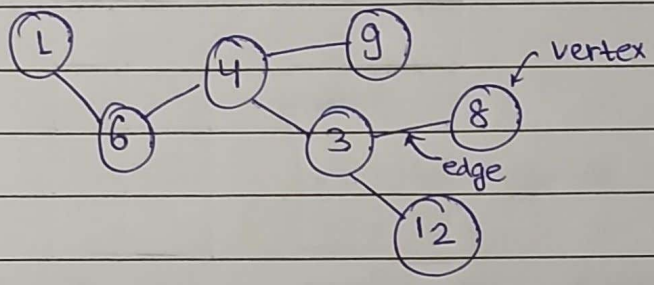
Date 02/08/24
Page

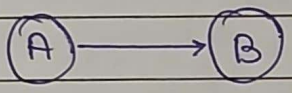
GRAPHS

→ network of nodes



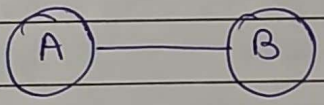
- Edges :-

(1) Uni-Directional / Directed Graph



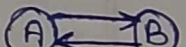
(we can't go from B to A)

(2) Undirected Graph

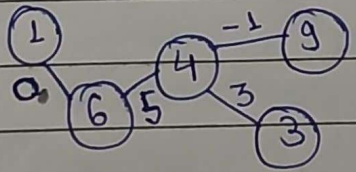


can go from B to A
or A to B

also represented as Bidirectional Graph ⇒

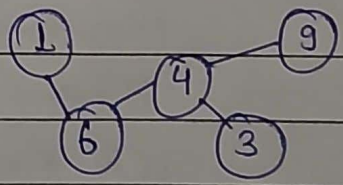


Note :- every edge has a property associated with it called Weight.



(0, 5, 3, -1) ≡ weights

Weighted Graph
(un-directed)



Unweighted graph

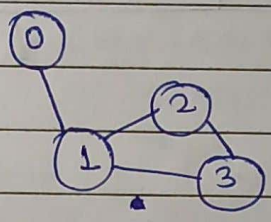
Date
Page

Storing a Graph
- Adjacency List
- Adjacency matrix
- Edge List
- 2D matrix (implicit)

Defining a structure / representing a Graph.
like Linked List and Binary Trees represented by making class Node{ data, left, right }
this also directly comes under Java Collections Framework.

⇒ I. Adjacency List

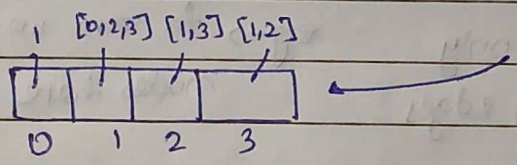
List of Lists



✓ vertices = 4 (no. of nodes)
✓ edges = 4
✓ unweighted,
✓ undirected.

0 → (1) → List[0]
1 → (0, 2, 3) → list[1]
2 → (1, 3) → List [2]
3 → (1, 2) → List [3]

can be stored as :-
- ArrayList<ArrayList> ← check codes of same (earlier)
- Array<ArrayList>
- hashmap<int, lists>
  - vertex (key)
  - value

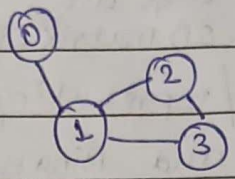


Benefits :-
~~(1)~~ extra info.
(2) optimised
✓ find your neighbour O(K) e.g. (0,2,3) elements in 1 (Array<ArrayList>)

✓ Array of ArrayList<Edge>
- source
- destination
- weight → by default (1) ← we're letting this by own.

✓ e.g. edge b/w 1 & 3 has → source = 1
// destination = 3
weight = 1 (taking as default)

II. Adjacency matrix



V = 4

E = 4

∴ space req for making adjacency matrix → $O(V^2)$

more than adjacency list

	0	1	2	3
0	0	1	0	0
1	1	0	1	1
2	0	1	0	1
3	0	1	1	0

→ You know how this is filled

problem checking one no. 2 times like 0 with 1 in 1st row then again 1 with 0 in 2nd row

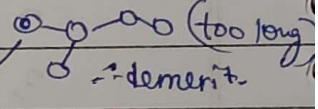
in list we were getting $O(K)$ space complexity

↓ no. of edges

∴ time complexity $O(1)$

but here $O(V)$

↓ no. of vertices

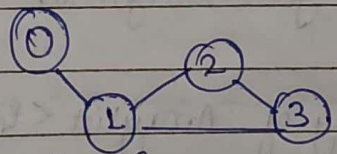
for like this →  (too long) ∴ demerit.

e.g. → for zero we're only checking no. of its edges

e.g. → for zero we are checking all nodes here.

Benefit :- we can store weighted info. also in matrix.

III. Edge list

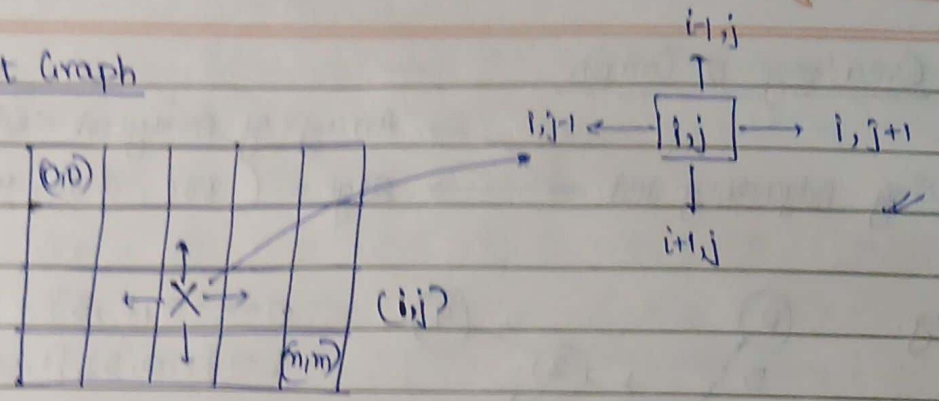


Edges = { {0,1}, {1,2}, {1,3}, {2,3} }
e1 e2 e3 e4

also store weights → { {0,1,4}, {1,2,0}, {1,3,-4}, {2,3,5} }
e1 e2 e3 e4

e.g. Min. Spanning Tree → where we sort edges using edge list it becomes easier.

Implicit Graph

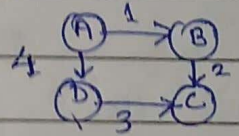


Applications of Graphs

a. Shortest path in MAPS. eg. going from A to C (feasible?)

