Algorithms & Data Structure

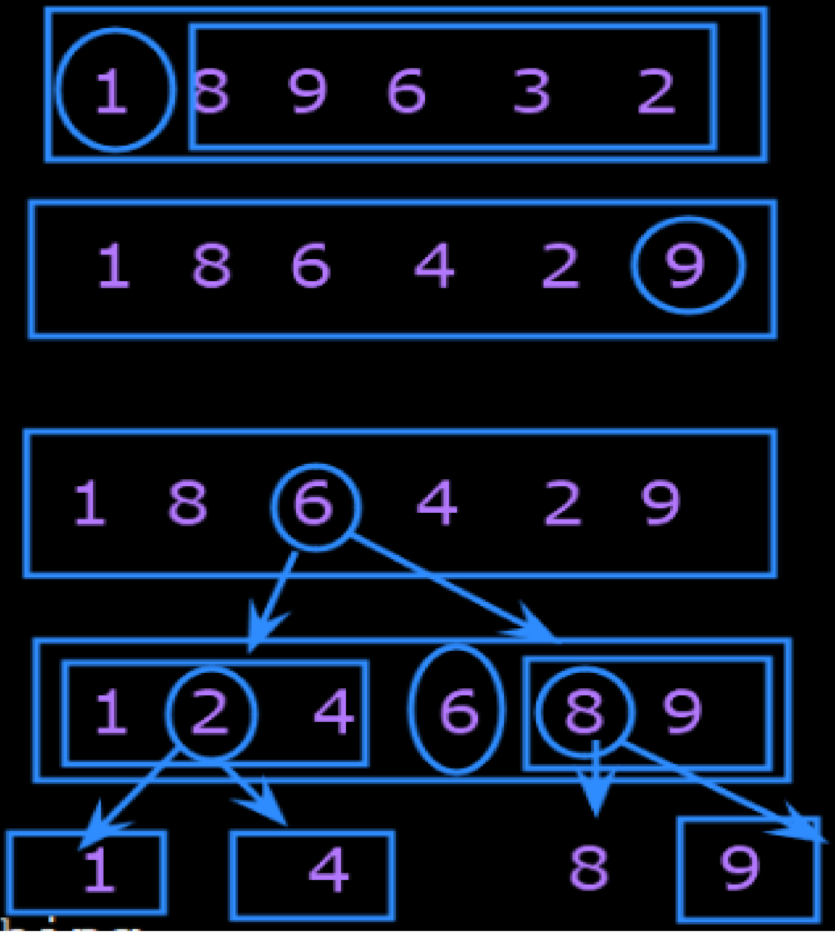
Kiran Waghmare

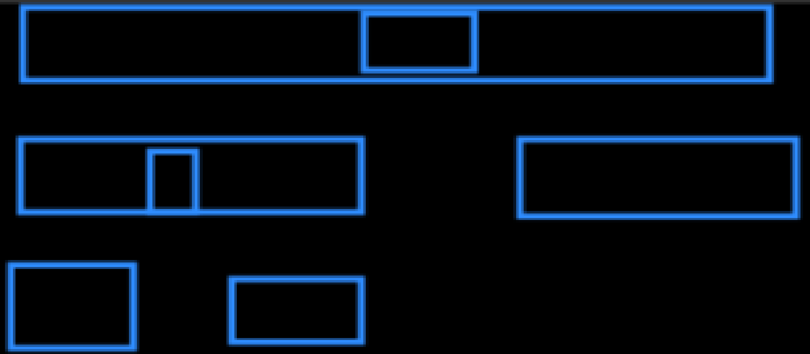
Day11: Algorithms and Data Structure Sessio

