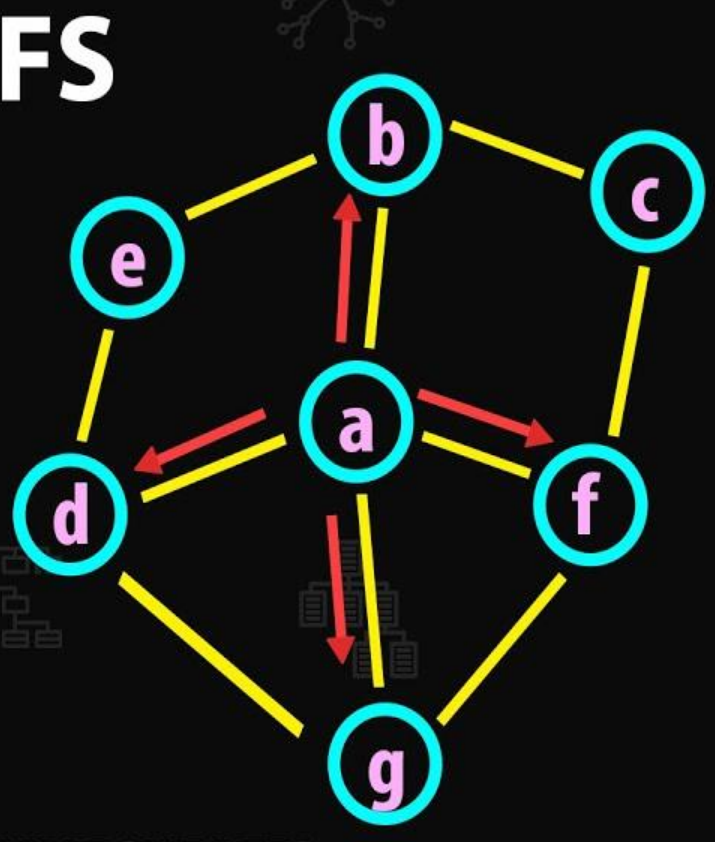
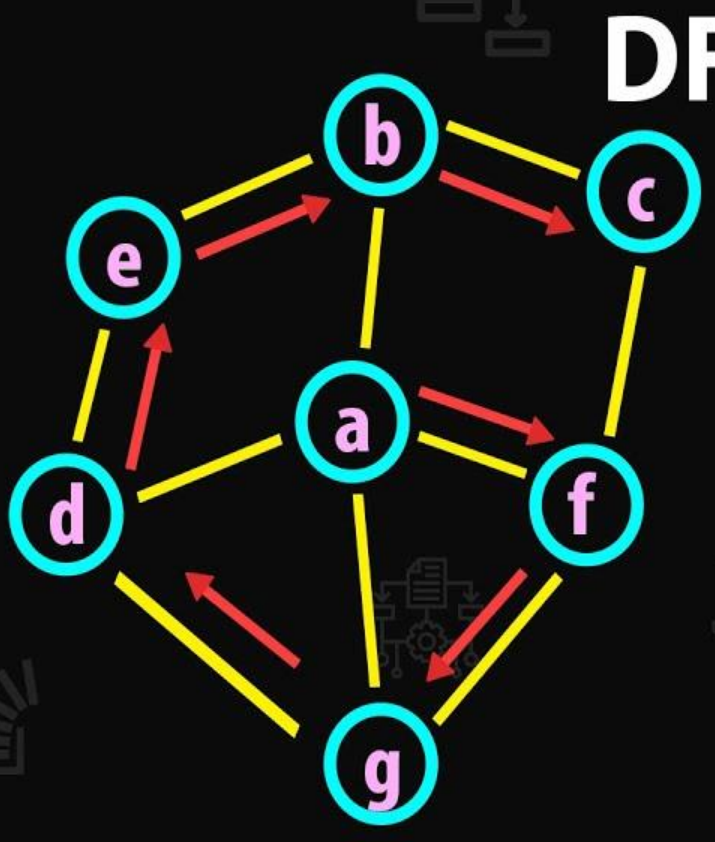
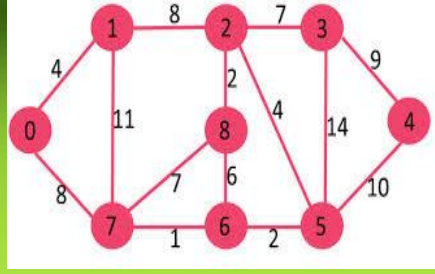
GRAPH TRAVERSAL - BFS & DFS





SIMPLE SNIPPETS

GRAPHS



A graph is a unique data structure in programming that consists of finite sets of nodes or vertices and a set of edges that connect these vertices to them. At this moment, adjacent vertices can be called those vertices that are connected to the same edge with each other.

Types of Graph

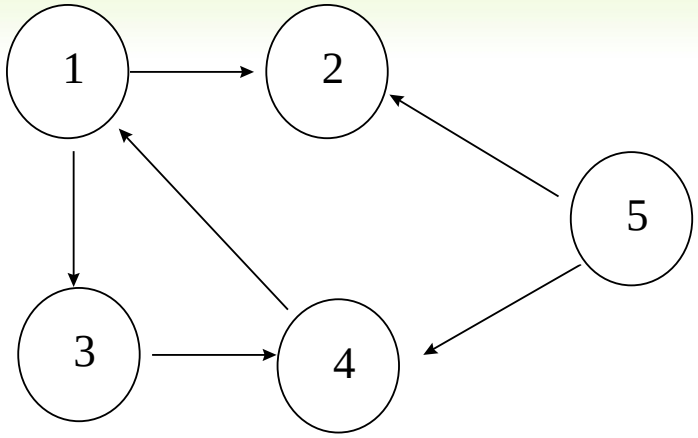
- connected and disconnected graphs,
- bipartite graphs,
- weighted graphs,
- directed and undirected graphs,
- simple graphs.

IMPLEMENTING A GRAPH

- Implement a graph in two ways:
  - Adjacency List
  - Adjacency-Matrix

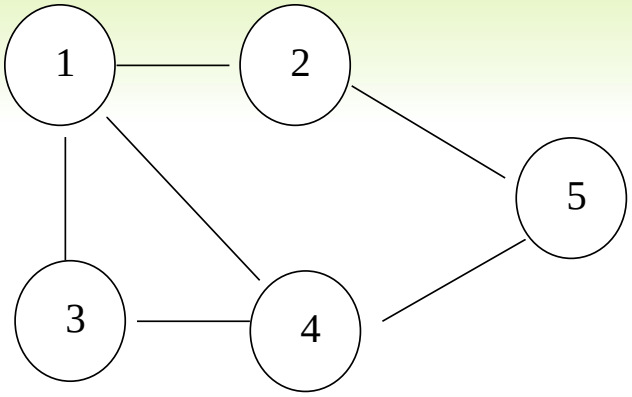
ADJACENCY LIST

- List of pointers for each vertex



1	→	2	→	3
2	→			
3	→	4		
4	→	1		
5	→	2	→	4

UNDIRECTED ADJACENCY LIST



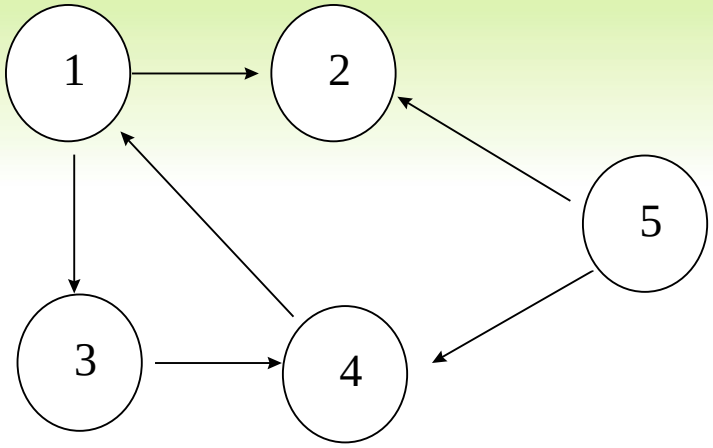
1	→	2	→	3	→	4
2	→	1	→	5		
3	→	4	→	1		
4	→	1	→	3		
5	→	2	→	4		

ADJACENCY LIST

- The sum of the lengths of the adjacency lists is 2|E| in an undirected graph, and |E| in a directed graph.

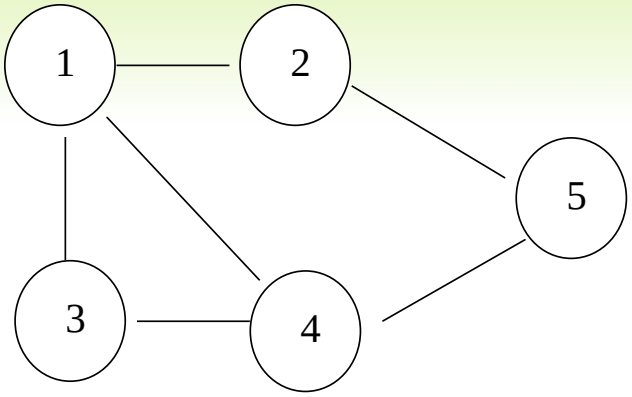
- The amount of memory to store the array for the adjacency list is O(V+E).

ADJACENCY MATRIX



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

UNDIRECTED ADJACENCY MATRIX



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

ADJACENCY MATRIX

- The matrix always uses $\Theta(v^2)$ memory. Usually easier to implement and perform lookup than an adjacency list.

SEARCHING A GRAPH

- **Search:** The goal is to methodically explore every vertex and every edge; perhaps to do some processing on each.
- For the most part in our algorithms we will assume an adjacency-list representation of the input graph.
- Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal.

- Transportation networks.
- Communication networks.
- Information networks.
- Social networks.
- Dependency networks.

One of the fundamental operations in a graph is that of traversing a sequence of nodes connected by edges.

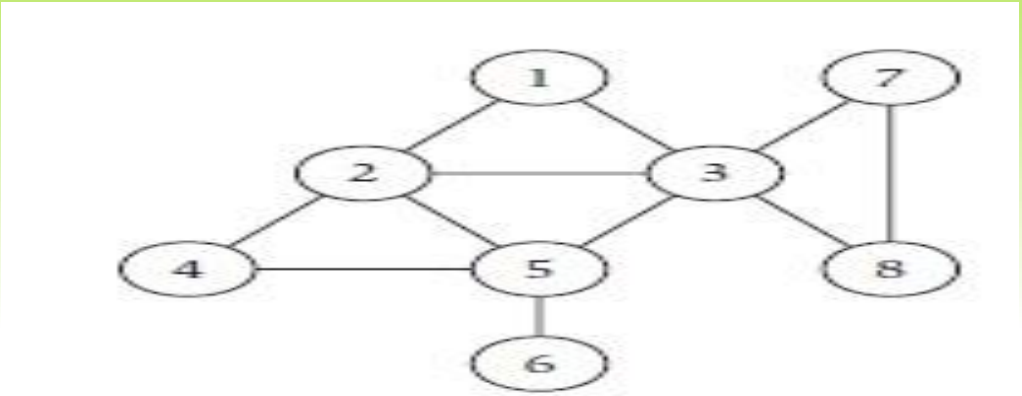
Graph Connectivity and Graph Traversal

- . Suppose we are given a graph G = (V, E) and two particular nodes s and t. We'd like to find an efficient algorithm that answers the question: Is there a path from s to t in G?. We will call this the problem of determining s-t connectivity.{maze solving Problem}

BREADTH FIRST SEARCH

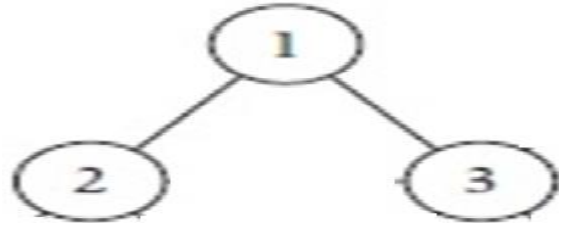
- BFS is an algorithm for traversing a graph in search of an element.
- is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. BFS is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.
- For each node, search its immediate children. If found we are done, else for each of the children search their children and so on.
- BFS can be used to check connectivity of a graph (i.e where the graph has disjointed sub-graphs or isolated vertices).

- In this algo, given a graph G= (V, E) and a source vertex S, we explore from S in all possible directions, adding nodes one “layer” at a time.
- **Example 1**



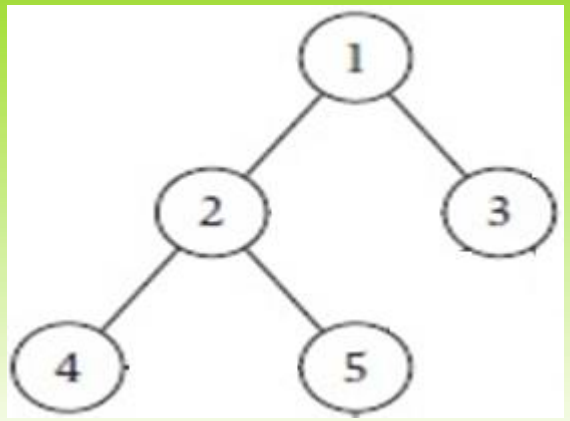
Consider the source vertex S=1. Add to BFS tree T @ Layer 0.