↓

Djikstra, Bellmann Ford

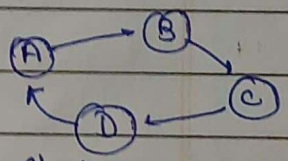


∴ A→B→C = 3 ✓

A→D→C = 7 ✗

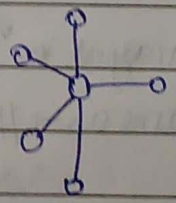
b. Implement connection b/w profiles (SOCIAL NETWORK)

c. Shortest cyclic route (DELIVERY NETWORK)



Shortest cyclic route to deliver at B, C, D & return.

d. Expressing molecules (Phy. + Chem)



e. Routing algorithms (fastest data transfer)

f. Machine learning (computation Graphs)

↳ to explain neural networks for calc.

g. Dependancy Graph windows → vs code → extension → better code

(without windows no further process ahead like vs code)

h. Computer Vision for Image segmentation.

detecting face, eyes, etc.

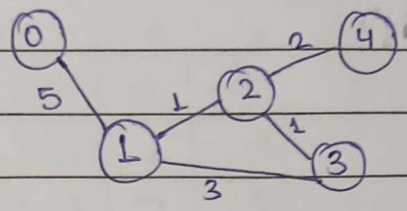
i. Graph databases.

j. Research (human neurons connections)

Creating a Graph

using Adjacency list → Array of ArrayList <edges>
→ Edge = (src, dest, weight)

e.g.



0 → {0,1,5}
1 → {1,0,5}, {1,2,1}, {1,3,3}
2 → {2,1,1}, {2,3,1}, {2,4,2}
3 → {3,2,1}, {3,1,3}
4 → {4,2,2}.

Note:- vertex & source will always be same.

Code ⇒

```
~~public~~ class Edge {
    int src;
    int dest;
    int wt;

    constr of all
};

psvm {
    int V = 5;          V = vertices

    array of arraylist ↔ ArrayList<Edge> [] graph = new ArrayList[V];
        ↳ [same as → // int[] arr = new int[V];
                        ↳ array of integer ]
```

initially, Arraylist if all elements are null so we've to convert it into empty arraylist.

Basically adding Arraylist to all index of array.

```
    for (int i = 0; i < V; i++){
        graph[i] = new ArrayList<>();
    }
```

creating empty arraylist to all index.

adding edges info at vertices →

```
// 0th vertex                    src dest. wt.
graph[0].add(new Edge(0, 1, 5));
// 1 vertex
" [1] " "  (1, 0, 5));
:
// 2, vertex, 3, 4.
```

neighbours

```
// 2's neighbours :-        where 2 neighbour is stored
for (int i=0 ; i< graph[2].size(); i++){
    Edge e = graph[2].get(i); // src, dest, wt.
    sout(e.dest);
}
```

inside every arraylist ← edges are stored

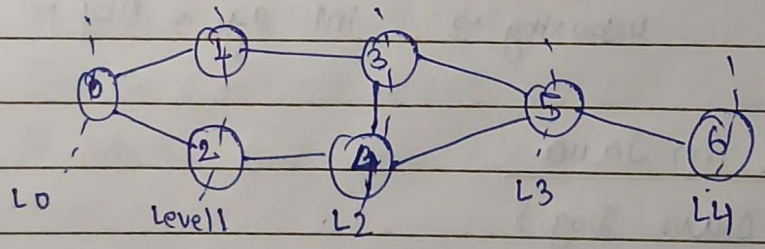
because destinations are the neighbours of vertex.

• Graph Traversals

1. BFS (Breadth First Search)
2. DFS (Depth " ")

⇒ BFS

↳ Go to immediate neighbours first.



∴ it is also called level order traversals (Breadth ⇑ travel karo)

↳ can't access 3 before 2 after 1.

" " 5 " 3, 4.

```
node.next = insertRec(val, indx--, node.next);
return node;
}
```

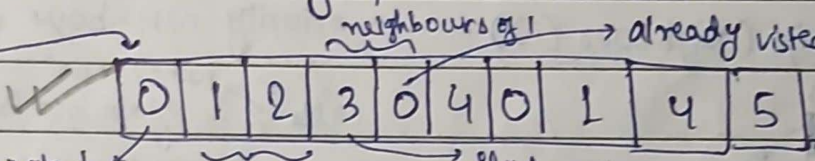
→ write before : `public void insertRec(int val, int indx){ head = insertRec(val, indx, head); }`

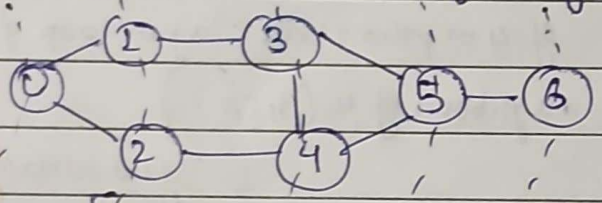
BFS continued...

Make a boolean type array and whenever we found an element make it true.

- visited :- 1. vis[current] = True
  2. print it
  3. neighbours add queue.

in the answer array put 1st element at 0th index





put true for 1 & 4 check for future

ely for 2.

(neighbours of 1) Don't put true for 3 right now

level order traversals

Hence, queue (FIFO)

q.remove() → current node

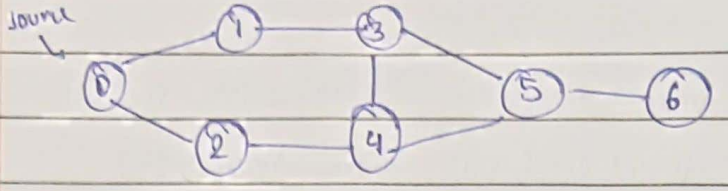
if(!visited[current])

visited[current] + print(current)

popularity.ai

* DFS (Depth first search)

directly jump into code



0 → 1 → 3 → 4 → 2 → 5 → 6

∵ 0 → 1 (0,3) → 3 (4,1,5) → 4 (2,3,5) → 2 (0,4) → 5 (3,4) → 6 ✓

→ Priority on the basis of neighbour (not ~~as it~~ on basis of levels).

recursion is implicit stack

recursion (implicit stack) :- dfs (graph, curr) → e.dest → neighbour

① current → visit ⟶ ie. print + vis[curr] = true.