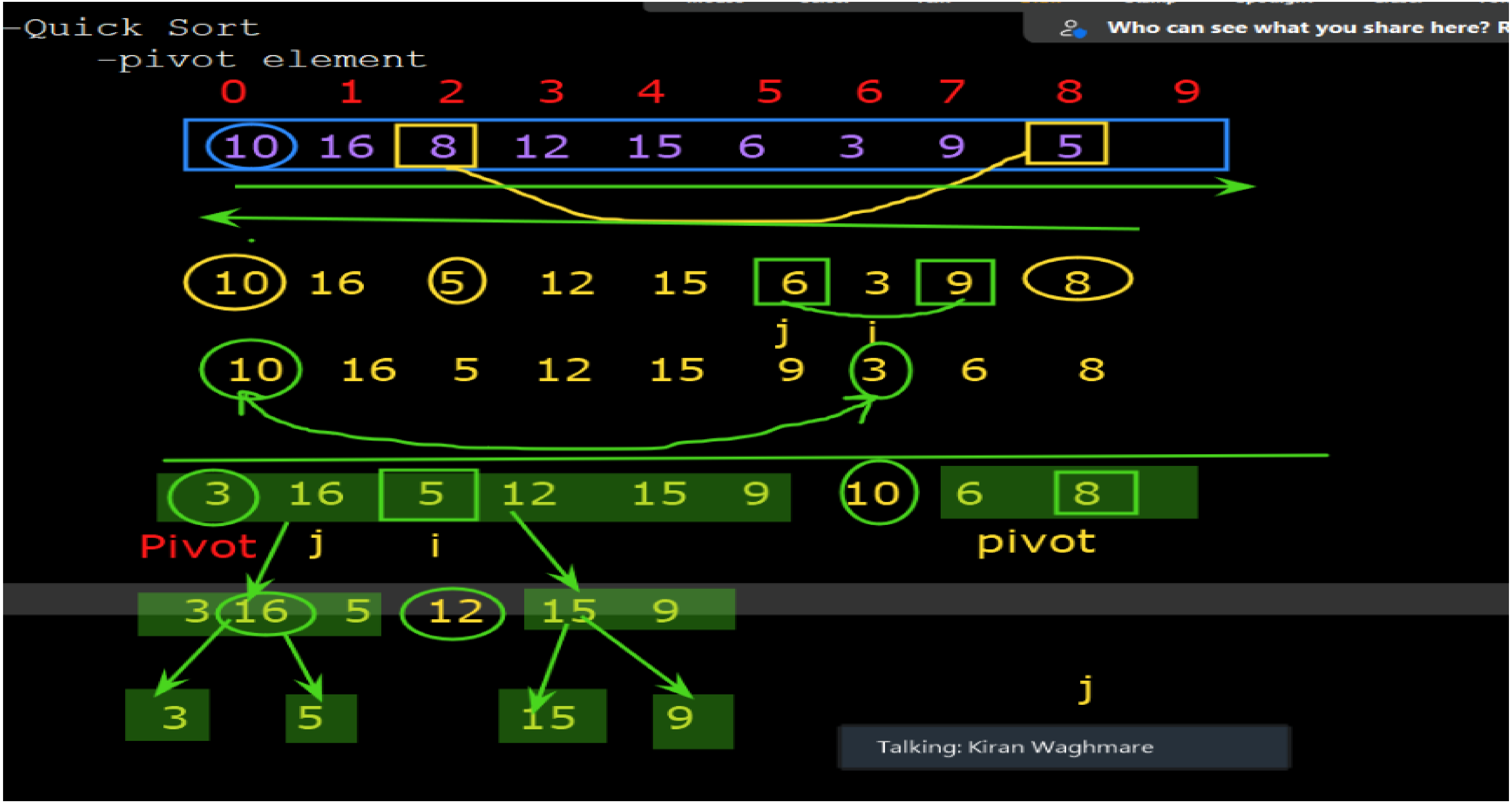
Date :16/06/2021

-Quick Sort

-pivot element



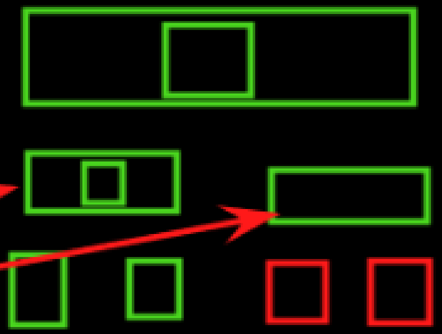


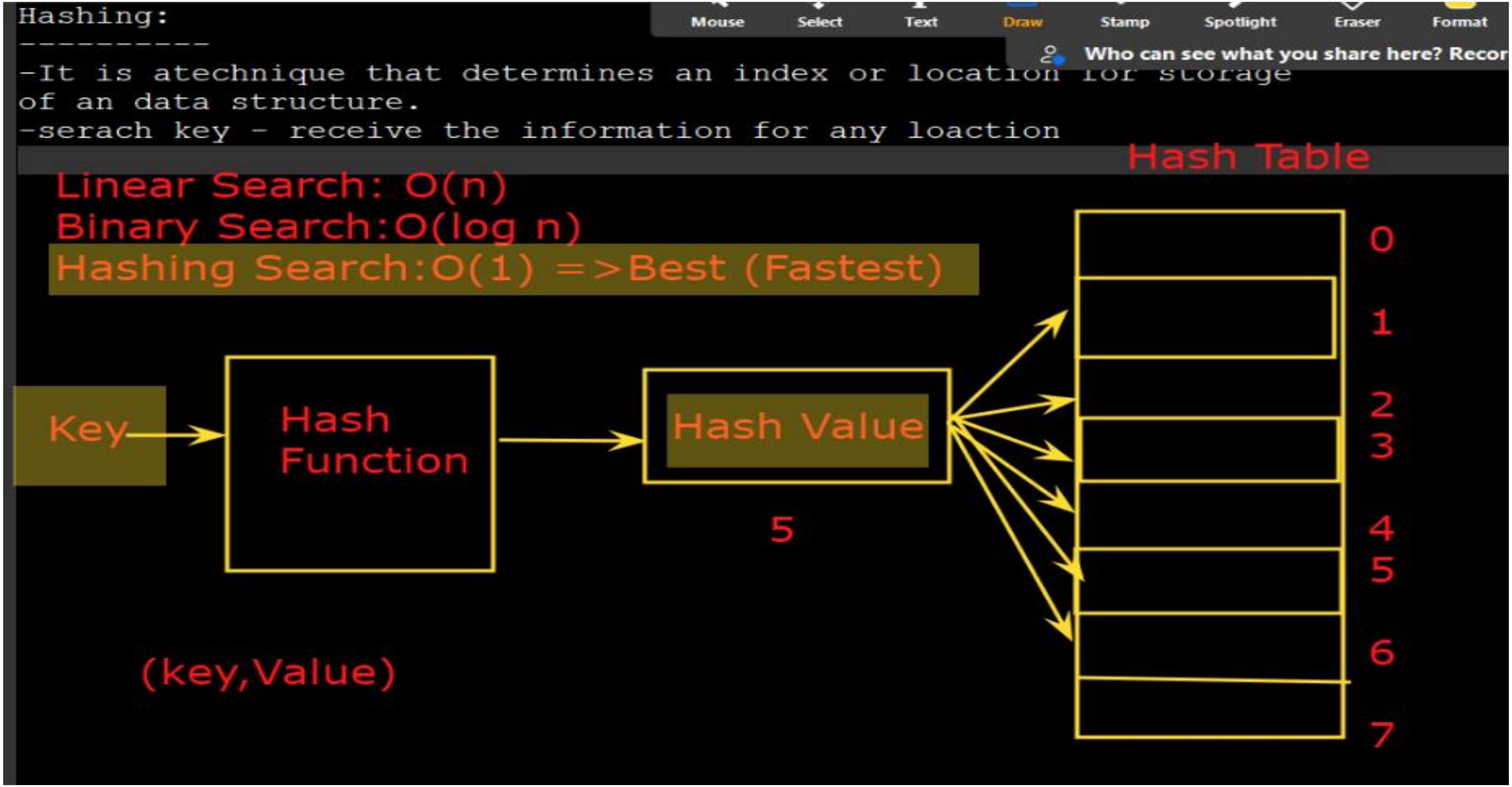

-Quick Sort
-pivot element
0 1 2 3 4 5 6 7 8 9
10 16 8 12 15 6 3 9 5
10 16 5 12 15 6 3 9 8
j i
10 16 5 12 15 9 3 6 8
3 16 5 12 15 9 10 6 8
Pivot j i pivot
3 16 5 12 15 9
3 5 15 9
j
Who can see what you share here?
Talking: Kiran Waghmare

```
//partition
static int partition(int[] arr, int low, int high)
{

    int pivot = arr[high];

    int i = (low - 1);
    for(int j = low; j <= high - 1; j++)
    {

        if (arr[j] < pivot)
        {

            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}
```

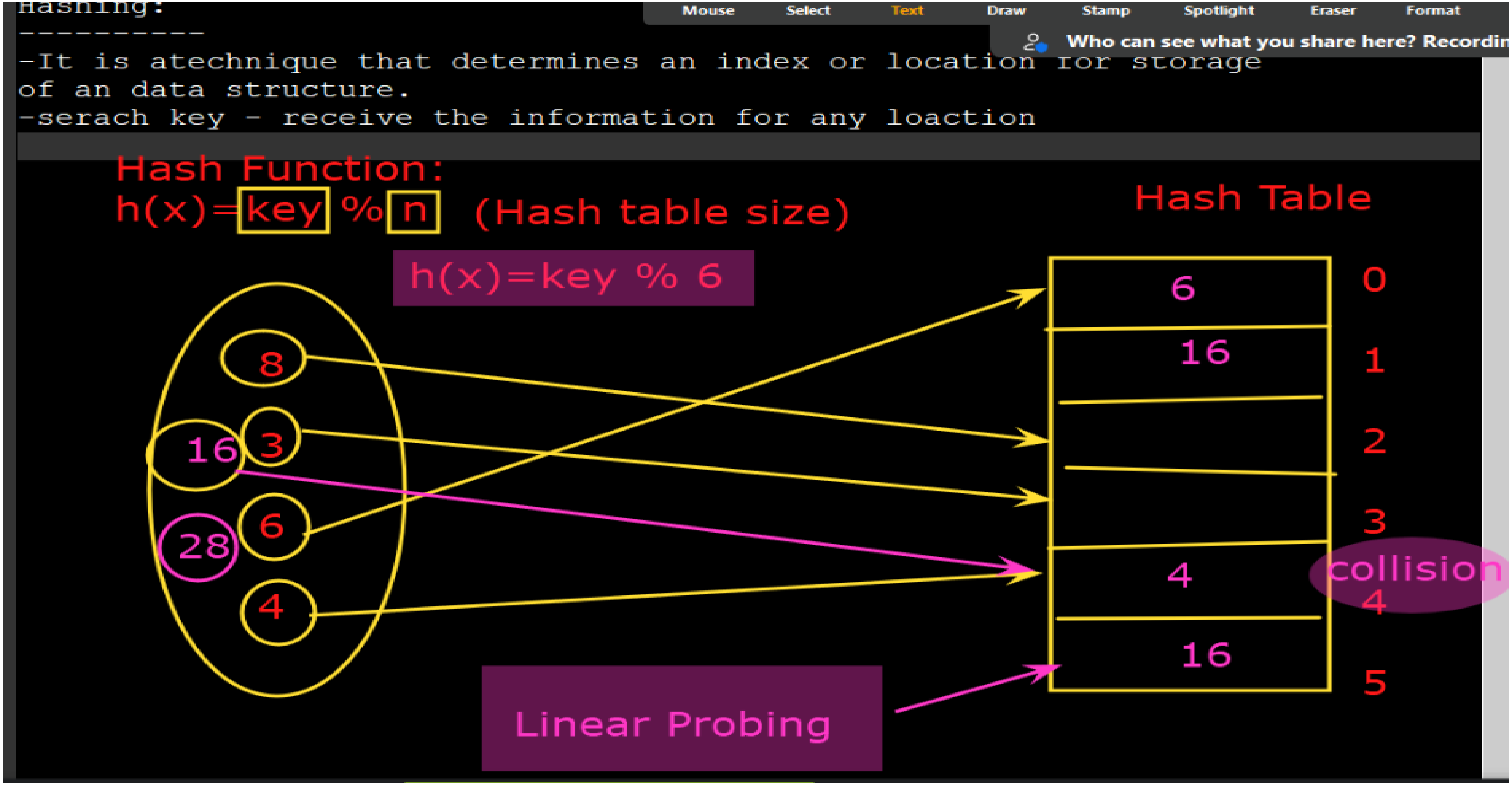
Talking: Kiran Waghmare

```java
static void quickSort(int[] arr, int low, int high)
{
    if (low < high)
    {

        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

static void display(int[] arr, int size)
{
    for(int i = 0; i < size; i++)
        System.out.print(arr[i] + " ");

    System.out.println();
}
```




