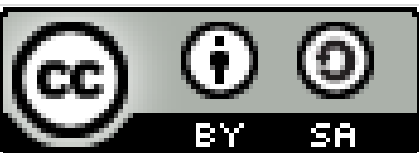


**Subject : Data Structures**  
**Subject\_code : CS-2201**  
**Course : B.Tech.(III Sem.)**

By  
Poonam Saini  
Department of Computer Science & Engineering  
Sir Padampat Singhania University  
Udaipur

# Graphs



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/). This presentation is released under Creative Commons-Attribution, on 4.0 License. You are free to use, distribute and modify it, including for commercial purposes, provided you acknowledge the source.

# Introduction

---

- A graph  $G$  consists of two things:
  - A set  $V$  of elements called nodes/vertices
  - A set  $E$  of edges such that edge  $e$  in  $E$  is identified with a unique pair  $[u,v]$  of nodes in  $V$  denoted by
$$e=[u,v] \quad \text{or} \quad G=(V,E)$$

Suppose  $e=[u,v]$ , then  $u$  and  $v$  are called the endpoints of  $e$  and are said to be adjacent nodes or neighbours.

---

# Graphs

---

## □ Degree of a Node

$\deg(u)$ , is the no. of edges containing  $u$ .

If  $\deg(u)=0$  i.e.  $u$  does not belong to any edge then  $u$  is called the isolated node.

## □ Path $P$ from $u$ to $v$

Path  $P$  of length  $n$  from node  $u$  to  $v$  is defined as a sequence of  $n+1$  nodes.

$$P=(v_0, v_1, v_2, \dots, v_n)$$

Such that  $u=v_0$  and  $v=v_n$  If  $v_0=v_n$  then  $P$  is said to be closed.

# Graphs

---

## □ Cycle

**A cycle is a closed simple path with length 3 or more. A cycle of length  $k$  is called a  $k$ -cycle.**

---

# Propositions

---

## □ **Connected Graph**

A graph  $G$  is said to be a connected graph if there is a path between any two of its nodes.

## □ **Complete Graph**

A graph  $G$  is said to be a complete if every node  $u$  in  $G$  is adjacent to every other node  $v$  in  $G$ . A complete graph with  $n$  nodes will have  $n(n-1)/2$  edges.

---

# Propositions

---

## □ Tree Graph

A connected graph  $T$  without any cycles is called a tree graph or free graph or simply a tree. If  $T$  is a finite tree with  $m$  nodes then  $T$  will have  $m-1$  edges.

## □ Labelled/Weighted Graph

A graph  $G$  is said to be labelled if its edges are assigned data.  $G$  is said to be weighted if each edge  $e$  in  $G$  is assigned a nonnegative numerical value  $w(e)$  called the weight or length of  $e$ .

# Prepositions

---

## □ Multiple Edges

Distinct edges  $e$  and  $e'$  are called multiple edges if they connect the same endpoints i.e.  $e=[u,v]$  and  $e'=[u,v]$  and the graph having multiple edges is known as a multigraph.

## □ Loops

An edge  $e$  is called a loop if it has identical endpoints i.e.  $e=[u,v]$

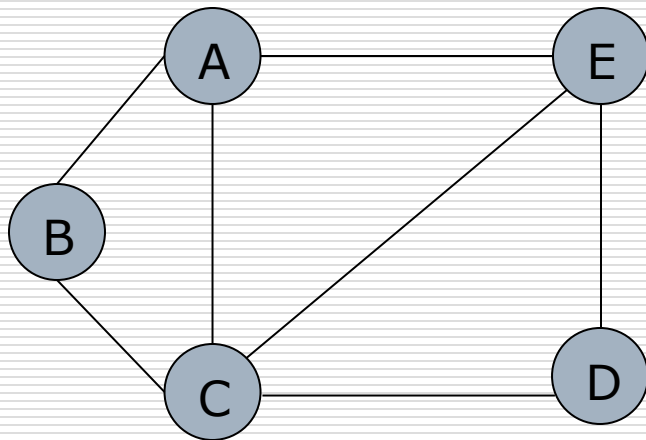
## □ Finite Multigraph

A multigraph  $M$  is said to be finite if it has a finite number of nodes and a finite no. of edges.



# Graph Examples

---



## Graph

$\deg(A)=3$ ,  $\deg(c)=4$

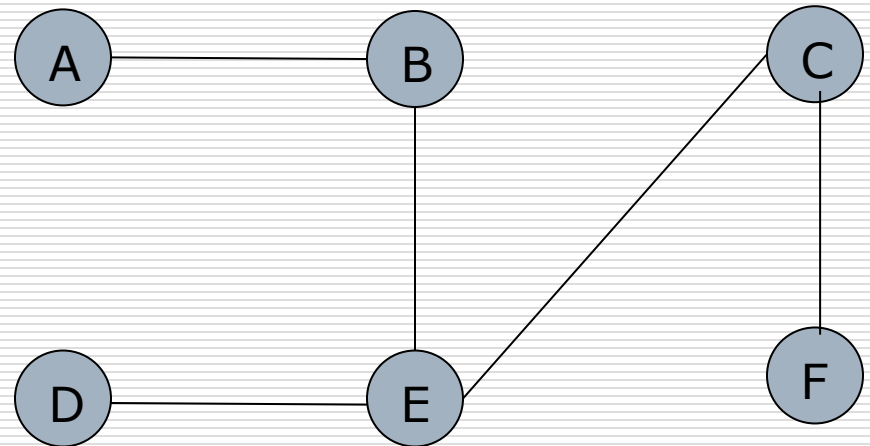
Edges=7

Path length from B to E:

(B,A,E) and (B,C,D,E)

4-cycles:

(A,B,C,E,A), (A,C,D,E,A)



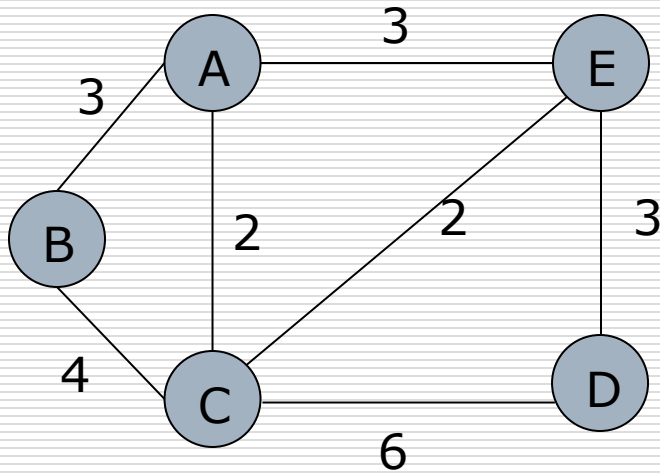
## Tree

Nodes=  $m=6$

Edges= $m-1=5$

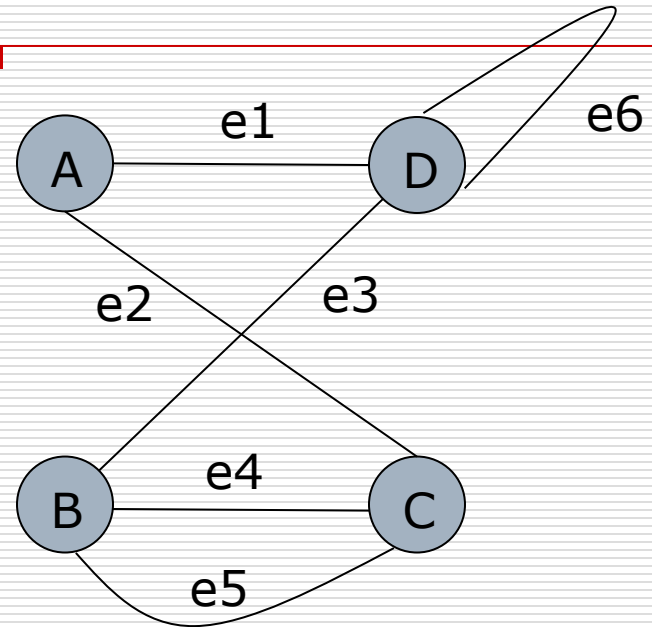
There is a unique simple path between any two nodes of the tree graph

# Graph Examples



## Weighted Graph

$P1=(B,C,D)$  and  
 $P2=(B,A,E,D)$  are both  
paths from node B to D.  
Although P2 contains  
more edges than P1 the  
weight  $w(P2)=9$  is less  
than the weight  $w(P1)=10$



## Multigraph

Multiple edges[BC] i.e. e4  
and e5  
Loop e6=[D,D]

# Directed Graph

---

- A directed graph  $G$  also called a dgraph or graph is the same as a multigraph except that each edge  $e$  in  $G$  is assigned a direction or in other words each edge  $e$  is identified with an ordered pair  $(u,v)$  of nodes in  $G$  rather than an unordered pair  $[u,v]$ .
  - Suppose  $G$  is a directed graph with directed edges  $e=(u,v)$ . Then  $e$  is also called an arc
-

# Directed Graph

---

- $e$  begins at  $u$  and ends at  $v$ .
- $u$  is the origin or initial point of  $e$  and  $v$  is the destination or terminal point of  $e$ .
- $u$  is the predecessor of  $v$  and  $v$  is successor or neighbour of  $u$ .
- $u$  is adjacent to  $v$  and  $v$  is adjacent to  $u$ .
- Outdegree of  $u = \text{outdeg}(u) =$  No. of edges beginning at  $u$ .
- Indegree of  $u = \text{indeg}(u) =$  No. of edges ending at  $u$