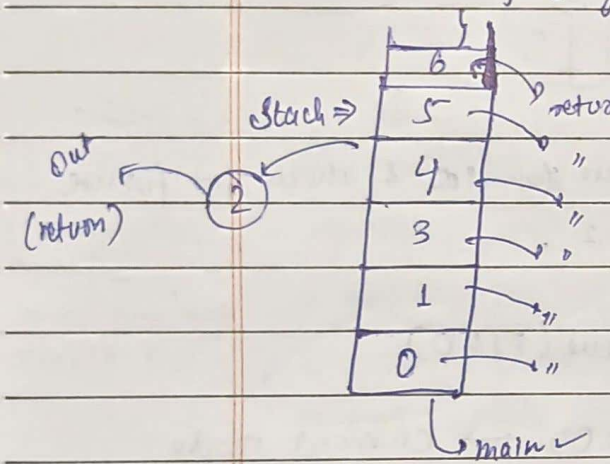
② for (int i=0 to k) ⟶ for neighbours (k = total no. of neighbours)

if (!vis[curr]){ → not visited

dfs (neighbour (i)) ⟶ ∵ priority neighbour wise (socho)

calls 2 ∴ print + true ∴ again loop for neighbours of 2 ie. (0,4) → ∴ again loop for neighbours of 4 → print + true ∴ again loop for neighbours of 4 (3,5,2) - - - -

∵ true already dala hua hai ∴ visit only if curr = false



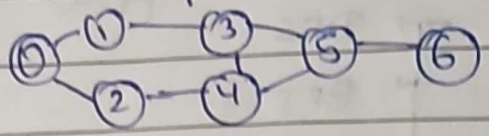
∴ [print → 0 1 3 4 2 5 6] ⇒ agar kisi of neighbour already visited hai toh out from stack.

* Code :- static void dfs (ArrayList<Edge> [] graph, int curr, boolean vis[]){

```
sout(curr + " ");
vis[curr] = true;
for(int i=0; i< graph[curr].size(); i++){
  Edge e = graph[curr].get(i);
  if (!vis[e.dest]) {   → [e.dest] i.e. neighbour visit tha ya nhi
    dfs(graph, e.dest, vis);
  }
}
```

Has Path?

↳ to check if 5 is also in path with ~~0~~ 0

in 

⇒ hasPath (graph, src, dest, vis) {

(i) src == dest
True

(ii) visit src

(iii) if (!(vis[neigh]) && hasPath (neigh, dest)) → (neighbour is not visited) and (neighbour is e.dest) ∴ true

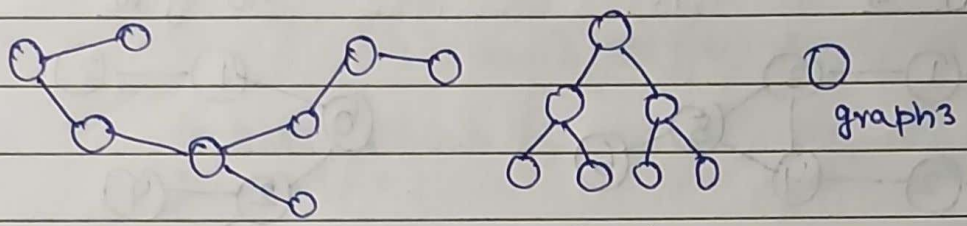
True

}

else false

04/09/24

Connected Components

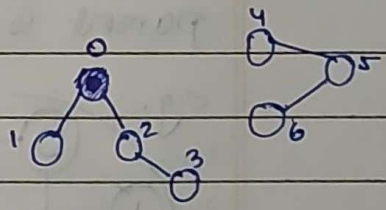


Combining graph 1, 2, 3 together is also a graph with multiple connected components.

If we connect them (1 & 2) via 1 edge ∴ connected component will be 1.

DFS

vis[] → boolean → initialize all with F. eg.



for (int i = 0; graph.length) {

if (! vis[i] → using this we can visit from 0 to 6 etc.

dfs util

}

↳ util name is convention for helper func.

dfsutil {

DFS

}

} helper function → you know what is an helper func.

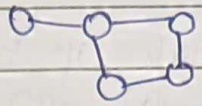
→ concept of subpart for main func.

e.g. swap func. is a helper function.

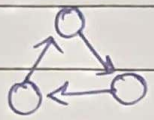
∴ code for BFS as well as DFS.

Date 07/09/24
Page

* Detect Cycle in Graph



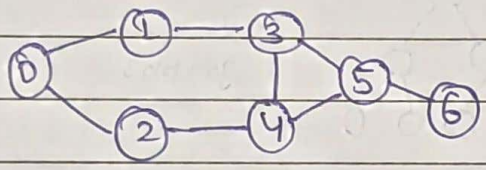
Undirected → DFS
→ BFS
→ DSU (Disjoint set Union)



Directed → DFS
→ BFS
→ Topological sort (Kahn's Algo.)

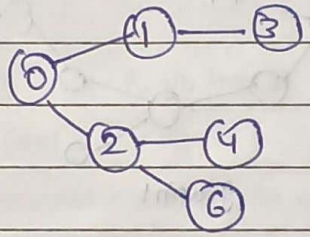
I. Undirected Graph

cycle



True

no cycle

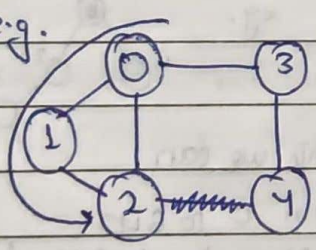


false

Condition: when the neighbour of any node is already visited.

parent is the node through which current comes.

e.g.



here 1 is the parent of 2
0 " " " " 1.

∴ case 1: vis[neigh] ✓ i.e. neighbour already visited
parent ✗ but not its parent

↓

always cycle exist
∴ True

case 2: vis[neigh] ✓ parent ✓ → not necessary ∴ continue
cycle exist

case 3: vis[neigh] ✗ → normal DFS → call (visit)
if cycle → ∴ True.

Since, we're starting from 0 ∴ let its parent be -1.

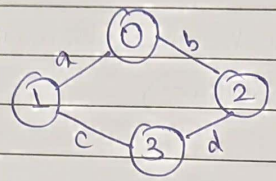
~~detectcycle~~dfsUtil(graph, vis, curr., parent)
vis[curr] = true;
for(i=0 to graph[curr].size()){
edge e = graph[curr].get(i);
case 1 ← if(vis[e.dest] && e.dest != parent){ → neighbour
return true;
}
case 3 ← else if(!vis[e.dest]){ if(detectcycleUtil(graph, vis, edest, curr)
return true; → socho-easy. (if we break neighbour)
}return true;
}// do nothing for case 3
return false; → cycle does not exist..

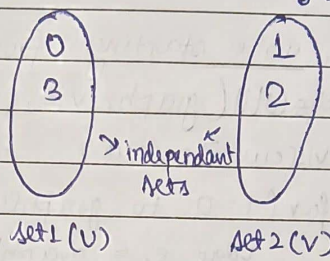
↓
check [for removing edge still true].

* Bipartite Graphs

for every edge (u,v) either u belongs to U and v belongs to V or u belongs to V and v to U i.e. there is no edge that connects vertices of same set. U or V

e.g.1





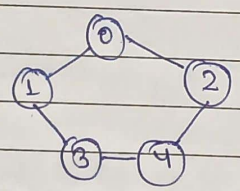
Koi bhi ek aisa edge utha lo jiske connecting nodes ek set me nhi hone chahiye for whole graph then that graph is called Bipartite.

