UNIT - III: GRAPHS

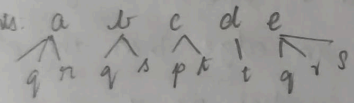
- Tree have → parent child relationship
- Graph have no ↗
- Graph - collection of nodes & edges

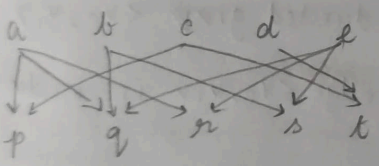
Ex:

1. There are m no. of applicants applying for n no. of post where there is a possibility of having more than one applicant suitable for job & more than one post can also suitable for an applicant. This scenario is suitable for graph

Posts: p q r s t

Applicants: a b c d e





⊙ a - p
b - q
c - t
d - (-)
e - r

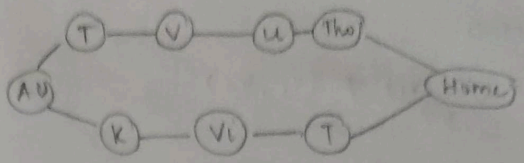
d - jobless
s - vacant

2. Analysis for minimal job

d - t
c - p
a - q
b - s
e - r

- Mapping are done by group

3. Identifying the best route

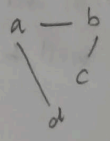


- water distribution
- Network
- Transport

GRAPH:

- A graph (G) can be represented by G = (V, E) where V is set of vertices, E is set of edges connecting the vertices in V.
- E is represented (U, V)
  U & V are vertices.
  ⇒ e(U, V)

⊙ G



⇒ G = (V, E)

V = {a, b, c, d}

E = {(a, b), (b, c), (a, d)}

undirected graph:

→ undirected graph is one in which pair of vertices in edge is unordered i.e.,

$(V_0, V_1) = = (V_1, V_0)$

UNIT - III: GRAPHS

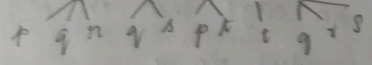
- Tree have → parent child relationship
- Graph have no ↗
- Graph - collection of nodes & edges

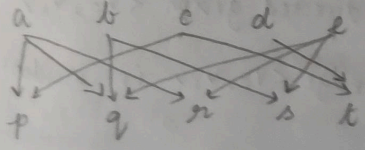
Ex:

1. There are m no. of applicants applying for n no. of post where there is a possibility of having more than one applicant suitable for job & more than one post can also suitable for an applicant. This scenario is suitable for graph

Posts: p q r s t

Applicants: a b c d e





⊙ a - p
b - q
c - t
d - (-)
e - r

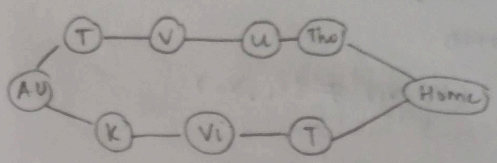
d - jobless
s - vacant

2. Analysis for minimal job

d - t
c - p
a - q
b - s
e - r

- Mapping are done by group

3. Identifying the best route

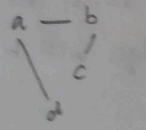


- ★ water distribution
- ★ Network
- ★ Transport

GRAPH:

- A graph (G) can be represented by G = (V, E) where V is set of vertices, E is set of edges connecting the vertices in V.
- E is represented (U, V)
  U & V are vertices.
  ⇒ e(U, V)

⊙ G



⇒ G = (V, E)

V = {a, b, c, d}

E = {(a, b), (b, c), (a, d)}

undirected graph:

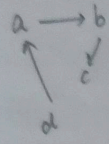
→ undirected graph is one in which pair of vertices in edge is unordered i.e.,

$(V_0, V_1) == (V_1, V_0)$

Directed graph:

→ Directed graph is one in which each edge has a ordered pair of vertices.

$(V_0, V_1) \neq (V_1, V_0)$



$E = \{(a,b), (b,c), (d,a)\}$

Complete graph:

→ A complete graph is graph that has a maximum no. of edges for undirected graph with n vertices & $n(n-1)/2$ edges.