Hashing:

-It is atechnique that determines an index or location for storage
of an data structure.
-serach key - receive the information for any loaction
Mouse
Select
Text
Draw
Stamp
Spotlight
Eraser
Format
Who can see what you share here? Recor
Hash Table
Linear Search: O(n)
Binary Search:O(log n)
Hashing Search:O(1) =>Best (Fastest)
Key
Hash Function
Hash Value
5
(key,Value)
0
1
2
3
4
5
6
7


Mouse
Select
Text
Draw
Stamp
Spotlight
Eraser
Format
Who can see what you share here? Recordin
Hashing:

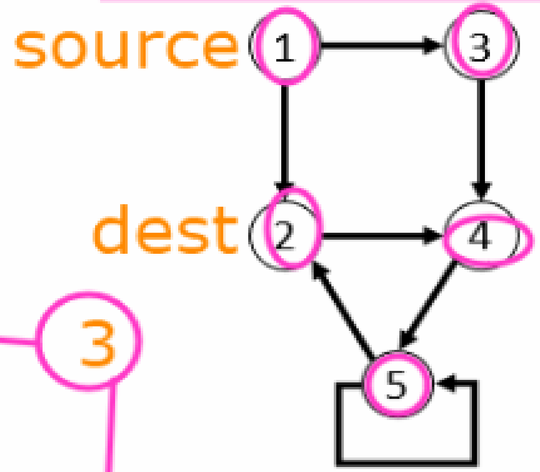
-It is atechnique that determines an index or location for storage
of an data structure.
-serach key - receive the information for any loaction
Hash Function:
h(x)=key % n (Hash table size)
h(x)=key % 6
8
16
3
28
6
4
Hash Table
6
16
4
16
0
1
2
3
collision
4
5
Linear Probing

Graph definition

A graph is specified by a set of vertices (or nodes) V and a set of edges E.

G (V,E)

Directed Graph



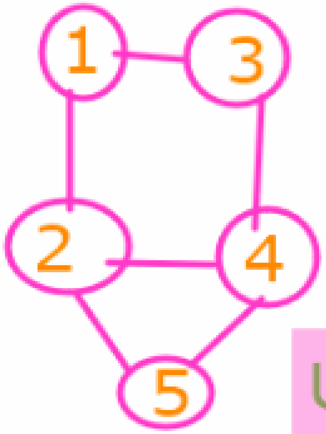
$V = \{1,2,3,4,5\}$ => set of no. of nodes

$E = \{(1,2), (1,3), (2,4), (3,4), (4,5), (5,2), (5,5)\}$

E(Source, dest)

Graphs can be directed or undirected.

Undirected graphs have a symmetric relation.



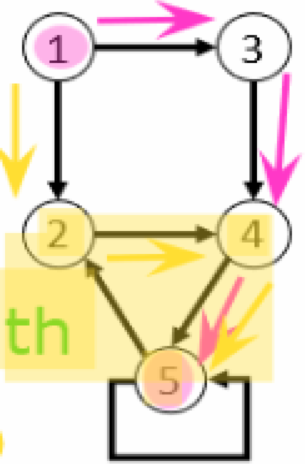
Undirected Graph

Graph definition

A graph is specified by a set of vertices (or nodes) V and a set of edges E.

Path : sequence of nodes that are followed in order

Cycle:



Closed Path

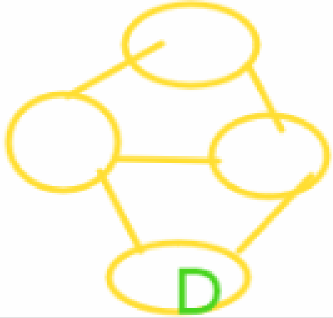
$$V = \{1,2,3,4,5\}$$

V={1, 2, 3,4,1,3}

$$E = \{(1,2), (1,3), (2,4), (3,4), (4,5), (5,2), (5,5)\}$$

Complete Graph

Graphs can be directed or undirected.
Undirected graphs have a symmetric relation.



