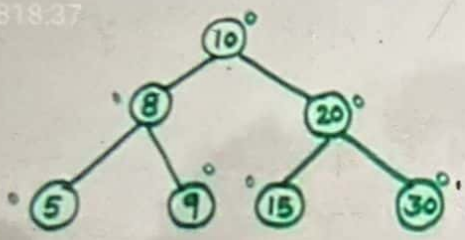
AVL TREE [SELF BALANCED BST]

10, 20, 8, 15, 5, 9, 30



$\min_h = \log n$

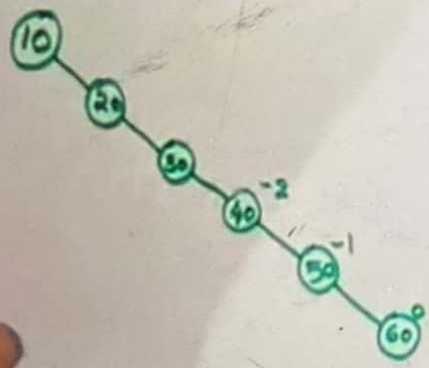
$\max_h = n$

- Balance factor → BF
  - → height of left subtree − height of right subtree
  - → {−1, 0, 1}

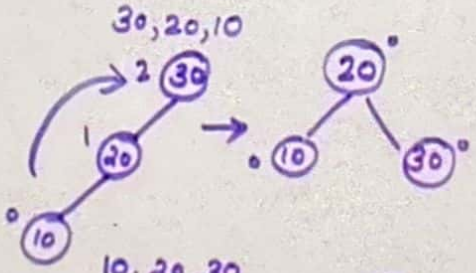
BF → $h_l - h_r = \{-1, 0, 1\}$

BF → or → $|h_l - h_r| \leq 1$

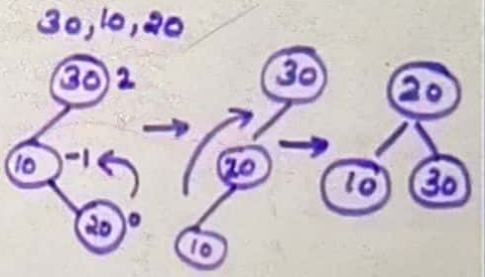
10, 20, 30, 40, 50, 60



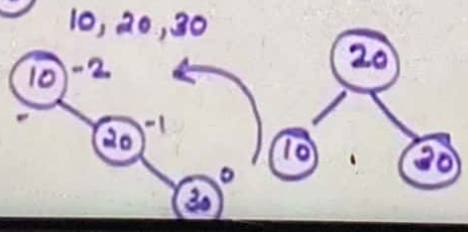
30, 20, 10



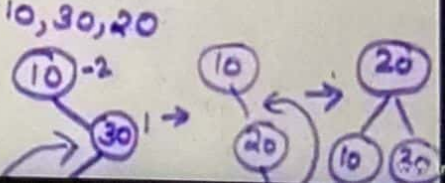
30, 10, 20



10, 20, 30



10, 30, 20

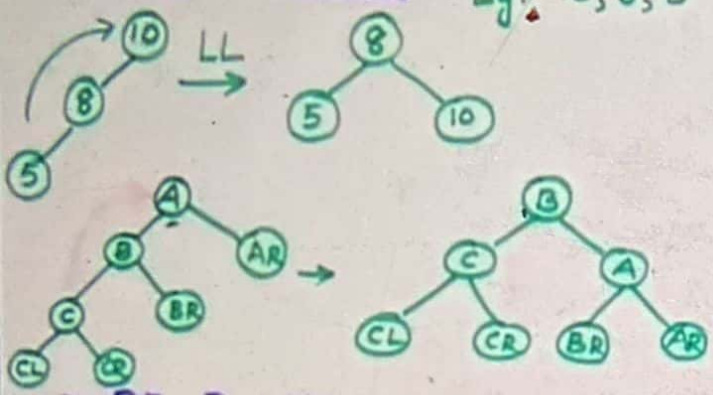


Scanned with CamScanner

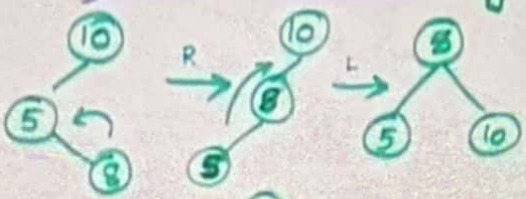
AVL Tree Rotations

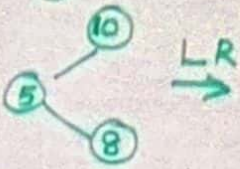
Single → LL, RR
Double → LR, RL

1) LL Rotation - Eg:- 10, 8, 5

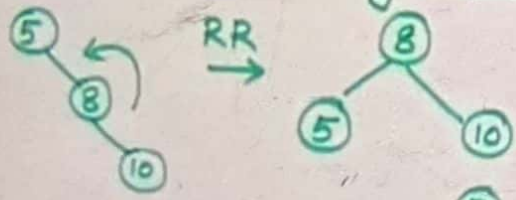


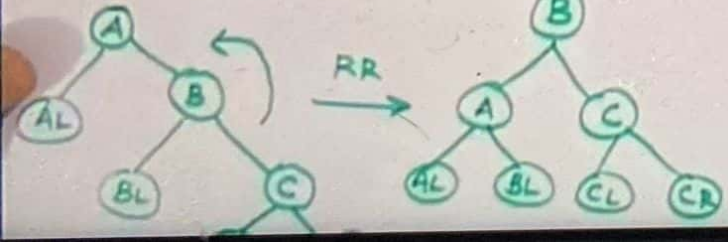
2) LR Rotation Eg:- 10, 5, 8



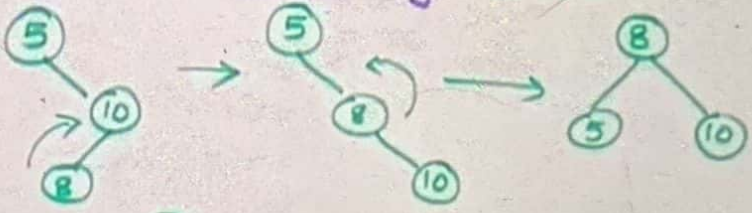


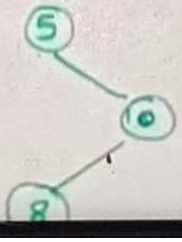
3) RR Rotation - Eg:- 5, 8, 10





4) RL Rotation Eg:- 5, 10, 8

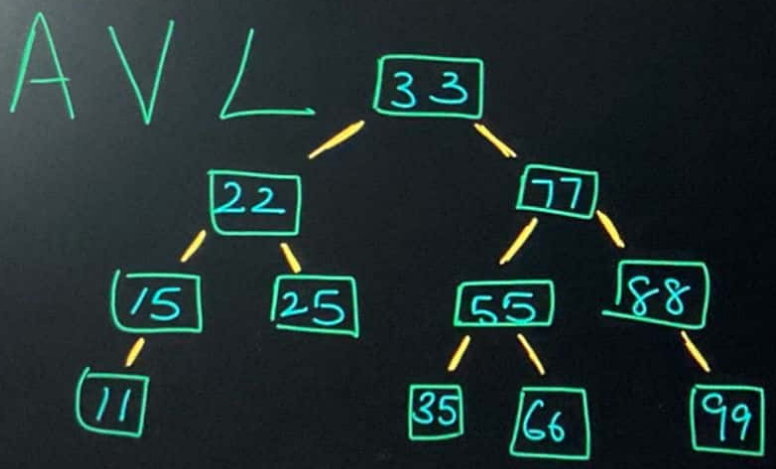




Scanned with CamScanner

Q) Delete the nodes in following order -

88, 99, 22, 33, 11, 15, 35, 66, 99, 77, 55



B·F = H(LST) − H(RST) {−1, 0, 1}

→ 3 different use cases for deletion -

1) For leaf node(n) - Delete the node(n)