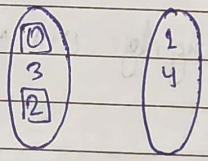
here, c → 1 and 3 | b → 0 and 2 | a → V | d → V

(1 → V, 3 → U, 0 → U, 2 → V)

Since, this graph can be divided into two sets with condition* ∴ It is a Bipartite graph

e.g.2

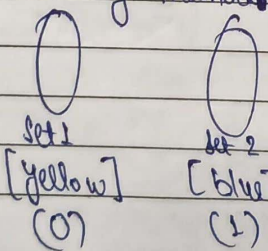




∴ Not Bipartite Graph.

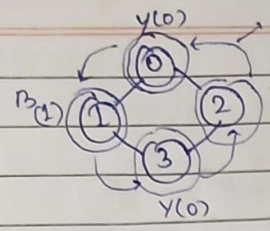
- How to check ?

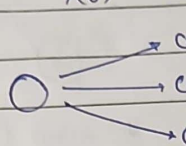
⇒ Coloring method :-



(-1) → no colour

neighbour is opposite colored ∴ Bipartite graph ✓



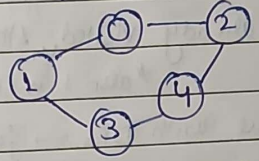


col[] = | 0 | 1 | 1 | 0 | 0 |
0 1 2 3 4

→ Non-bipartite.

0 → yellow
1 → blue
-1 → none

T.C. → O(V+E).

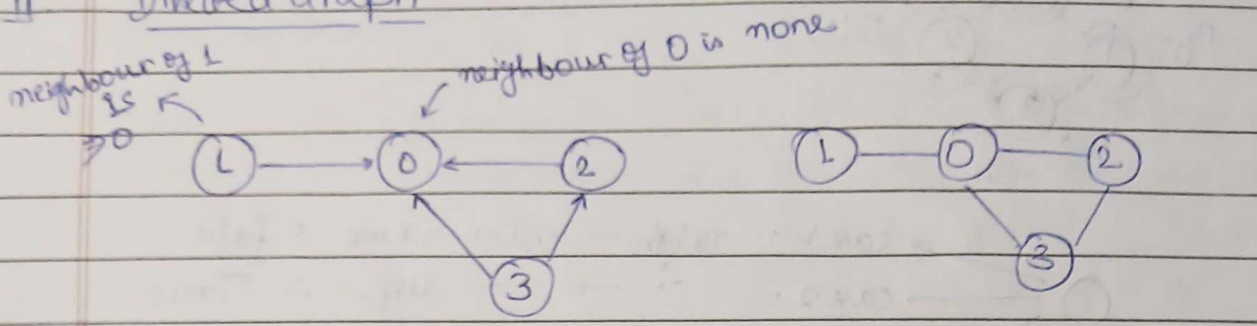


0 ? 1 : 0
↓
ternary operator

→ mtlb if 0 then next one will be 1 & if not 0 i.e. 1 then next one will be 0.

08/09/24

II Directed Graph



Directed (no cycle)

Undirected (Tree)

↓

Why undirected DFS approach fails?

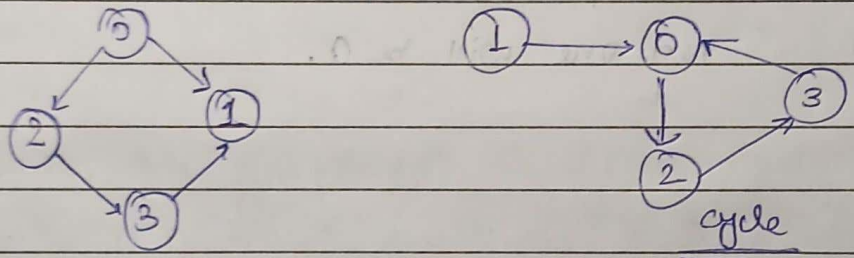
Here → parent of 1 is -1 & neighbour is 0

Since 0 is already visited, still no cycle exists.

also 1 is not parent of 0

Hence, parent with DFS → fails → [neighbour → vis. → not parent]

e.g.



no cycle

cycle

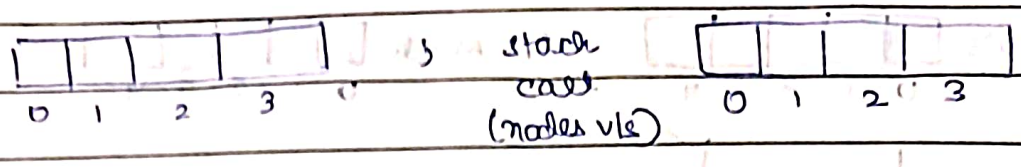
- **Approach:-**

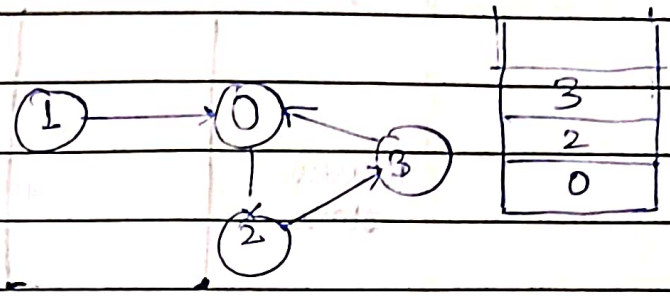
We know the basic pseudocode for DFS:

```
dfs(current){
  vis[current] = true;
for(all neighbours){
  if(!vis[neigh])
     visit ✓
}
```

Modification:-

We'll make a stack which tracks the record the current neighbours unlike boolean visited array.

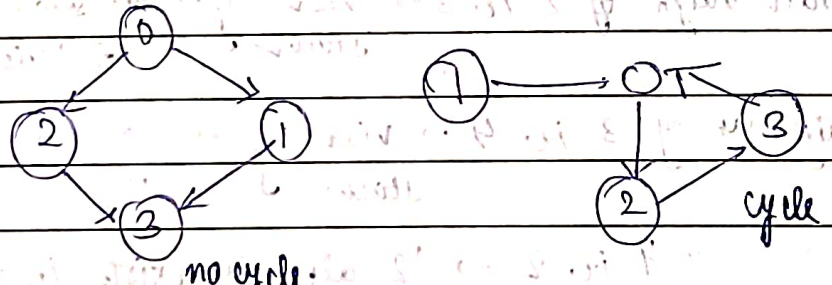




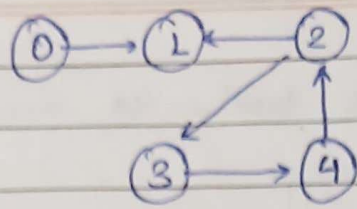
- Modified Pseudocode :-

```
dfs (curr){
    vis[curr] = true;
    stack[curr] = true;        → making this also true to show ki wo call ☆ under aa chuka hai
    for (all neighbours){      → graph.length
    if (stack[neigh] = true){  → to check usi pehle hi ~~if~~ stack ☆ aa gya hai.
∴ cycle exist → True           (here 0 pehle hi stack ☆ aa chuka hai for 3).
~~if{~~
    if (!vis[neigh])           → if neighbour is not visited
        isCycle (neigh) → True ~~if cycle~~ → if not visited then make it True.
    ~~if not → no cycle~~
    stack[curr] = false        → stack ☆ upar upar jaate waqt jab true kiya tha waise hi niche (pop) aate-aate false kr dena.
}
```

