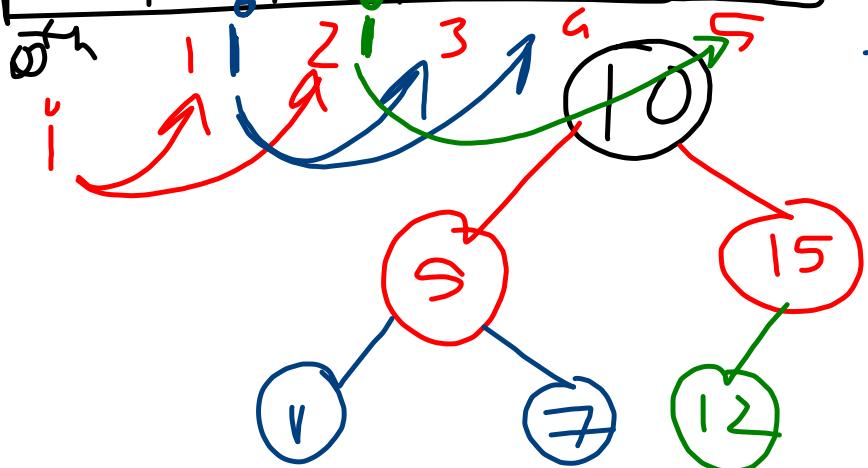
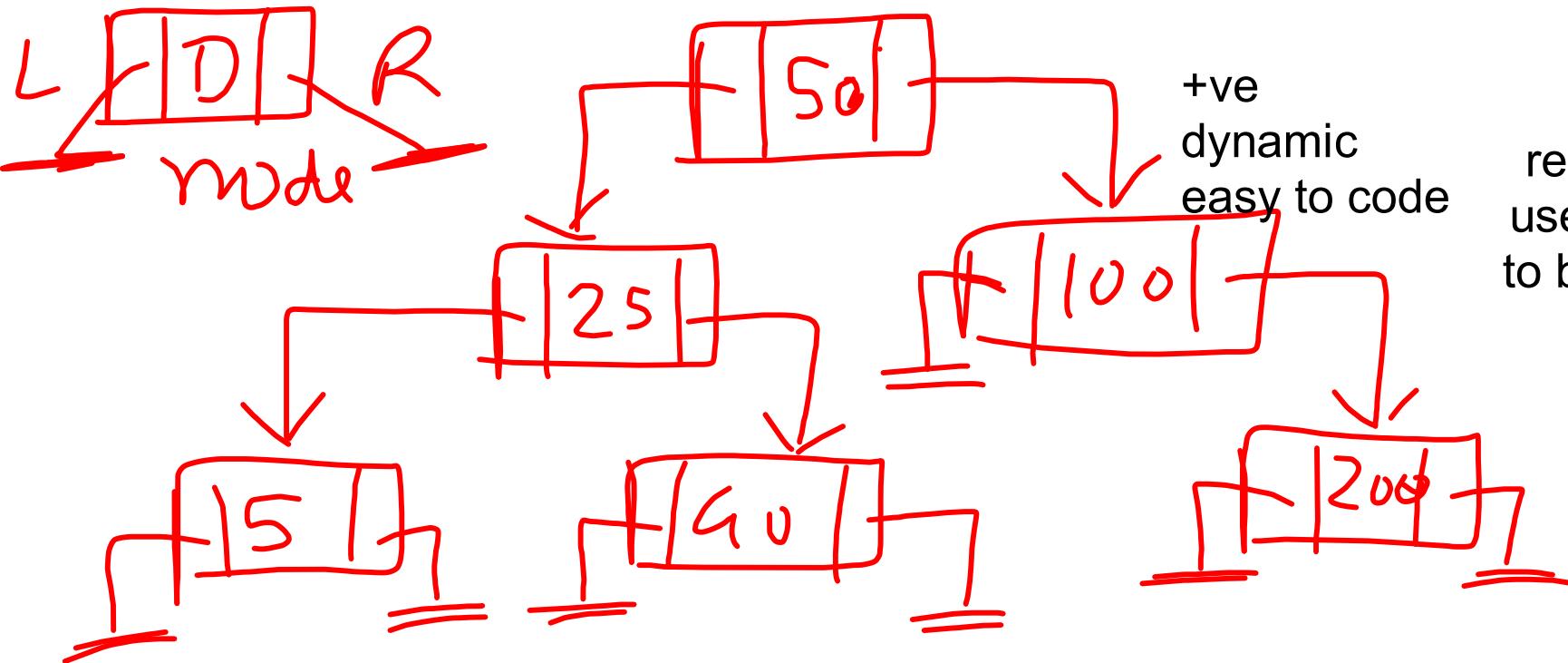


10	5	15	1	7	12
----	---	----	---	---	----



+ve  
static  
simple  
easy to  
code

Dis  
-Static  $\rightarrow$  fixed  
size  
-Flat structure  
to access  
Tree (Slow)

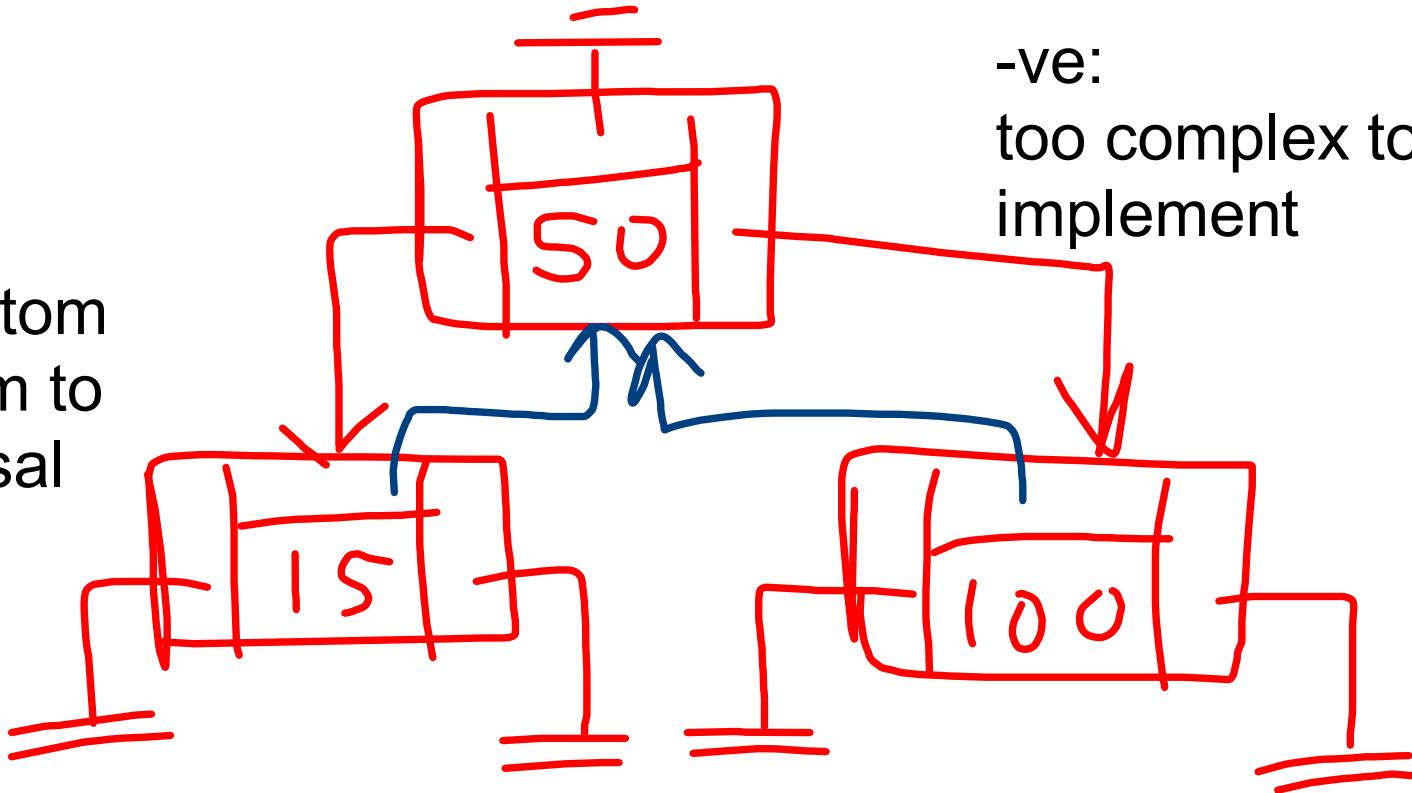


+ve  
dynamic  
easy to code

-ve  
recursion to be  
used as only top  
to bottom access  
possible

+ve  
-dynamic  
-top to bottom  
and bottom to  
top traversal  
possible

-ve:  
too complex to  
implement



DataStructures - NetBeans IDE 8.2

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

Search (Ctrl+I)

<default config... >

Source History Navigator Projects Files Services

...v Treemain.java GraphDemo.java StackDemo.java QueueLinearDemo.java CircularQueueDemo.java PriorityQueueDemo.java LinkedListLinear.java DynamicStackDemo.java DynamicQueueDemo.java

```
10  
11 class Node  
12 {  
13     int key;  
14     Node left, right;  
15  
16     public Node(int item)  
17     {  
18         key = item;  
19         left = right = null;  
20     }  
21 }  
22 public class Treemain {  
23     private Node root;
```