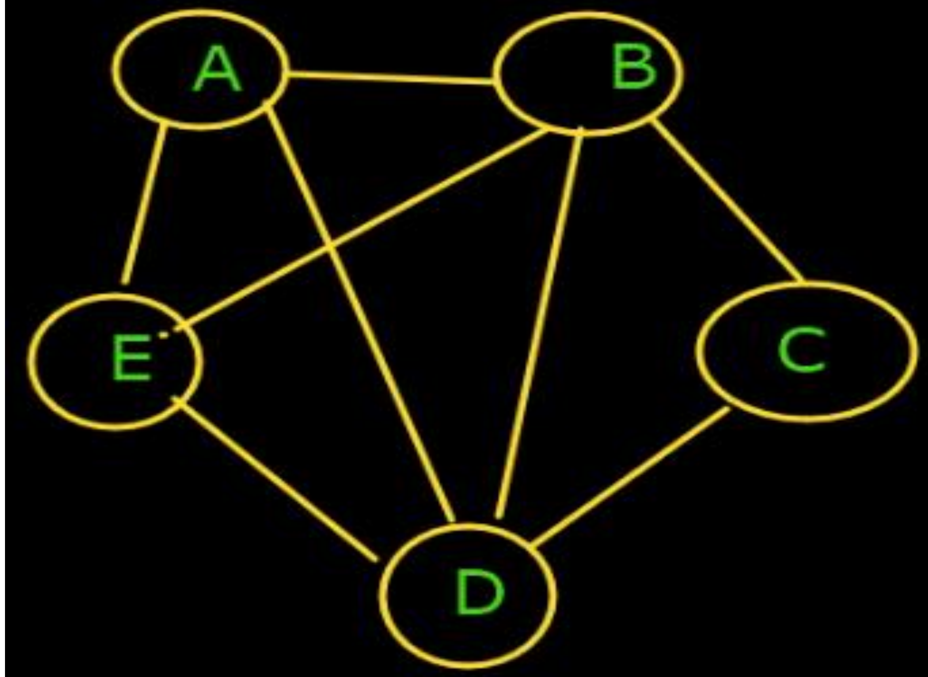
Connected Graph

$\boxed{n(n-1)/2 = \text{No. of edges}}$

-Graph

Mouse Select Text Draw Stamp Spotlight Eraser Format

Who can see what you share here? Rec



UNDIRECTED GRAPH

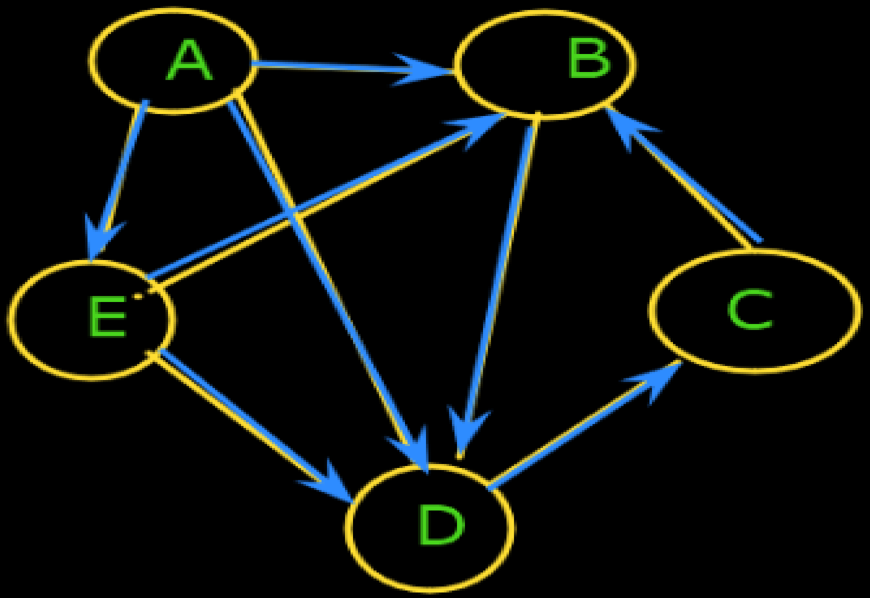
Adjacency Matrix

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

-Graph

Mouse Select Text Draw Stamp Spotlight Eraser Format

Who can see what you share here? Record

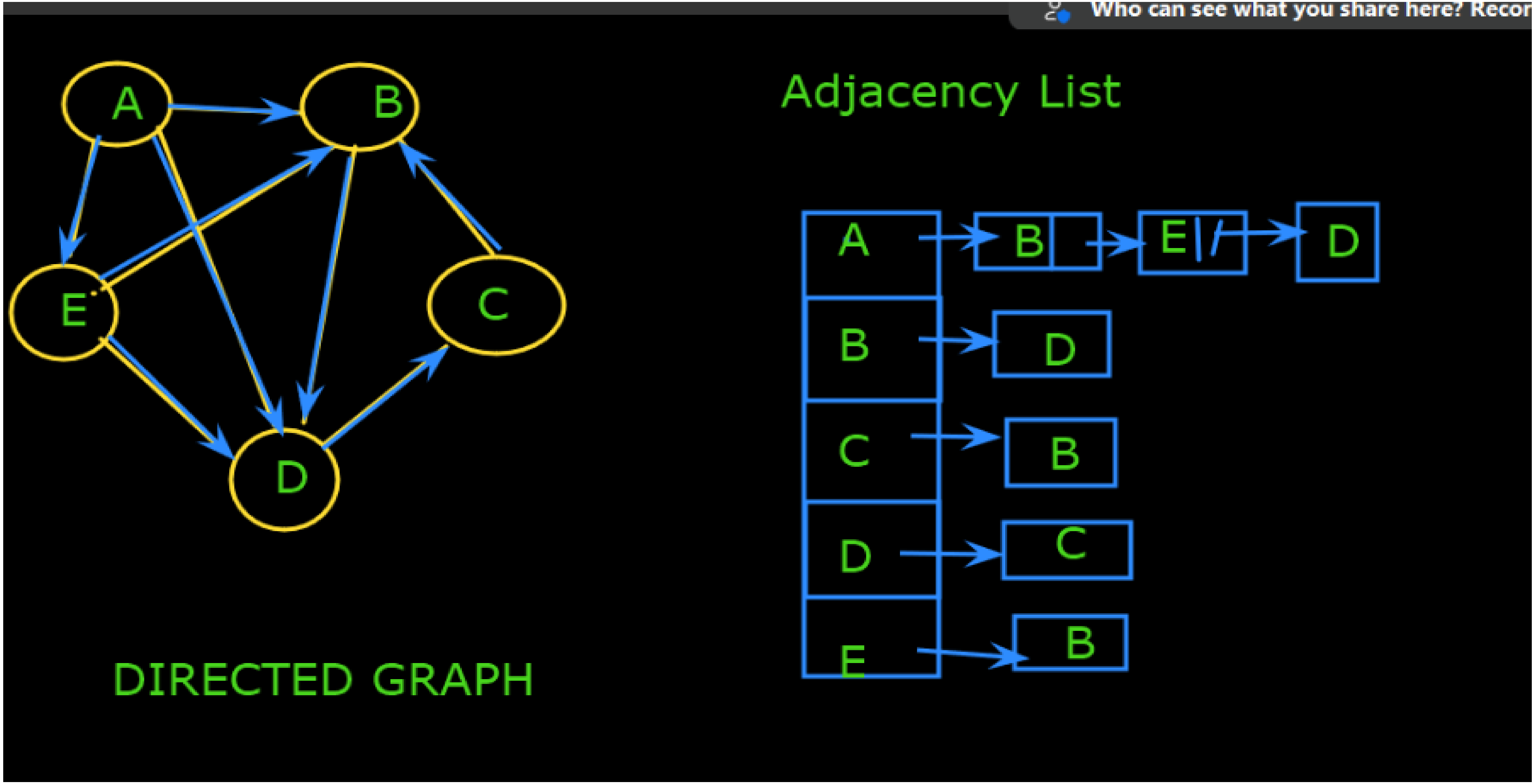


DIRECTED GRAPH

Adjacency Matrix

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

COMPLEXITY: O(n^2)


Who can see what you share here? Recor
A
B
E
C
D
DIRECTED GRAPH
Adjacency List
A
B
E
D
B
D
C
B
D
C
E
B

Finding the nodes reachable from another node

- **function explore(G, v):**
  // Input: $G = (V, E)$ is a graph
  // Output: visited(u) is true for all the
  //   nodes reachable from v

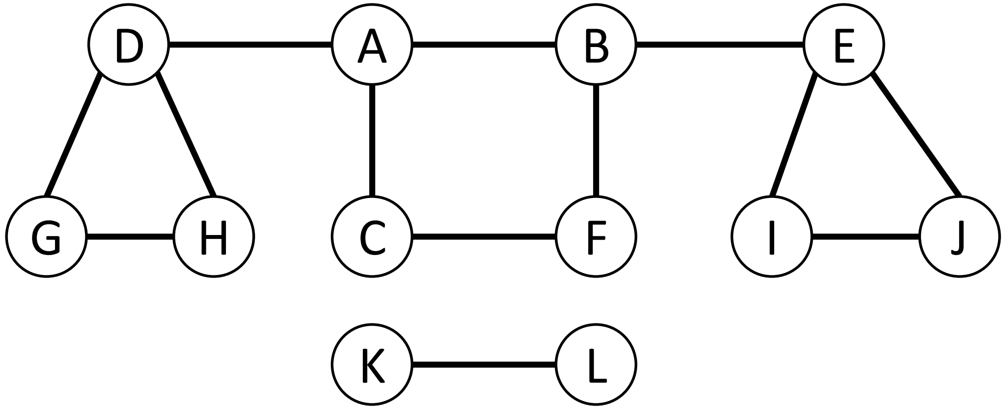
- visited(v) = true
- previsit(v)
- for each edge $(v, u) \in E$:
-   if not visited(u): explore(G, u)
- postvisit(v)