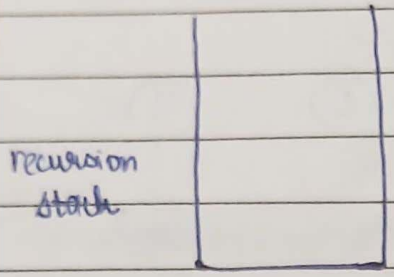
Dry Run karke dekho for :- (or refer video)



Example →



vis [ ][ ][ ][ ][ ] ; stack [ ][ ][ ][ ][ ]
0 1 2 3 4 0 1 2 3 4



1. start from 0 → vis ✓ stack ✓ } → true.

2. 0 → checks ki stack ka neighbour exist karta hai ya nhi? → No (∵ saare neighbour abhi unvisited hai)
Therefore visit kardo.

3. 1 → vis ✓ stack ✓ } true

4. No neighbour of 1 → return to 0 → 1 removes from stack
∴ 1 → false → no neighbour now of 0 also ∴ → 0 → false.

5. Now start from 2 → vis ✓ stack ✓

6. check for neighbour of 2 i.e. 1 → already visited! but not in STACK → ∴ no importance of 1 → ∴ don't take any action on 1.

7. check for next neigh. of 2 i.e. 3 → vis ✓ stack ✓ } ∵ unvisited tha i.e. ⇒ !vis[e.dest]

8. " " neighbour of 3 i.e. 4 → vis ✓ stack ✓ } ∵ " "
" "

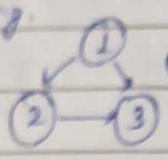
9. " " " " 4 i.e. 2 → '2' already exists in the stack
Hence, cycle exists.

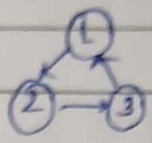
Time Complexity: O(V+E)
↳ ∵ DFS.

extra mad: stack Overflow → why DFS and not BFS for cycle ~~detection~~ finding in graphs?

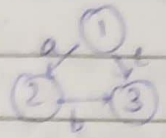
* Topological Sorting

→ works for DAG (directed Acyclic graphs) only.

eg.  ■ DAG ✓

eg.  DAG ✗ ∵ cycle

→ It is a linear order of vertices such that every directed edge $u \to v$, then vertex u comes before v in order.

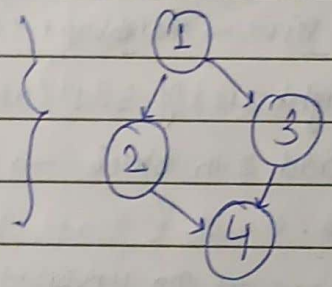
ie. 

⇒ a: 1 → 2 (u → v)
b: 2 → 3
c: 1 → 3

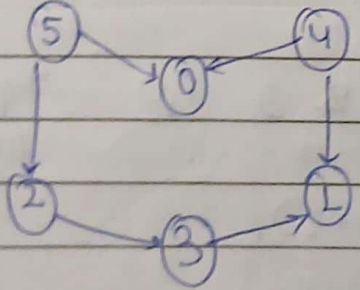
★ Dependency ★ → Konsa kaam kab perform hoga since konsa kaam kisse depend karta ✓ → matlb konsa kaam kis kaam ✓ bina nhi ho sakta.

eg. You cannot serve maggie before boiling water.
eg. You " deploy a s/w before fixing bugs

Action 1: boil water
" 2: add maggie
" 3: add masala
" 4: serve maggie



Example:-



possible runs:

5 4 2 3 1 0

4 5 2 3 1 0 → only after 4 and 5.

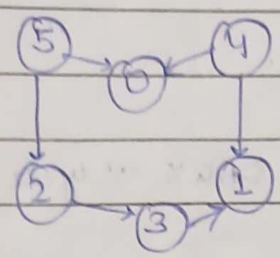
Topmost

for node 2 will come after 5 only since goes for 3 (only after 2)

only after 3

- Approach → modified DFS

"track what comes first"



DFS → stack add ∵ stack is a LIFO

∴ Jo last mei aaya hai woohi sabse pehle niklega (e.g. 5)

↳ ∴ put 5 at last.

dfs(graph, curr, vis, stack){
  vis[curr] = true;
  neigh → call (unvisited only)
  stack.add(curr)
}

↳ add hone ke baad jab stack se niklenge that will be my topological sorting order.

for (int i=0; i< graph.length; i++){
  if (unvisited)
    dfs() ✓

} ∵ to visit all vertex (connected components)

Dry run:-

1. Add 0 → vis✓ no neighbour → stack.add ✓
2. " 1 → vis✓ " " → " " ✓
3. " 2 → vis✓ → neighbour 3 vis✓ (unvisited) ∴ call → ~~add 3 in stack~~

check neighbour of 3: 1 (already visited) → no other neighbour of 3 → add 3 in stack → no other neighbour of 2 → add 2 in stack.

4. Add 4 → vis✓ → no unvisited neighbour → stack.add ✓
5. " 5 → " " " " "

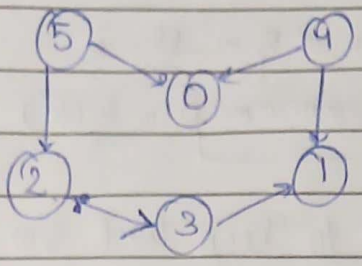
When stack pops out ⇒ 5 4 2 3 1 0 → Topological sorting order

Time Complexity: O(V+E).

Date 05/09/24
Page

* Topological Sort using BFS

"Kahn's Algorithm"



	0	1	2	3	4	5
in-degree : (incoming)	2	2	1	0	0	0
out degree : (out-going)	0	0	1	1	2	2

- Fact: A DAG has at least 1 vertex with in-degree 0 and one vertex with out-degree 0.
(Observation)

- We know Topological sort works on dependancy, so we can conclude that the top most node has no dependancy with in-degree = 0.

indeg :	2	2	1	1	0	0
	0	1	2	3	4	5

BFS (loop on Queue)

1. Put first those nodes with in-degree 0. ∵ Queue is FIFO.

4	5

→ topological sort logic

print 4 → then check for its neighbours

if in-degree → 0 ∴ add in Queue

∴ Decrease in-degree by 1 for 0 and 1.

