Graph Terminology

Graph

It's a way of connecting dot's.

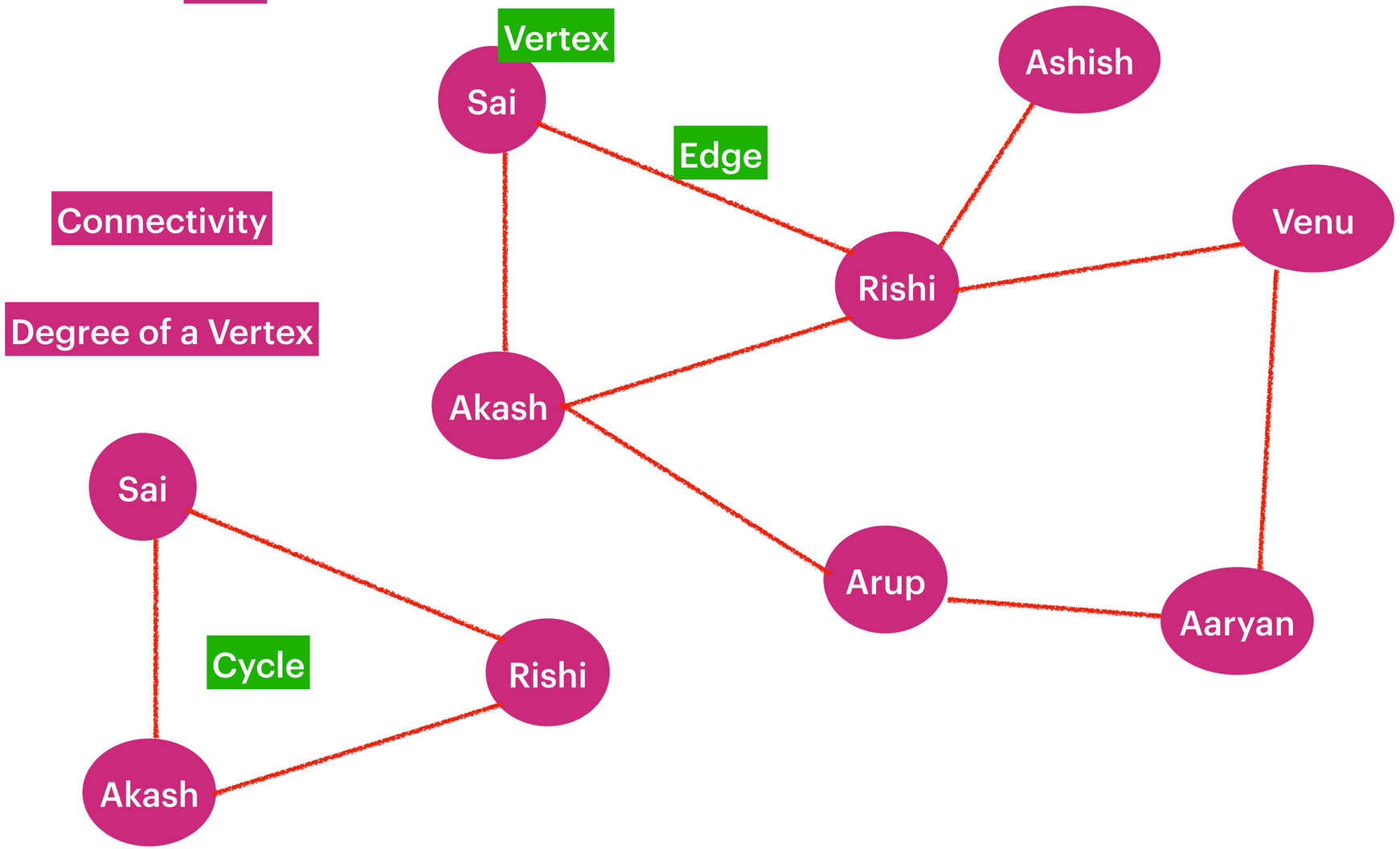
Path

Vertex

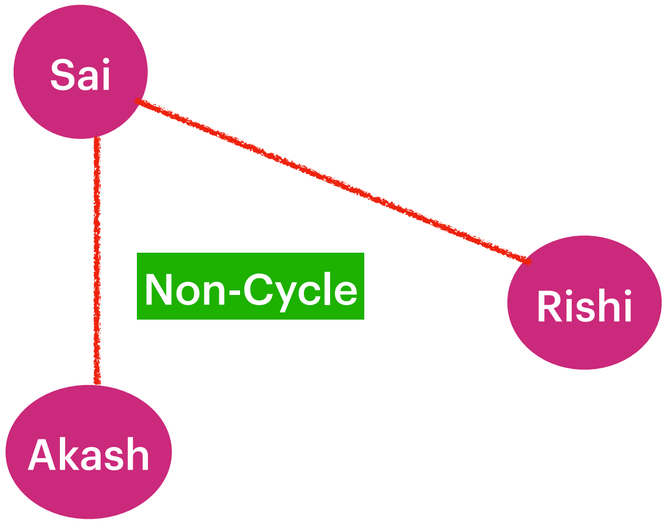
Edge

Connectivity

Degree of a Vertex



Cycle



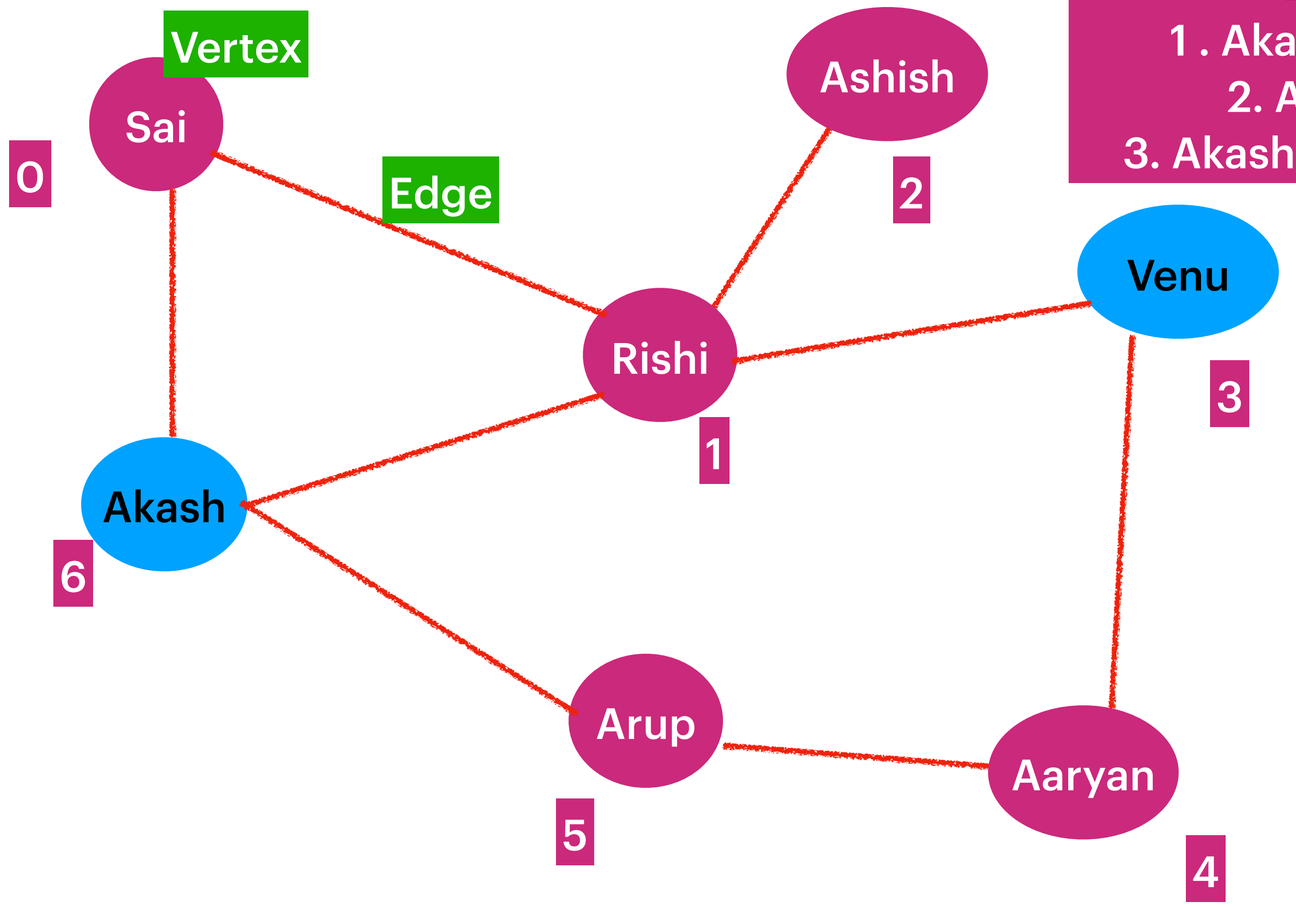
Non-Cycle

Path :
If there is connection exits,
Path just talks about no.of between Source & Destination vertices.

Graph

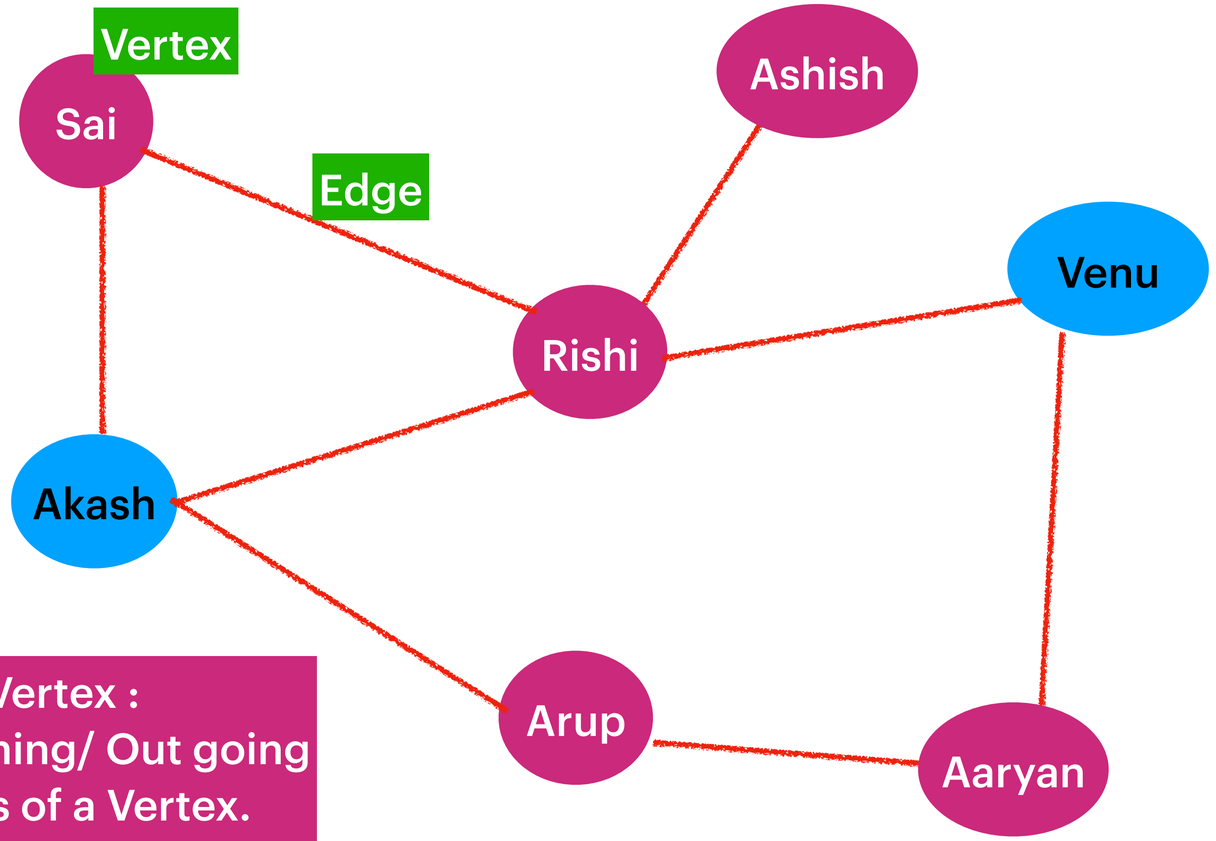
It's a way of connecting dot's.

Path between Akash <-> Venu
Does path exists ? Yes
1 . Akash — Sai — Rishi —Venu (3)
2. Akash — Rishi — Venu (2)
3. Akash — Arup — Aaryan — Venu (3)



Bidirectional Graph

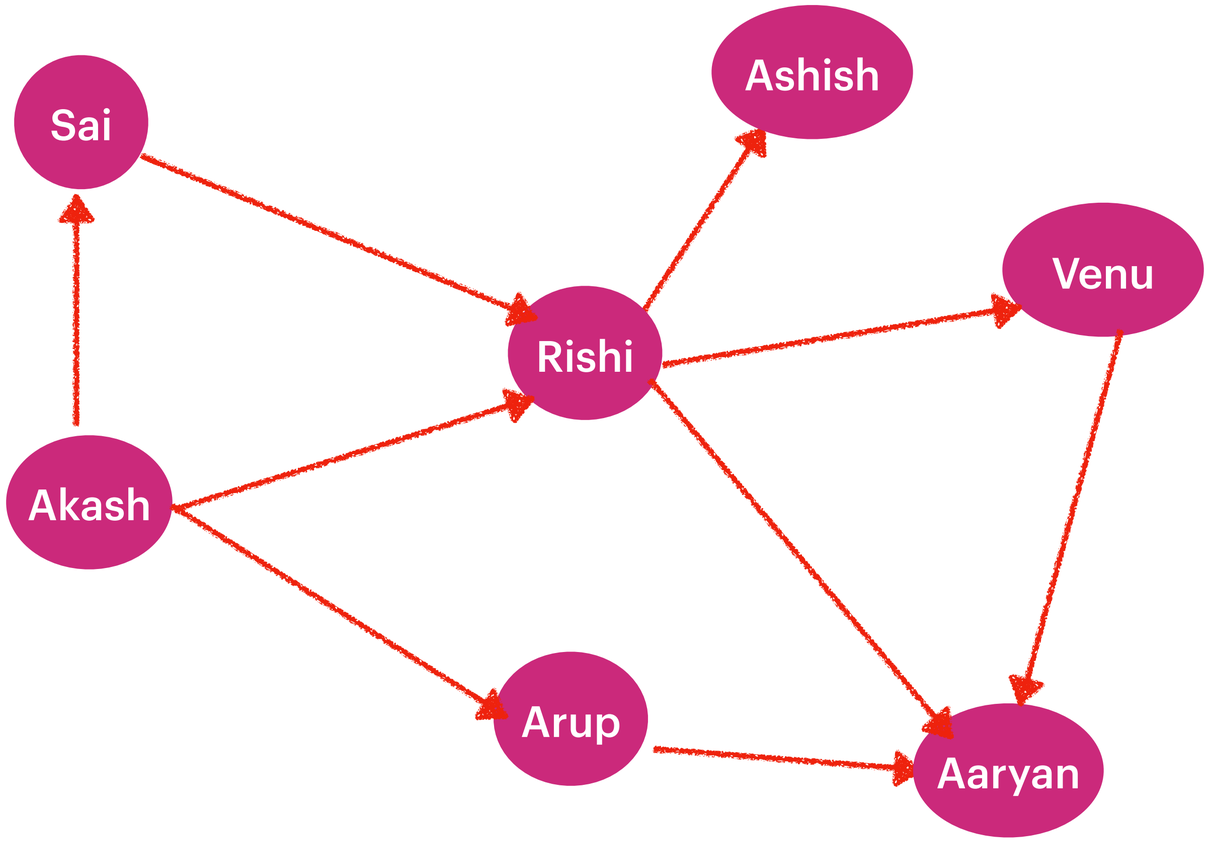
Here the connection exists in Both the ways.
Source <—> Destination



Degree Of Vertex :
Talks about Incoming/ Out going Connections of a Vertex.

DegreeOf(Akash) :
In-Degree =>3
Out-Degree =>3

Directed Graph



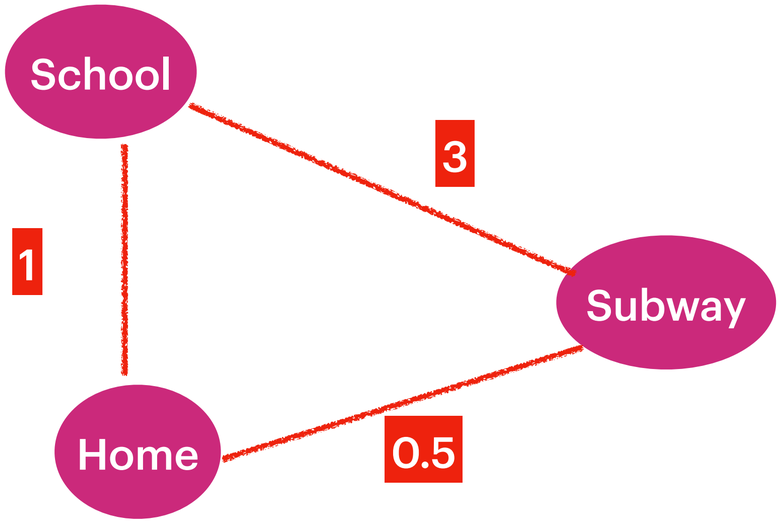
In-Degree (Sai) : 1

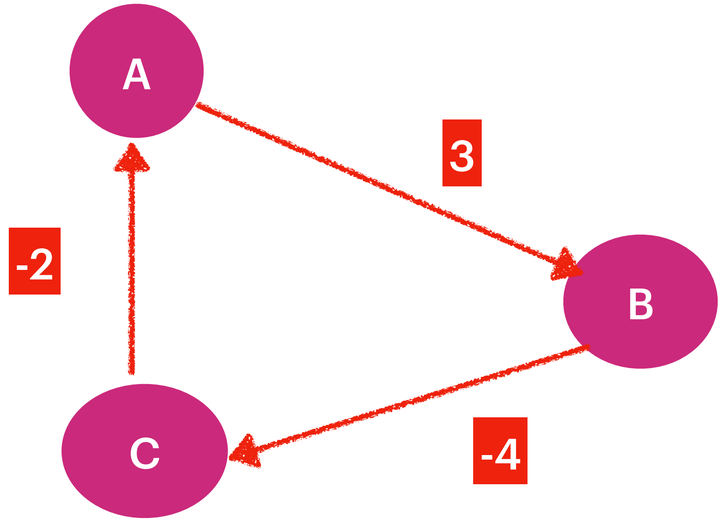
Out-Degree (Sai) : 1

In-Degree (Rishi) : 2

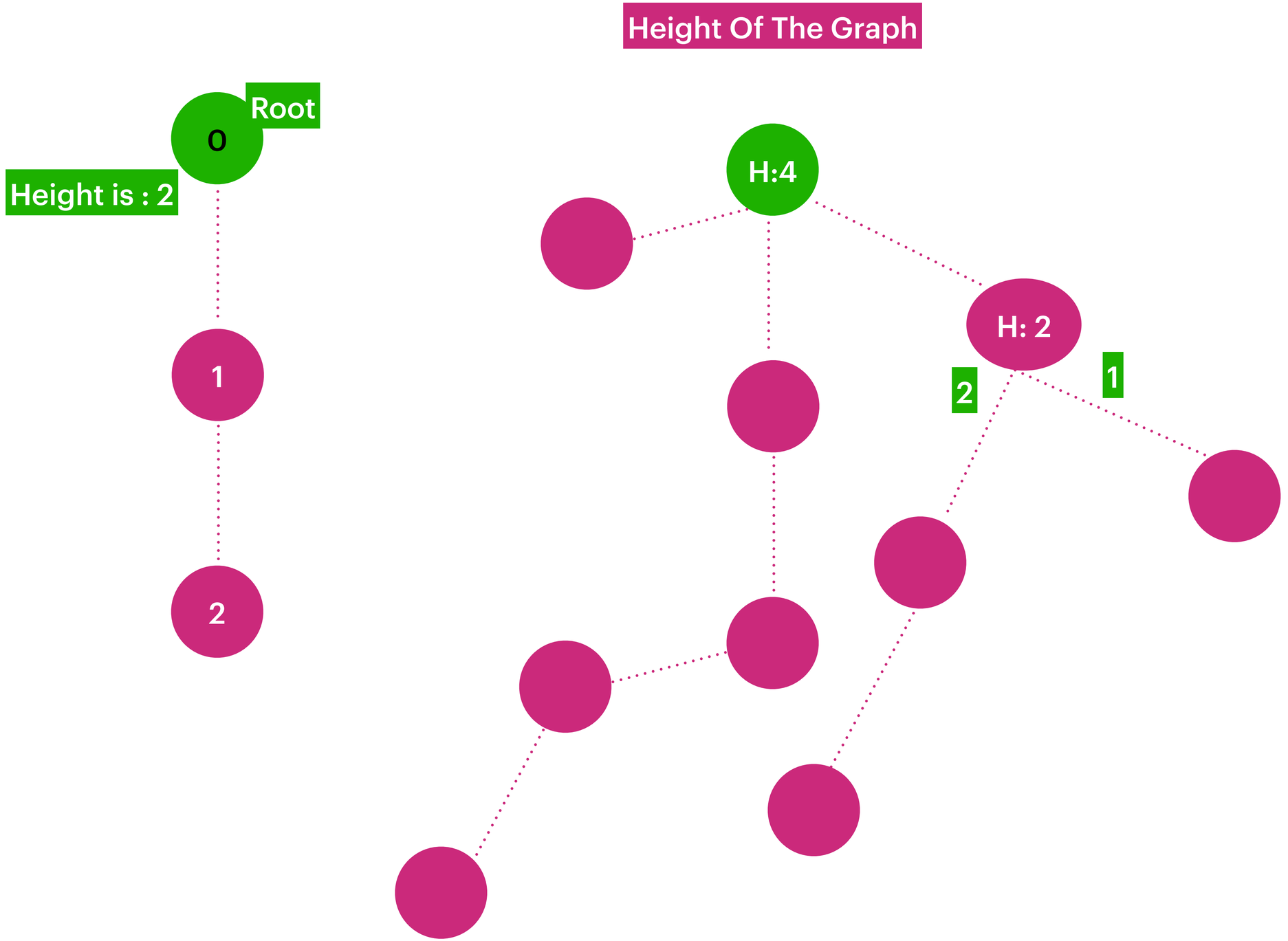
Out-Degree (Rishi) : 3

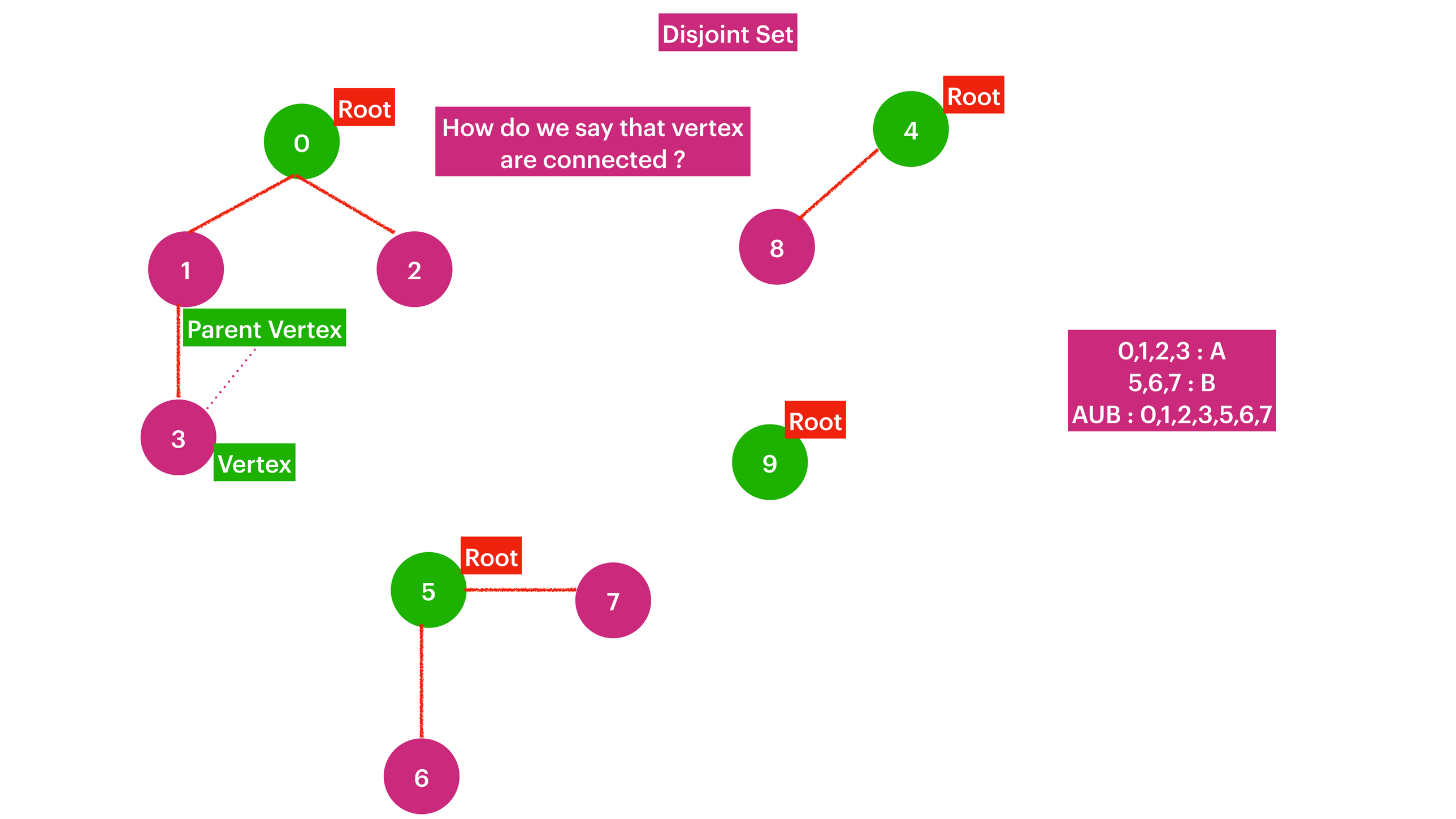
Weighted Graph

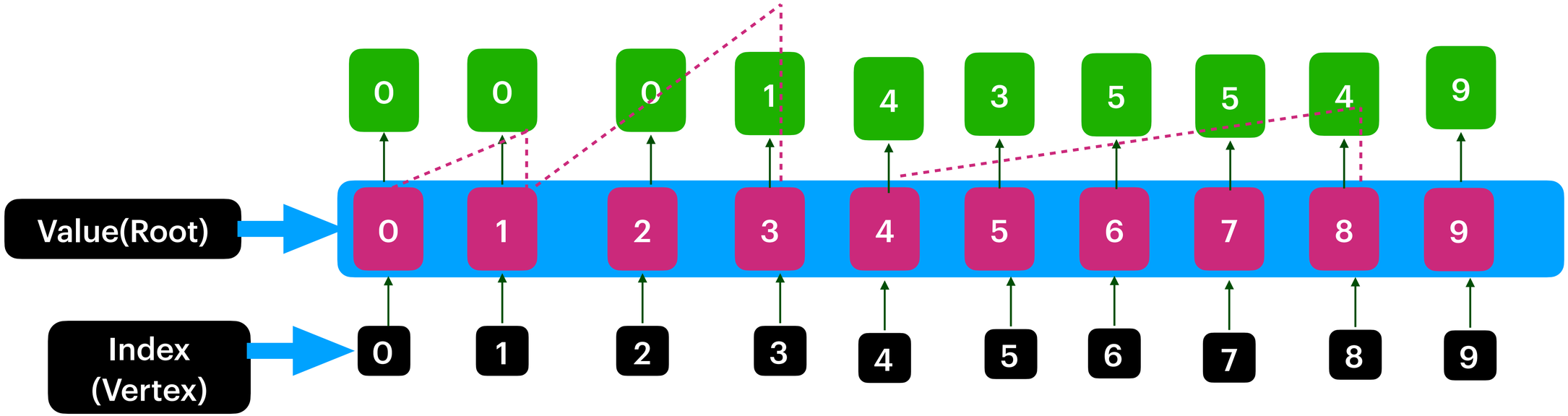




Negative Weight Cycle =
A -> B -> C =3 + (-4) + (-2) = -3


Height Of The Graph
Root
0
Height is : 2
1
2
H:4
H: 2
2
1


Disjoint Set
Root
0
How do we say that vertex are connected ?
Root
4
1
2
8
Parent Vertex
3
Vertex
0,1,2,3 : A
5,6,7 : B
AUB : 0,1,2,3,5,6,7
Root
9
Root
5
7
6



(0,1) , (0,2) , (1,3) , (4,8), (5,6), (5,7)

Connected(1,3) =
root(1) == root(3)
0 == 0 = true

connected(1,8)
root(1) == root(8)
0 == 4 (false)