- We discover nodes {2,3} from node 1 check if {2,3} are in T



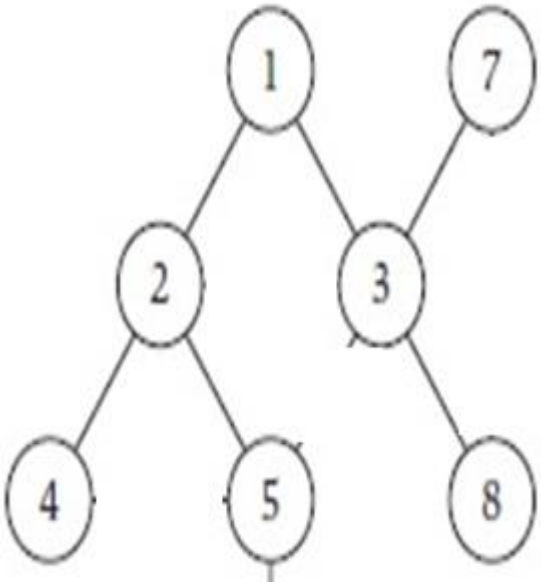
Layer L1

- We now discover nodes {3,4,5} from node 2 in L1
  - 3 is already present in T
  - Add 4,5 to T



Layer 2

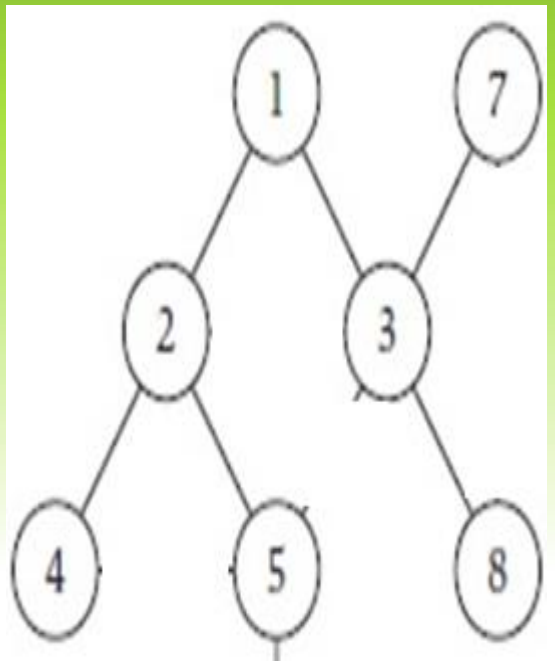
- We now discover the nodes {2,5,7,8} from the node 3 in L1.
  - 2 is already present in T
  - 5 is already present in T
  - Add 7,8 to T



L2

L2

- Now we consider each node in L2
- From 4 we discover node 5.
  - 5 is already there in T.

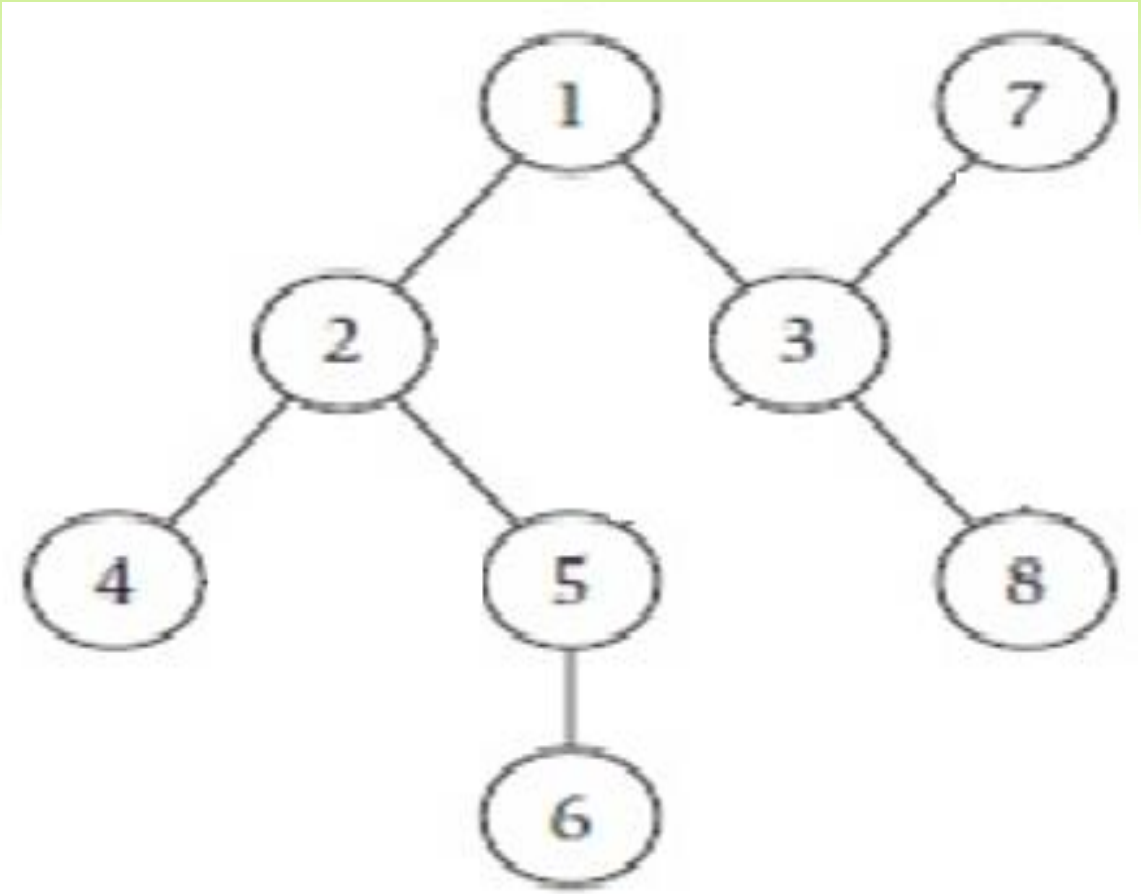


- From 5 we discover nodes {4,6}
  - 4 is already there in T
  - Add 6 to T

- From 7 we discover {3,8} and 8 we discover {3,7} which are already in T.

- For new node 6 added in L3, there are no more further nodes to be explored.
- The full BFS tree is as depicted below.

- L0



L2

L1

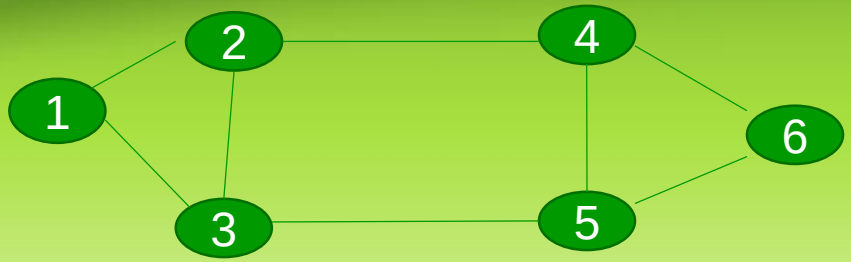
L2

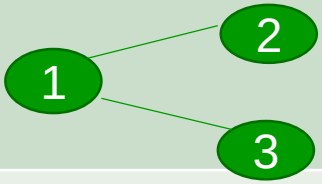
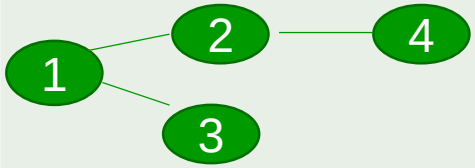
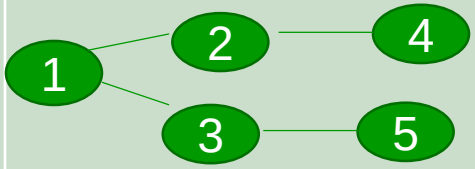
L1

L2

L3

USING THE ALGORITHM



Discovered						List		Edge	Tree
T	F	F	F	F	F	L[0] = 1	I=0		Ø
T	T	F	F	F	F	L[1] =2	I=1	U=1, V=2	
T	T	T	F	F	F	L[1] =2, 3	I=1	U=1, V=3	
T	T	T	T	F	F	L[2]= 4	I=2	U=2, V=1, V=3, V=4	
T	T	T	T	T	F	L[2] = 5	I=2	U=3, V=1, V=2, V=5	
T	T	T	T	T	T	L[3] = 6	I=3	U=4, V=2.5.6	

Now L[i] is empty. So the BFS Tree.

ALGORITHM

- **Purpose : To produce BFS tree for G=(V,E).**

- **Input : A graph G=(V,E) where v→ Vertices, E→ Edges and source vertex s.**

- **Output: A BFS tree T showing the reachable nodes from the source vertex s.**

- **A discovered array for all v E V. Initially it is zero.**

- **A list L[i] for each i=0,1,2,3……..**

```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = ∅
  While L[i] is not empty
    Initialize an empty list L[i + 1]
    For each node u ∈ L[i]
      Consider each edge (u, v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
```



```
      Add v to the list L[i + 1]
    Endif
  Endfor
  Increment the layer counter i by one
Endwhile
```

- **Runtime:** O(V+E) ; O(E) to scan through adjacency list and O(V) to visit each vertex. This is considered linear time in the size of G.

- **Claim:** BFS always computes the shortest path distance in d[i] between S and vertex I.

IMPLEMENTATION OF BFS ALGORITHM RUNS IN TIME O(MN) IF THE GRAPH IS GIVEN BY ADJACENCY LIST REPRESNTATION

- For loop processing a node “u” can take less than O(n) time if “u” has only few neighbors.
- Let n_u denote degree of node u, the **number of edges incident to node u** is $O(n_u)$, so the overall nodes is $O(\sum_{u \epsilon v} n_u)$.
- But $\sum_{u \epsilon v} n_u = 2m$, and so the total time spent considering edges over the whole algorithm is O(m).
- **NOTE:** We need an array of pointers of length *n* to set up the lists in Adj, and then we need space for all the lists. Now, the lengths of these lists may differ from node to node, each edge *e = (v, w)* appears in exactly two of the lists: the one for *v* and the one for *w*. Thus the total length of all lists is *2m = O(m)*.
- We need O(n) additional time to setup lists & manage array discovered.
- So total time spent is **O(m+n).**

- **Let T be a BFS tree, let x and y be nodes in T belonging to layers Li and Lj respectively, and let (x,y) be an edge of G. Then I & j differ by atmost 1.**

PROOF:

- By the way of contradiction, I & j differed by more than 1. Suppose I < j-1.
- _Consider a point in BFS algorithm when edge incident to “X” were being examined.
- Since “x” belongs to layer Li, the nodes discovered from “x” belongs to layer Li+1.
- If y is neighbor of “x” then it should have been discovered by this point & hence should belong to layer Li+1.