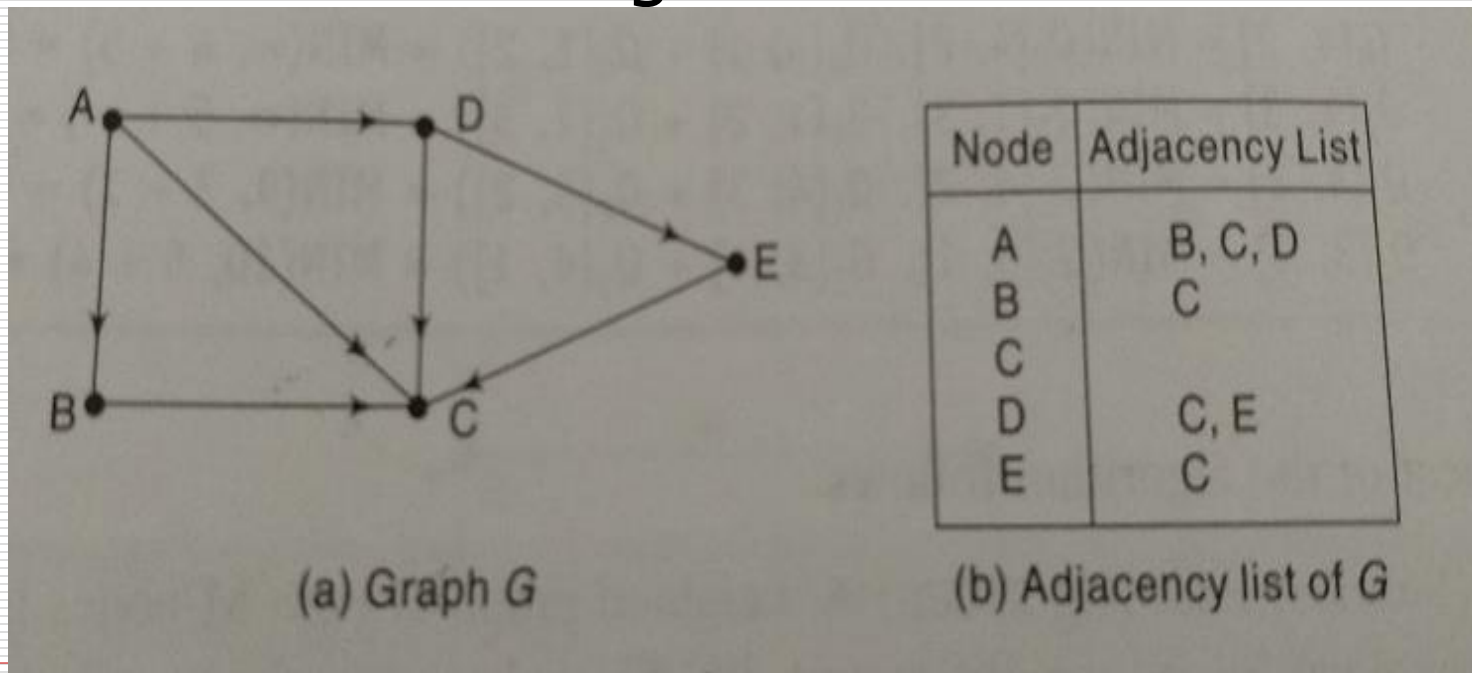
# Directed Graph

---

- Source: If a node has a positive degree but zero indegree is known as a source
  - Sink: If a node has a zero outdegree but positive indegree then it is called a sink.
-

# Linked Representation of a Graph

- Consider the given graph G. The adjacency list of each node in G contains its successor or neighbours.



# Linked Representation of a Graph

---

□ Note:

The linked representation contains two lists: A node list NODE and an edge list EDGE

**(a) NODE List:**

Each element in the list NODE will correspond to a node in G and it will be a record of the form

<b>NODE</b>	<b>NEXT</b>	<b>ADJ</b>	<b>//////////</b>
-------------	-------------	------------	-------------------

NODE-> Name or key value of the node

NEXT-> Pointer to the next node in the list

ADJ-> Pointer to the first element in adjacency list of the node which is maintained in the list EDGE.

The shaded area indicates that there may be other information in the record such as indegree or outdegree.

# Linked Representation of a Graph

---

## (b) Edge List:

Each element in the list EDGE will correspond to an edge of G and it will be a record of the form

<b>DEST</b>	<b>LINK</b>	
-------------	-------------	--

**DEST**->Location in the list **NODE** of the destination or terminal node of the edge

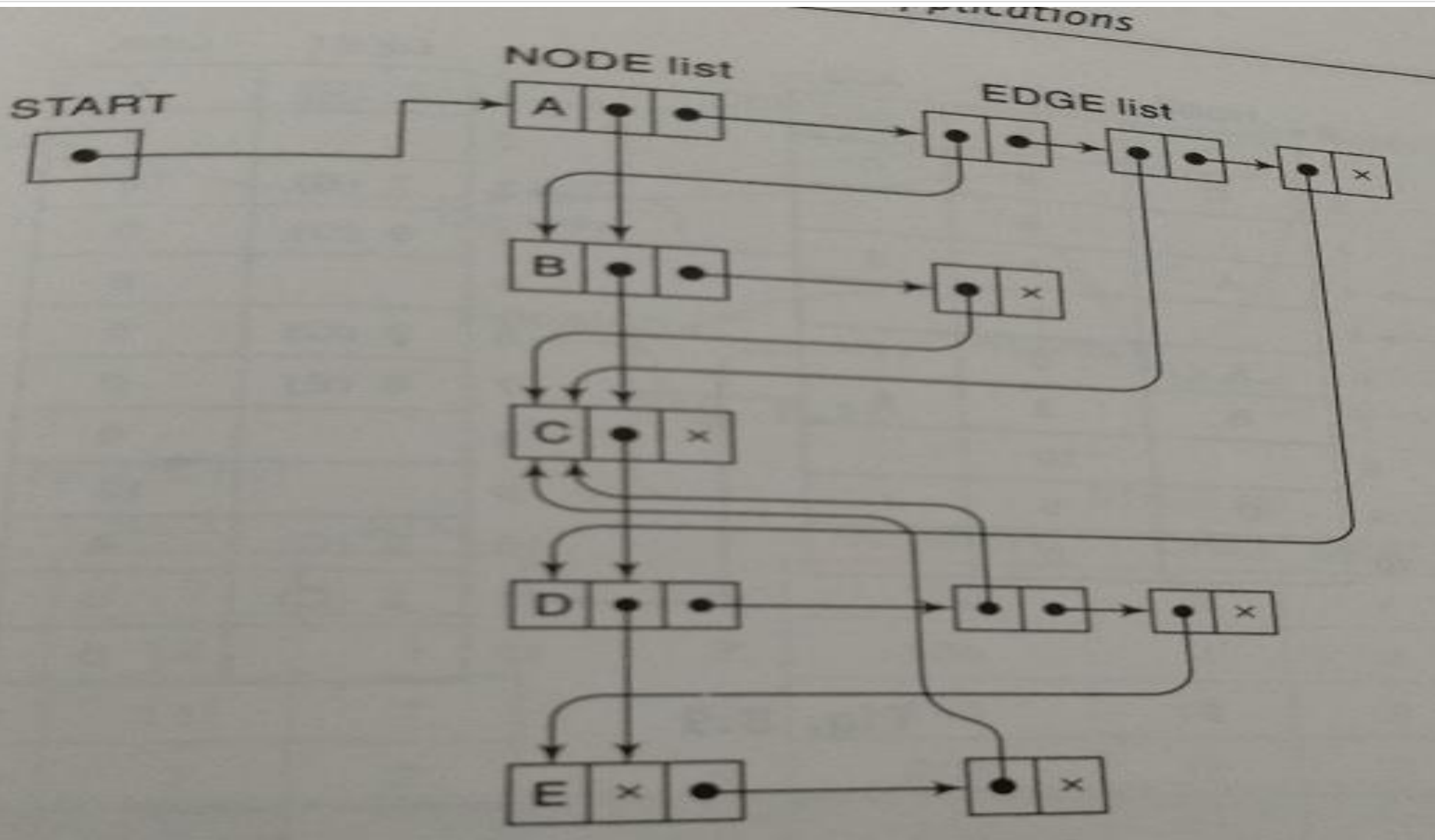
**LINK**-> Link together the edges with the same initial node i.e. the nodes in the same adjacency list.