→ 2) For node(n) with 1 child -
Link the parent node parent(n) with this child node child(n) & delete the node(n) in between

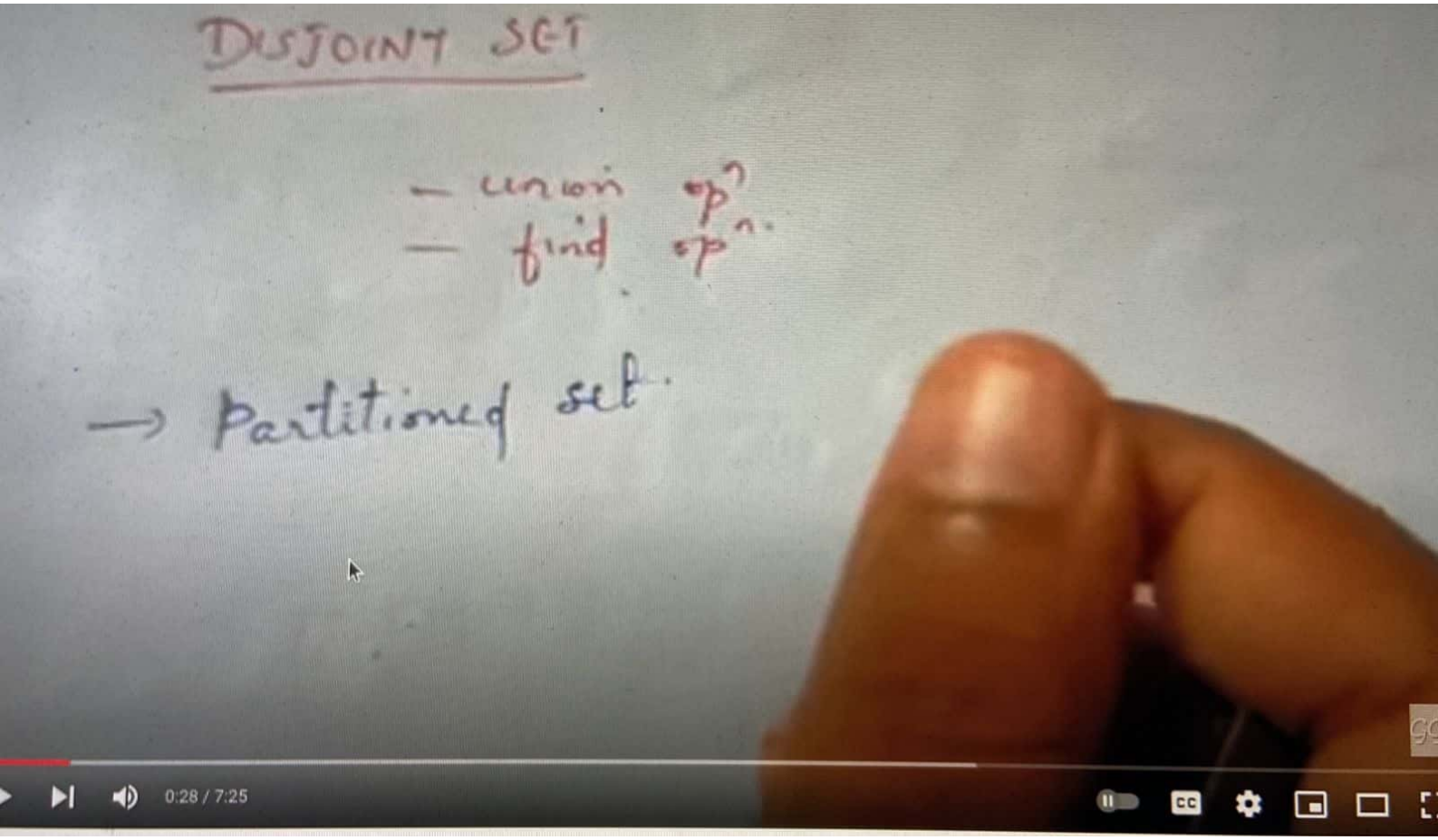
3) For node with 2 children -

a) Find the largest node(nMax) in left subtree.
Replace this node with the node to be deleted.
Delete nMax.
OR

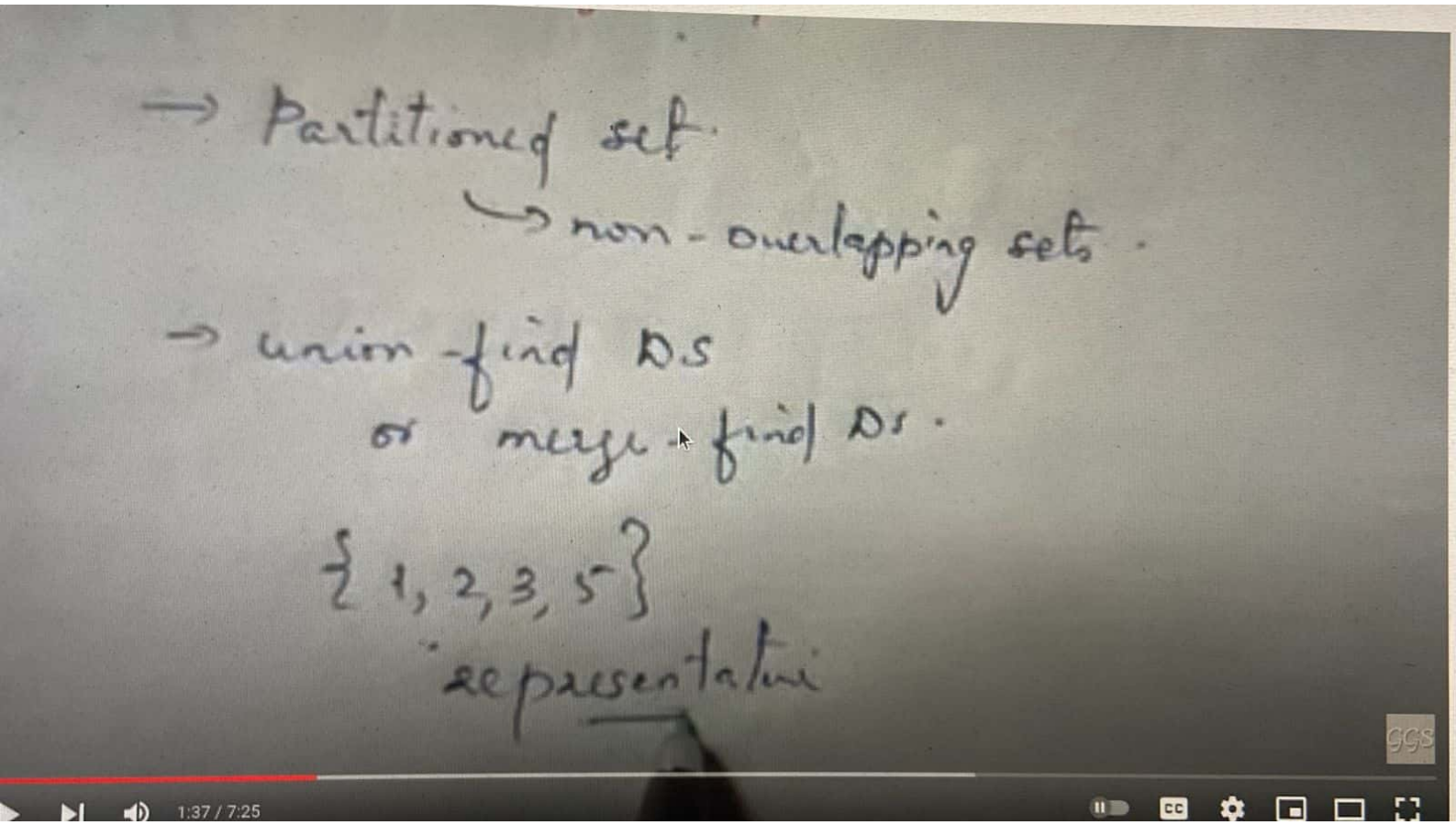
b) Find the smallest node(nMin) in right subtree.
Replace this node with the node to be deleted.
Delete nMin.