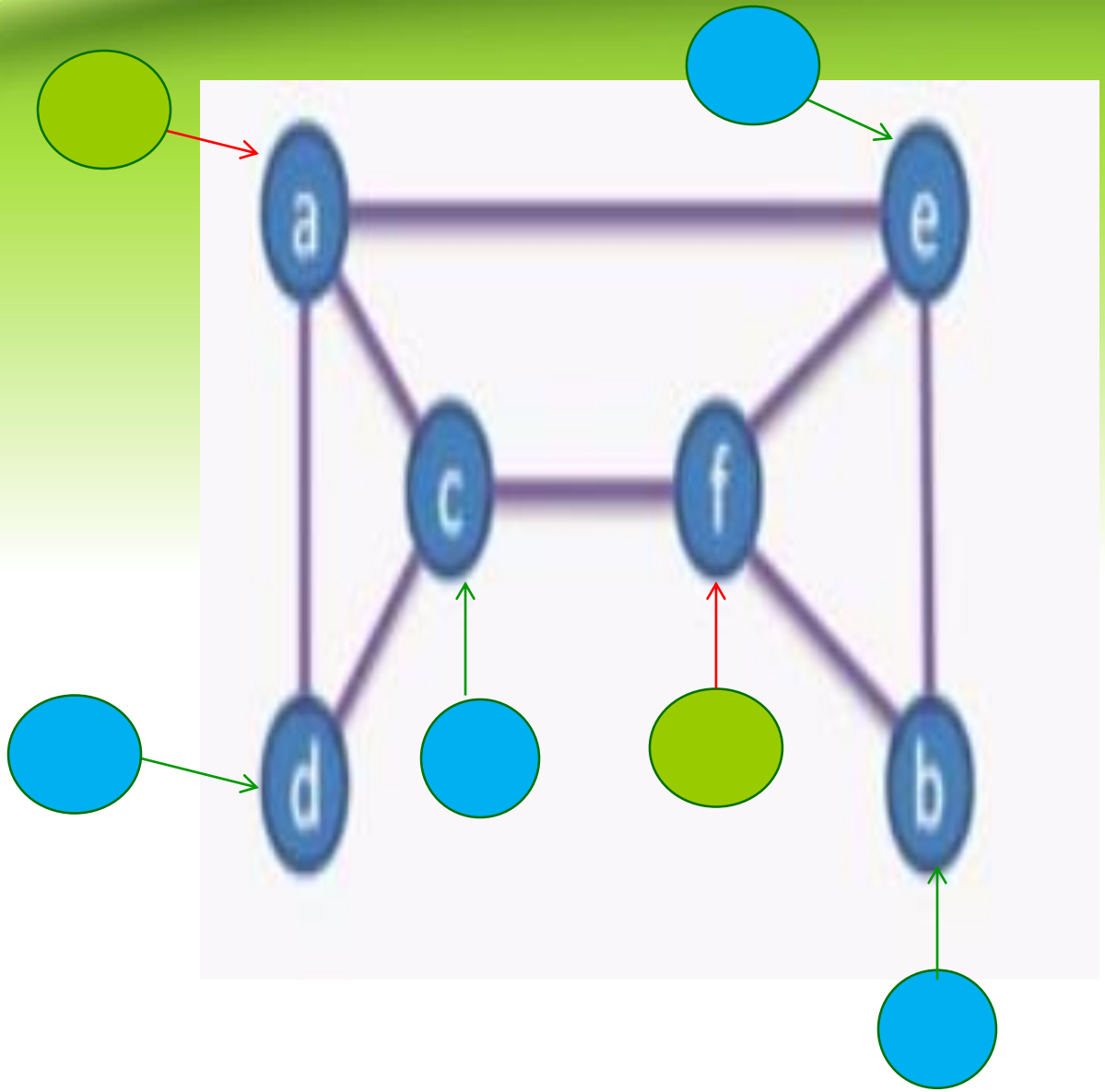
APPLICATIONS OF BFS(testing the given graph is biparatite or not)

PROBLEM:

- Clearly a triangle is not bipartite, since we can color one node **red**, another one **blue**, and then we can't do anything with the third node.
- consider a cycle *C* of odd length If we color node 1 red, then we must color node 2 blue, and then we must color node 3 red, and so on, then we must color node $2k + 1$ red, and it has an edge to node 1, which is also red. This demonstrates that there's no way to partition *C* into red and blue nodes as required.
- **If a graph G is bipartite, then it cannot contain an odd cycle.**

DESIGNING THE ALGORITHM

- First we assume the graph *G* is connected.
- we pick any node $s \in V$ and color it **red** and all the neighbors of *s* must be colored **blue**, then follows that all the neighbors of *these* nodes must be colored **red** and so on, until the whole graph is colored.
- Either we have a valid red/blue coloring of *G*, in which every edge has ends of opposite colors, or there is some edge with ends of the same color.
- Coloring procedure we have just described is essentially identical to the description of BFS
- *s* red, all of layer $L1$ blue, all of layer $L2$ red, and so on, coloring odd-numbered layers blue and even-numbered layers red.


a
e
c
f
d
b

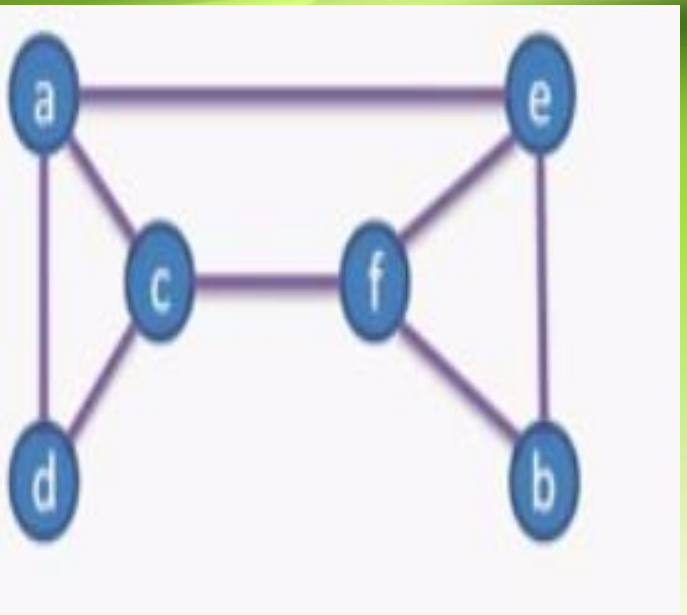
Depth First Search

Depth first search is an algorithm for traversing a graph in search of an element.

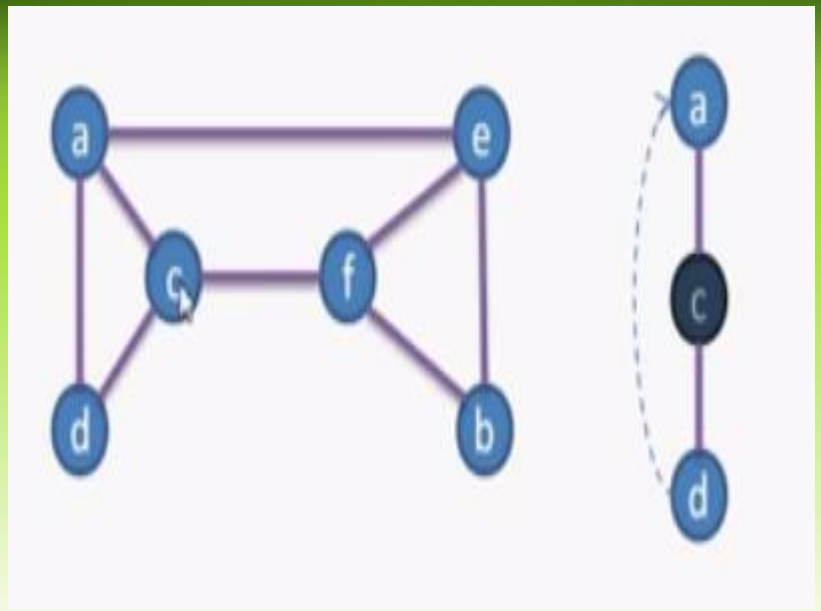
Start at a node/element of the graph and go deep on a single branch till we either hit an element or a dead end.

If we hit a dead end then backtrack and travel on a new branch.

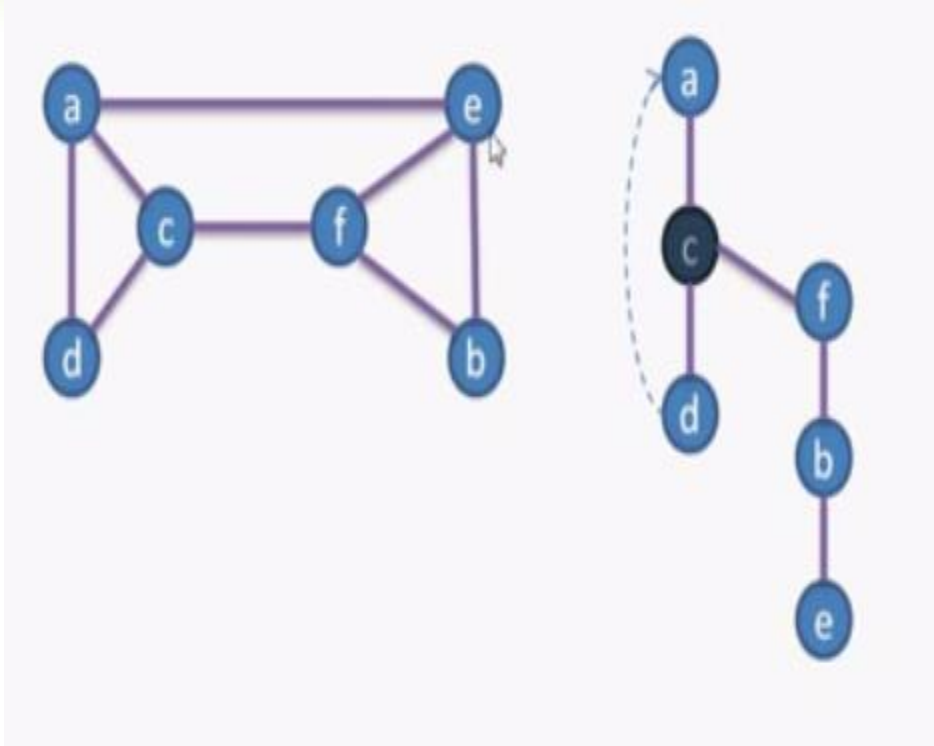
DFS can be used to check connectivity of a graph (i.e. where the graph has disjointed sub-graphs or isolated vertices).


