct with an exception that the first and last nodes of the path can be same.***

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.*



G7



G9

Graph Terminology

Cyclic graph :-

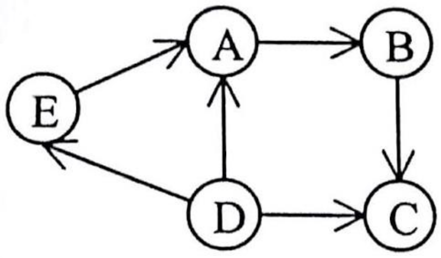
- ***A graph that has cycles*** *is called a cyclic graph.*

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.*



G5

Graph Terminology

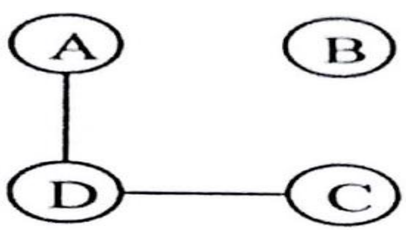
Degree :-

- *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

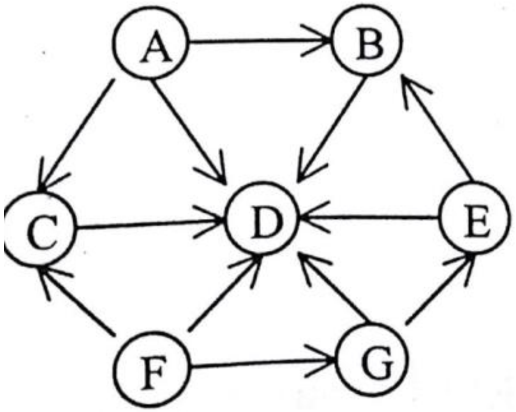
Graph Terminology

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.*



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 nde2 and so on.*
- *The entry of this matrix will be as-*

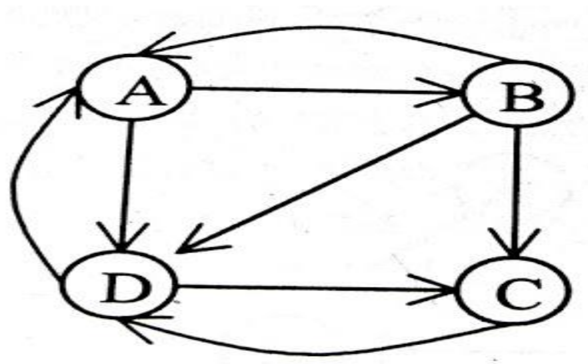
Arr[i][j] = 1 If there is an edge from node i to node j

= 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 :-

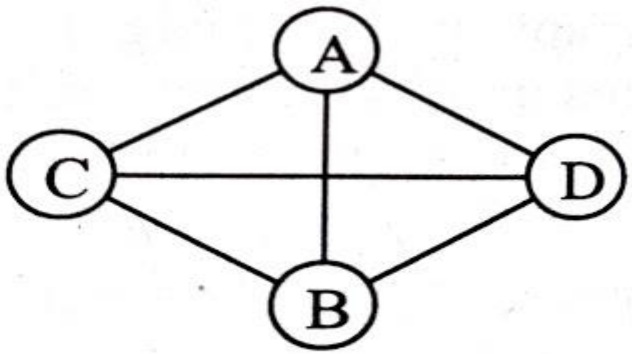


$$\text{Adjacency Matrix A} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

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

Representation of Graph

Adjacency Matrix :-



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

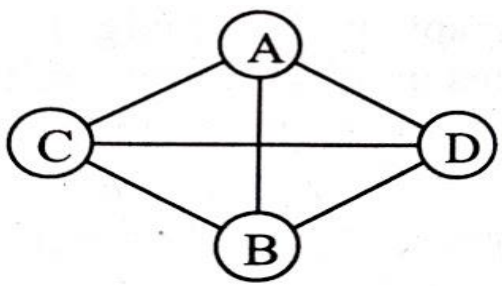
arr[0][1] = 1, which represents there is an edge in the graph from node A to node B.

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



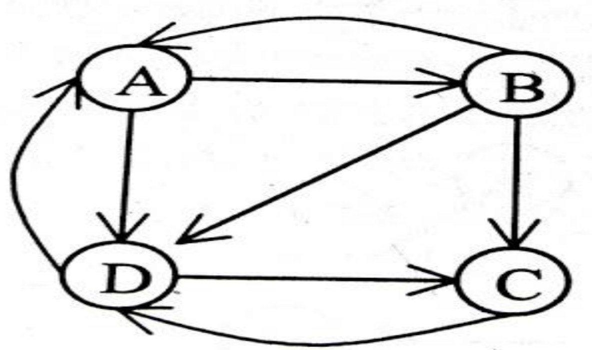
$$\text{Adjacency Matrix } A = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Representation of Graph

Adjacency Matrix :-

In directed graph,

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



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

Representation of Graph

Adjacency Matrix :-

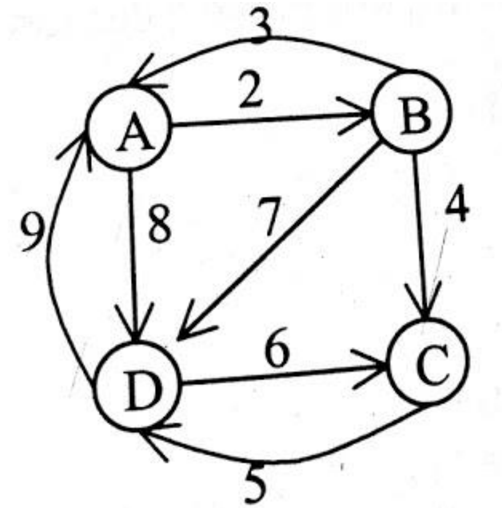
Weighted Graph

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

= 0 Otherwise

Representation of Graph

Adjacency Matrix :-



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

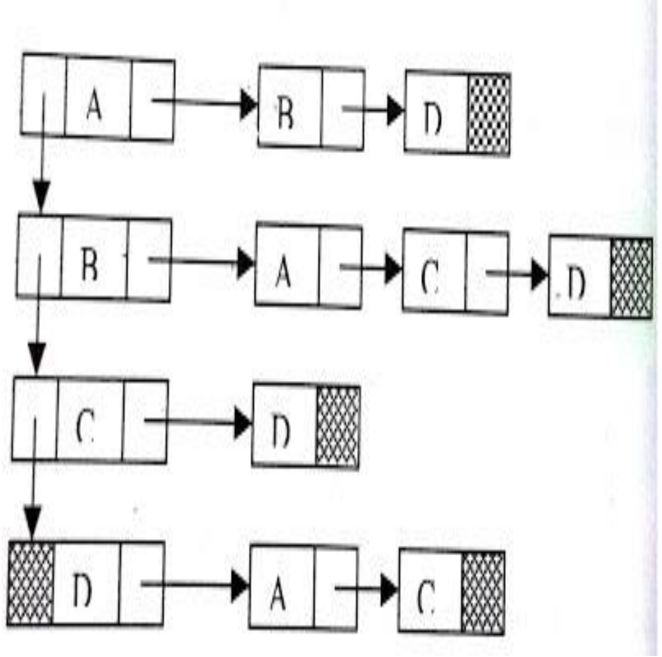
arr[0][1] = weight, which represents there is an weighted edge in the graph from node A to node B.

Adjacency List :-

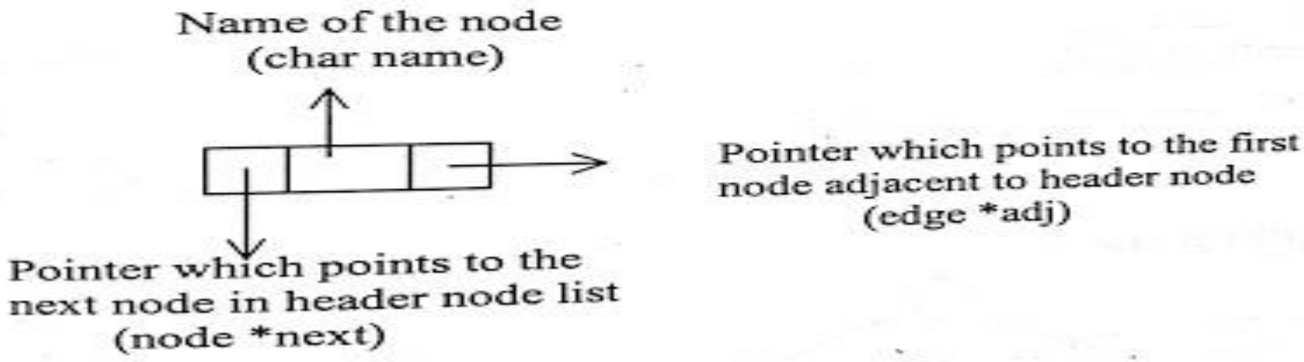
- *If the adjacency matrix of the graph is sparse then*
  - *it is more efficient to represent the graph through adjacency list.*
- *We will maintain* ***two lists.***
- *First list will keep* ***track of all the nodes*** *in the graph*
- *Second list will maintain* ***a list of adjacent nodes for each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