The shaded area indicates the other information corresponding to edge, **WEIGHT** etc.

---



# Schematic Diagram of Linked Representation of G in Memory



# Example

Suppose Friendly Airways has nine daily flights, as follows:

103 Atlanta to Houston

106 Houston to Atlanta

201 Boston to Chicago

203 Boston to Denver

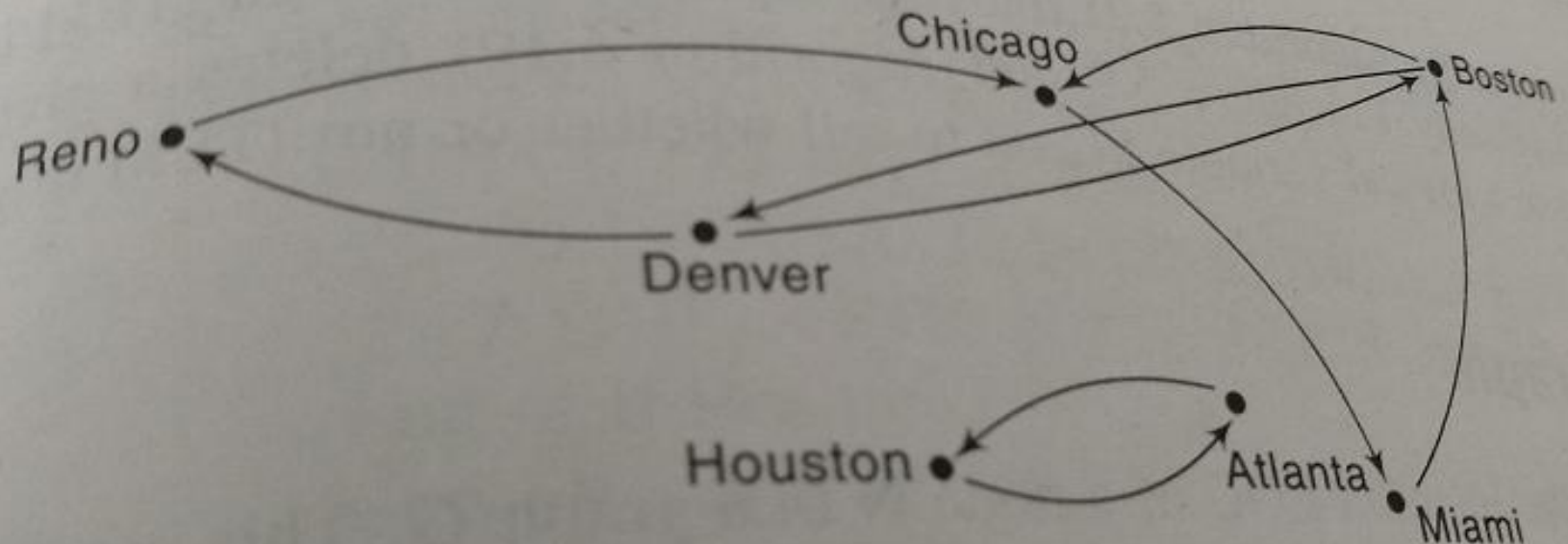
204 Denver to Boston

301 Denver to Reno

305 Chicago to Miami

308 Miami to Boston

402 Reno to Chicago



# Operations on Graphs

---

## 1. Searching in a Graph

### (a) Find LOC of node N in a graph G

Procedure 8.3: FIND(INFO, LINK START, ITEM, LOC) [Algorithm 5.2]  
Finds the location LOC of the first node containing ITEM, or sets  
LOC := NULL.

1. Set PTR := START.
2. Repeat while PTR  $\neq$  NULL:  
    If ITEM = INFO[PTR], then: Set LOC := PTR, and Return.  
    Else: Set PTR := LINK[PTR].  
    [End of loop.]
3. Set LOC := NULL, and Return.

# Operations on Graphs

---

## 1. Searching in a Graph

### (b) Find LOC of of an edge(A,B) in a graph G

**FINDEDGE(NODE, NEXT, ADJ, START, DEST, LINK, A, B, LOC)**  
This procedure finds the location LOC of an edge (A, B) in the graph G, or sets  $LOC := NULL$ .