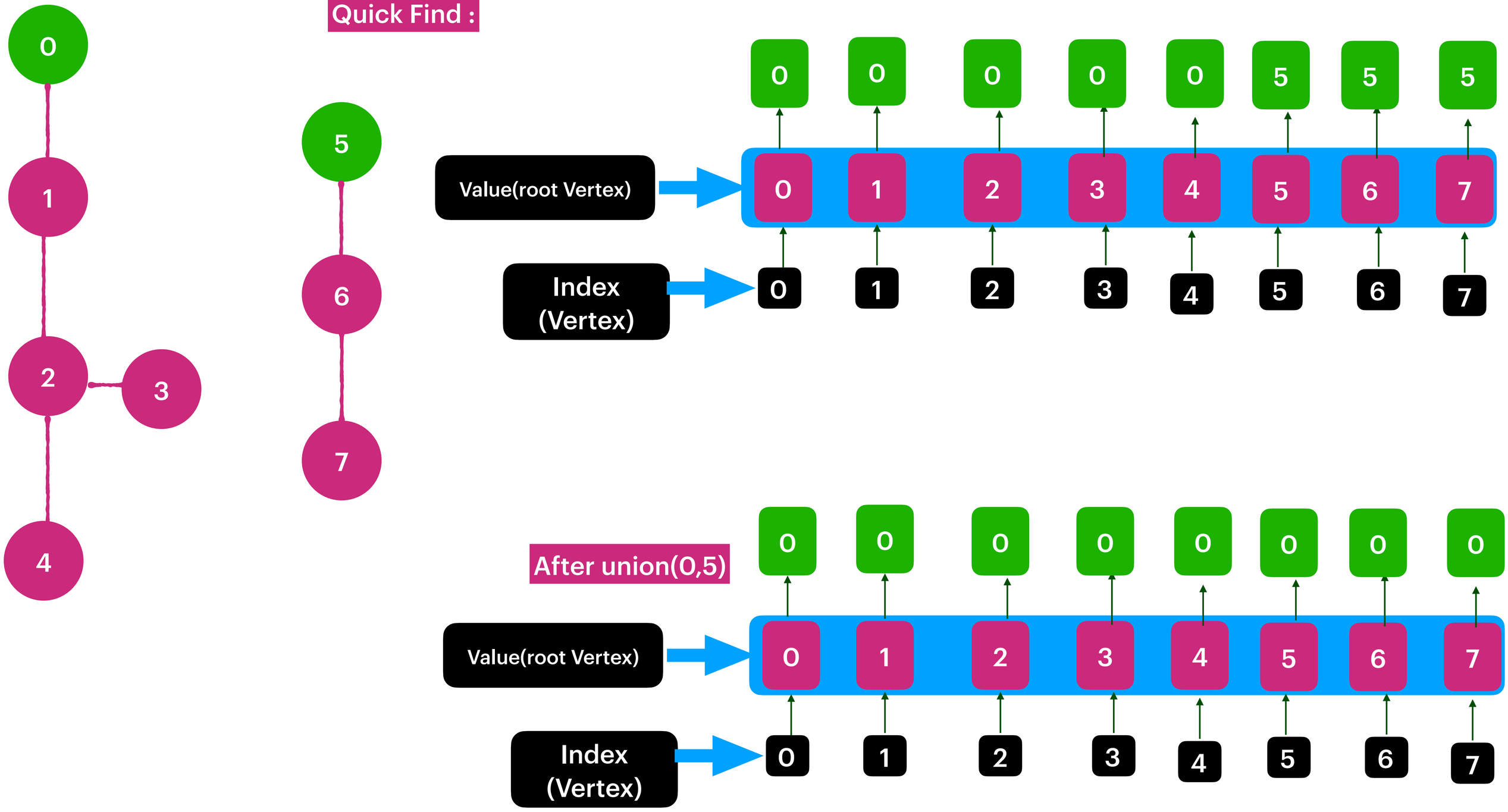
connected(5,7)
5 == 5 : true

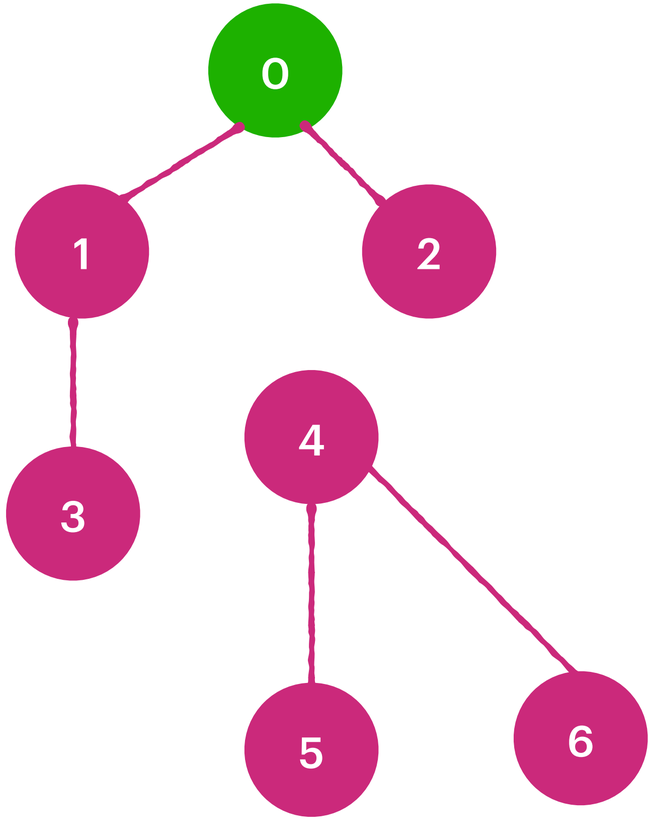
Quick Union:

union(vx, vy) => TimeComplexity : O(1)

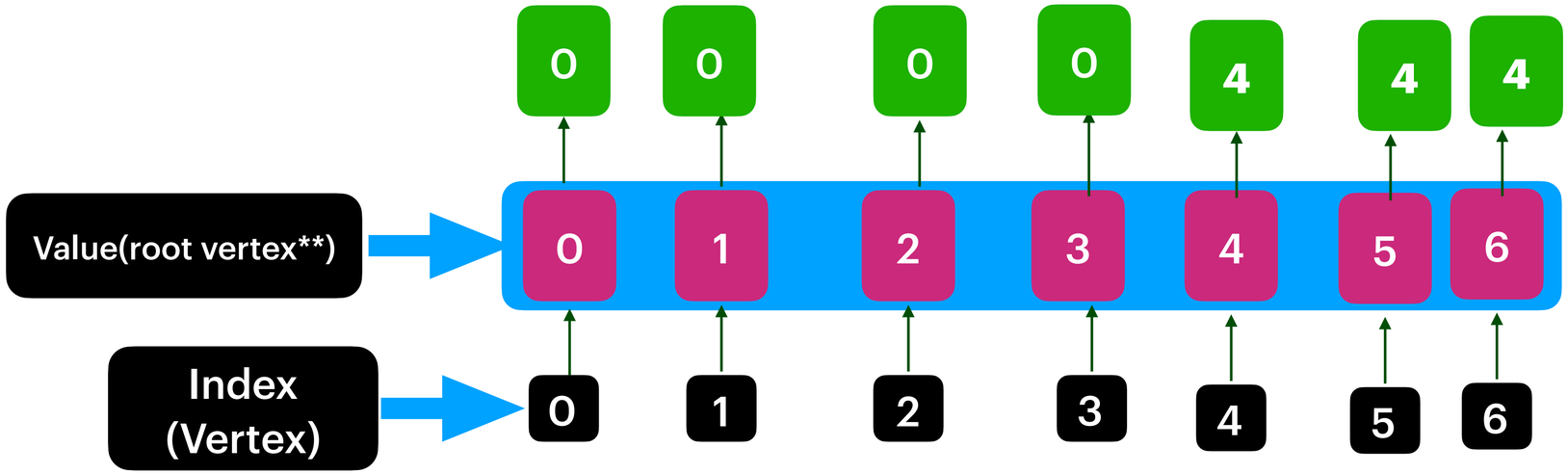
connection(vx, vy) => TimeComplexity : O(n) + O(n) = O(2n) = O(n)

find(vertex) => Time Complexity : O(n)


Quick Find :
0
1
2
3
4
5
6
7
Value(root Vertex)
Index
(Vertex)
0 0 0 0 0 5 5 5
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
After union(0,5)
Value(root Vertex)
Index
(Vertex)
0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7



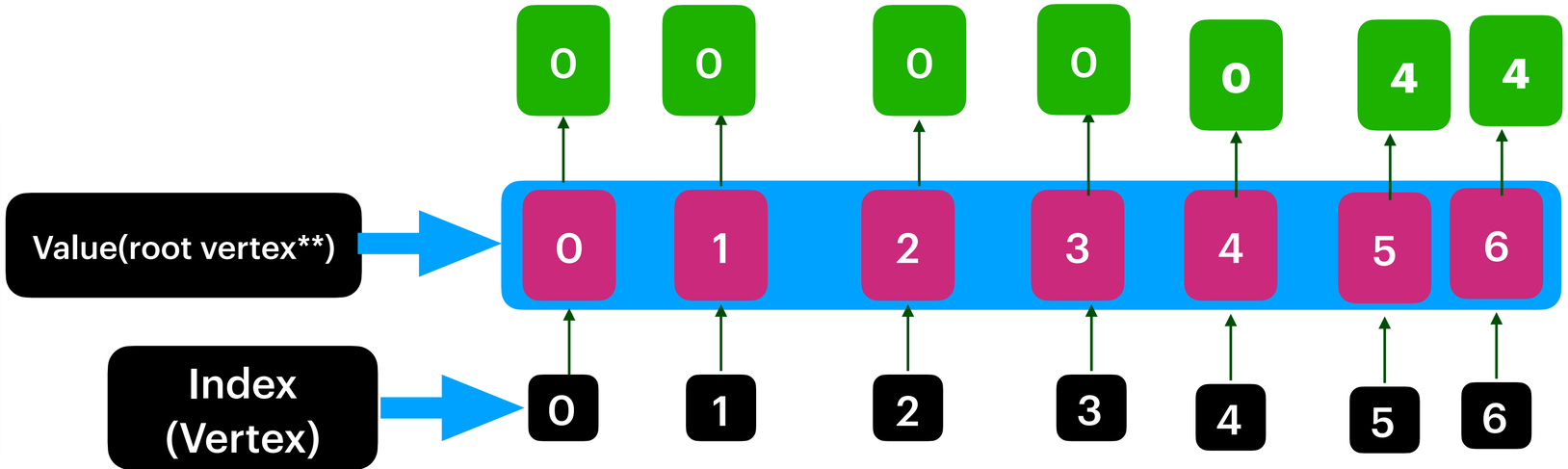
Quick Union :

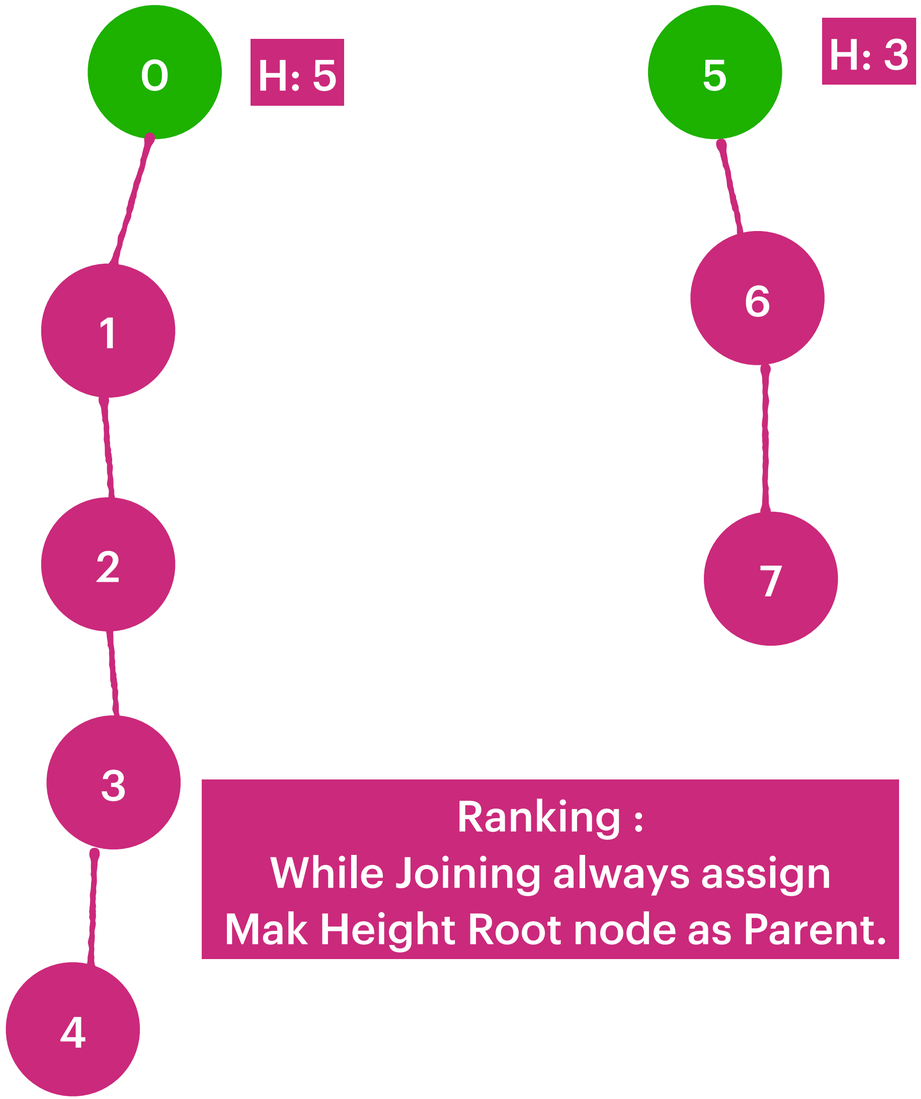


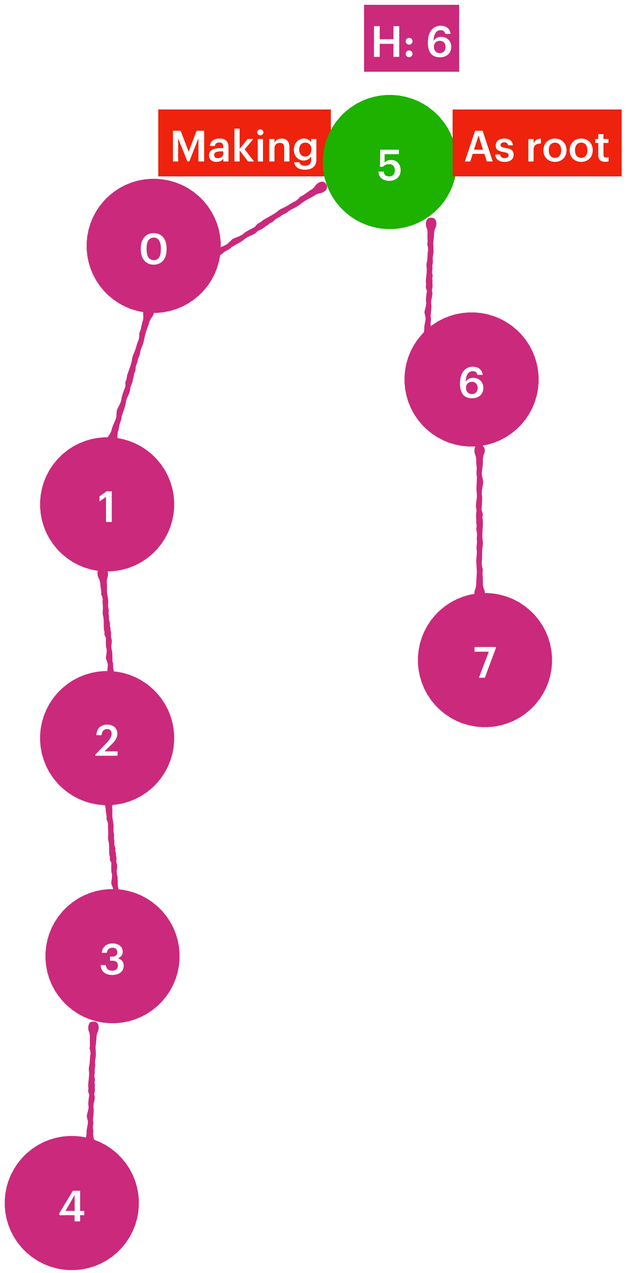
union(0,1)
union(0,2)
union(1,3)
union(4,5)
union(4,6)

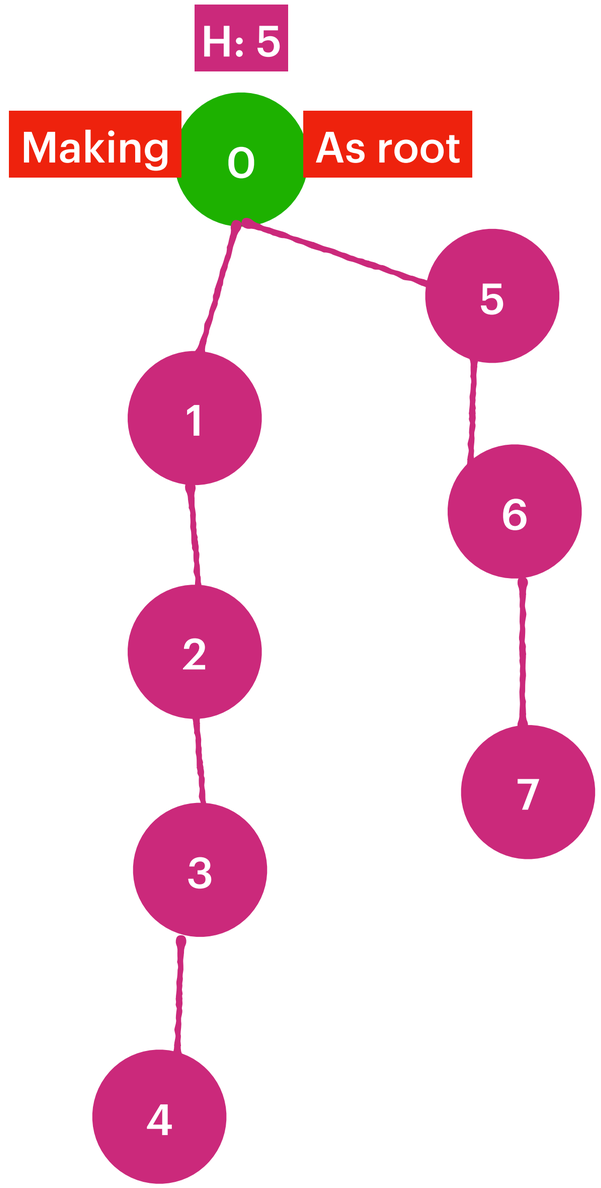
union(0, 1)
Int rootX = find(x); // 0
Int rootY = find(y); // 1
root[rootY] = rootX; //

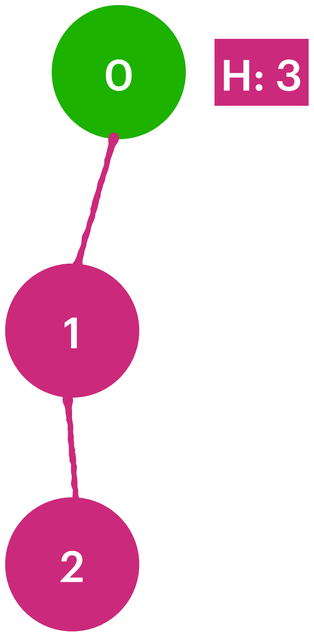
After union(3,5)
Int rootX = find(3) = 0
Int rootY = find(5) = 4
root[rootY] = rootX
root[4] = 0