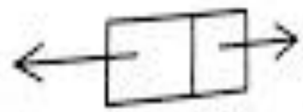
Structure of header node :




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

Structure of edge :

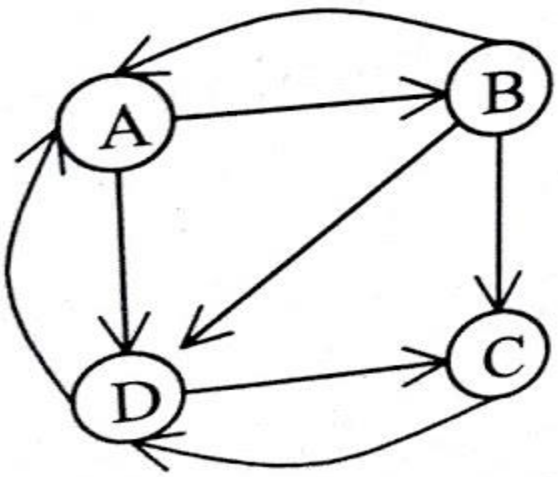
Name of the destination node of the edge (char dest)

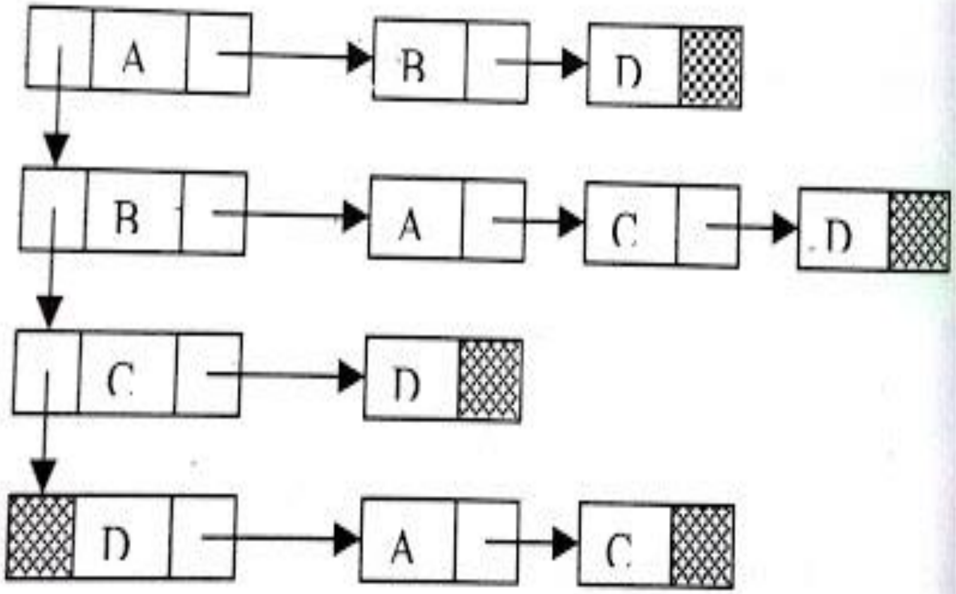


Pointer which points to the next adjacent node of header node (edge *link)

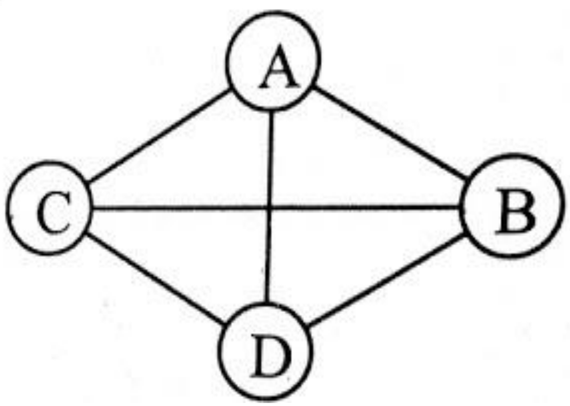
```
struct edge {
        char dest;
        struct edge *link;
    };
```

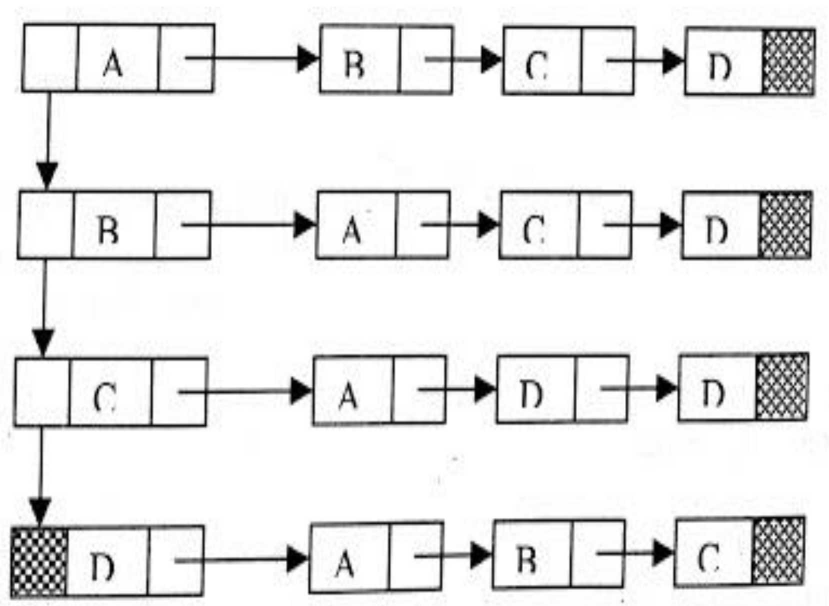
Adjacency list:



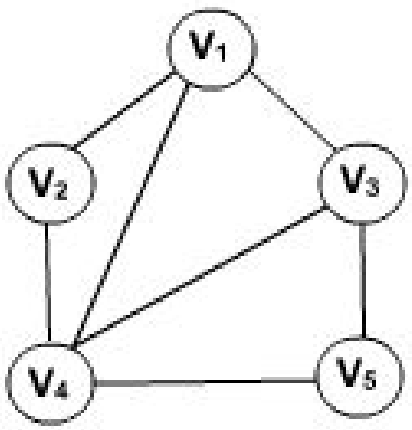


Adjacency list:



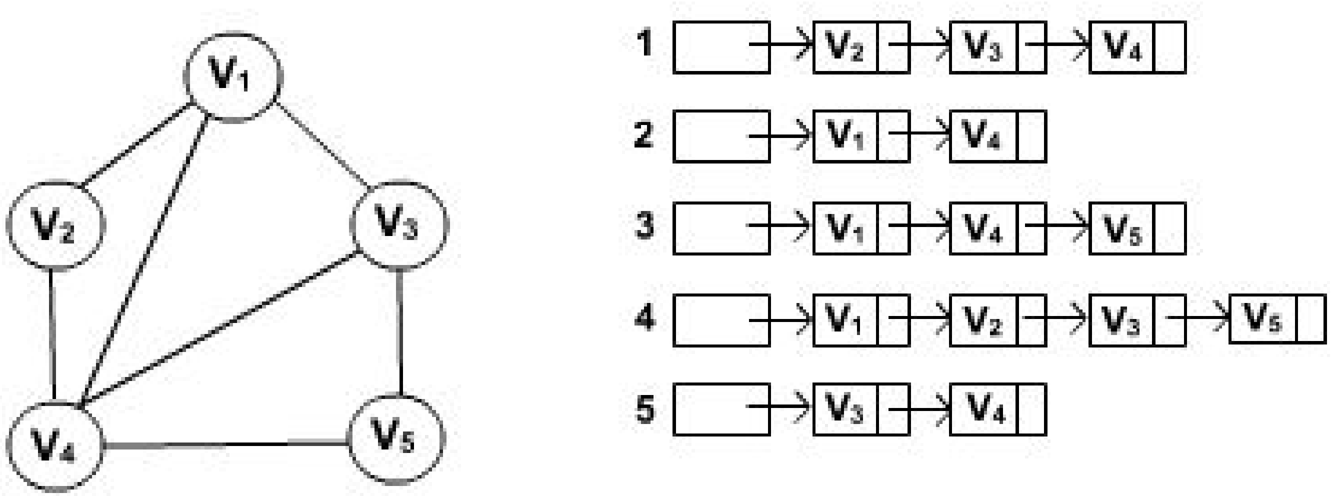


Adjacency Matrix Representation:



	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-*

$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & 0 & 1 & 0 & 1 \\
B & 1 & 0 & 1 & 1 \\
C & 0 & 0 & 0 & 1 \\
D & 1 & 0 & 1 & 0
\end{array}
$$