1. Call **FIND(NODE, NEXT, START, A, LOCA)**.
2. CALL **FIND(NODE, NEXT, START, B, LOCB)**.
3. If  $LOCA = NULL$  or  $LOCB = NULL$ , then: Set  $LOC := NULL$ .  
Else: Call **FIND(DEST, LINK, ADJ[LOCA], LOCB, LOC)**.
4. Return.



# Operations on Graphs

---

## 2. Inserting in a Graph

### (a) Inserting a node in G

Procedure 8.6: `INSNODE(NODE, NEXT, ADJ, START, AVAILN, N, FLAG)`  
This procedure inserts the node N in the graph G.

1. [OVERFLOW?] If `AVAILN = NULL`, then: Set `FLAG := FALSE`, and Return.
2. Set `ADJ[AVAILN] := NULL`.
3. [Removes node from AVAILN list]  
Set `NEW := AVAILN` and `AVAILN := NEXT[AVAILN]`.
4. [Inserts node N in the NODE list.]  
Set `NODE[NEW] := N`, `NEXT[NEW] := START` and `START := NEW`.
5. Set `FLAG := TRUE`, and Return.

# Operations on Graphs

---

## 2. Inserting in a Graph

### (b) Inserting a edge in G

Procedure 8.7: INSEGE(NODE, NEXT, ADJ, START, DEST, LINK, AVAILE, A, B, FLAG)  
This procedure inserts the edge (A, B) in the graph G.

1. Call FIND(NODE, NEXT, START, A, LOCA).
2. Call FIND(NODE, NEXT, START, B, LOCB).
3. [OVERFLOW?] If AVAILE = NULL, then: Set FLAG := FALSE, and Return.
4. [Remove node from AVAILE list.] Set NEW := AVAILE and AVAILE := LINK[AVAILE].
5. [Insert LOCB in list of successors of A.]  
Set DEST[NEW] := LOCB, LINK[NEW] := ADJ[LOCA] and  
ADJ[LOCA] := NEW.
6. Set FLAG := TRUE, and Return.

# Example Problem

8.13 A graph  $G$  is stored in memory as follows:

NODE	A	B		E		D	C	
NEXT	7	4	0	6	8	0	2	3
ADJ	1	2		5		7	9	
	1	2	3	4	5	6	7	8

START = 1, AVAILN = 5

DEST	2	6	4		6	7	4		4	6
LINK	10	3	6	0	0	0	0	4	0	0
	1	2	3	4	5	6	7	8	9	10

AVAILE = 8

Draw the graph  $G$ .



# Example Solution

First find the neighbors of each  $\text{NODE}[K]$  by traversing its adjacency list, which has the pointer  $\text{ADJ}[K]$ . This yields:

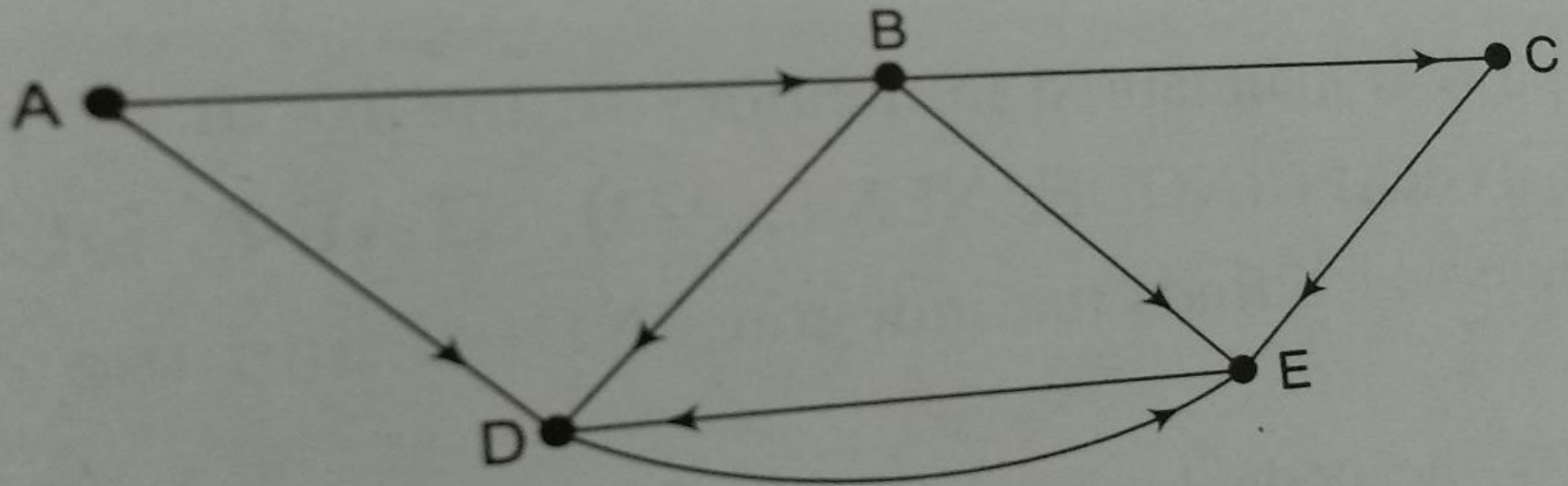
A: 2(B) and 6(D)

C: 4(E)

E: 6(D)

B: 6(D), 4(E) and 7(C)

D: 4(E)





# Traversing a Graph

---

During the execution of the algorithm, each node  $N$  of  $G$  will be in one of the three states called status of  $N$  as follows:

- **STATUS=1(Ready State)**  
It is the initial state of the node  $N$ .
  - **STATUS=2(Waiting State)**  
The node  $N$  is on the queue or stack waiting to be processed.
  - **STATUS=3(Processed State)**  
The node  $N$  has been processed.
-

# Traversing a Graph

---

## 1. Breadth-First-Search

- First examine the starting node A
- Then examine all the neighbors of A
- Then examine all the neighbors of neighbors of A and so on.