SIMPLE ... 3:07 / 26:47 • BST deletion use cases ›

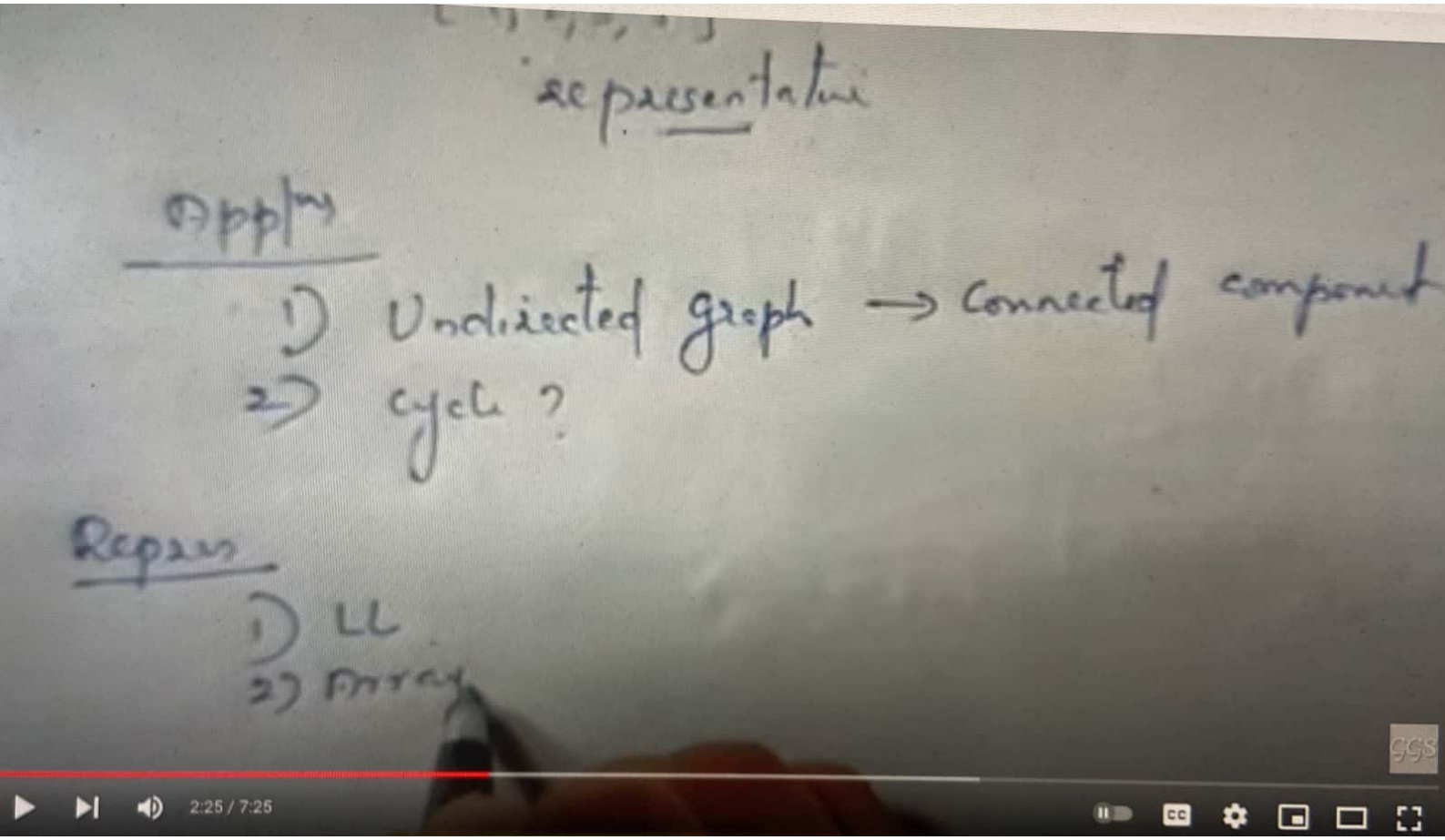
Scanned with CamScanner



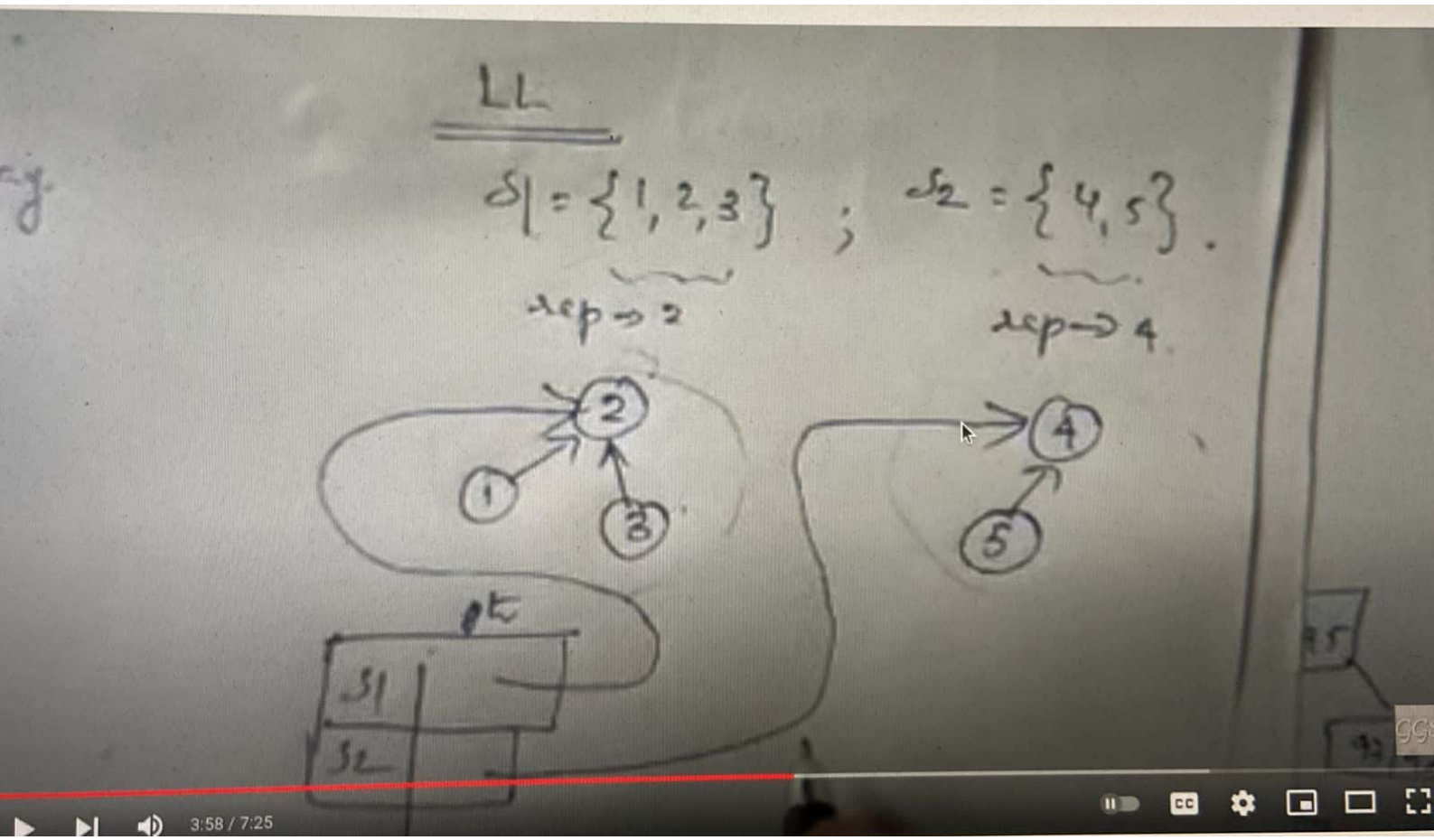
Scanned with CamScanner



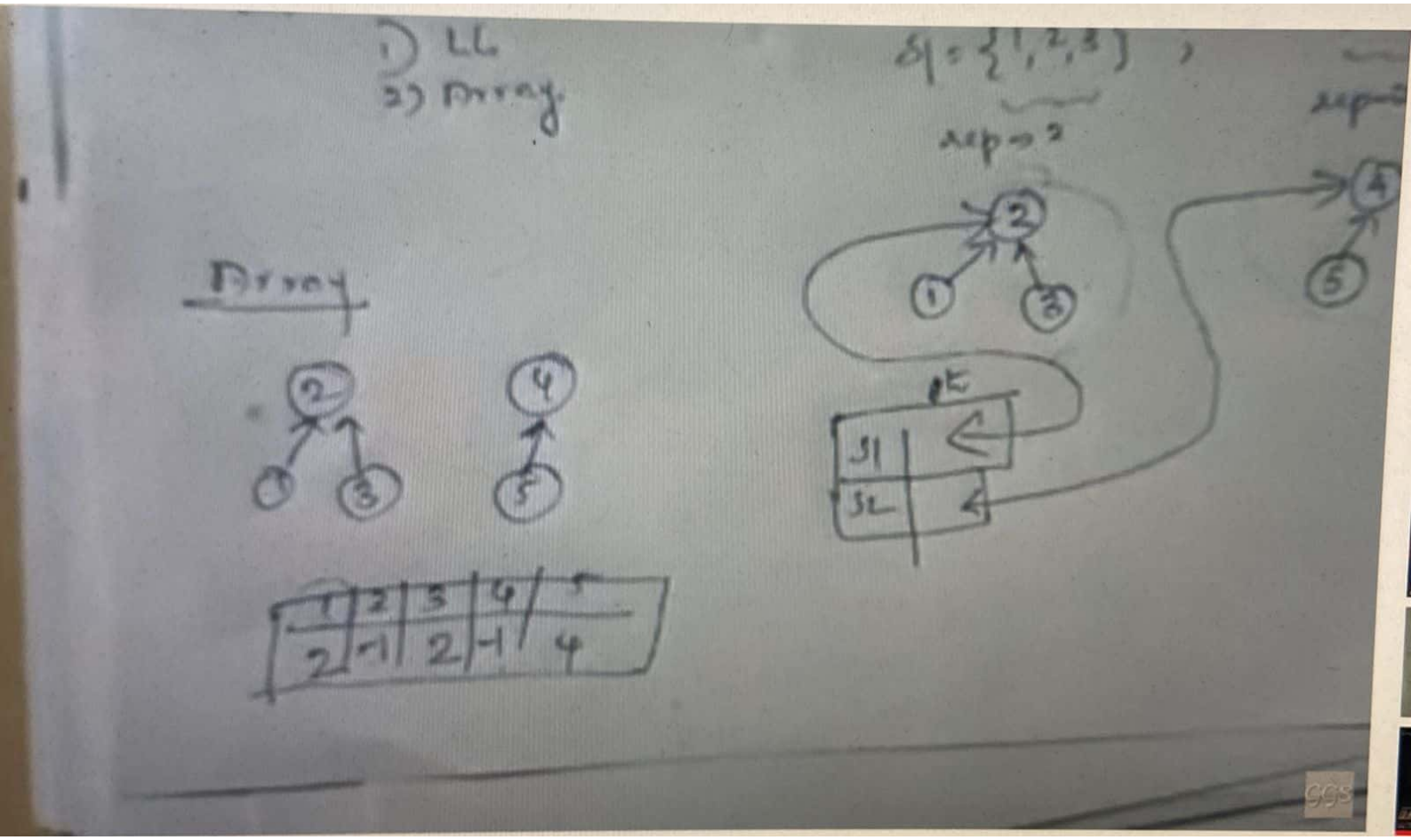