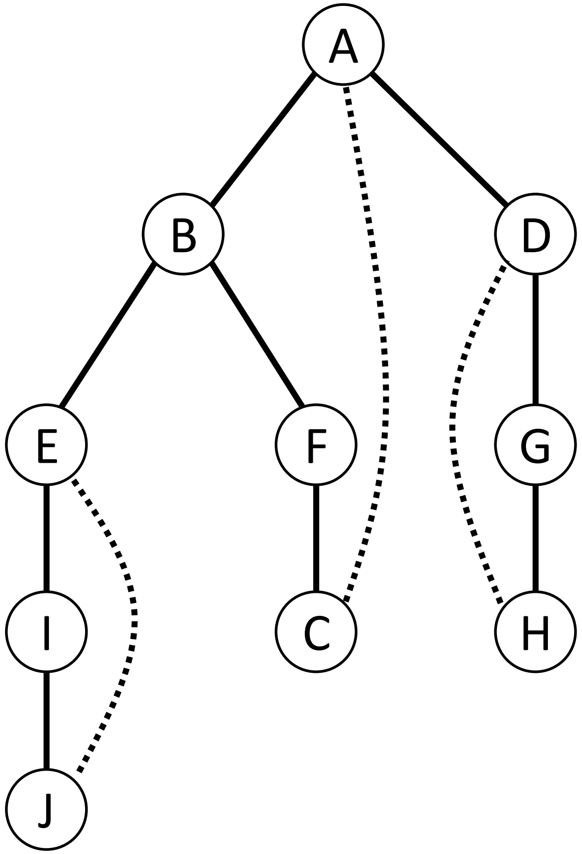
Notes:

- Initially, visited(v) is assumed to be *false* for every $v \in V$.
- pre/postvisit functions are not required now.

Finding the nodes reachable from another node

- **function explore(G, v):**
  visited(v) = true
  for each edge $(v, u) \in E$:
  if not visited(u): explore(G, u)





Dotted edges are ignored (*back edges*): they lead to previously visited vertices.
The solid edges (*tree edges*) form a tree.

Depth-first search

- **function DFS(G):**
- **for all $v \in V$:**
  visited(v) = false
- **for all $v \in V$:**
  if not visited(v): explore(G, v)

DFS traverses the entire graph.

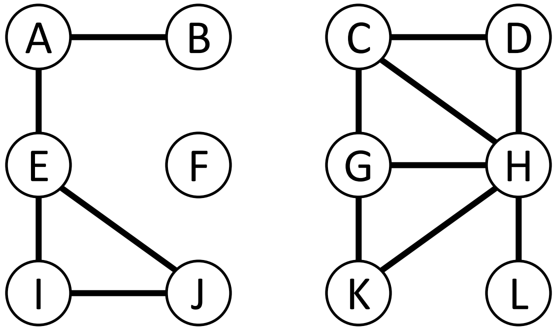
Complexity:

- Each vertex is visited only once (thanks to the chalk marks)
- For each vertex:
  - A fixed amount of work (pre/postvisit)
  - All adjacent edges are scanned

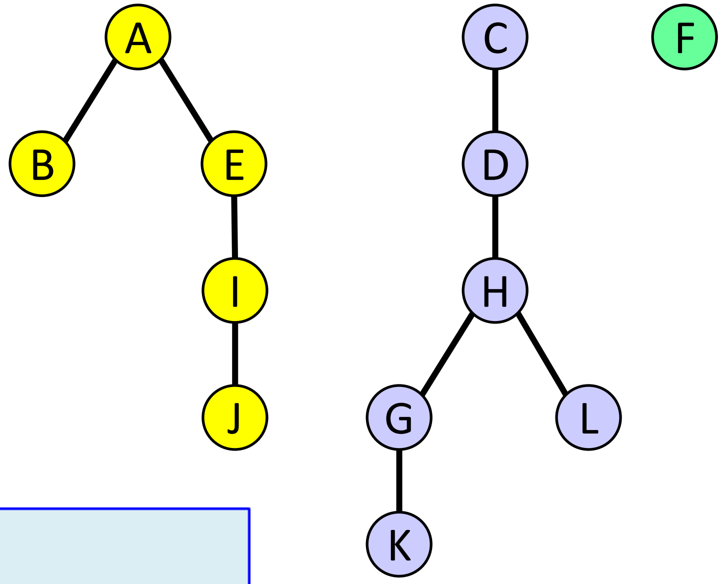
Running time is $O(|V| + |E|)$.
Difficult to improve: reading a graph already takes $O(|V| + |E|)$.

DFS example

Graph

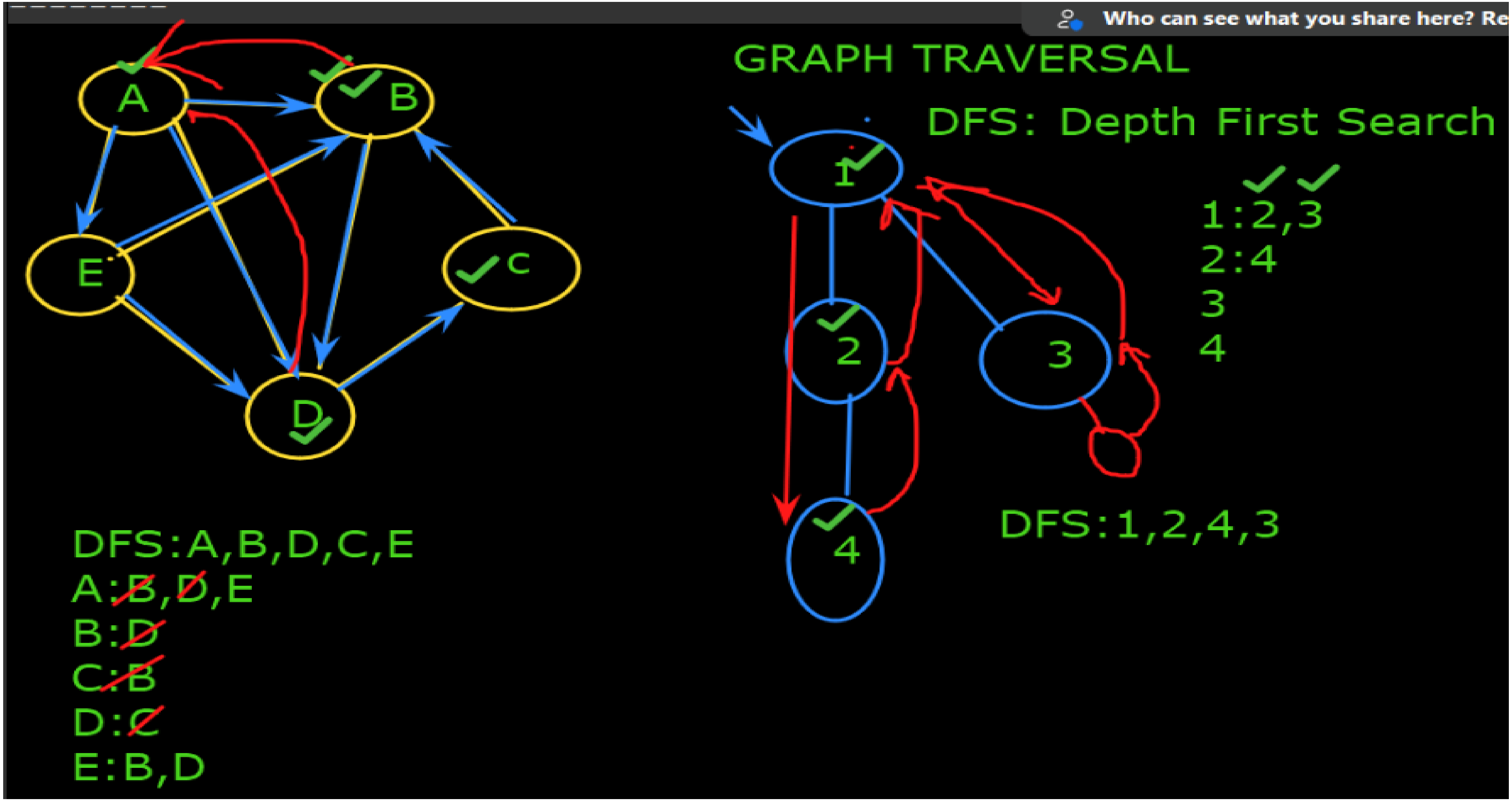


DFS forest




```
function DFS(G):
  for all v ∈ V:
    visited(v) = false
  for all v ∈ V:
    if not visited(v): explore(G,v)
```

- The outer loop of DFS calls ***explore*** three times (for A, C and F)
- Three trees are generated. They constitute a *forest*.