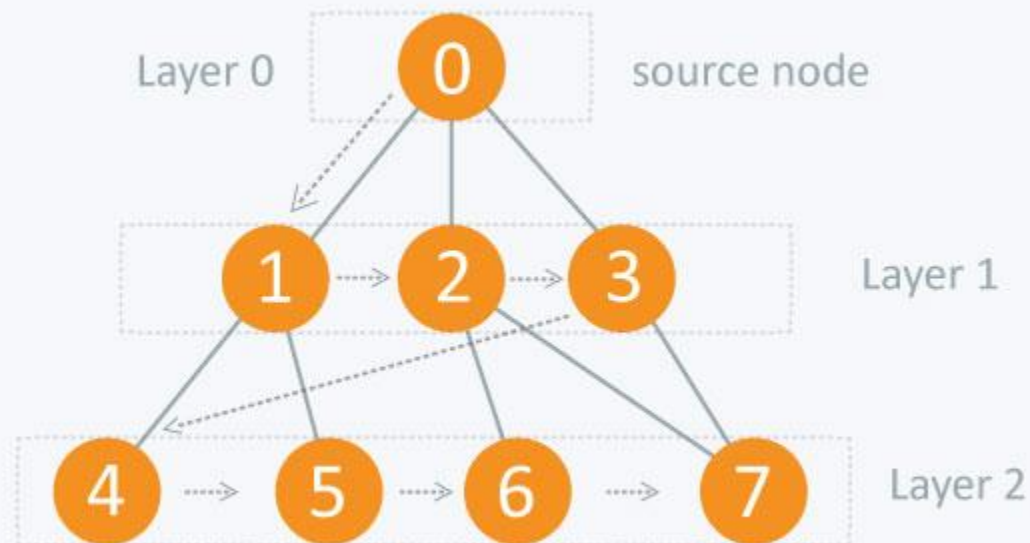
**NOTE:** No node is processed more than once

---

# Traversing a Graph

---

## 1. Breadth-First-Search



# Traversing a Graph

---

- **Breadth-First-Search Algorithm**  
(This algorithm use a Queue to hold nodes)
  1. Initialize all nodes to the ready state(**STATUS=1**)
  2. Put the starting node A in QUEUE and change its status to the waiting state(**STATUS=2**)
  3. Repeat steps 4 and 5 until QUEUE is empty

# Traversing a Graph

---

- **Breadth-First-Search Algorithm**

4. Remove the front node N of QUEUE. Process N and change the status of N to STATUS=3.

5. Add to the rear of QUEUE all the neighbors of N that are in STATUS=1 and change their status to STATUS=2

[End of Step 3 Loop]

6. Exit

---

# Traversing a Graph

---

## 2. Depth-First-Search Algorithm

(This algorithm use a Stack to hold nodes)

1. Initialize all nodes to the ready state(**STATUS=1**)
  2. Push the starting node A in **STACK** and change its status to the waiting state(**STATUS=2**)
  3. Repeat steps 4 and 5 until **STACK** is empty
-

# Traversing a Graph

---

- **Depth-First-Search Algorithm**

**4. Pop the top node N of STACK. Process N and change the status of N to STATUS=3.**

**5. Push onto STACK all the neighbors of N that are in STATUS=1 and change their status to STATUS=2**

**[End of Step 3 Loop]**

**6. Exit**

---

# Traversing a Graph

---

- **Depth-First-Search Algorithm**

It employs the following rules.

**Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

**Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

**Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

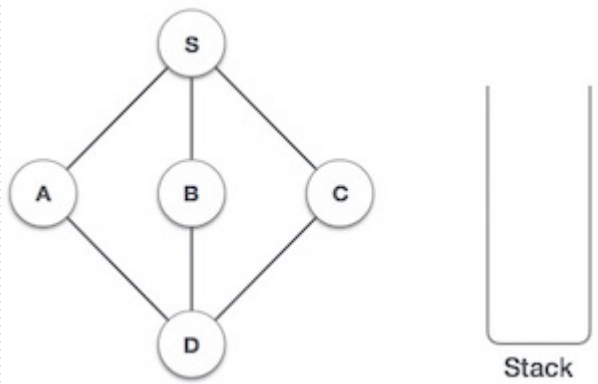
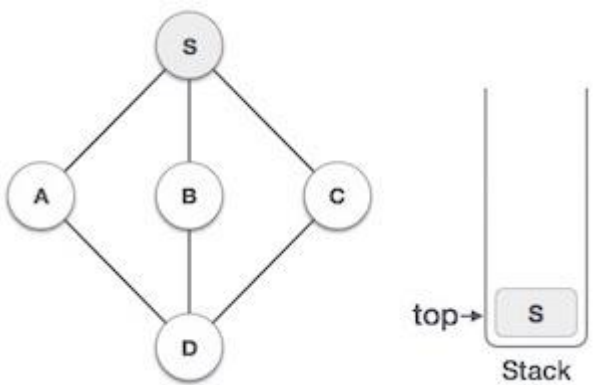
---