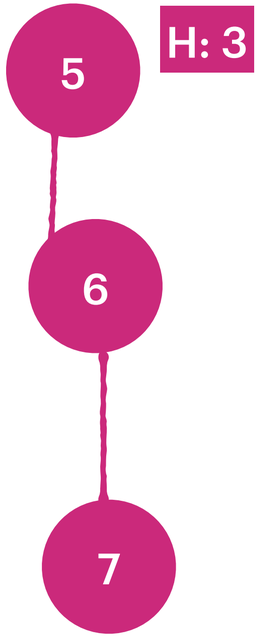


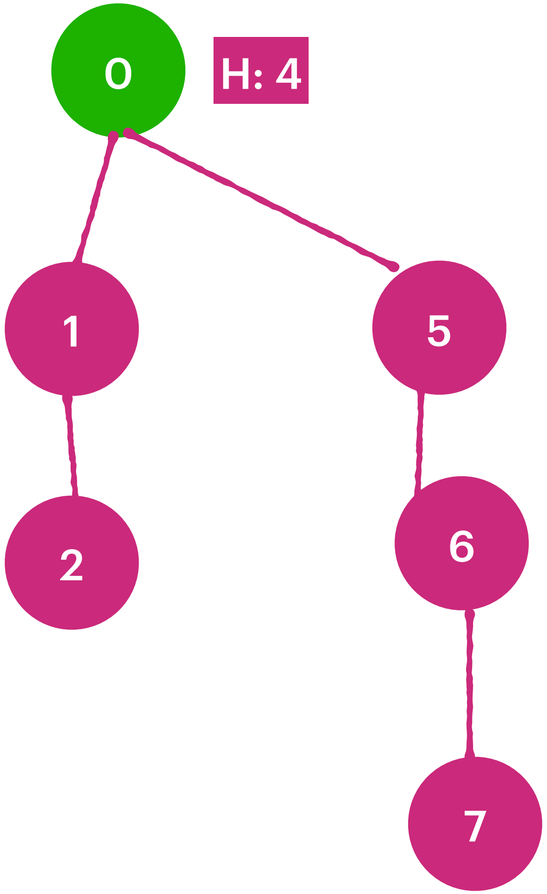

0
H: 5
1
2
3
4
5
H: 3
6
7
Ranking :
While Joining always assign
Mak Height Root node as Parent.

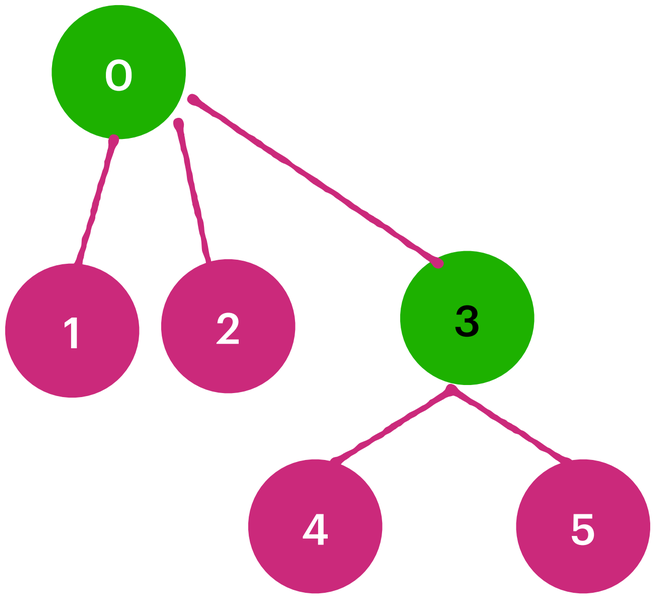

H: 6
Making
5
As root
0
1
2
3
4
6
7

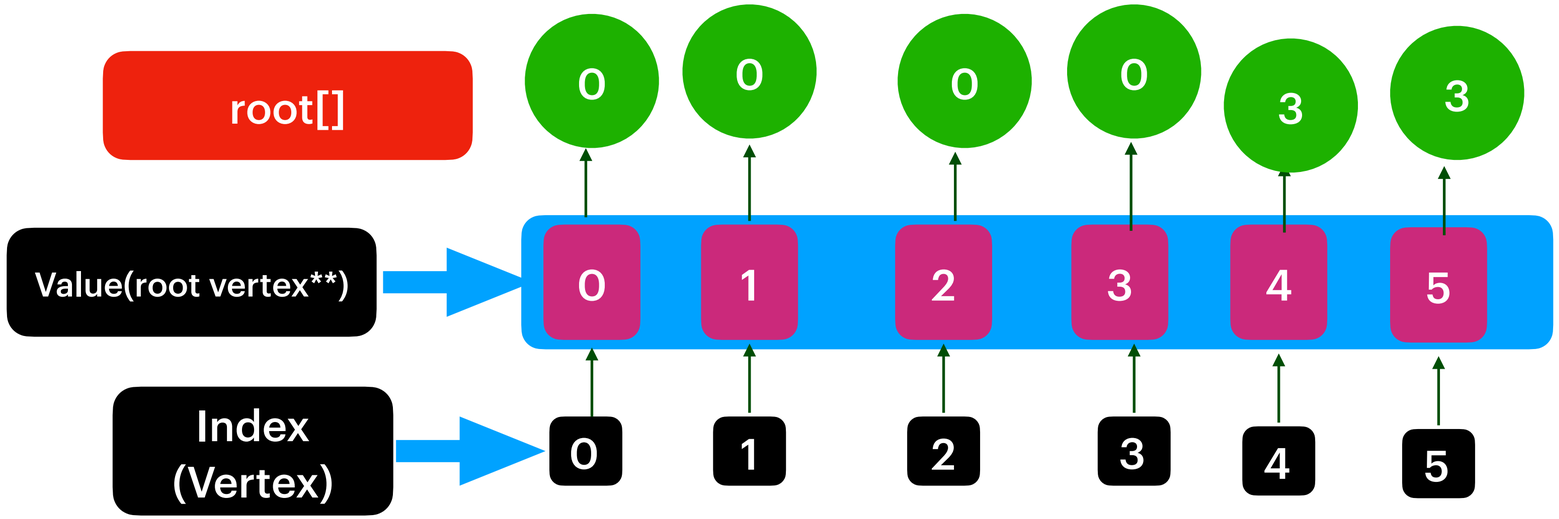

H: 5
Making
0
As root
1
2
3
4
5
6
7

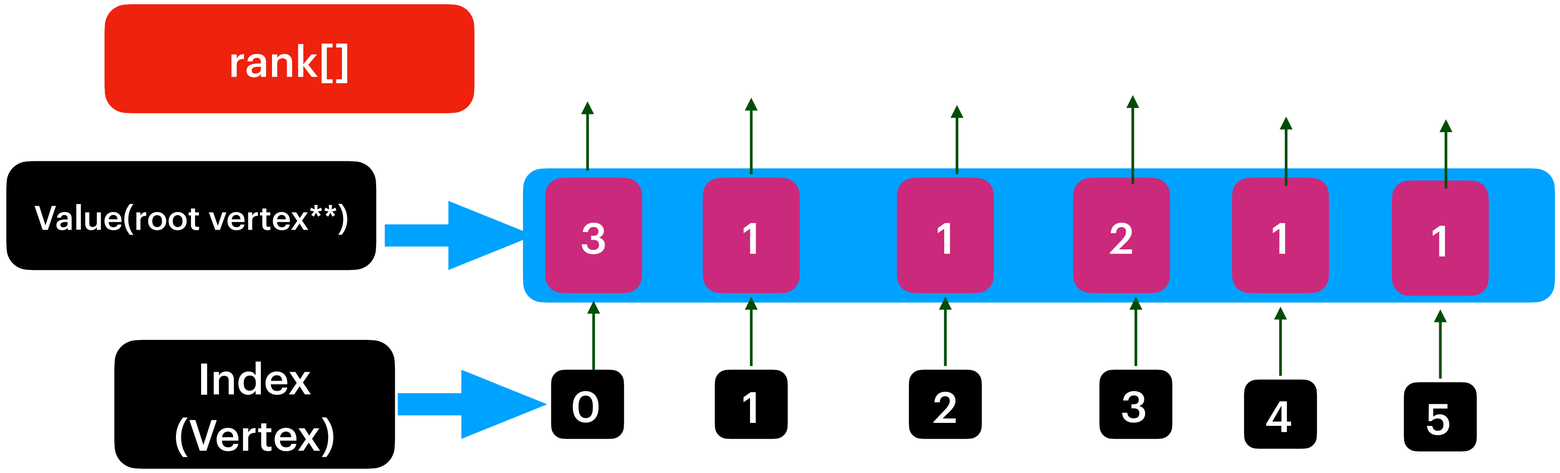

0
H: 3
1
2

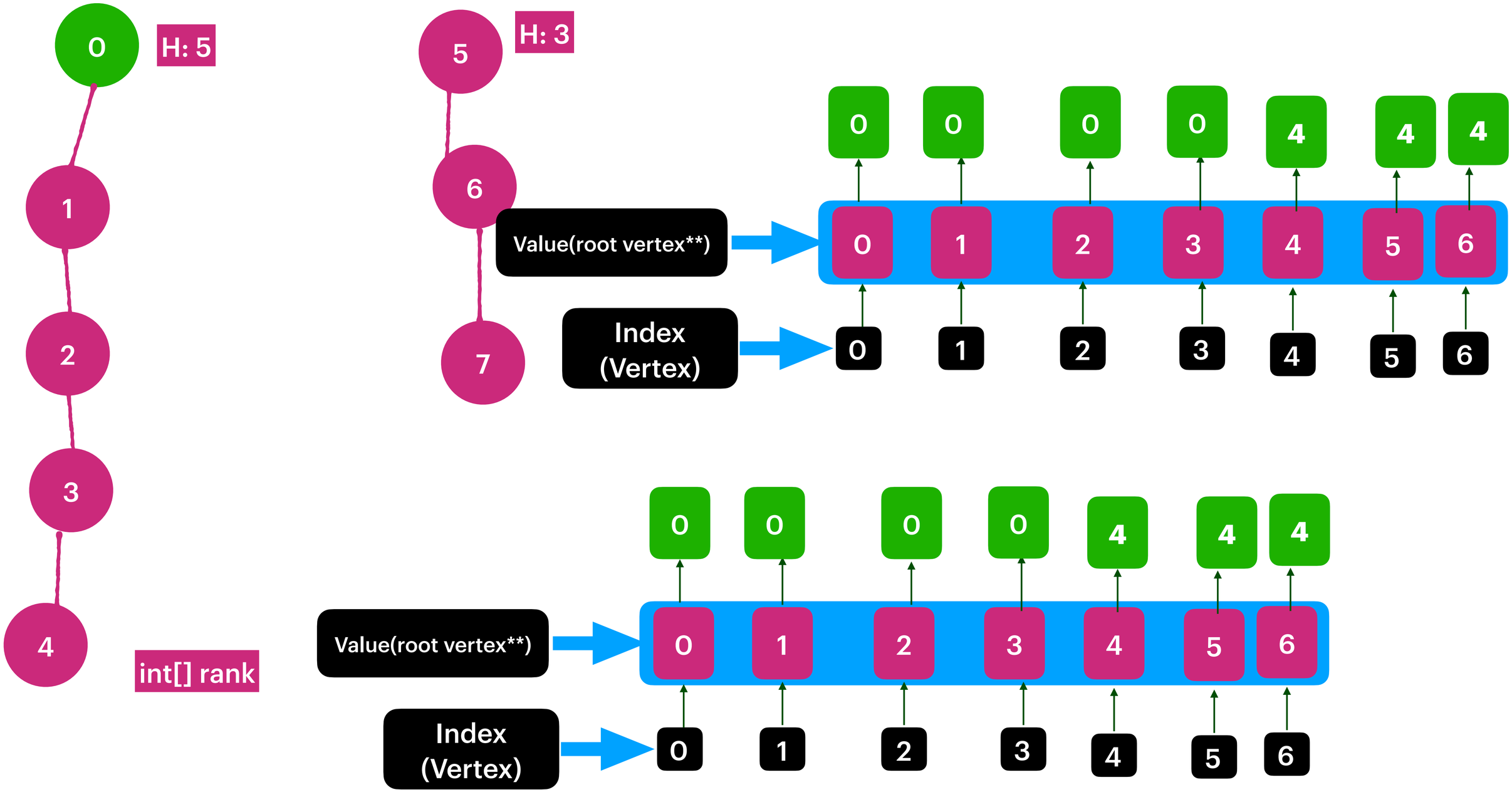

5
H: 3
6
7

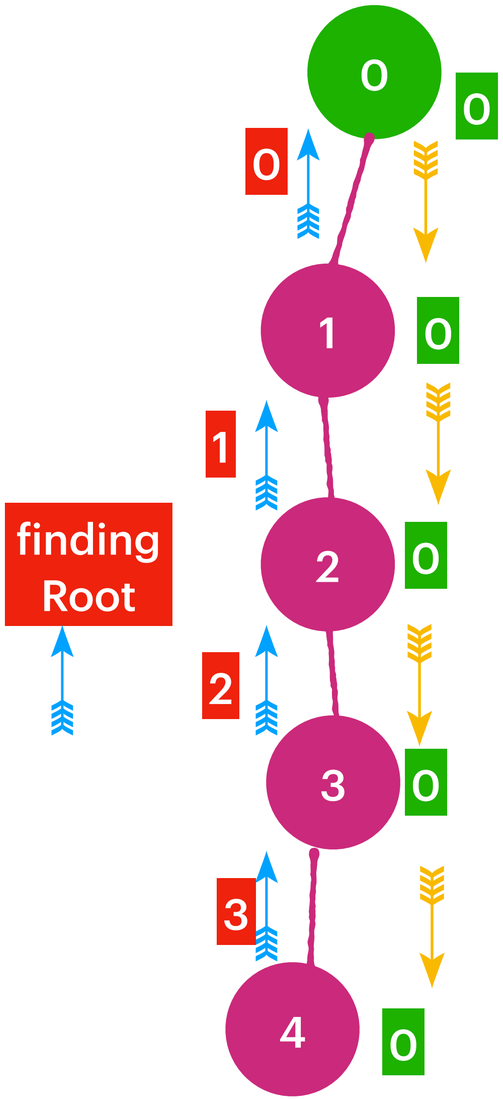

0
H: 4
1
5
2
6
7


0
1
2
3
4
5


root[]
Value(root vertex**)
Index
(Vertex)
0
0
0
0
3
3
0
1
2
3
4
5
0
1
2
3
4
5


rank[]
Value(root vertex**)
Index
(Vertex)
3
1
1
2
1
1
0
1
2
3
4
5


0
H: 5
1
2
3
4
int[] rank
5
H: 3
6
7
Value(root vertex**)
Index
(Vertex)
0 0 0 0 4 4 4
0 1 2 3 4 5 6
0 1 2 3 4 5 6
Value(root vertex**)
Index
(Vertex)
0 0 0 0 4 4 4
0 1 2 3 4 5 6
0 1 2 3 4 5 6



O(H)

Path Optimization

While traversing back, update the child vertex root.

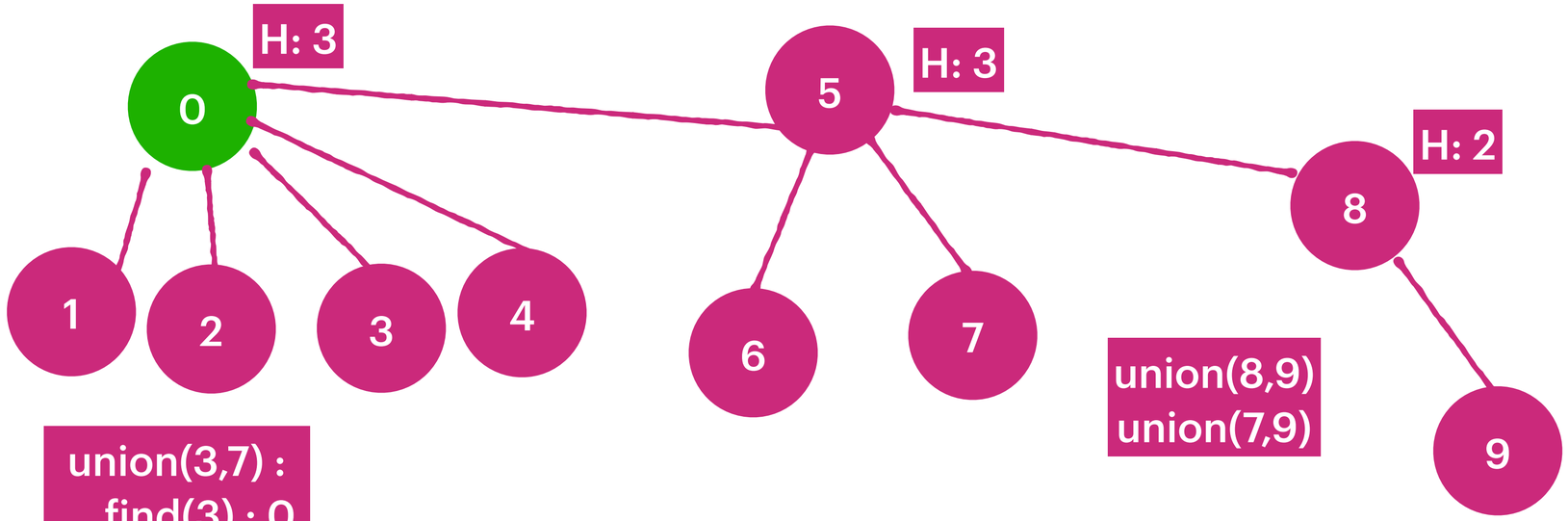
Update the vertex value root

finding Root

Connect(0,4)

Find(0) : 0

find(4) : 4 -> 3 -> 2 -> 1 -> 0 : 0



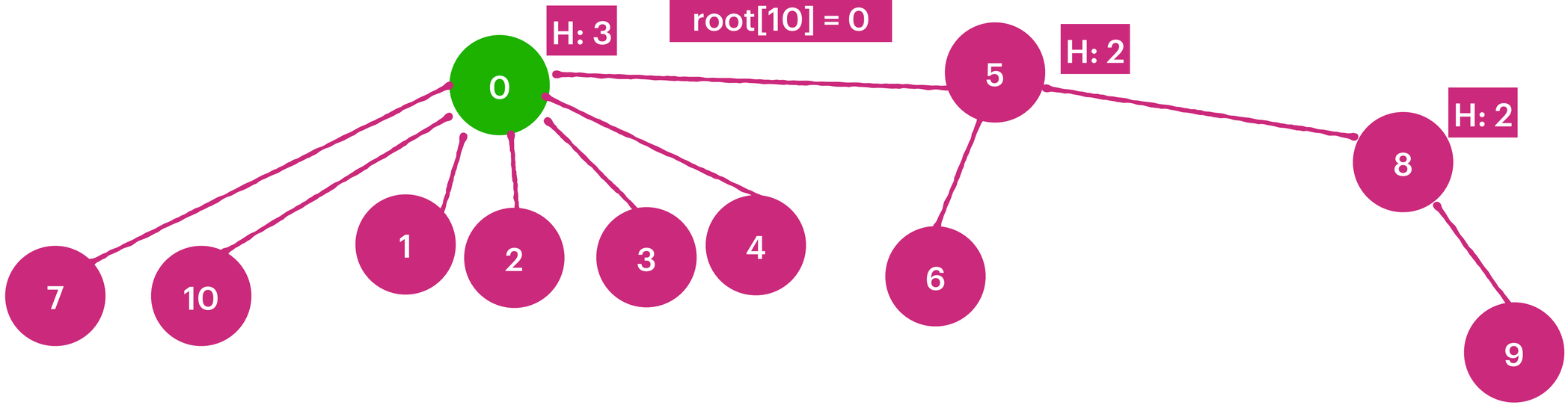
union(3,7) :
find(3) : 0
find(7) : 5
root[5] = 0

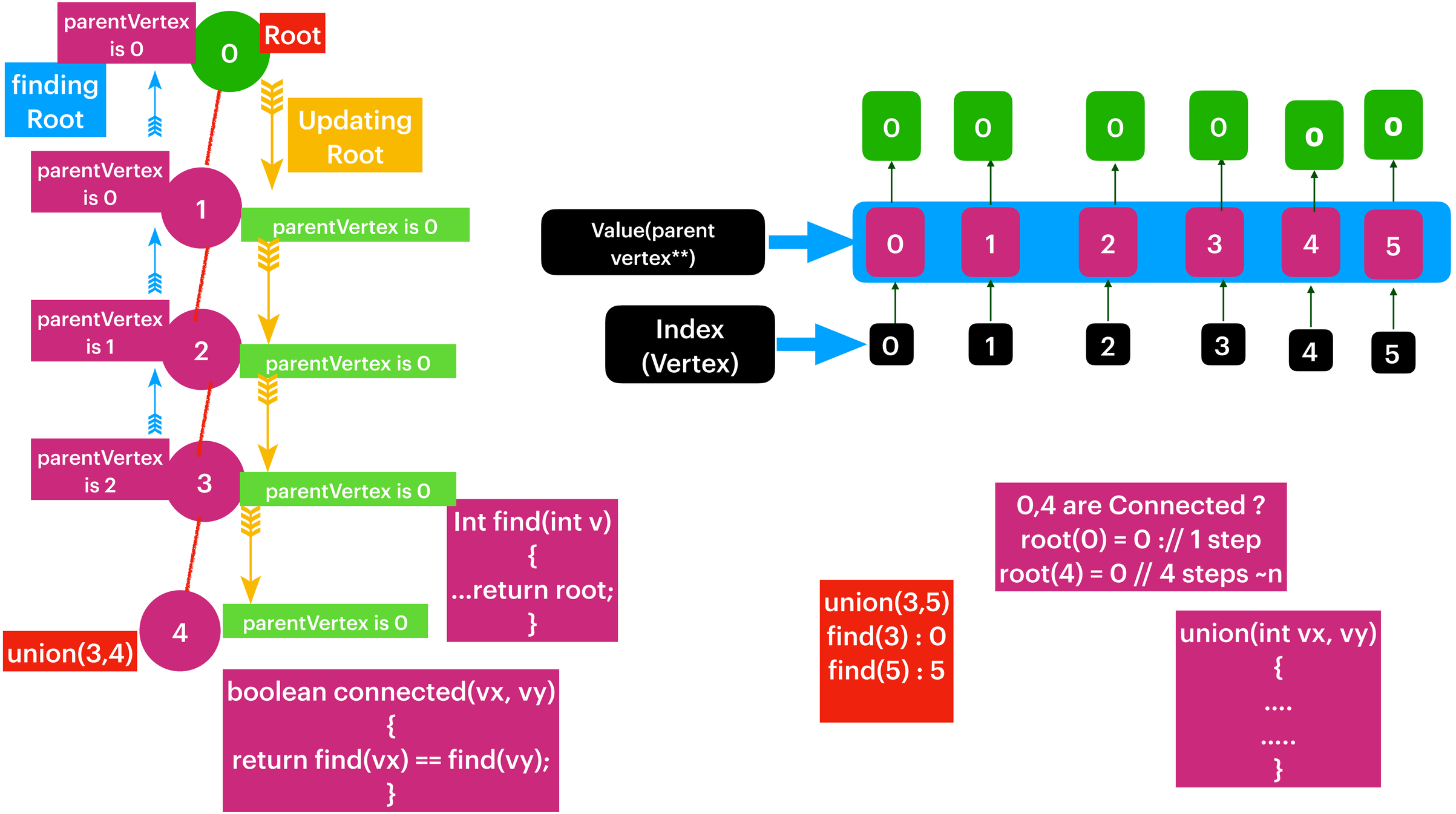
union(8,9)
union(7,9)

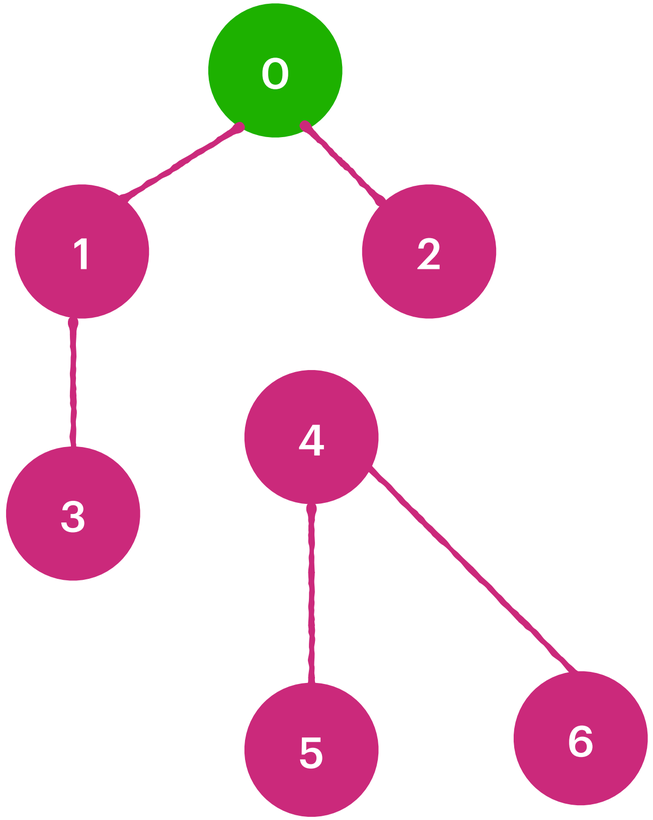
isConnected(1,9) :

find(1) : 0 // 1step
find(9): 8->5->0 : 0 // 3steps
returns true.

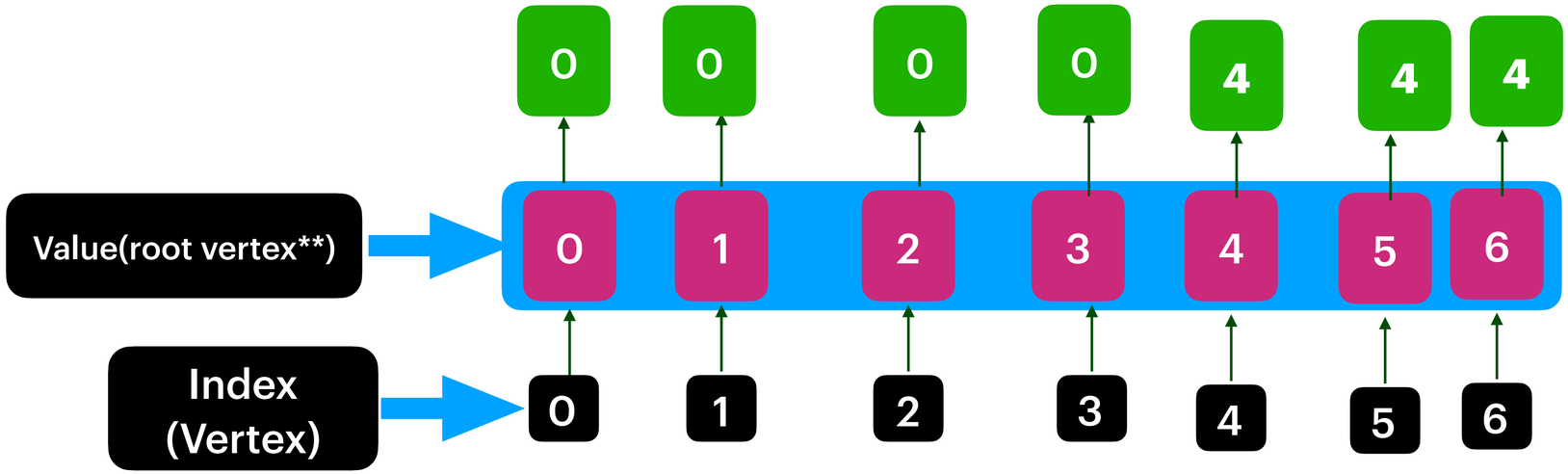
union(7,10)
find(7) : 5 -> 0
find(10) : -> 10
root[10] = 0




parentVertex is 0
0
Root
finding Root
Updating Root
parentVertex is 0
1
parentVertex is 0
parentVertex is 1
2
parentVertex is 0
parentVertex is 2
3
parentVertex is 0
4
parentVertex is 0
union(3,4)
Int find(int v)
{
...return root;
}
boolean connected(vx, vy)
{
return find(vx) == find(vy);
}
Value(parent vertex**)
Index (Vertex)
0 0 0 0 0 0
0 1 2 3 4 5
0 1 2 3 4 5
0,4 are Connected ?
root(0) = 0 :// 1 step
root(4) = 0 // 4 steps ~n
union(3,5)
find(3) : 0
find(5) : 5
union(int vx, vy)
{
....
.....
}



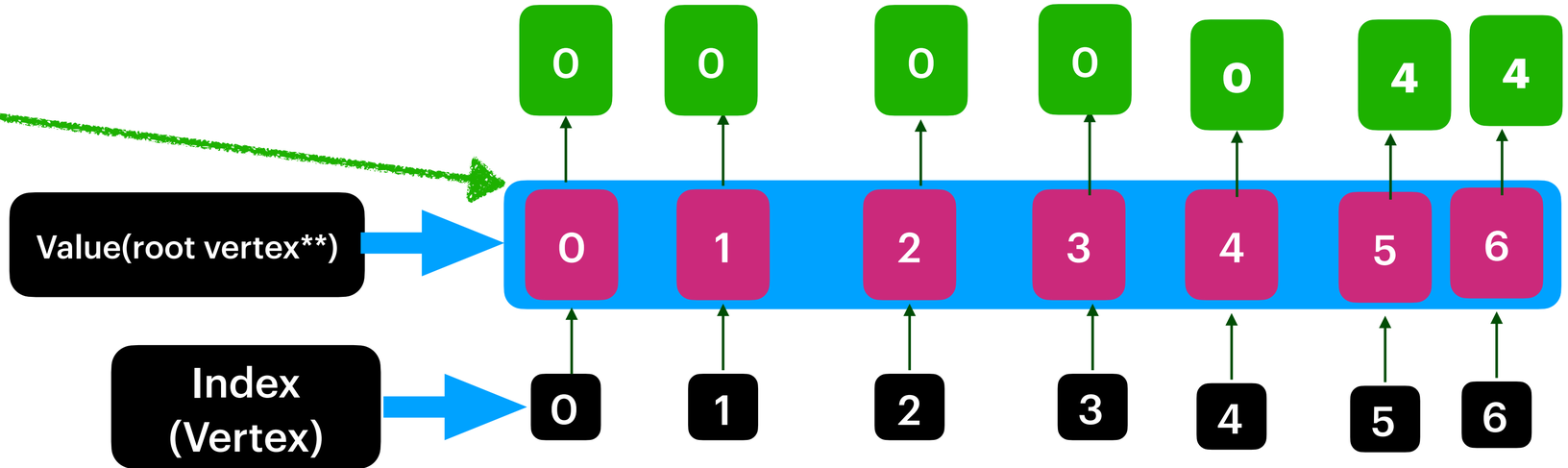
Quick Union :

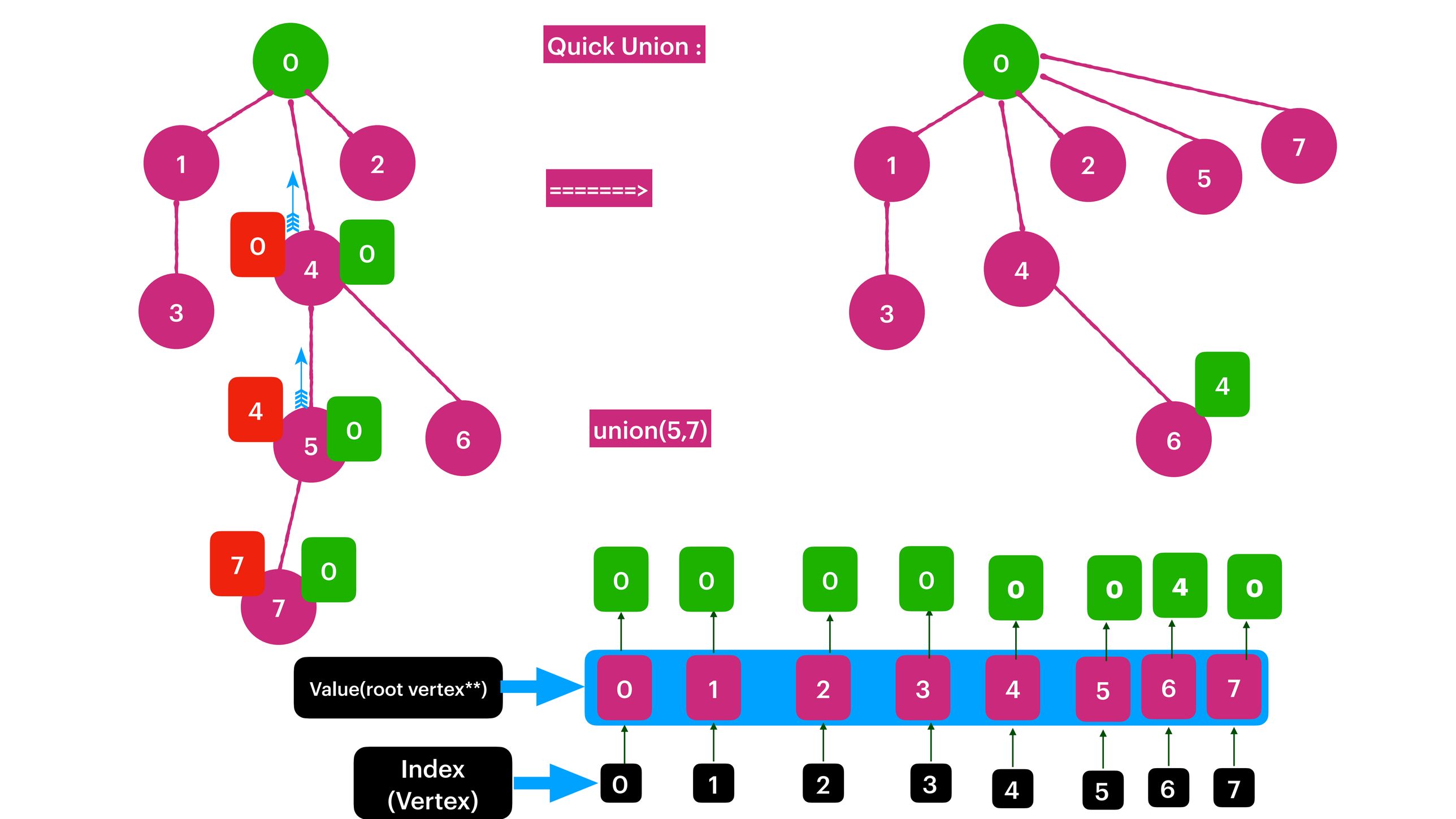


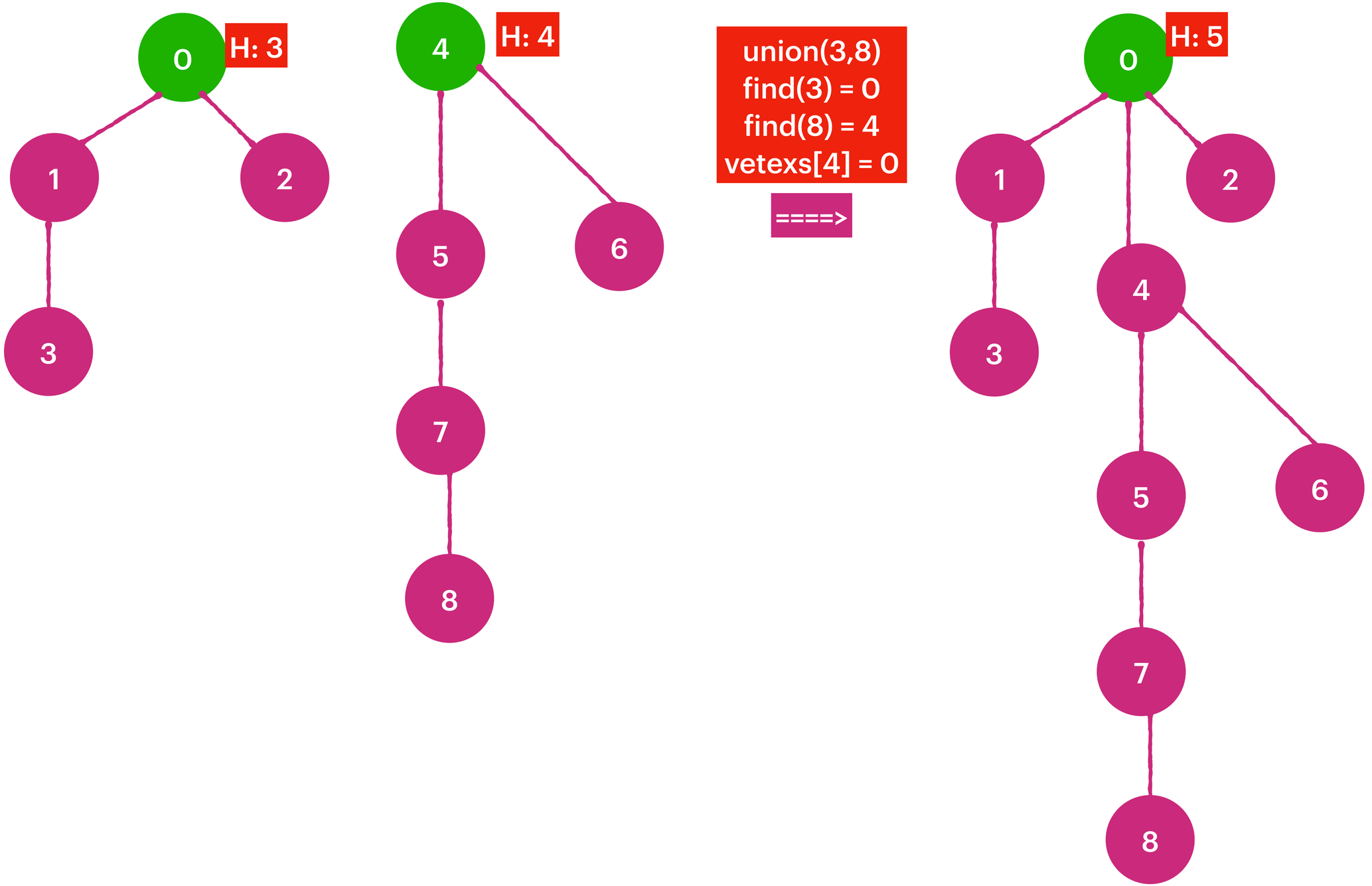
union(0,1)
union(0,2)
union(1,3)
union(4,5)
union(4,6)