- *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.***

$$
\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 0 & 1 & 0 \\
B & 1 & 0 & 1 & 1 & 0 \\
C & 0 & 0 & 0 & 1 & 0 \\
D & 1 & 0 & 1 & 0 & 0 \\
E & 0 & 0 & 0 & 0 & 0
\end{array}
$$

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

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

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

- *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-*

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

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-*

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

- *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-*

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

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-*

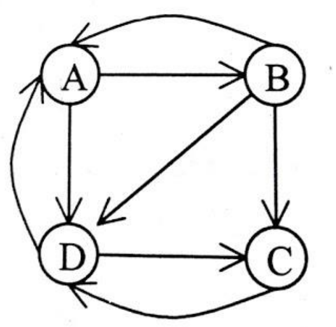
$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

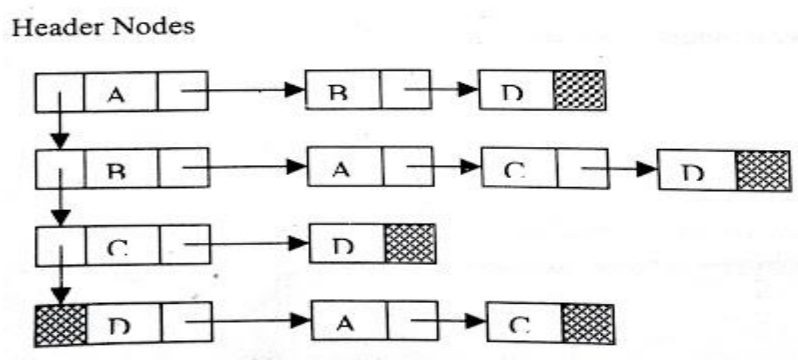
- *To **delete the edge which is in between B and C,***
- *Change the **entry 1 to 0 at the position 2nd row, 3rd column**.*
- *Adjacency matrix will be-*

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

Insertion in Adjacency list

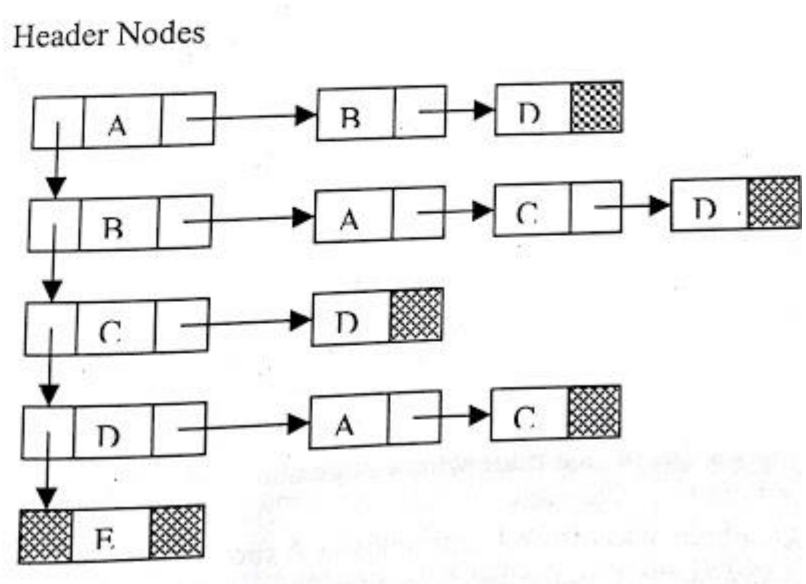
- ***Node insertion :-***
- *Insertion of node in adjacency list requires only* ***insertion of that node in header nodes*** *of adjacency list.*
- *Let us take a graph G-*
- ***The adjacency list for this graph will be as-***





Insertion in Adjacency list

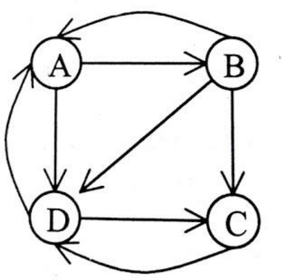
- *Suppose we want to* ***insert one node E***
- ***Addition of node E in header node only.***
- *Now the adjacency list will be-*

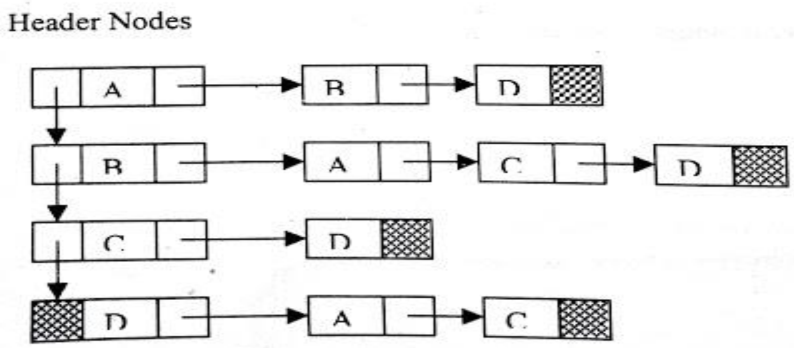


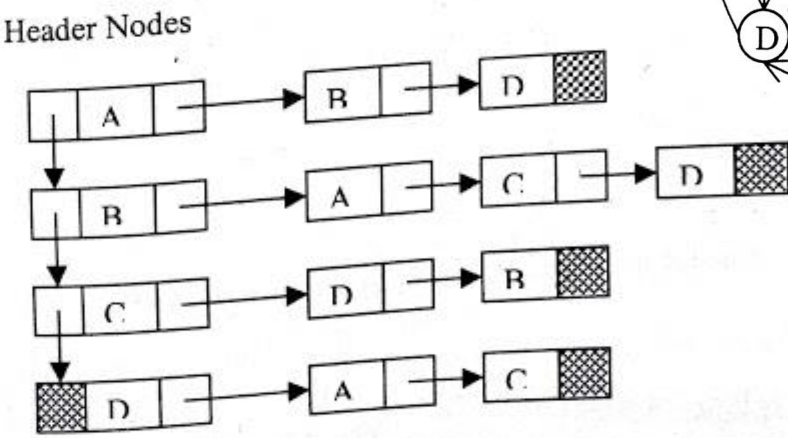
Edge insertion in Adjacency list

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.*





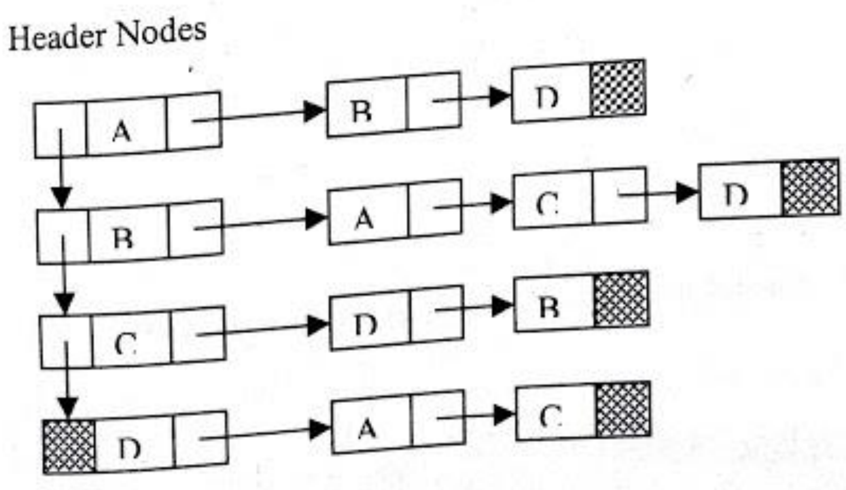


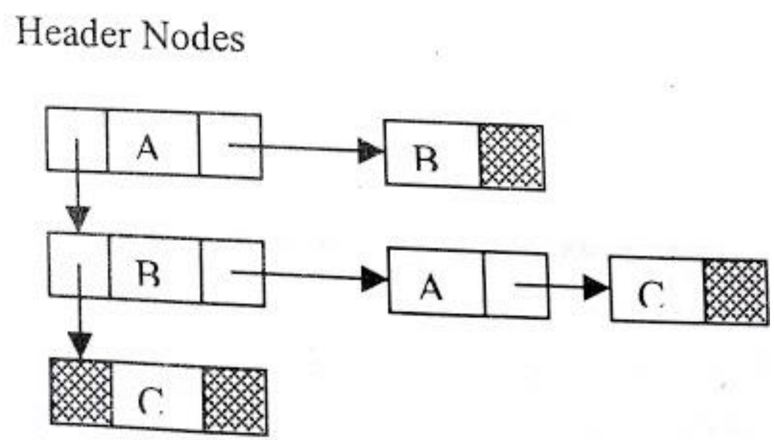
Deletion in Adjacency list :

- ***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 :