# Depth-First Search

## Example

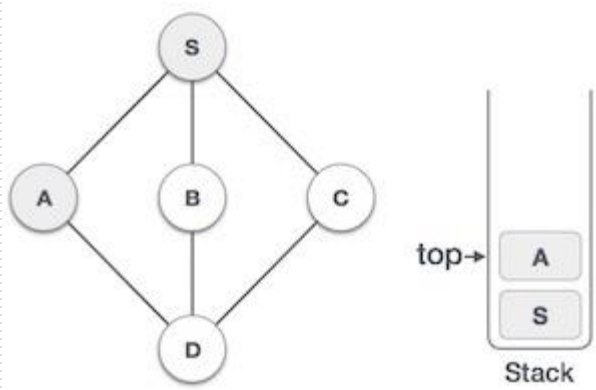
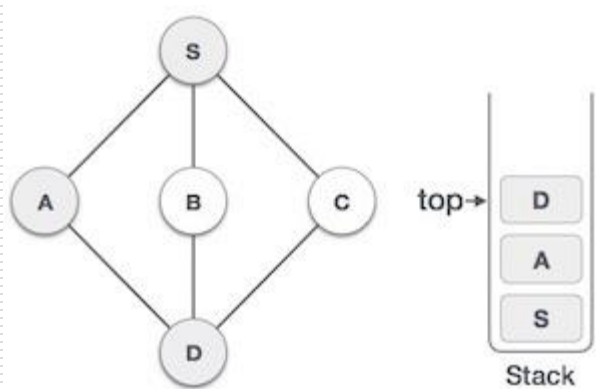
# Search

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

# Depth-First Search

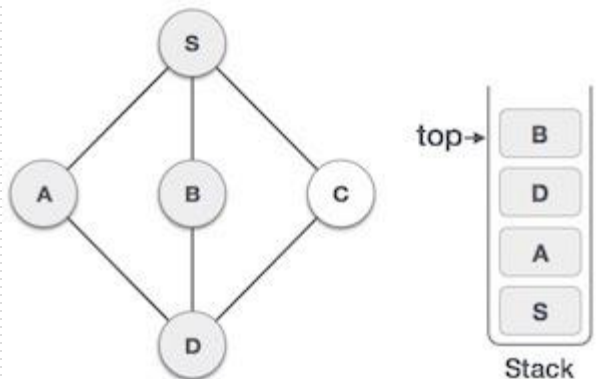
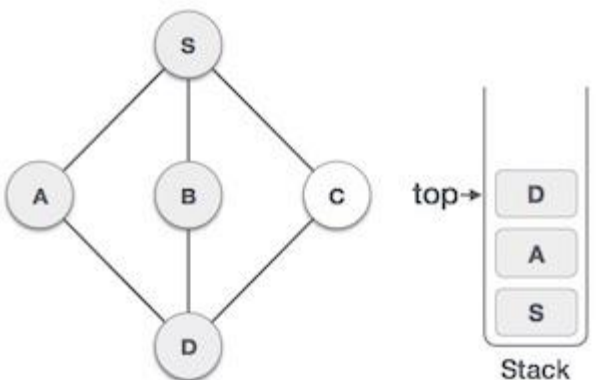
## Example

# Search

Step	Traversal	Description
3		Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.
4		Visit <b>D</b> and mark it as visited and put onto the stack. Here, we have <b>B</b> and <b>C</b> nodes, which are adjacent to <b>D</b> and both are unvisited. However, we shall again choose in an alphabetical order.

# Depth-First Example

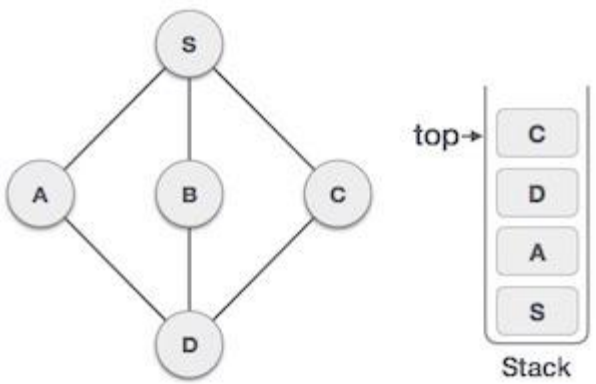
# Search

Step	Traversal	Description
5		We choose <b>B</b> , mark it as visited and put onto the stack. Here <b>B</b> does not have any unvisited adjacent node. So, we pop <b>B</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 <b>D</b> to be on the top of the stack.