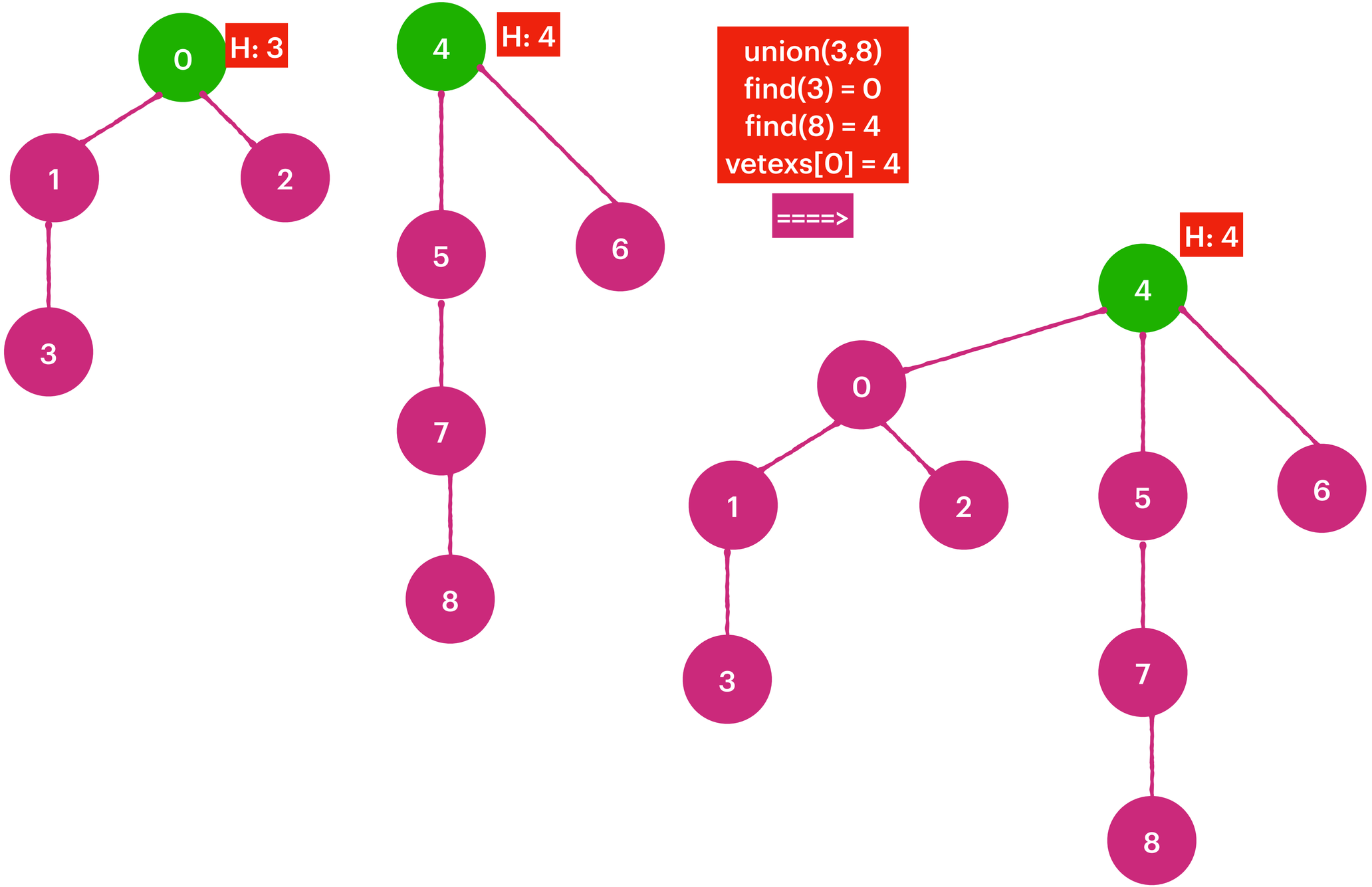
union(0, 1)
Int rootX = find(x); // 0
Int rootY = find(y); // 1
root[rootY] = rootX; //

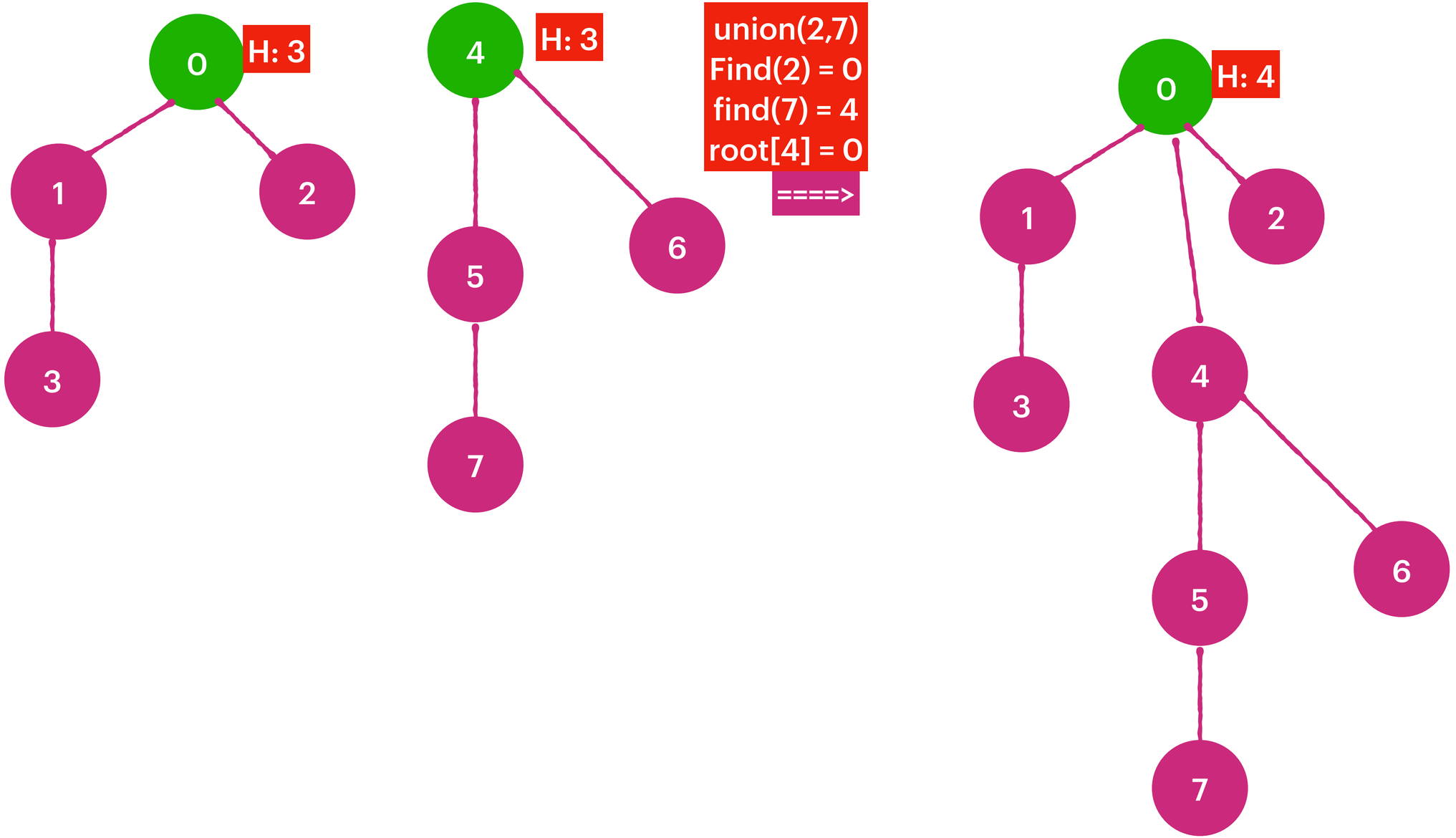
After union(3,5)
Int rootX = find(3) = 0
Int rootY = find(5) = 4
root[rootY] = rootX
root[4] = 0

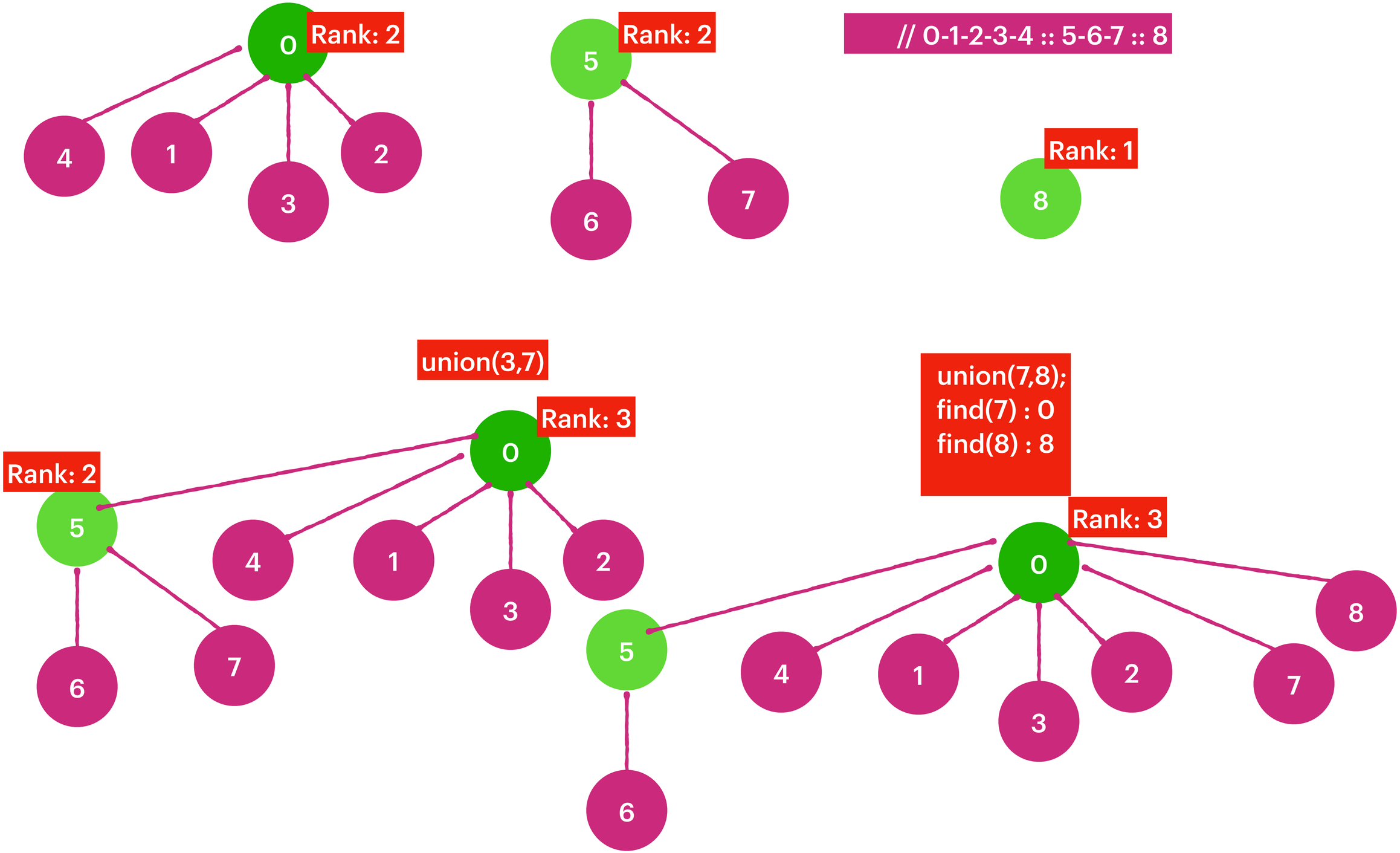


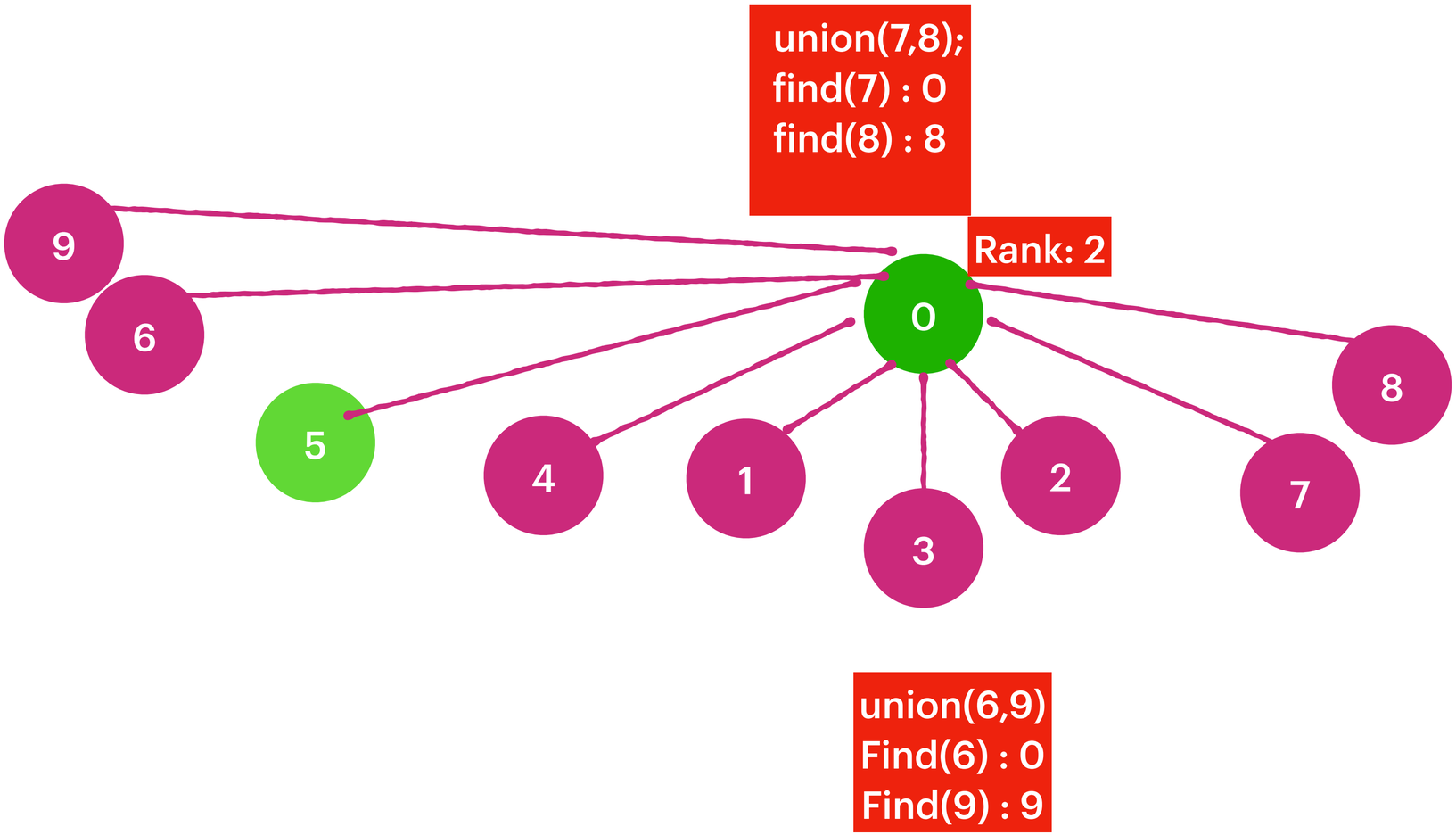

Quick Union :
=======>
union(5,7)
Value(root vertex**)
Index
(Vertex)


0
H: 3
1
2
3
4
H: 4
5
6
7
8
union(3,8)
find(3) = 0
find(8) = 4
vetexs[4] = 0
====>
0
H: 5
1
2
3
4
5
6
7
8


0
H: 3
1
2
3
4
H: 4
5
6
7
8
union(3,8)
find(3) = 0
find(8) = 4
vetexs[0] = 4
====>
4
H: 4
0
1
2
5
6
3
7
8


0
H: 3
1
2
3
4
H: 3
5
6
7
union(2,7)
Find(2) = 0
find(7) = 4
root[4] = 0
====>
0
H: 4
1
2
3
4
5
6
7


0
Rank: 2
4
1
3
2
5
Rank: 2
6
7
// 0-1-2-3-4 :: 5-6-7 :: 8
Rank: 1
8
union(3,7)
Rank: 3
0
Rank: 2
5
4
1
3
2
6
7
5
6
union(7,8);
find(7) : 0
find(8) : 8
Rank: 3
0
4
1
3
2
8
7


union(7,8);
find(7) : 0
find(8) : 8
Rank: 2
9
6
0
8
5
4
1
2
7
3
union(6,9)
Find(6) : 0
Find(9) : 9

Graph Valid Tree

You have a graph of n nodes labeled from 0 to n - 1. You are given an integer n and a list of edges where edges[i] = [ai, bi] indicates that there is an undirected edge between nodes ai and bi in the graph. Return true if the edges of the given graph make up a valid tree, and false otherwise.

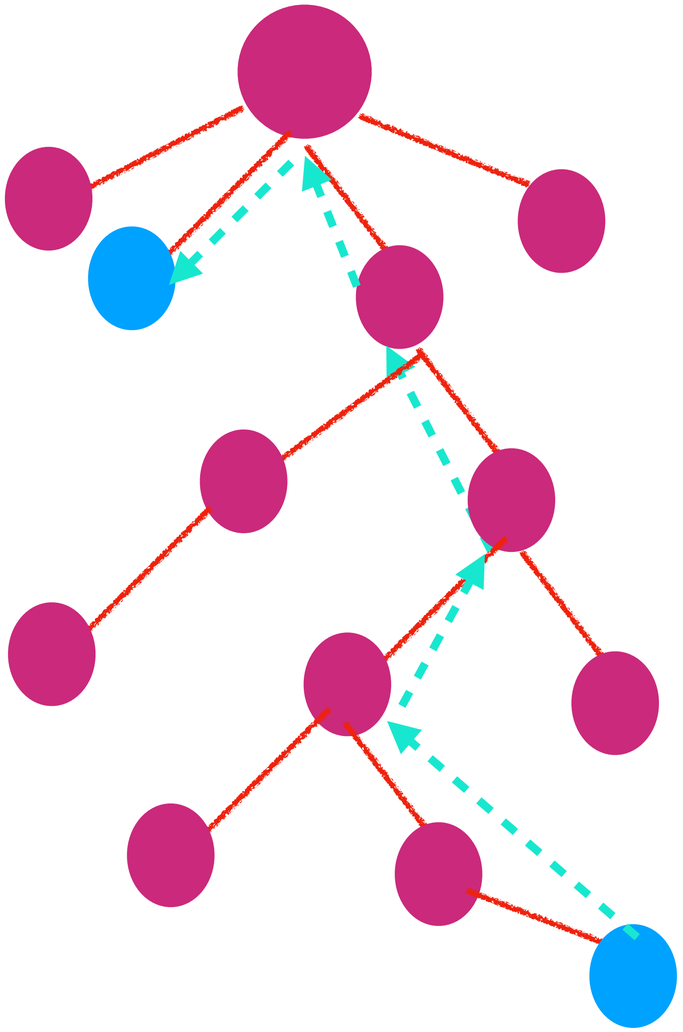
Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
Output: true

Input: n = 5,
edges = [[0,1],[0,4],[1,4],[2,3]]
Output: false

Input: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
Output: false

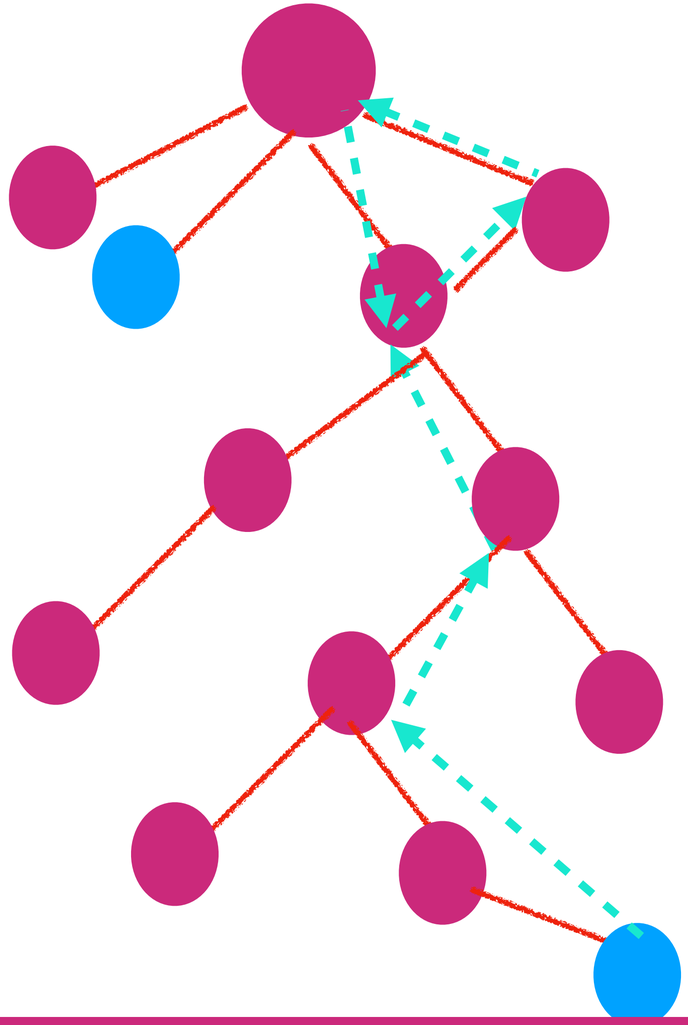
Constraints :
1 <= n <= 2000
0 <= edges.length <= 5000
edges[i].length == 2
0 <= ai, bi < n
ai != bi (No Self loop)

Graph with Valid Tree



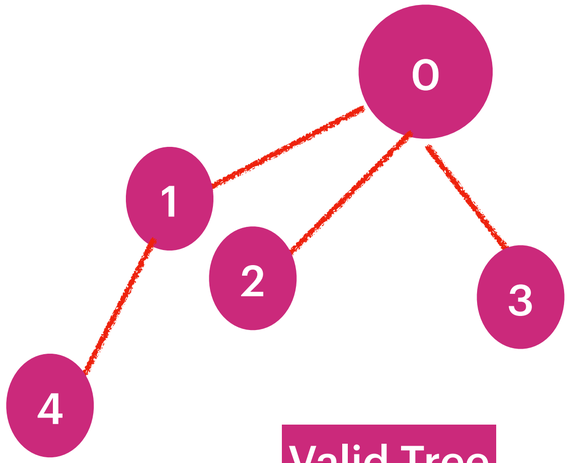
Its a valid tree because
From one branch we can move to another branch

Graph with Invalid Tree



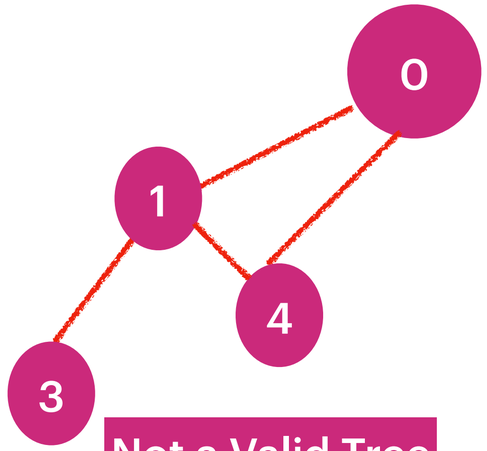
Its not a valid tree because
, a child node has more than one immediate parent so it causes cycle.

Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
Output: true



Valid Tree

Input: n = 5,
edges = [[0,1],[0,4],[1,4],[1,3]]
Output: false

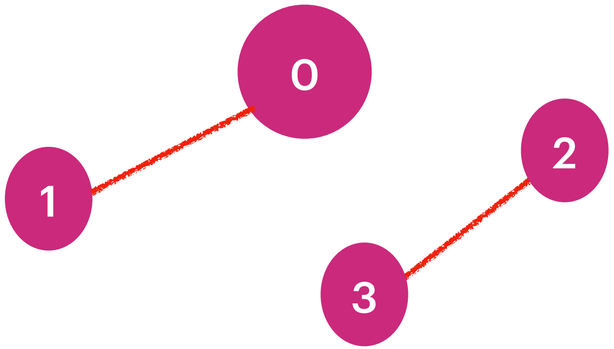


Not a Valid Tree

Input: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
Output: false

For n vertexes we can have at max n-1 edges.
Here n = 5 and edges = 5 so Not a Valid Tree.

n=4 [[0,1], [2,3]]



Its not a Valid Tree because

[[0,1],[0,4],[1,4],[1,3]] n = 5

0[0]
1[1]
2[2]
3[3]
4[4]

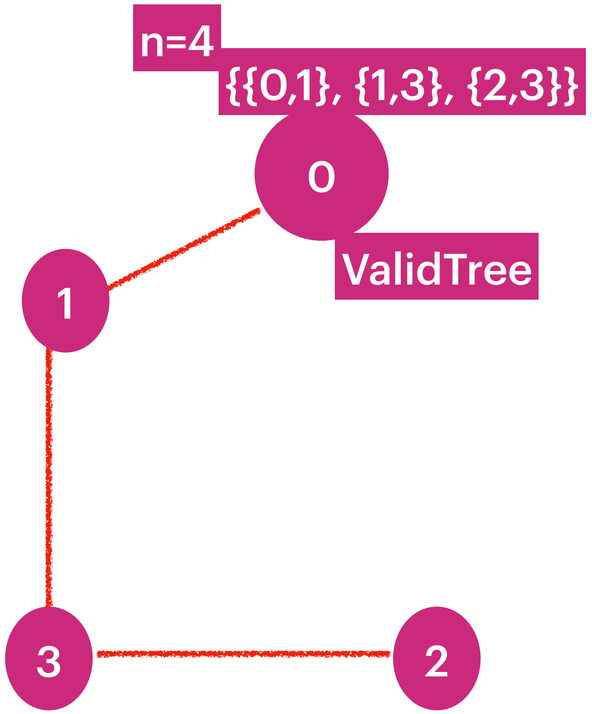
[0,1] => 0[0]-1[0]

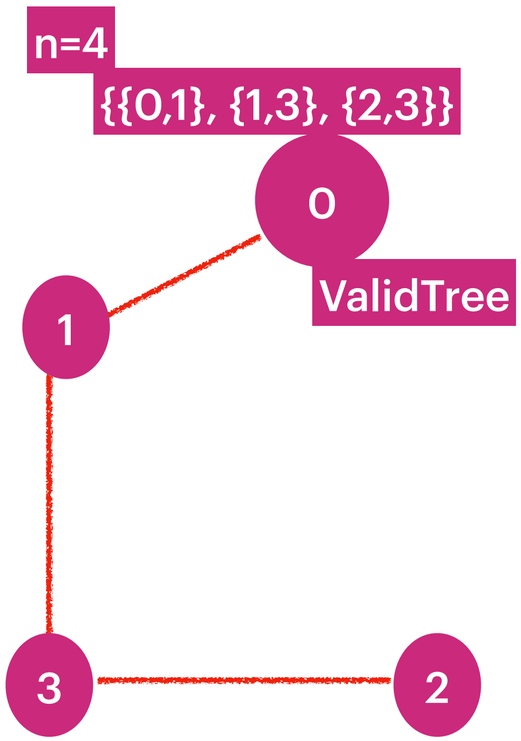
[0,4] => 0[0]-1[0]-4[0]

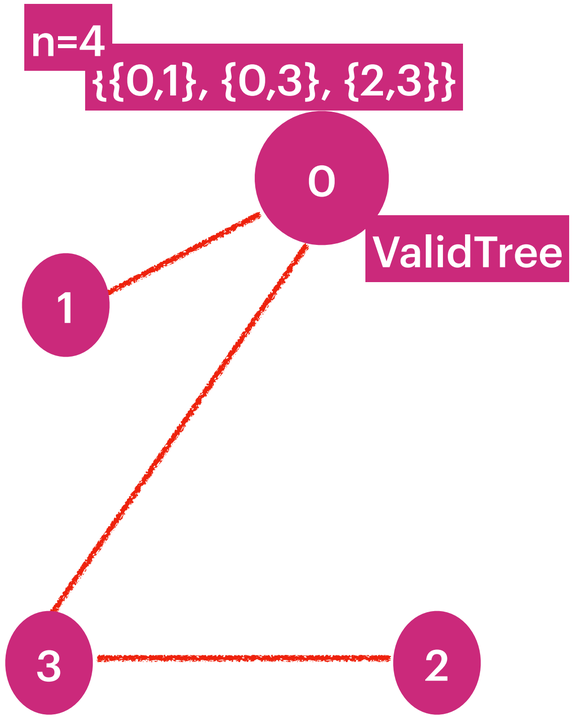
[1,4] => 1[0] - 4[0] They are already connected : Its a loop return false

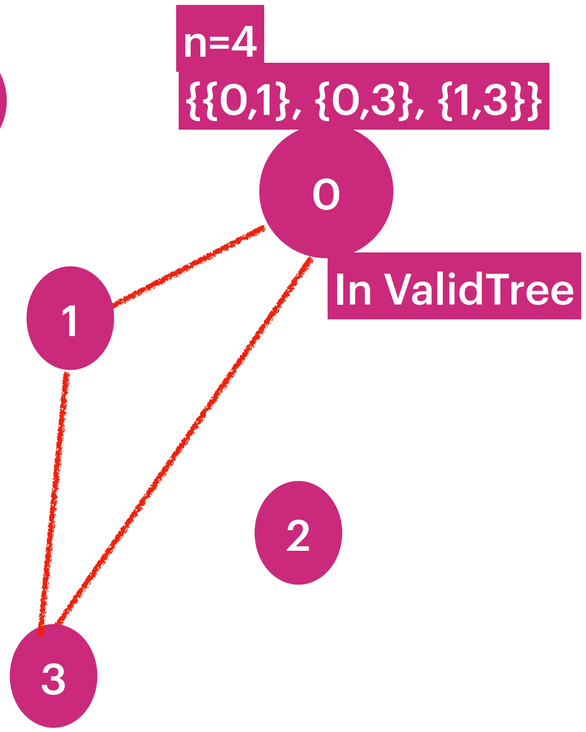
Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]

0[0]
1[1]
2[2]
3[3]
4[4]
[0,1] => 0[0]-1[0]
[0-2] => 0[0]-1[0]- 2[0]
[0-3] => 0[0]-1[0]- 2[0]-3[0]
[1-4] =>0[0]-1[0]- 2[0]-3[0]-4[0] Its a valid Tree : return true


n=4
{{0,1}, {1,3}, {2,3}}
0
ValidTree
1
3
2


n=4
{{0,1}, {1,3}, {2,3}}
0
ValidTree
1
3
2


n=4
{{0,1}, {0,3}, {2,3}}
0
ValidTree
1
3
2


n=4
{{0,1}, {0,3}, {1,3}}
0
In ValidTree
1
2
3

Number of Connected Components in an Undirected Graph

You have a graph of n nodes. You are given an integer n and an array edges where edges[i] = [ai, bi] indicates that there is an edge between ai and bi in the graph.