Output



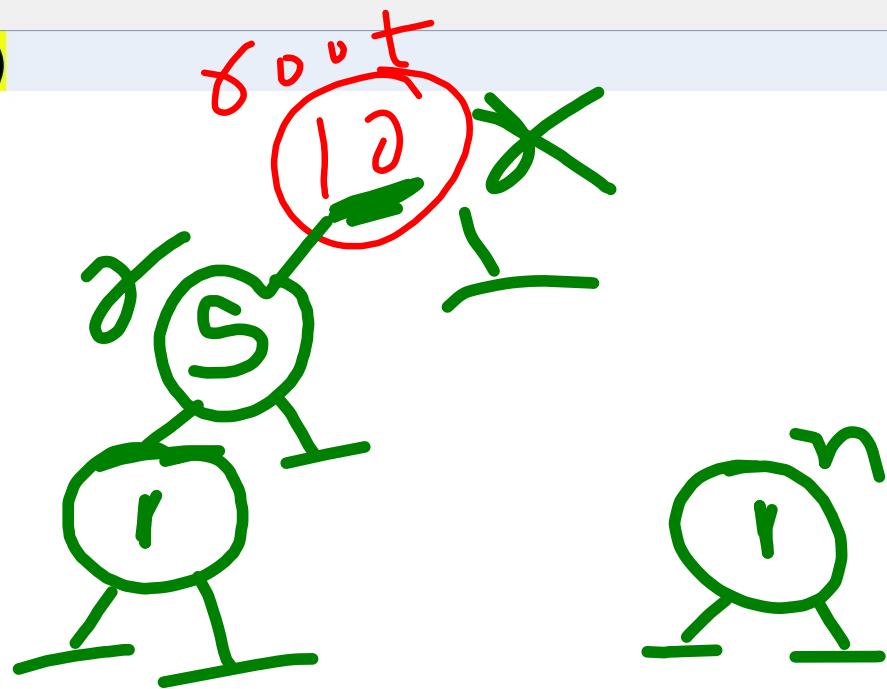
...va Treemain.java x GraphDemo.java x StackDemo.java x QueueLinearDemo.java x CircularQueueDemo.java x PriorityQueueDemo.java x LinkedListLinear.java x DynamicStackDemo.java x DynamicQueueDemo.ja..

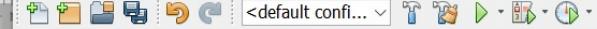
Source History

```
public void insert(Node r,Node n)
```

```
{  
    if (root==null)  
    { root=n; }  
    else  
    {  
        if(n.key<r.key)  
        {  
            if(r.left==null)  
                r.left=n;  
            else  
                insert(r.left,n);  
        }  
        else
```

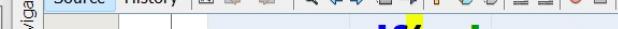
```
insert(10,1);  
insert(5,1);
```





...va Treemain.java x GraphDemo.java x StackDemo.java x QueueLinearDemo.java x CircularQueueDemo.java x PriorityQueueDemo.java x LinkedListLinear.java x DynamicStackDemo.java x DynamicQueueDemo..

Source History

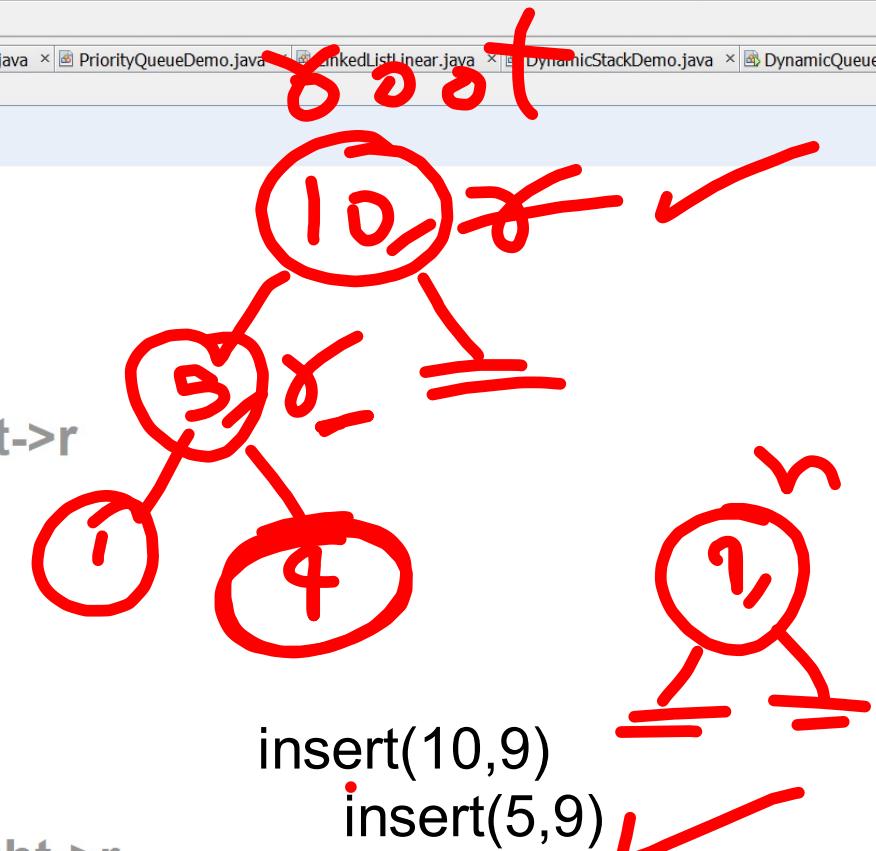


Services Projects Files



Output

```
if(n.key<r.key)
{
    if(r.left==null)
        r.left=n;
    else
        insert(r.left,n); //-->r.left->r
}
else
{
    if(r.right==null)
        r.right=n;
    else
        insert(r.right,n); //r->right->r
}
```





...v Treemain.java x GraphDemo.java x StackDemo.java x QueueLinearDemo.java x CircularQueueDemo.java x PriorityQueueDemo.java x LinkedListLinear.java x DynamicStackDemo.java x DynamicQueueDemo..

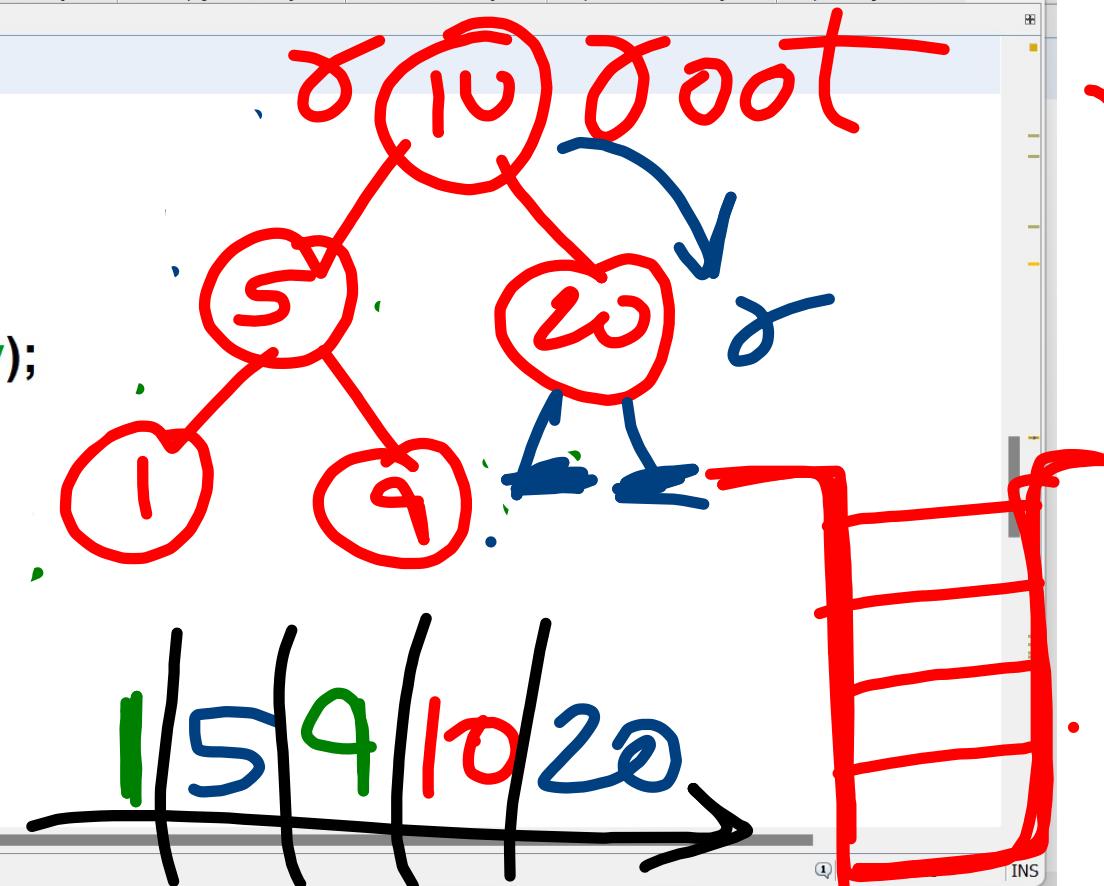
Source History

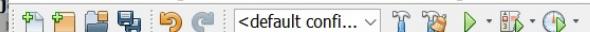
**public void inorder(Node r)**

```
52  
53     if(r!=null)  
54     {  
55         inorder(r.left);  
56         System.out.println(r.key);  
57         inorder(r.right);  
58     }  
59  
60 }
```