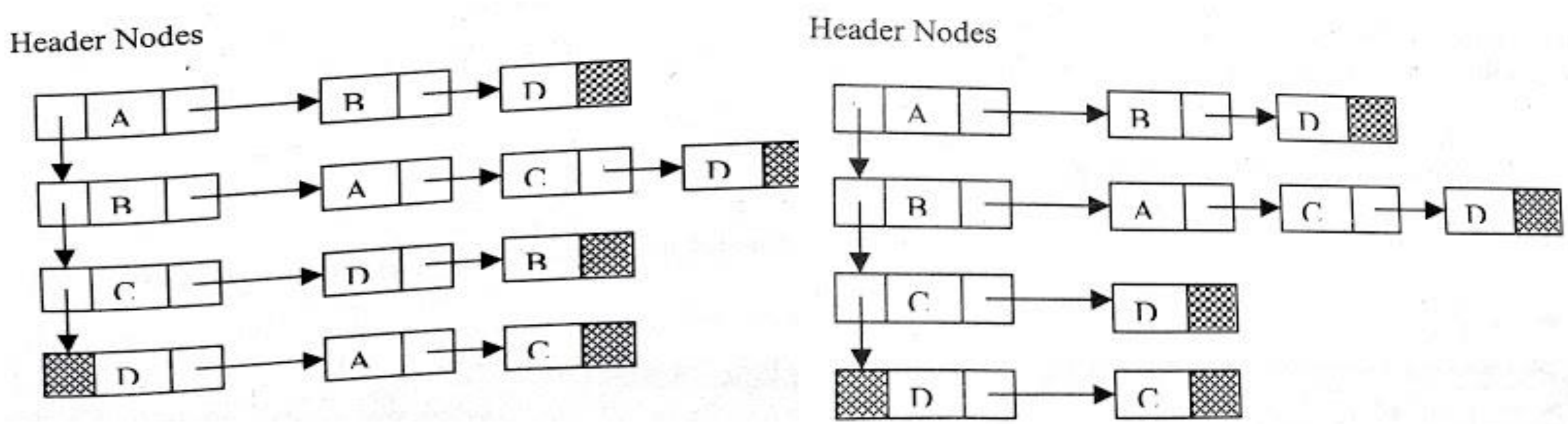
Edge deletion :-

- *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.*



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.**

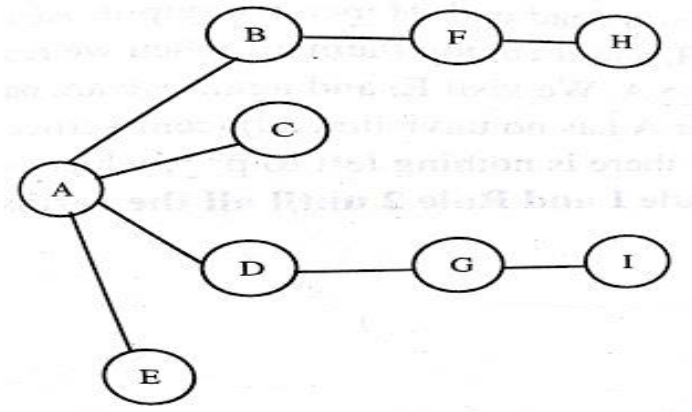
- **The distance from start vertex is called depth**

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 :***

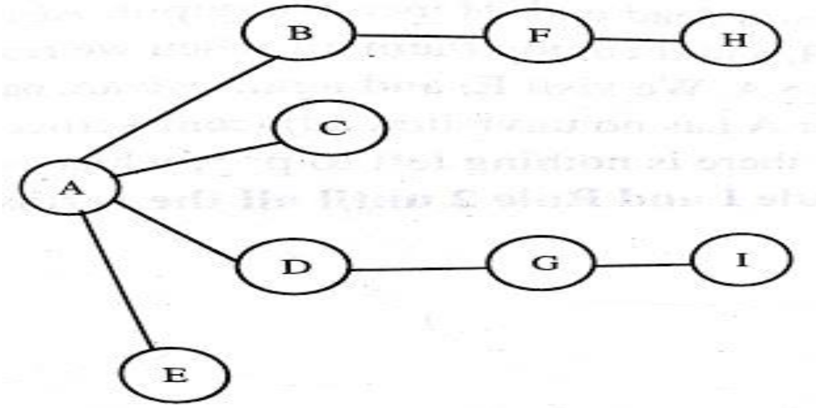
Let us consider the following graph-



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

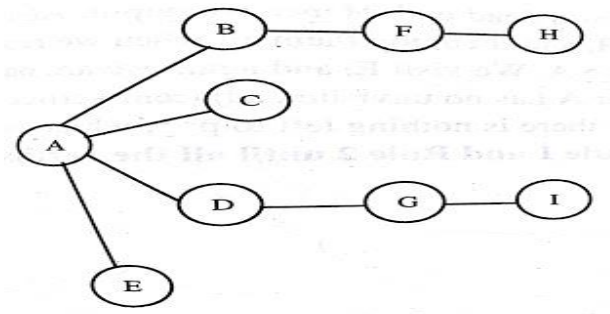
Depth First Search

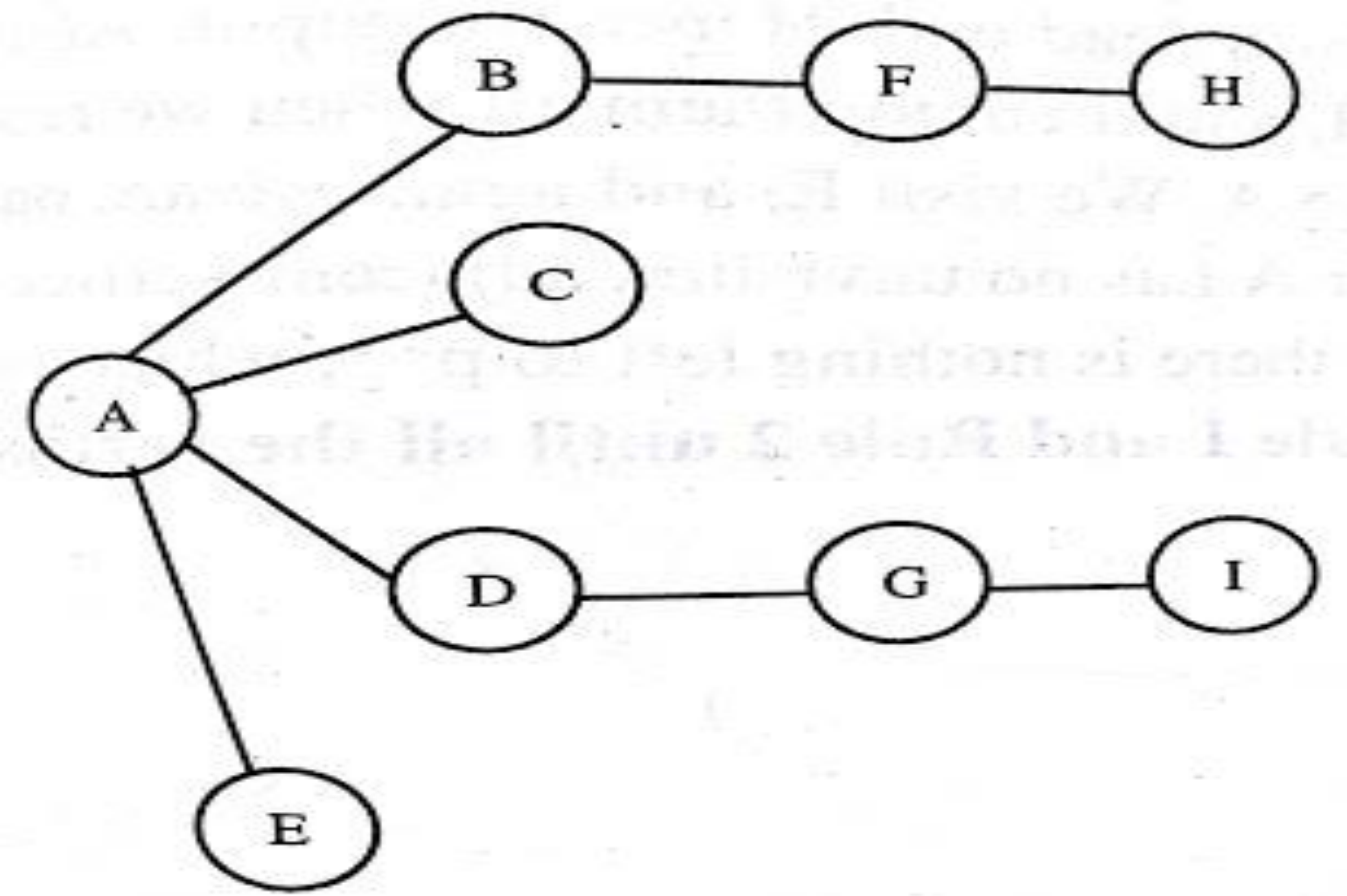
- To carry out the depth-first search, we pick a starting point, vertex A.