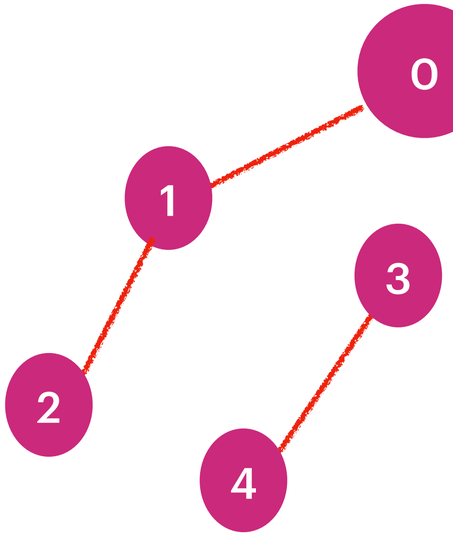
Return the number of connected components in the graph.

Input: n = 5, edges = [[0,1],[1,2],[3,4]]
Output: 2

Input: n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]
Output: 1

Constraints : 1 <= n <= 2000
1 <= edges.length <= 5000
edges[i].length == 2
0 <= ai <= bi < n
ai != bi
There are no repeated edges.

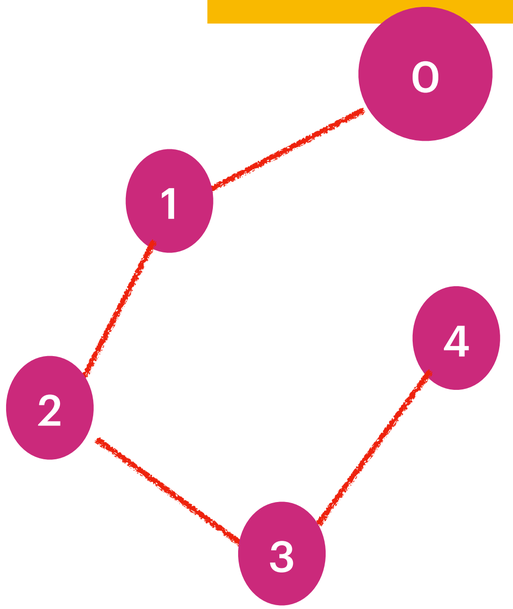
Input: n = 5, edges = [[0,1],[1,2],[3,4]]
Output: 2



edgeCount = 5

[0,1] = edgeCount - - = 4
[1,2] = edgeCount - - = 3
[3,4] = edgeCount - - = 2

Input: n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]
Output: 1



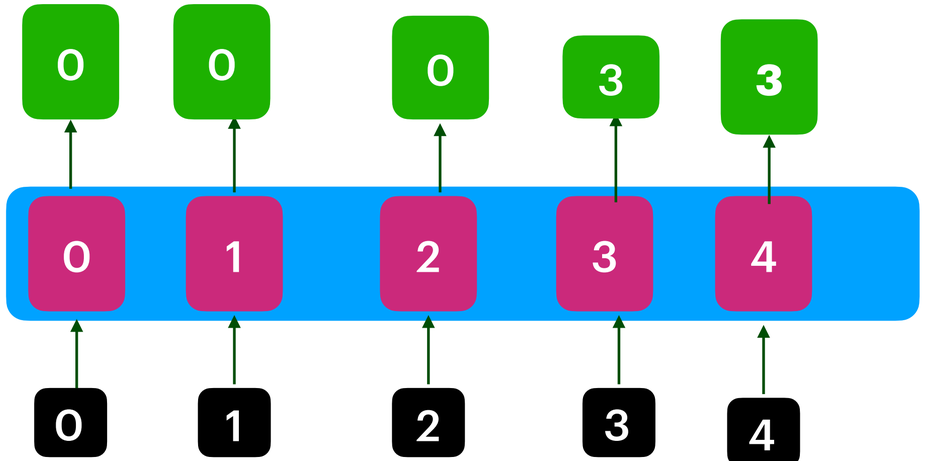
edgeCount = n = 5
edgeCount
[0,1] = edgeCount- - = 4

[1,2] = edgeCount- - = 3

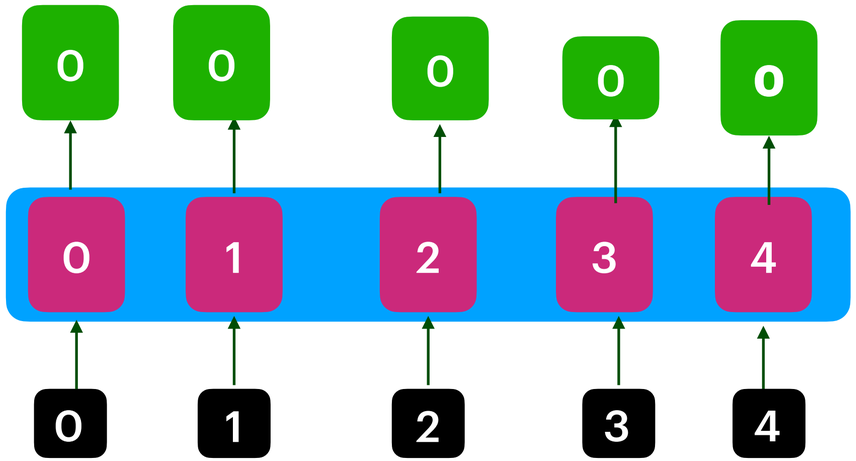
[2,3] = edgeCount- - = 2

[3,4] = edgeCount- - = 1

Connected Components / Paths. = 2



Connected Components / Paths. = 1



Number of Provinces

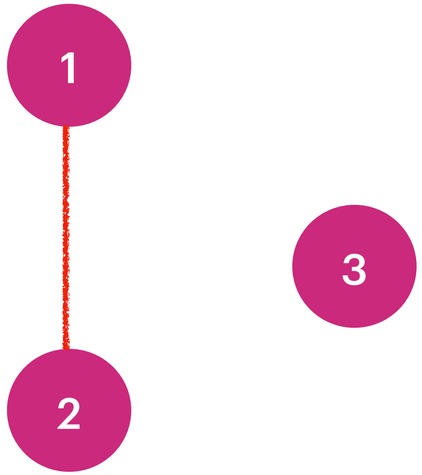
There are n cities. Some of them are connected, while some are not. If city a is connected directly city b, and city b is connected directly with city c, then city a is connected indirectly with city c.
A province is a group of directly or indirectly connected cities and no other cities outside of the group.
You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the ith city and the jth city are directly connected, and isConnected[i][j] = 0 otherwise.
Return the total number of provinces.

Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]
Output: 2

Input: isConnected = [[1,0,0],[0,1,0],[0,0,1]]
Output: 3

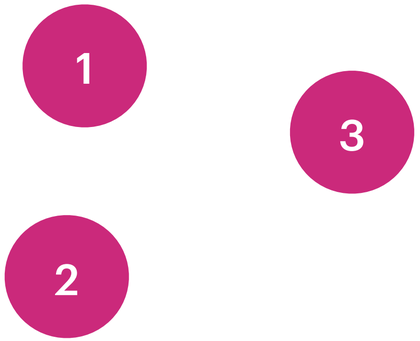
Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]
Output: 2

Count = 3
Count - - = 2



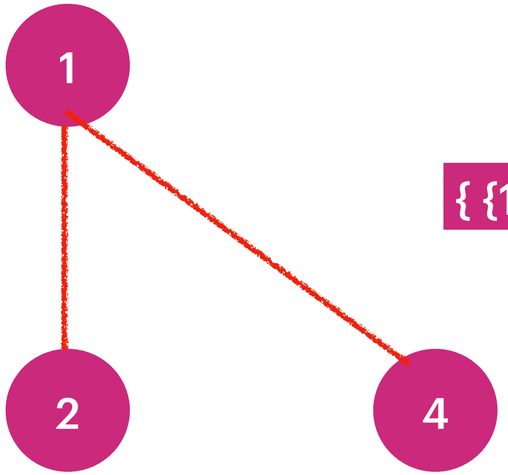
	C0	C1	C2
	[0,0]	[0,1]	[0,2]
R0	1	1	0
	[1,0]	[1,1]	[1,2]
R1	1	1	0
	[2,0]	[2,1]	[2,2]
R2	0	0	1

Input: isConnected = [[1,0,0],[0,1,0],[0,0,1]]
Output: 3



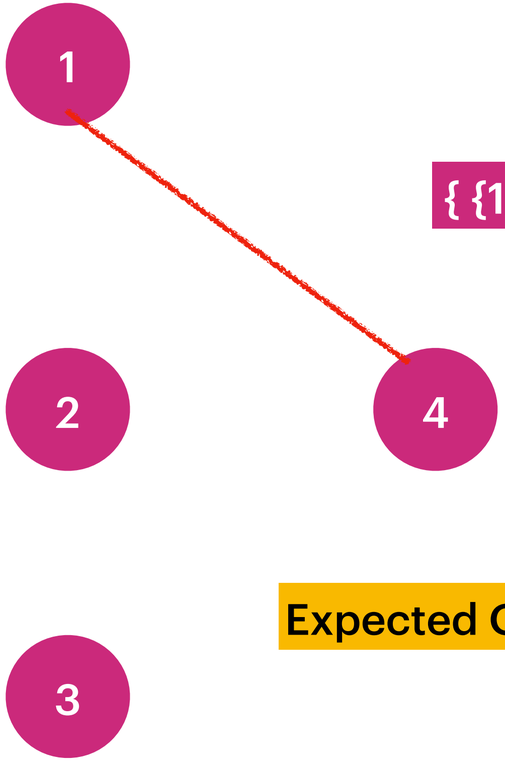
	C0	C1	C2
	[0,0]	[0,1]	[0,2]
R0	1	0	0
	[1,0]	[1,1]	[1,2]
R1	0	1	0
	[2,0]	[2,1]	[2,2]
R2	0	0	1

Count = 3
Count - - = 2



{ {1,1,0,1 }, {1,1,0,0},{0,0,1,0},{1,0,0,1} }

Expected Output : 2



{ {1,0,0,1 }, {0,1,0,0},{0,0,1,0},{1,0,0,1} }

Expected Output : 3

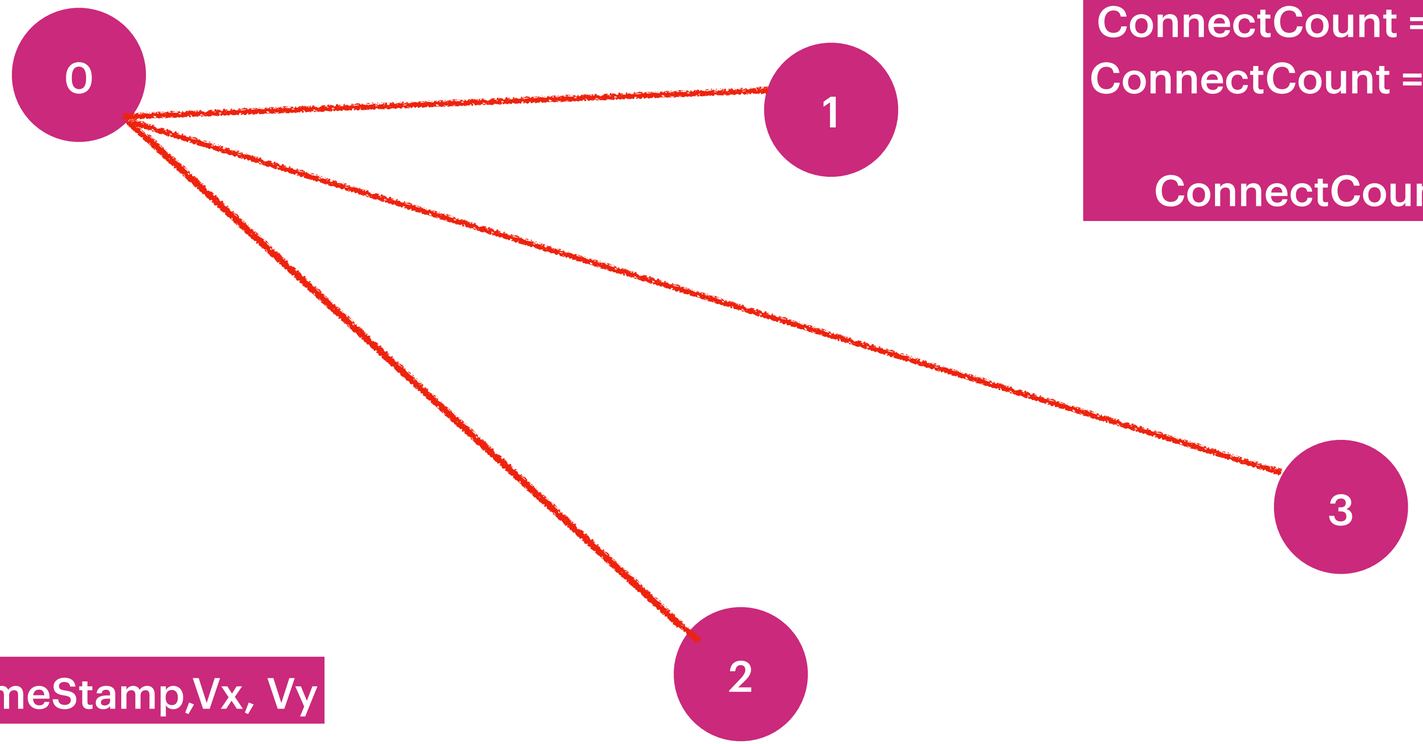
The Earliest Moment When Everyone Become Friends in Instagram

There are n people in a social group labeled from 0 to n - 1.
You are given an array logs where logs[i] = [timestampi, xi, yi] indicates that xi and yi will be friends at the time timestampi.
Friendship is symmetric. That means if a is friends with b, then b is friends with a. Also, person a is acquainted with a person b if a is friends with b, or a is a friend of someone acquainted with b.
Return the earliest time for which every person became acquainted with every other person. If there is no such earliest time, return -1.

Input: logs = [[20190101,0,1],[20190104,3,4],[20190107,2,3],[20190211,1,5],[20190224,2,4],[20190301,0,3],[20190312,1,2],[20190322,4,5]], n = 6
Output: 20190301

Input: logs = [[0,2,0],[1,0,1],[3,0,3],[4,1,2],[7,3,1]], n = 4
Output: 3

[[9,3,0],[0,2,1],[8,0,1],[1,3,2],[2,2,0],[3,3,1]]
4

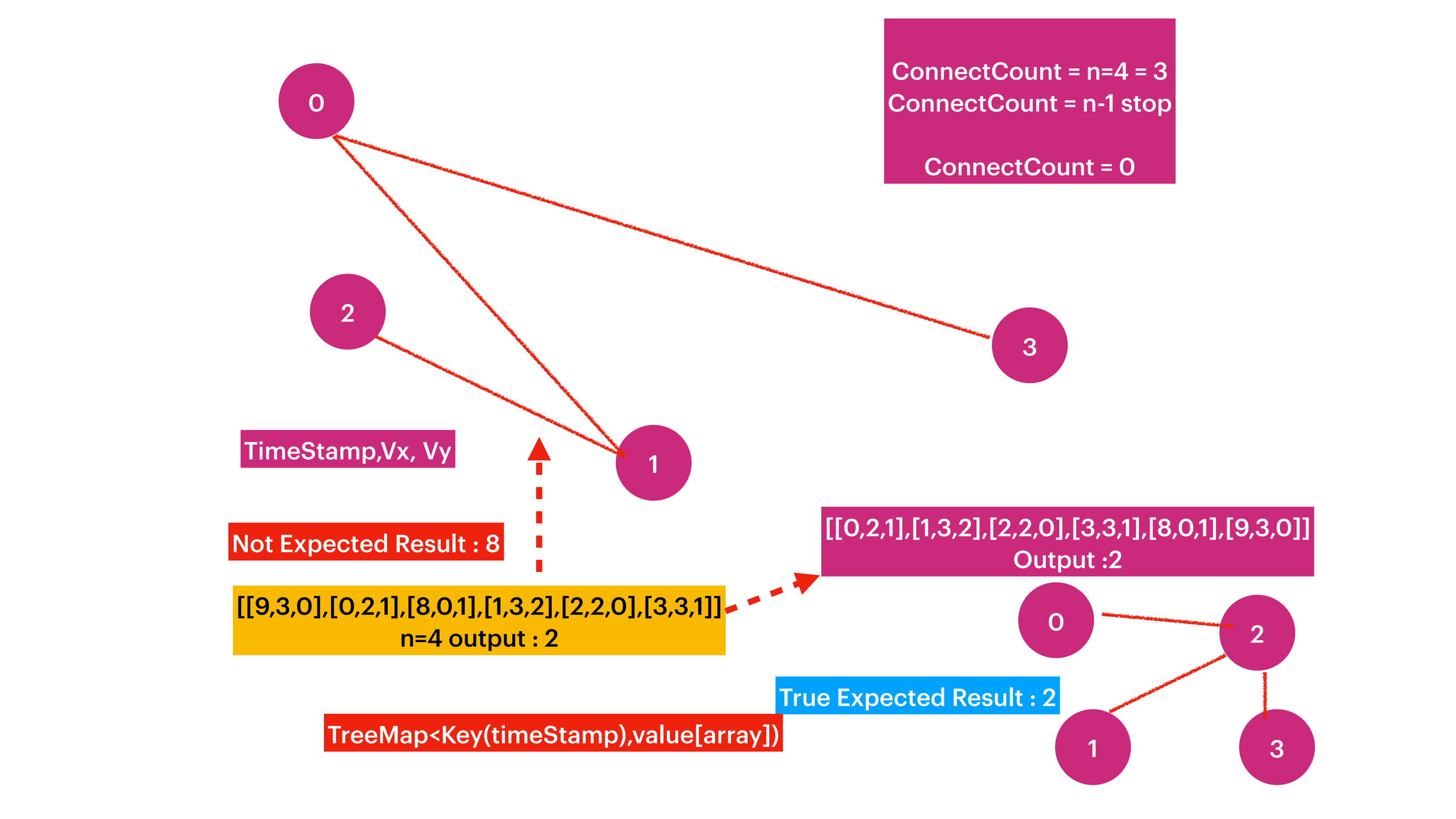


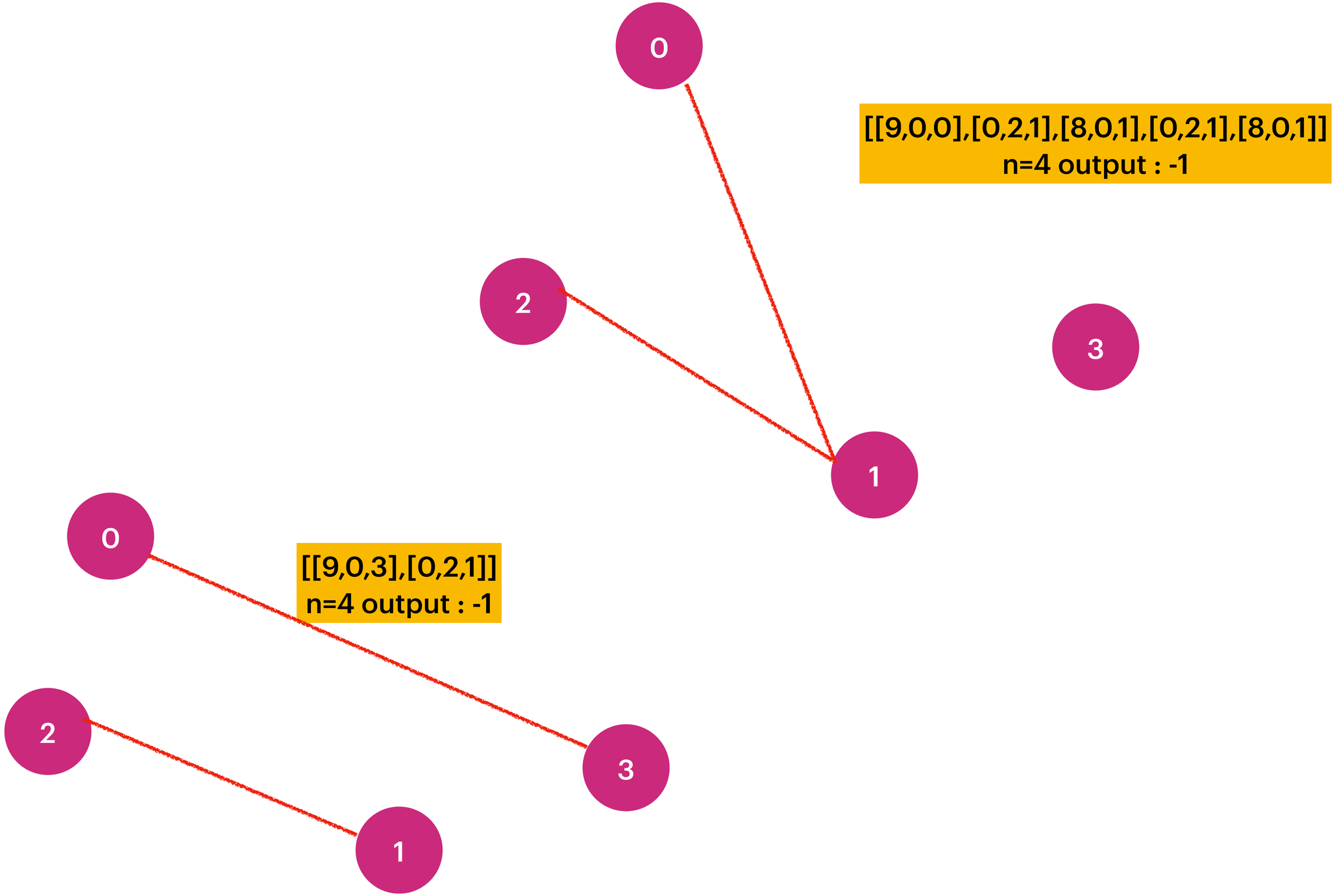
ConnectCount = n=4 = 3
ConnectCount = n-1 stop

ConnectCount = 0

TimeStamp,Vx, Vy

Input: logs = [[0,2,0],[1,0,1],[3,0,3],[4,1,2],[7,3,1]], n = 4
Output: 3

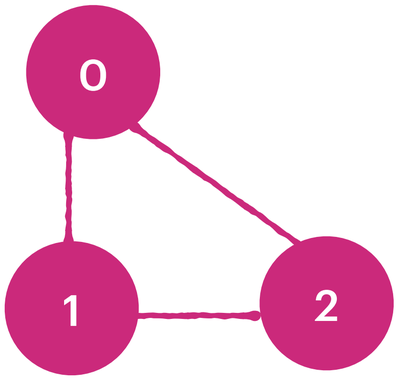

ConnectCount = n=4 = 3
ConnectCount = n-1 stop
ConnectCount = 0
0
2
3
1
TimeStamp,Vx, Vy
Not Expected Result : 8
[[9,3,0],[0,2,1],[8,0,1],[1,3,2],[2,2,0],[3,3,1]]
n=4 output : 2
[[0,2,1],[1,3,2],[2,2,0],[3,3,1],[8,0,1],[9,3,0]]
Output :2
0
2
1
3
True Expected Result : 2
TreeMap<Key(timeStamp),value[array])


0
2
3
1
[[9,0,0],[0,2,1],[8,0,1],[0,2,1],[8,0,1]]
n=4 output : -1
0
2
3
1
[[9,0,3],[0,2,1]]
n=4 output : -1

Design Graph :

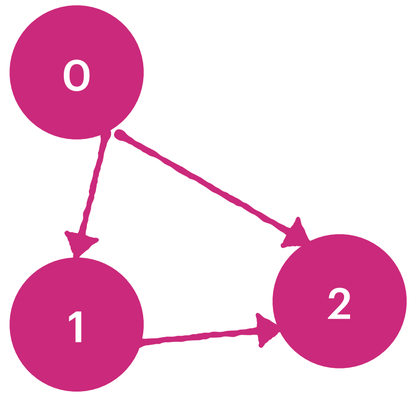
- AddVertex → O(1)
- AddEdge → O(1)
- RemoveEdge → O(1)
- Remove Vertex → O(V)
- Print → O(V+E)

n = 3, edges = [[0,1],[0,2],[1,2],[2,1],[2,0],[1,0]]

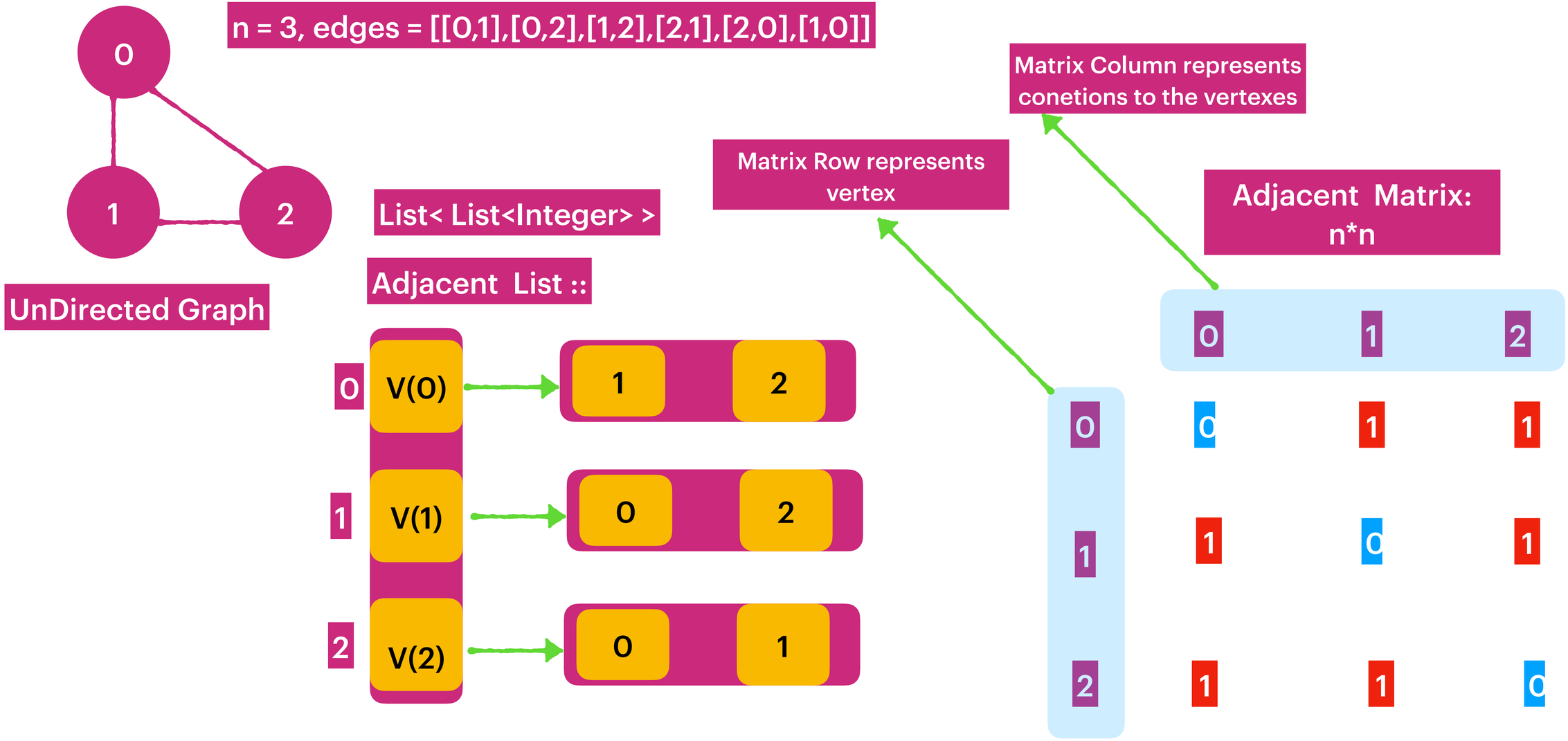


UnDirected Graph

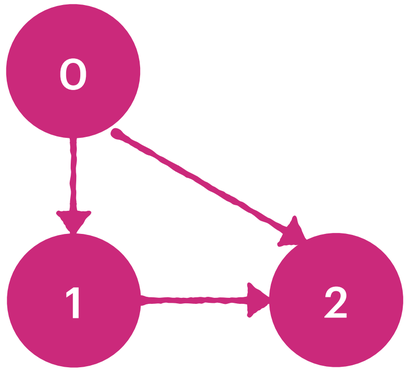
n = 3, edges = [[0,1],[1,2],[0,2]]