**public void counter(Node r)**

```
62  
63     if(r!=null)  
64     {
```

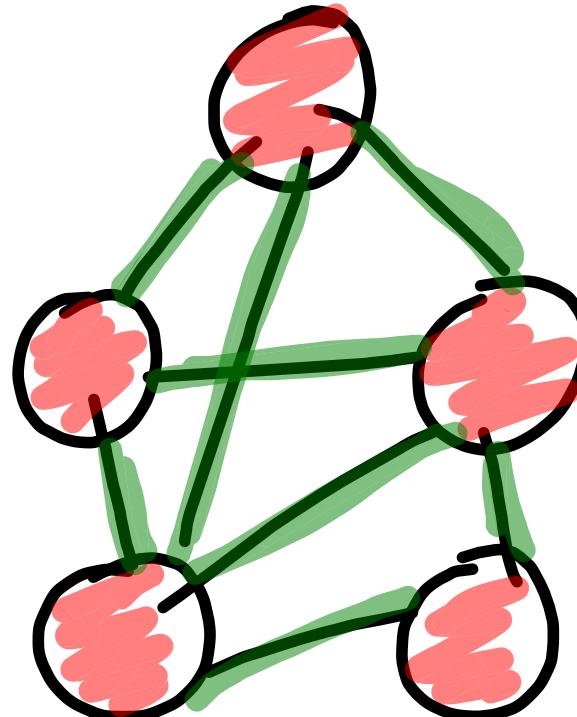


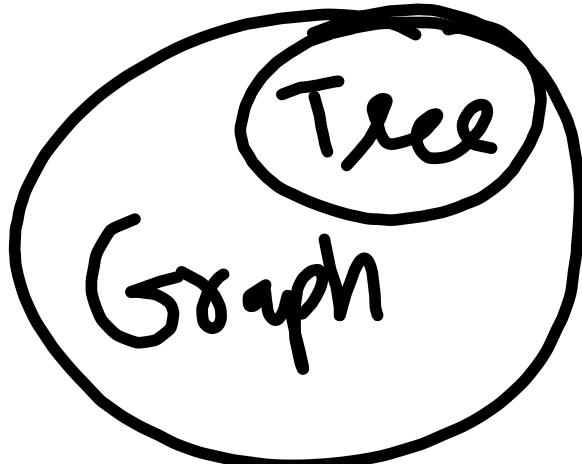


```
public void counter(Node r)
{
    if(r!=null)
    {
        counter(r.left);
        // if(r.left==null && r.right==null)
        Treemain.c++;
        counter(r.right);
    }
}
```

# Graph is

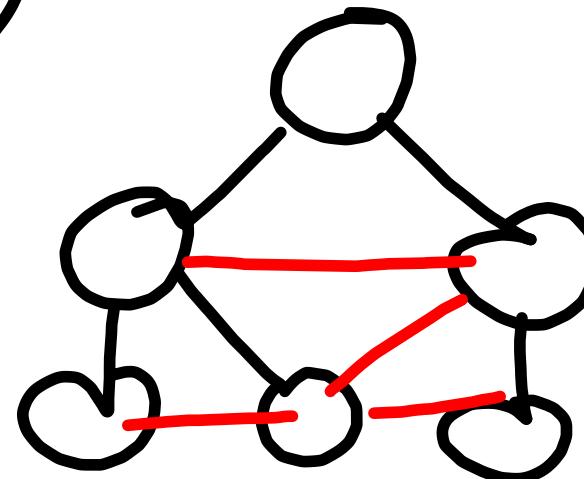
- A data structure that consists of a set of nodes (vertices) and a set of edges that relate the nodes to each other.
- The set of edges describes relationships among the vertices.
- A graph  $G$  is defined as follows:
- $G=(V,E)$
- $V(G)$ : a finite, nonempty set of vertices
- $E(G)$ : a set of edges (pairs of vertices)

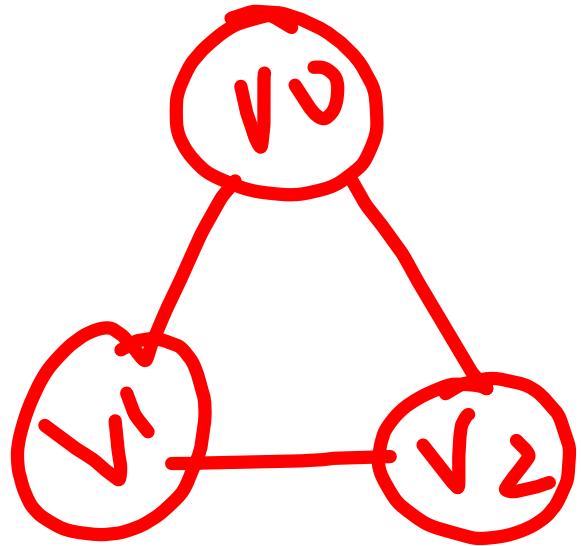




every tree is a graph  
but every graph is not tree.

a graph without cycle can be called tree



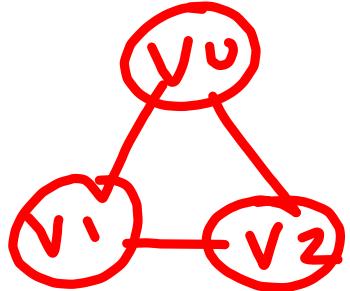
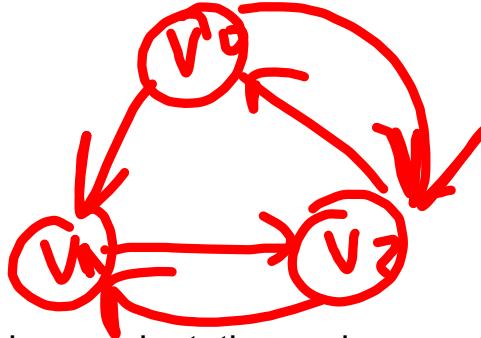
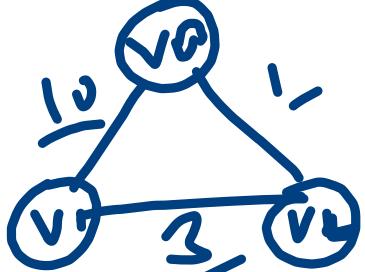
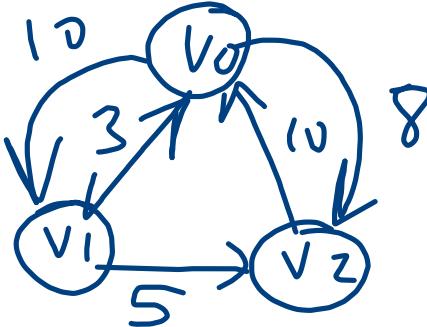


$G(V, E)$

$V = \{v_0, v_1, v_2\}$

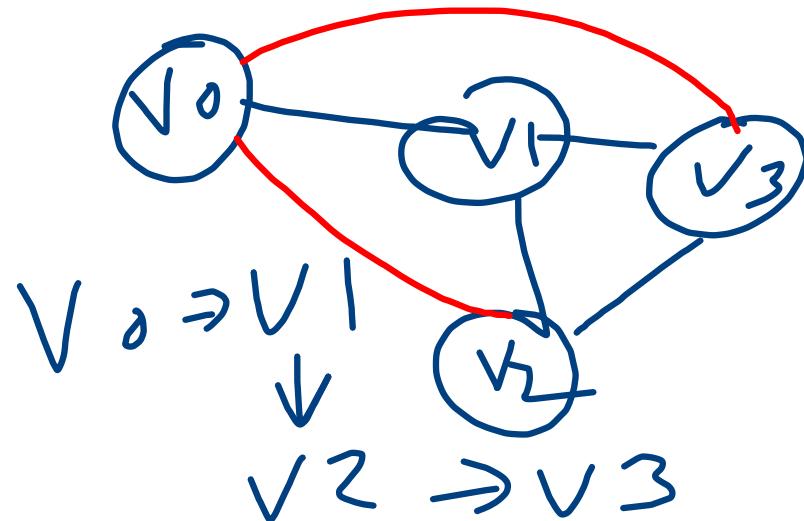
$E = \{(v_0, v_1), (v_1, v_2), (v_2, v_0)\}$

# Types

	UNDIRECTED	DIRECTED
UNWEIGHTED	 <p>basic connectivity</p>	 <p>shows orientation and connectivity</p>
WEIGHTED	<p>any unit that can be used to take decisions</p> 	