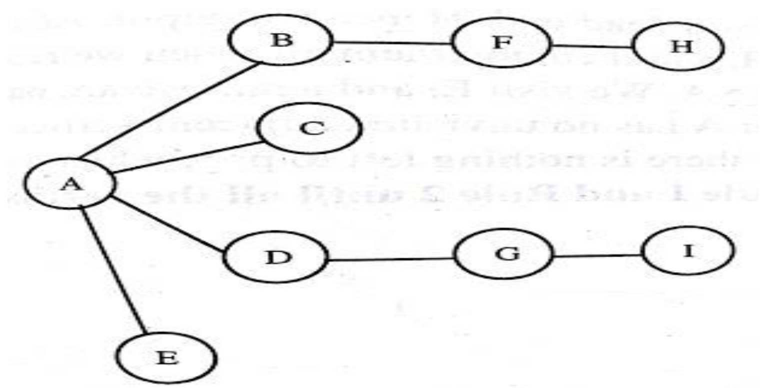
- We can do three things:
- Visit the vertex
- Push it onto stack so we can remember it, and
- Mark it, so it is not visited again.

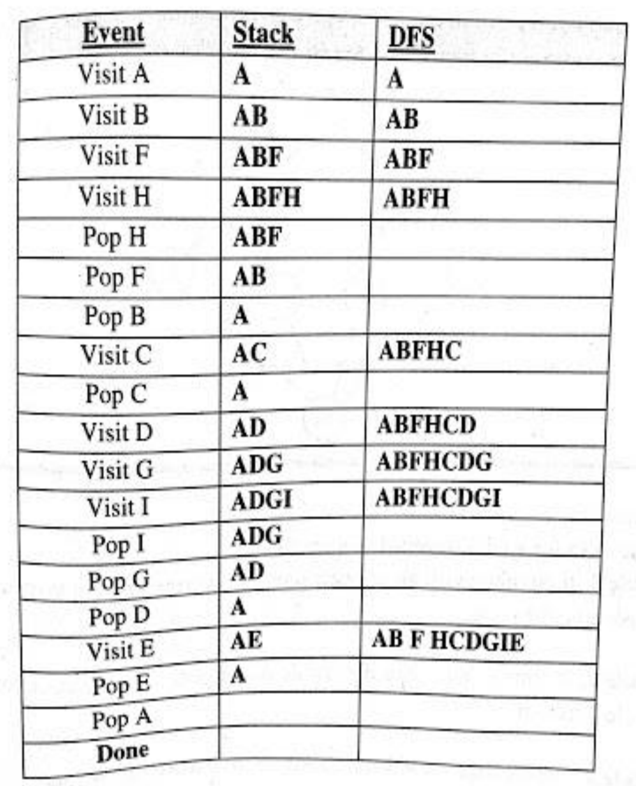
Depth First Search




A
B
C
D
E
F
G
H
I

Depth First Search





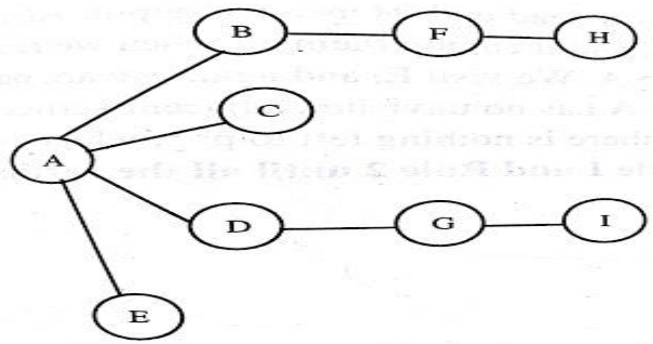
Event	Stack	DFS
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		

- *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*

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

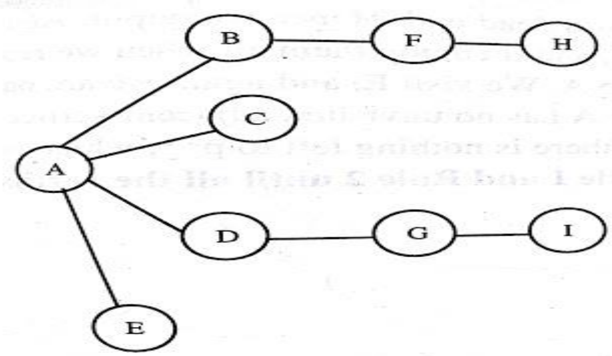
Rule 2

- *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 adjacen**t 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.***

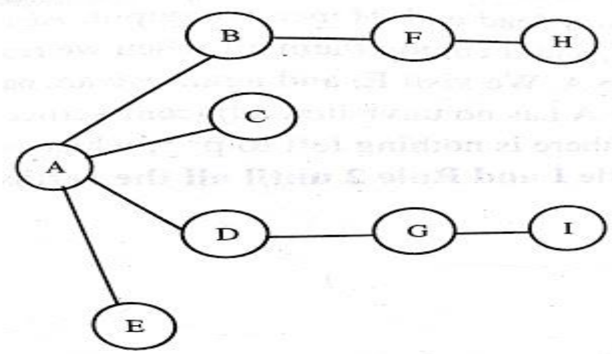
Event	Stack	DFS
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		



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		

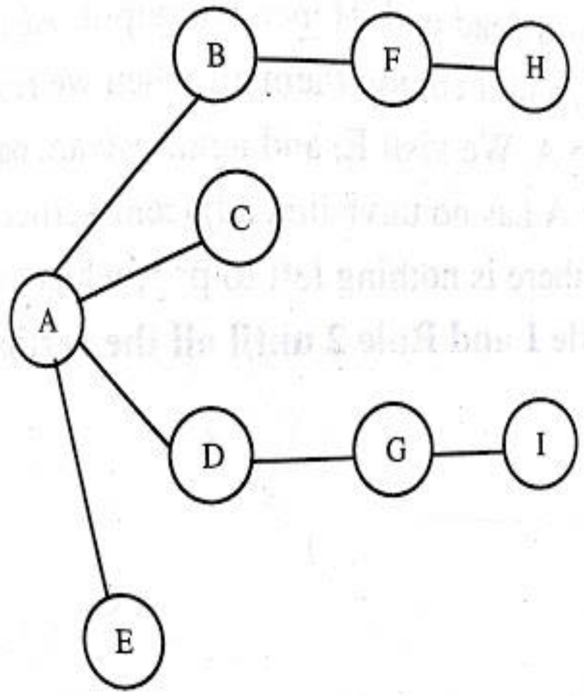


Depth First Search

- ***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:***



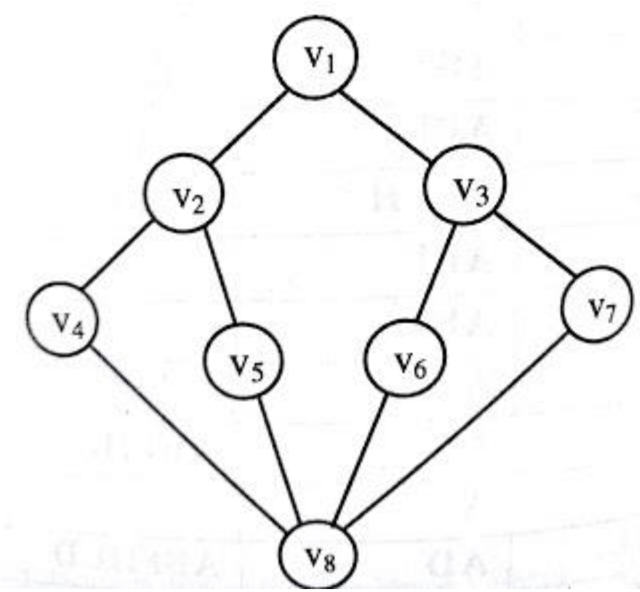
Event	Stack	DFS
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		