Directed Graph


n = 3, edges = [[0,1],[0,2],[1,2],[2,1],[2,0],[1,0]]
0
1
2
UnDirected Graph
List< List<Integer> >
Adjacent List ::
0
V(0)
1
2
1
V(1)
0
2
2
V(2)
0
1
Matrix Row represents vertex
Matrix Column represents conetions to the vertexes
Adjacent Matrix: n*n
0
1
2
0
0
1
1
1
1
0
1
2
1
1
0

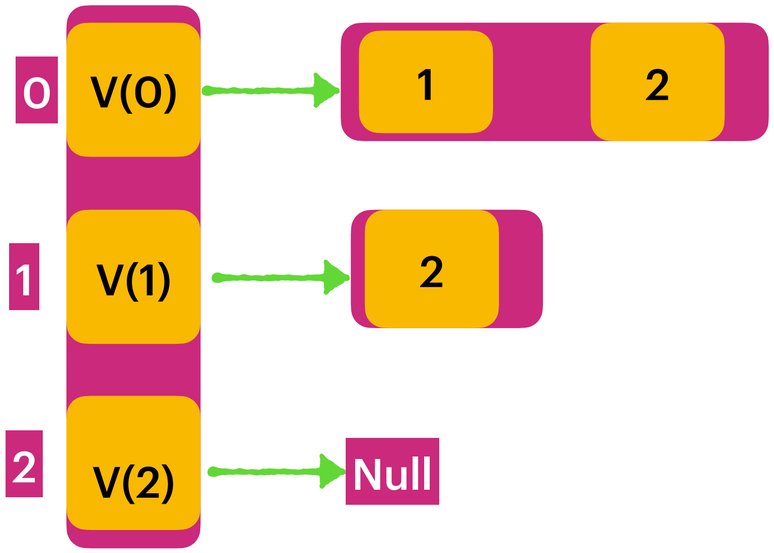
n = 3, edges = [[0,1],[1,2],[0,2]]



Directed Graph

List< List<Integer> >

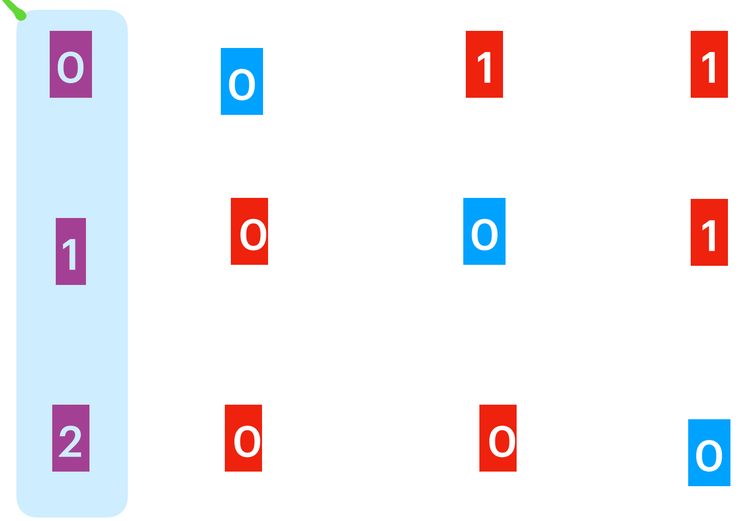
Adjacent List ::



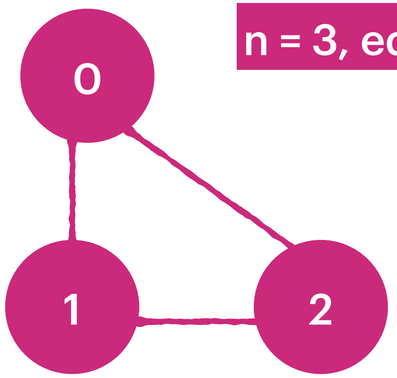
Matrix Row represents vertex

Matrix Column represents conetions to the vertexes

Adjacent Matrix: n*n



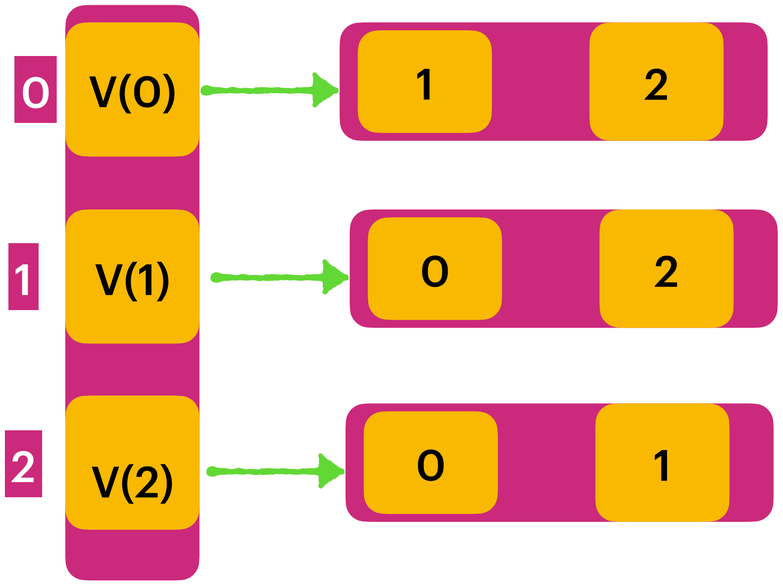
	Adjacent List	Adjacent Matrix
Add Vertex	O(1)	O(n^2)
Remove Vertex	O(V+E)	O(n^2
Add Edge	O(1)	O(1)
Remove Edge	O(E)	O(1)



n = 3, edges = [[0,1],[0,2],[1,2],[2,1],[2,0],[1,0]]

List< LinkedHashSet >

Adjecent Hashing

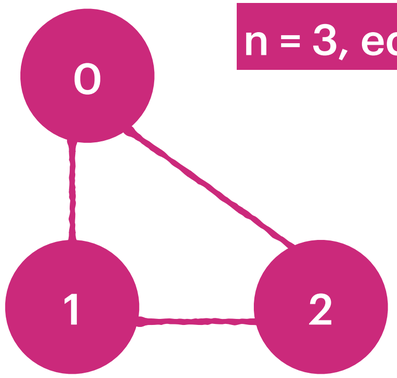



```
removeVertex(2) :
for(v:o to  Vn )
{
if(list.get(v).contains(2))
{
list.get(v).remove(2)
}
}
list.remove(2);
```

Hashing

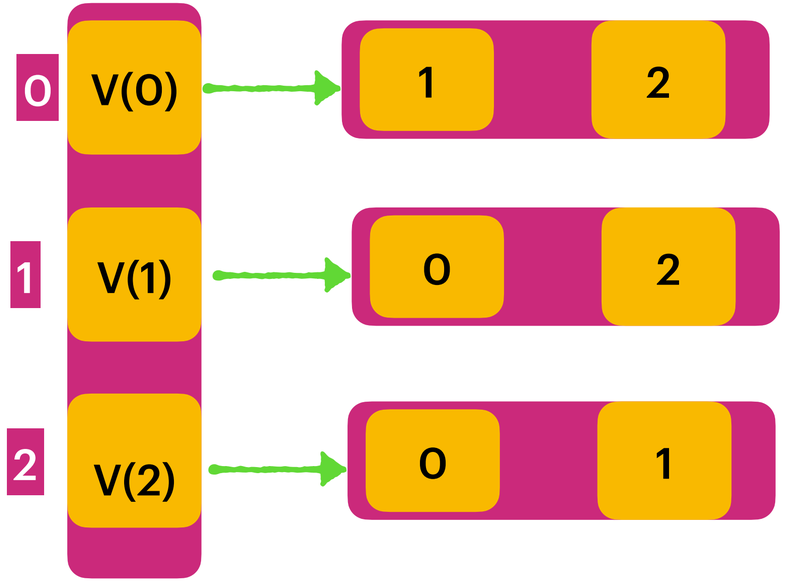
Add Vertex	O(1)
Remove Vertex	O(V)
Add Edge	O(1)
Remove Edge	O(1)

Print O(V+E)



n = 3, edges = [[0,1],[0,2],[1,2],[2,1],[2,0],[1,0]]

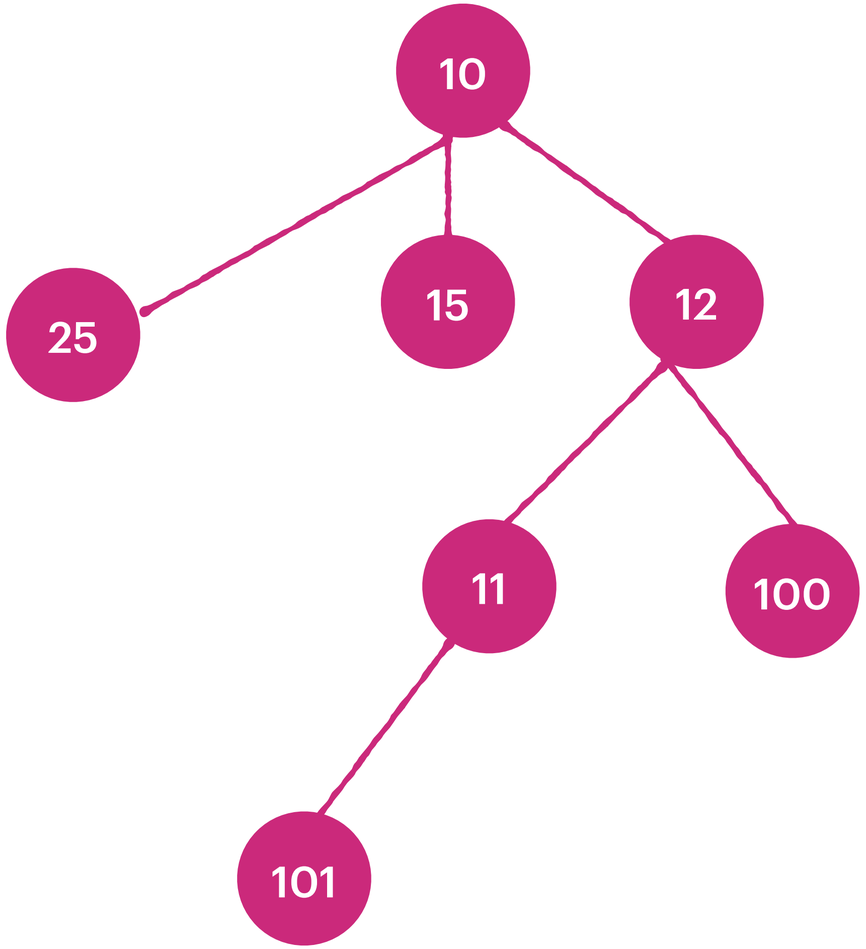
List< LinkedHashSet >



Is this can be applied in RealWorld ?
Limitations ***
1. Graph should be fixed .
2. Demands vertexes must Be in the range of 0 to n-1
3. Occupy the fixed memory.

Hashing

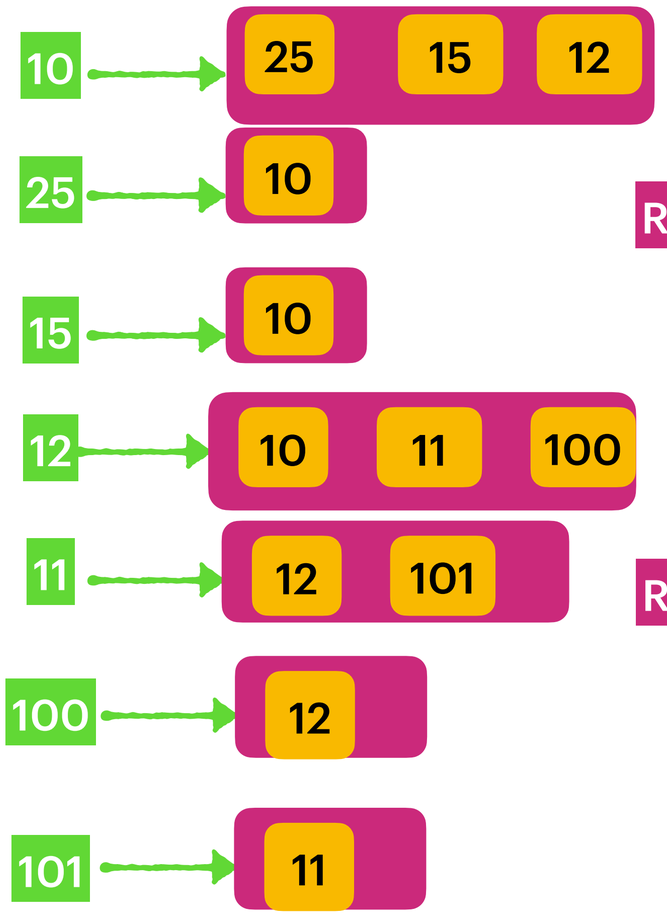
Add Vertex	O(1)
Remove Vertex	O(V)
Add Edge	O(1)
Remove Edge	O(1)
Print	O(V+E)




```
10 => [25, 15, 12]
25 => [10]
15 => [10]
12 => [10, 11, 100]
11 => [12, 101]
100 => [12]
101 => [11]
```

Map<VertexNumber, Connections>
Map<Integer, Set<Integer>>

Hashing



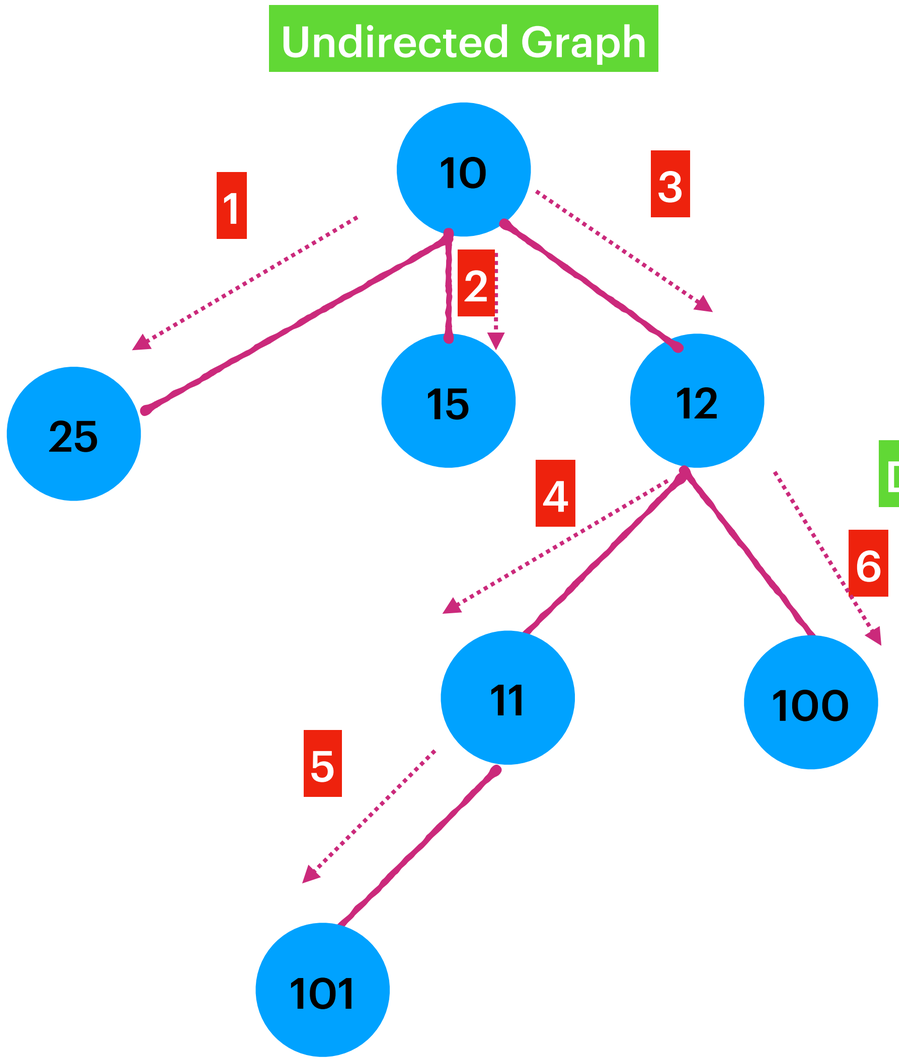
Add Vertex O(1)

Remove Vertex O(E)

Add Edge O(1)

Remove Edge O(1)

Print O(V+E)



Depth First Search

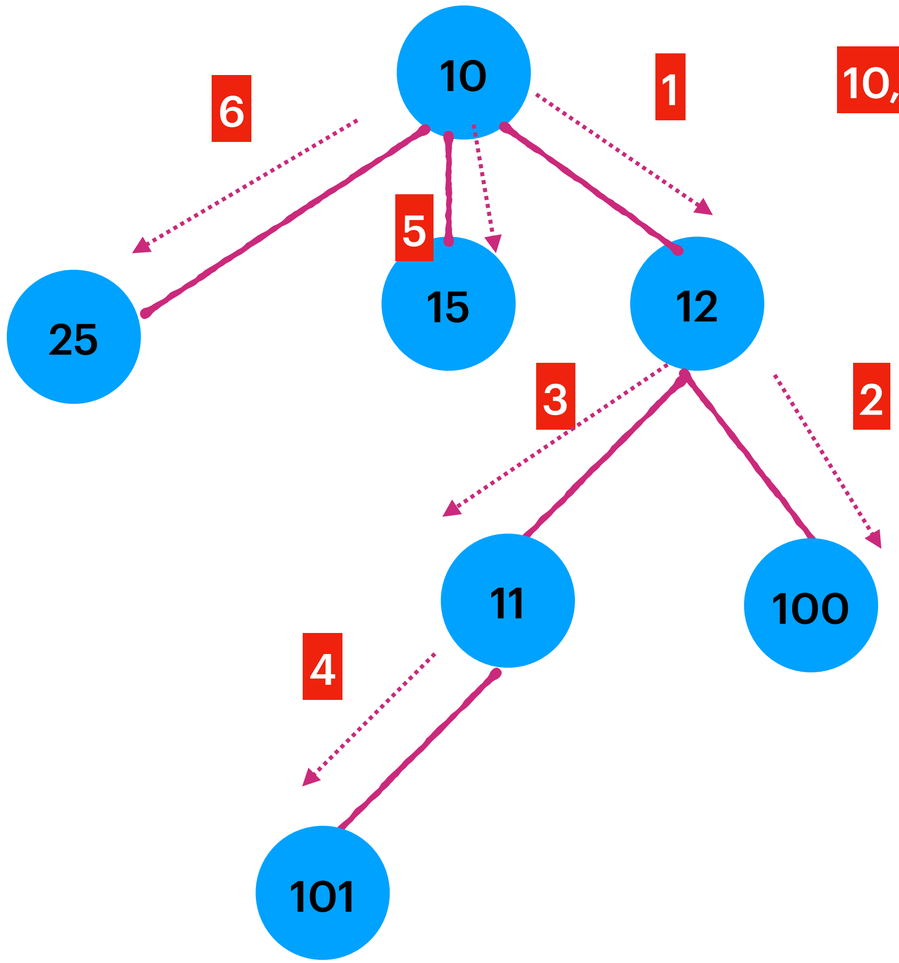
Moving => from root- left to right

10,25,15,12,11,101,100

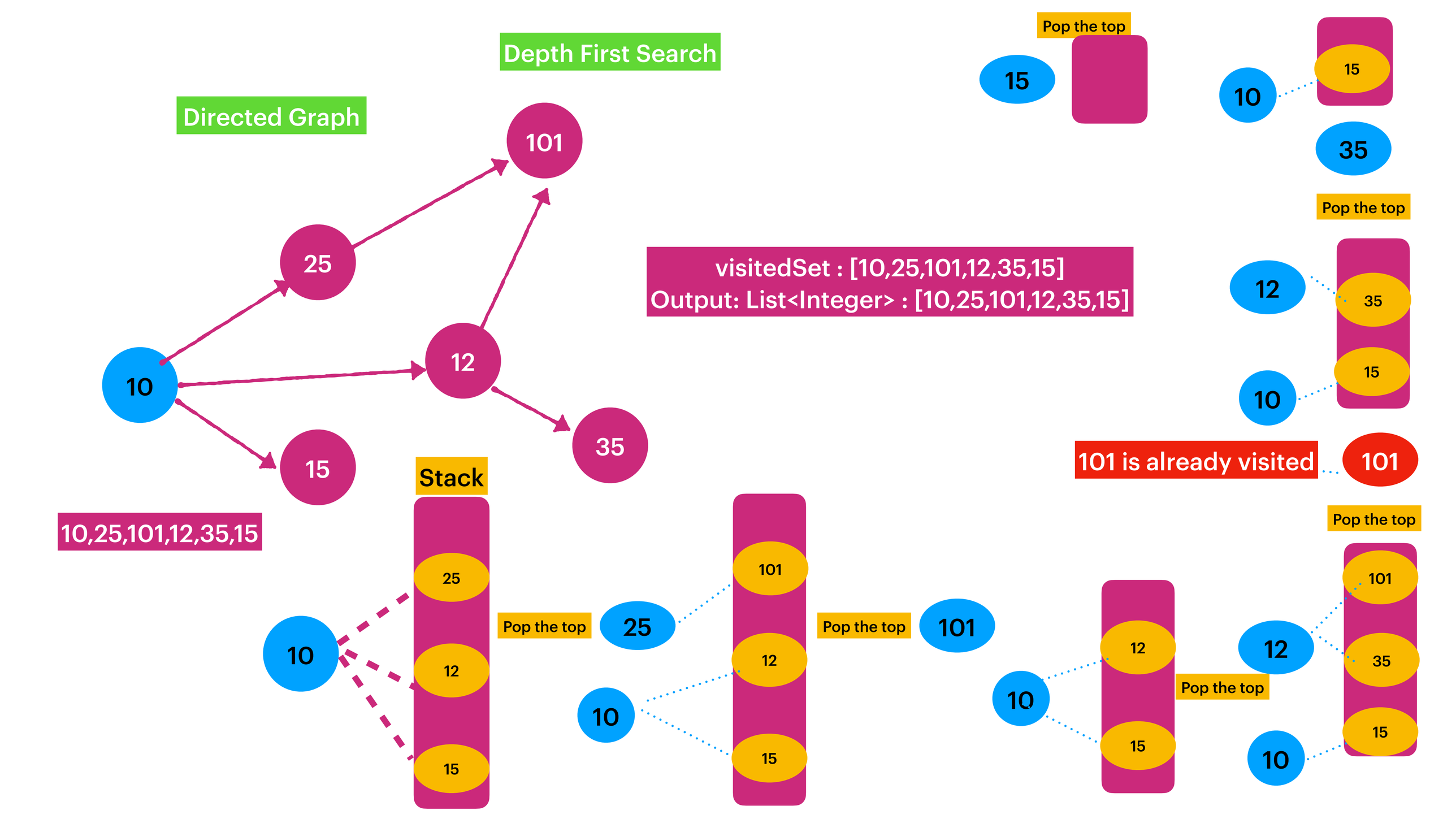
Undirected Graph

Depth First Search

Moving => from root - right to left

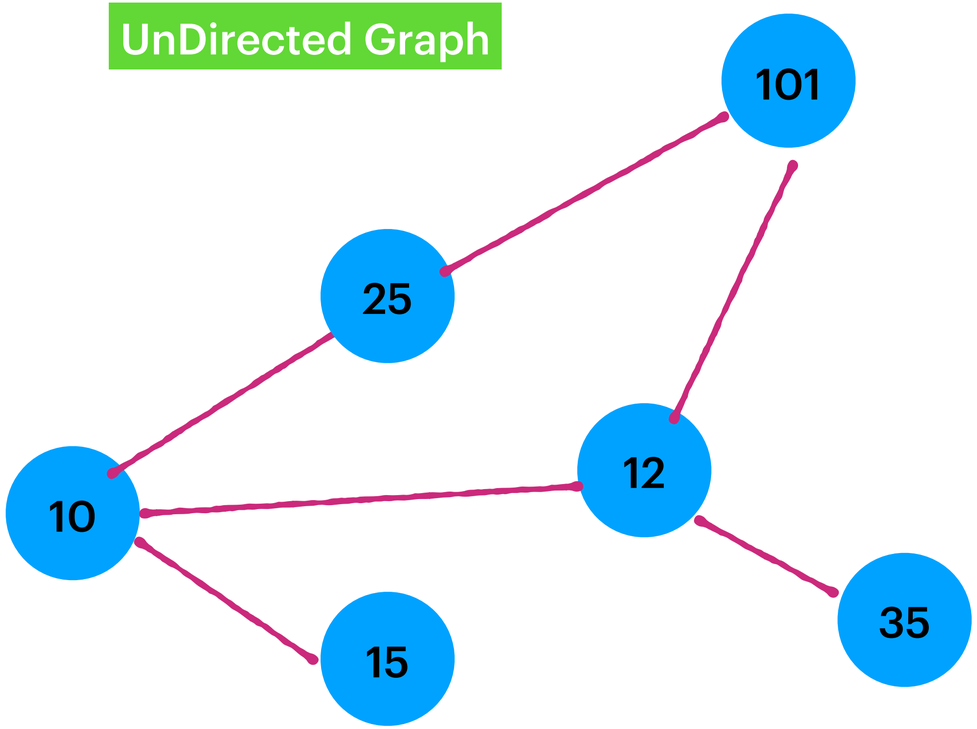


10, 12,100,11,101,15,25


Depth First Search
Directed Graph
101
25
10
12
35
15
visitedSet : [10,25,101,12,35,15]
Output: List<Integer> : [10,25,101,12,35,15]
10,25,101,12,35,15
Stack
25
12
15
Pop the top
101 is already visited
101

Depth First Search

UnDirected Graph



10 => [15, 12, 25] 15 => [10]
12 => [10, 101, 35]
25 => [10, 101]
101 => [25, 12]
35 => [12]

Stack []

Set [10,25,101,12,35,15]

OutputList[10,25,101,12,35,15]

DFS [10, 25, 101, 12, 35, 15]

Traverse All the Vertexes + Visit the unique Edges
Time Complexity : O(V+E)

V(10) => [15,12,25]
V(25) => [10,101]
V(101) => [25,12]
V(12) => [10,101,35]
V(35) => [12]
V(15) => [10]

start : 10
stack(v(10)) : 1

v(10) => stack ([15,12,25]) = 3 : visited : [v(10)]
v(25) => stack ([15,12,10,101]) = 2: visited : [v(10),v(25)]
v(101) => stack ([15,12,10,25,12]) = 2 visited : [v(10),v(25), v(101)]
v(12) => stack ([15,12,10,25,10,101,35]) = 3
visited : [v(10),v(25), v(101),v(12)]
V(35) => stack ([15,12,10,25,10,101,12]) = 1
visited : [v(10),v(25), v(101),v(12),v(35)]