⊙ UNDIRECTED GRAPH

$|V| = n$ &

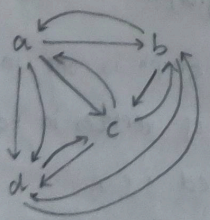
$|E| = n(n-1)/2$ edges

An undirected graph is called as complete graph if no. of vertices is n & no. of edges will be $n(n-1)/2$

⊙ DIRECTED GRAPH

$|V| = n$ &

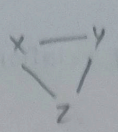
$|E| = n(n-1)$ edges



$n = 4$

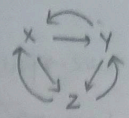
$n(n-1) = 12$ edges

(*)



$V = \{x, y, z\}$

$E = \{(x,y)(x,z)(y,z)\}$



$V = \{x, y, z\}$

$E = \{(x,y)(y,x)(x,z)(z,x)(y,z)(z,y)\}$

Adjacent:

→ If (V_0, V_1) is an edge in an directed graph then V_0 & V_1 are adjacent.

→ Directed: $\langle V_0, V_1 \rangle$

Undirected: (V_0, V_1)

Incident:

→ In an undirected graph (V_0, V_1) is an edge then the edge (V_0, V_1) is incident on V_1

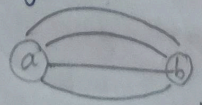
→ In directed graph $\langle V_0, V_1 \rangle$, the edge $\langle V_0, V_1 \rangle$ is incident on V_1

self loop: (feedback, edge)

→ It is special single edges that starts & ends at same vertices.

Multiple edge:

→ It is very much possible for two vertices ~~for~~ to have more than one edge. such edge is known as Multiple edge otherwise simple edge.



→ Magnetic Field.

↳ Multiple edge

(*) Based on multiple edge

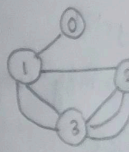
Graph — simple graph
— Multigraph
— pseudograph

Multiplicity of graph

— The no. of multipl
two given vertex
edge's multiplicity

— (0, 1, 2, 3) is

(0,1) (1,2) (1,3) (2



Multiplicity of ed

Edge: (0,

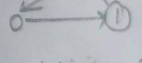
(1,

(1

(

Multiplicity of gr

(*) directed graph



Multiplicity :

Subgraph:

G' is the subgraph

$G' = (V', E')$

$V(G') \subseteq V(G)$

$E(G') \subseteq E(G)$

(i) Null graph (ii)

(iii) (2) (iv)

Based on multiple edges,

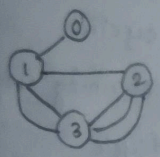
- Graph
  - Simple graph — Has no multiple edges & loops
  - Multigraph — Has multiple edges but no loops
  - Pseudograph — Has both multiple edges & loops

Multiplicity of graph:

- The no. of multiple edges b/w two given vertex is called that edge's multiplicity

- (0, 1, 2, 3) is an undirected graph

(0,1) (1,2) (1,3) (2,3) (1,3) (2,3) (2,3)

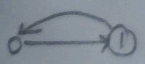


Multiplicity of edge:

Edge: (0,1) → 1
(1,2) → 1
(1,3) → 2
(2,3) → 3

Multiplicity of graph is 3

② directed graph



Multiplicity : 2

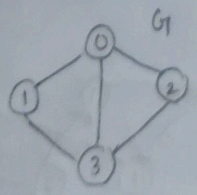
Subgraph:

G' is the subgraph.