Scanned with CamScanner



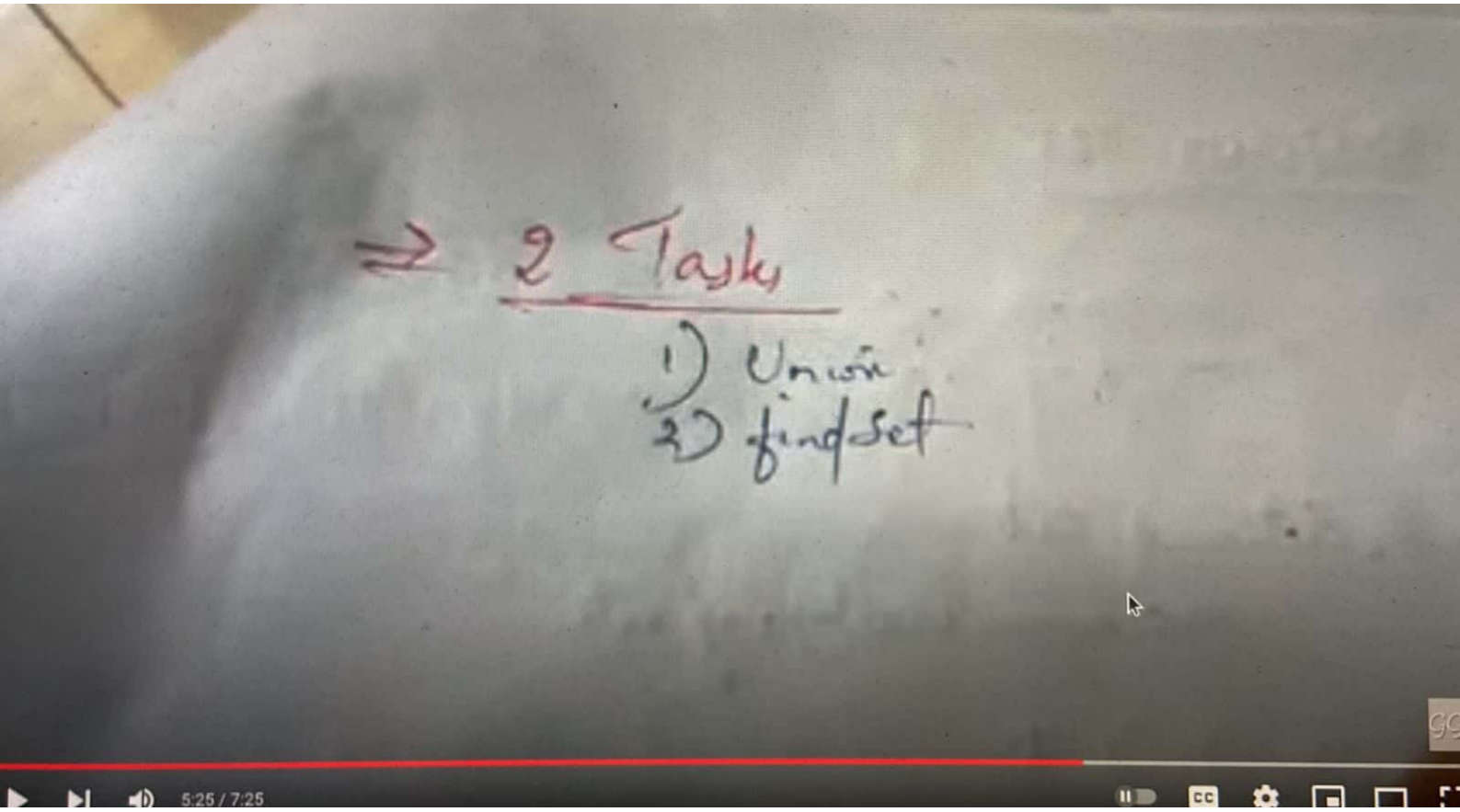
Scanned with CamScanner



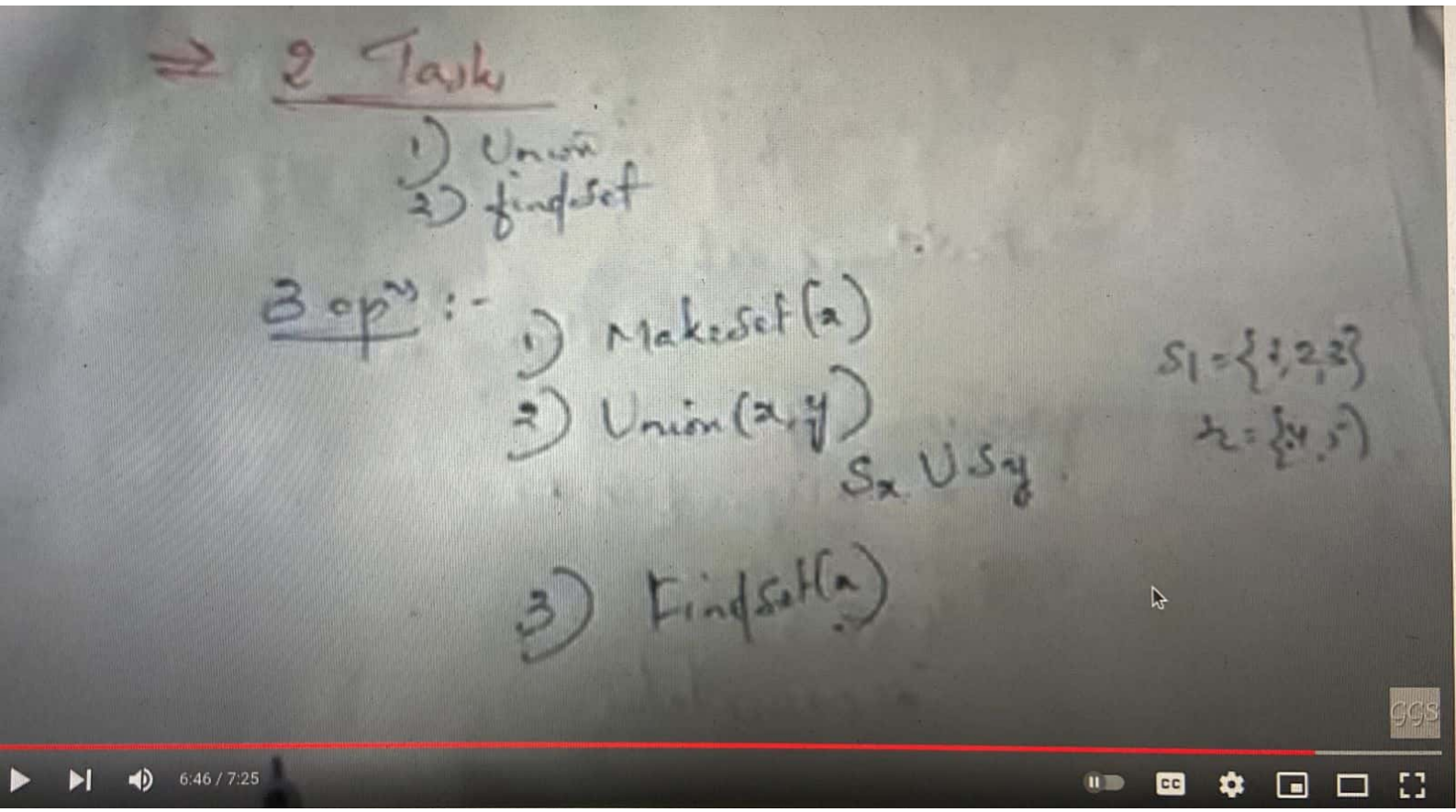
Scanned with CamScanner



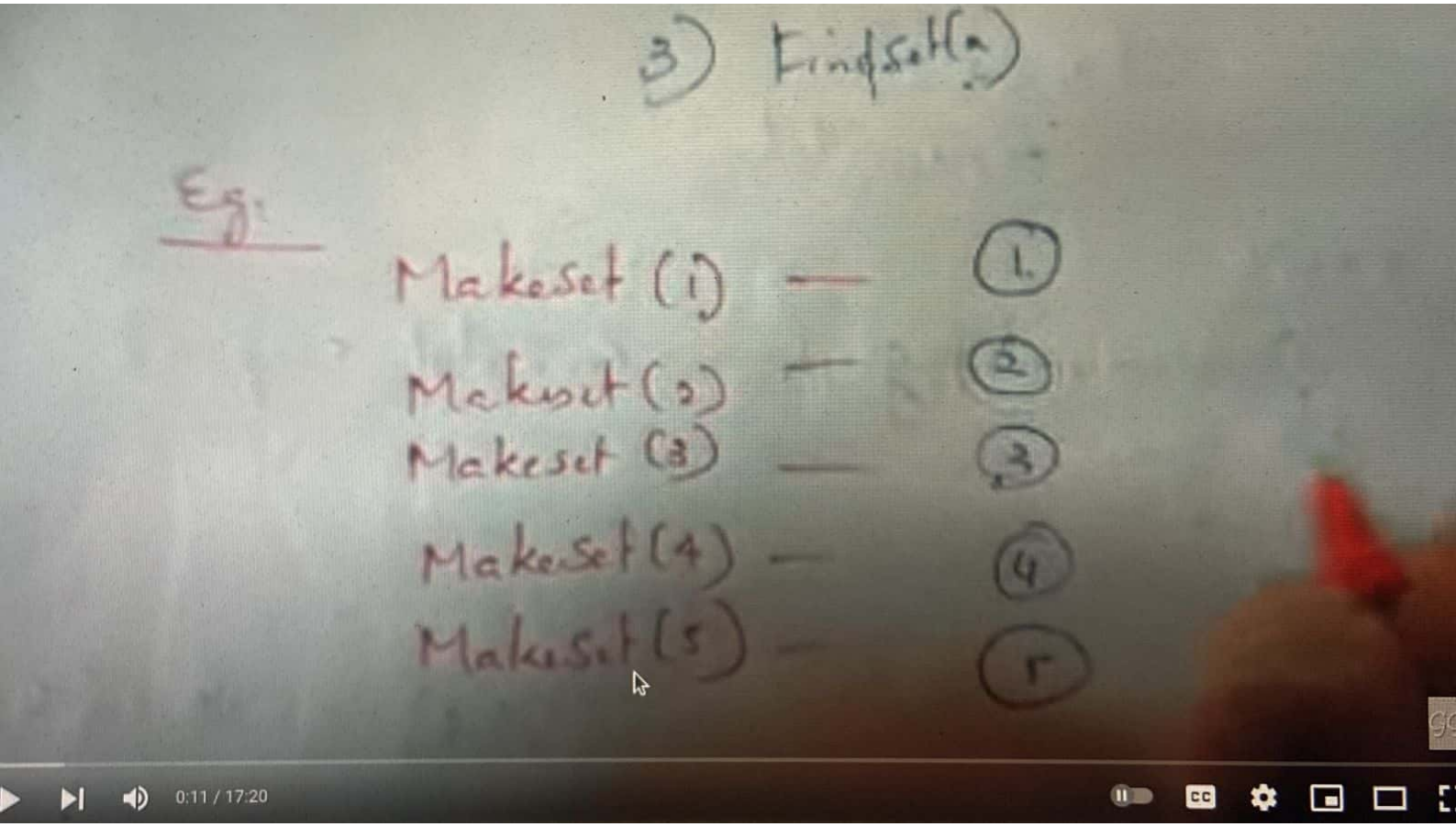
Scanned with CamScanner



Scanned with CamScanner



Scanned with CamScanner



Scanned with CamScanner

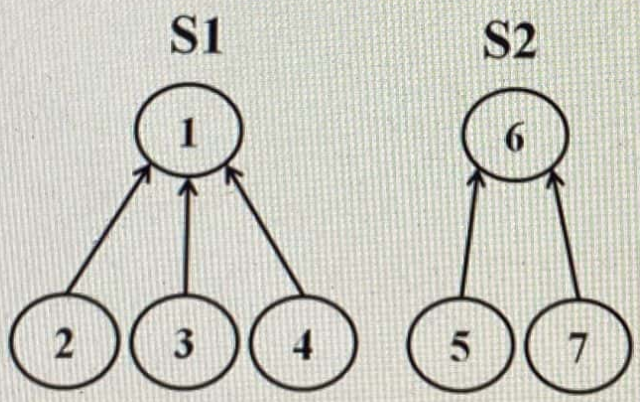
Disjoint Set operations

- **Make set**
  - Make-set(x) function will create a set that containing one element x