0 (2) → 1 ; 1 (2) → 1

2. Now print 5 → then check for its neighbours → decrease their in-degree by 1 0 (1) → 1 (0) ; 2 (1) → 1 (0) → if both got in-degree zero → add in Queue.

4	5	0	2

→ print 0

3. 0 has no neighbour ∴ no change

4. 2's neighbour → 3 → in-degree reduced by 1 ∴ now become zero ∴ add in Queue.

print ✓

4	5	0	2	3

5. Print 3 ✓ → 3's neighbour: 1 → indegree(1) → 0 ← → add in Queue ∴

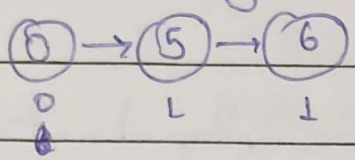
4	5	0	2	3	1

6. No neighbour of 1 ∴ no change → print 1.

4	5	0	2	3	1

Hence I/O/P = 4 5 0 2 3 1

• Why no need of visited array check?

e.g. 

∴ print 0 → reduce neigh in-deg. by 1 → 5 → dly 6

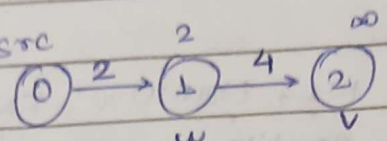
now become 0 (in-degree)

Matlab isse pehle sabhi ka indegree ← 0 hi honge for sure ∴ no need to keep track via visited array.

Date 09/09/24
Page

* Dijkstra's Algorithm

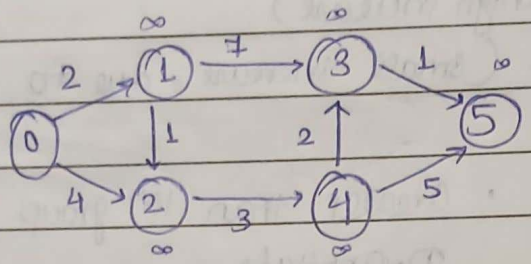
- greedy based algorithm.
- Shortest path from source to all vertices (weighted graph)



initially initialized with ∞

Logic: [dist[u] + wt(u,v) < dist[v]]

∴ dist[v] = dist[u] + wt(u,v)



Source = 0 BFS manner

1. update all neighbours of source (hamesha direct ki distance update hota hai) 1 → ∞ = 2
   2 → ∞ = 4
2. Now check for not visited + shortest ∴ 1.
   (1,2,3,4,5) (1) ↳ 1→2→4 ↳ ∴ visit this
3. 1st neighbour of 1 → 3 (∞) ∴ dist[1] + wt(u,v) < ∞ = dist[v]
   ∴ dist[v] = dist[1] + 7 = 2+7 = 9
   ully 2nd neighbour of 1 → 2 (4) ∴ 2+1 < 4
   ∴ dist[2] = 3
4. Again check for visit x → (2,3,4,5) (2) ∴ visit 2.
   shortest ✓ ∴ 3
5. neighbour of 2 → 4 (∞) ∴ 3+3 < ∞ ∴ dist[4] = 6.
6. Shortest → (4) ∴ visit 4
   visited x → 3,4,5

7. Neighbour of 4 → 3 ∴ 6+2 < 9 ∴ dist[3] = 8
" " 4 → 5 ∴ 6+5 < ∞ ∴ dist[5] = 11

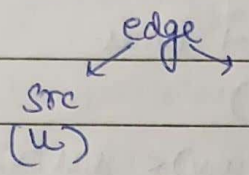
8. shortest → (3) ✓ → visit.

9. Neighbour of 3 → 5 ∴ 8+1 < 11 ∴ dist[5] = 9

10. No Node after 5 → ∴ visit 5

Hence final distance :-
0 → 0 (0)
0 → 1 (2)
0 → 2 (3)
0 → 3 (8)
0 → 4 (6)
0 → 5 (9)

- **Implementation :-**



(1). Initialise dist. src → src = 0, else ∞

(2) priority Queue to store pairs(node, dist)
→ for shortest dist. node
help of comparable comparator
{ covered in greedy } { read pdf of greedy.

```
while (priority queue is not empty) {
    take out curr (if not visited) → visit
    if (vis[curr] == false) {
        neighbours → dist[u] + wt(u,v) < dist[v]
            ∴ update dist[v]
    }
}
```

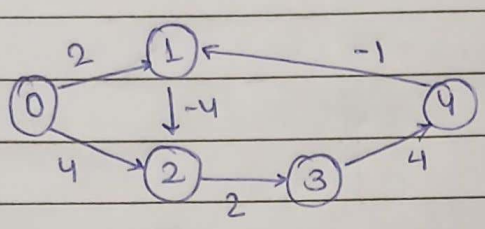
with priority Queue → T.C. = O(V + E log V) ; without PQ → T.C. = O(V²)

Check code seriously !!!!! — internal sorting in PQ.
dist. boolean for visited

* Bellman Ford Algorithm

- For the cases in which negative weighted edges comes, we don't solve with Dijkstra's Algo. due to any casualties.
- DP based algorithm
- Bellman Ford can do same operation as Dijkstra + for negative edges also.

→ Shortest paths from source to all vertices (negative edges)
Time complexity → $O(VE)$ i.e. > T.C. of Dijkstra
∴ Dijkstra is preffered for +ve edges.



E(edges) = 6
V(vertex) = 5

Perform this operation V-1 times
for all edges (u,v)
if (dist[u] + wt(u,v) < dist[v]){
dist[v] = dist[u] + wt(u,v)
}
} called as Relaxation (∵ we're relaxing our distance)

∴ for (int i=0 to V-1) } Pseudo code
edge (u → v)
relaxation ✓

		0	1	2	3	4
dist[] →		0	∞	∞	∞	∞
↓ Vertex-1	1	0	2	~~4~~ -2	0	4
	2	0	2	-2	0	4
	3	0	2	-2	0	4
	4	0	2	-2	0	4

⇒ ans.

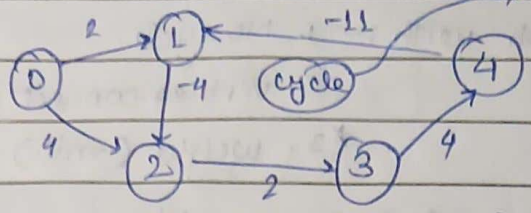
Coincidently isme all 4 iterations ka same aa gye but sab kinhi aayenge always (V-1) wala hi ans. hoga.

Date
Page

★ Note :- Doesn't work for negative weight cycles.

↓

Reason :→



weight of this cycle ⇒ -4+2+4-11
= -9.

Bellman Ford will not work

min. Path for 1 :-

0 → 1 :. 2

0 → 1 → 2 → 3 → 4 → 1 :. -7 ✗

this will be the shortest path but we've already walked on 1 so it won't make any sense of walking over it again to get min. Path LOL.