# Depth-First Search

## Example

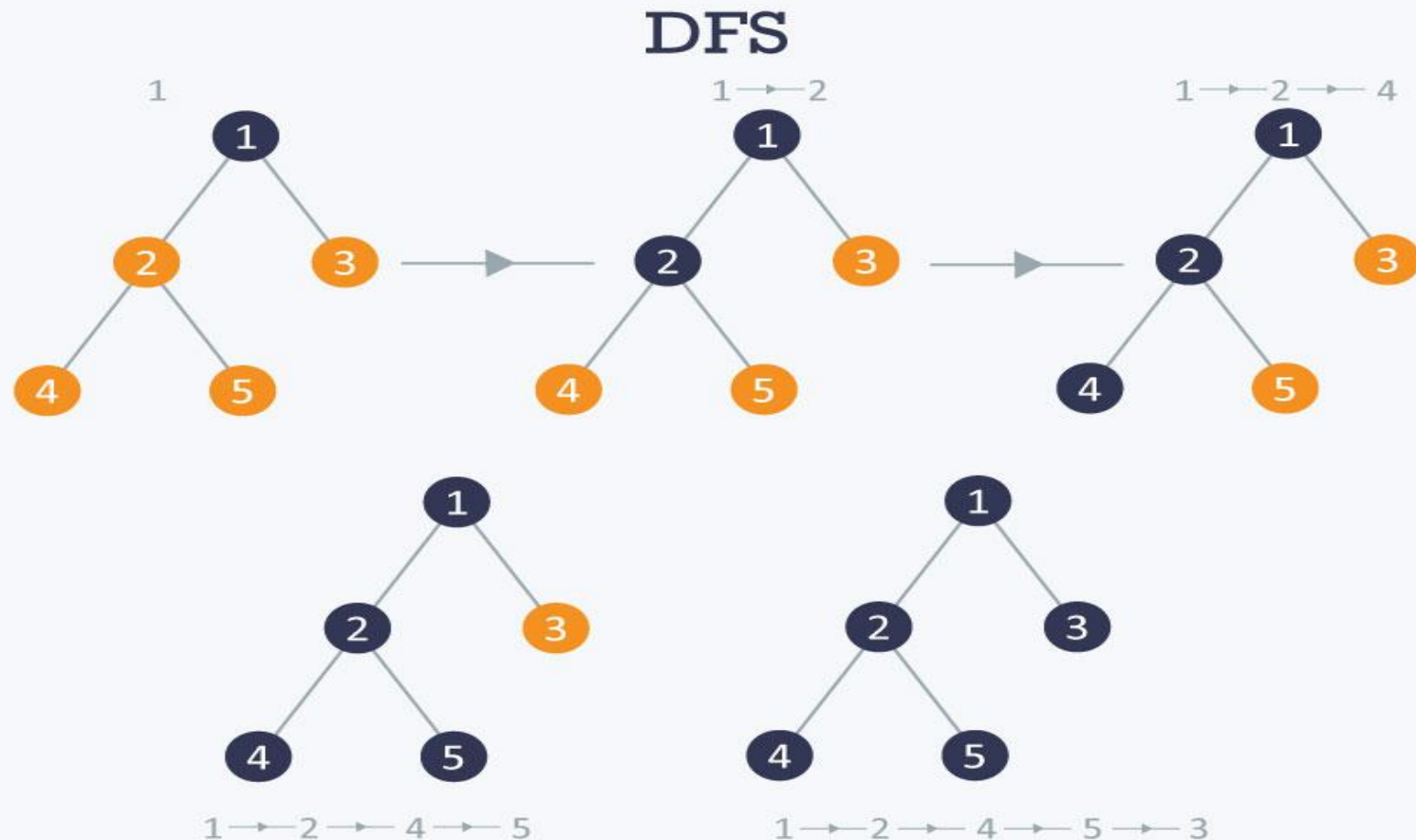
# Search

Step	Traversal	Description
7		Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b> , mark it as visited and put it onto the stack.

**NOTE:** 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.

# Traversing a Graph

- Depth-First-Search Algorithm



# Hashing

---

- It is a technique used for performing almost constant time search in case of insertion, deletion and find operation.
  - Hash tables are used to implement the concept of hashing.
-

# Hashing

---

- Hash tables efficiently implement the keyed array data structure also known as associative array or map.
  - In each keyed array, each element is associated with a key. A key is used to find an element instead of an index no.. Therefore, Hash table is one basic form of keyed array.
-

# Hashing

---

- Since Hash table has to be coded using an indexed array, there has to be some way of transforming a key to an index number. That way is called the Hashing function.
-



# Hashing Functions

---

- The hashing function should return a value based on a key and the size of the array, the hashing table is built on.
  - Function which helps us in generating the value corresponding to key is known as hashing function.
-

# Hashing Functions

---

- For eg: Suppose we want to organize a list of about 260 addresses by people's last names. In this case, people's last names are used as keys in hash table.
  - Create a hashing function but before that create a relationship between letters and numbers.
  - Assign  $A \rightarrow 0, B \rightarrow 1, \dots, Z \rightarrow 25$
-

# Hashing Functions

---

- Now organize the hash table based on first letter of the last name.
- As we have 260 elements, we can multiply the first letter by the last name by 10.
- So when a key like “Smith” is given, the key would be transformed to  $\text{index}(\text{S} \rightarrow 18 \text{ and } 18 * 10 = 180)$  180.
- This index no. is used to access an element directly, a hash table access time is small.

# **Collision Resolution in Hashing Functions**

---

- Problem may arise when we have last names with the same first letter.
  - The situation when two keys sent to the same location in the array is known as collision. To address this problem, following are the two main collision resolving techniques:
    - (1) Open Hashing/ Separate Chaining
    - (2) Closed Hashing/ Open Addressing
-

# **Collision Resolution in Hashing Functions**

---

## **(1) Open Hashing/Separate Chaining:**

**In this strategy, collision is resolved by keeping the conflicting elements in a list i.e. to keep all elements in a list which generate same hash.**

**A linked list is stored at each element in the hash data structure. When a collision occur, an element can be added into the linked list i.e. stored at the hash index.**

---

# Collision Resolution in Hashing Functions

---

## (2) Closed Hashing/Open Addressing:

In this strategy, collision is resolved by placing the conflicting element near to the slot generated by the hash function.

The **disadvantage** of closed hashing is that it consumes more space as compared to open hashing also it has less flexibility in accommodating for duplicate hash element.

---

# Collision Resolution in Hashing Functions

---

**(2) Closed Hashing/Open Addressing:**

The **advantage** of closed hashing is that it reduces the overhead of introducing new data structure and reduces cost of new memory allocation per new element insertion.

---