a
e
c
f
d
b

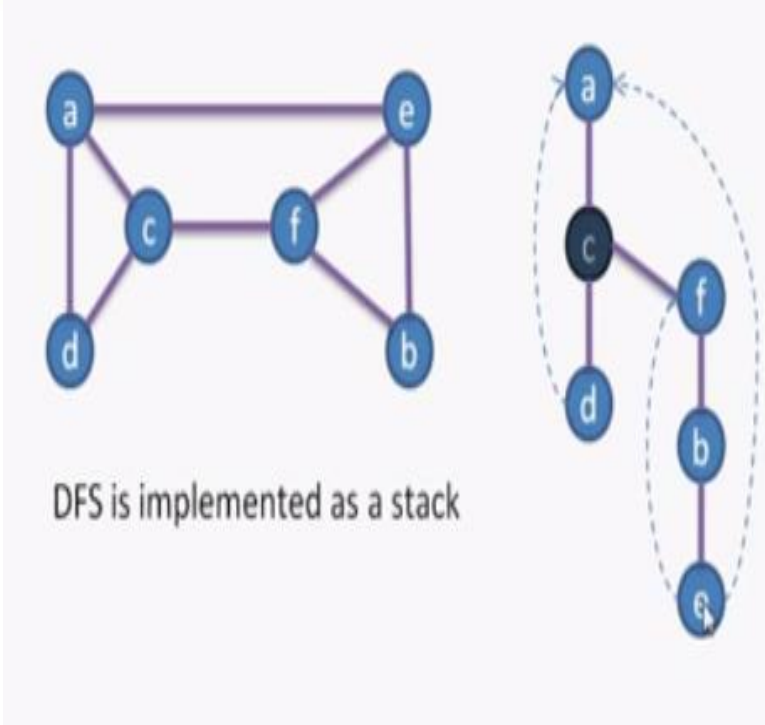
1


a
e
c
f
d
b
a
c
d

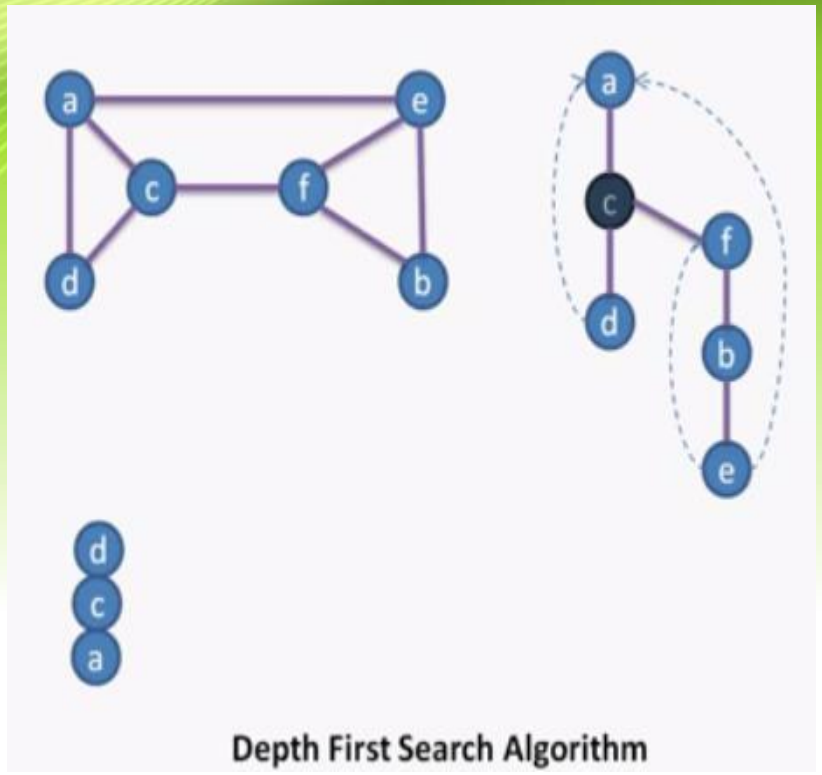
2


a
e
c
f
d
b
a
c
f
d
b
e

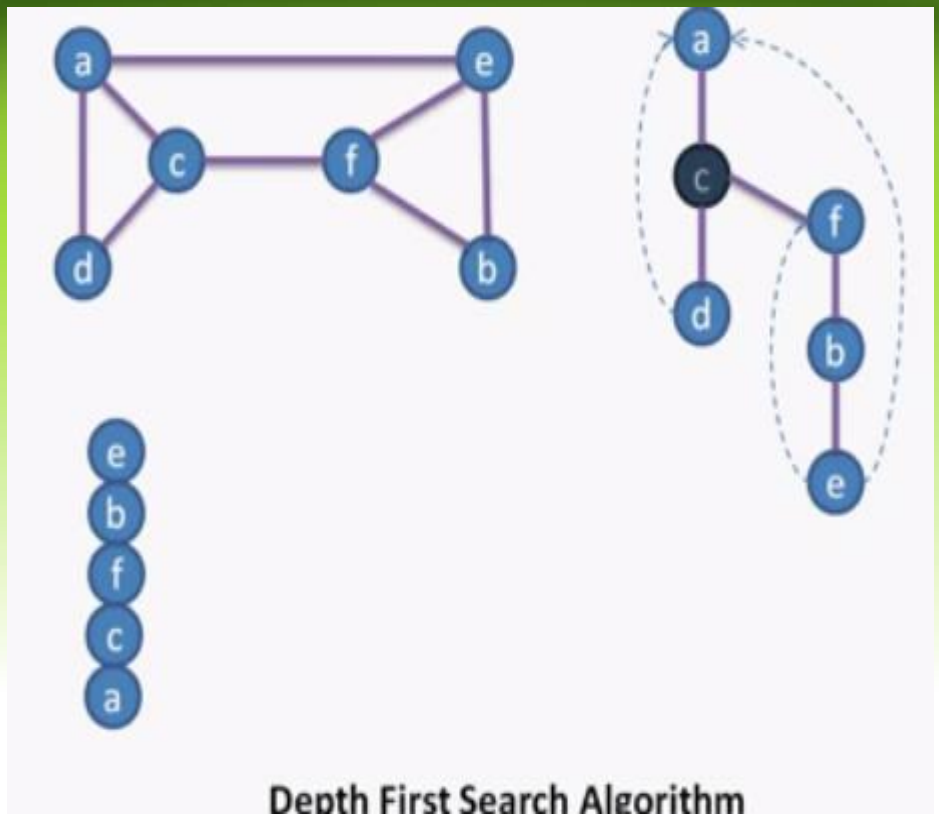
3


a
e
c
f
d
b
DFS is implemented as a stack
a
c
f
d
b
e

4


a
e
c
f
d
b
a
c
f
d
b
e
d
c
a
Depth First Search Algorithm

5


a
e
c
f
d
b
a
c
f
d
b
e
e
b
f
c
a
Depth First Search Algorithm

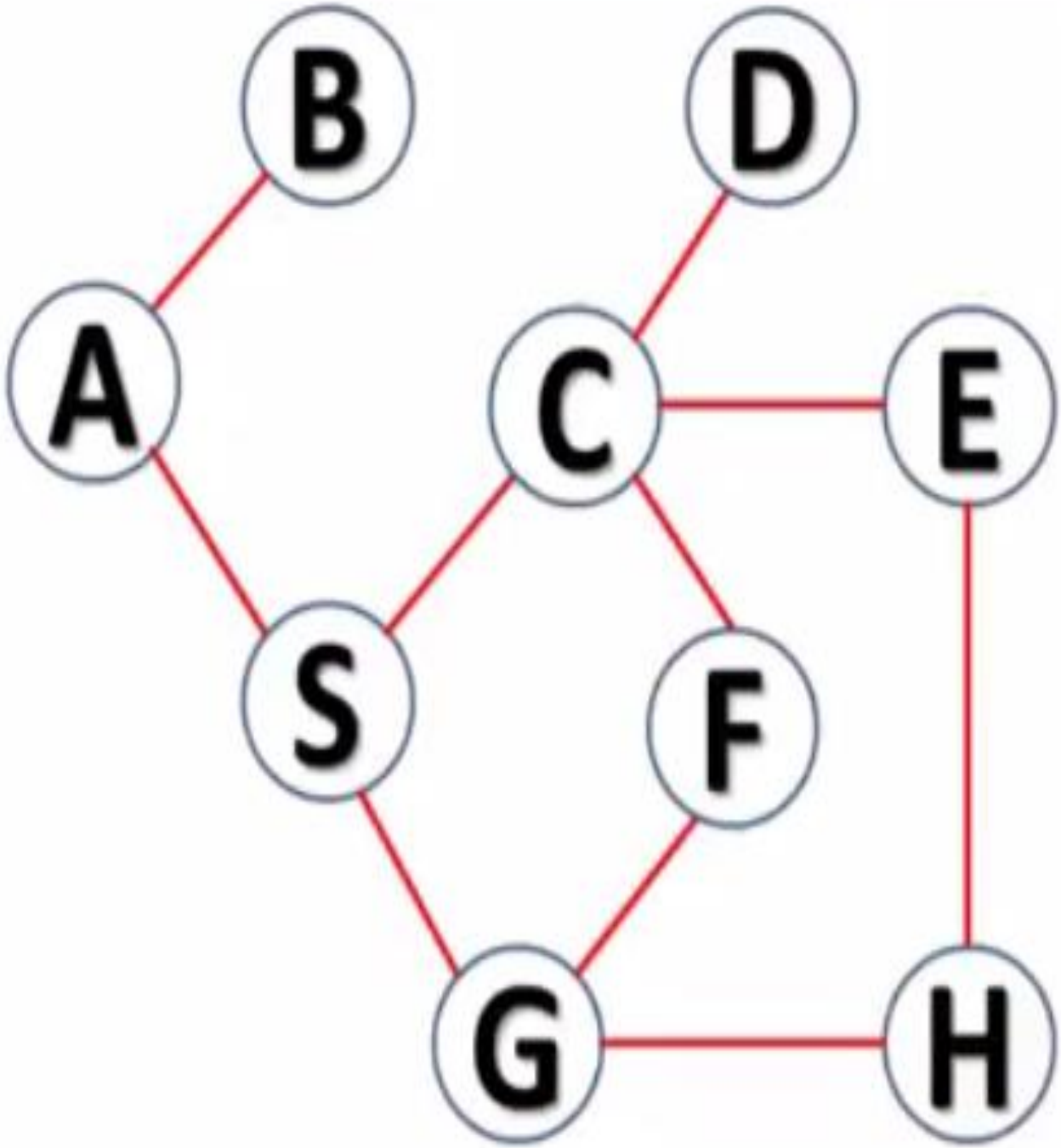
6


a
e
c
f
d
b
a
c
f
d
b
e
a
Depth First Search Algorithm

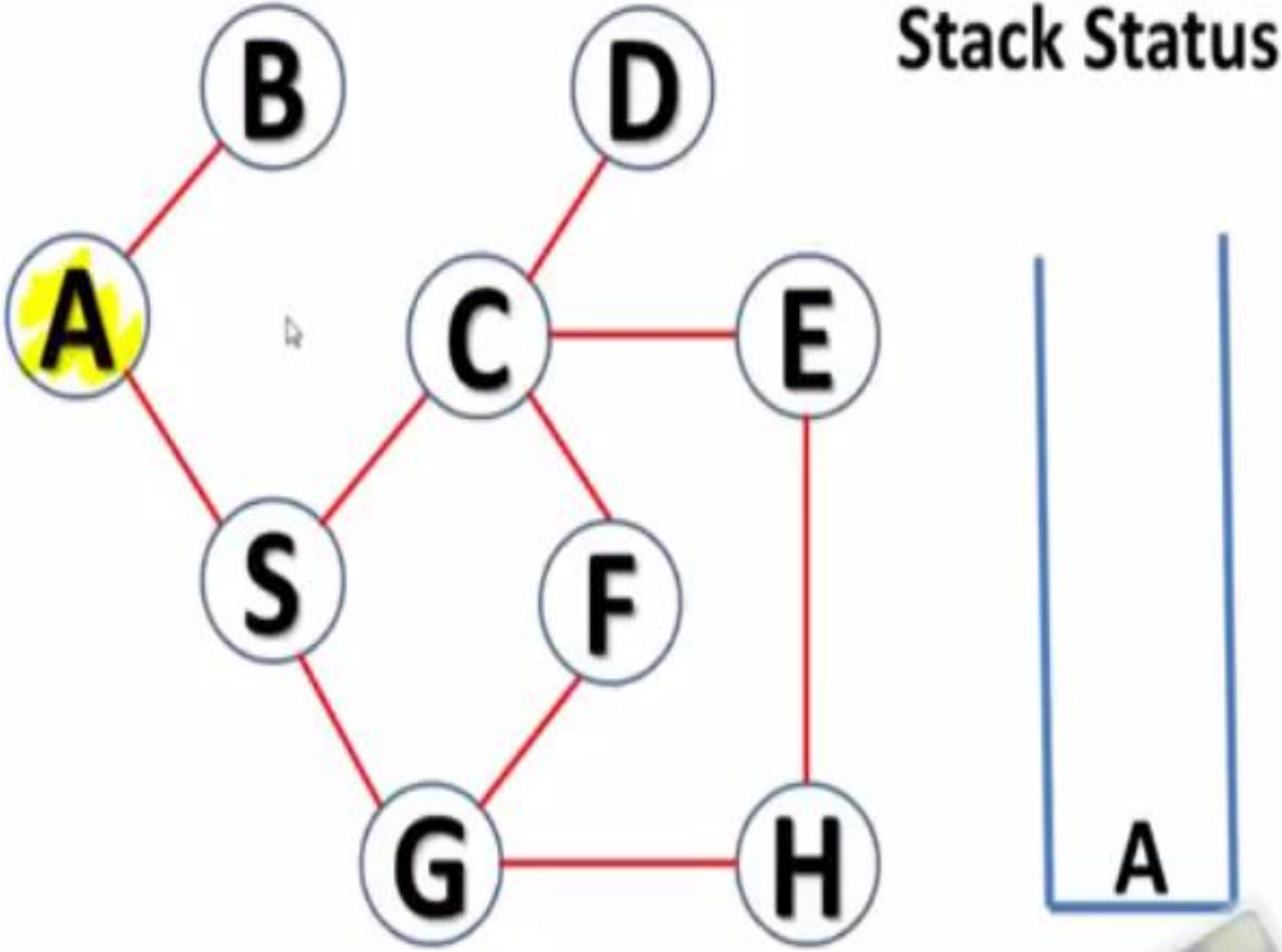
```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

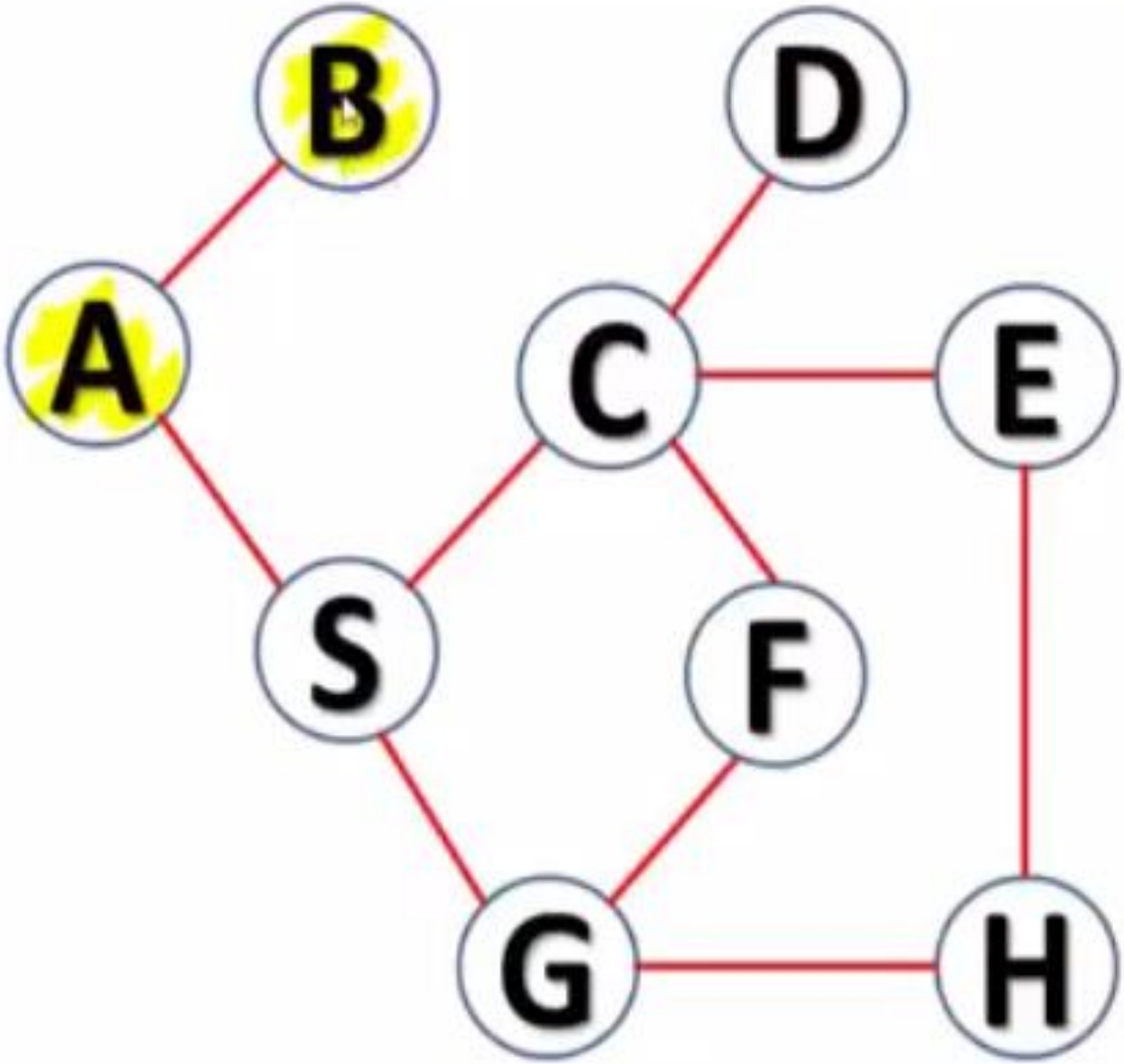
```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

```
DFS-recursive(G, s):
        mark s as visited
        for all neighbours w of s in Graph G:
            if w is not visited:
                DFS-recursive(G, w)
```