Required DFS is ABFHCDGIE

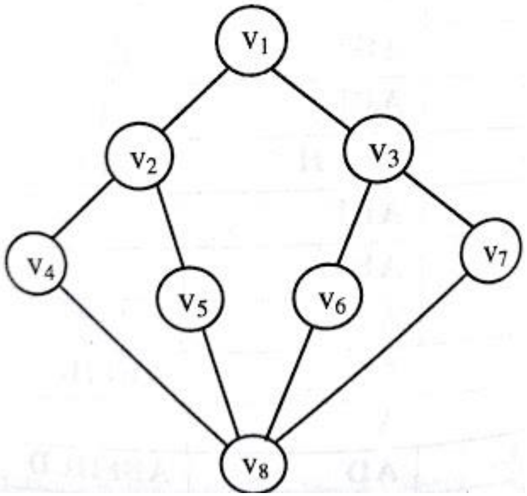
DFS-Procedure using Stack-

- 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



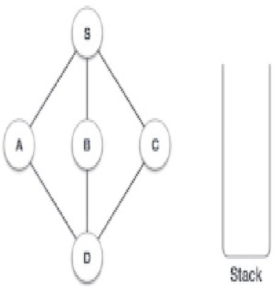
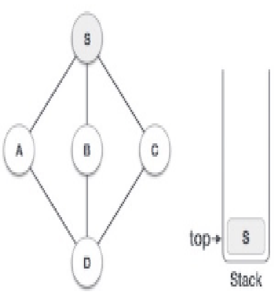
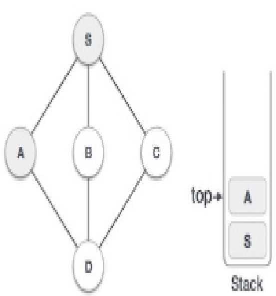
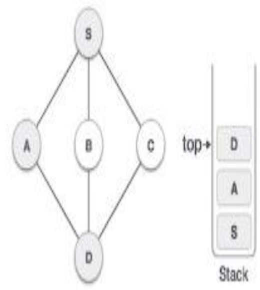
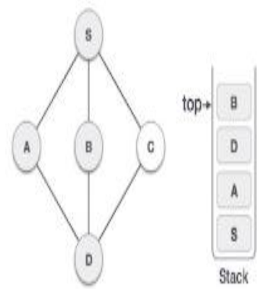
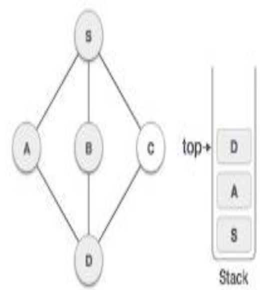
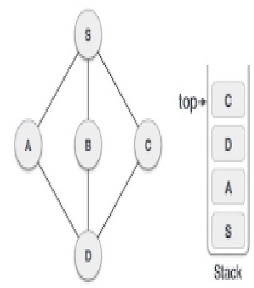
Perform DFS



Event	Stack	DFS
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

Depth-first search using Backtracking

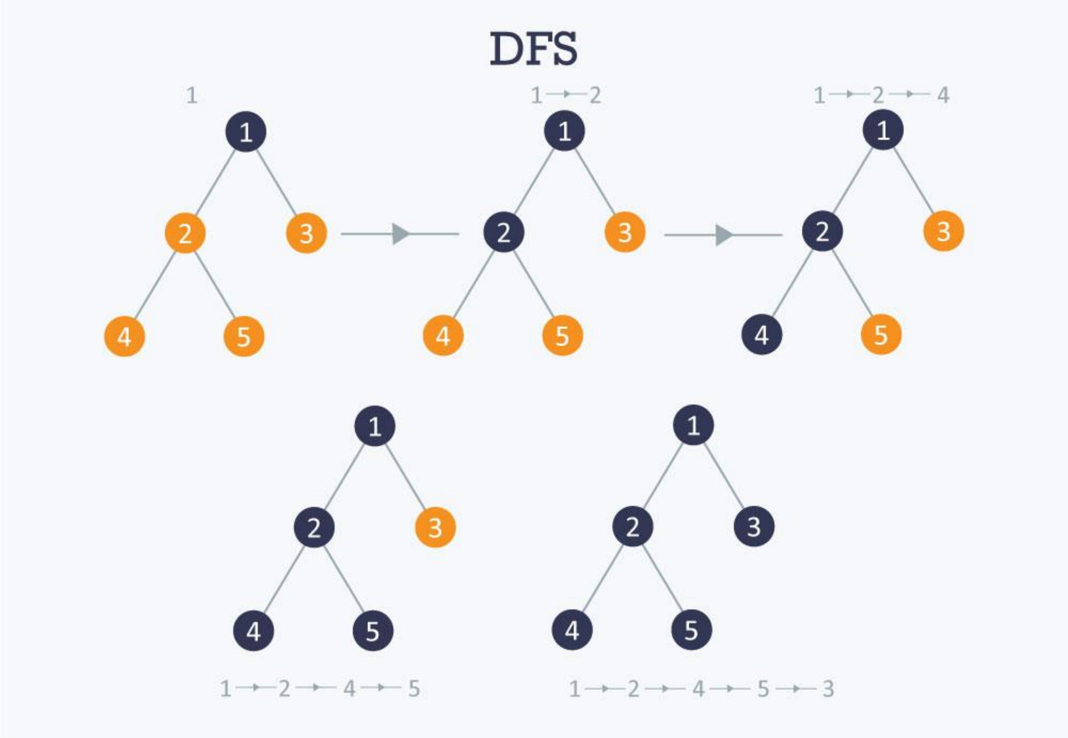
Step	Traversal	Description
1		Initialize the stack.
2		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark A as visited and put it onto the 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.

As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Courtesy: https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

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.**

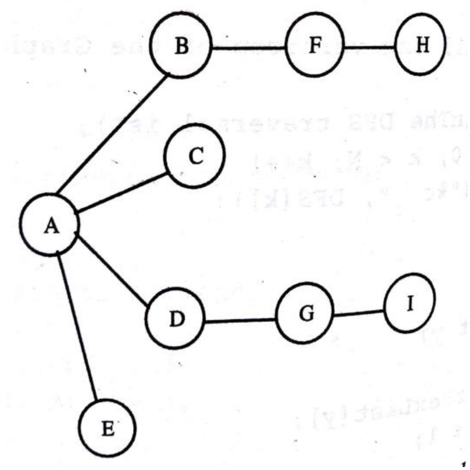
- It moves **outward in all directions, one level at a time.**

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.*

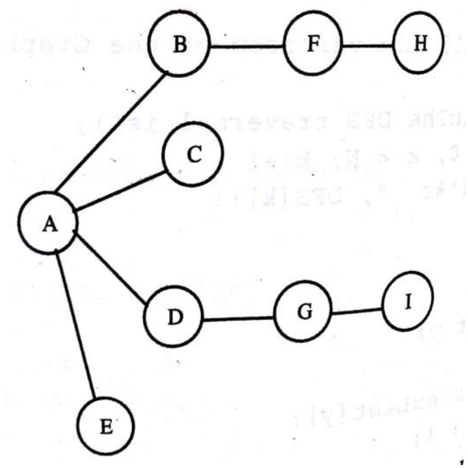
Breadth First Search

- *Let us consider the following graph-*

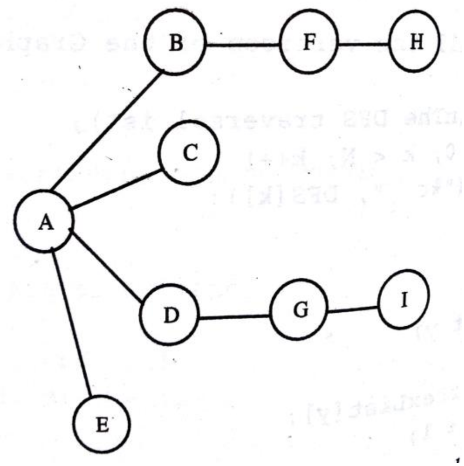


- *In BFS, all the vertices adjacent to start vertex are visited, In first pass*
- *This kind of search is implemented **using a Queue***

Breadth First Search

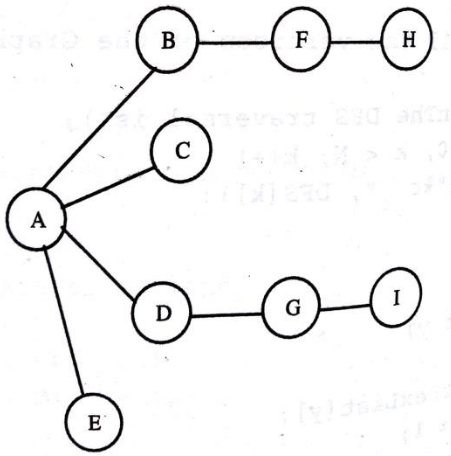


Breadth First Search



Breadth First Search

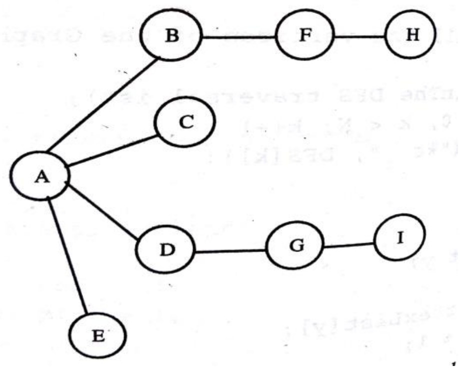
- *Let us consider the following graph-*



- 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	ABCDEFGH
Remove G	H	
Visit I	HI	ABCDEFGHI
Remove H	I	
Remove I		
Done		

Breadth First Search

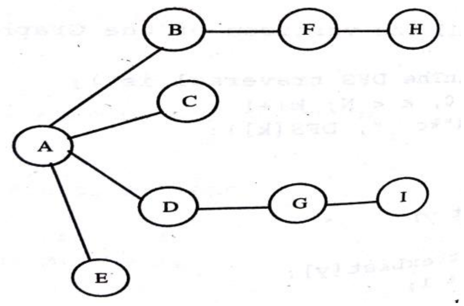


- *Visit all the vertices adjacent to A, inserting each one into the queue during each visit.*
- *We visit A, B, C, D and E.*
- *At this point the queue (from front to rear) contains **B, C D and E.***
- *There are **no more unvisited vertices adjacent to A**,*

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	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.