Who can see what you share here? Re
GRAPH TRAVERSAL
DFS: Depth First Search
A
B
C
D
E
DFS:A,B,D,C,E
A:~~B~~,~~D~~,E
B:~~D~~
C:~~B~~
D:~~C~~
E:B,D
1
2
3
4
1:2,3
2:4
3
4
DFS:1,2,4,3

BFS algorithm

- **BFS visits vertices layer by layer: $0,1,2,\ldots,d$.**

- **Once the vertices at layer d have been visited, start visiting vertices at layer $d+1$.**

- **Algorithm with two active layers:**
  - Vertices at layer d (currently being visited).
  - Vertices at layer $d+1$ (to be visited next).

- **Central data structure: a queue.**

BFS algorithm

- function BFS(G, s)

  // Input: Graph $G(V, E)$, source vertex s.

  // Output: For each vertex u, dist[u] is

  //   the distance from s to u.

  for all $u \in V$: dist[u]= ∞

  dist[s] = 0

  Q = {s} // Queue containing just s

  while not Q.empty():

   u = Q.pop_front()

   for all $(u, v) \in E$:

    if dist[v] = ∞:

     dist[v] = dist[u] + 1

     Q.push_back(v)

Runtime $O(|V| + |E|)$: Each vertex is visited once, each edge is visited once (for directed graphs) or twice (for undirected graphs).

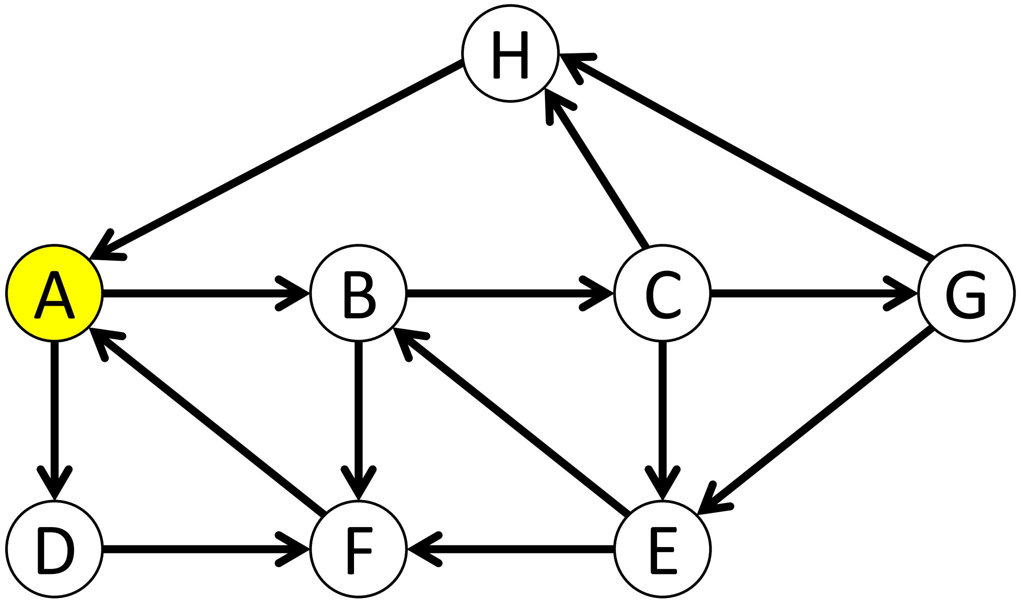
Reachability: BFS vs. DFS

Input: A graph G and a source node s.
Output: $\forall u \in V$: reached$[u] \Leftrightarrow u$ is reachable from s.

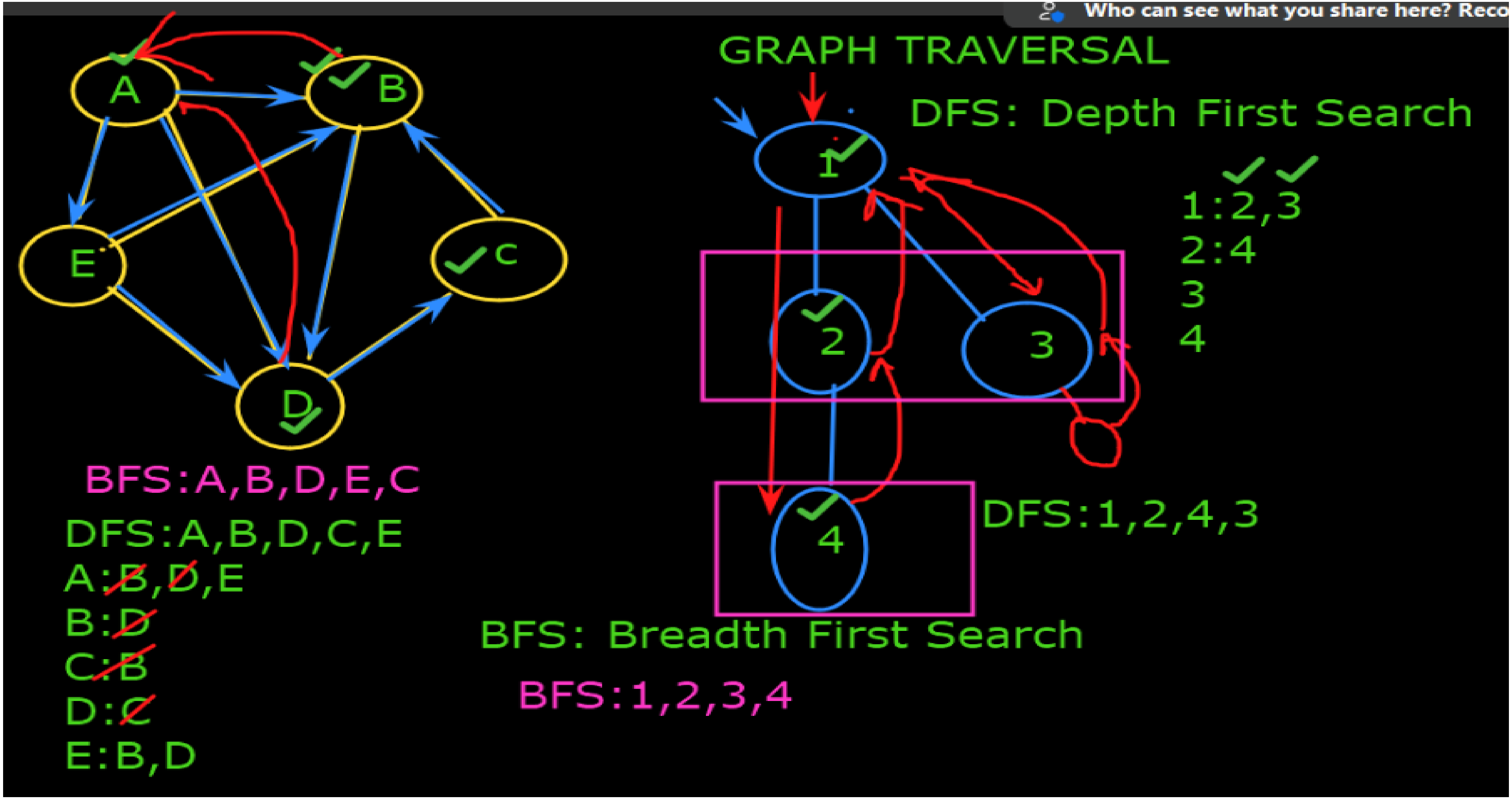
```
function BFS(G, s)

  for all u ∈ V:
    reached[u] = false

  Q = [] // Empty queue
  Q.push_back(s)
  reached[s] = true
  while not Q.empty():
    u = Q.pop_front()
    for all (u,v) ∈ E:
      if not reached[v]:
        reached[v] = true
        Q.push_back(v)
```

```
function DFS(G, s)

  for all u ∈ V:
    reached[u] = false

  S = []    // Empty stack
  S.push(s)
  while not S.empty():
    u = S.pop()
    if not reached[u]:
      reached[u] = true
      for all (u,v) ∈ E:
        if not reached[v]:
          S.push(v)
```