$G' = (V', E')$

$V(G') \subseteq V(G)$

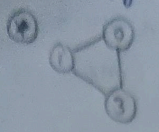
$E(G') \subseteq E(G)$



(i) Null graph (ii) ⓪

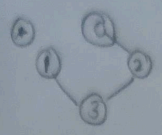
(iii) ② (iv) ⓪——②

?: Identify subgraph for given graph G[G']



$V' = \{0, 1, 3\}$

$E' = \{(0,1)(1,3)(0,3)\}$

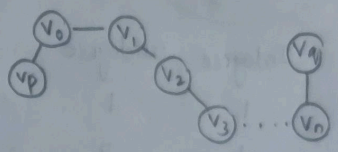


$V' = \{0, 2, 3, 1\}$

$E' = \{(0,2)(2,3)(3,1)\}$

Path: (walk)

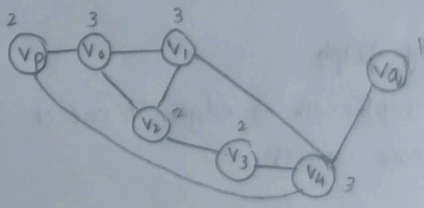
→ A path from vertices v_p to v_q in graph G is a set vertices b/w v_p & v_q, such that $(v_p, v_0), (v_0, v_1), (v_1, v_2) \ldots$ $\ldots (v_n, v_q)$ in undirected graph.



Path $(v_p, v_q) = (v_p, v_0, v_1, v_2, \ldots v_n, v_q)$

→ LENGTH OF PATH:

• No. of edges on a path is known as length of a path.



$P(v_1, v_4) = (v_1, v_4)$ Length = 1

$P(v_1, v_4) = (v_1, v_2, v_3, v_4)$ Length = 3

→ SIMPLE PATH:

It is a path in which all vertices ~~all~~ except possibly the first & last are distinct

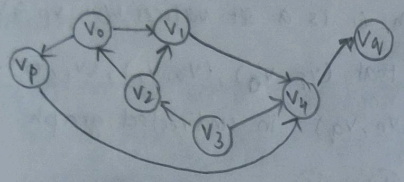
→ CYCLIC PATH: (closed path)

It is simple path in which first & last vertex are same.

Degree:

- Indegree
- outdegree

In directed graph, indegree of vertex e is no. of edges that have v as its head.



	Indegree	outdegree	
V_p	1	1	= 2
V_0	1	2	= 3
V_1	2	1	= 3
V_2	1	2	= 3
V_3	–	2	= 2
V_4	3	1	= 4
V_q	1	–	= 1

Empty graph:

– Graph with no edges & one or more vertices.

Null graph:

– No edges, No vertex

Infinite & Finite graph:

– The graph with infinitely many vertices or edges or both is referred to as an infinite graph otherwise it is finife graph.

- A path without any repeated vertices is called simple path.
- A ~~graph~~ path with distinct edges is referred to as trail.
- A closed trail is known as Tour or circuit.

⇒ Based on cyclic nature:

* A graph with no cycle will be a Acyclic graph
* we can call tree as graph when it is cyclic
* A graph containing single cyclic is called unicycle.

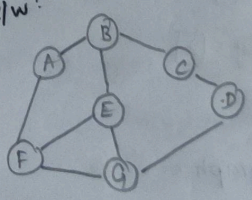
Hamiltonian graph (spanning tree)

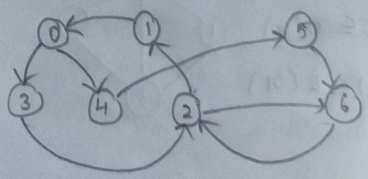
– that uses all vertices exactly once

Eulerian graph

– If a cycle uses all of its edges exactly once then it is known Eulerian.

H/w:





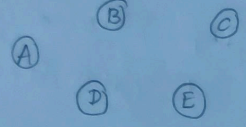
28/11/23

1. Draw a undirected complete graphs with 5 nodes

V(G): A, B, C, D, E

E(G): (A,B) (A,C) (B,E) (B,D) (C,B) (C,D) (D,E) (D,A) (E,C) (E,A) - 10 Edges

2. Draw an empty graphs with 5 vertices



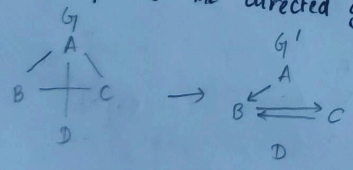
Null graph - Nothing / Empty

3. Convert the given undirected graph to a directed graph where G=(V,E)

V(G) is (A, B, C, D)