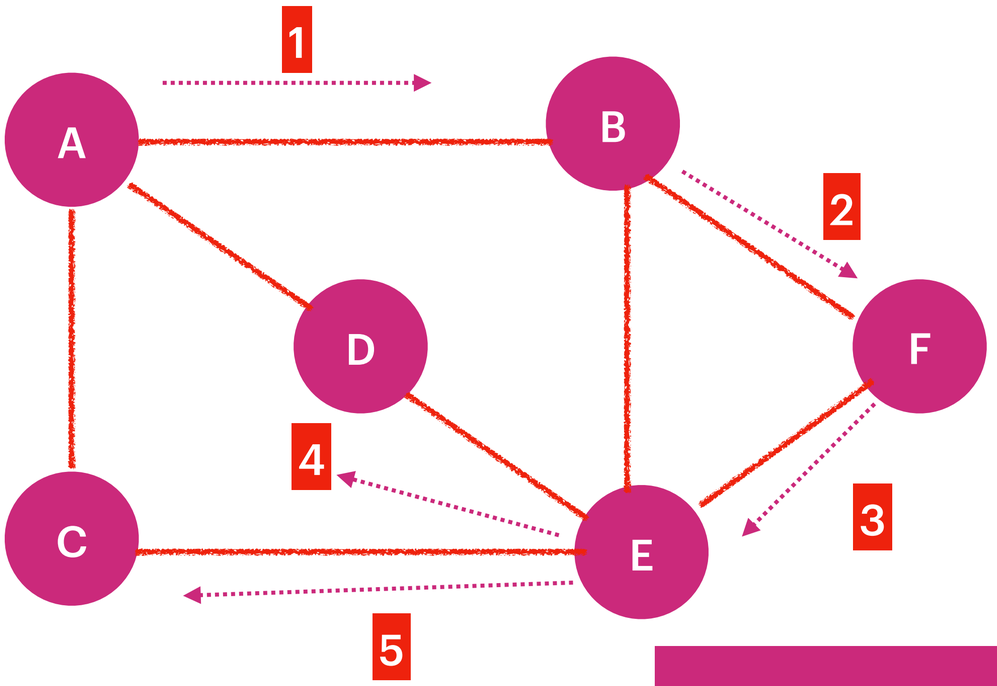
v(12) = 0 stack ([15,12,10,25,10,101]
v(101) = 0 stack ([15,12,10,25,10])
v(10) = 0 stack ([15,12,10,25])
v(25) = 0 stack ([15,12,10])
v(10) = 0 stack ([15,12])
v(12) = 0 stack ([15])
v(15) = 1 stack ([10]) visited : [v(10),v(25), v(101),v(12),v(35),v(15)]
v(10) = 0stac([]) visited : [v(10),v(25), v(101),v(12),v(35),v(15)]

numberOf.Connections(V) + Unique[Edges]

13V + 12(E) = V + E

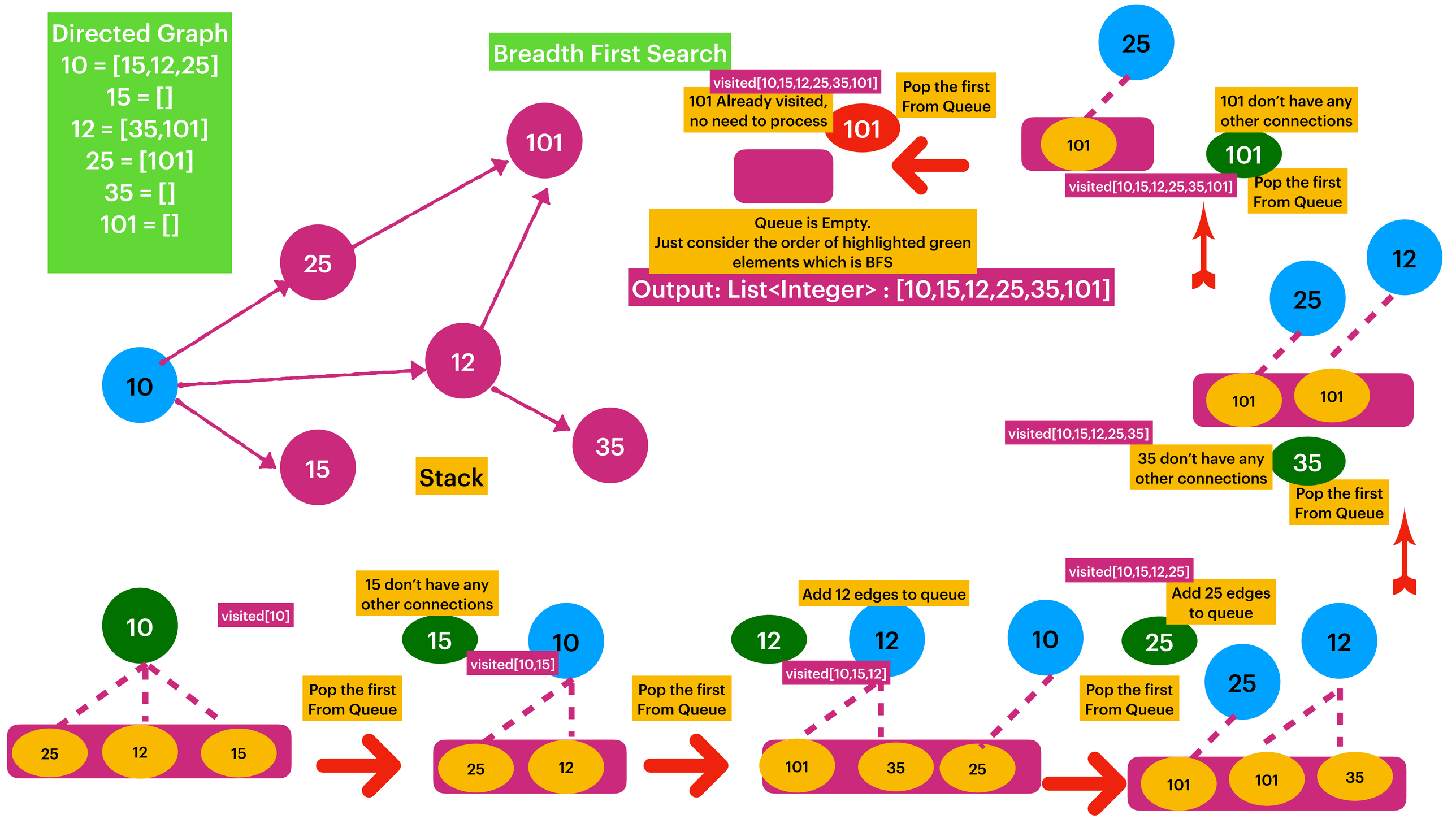
Analysis's Of DFS Time Complexity : O(V+E)

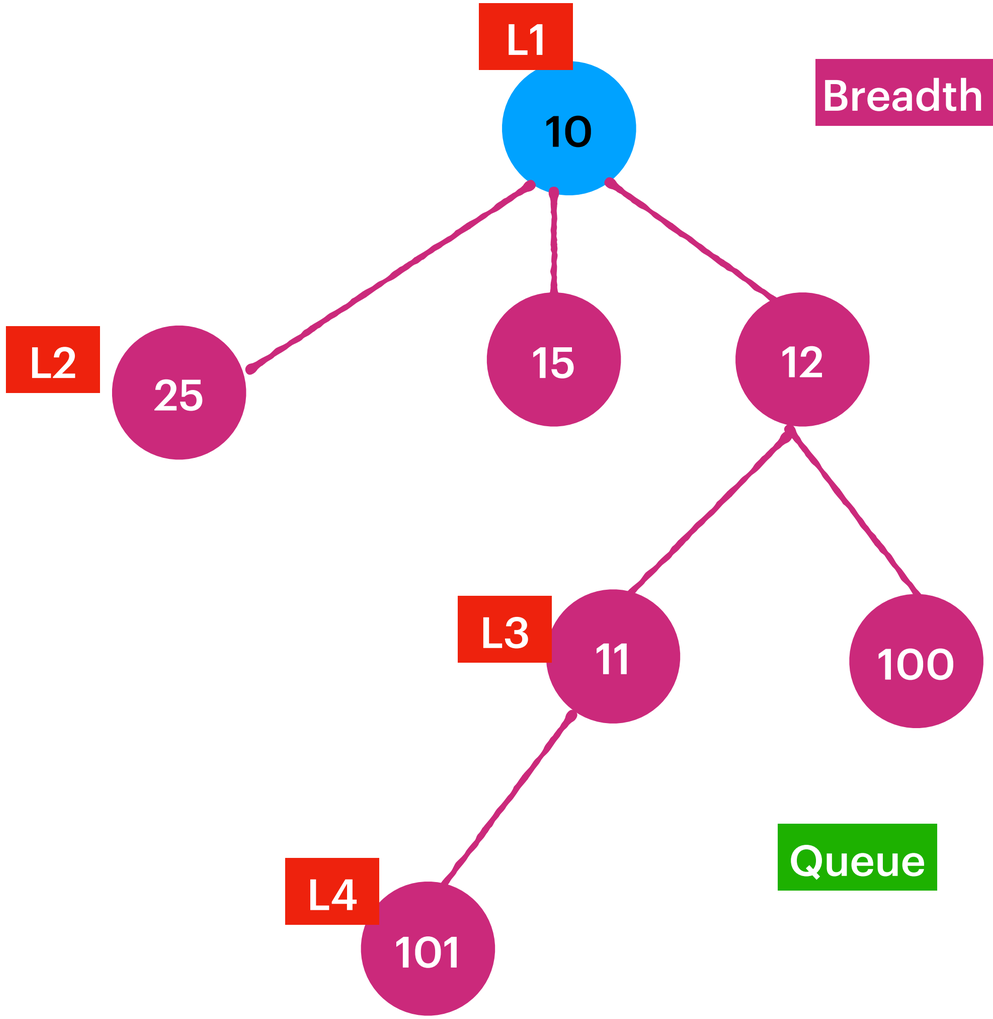
Depth First Search



A = [C,D,B]
B = [A,E,F]
C = [A,E]
D = [A,E]
E = [C,D,B,F]
F =[E,B]

Stack []
Visited [A,B,F,E,D,C]
DFS [A,B,F,E,D,C]


Directed Graph
10 = [15,12,25]
15 = []
12 = [35,101]
25 = [101]
35 = []
101 = []
Breadth First Search
visited[10,15,12,25,35,101]
101 Already visited, no need to process
101
Pop the first From Queue
Queue is Empty.
Just consider the order of highlighted green elements which is BFS
Output: List<Integer> : [10,15,12,25,35,101]
25
101
101 don't have any other connections
101
Pop the first From Queue
visited[10,15,12,25,35,101]
12
25
101
101
visited[10,15,12,25,35]
35 don't have any other connections
35
Pop the first From Queue
10
15
25
12
35
Stack
visited[10]
10
25
12
15
15 don't have any other connections
visited[10,15]
15
10
Pop the first From Queue
25
12
Add 12 edges to queue
visited[10,15,12]
12
12
Pop the first From Queue
101
35
25
visited[10,15,12,25]
Add 25 edges to queue
10
25
Pop the first From Queue
25
12
101
101
35



Breadth First Search

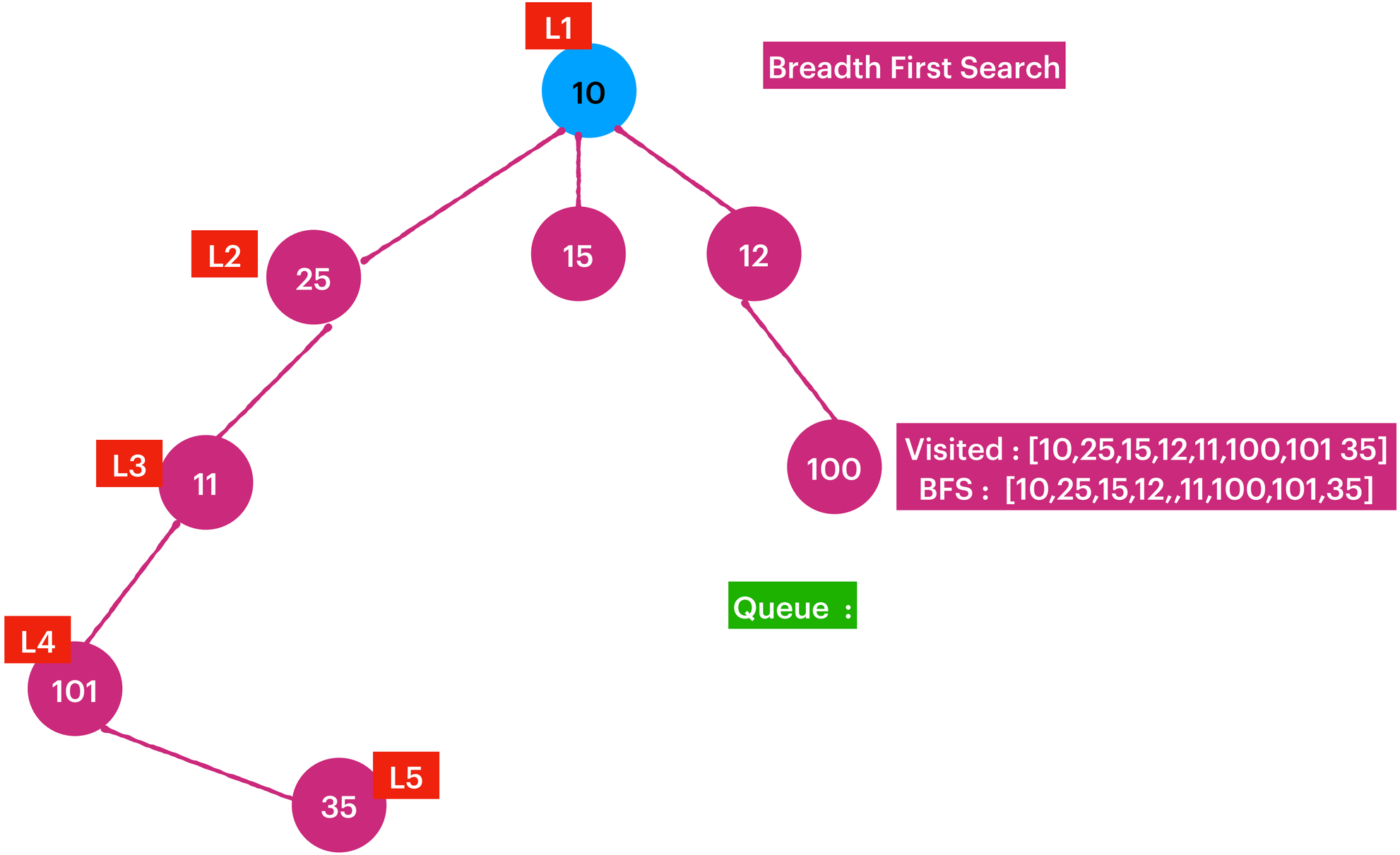
10 = [25,15,12]
25 = [10]
15 = [10]
12 = [10,11,100]
11 = [12,101]
100 = [12]
101 = [11]

Visited : [10,25,15,12,11,100,101]

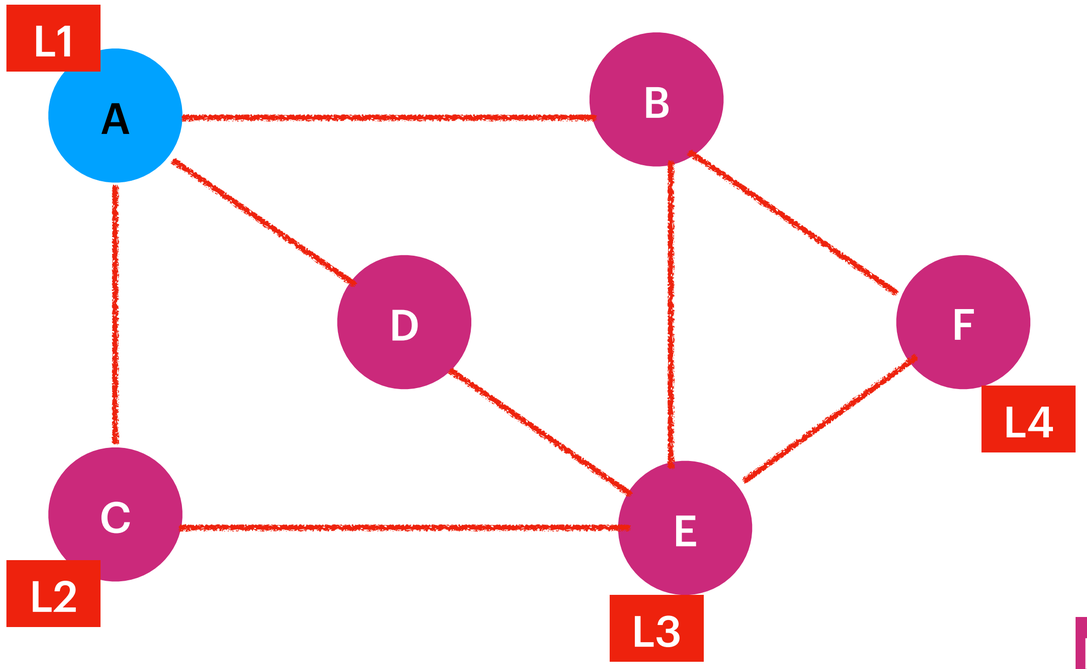
BFS : [10,25,15,12,11,100,101]

Queue

BFS [10, 25, 15, 12, 11, 100, 101]


L1
10
Breadth First Search
L2
25
15
12
L3
11
100
Visited : [10,25,15,12,11,100,101 35]
BFS : [10,25,15,12,,11,100,101,35]
Queue :
L4
101
L5
35

Breadth First Search



A = [C,D,B]
B = [A,E,F]
C = [A,E]
D = [A,E]
E = [C,D,B,F]
F =[E,B]

BFS : A,C,D,B,E,F

Find if Path Exists in Graph

There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex start to vertex end.

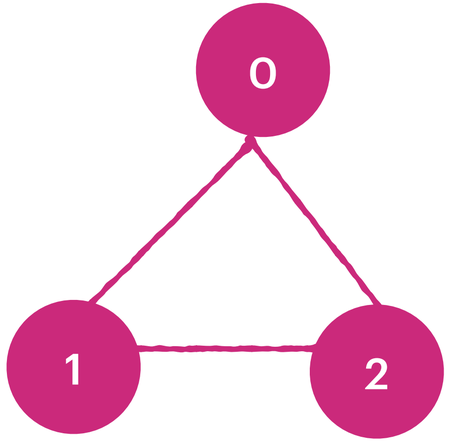
Given edges and the integers n, start, and end, return true if there is a valid path from start to end, or false otherwise.

Input: n = 3, edges = [[0,1],[1,2],[2,0]], start = 0, end = 2
Output: true
Explanation: There are two paths from vertex 0 to vertex 2:
- 0 → 1 → 2
- 0 → 2

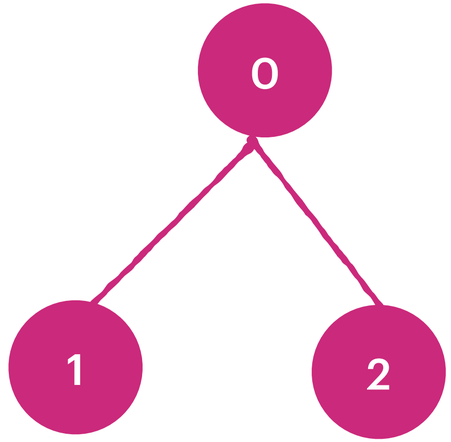
Input: n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], start = 0, end = 5
Output: false
Explanation: There is no path from vertex 0 to vertex 5.

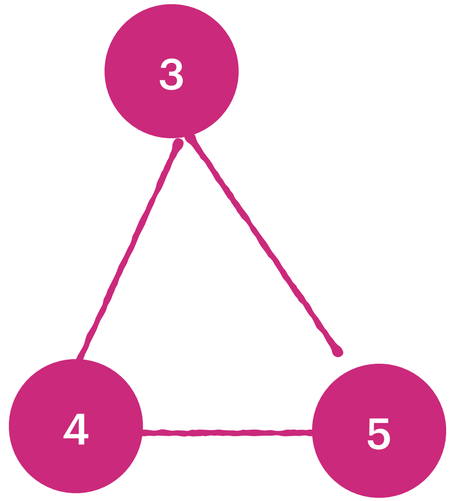
Constraints:
$1 <= n <= 2 * 10^5$
$0 <= edges.length <= 2 * 10^5$
$edges[i].length == 2$
$0 <= u_i, v_i <= n - 1$
$u_i != v_i$
$0 <= start, end <= n - 1$
There are no duplicate edges.
There are no self edges.

Input: n = 3, edges = [[0,1],[1,2],[2,0]], start = 0, end = 2
Output: true
Explanation: There are two paths from vertex 0 to vertex 2:
- 0 → 1 → 2
- 0 → 2



Solution 1 DisJoint Set : O(1) :
Solution 2. DFS : O(V+E)
Solution 3 BFS : O(V+E)





Input: n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], start = 0, end = 5
Output: false
Explanation: There is no path from vertex 0 to vertex 5.

All Paths From Source to Target

Given a directed acyclic graph (DAG) of n nodes labeled from 0 to n - 1,
find all possible paths from node 0 to node n - 1 and return them in any order.
The graph is given as follows: graph[i] is a list of all nodes you can visit from node i
(i.e., there is a directed edge from node i to node graph[i][j]).

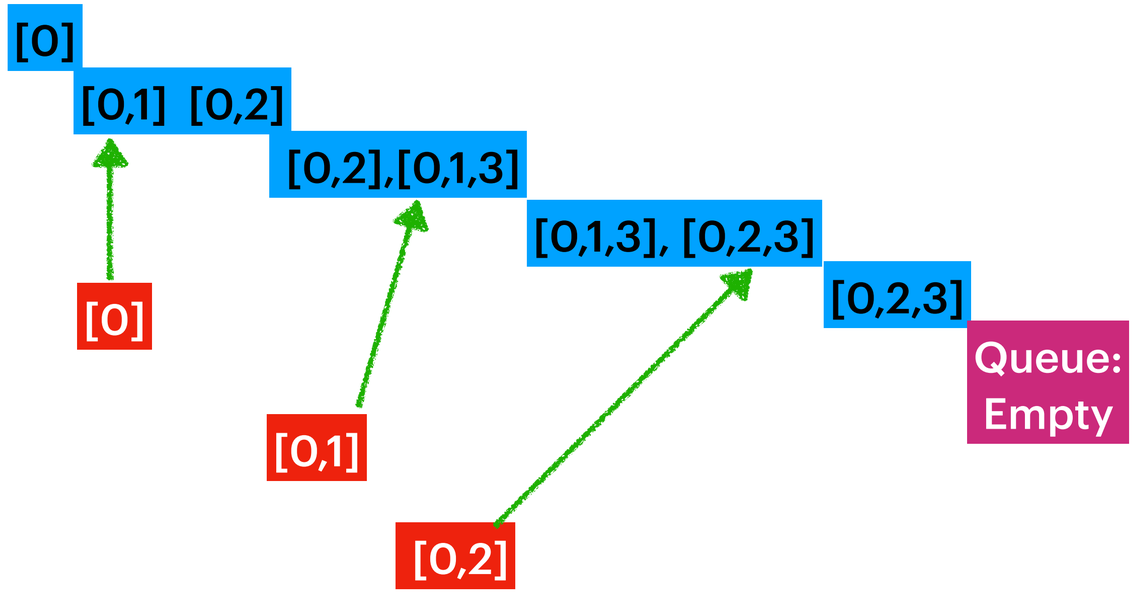
Input: graph = [[1,2],[3],[3],[]]
Output: [[0,1,3],[0,2,3]]
Explanation: There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.

Input: graph = [[4,3,1],[3,2,4],[3],[4],[]]
Output: [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]

n == graph.length
2 <= n <= 15
0 <= graph[i][j] < n
graph[i][j] != i (i.e., there will be no self-loops).
All the elements of graph[i] are unique.
The input graph is guaranteed to be a DAG.

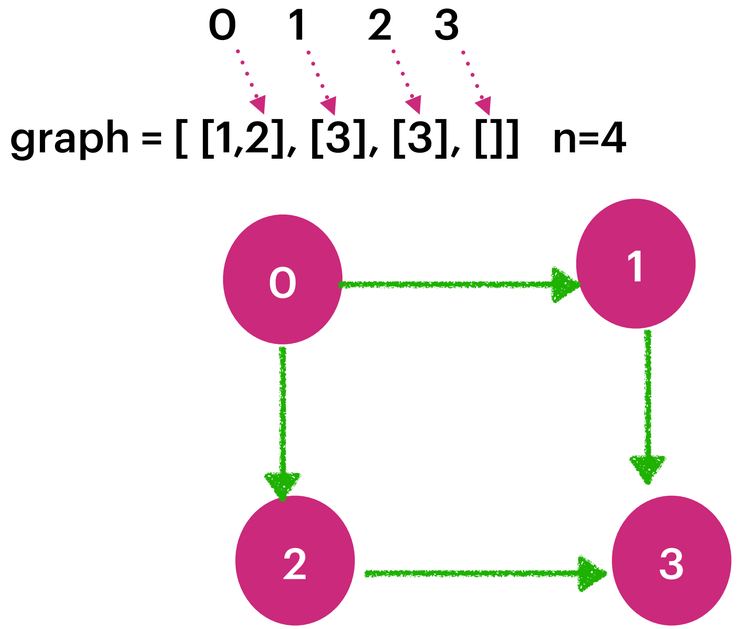
All Paths From Source to Target

Input: graph = [[1,2],[3],[3],[]] n=4
Output: [[0,1,3],[0,2,3]]
Explanation: There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.



[0,1,3]

Last Element is Equals to Target.



Paths From 0 to 3

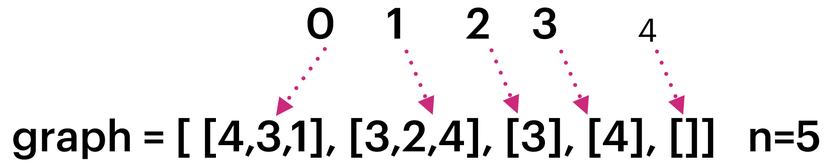
0 —> 1 —> 3 , 0—> 2 —> 3

[0,2,3]

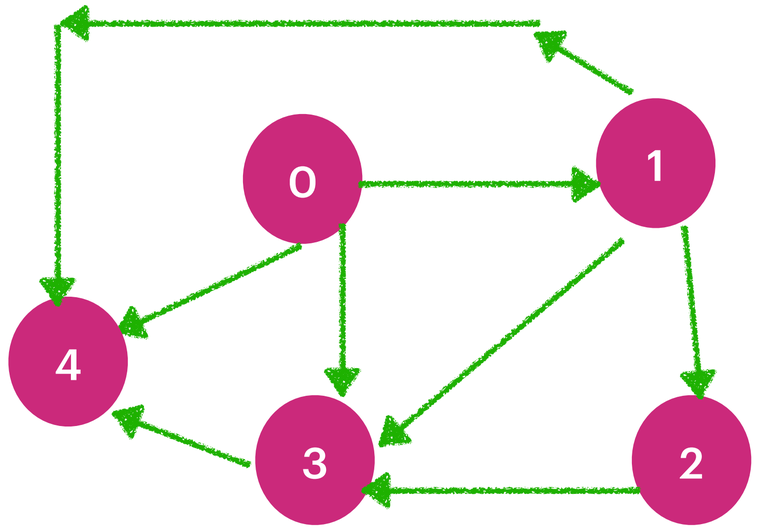
Last Element is Equals to Target.

All Paths From Source to Target

n == graph.length
2 <= n <= 15
0 <= graph[i][j] < n
graph[i][j] != i (i.e., there will be no self-loops).
All the elements of graph[i] are unique.
The input graph is guaranteed to be a DAG.



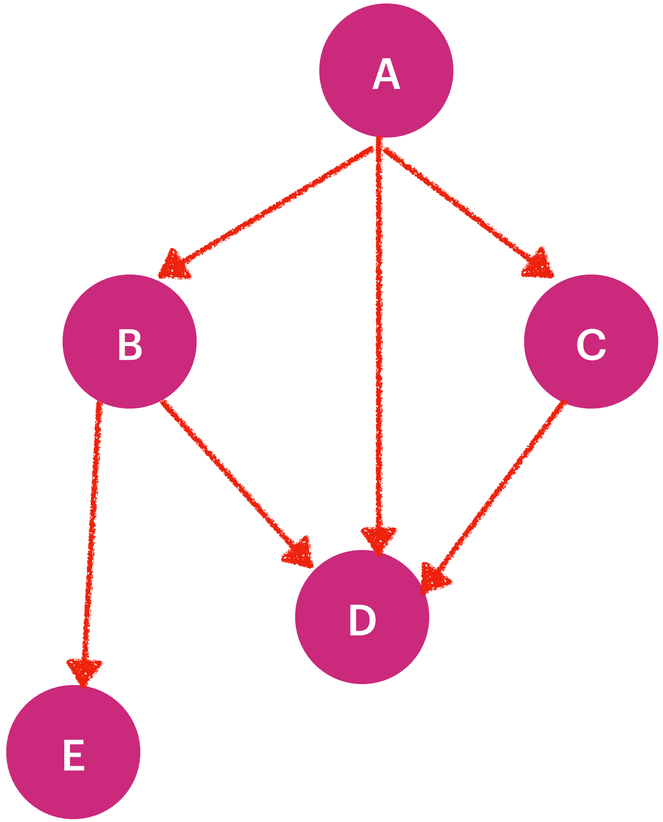
Input: graph = [[4,3,1],[3,2,4],[3],[4],[]] n=5
Output: [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]



Paths From 0 to 4

0->4 , 0->3->4, 0->1->3->4, 0->1->2->3->4, 0->1->4

BFS
Source : A
Target : D



[A]

[A,B]

[A,C]

[A,D]

Last Element is Equals to Target.