Stack Status

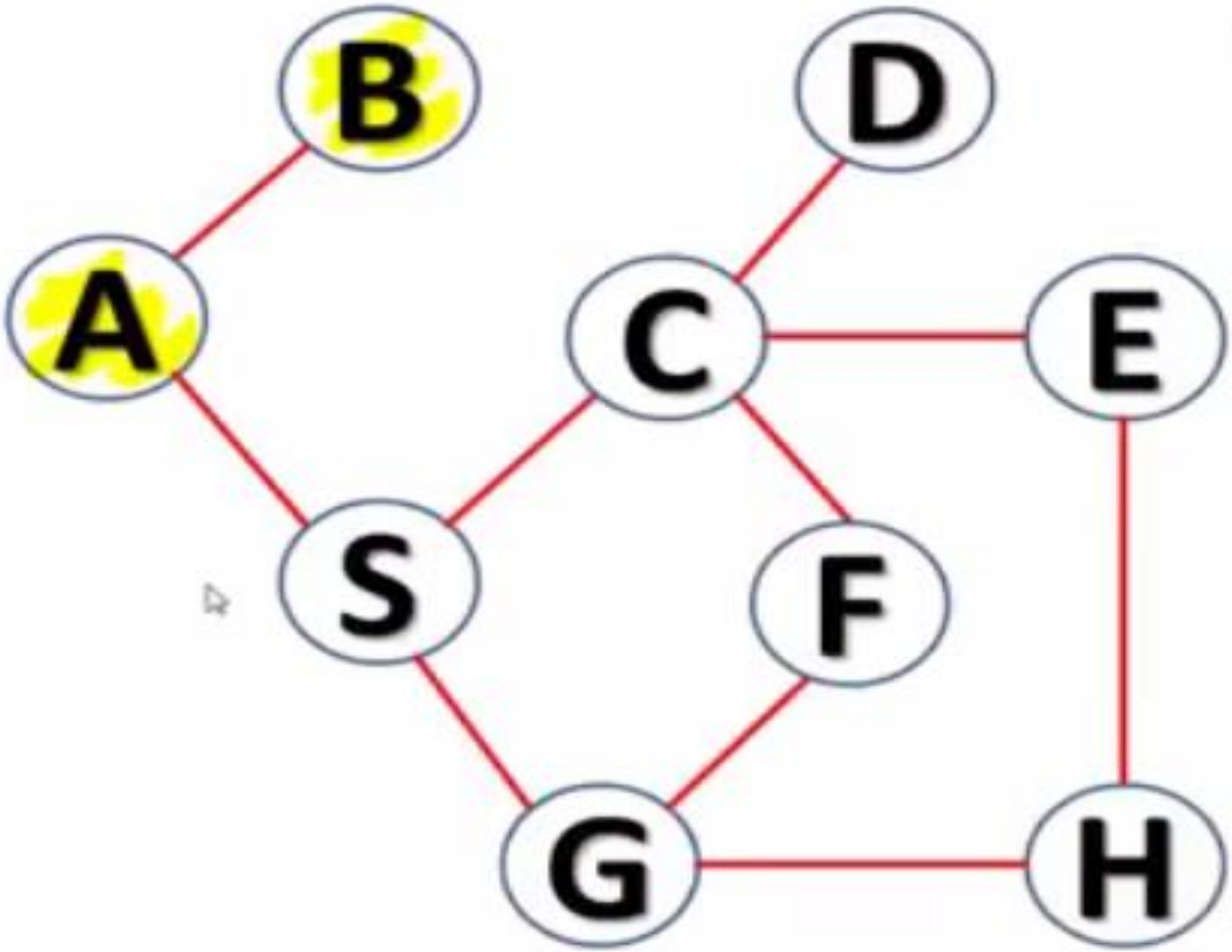

Stack Status
B
D
A
C
E
S
F
G
H
A



Stack Status

OUTPUT : A B

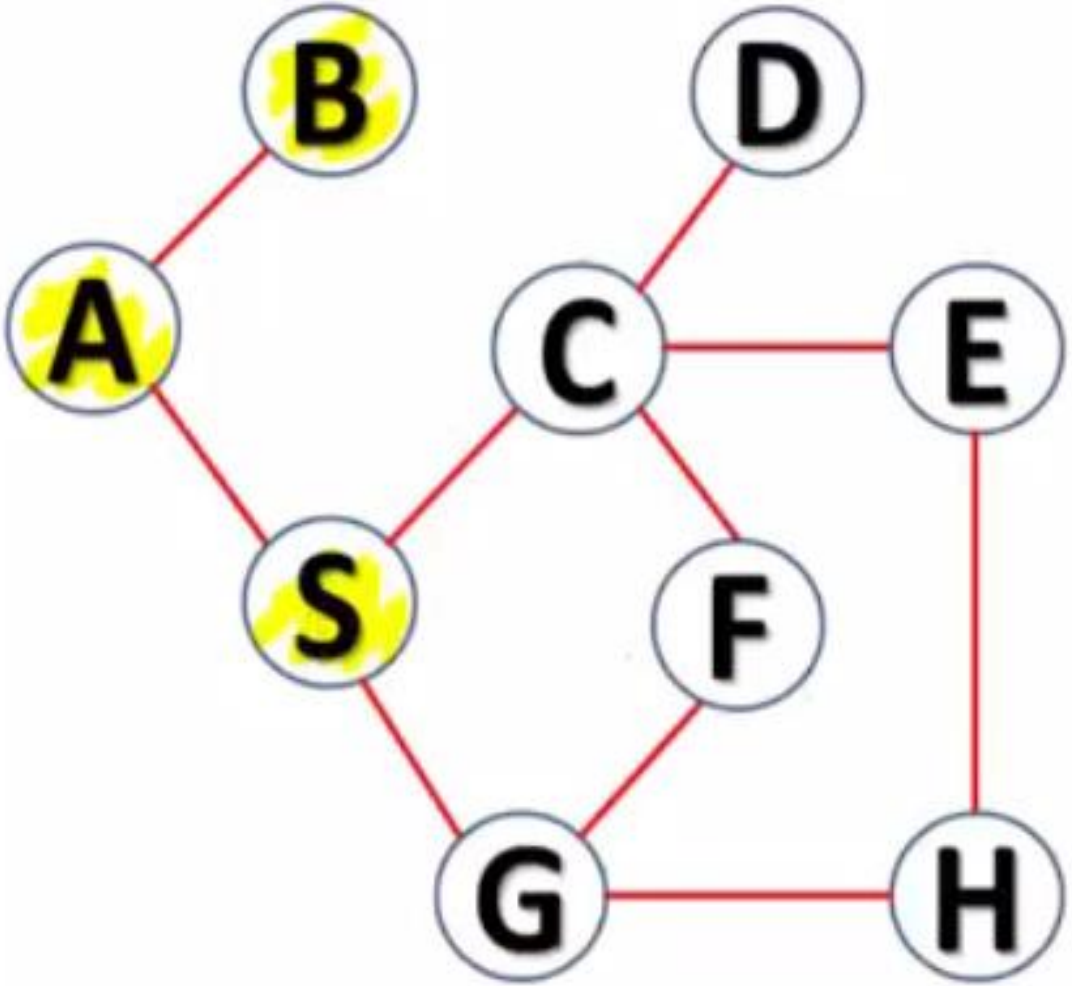
DEPTH FIRST SEARCH



Stack Status

OUTPUT : A B

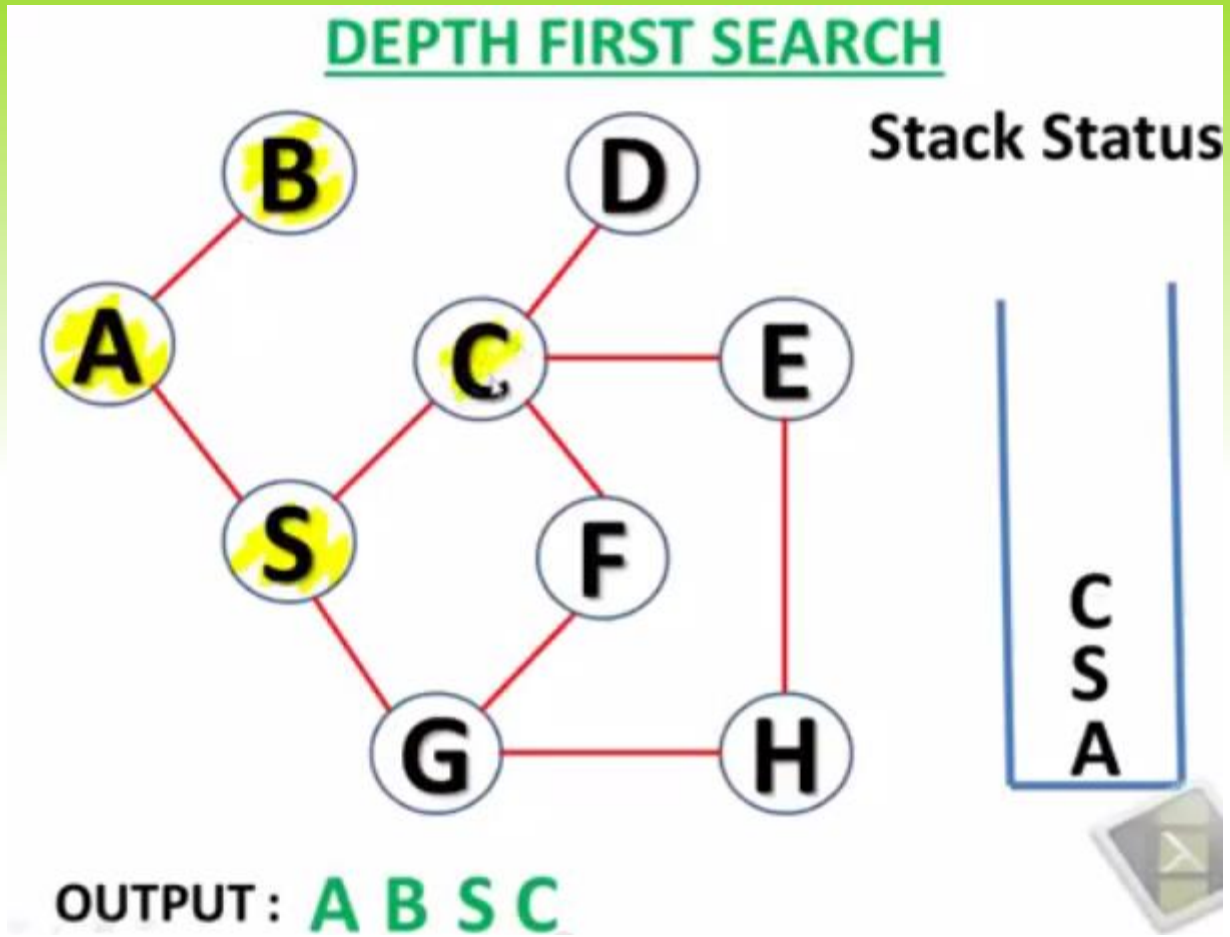
DEPTH FIRST SEARCH

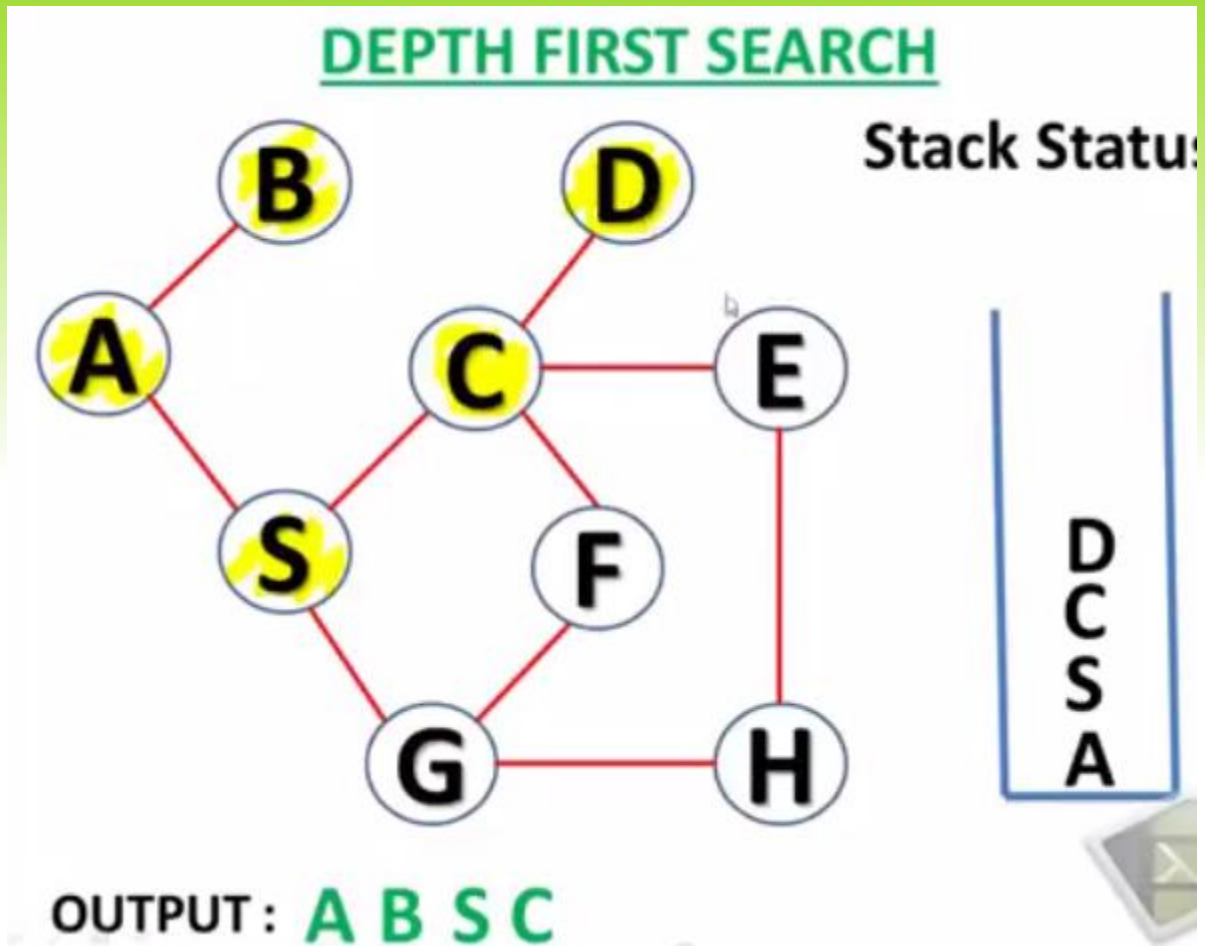


Stack Status

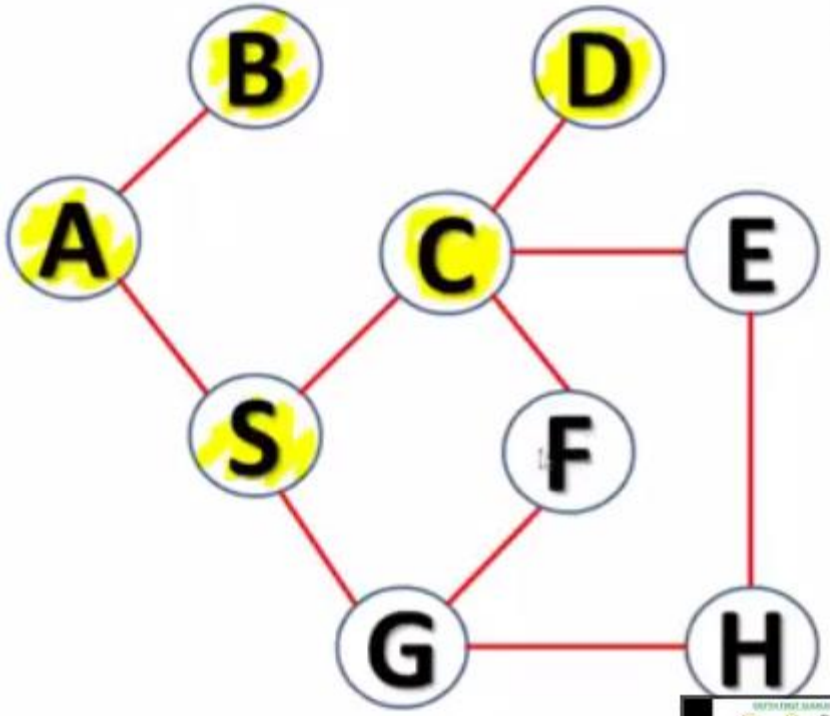