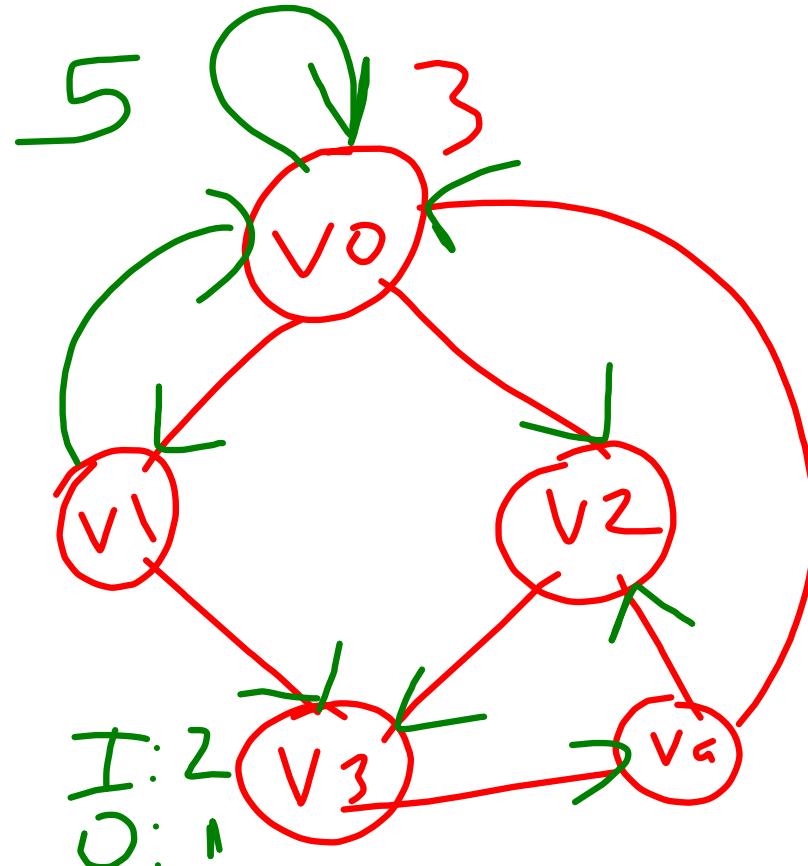
# Terminologies

- **Adjacent nodes:** two nodes are adjacent if they are connected by an edge
- **Path:** a sequence of vertices that connect two nodes in a graph
- A simple path is a path in which all vertices, except possibly in the first and last, are different.
- Complete graph: a graph in which every vertex is directly connected to every other vertex.
- A cycle is a simple path with the same start and end vertex.

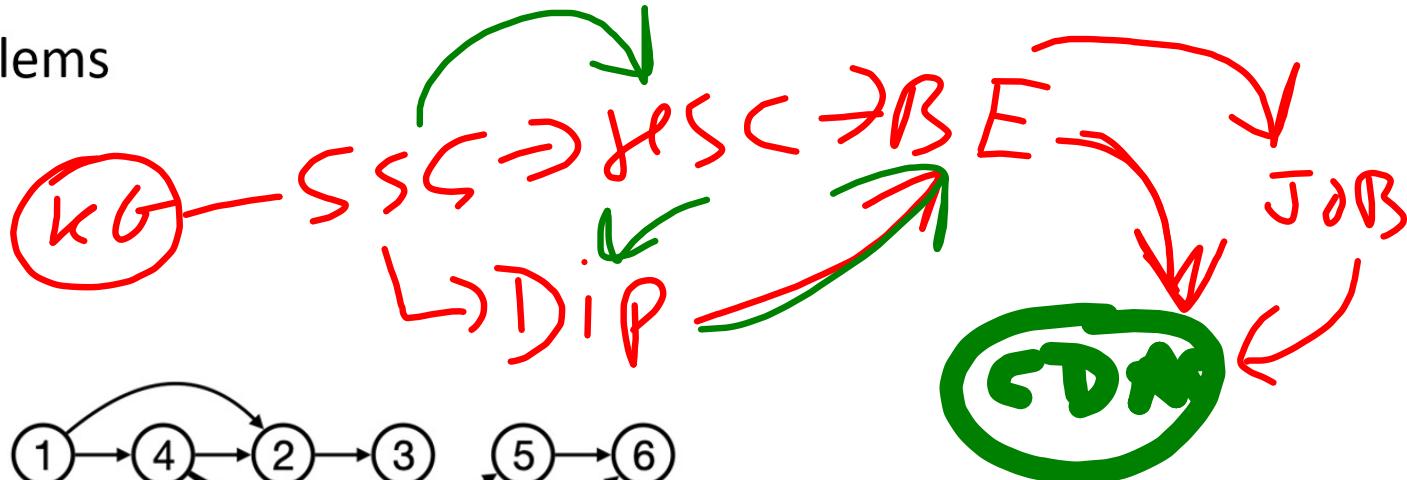
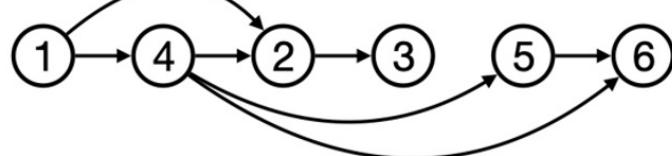
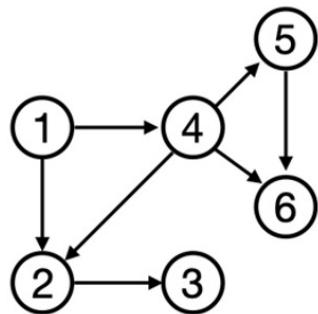


# Terminologies

- A cycle is a simple path with the same start and end vertex.
- The degree of vertex / is the no. of edges incident on vertex.
- Directed graphs are strongly connected if there is a path from any one vertex to any other.
- Directed graphs are weakly connected if there is a path between any two vertices, ignoring direction