4:05 / 9:01
CS KTU Lectures
Scanned with CamScanner

Disjoint Set operations

- **Find**
  - Determine which subset a particular element is in.
  - This can be used for determining if two elements are in the same subset.
  - Example:



- Find(3) will return 1, which is the root of the tree that 3 belongs
- Find(6) will return 6, which is the root of the tree that 6 belongs

Scanned with CamScanner

Disjoint Set operations

- **Find Algorithm**

Algorithm Find(i)

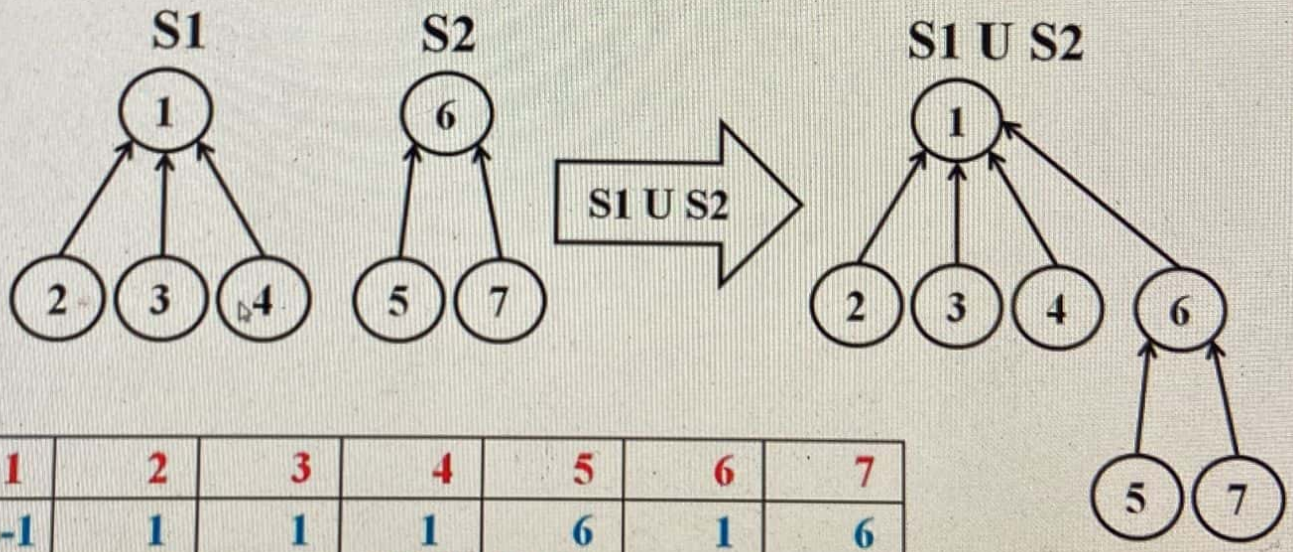
```
{
    while p[i] > 0 do
        i = p[i]
    return i
}
```

5:31 / 9:01

Scanned with CamScanner

Disjoint Set operations

- **Union**
  - Join two subsets into a single subset.
  - Here first we have to check if the two subsets belong to same set. If no, then we cannot perform union



i	1	2	3	4	5	6	7
p	-1	1	1	1	6	1	6

CS KTU Lectures

Scanned with CamScanner

Disjoint Set operations

- **Union Algorithm**

Algorithm Union(i, j)

```
{
    X =Find(i)
    Y = Find(j)
    If X != Y then
        p[Y] = X
}
```


Scanned with CamScanner

Disjoint Set Applications

- It is easier to check whether the given two elements are belongs to the same set.
- Used to merge 2 sets into one

8:48 / 9:01
CS KTU Lectures

Scanned with CamScanner

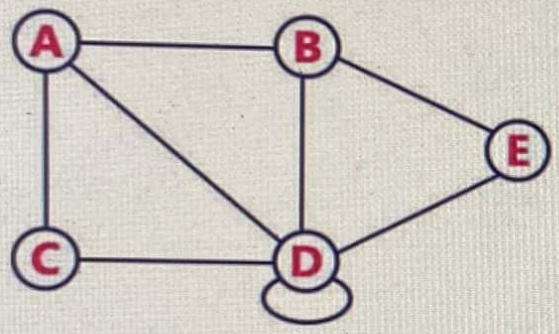
Graphs

- A graph **G = (V , E)**
  - V → Set of vertices
  - E → Set of edges.
- Representations of graph
  - Adjacency Matrix
  - Adjacency List

0:09 / 5:34

Scanned with CamScanner

Adjacency Matrix

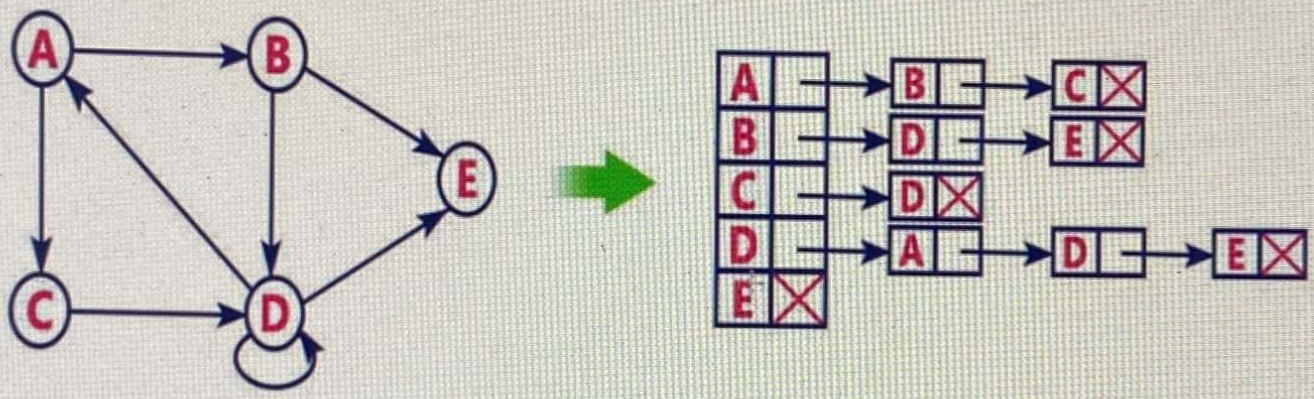


	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

- It is a 2D array(say adj[][]) of size |V|x|V| where |V| is the number of vertices in a graph.
  - If adj[i][j] = 1, then there is an edge from vertex i to vertex j.
  - For a weighted graph if adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w
- Adjacency matrix for undirected graph is always symmetric.

Scanned with CamScanner

Adjacency List



- An array of linked lists is used.
- Size of the array is equal to number of vertices.
- An entry array[i] represents the linked list of vertices adjacent to the $\mathbf{i}^{th}$ vertex.
- This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists.