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

Let G' be the directed graph



4. Can we call a binary tree as graph? Justify.

Based on

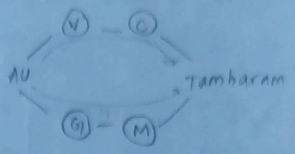
there are 2 graphs:

i) weighted

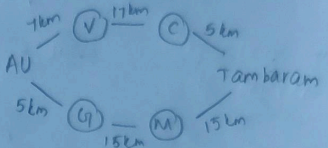
ii) unweighted (Till now unweighted)

Since edge

unweighted P/I



weighted:



weigh here is distance.

sometimes - Traffic, fuel, road quality

Implementation of Graph:

i) Array - 2D → Because graph is not hierarchical

ii) Linked list

↓

vertices

edges: (u, v) tuples

HW:

How 2D Array is in physical memory

2D Array

int A[2][3] (R, C)

int B[]

29/11/23

Graph Repres

{ i) Adjacency
ii) Adjacency
iii) Multilist

• 2D Array:

No. of r, c →

→ n vertic

⇒ adj - m

eg: adj_mat

⇒ Empty graph

	A
A	0
B	0
C	0

⇒

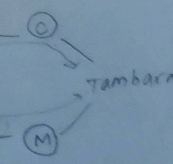
	A
A	0
B	1
C	1

Q: You have a
4 nodes - A, B,
undirected. w
matrix:

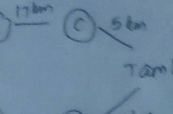
	A	B
A	0	1
B	1	0
C	1	1
D	1	1

• undirected -

ed 0/1



:



here is distance.

- Traffic, fuel, road quality

tion of Graph:

y - 2D →

ed l

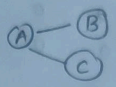
chical

physical memory

2/11/23

Graph Representation:

{ i) Adjacency Matrix - Array
ii) Adjacency List } Linked List
iii) Multilist



- 2D Array:

No. of r, c → Based on vertices

→ n vertices

⇒ adj-mat[n][n]

eg: adj_mat[3][3]

⇒ Empty graph with 3 vertices

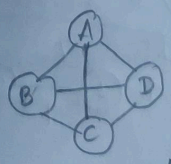
undirected, unweighted graph

	A	B	C
A	0	0	0
B	0	0	0
C	0	0	0

⇒

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

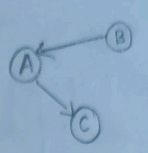
Q: You have a complete graph with 4 nodes - A, B, C, D which is undirected. write the adjacency matrix:



A-A = 0

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

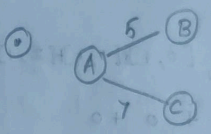
- undirected - symmetric in nature



directed unweighted graph

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

- directed - Not symmetric in nature



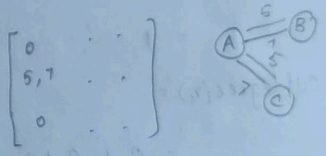
undirected weighted graph

	A	B	C
A	0	5	7
B	5	0	0
C	7	0	0

If multigraph → edge - diff weights

↓

create structure for the weights.



Find a way a find indegree.

- $\text{Indegree}(V_i) = \sum_{j=0}^{n-1} A[j, i]$

i = row, j = column

↓ sum of column

- $\text{outdegree}(V_i) = \sum_{j=0}^{n-1} A[i, j]$

↓ sum of rows.