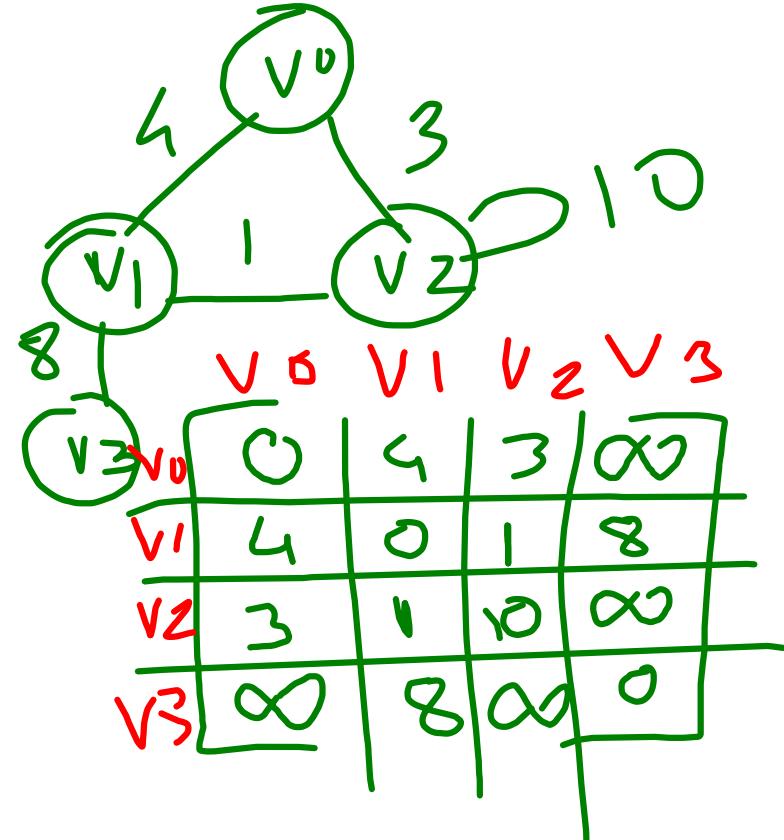


- Handling flow problems
- Topological sort



# Graph representation

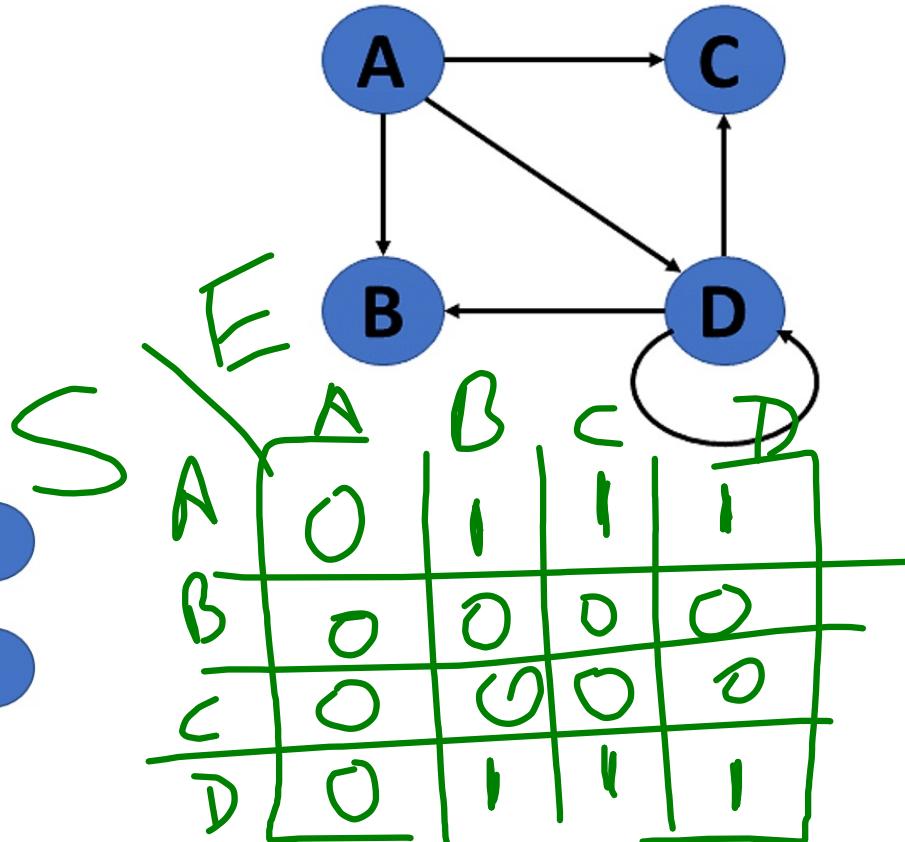
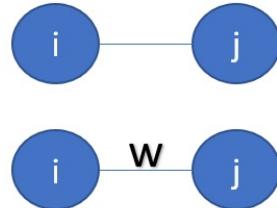
- Adjacency matrix
  - $V \times V$  matrix(storage of  $v \times v$ )
  - Static
  - Easy
  - To fill if unweighted
  - $\text{AdjMat}[i][j]=1/0$
  - To fill if weighted
  - $\text{AdjMat}[i][j]=w/\infty$



# Graph representation

- Adjacency matrix

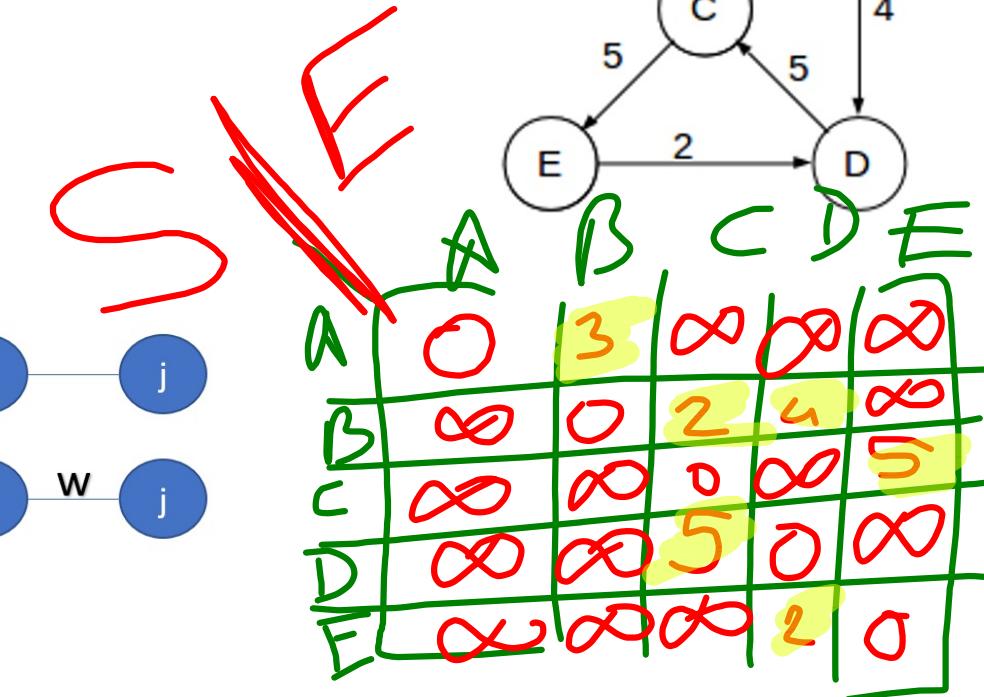
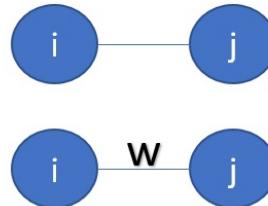
- $V \times V$  matrix
- Static
- Easy
- To fill if unweighted
- $\text{AdjMat}[i][j]=1/0$
- To fill if unweighted
- $\text{AdjMat}[i][j]=w/\infty$



# Graph representation

- Adjacency matrix

- V x V matrix
- Static
- Easy
- To fill if unweighted
- AdjMat[i][j]=1/0
- To fill if unweighted
- AdjMat[i][j]=w/ $\infty$

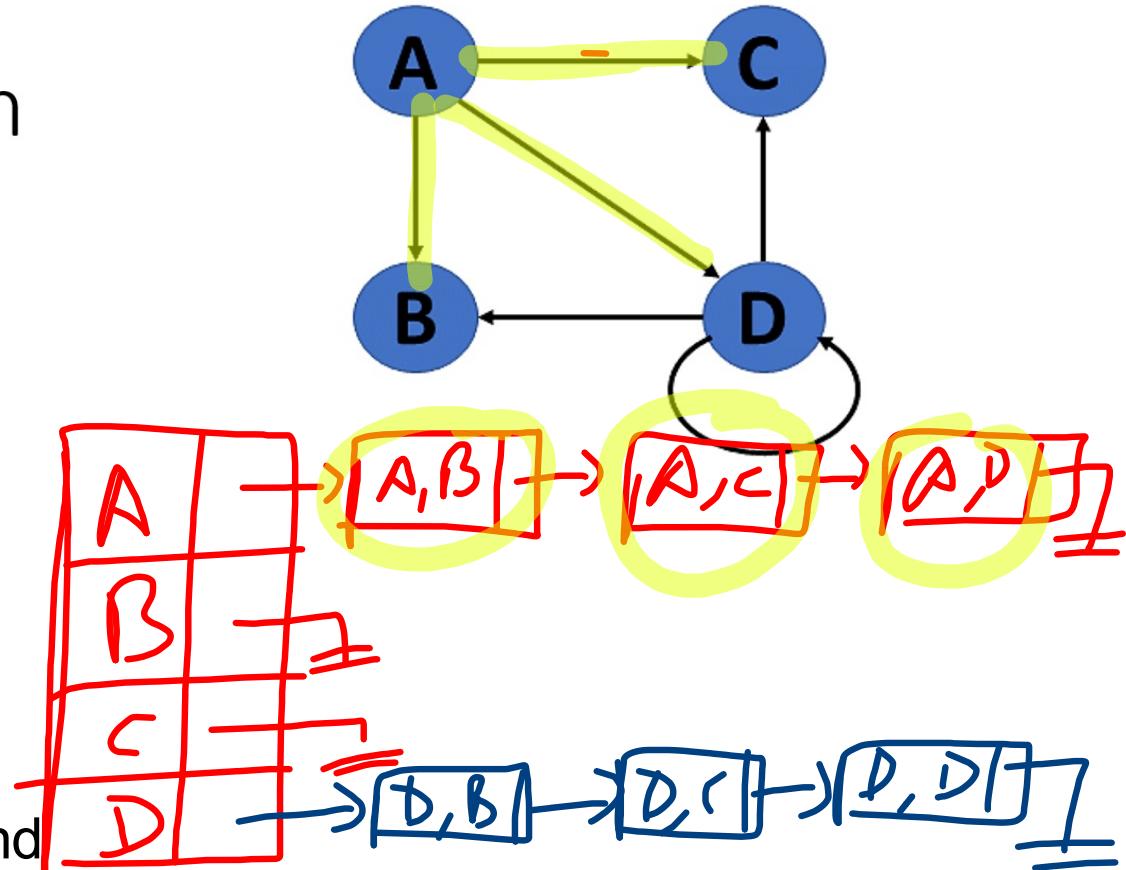


# Graph representation

- Adjacency List
  - Array of linked list
  - Each node is an edge
  - Can grow or shrink with need
  - Storage of  $(v+e)$

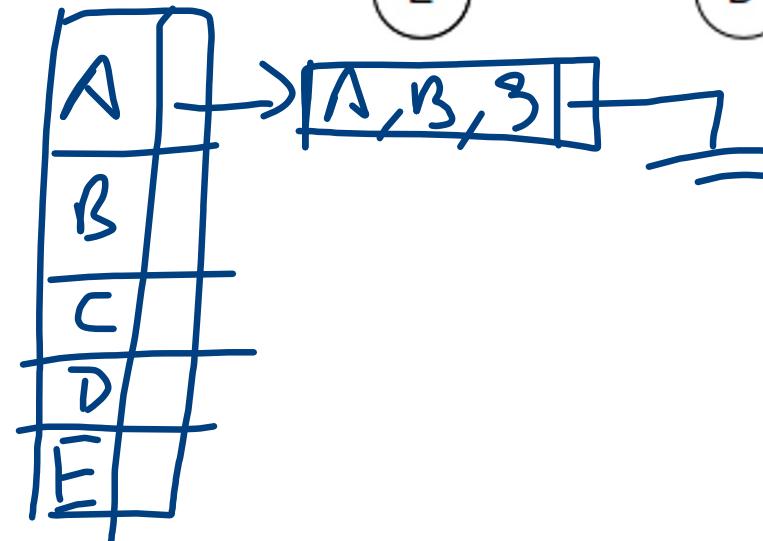
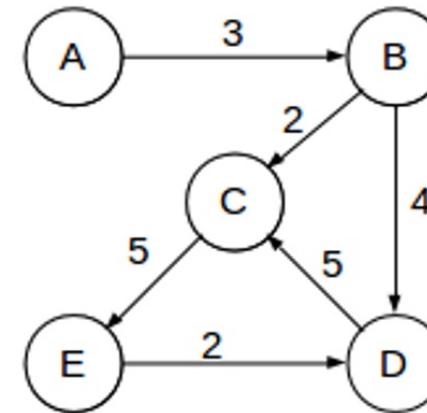
+ve: dynamic and uses space effectively