OUTPUT : A B S

DEPTH FIRST SEARCH



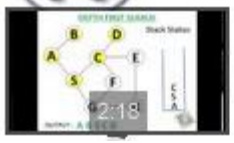

DEPTH FIRST SEARCH
Stack Statu
B
D
A
C
E
S
F
G
H
D
C
S
A
OUTPUT : A B S C

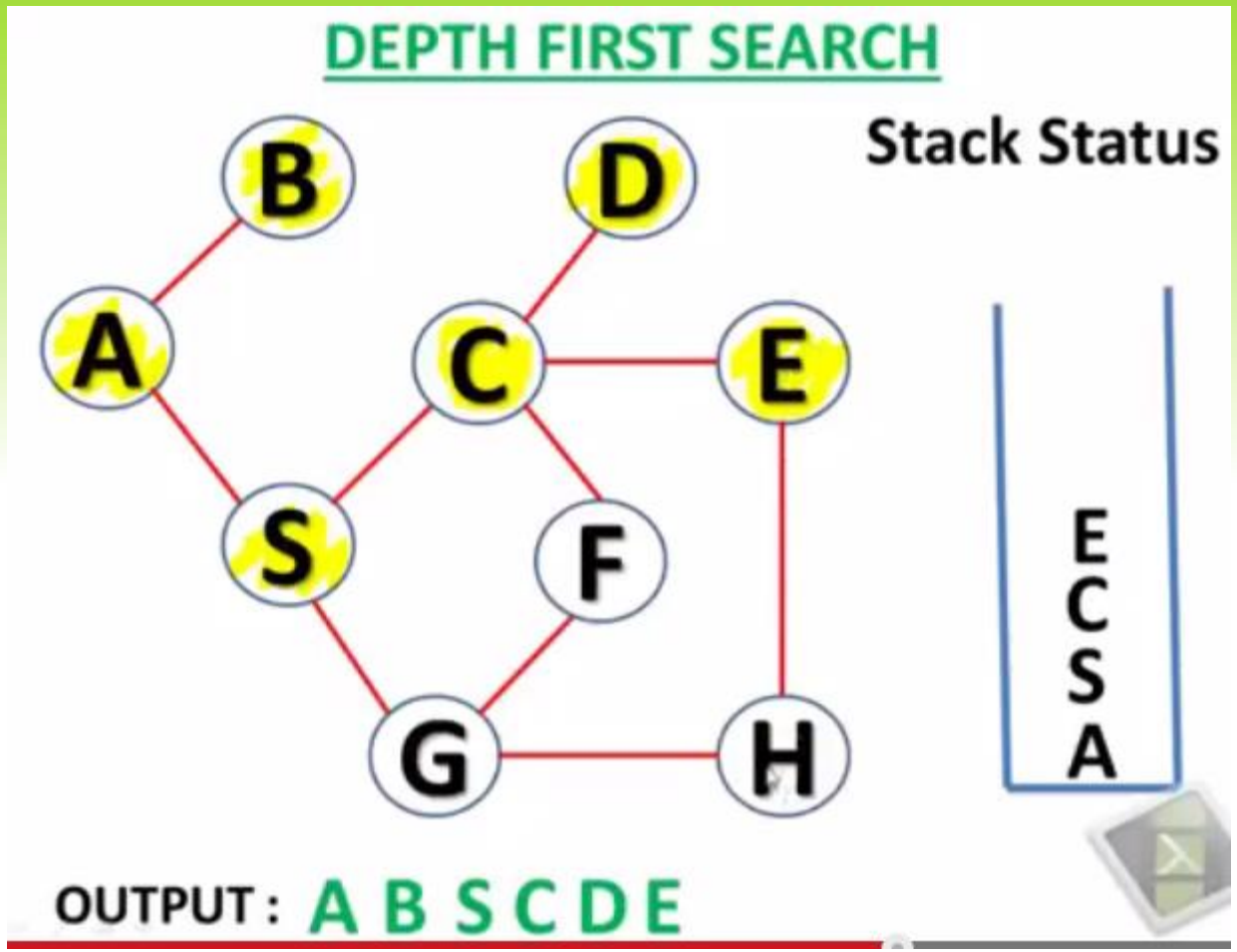
DEPTH FIRST SEARCH



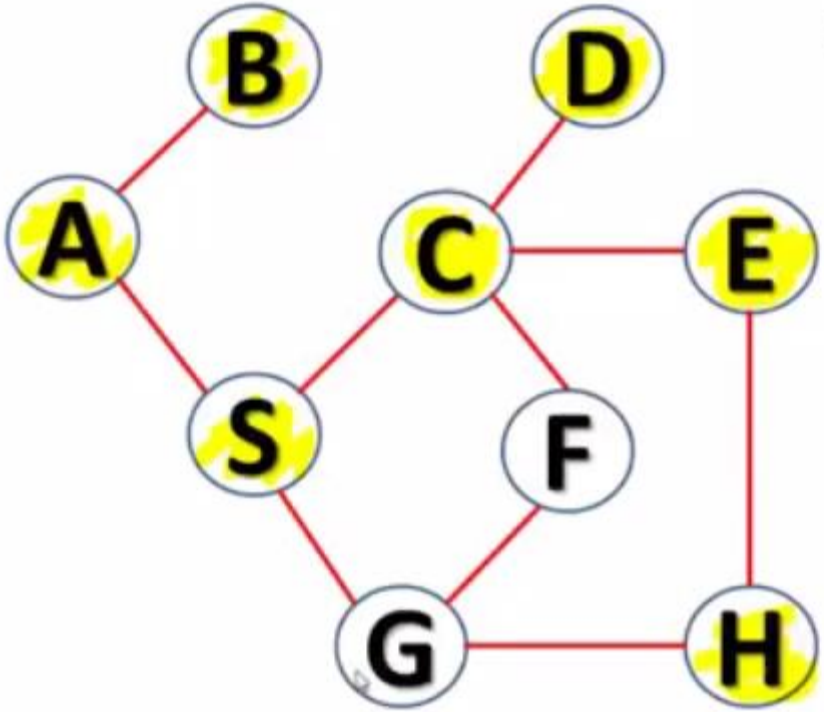
Stack Status

OUTPUT : A B S C D




DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
E
C
S
A
OUTPUT : A B S C D E