	0 A	1 B	2 C
0 A	0	0	1
1 B	1	0	0
2 C	0	0	0

↓ i

(i ← j) Indegree
(i → j) outdegree

Indegree $(V_0) = \sum_{j=0}^{2} [0,0] + [1,0] + [2,0]$

$= 0 + 1 + 0$

Indegree $(V_A) = 1$

Indegree $(V_1) = \sum_{j=0}^{2} [0,1] + [1,1] + [2,1]$

$= 1 + 0 + 0$

Indegree $(V_B) = 1$

Indegree $(V_2) = \sum_{j=0}^{2} [0,2] + [1,2] + [2,2]$

$= 0 + 0 + 0$

Indegree $(V_C) = 0$

Outdegree $(V_0) = \sum_{j=0}^{2} [0,0] + [0,1]$

$= 0 + 1 + 0$

$= 1$

Outdegree $(V_1) =$

$= 0 + 0 + 0$

$= 0$

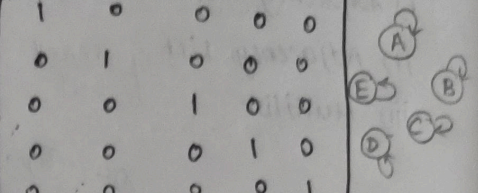
Outdegree $(V_2) =$

$= 1 + 0 + 0$

$= 1$

Empty graph with 5 vertices:
You've 5 × 5 matrix..

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



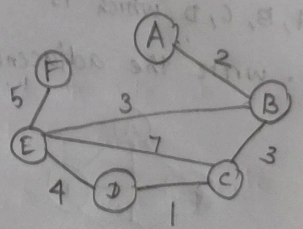
→ This is self-loop graph
(self-referential)

You've 6 rows & 6 columns : weighted graph. Draw the graph.

	A	B	C	D	E	F
A	0	2	0	0	0	0
B	2	0	3	0	3	0
C	0	3	0	1	7	0
D	0	0	1	0	4	0
E	0	3	7	4	0	5
F	0	0	0	0	5	0

undirected (symmetric)

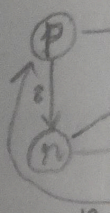
(A,B) (B,A) (B,C) (B,E)
(C,B) (C,D) (C,E)
(D,C) (D,E)
(E,B) (E,C) (E,D) (E,F)
(F,E)



HW:
Think about how to insert a vertex, edges
what kind of modification required
in

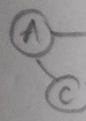
16/12/23

Represent
Adjacency

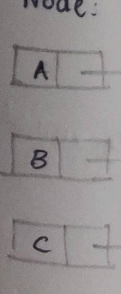


	p
p	0
q	0
r	0
s	10
t	0

Represent th
Adjacency
→ Linked
→ For ea
information
linked lis

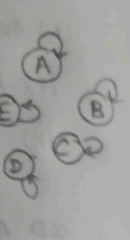


Node:



with 5 vertices:
matrix..

C	D	E
0	0	0
0	0	0
1	0	0
0	1	0
0	0	1



self-loop graph
(self-referential)

ws & columns : weighted
the graph.

	D	E	F
0	0	0	0
3	0	3	0
0	1	7	0
1	0	4	0

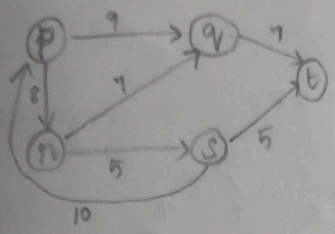
E)

D)(E,F)

ow to
insert a vertex, edges
f modification required

15/12/23

Represent the graph using Adjacency matrix:

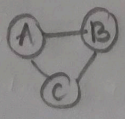


	p	q	r	s	t
p	0	9	8	0	0
q	0	0	0	0	7
r	0	7	0	5	0
s	10	0	0	0	5
t	0	0	0	0	0

Represent the graph using adjacency list:

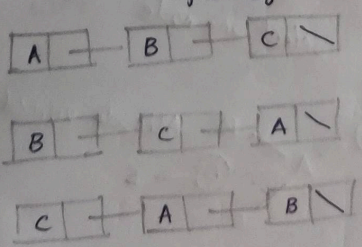
→ Linked list representation

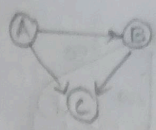
→ For each node, its neighbour information is maintained in a linked list.