-ve: too complex to code and use



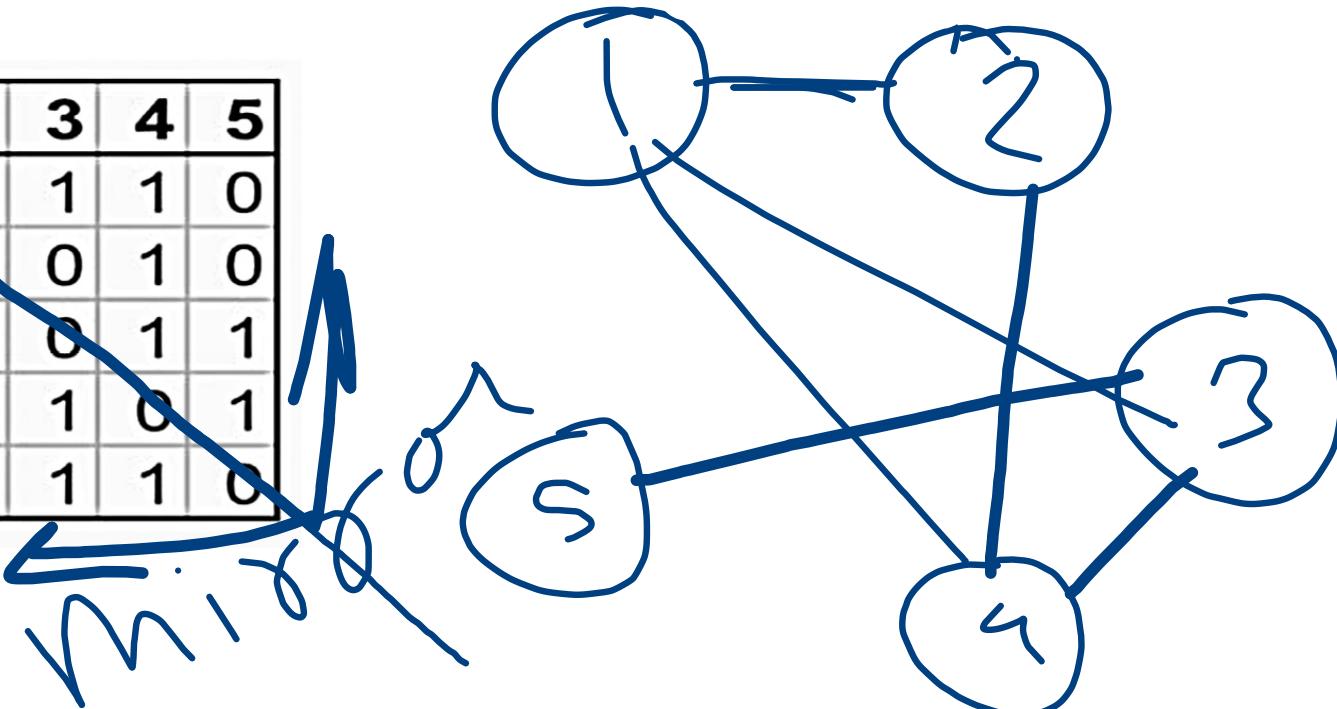
# Graph representation

- Adjacency List
  - Array of linked list
  - Each node is an edge
  - Can grow or sharing with need



# Create a Graph

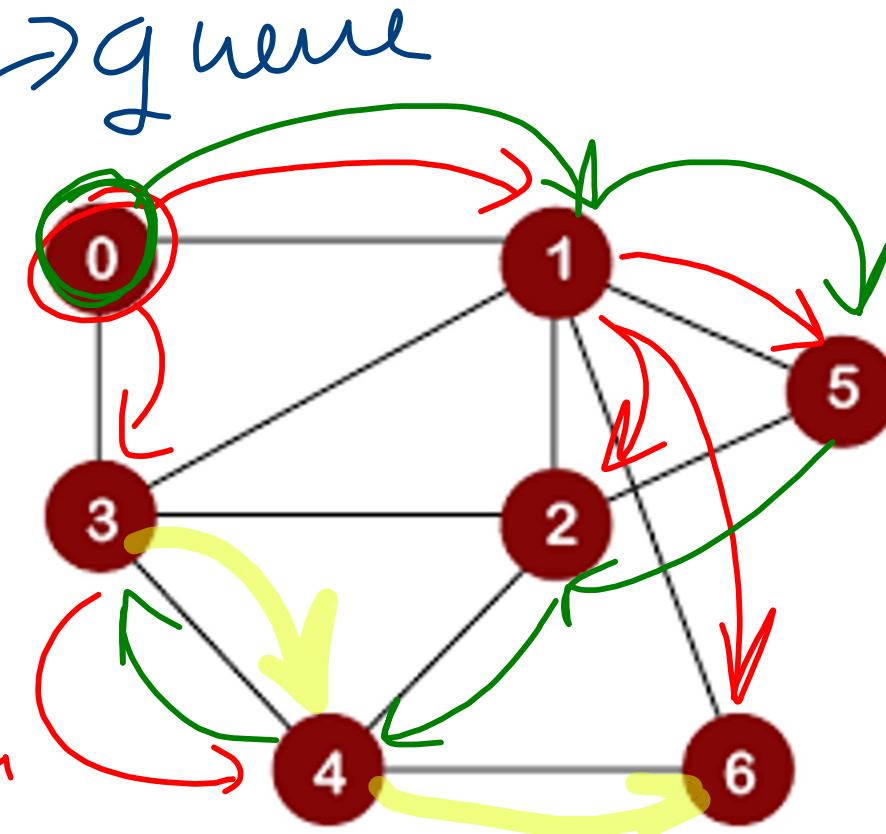
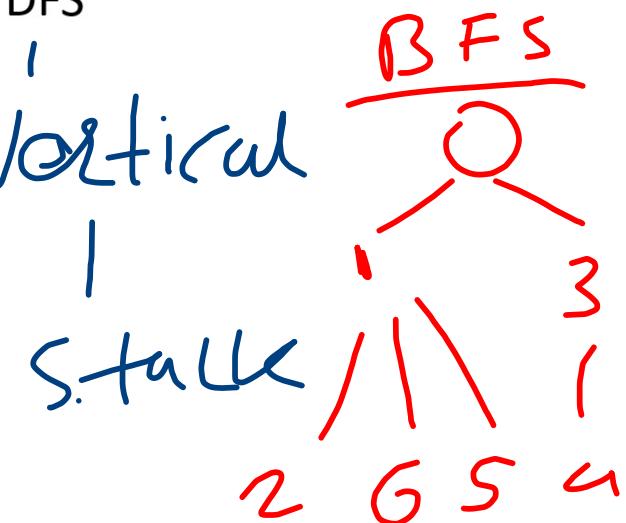
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	0
3	1	0	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0