Pseudo Code → (1). initialize all distance except source → ∞.

(2). int v = graph.length

for (i to v-1){

for (j to graph.length) { → to get all vertices from graph

for (k to graph [j]. size()) { → edges respective to all these vertex

Edge e = graph[j] . get(k); → each edge → vertex's edge

int u = e.src;

int v = e.dest;

int wt = e.wt;

~~if (relaxation condition){~~

if (dist[u] != Integer.MAX_VALUE && dist[u]+wt < dist[v]

dist [v] = dist[u]+wt ;

}

In Java if we add any integer with ∞ → it will make it -ve

// print → for (int i = 0 to dist. length) {

sout (dist[i] + " ");

}

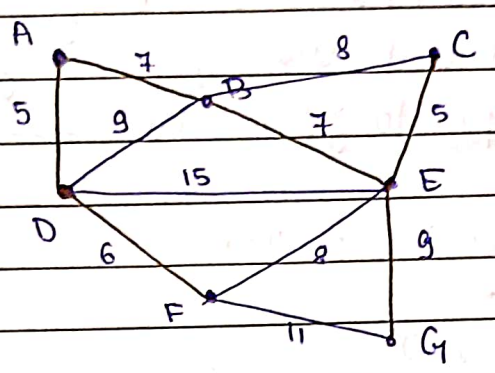
sout();

10/03/24

* Minimum Spanning Tree

⇒ It is a subgraph with :- 1. No cycle
2. vertices connect (all)
*3. weight (min)

⇒ That edges which have the minimum weight across a vertex which when connecting (considering only them) gives a subgraph is called Min. Spanning Tree.

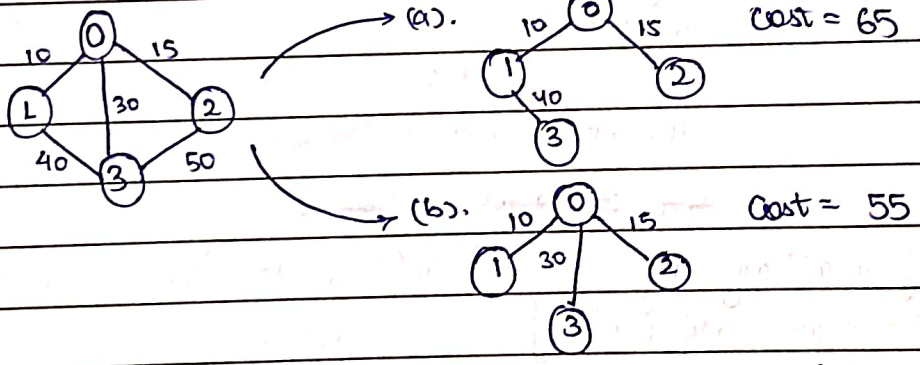


Only those vertices connected via black edges are called Min. Spanning Tree.

algorithm for solving MST is called

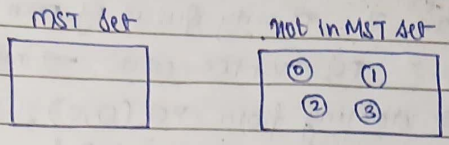
- **Prim's Algorithm**
  ↳ works for undirected + weighted MST

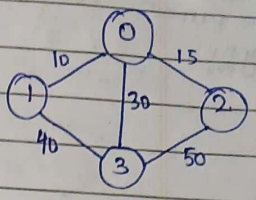
eg.



∵ cost (a) > cost (b) ∴ MST → (b) ✓
from 'a' and 'b'.

- **Dry run:-**





VS [| | | |]
0 1 2 3

cost = 0

(1). Choose any vertex (let 0) → make it true in visit → Put in MST set & remove from not in MST set.

(2). Make the distance of 0 → Integer-Min and check for other vertex going from MST to Non MST set (1, 2, 3) → check their costs 0-1 (10), ~~0-3~~ 0-3 (30), 0-2 (15).

(3). Select that node which have min. cost (MST → Non-MST set) i.e. 1 : (10)
∴ add 1 in MST set. also make cost = 0+10 = 10
↳ make it True in visit

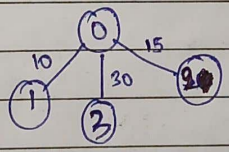
(4). Check for edges going from (0,1) to (2,3) and select for minimum one ⇒ (1-2) None, (1-3) 40, (0-2) 15, (0-3) 30.
(1-2) None: Don't consider; (0-2) 15: Min.

(5). add 2 (visit ✓) in MST set, and make cost = 10+15 = 25.
Obviously remove from Non MST set

(6). Check for edges going from 0, 2, 1 to 3, i.e. (0-3) 30, (2-3) 50, (1-3) 40.
(0-3) 30: Min. with min. cost

(7). add edge 3 from (0-3) (visit ✓) in MST set & cost = 25+30 = 55.

Hence,



Note: If in ques. asked to store Edge ∴ store edge in MST via Arraylist (AL<edge>) → cost ∴ Total MST cost

Prim's Algo may vary on Ques.
∴ Dont memorise for one purpose only !!!

Pseudo code =>
1. make vis[]
2. " priority queue of pair type
   PQ<vertex, cost> → min. pair (↓cost)
3. starting from PQ (0,0), initialise cost=0
4. while (pq. is not empty) {
   curr → (v, cost)
   if (not vis) {
   visit ✓
   add in MST (ans) → cost
   add → neighbours → in PQ
   } } to check neighbour's edge can be added in PQ or not.

Note:- A priority queue is basically a min. heap structure which sorts values, here in order to sort on the basis of vertex or cost we use implements Comparable interface <Pair>
∴ @Override
return this.cost - p2.cost; → sorting on the basis of cost.
if we reverse this we'll get sort in descending order.

also reading material pdf ← revise Greedy (already covered) given there.
Also watch later YT on comparators in Java.

Date 10/09/24
Page

* Disjoint Set [data structure]

Find : which set does the element belongs to
Union : merge.

n = 8

grp1 (set1) : (1)-(2)-(3)-(4)
grp2 (set2) : (5)-(6)-(7)-(8)

find(4) → set1
find(6) → set 2

union(4,8) → (1)-(2)-(3)-(4) with (5)-(6)-(7)-(8)
(4 → set1, 8 → set2)

find(6) → set 1

union(1,5) → "Cycle Detection"
(1 → set1, 5 → set2)

⇓

when we do union of elements in same set cycle is formed.

∴ It is used for : (1) Cycle Detection
(2) Kruskal's algo (to find MST) → apart from Prim's Algo.