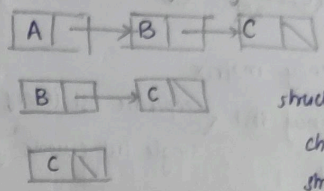
unweighted graph
(self loop also possible)

Node: Adjacency list:



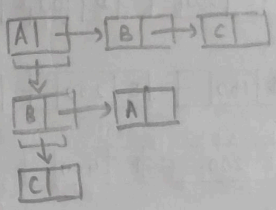


Node Adjacency list




```
struct alist {
    char label;
    struct alist *node;
    int weight;
};
```

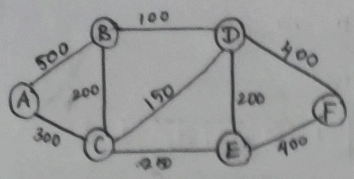
p
q
r
s
t



Label

Data	anode
nextnode	

```
struct glist
{
    char label;
    struct glist *anode;
    //int weight;
    struct glist *nearvertices;
};
```

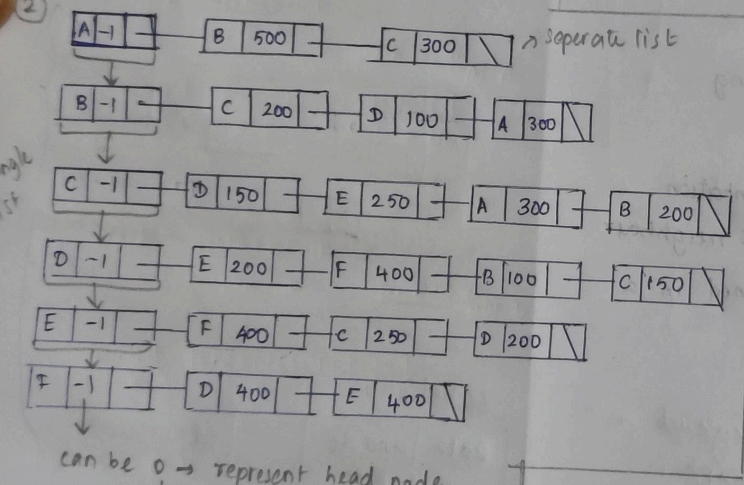


1) Adjacency matrix
2) Adjacency list ← separate list (every node)
single list (whole graph)

① Adjacency matrix

	A	B	C	D	E	F
A	0	500	300	0	0	0
B	500	0	200	100	0	0
C	300	200	0	150	250	0
D	0	100	150	0	200	400
E	0	0	250	200	0	400
F	0	0	0	400	400	0

② Adjacency list



↗ separate list

single list

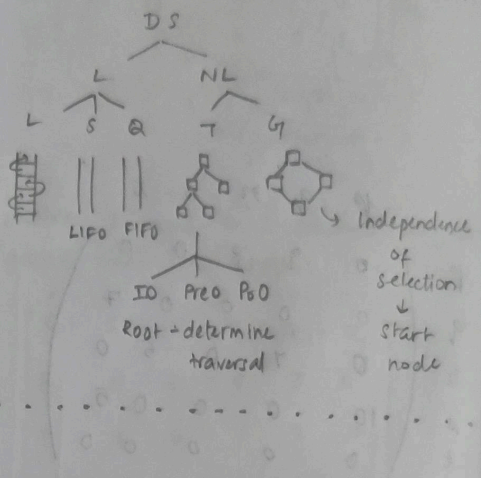
can be 0 → represent head node
↳ like a flag

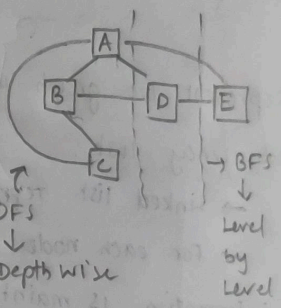
comparison of Adj Matrix & Adj List:

ASPECT	ADJ MATRIX	ADJ LIST
• Memory representation	$O\|V^2\|$	$O(\|V\| \to \|E\|)$
• Suitability	Dense	Sparse
• Insertion & Deletion	Difficult	Easier