Reachability: BFS vs. DFS



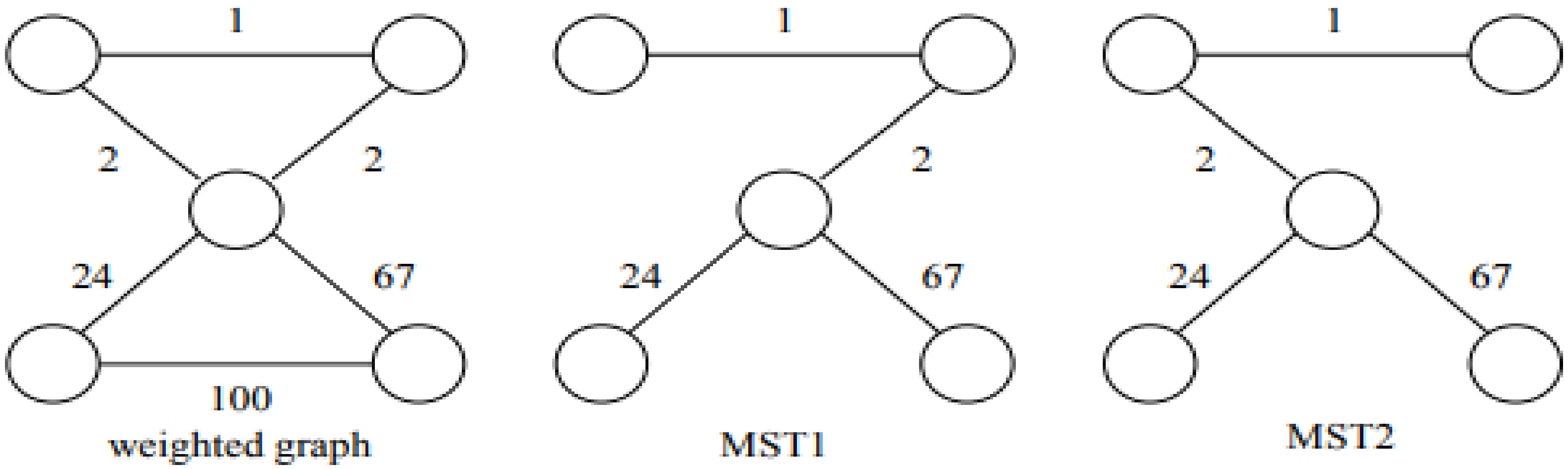

```
DFS order:   A   B   C   E   F   G   H   D

BFS order:   A   B   D   C   F   E   G   H
Distance:    0   1   1   2   2   3   3   3
```


Who can see what you share here? Reco
GRAPH TRAVERSAL
DFS: Depth First Search
A
B
C
D
E
1
2
3
4
1:2,3
2:4
3
4
BFS:A,B,D,E,C
DFS:A,B,D,C,E
A:~~B~~,~~D~~,E
B:~~D~~
C:~~B~~
D:~~C~~
E:B,D
DFS:1,2,4,3
BFS: Breadth First Search
BFS:1,2,3,4

Minimum Spanning Trees

Remark: The minimum spanning tree may not be unique. However, if the weights of all the edges are pairwise distinct, it is indeed unique (we won't prove this now).

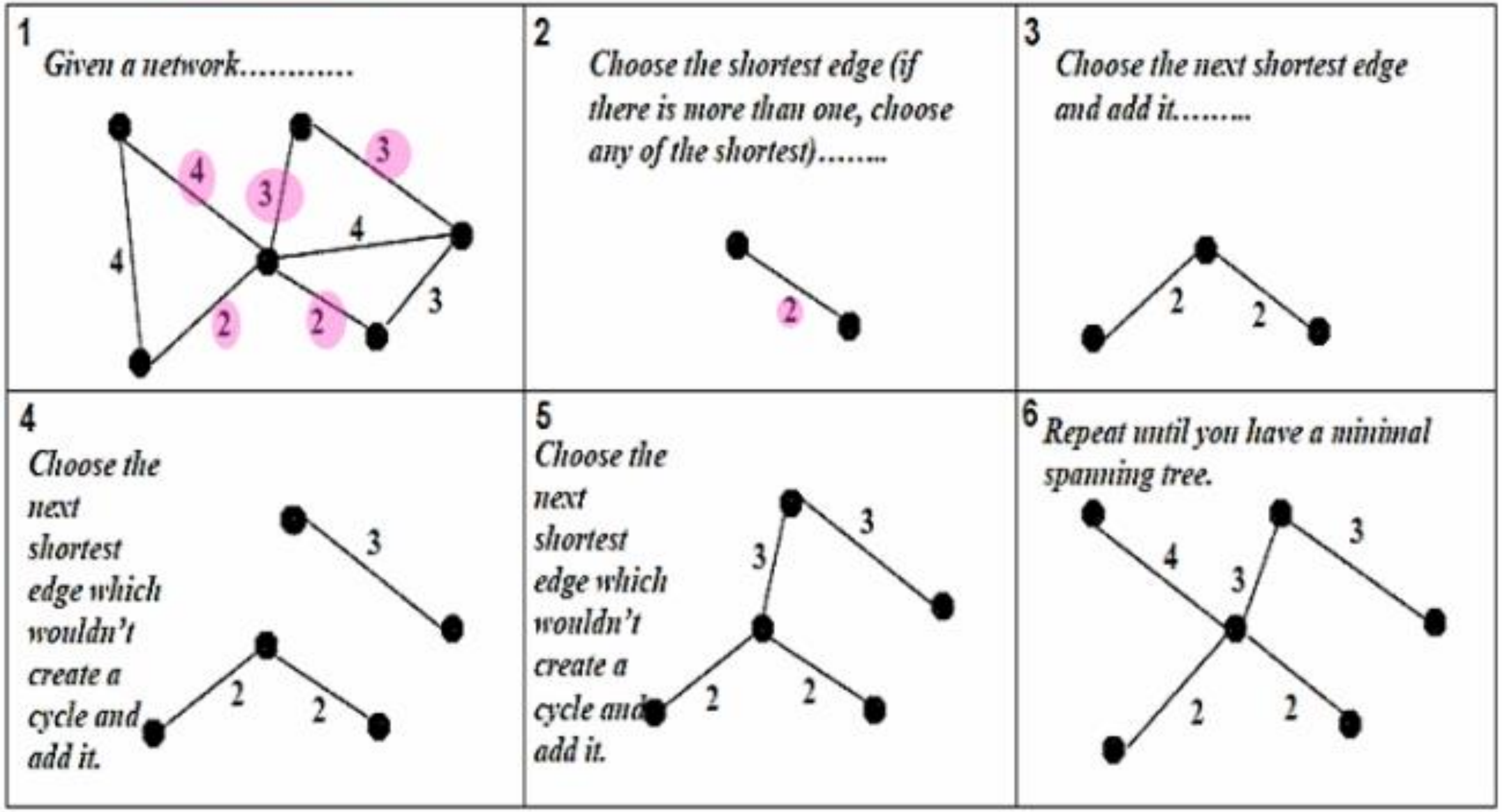
Example:



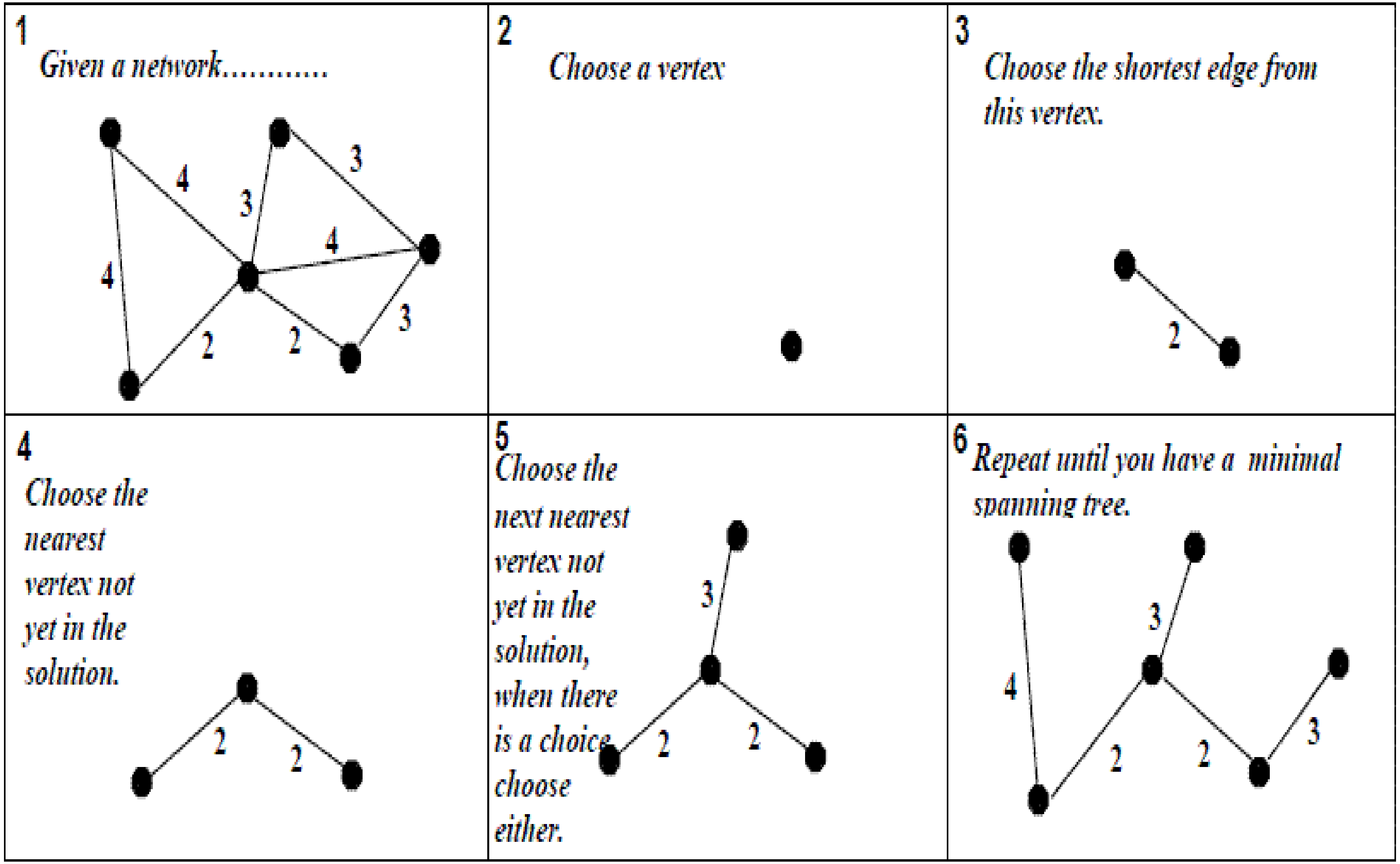
Finding Minimum Spanning Trees

1. Kruskal’s algorithm,
2. Prim’s algorithm

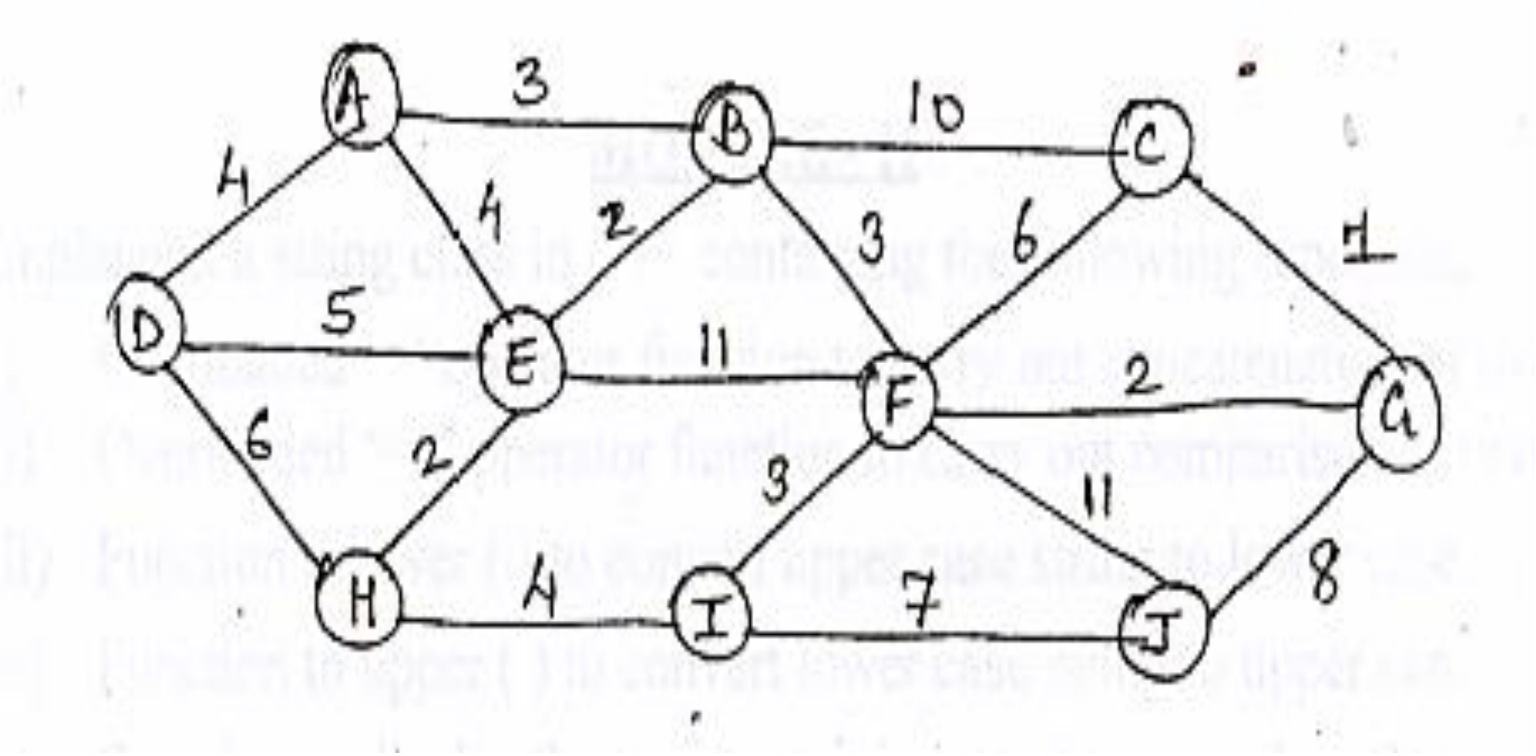
Kruskal's Algorithm



Prim's Algorithm



Obtain minimum cost spanning tree for the following graph using Kruskal's algorithm.



Obtain minimum cost spanning tree for the following graph using Kruskal's algorithm.

8