Scanned with CamScanner

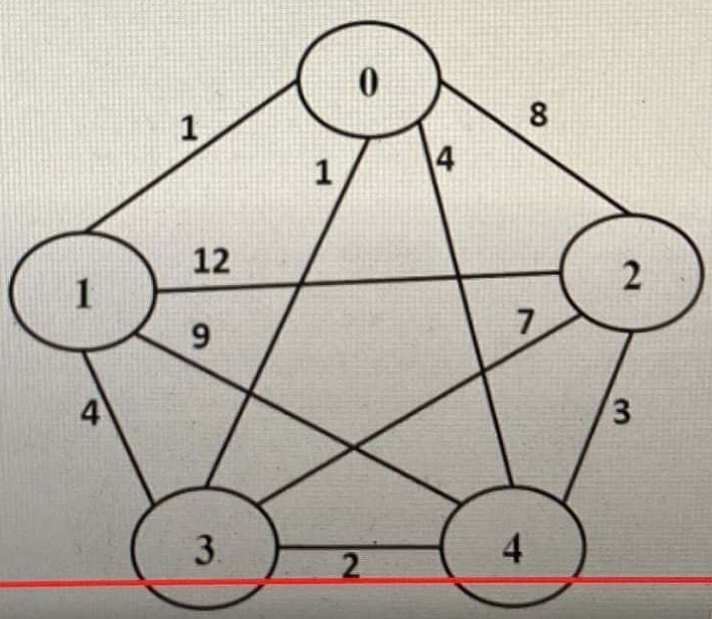
Type of Graphs

- **Undirected Graph:** A graph with only undirected edges.
- **Directed Graph:** A graph with only directed edges.
- **Directed Acyclic Graphs (DAG):** A directed graph with no cycles.
- **Cyclic Graph:** A directed graph with at least one cycle.
- **Weighted Graph:** It is a graph in which each edge is given a numerical weight.
- **Disconnected Graphs:** An undirected graph that is not connected.

Scanned with CamScanner

Qtn) Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. Construct the graph.

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$



5:30 / 5:34

Scanned with CamScanner

GRAPH TRAVERSALS

Breadth First Search

```
BFS(G,s)
1. for each vertex u ∈ G.V-{s}
       u.color = WHITE
       u.d = ∞
       u.π = NIL
2. s.color = GRAY
3. s.d = 0
4. s.π = NIL
6. Q = ϕ
6. ENQUEUE(Q,s)
7. WHILE Q ≠ ϕ
8.     u = DEQUEUE(Q)
9.     for each v ∈ G.Adj(u)
10.        If v.color == WHITE
               v.color = GREY
               v.d = u.d+1
               v.π = u
               ENQUEUE(Q,v)
11.    → u.color = BLACK
```

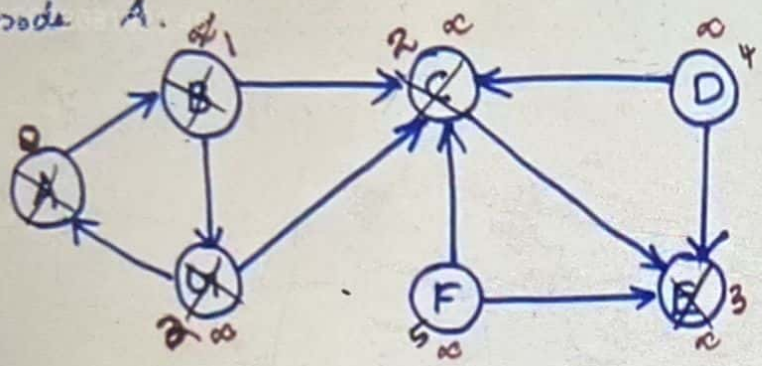
Basic Idea: Need to mark vertices that we have encountered

* Undiscovered (WHITE)
* Discovered (GREY)
* Completely explored (BLACK ~~WHITE~~)

$O(V+E)$

Scanned with CamScanner

Perform BFS traversal on the graph starting from node A.



A	B	C	D	E	F	S
W	W	W	W	W	W	W

A G1

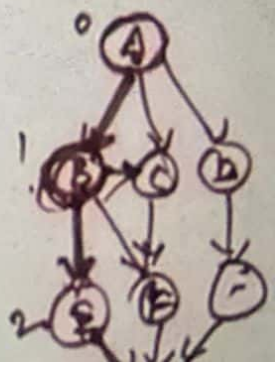
1 G1

Q [A] [B] [C] [D] [E] [F]

DEPTH FIRST SEARCH (DFS)

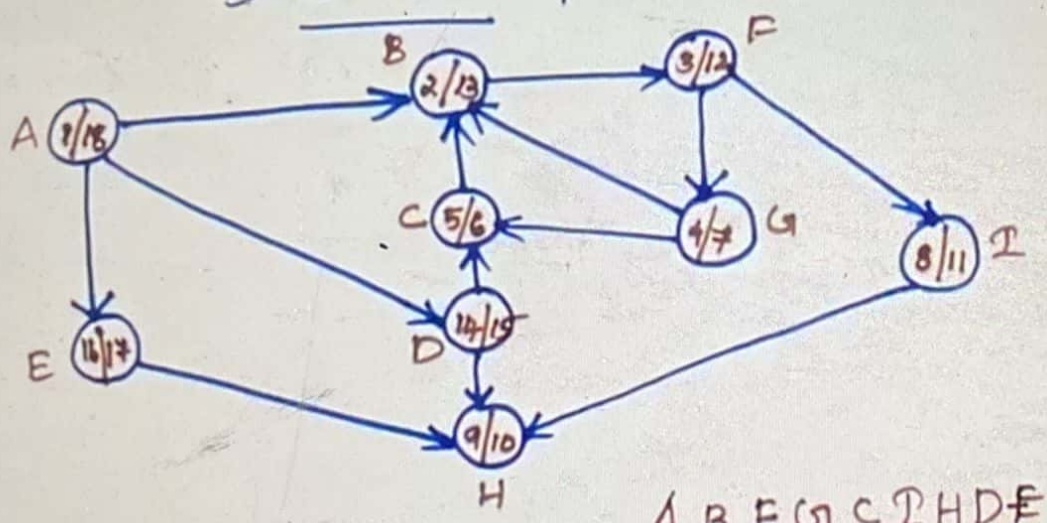
```
DFS(G)
1. for each vertex u ∈ G.V
       u.color = WHITE
       u.π     = NIL
2. time = 0
3. for each vertex u ∈ G.V
       if u.color == WHITE
           DFS_VISIT(G,u)
```

```
DFS_VISIT(G,u)
1. time = time + 1
2. u.d = time
3. u.color = GREY
4. for each v ∈ G.Adj[u]
       if v.color == WHITE        u → v
           v.π = u
           DFS_VISIT(G,v)
5. u.color = BLACK
6. time = time+1
7. u.f = time
```



Scanned with CamScanner

DFS Example



~~O(|V|+|E|)~~

$O(|V|+|E|)$

Tree ✓

Back ✓

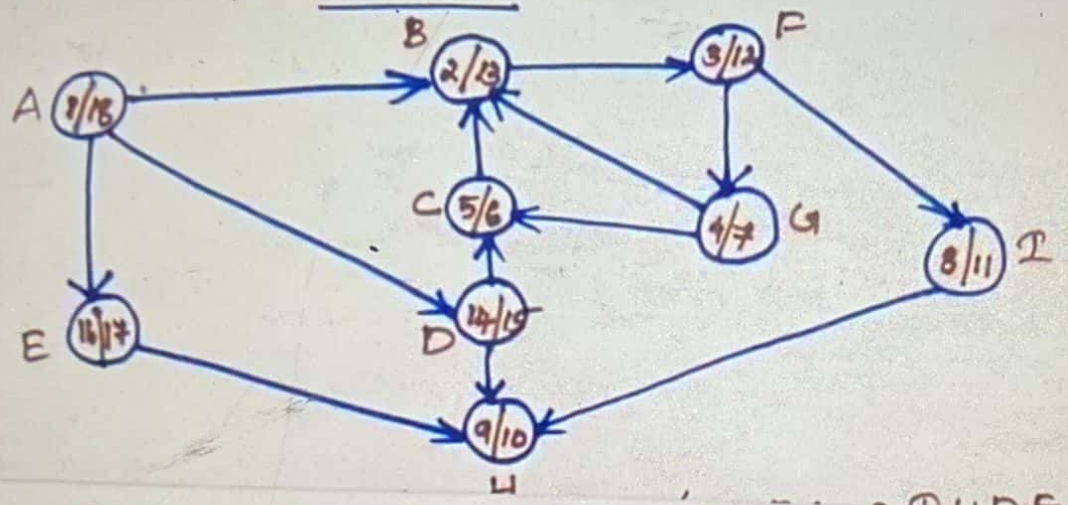
Forward ✓

Cross ✓

A B F G C I H D E

Scanned with CamScanner

DFS Example



~~O(|V|+|E|)~~

$O(|V|+|E|)$

Cross ✓

Tree ✓

Back ✓

Forward ✓

(u,v) Tree Edge ($d(u) < d(v)$ & $f(v) < f(u)$)