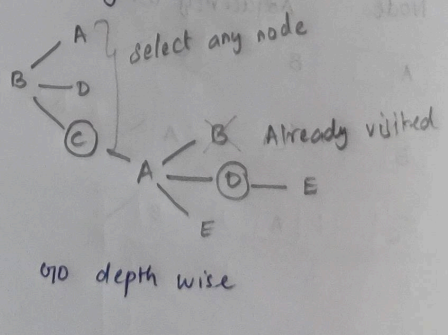
21/12/23

Graph Traversal:
(i) Depth first search
(ii) Breadth first search



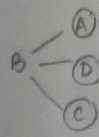


DFS:
Based on adj list visit nodes randomly first



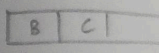
O/P depth wise

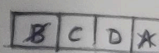
BFS:
complete visiting le



visit all adjacent node of B first

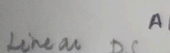
DFS:
start at any node
Input from use
Maintain list





Adj

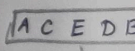
Linear DS



Adj of

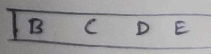
Stack - Element in
recently

visited list



↓
Simple li

Linear DS



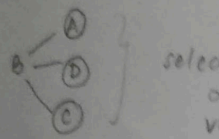
LIFO

DFS ☑ suitable DS →
Depth wise → La
ins
↓
Stack

visited list → List

BFS:

complete visiting level by level



select any one or visit their adjacent

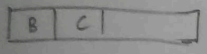
visit all adjacent node of B first

DFS:

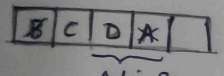
start at any node

Input from user: A (visited list)

Maintain list of Adj of A



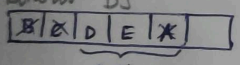
AB → can visit like AC



Adj B

ABC

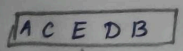
Linear DS



Adj of C

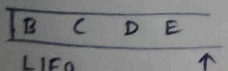
Stack - Element inserted recently accessed

① visited list



Simple list DS

Linear DS



DFS ② suitable DS → stack

Depth wise → last inserted (top)

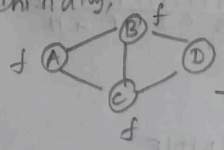
Stack

③ visited list → list

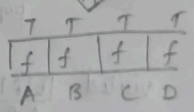
H/W: write Algorithm for given pseudocode

```
DFS(G, v)   (v-starting vertex input from user)
Stack S={}
for each vertex u in G,
        set visited[u] = false
Push S,v        make every node as not visited node
while (S is not empty) do
    u = pop S
    if (not visited[u]) then
        visited[u] = true
        for each unvisited neighbour w of u
            Push S,w    Adj of u
    end if
end while
END DFS()
```

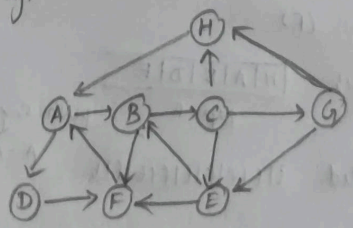
④ Initially,



→ Making visited as True



⊙ eg:



A : B D
B : C F
C : G E H
D : F
E : F B
F : A
G : E H
H : A

Step - 1:

Graph, V
(G, H)

stack: [ ] // Empty

visited: [f|f|f|f|f|f|f|f] // 8 - Size
A - H

Step - 2:

stack: [H]

visited: [f|f|f|f|f|f|f|f]

Step - 3: (H) - visited

stack: [H|A]

pop H

push s, w ↦ Adj of H
↓
A

visited [f|f|f|f|f|f|f|t]
A B C D E F G H

Step - 4: (A)

stack [H|A|B|D]

pop A

push s, w ↦ of A

visited: [t|f|f|f|f|f|f|t]
A B C D E F G H

Step - 5: (D)

stack [H|A|B|D|E]

pop D

push s, w ↦ of D

visited [t|f|f|t|f|f|f|t]
A B C D E F G H

Step - 6: (F)

stack [H|A|B|D|F]

pop F

push s, w
↓
A - Already visited

visited t|f|f|f|t|f|f|t|f|t

Step - 7: (B)

stack: ↑