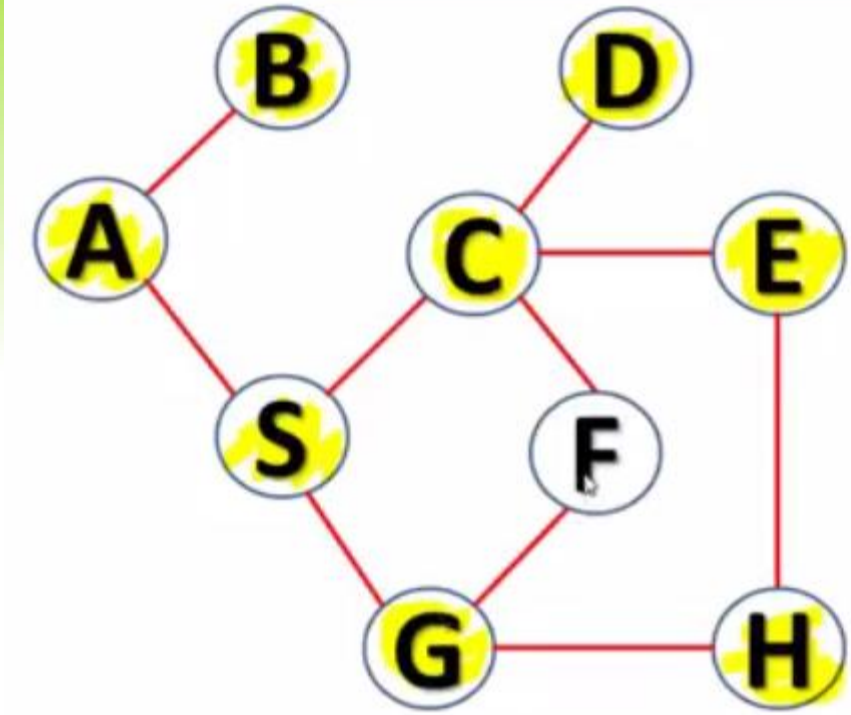
DEPTH FIRST SEARCH



Stack Status

OUTPUT : A B S C D E H

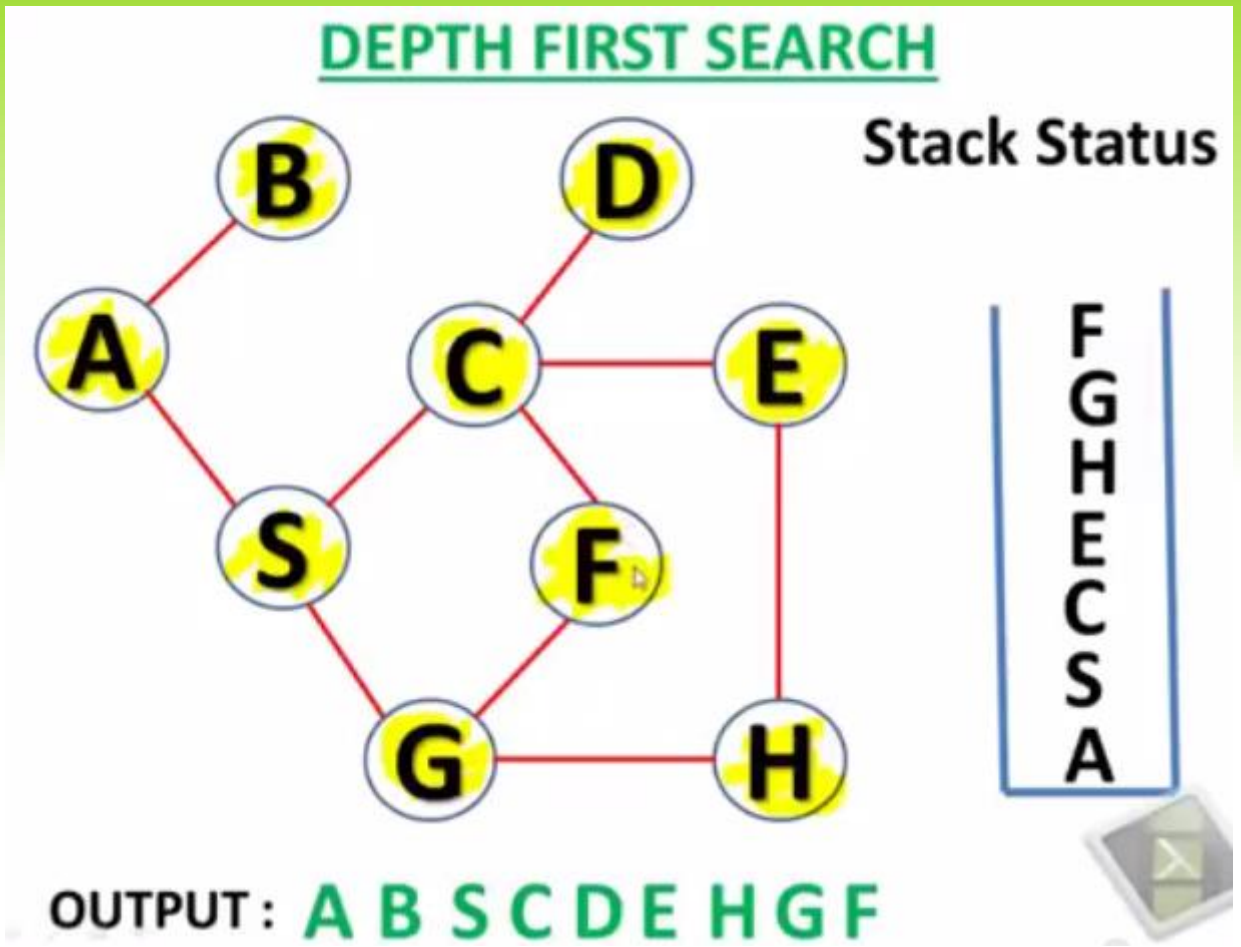
DEPTH FIRST SEARCH

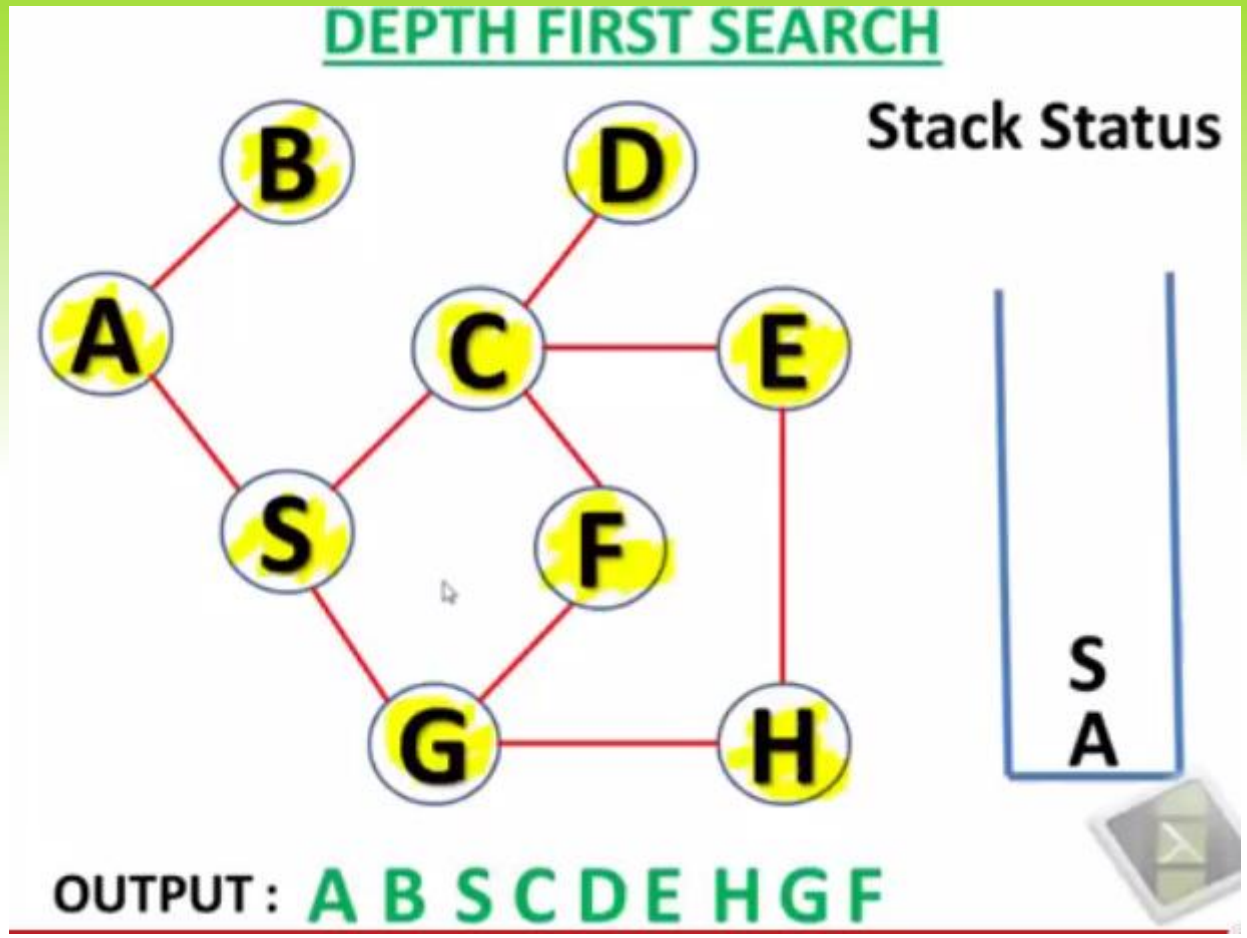


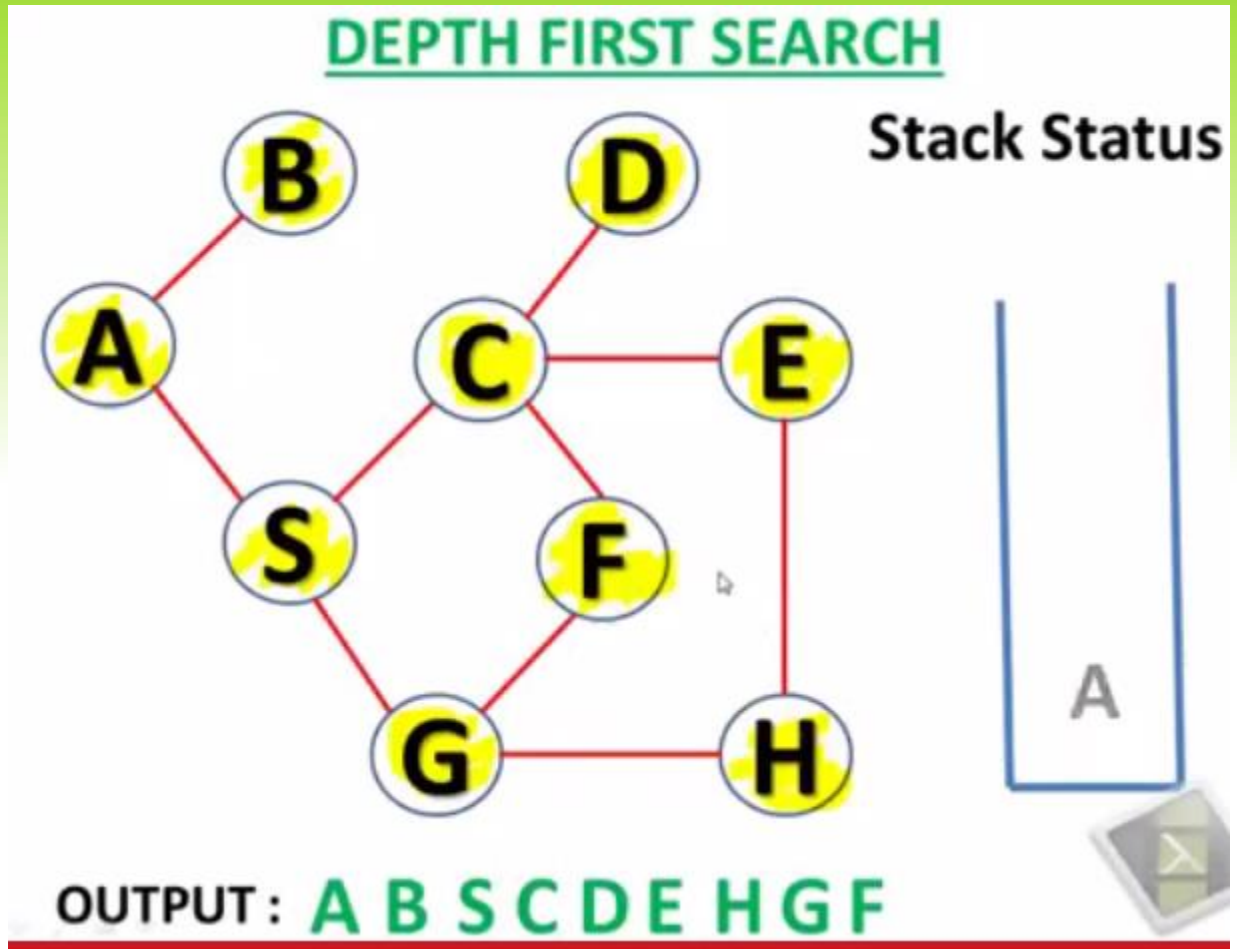
Stack Status

OUTPUT : A B S C D E H G

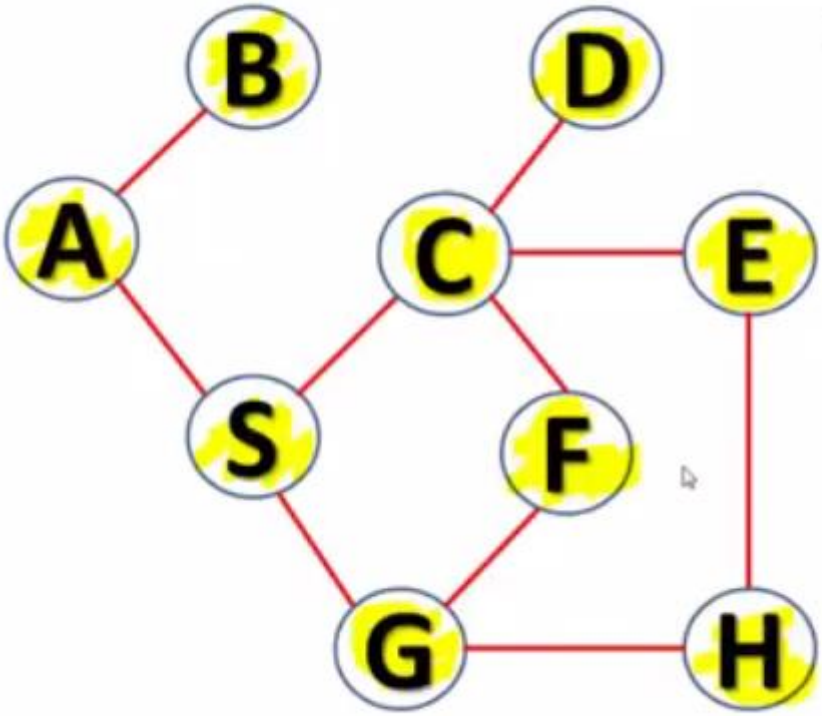
DEPTH FIRST SEARCH




DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
S
A
OUTPUT : A B S C D E H G F


DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
A
OUTPUT : A B S C D E H G F

DEPTH FIRST SEARCH



Stack Status

Stack Empty

OUTPUT : A B S C D E H G F

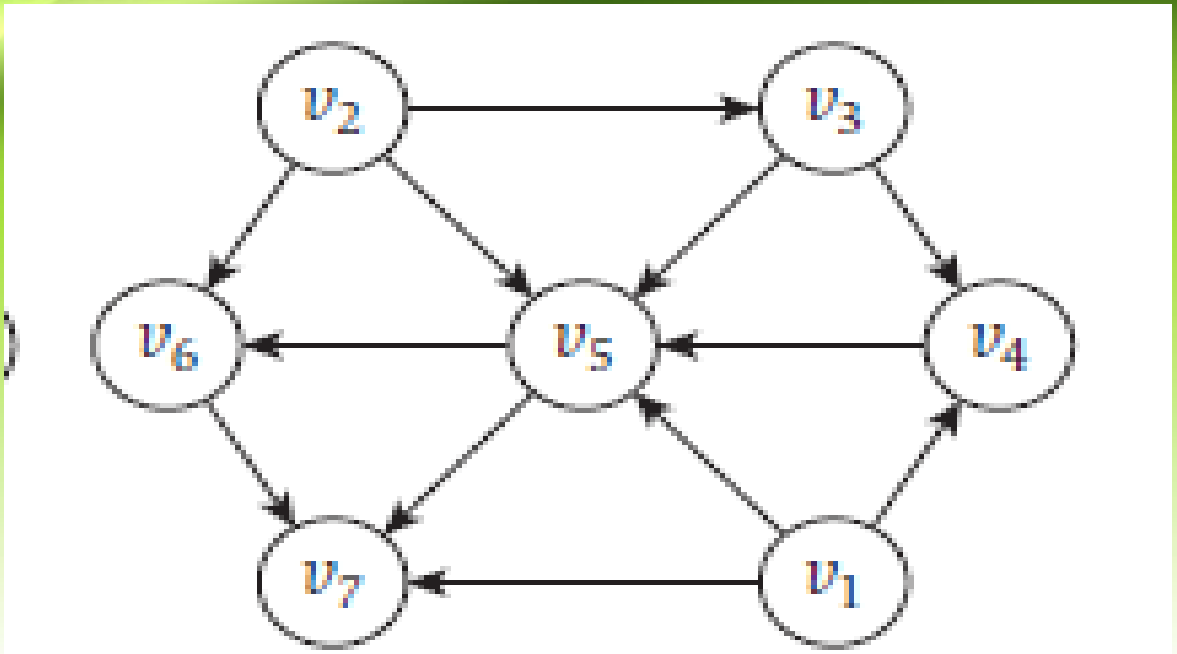
DAG & TOPOLOGICAL ORDERING

- If a directed graph has no cycle, we call it a "Directed Acyclic Graph" or DAG.

Problem

- Suppose we have a set of tasks labeled {1, 2, . . . , *n*} that need to be performed, and there are dependencies among them.
- For certain pairs *i* and *j*, that *i* must be performed before *j*. For example, the tasks may correspond to a **pipeline of computing jobs**, with assertions that the output of job *i* is used in determining the input to job *j*, and hence job *i* must be done before job *j*.

- We represent such an interdependent set of tasks by introducing a node for each task, and a directed edge *(i, j)* whenever *i* must be done before *j*.
- If it contained a cycle *C*, there would be no way to do any of the tasks in *C*: since each task in *C* cannot begin until some other one completes, no task in *C* could ever be done, since none could be done first.
- For a directed graph *G*, we say that a *topological ordering* of "*G"* is an ordering of its nodes as *v*1, *v*2, . . . , *vn* so that for every edge *(vi , vj)*, we have $i < j$. i.e all edges point "forward" in the ordering.

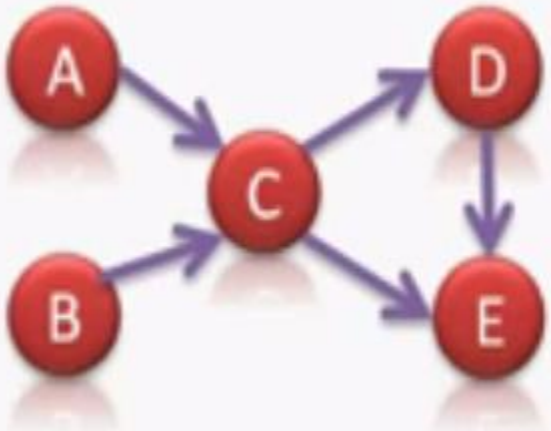


- In this we have labelled the nodes with a topological ordering.

Topological Sorting Algorithm

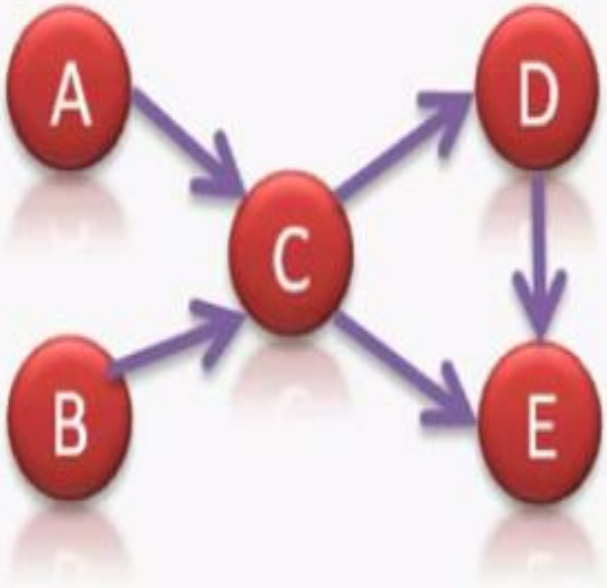
•If we have a directed graph G then topological sorting/ordering of the graph is a "linear" order of the vertices of the graph such that if "uv" is an edge in the graph then "u" should come before "v" in the topological ordering.

•Lets assume that a student has to take five courses/subjects before he gets a degree and they can be represented as below dependency graph. In what order should he study the courses so that the dependencies are satisfied.



Topological Sorting

Algorithm 1 : Topological Sorting using Depth First Search

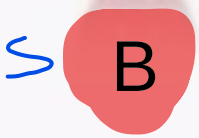


when all the neighbors are visited, push to stack after recursive call

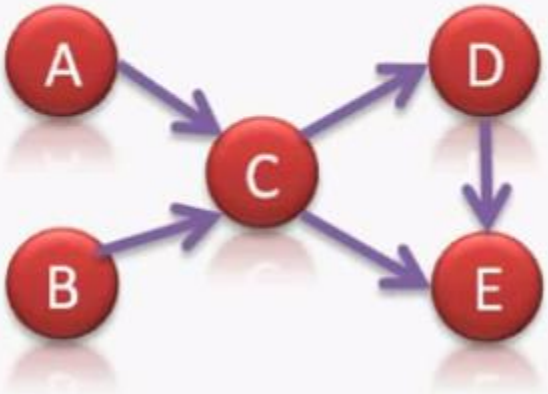
Stack

top

B added later as an unvisited component



Algorithm 1 : Topological Sorting using Depth First Search



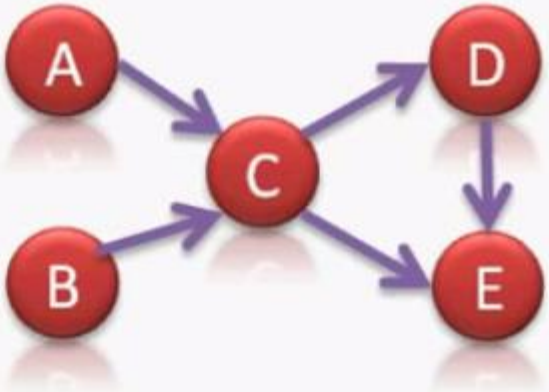
Stack

Pop Elements

Reading Backwards

Topological Sorting

Algorithm 1 : Topological Sorting using Depth First Search



Stack

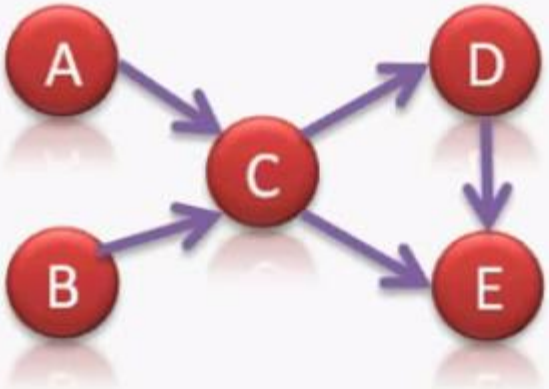
Pop Elements

Reading Backwards

B A C D E

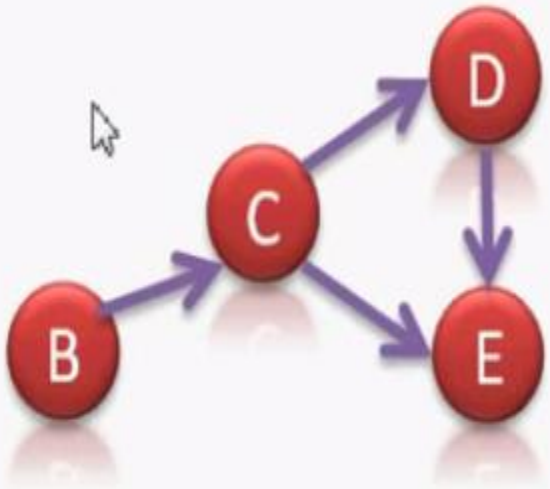
Source Removal

Algorithm 2 : Topological Sorting using Source Removal



Topological Sorting

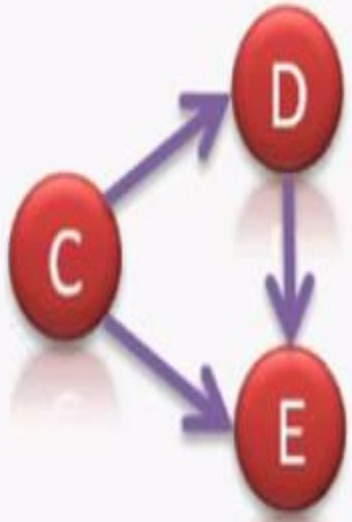
Algorithm 2 : Topological Sorting using Source Removal



Sorted List

A

Algorithm 2 : Topological Sorting using Source Removal



Sorted List

A B

Algorithm 2 : Topological Sorting using Source Removal

Sorted List

A B C

Algorithm 2 : Topological Sorting using Source Removal

Sorted List

Algorithm 2 : Topological Sorting using Source Removal

Sorted List

- **IF G HAS A TOPOLOGICAL ORDERING, THEN G IS A DAG.**

- **PROOF:** By the way of contradiction, that G has a topological ordering v1,v2,v3v, , , ,vn, and also has a cycle.
- Let Vi is the lowest- indexed node on C, and let Vj be the node on C just before Vi, thus (Vj,Vi) is an edge.

- But by our choice of i, we have j>i, which contradicts the assumption that v1,v2,v3, .. ,vn was a topological ordering.