# Graph Searching

- BFS  $\rightarrow$  Horizontal Fan
  - DFS

Vertical

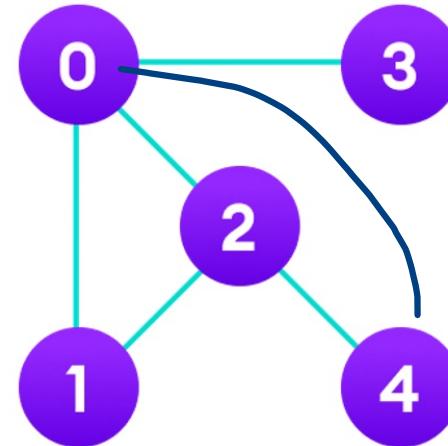


DFS

O → S → 6 → Q-3

# BFS

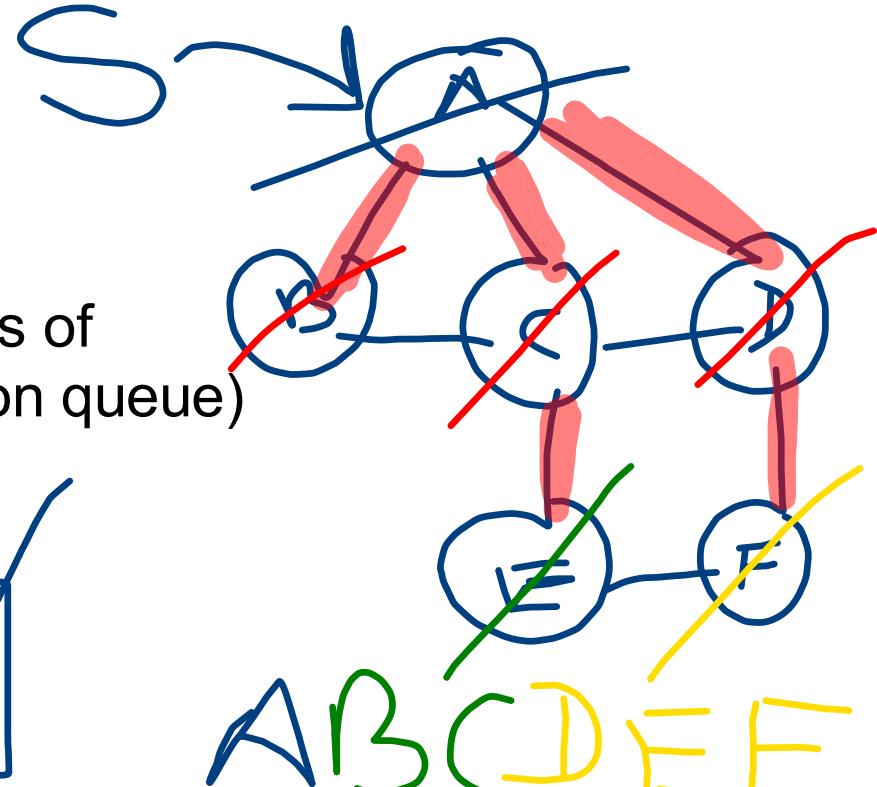
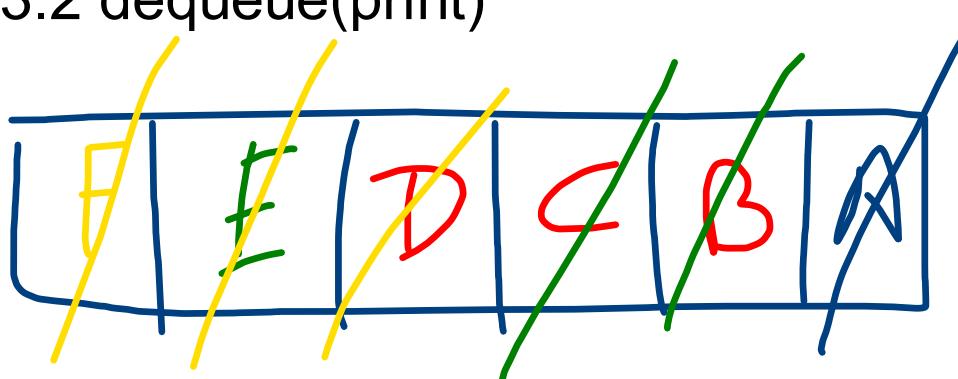
1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.



BFS:(queue)

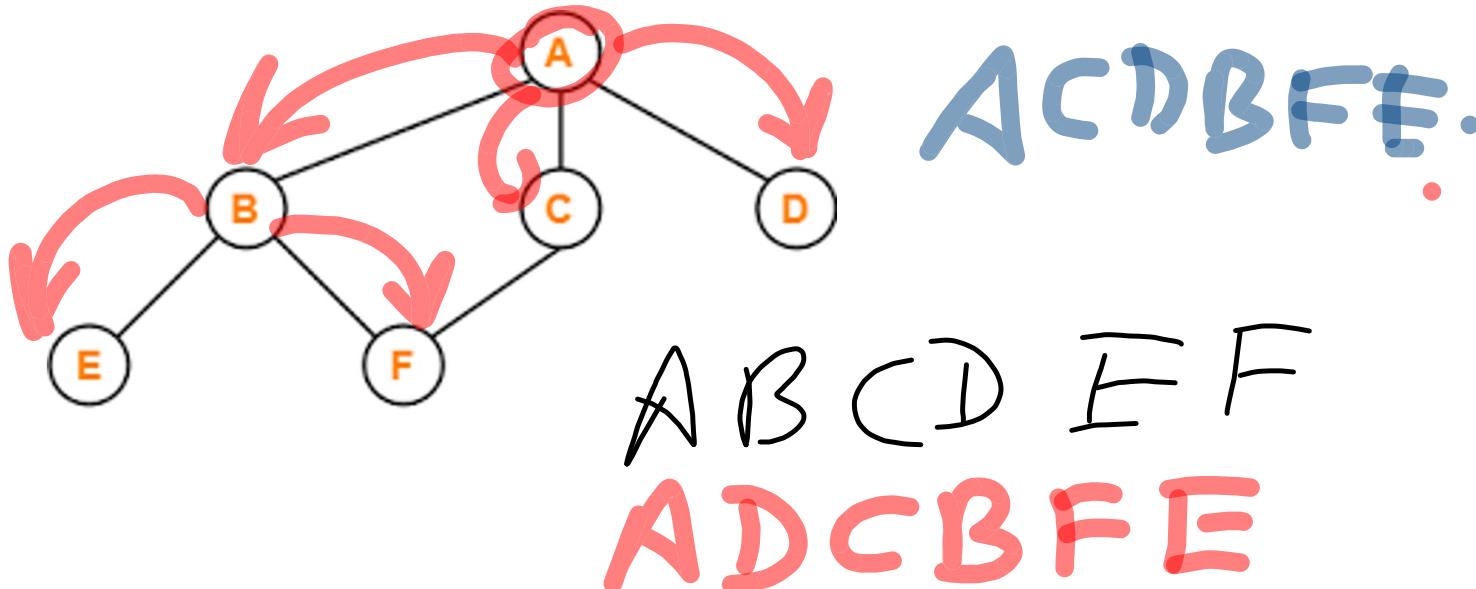
- 1.accept source
- 2.mark source visited and enqueue
- 3.Till queue not empty

- 3.1 search and insert all neighbours of  
q[front] in queue(if not already on queue)
- 3.2 dequeue(print)



## BFS

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.



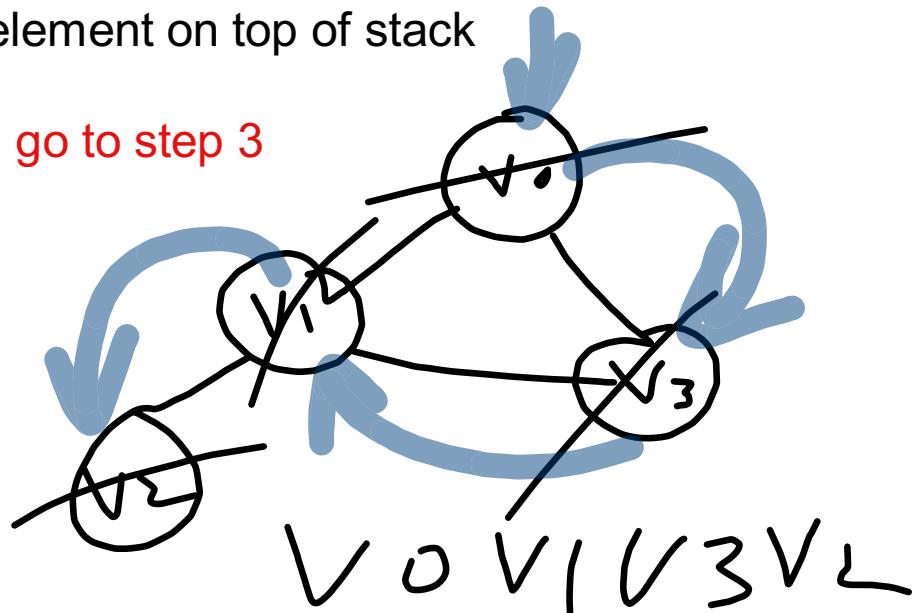
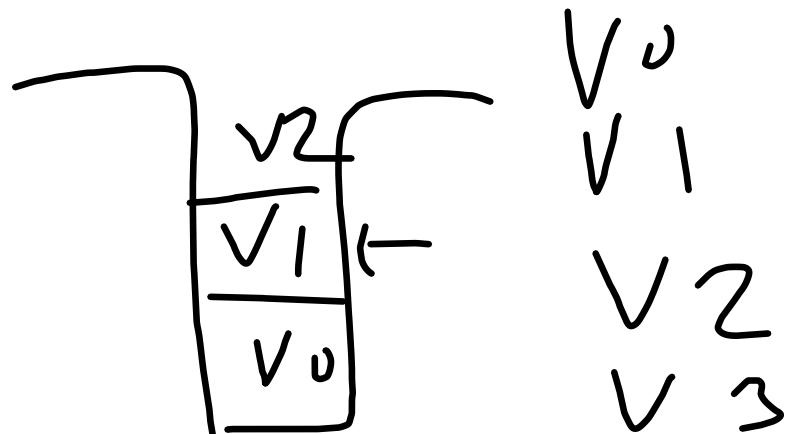
DFS:(stack)