Breadth First Search

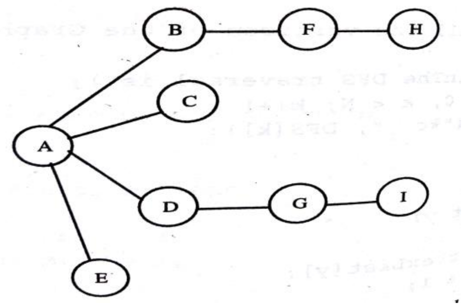


- *So we* ***remove B*** *from the queue and* ***look for vertices adjacent to it***
- *We find* ***F, so we insert*** *it in the queue.*
- *There are* ***no more unvisited vertices adjacent to B,***

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	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.

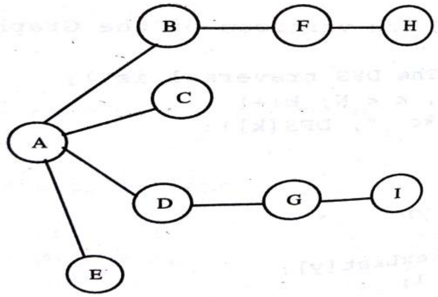
Breadth First Search



- **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	ABCDEFGH
Remove G	H	
Visit I	HI	ABCDEFGHI
Remove H	I	
Remove I		
Done		

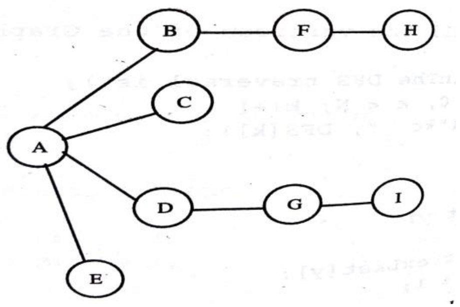
Breadth First Search



- 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***

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	ABCDEFGH
Remove G	H	
Visit I	HI	ABCDEFGHI
Remove H	I	
Remove I		
Done		

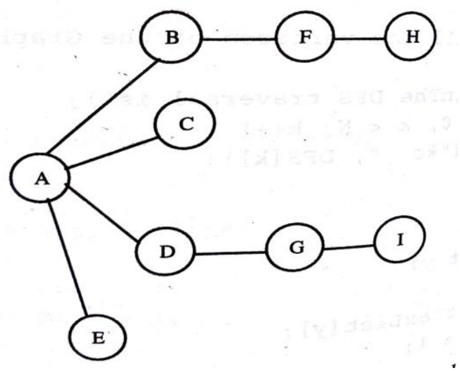
Breadth First Search



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	ABCDEFGH
Remove G	H	
Visit I	HI	ABCDEFGHI
Remove H	I	
Remove I		
Done		

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

Breadth First Search



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

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	ABCDEFGH
Remove G	H	
Visit I	HI	ABCDEFGHI
Remove H	I	
Remove I		
Done		

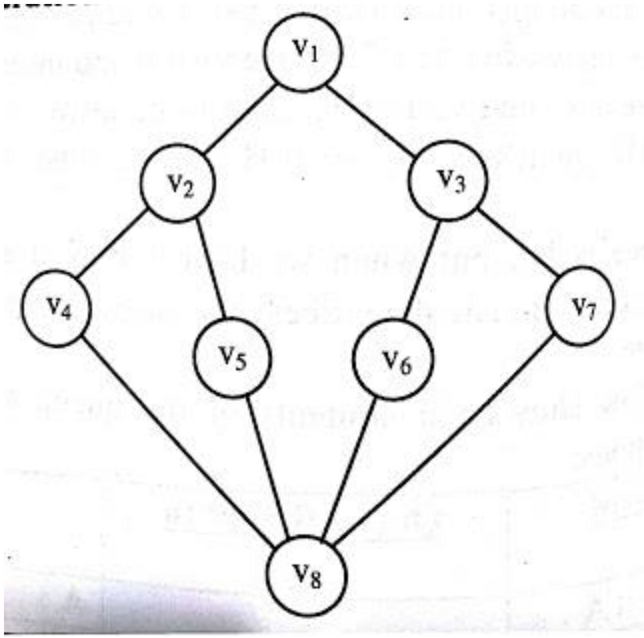
BFS-Procedure-using Queue

- Use array implementation of queue to keep the unvisited neighbors of the node
  - Initially **Queue is empty**, **front=-1,rear=-1**
- 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

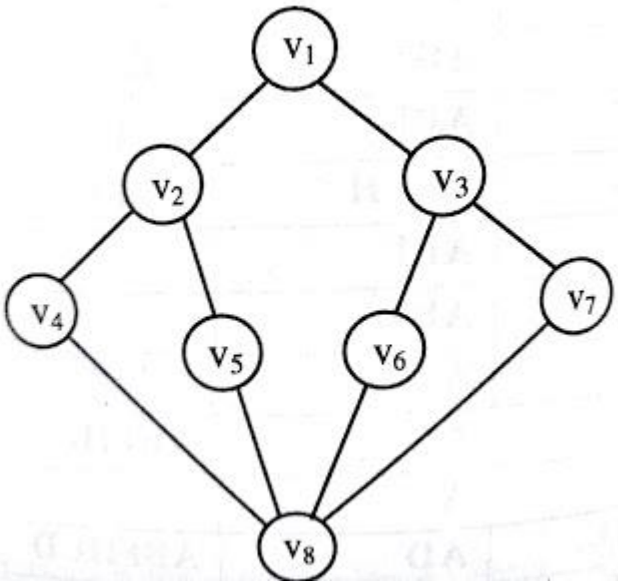
BFS- Procedure using Queue

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



Event	Queue (Front to Rear)	BFS
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		

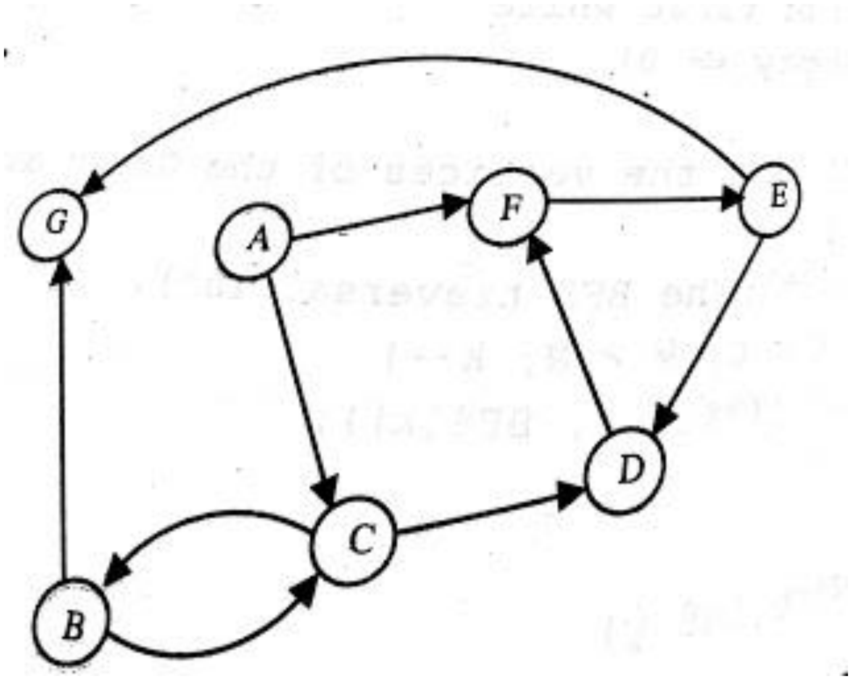
The required BFS is V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8.