11/09/24

- implementation :

starting :→ parent →

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

initially sab apne group ke leader hain

rank → (height)

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

find → leader
Union → 2 groups join

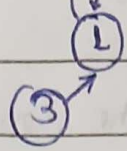
(0) (1) (2) . . . . (7)

For given set of instructions check how'll this works :—

→ next page

union (1,3)
find (3)
union (2,4)
union (3,6)
union (1,4)
find (3)
union (1,5)

(1). union (1,3)



- Now rank of 1 will be increased from 0→1
- Also parent of 3 updated from 3 to 1.

(2) find (3) → • check for parent of 3 i.e. 1
• Now check rank of 1 i.e. 1
Hence find(3) → gives 1.
∴ Group leader = 1.

(3). union(2,4) :

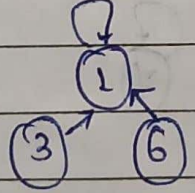
- Now 2 ya 4 any one can be leader ∴ let 2.
- Therefore ~~leader~~ parent of 4 ⇒ 2 and parent of 2 already 2.
- Also rank of 2 updated to 1.

(4). union(3,6)

- 3 का union matlab 3 के poora group का union hoga.
- look for rank of 6→0, rank of 3 → means iske parent ki rank dekho i.e. 1 → 1
(group के leader की rank dekhi jaati hai sirf)
∵ 6 के group का leader 6 hi hai ∴ uska rank = 0
and 3 " " " " 1 hai ∴ uska rank = 1

Hence, Jiska rank kam hoga wo aake ~~uska~~ bade wale ke leader ke saath join hoga.

∴



(5). union(1,4)

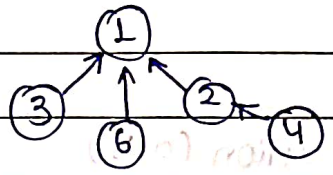
- Since union sirf parent pr hota hai ∴ 1 and 2
- Also since both have same rank ∴ koi bhi kisi pr bhi join ho sakta hai. (by default 2 → 1 pr join ho gaya)

∴ parent of 2 (now) = 1

Hence,

- rank of 1 = 2

updated Tree ⇒



- rank of 2 = 1
- parent of 2 = 1, parent of 3 = 1, parent of 6 = 1, parent of 4 = 2

Note:- Final leader wo ho hai jiska leader wo khud hota hai.

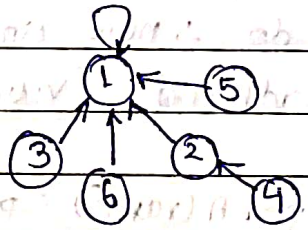
(6). find(3)

- ∴ leader of 3? ⇒ from parent check ~~too~~ for 3 i.e. 1
  Now check parent of 1 i.e. 1.
  Hence, final leader = 1.

eg. find(4) gives 1 → socho step by step.

(7). union(1,5)

∴ parent of 5 = 1 ∴



- Pseudo Code =>
  - make int n, par[n], rank[n]
  - initialise par[i] = i ∵ sab khud ka parent at first and all rank at starting → 0 i.e. rank[n] → 0
  - find (x)

∴ for supreme parent → if (x = par[x]){
return x;
}
else{
return find(par[x])

- union (a, b)

∵ we find union for parent only

∴ get parA, parB

to get parA use find(a)

use find (b) do get parent of b

Now => (check for rank)

1. if equal => ∴ par[parA] = parB } ya opposite
   rank[parB]++

Jab dono rank equal ho to kisibhi ek ko (here → let B) ka bada samajh lo and parent of (parent of A) ka B bana do ∴ Now since attach ho gya to B ka rank badha do (v. simple)

2. parA (rank) < parB (rank)

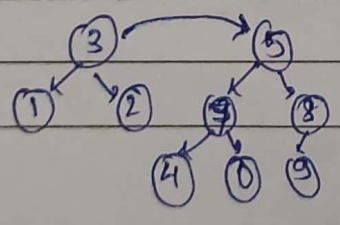
(∴ same operation that A → B ke judega)

∴ par[parA] = parB

but rank will not change (obviously)

∵ badhi height of tree pe chota height of tree ko connect hone ke baad farak nhi padta.

e.g.

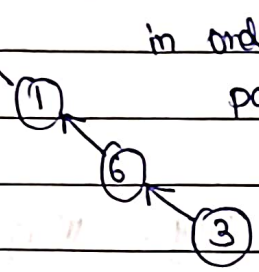


B. if parA(rank) > parB(rank)

∴ par[parB] = parA ;

// check code ↑

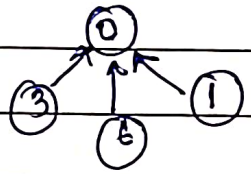
- path compression optimisation for union and find :→

for this → 

in order to find parent of 3 it will check parent of 6 → parent of 1 → ~~parent~~ 0 ✓

rather store values for next time & keep on updating.

ie. 3 ka parent 0 stored
6 " " " "
1 " " " "



∴ optimised

if parent updated sab se ek baar hi change ho jayega.

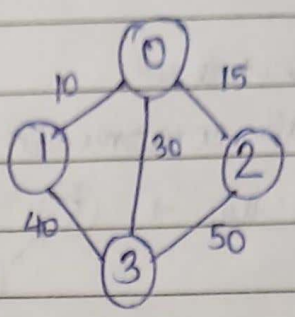
Note :- Time complexity of union & find comes (mathematically) as O(4K) → ∴ O(4) ∴ O(1) → Hence, constant time

★ Therefore it is used for (1) detecting cycles in graph
(2) finding MST in kruskal Algo.

11/09/24

* Kruskal's Algorithm → for MST (greedy)

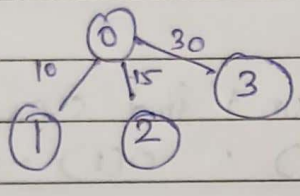
1. Sort edges
2. Take min. cost edge
   ↓
   ✗ form cycle
   include ans.



∴ from example:-

Sorted Edges	Weight	
(0,1)	10	— ①
(0,2)	15	— ②
(0,3)	30	— ③
(1,3)	40	→ ✗ ∵ forms CYCLE
(2,3)	50	→ ✗ ∵ CYCLE

∴



AL<Edge> → Collections.sort()
↓ gives sorting of the elements

define parameter
cost → to sort on the basis of cost

for (int i = 0 to V-1) → (Count MST)
∑ ∵ 2 vertex ~~ki~~ connect karne ke liye V-1 edges are required

- do check cycle we concept of union & find
  ∴ to ~~ind~~ include we union (a,b) ⇒ (a) → (b)

- But if parA and parB are same ∴ CYCLE condition.
  ∴ do not include in this case.

Hence → (pseudocode)

```
for (count = 0 to V-1) {
    Edge e → a (src)
             b (dest)
    using find(e.src)  get → par(A) , par(B)  ← find(e.dest)
    if same → don't include ∴ cycle
    if diff. → include → union(parA, parB)
}
```

∴ Time complexity: $O(V + E \log E)$

//check code