AB, BF, I, H $1 < 2$ $13 < 18$

(u,v) Back Edge ($d(v) < d(u)$ & $f(u) < f(v)$)

(C,B) $2 < 5$ $6 < 13$

Cross Edge $d(v) < d(u)$ $f(v) < d(u)$

Scanned with CamScanner

TOPOLOGICAL SORT

A topological sort of DAG, $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

(u, v) $u \rightarrow v$

→ Vertices are arranged in order of decreasing finishing time

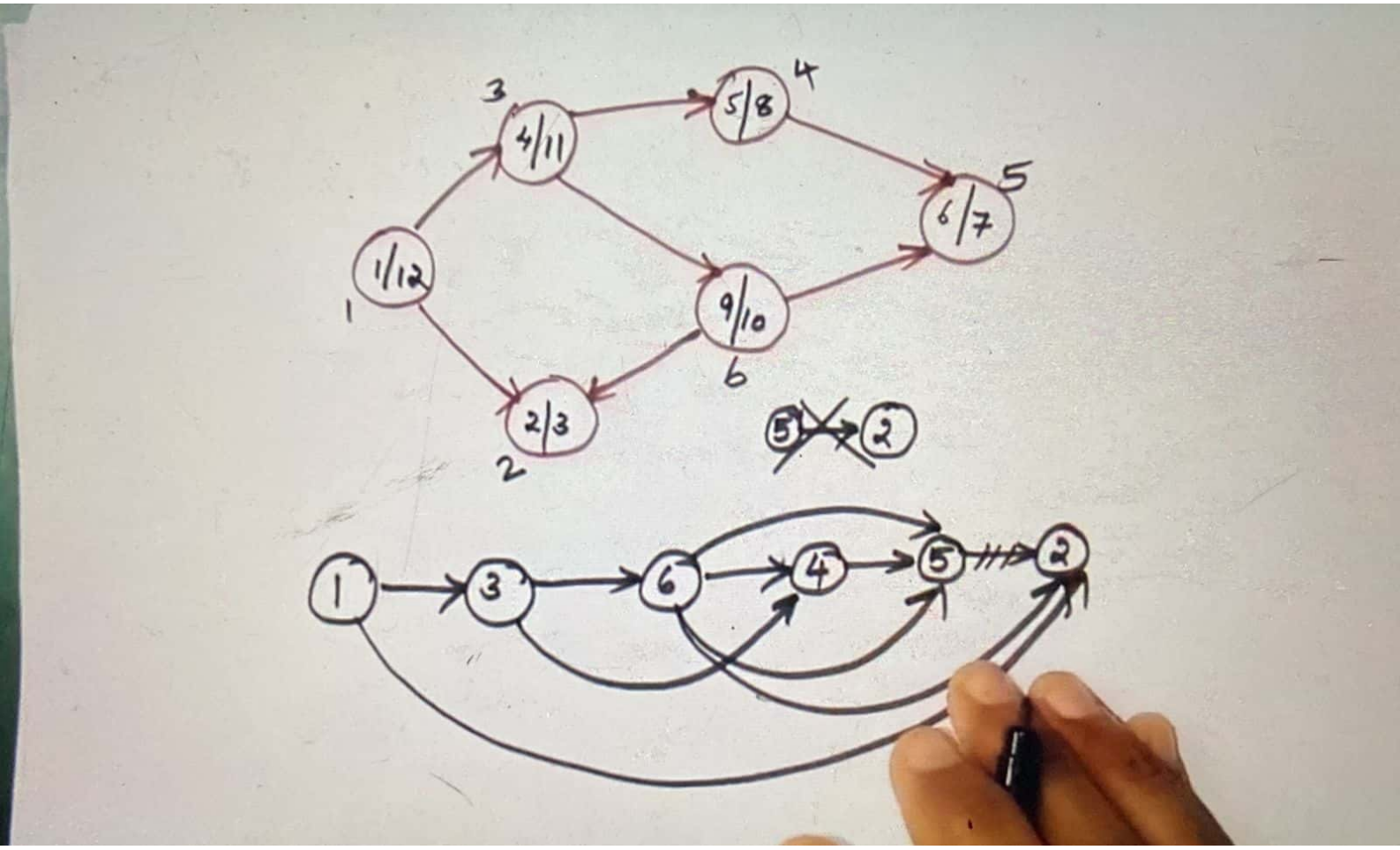
→ $\Theta(V+E)$ ✓

TOPOLOGICAL-SORT(G)

1. Call DFS(G) to compute finishing times $v.f$ for each vertex v.
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices.

1/6 2/5 3/4
A → B → C

Scanned with CamScanner



Scanned with CamScanner

STRONGLY CONNECTED COMPONENTS

A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C vertices u and v are reachable from each other.

→ One of a classical application of DFS

→ Use 2 DFS for this

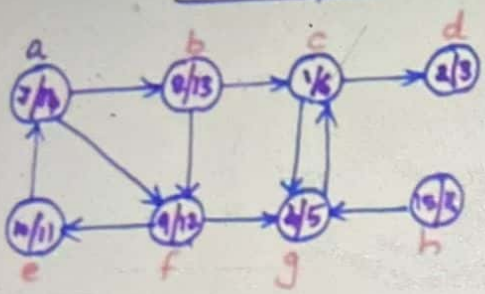
Time Complexity - $O(V+E)$

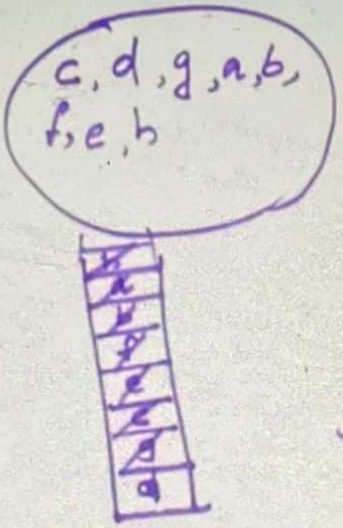
STRONGLY_CONNECTED_COMPONENTS (G)

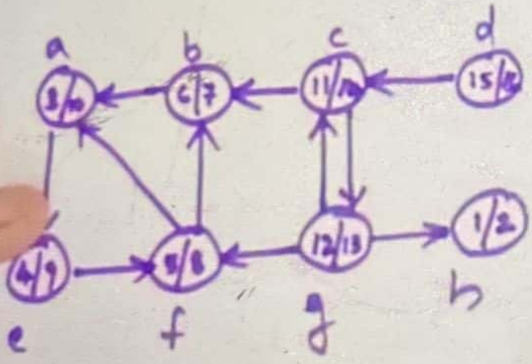
1. Call DFS(G) to compute finishing times $u.f$ for each vertex u
2. Compute G^T
3. Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$
4. Output the vertices of each tree as a seperate strongly connected components.

Scanned with CamScanner

Example







Strongly Connected Compo

→ b

→ a e f b

→ g c

→ d

Scanned with CamScanner