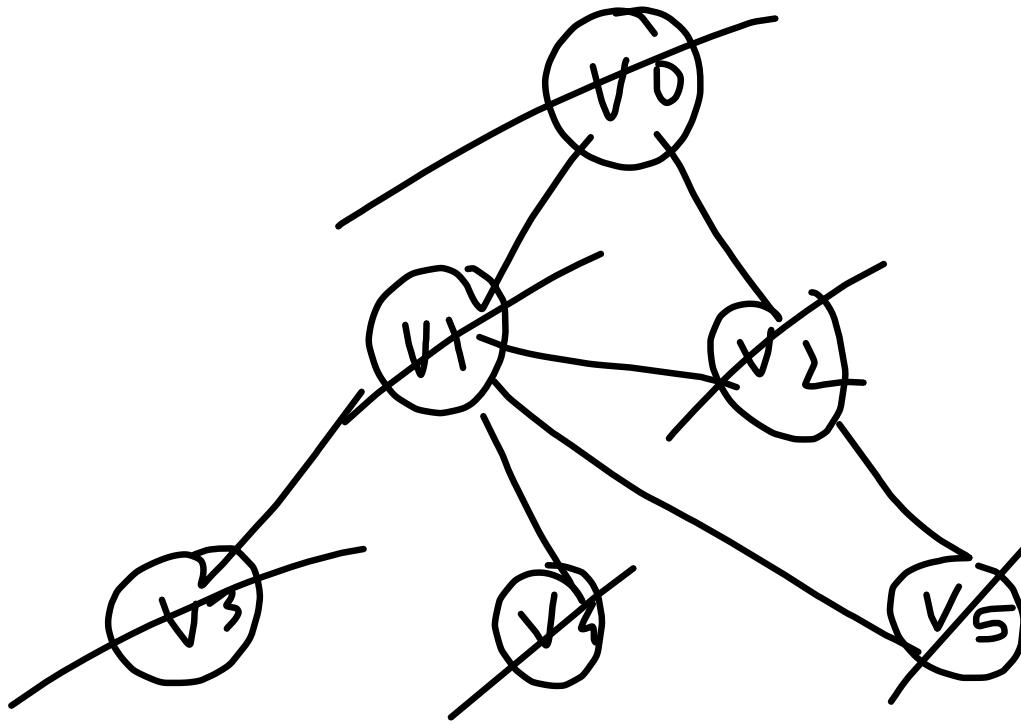
- 1.accept source
- 2.mark visited push on stack
- 3 till stack not over

3.1 search any one unvisited neighbour of element on top of stack

3.2 mark it visited & push on stack

**3.3 if no unvisited neighbour found pop and go to step 3**





$v_0 v_1 v_2 v_5$   
 $v_3 v_4$

DataStructures - NetBeans IDE 8.2

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

Search (Ctrl+I)

GraphDemo.java StackDemo.java QueueLinearDemo.java CircularQueueDemo.java PriorityQueueDemo.java LinkedListLinear.java DynamicStackDemo.java DynamicQueueDemo.java Circular...

Source History

```

public void DFS(int source)
{
    visited[source]=1;
    System.out.println("V"+source);
    for(int i=0;i<v;i++)
    {
        if(g[source][i]==1 && visited[i]!=1)
            //neighbour and unvisited
        {
            DFS(i);
        }
    }
}

```

DFS Tracing:

DFS search(int source int key)

Output

53:8 INS

Toolbar icons: Pencil, Circle, Line, Arrow, Hand, Magnifying glass, Pen, AI, Scissors, Keyboard.

DataStructures - NetBeans IDE 8.2

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

Search (Ctrl+I)

Source History

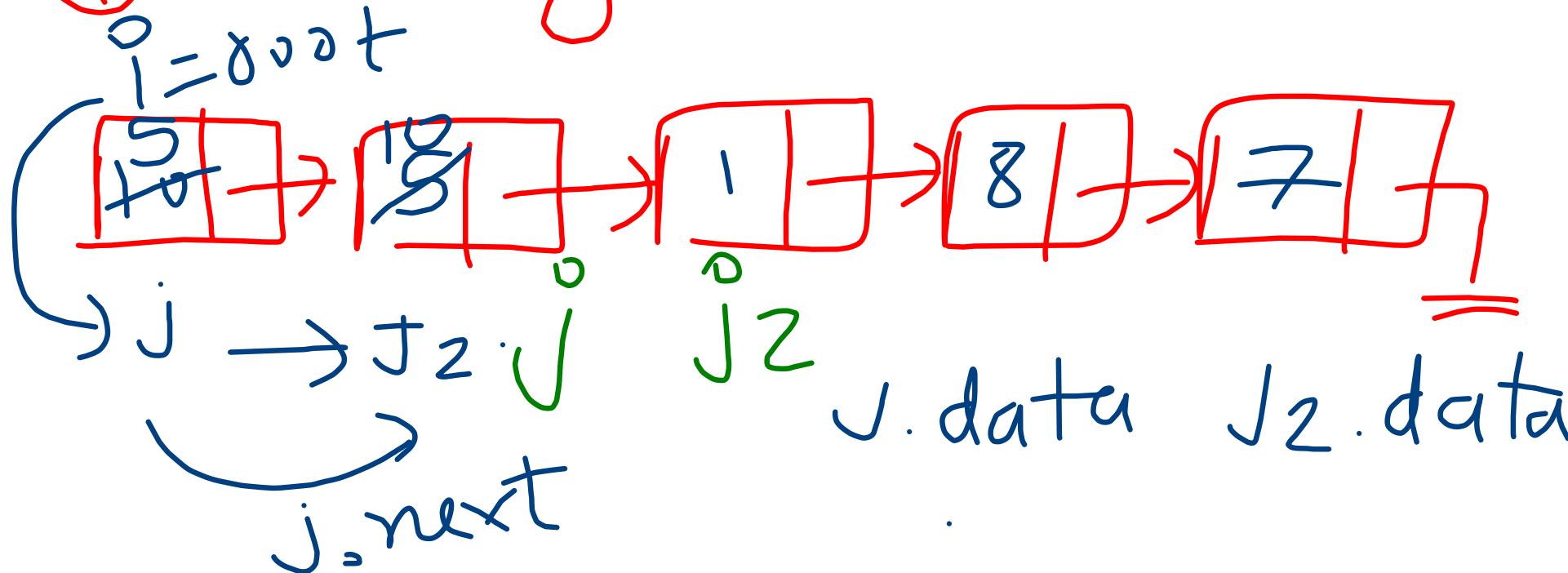
```

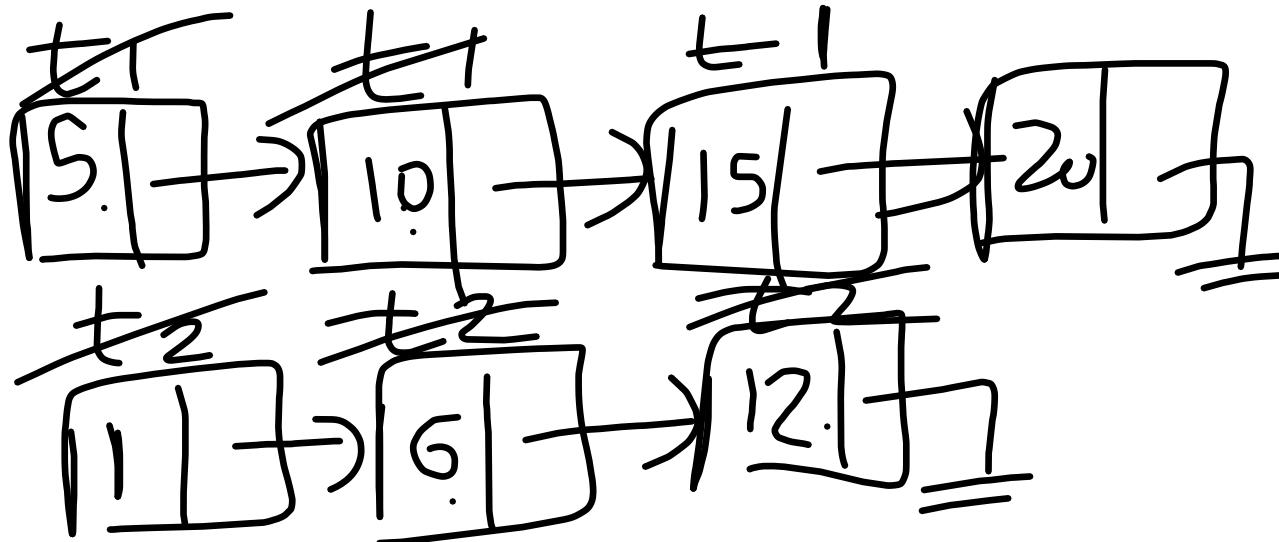
89 q[++rear]=source;//enqueue
90 while(front<=rear)//not empty
91 {
92     int element=q[front++];//dequeue
93     System.out.print("V"+element+"-");
94
95     for(int i=0;i<v;i++)
96     {
97         if(g[element][i]==1 && visited[i]!=1)//neighbour and unvisited
98         {
99             visited[i]=1;//visited
100            q[++rear]=i;//enqueue
101        }
102    }
103 }

```



# ① Sorting LinkedList





Soft  
merger