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***

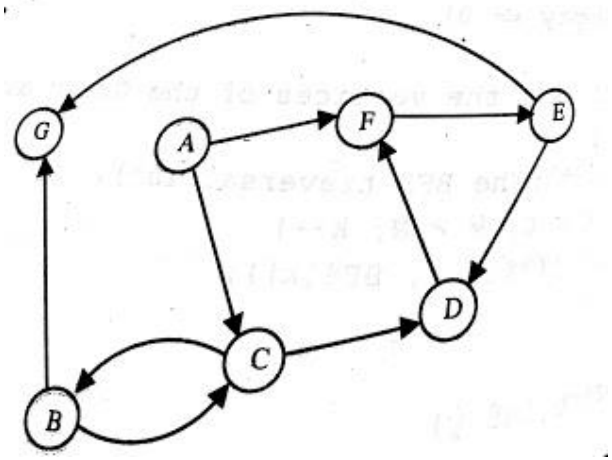
Find DFS & BFS in a di-graph

- *Let us consider the following di-graph-*



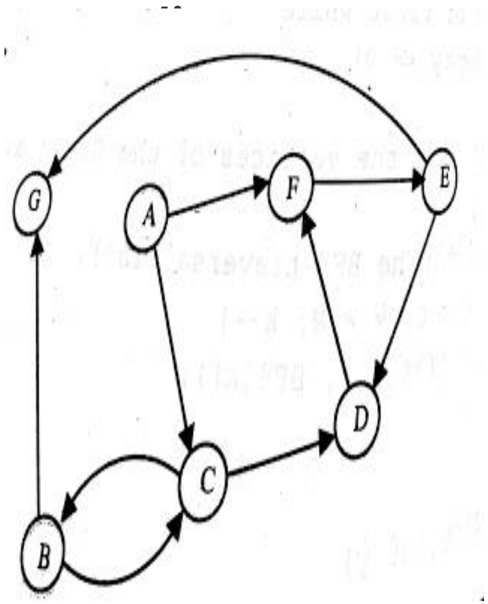
Covering nodes in Alphabetical Order

Find DFS & BFS in a di-graph



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	

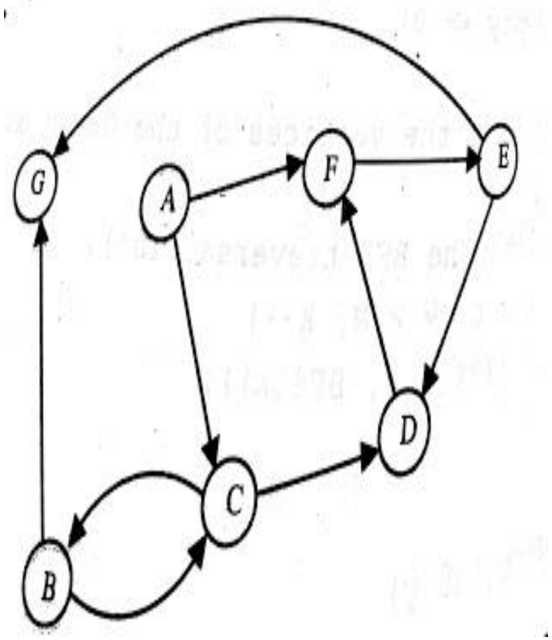
- *The required DFS is A C B G D F E.*

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:*



Event	Queue (Front to Rear)	BFS
Visit A		A
Visit C	C	AC
Visit F	CF	ACF
Remove C	F	
Visit B	FB	ACFB
Visit D	FBD	ACFBD
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?

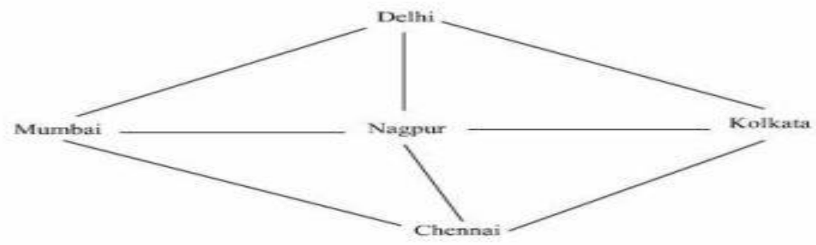
- Graphs serve as models of a wide range of objects:
  - A roadmap
  - A map of airline routes
  - A layout of an adventure game world
  - A schematic of the computers and connections that make up the Internet
  - The links between pages on the Web
  - The relationship between students and courses
  - 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= { Delhi, Chenai, Kolkata, Mumbai, Nagpur }

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



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.**

Applications of Graph Data Structure

- In Facebook,
  - **users** are considered to be the **vertices** and
  - if they are **friend**s 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

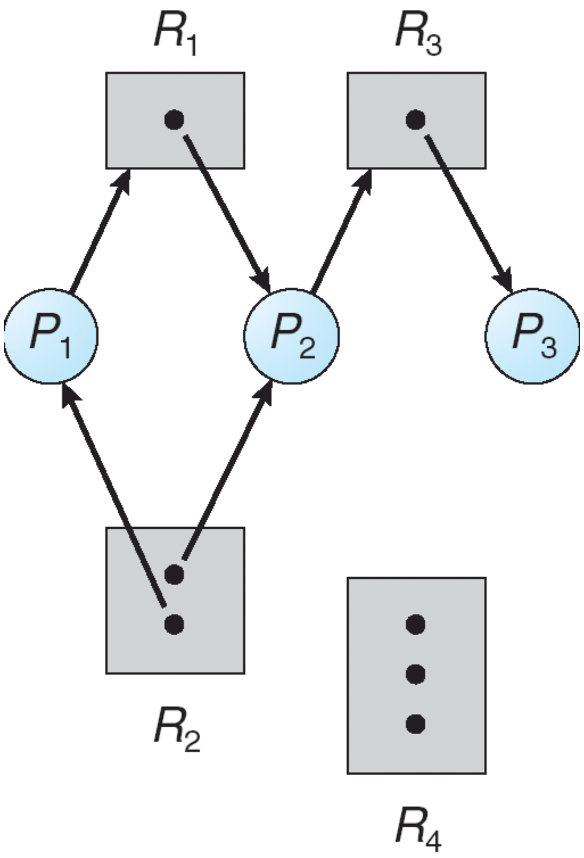
- In World Wide Web,
  - **web pages =** vertices.
  - There is an edge from a page u to other page v if there is **a link of page v on page u**.
  - This is an example of Directed graph.
  - It was the basic idea behind **Google Page Ranking Algorithm.**

Applications of Graph Data Structure

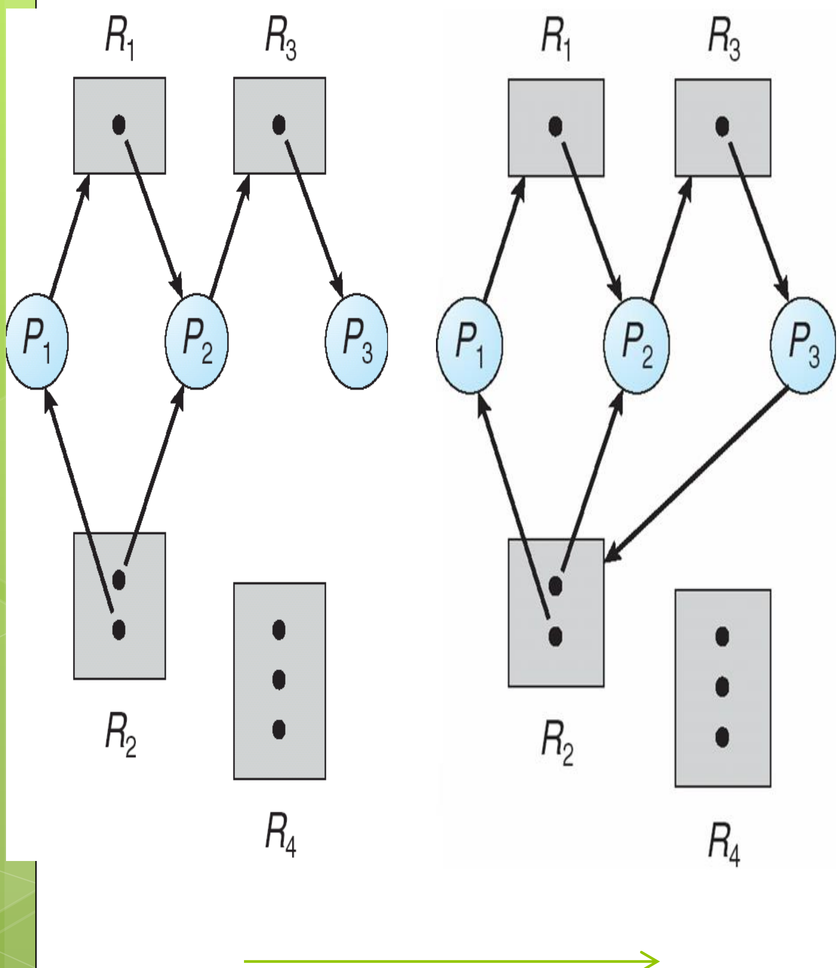
- In Operating System,
  - we come across the **Resource Allocation Graph** where
    - **each process and resources are considered to be vertices.**

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



- Suppose P3 requests for R2,
- Since no resource instance is free, a request edge P3->R2 is added
- So, Now Two cycles exist –

P1->R1->P2->R3->P3->R2->P1

P2->R3->P3->R2->P2

- P1,P2,P3 are deadlocked

extra

DFS- Procedure using Stack

1) Push **starting node into the stack**
2) **Pop an element** from the stack, if it has not been traversed then traverse it, If it has been already traversed then just ignore it. After traversing, make the value of visited array for this node as true.
3) Now **push all the unvisited adjacent nodes of the popped element on stack** . Push the element even if it already on the stack
4) Repeat steps 3 and 4 until stack is empty