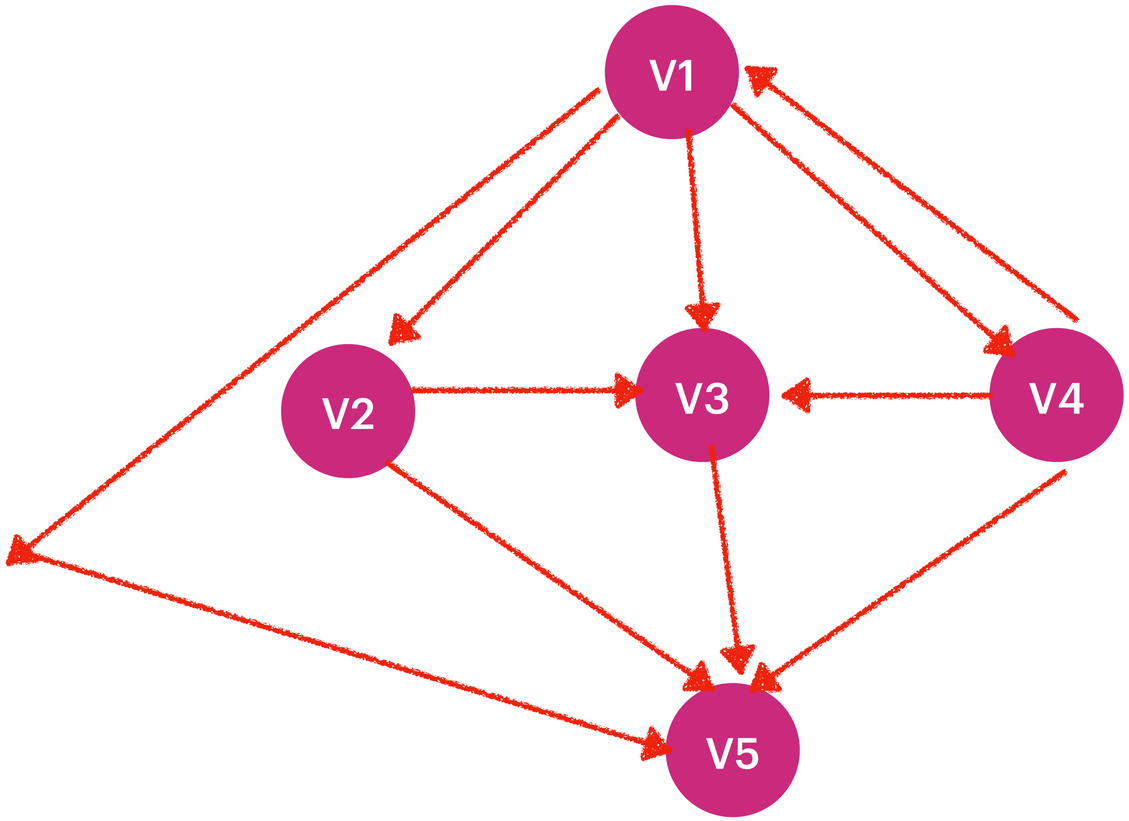
[A,B,D]

Last Element is Equals to Target.

[A,B,E]

[A,C,D]

Last Element is Equals to Target.



V * 2 ^ V

In worst case as per Graph Theory For each vertex we can make (2^V-1 - 1) unique paths. So in worst cast time complexity is O(V*2^V)

Source (V1) Target (V5)

V1 -—> V5 through V3 IFF V1——>V3 & V3——>V5

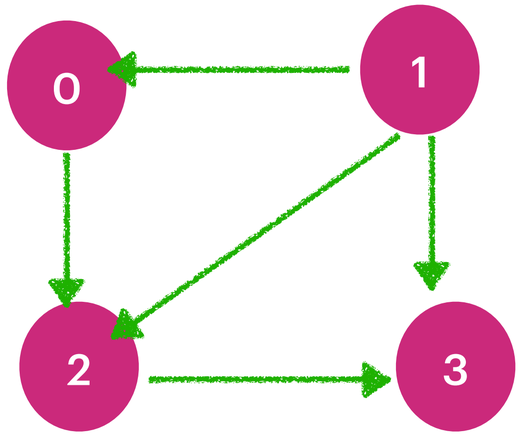
V1 -—> V5 through V2 IFF V1->V2 & V2->V5

All Paths From Source to Target : DFS

Input: graph = [[1,2],[2,3],[3],[]] n=4
Output: [[0,1,3],[0,2,3],[0,1,2,3]]
Explanation: There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.
0 -> 1 -> 2 ->3

0 1 2 3

graph = [[1,2], [2,3], [3], []] n=4



Empty

Pop [0,1,2,3]
Last element in the path is target

[0,1,2,3]

Pop [0,1,2] => Added edges of 2 to the path
[0,1,2,3] push to stack

[0,1,2]

Pop [0,1,3]
Last element in the path is target, take it

[0,1,3]
[0,1,2]

Pop [0,1] => Added edges of 1 to the path
[0,1,2] [0,1,3] push to stack

[0,1]

[0,2,3]
[0,1]

Pop [0,2,3]
Last element in the path is target, take it

[0,2]
[0,1]

Pop [0,2] => Added edges of 2 to the path
[0,2,3] push to stack

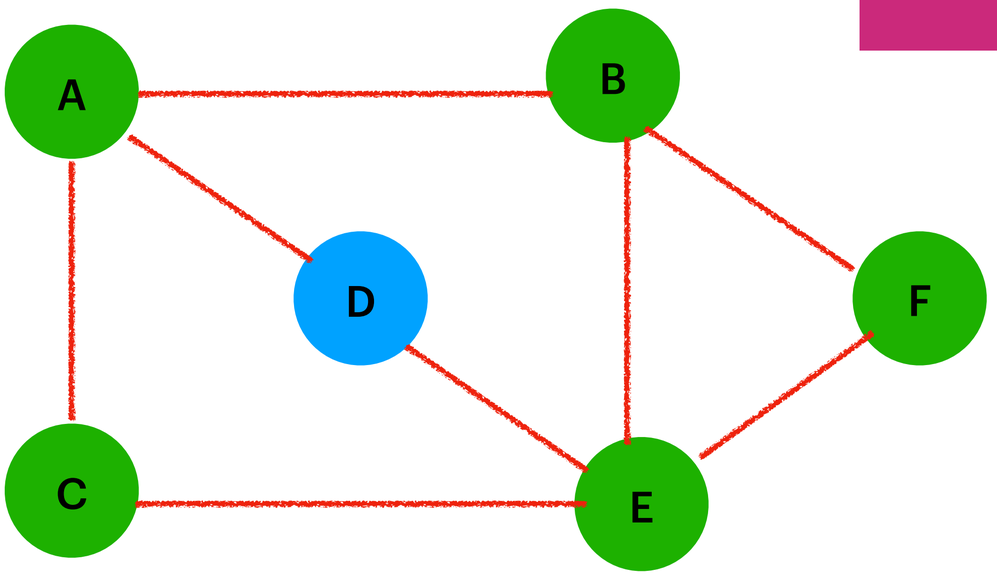
[0]

[0] -> 1,2

Paths From 0 to 3

0 —> 1 —> 3 , 0—> 2 —> 3 ,
0 —> 1 —> 2 —> 3

Print All the Paths From A -> F
DFS : (V-1) !
BFS : (V-1)!

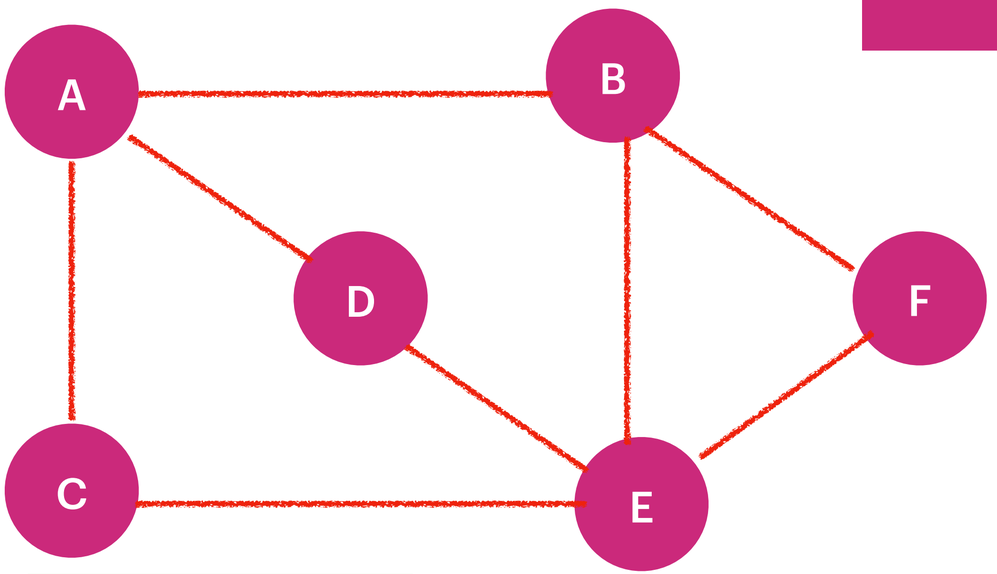


BFS [
[A, B, F],
[A, C, E, F],
[A, D, E, F],
[A, B, E, F],
[A, C, E, B, F],
[A, D, E, B, F]
]

output : [[A,B,F], [A,B,E,F],[A,D,E,F],[A,D,E,B,F],[A,C,E,F],[A,C,E,B,F]]

ABF,ACEBF,ADEF,ABEF,ACEBF,ADEBF

Shortest Path between A & F :
BFS : O(V+E)
Output : [A,B,F]



BFS A->F [A, B, F]
BFS B->D [B, A, D]
BFS E->A [E, B, A]
BFS C->B [C, A, B]
BFS E->F [E, F]

A [C,D,B]

Queue [AC,AD,AB] V[A]

AC :

Queue [AD,AB,ACA,ACE] V[A,C]

AD:
Queue [AB,ACA,ACE,ADA,ADE] : V[A,C,D]

AB: V[A,C,D,B,E]
Queue [ABF,ACE[C,B,F]]
ABF -> F is Target Shortest Path

Clone Graph

Given a reference of a node in a connected undirected graph.
Return a deep copy (clone) of the graph.
Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

```
class Node {
public int val;
public List<Node> neighbors;
}
```

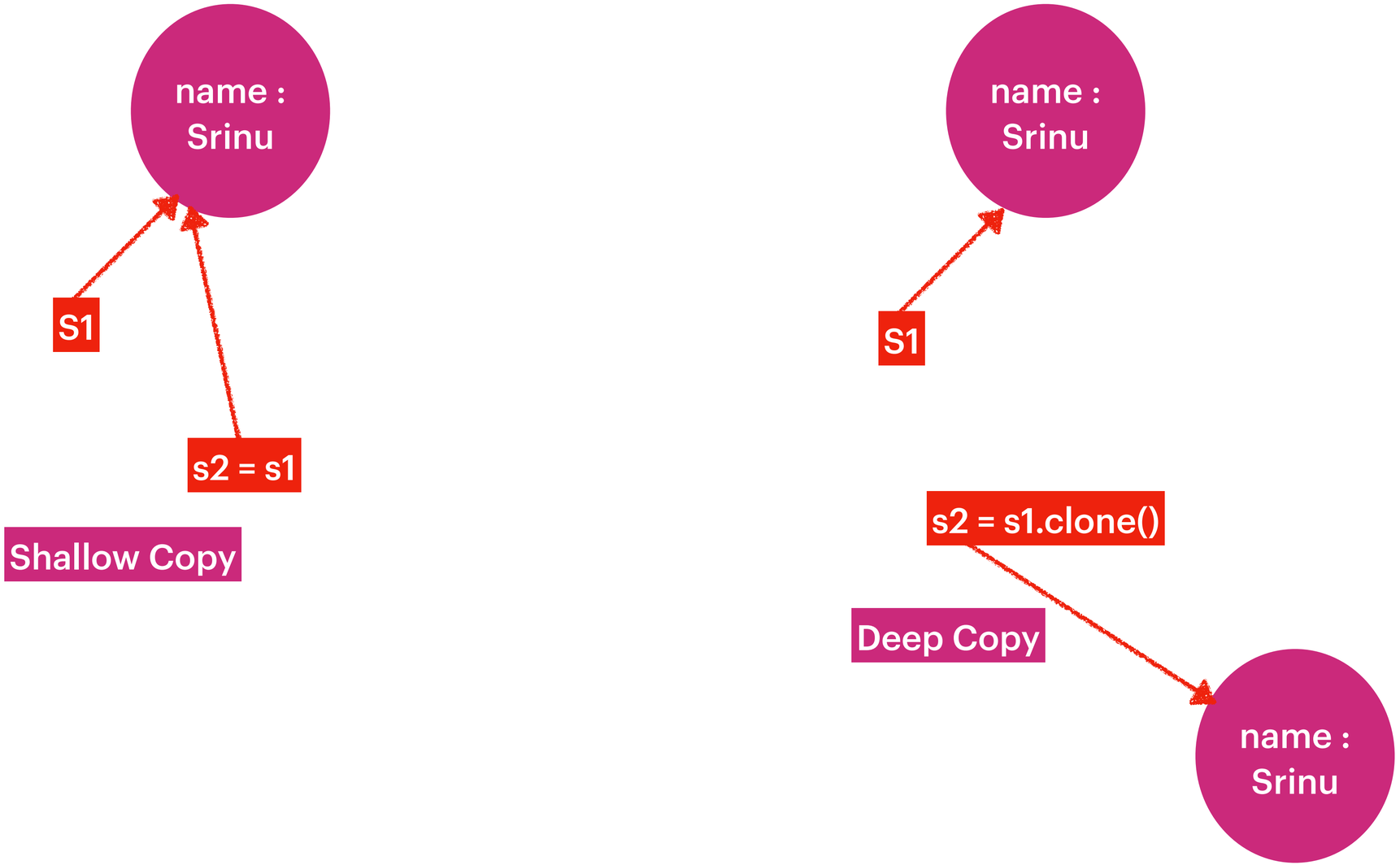
Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
Explanation: There are 4 nodes in the graph.
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

Constraints:
The number of nodes in the graph is in the range `[0, 100]`.
1 <= Node.val <= 100
`Node.val` is unique for each node.
There are no repeated edges
and no self-loops in the graph.
The Graph is connected and all nodes can
be visited starting from the given node.

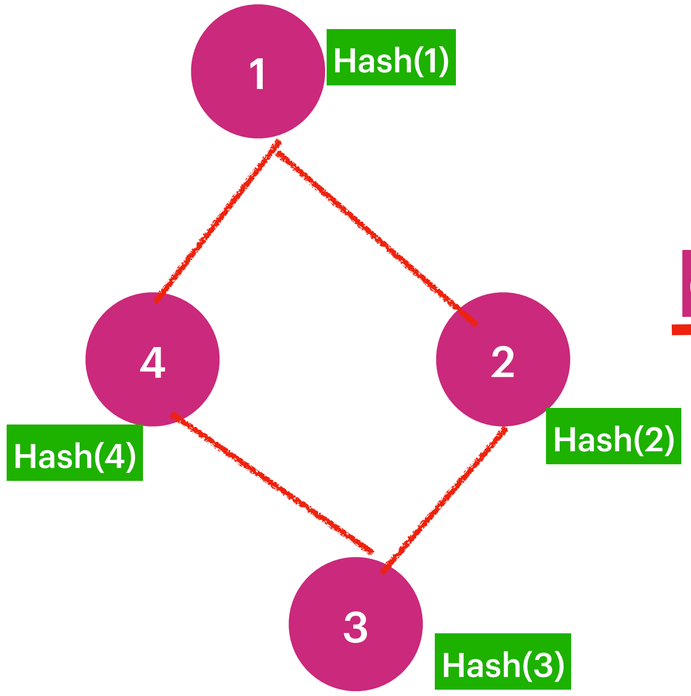
Input: adjList = [[2],[1]]
Output: [[2],[1]]

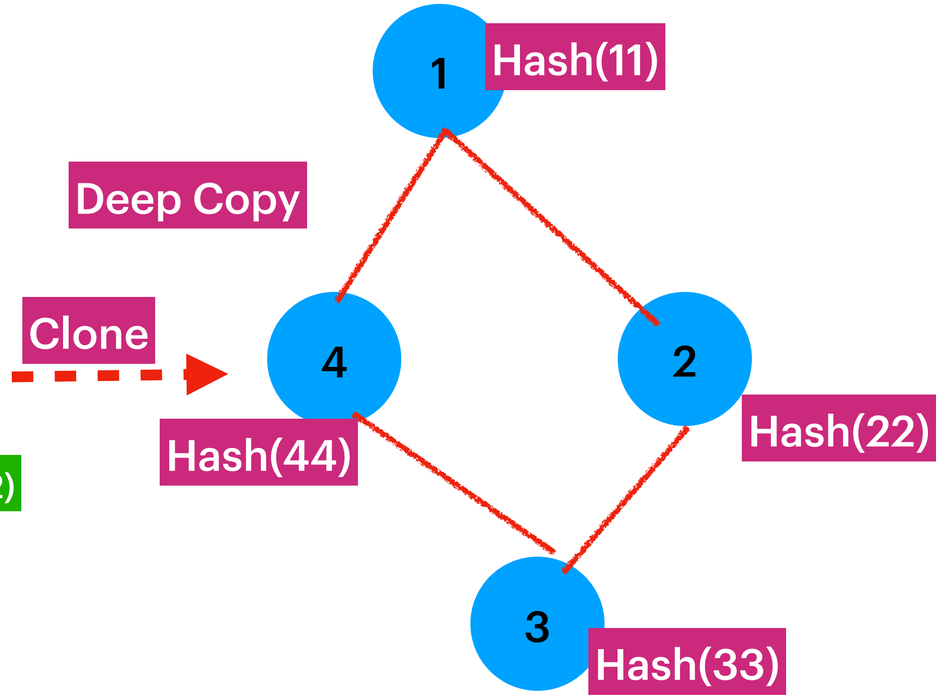
```
class cloneGraph {
public Node cloneGraph(Node node) {
}

}
```

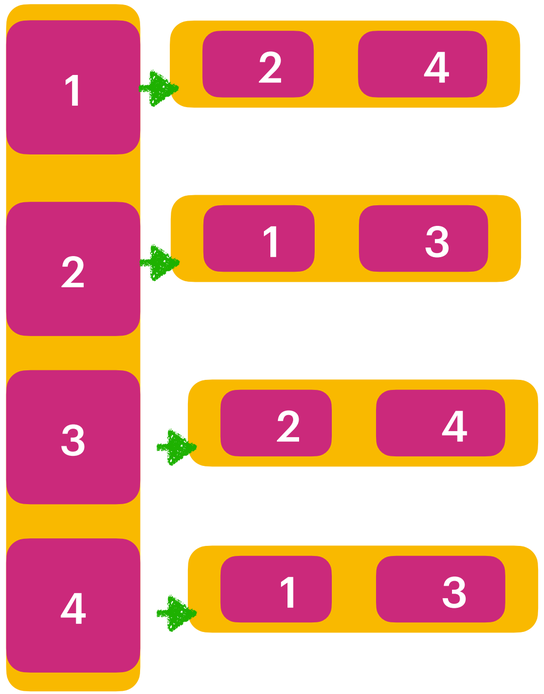
Input: adjList = [[]]
Output: [[]]
Explanation: Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

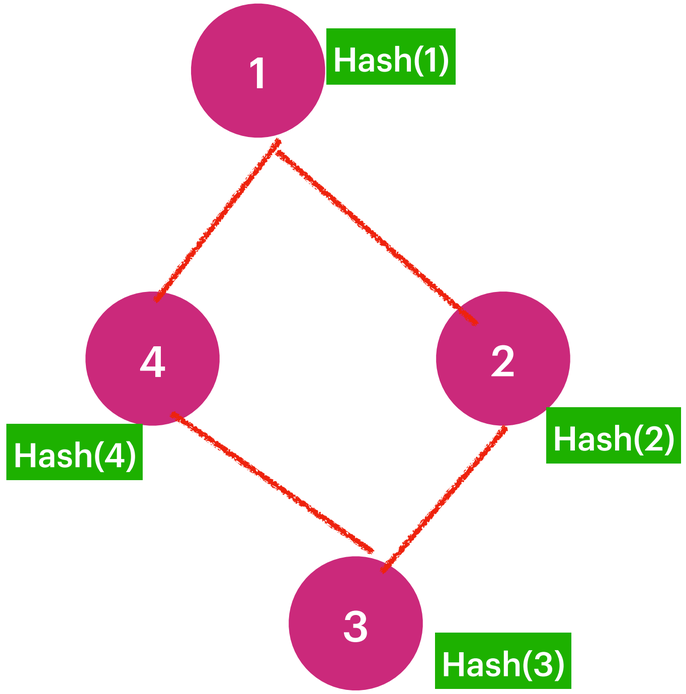

name :
Srinu
S1
s2 = s1
Shallow Copy
name :
Srinu
S1
s2 = s1.clone()
Deep Copy
name :
Srinu

Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
Explanation: There are 4 nodes in the graph.
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).





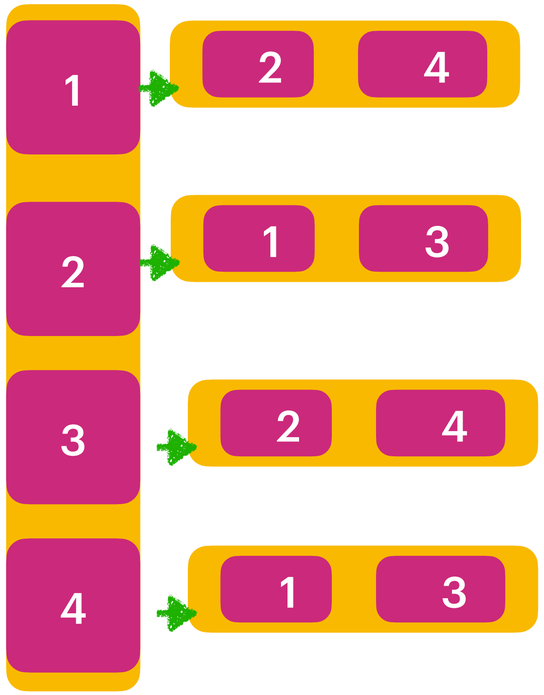




CloneNode(1)
4: 2

CloneNode(4) :
1copyRefOfCloneNode(1) :
3

CloneNode(2) :



Construct N-ary Tree & Level Order Traversal

Given an n-ary tree, return the *level order* traversal of its nodes' values.
Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).

nput: root = [1,null,3,2,4,null,5,6]
Output: [[1],[3,2,4],[5,6]]

Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
Output: [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

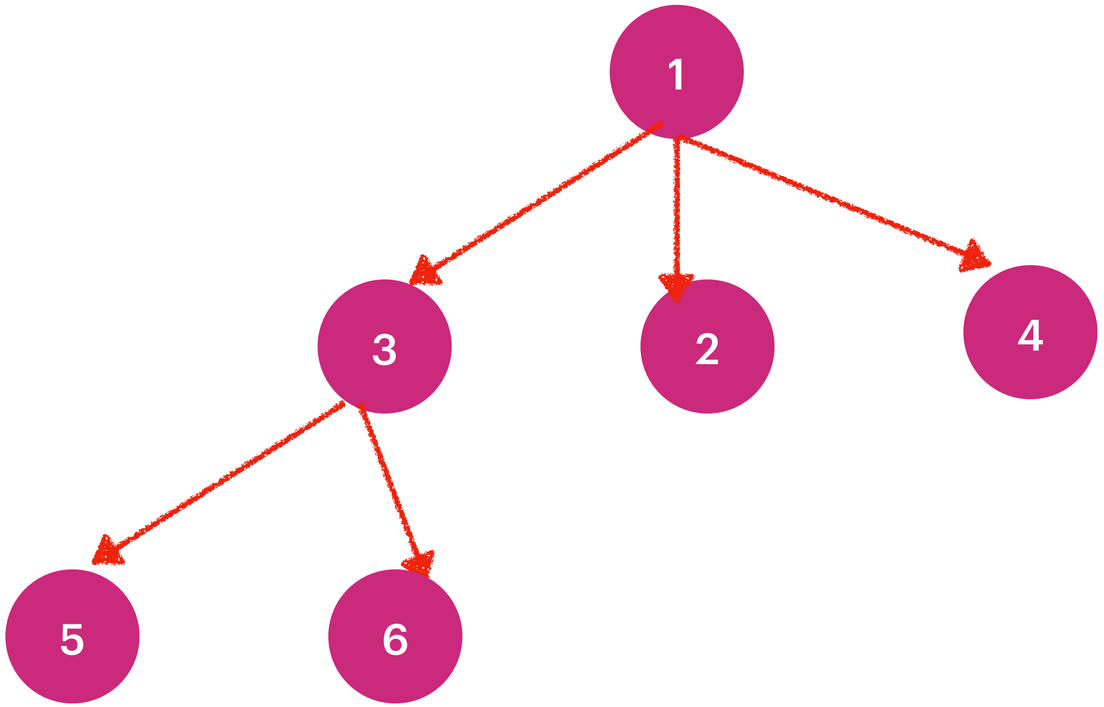
```
class Node {
public int val;
public List<Node> children;
}
```

Constraints:
The height of the n-ary tree is less than or equal to `1000`
The total number of nodes is between `[0, 10⁴]`

```
class Solution {
public List<List<Integer>> levelOrder(Node root) {
}
}
```

nput: root = [1,null,3,2,4,null,5,6]
Output: [[1],[3,2,4],[5,6]]

Construct N-ary Tree & Level Order Traversal



Output: [[1],[3,2,4],[5,6]]

Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
Output: [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

Queue[node[1])

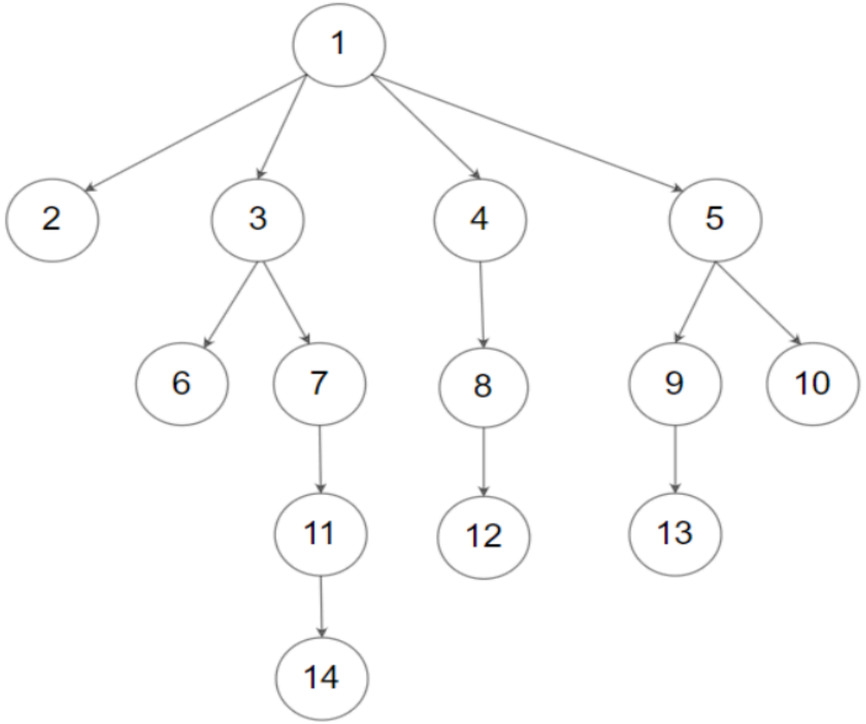
L1 [1]
Queue : [2,3,4,5]

L2[2,3,4,5]
Queue[6,7,8,9,10]

L3[6,7,8,9,10]
Queue[11,12,13]

L4 :[11,12,13]
Queue: [14]

L5[14]
Queue:[]



Construct N-ary Tree & Level Order Traversal

Output: [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

Walls and Gates

You are given an m x n grid rooms initialized with these three possible values.
-1 A wall or an obstacle.
0 A gate.
INF Infinity means an empty room.
We use the value 2^31 - 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.
Fill each empty room with the distance to its nearest gate.
If it is impossible to reach a gate, it should be filled with INF.

Input: rooms = [
[2147483647,-1,0,2147483647],
[2147483647,2147483647,2147483647,-1],
[2147483647,-1,2147483647,-1],
[0,-1,2147483647,2147483647]
]
Output: [
[3,-1,0,1],
[2,2,1,-1],
[1,-1,2,-1],
[0,-1,3,4]
]

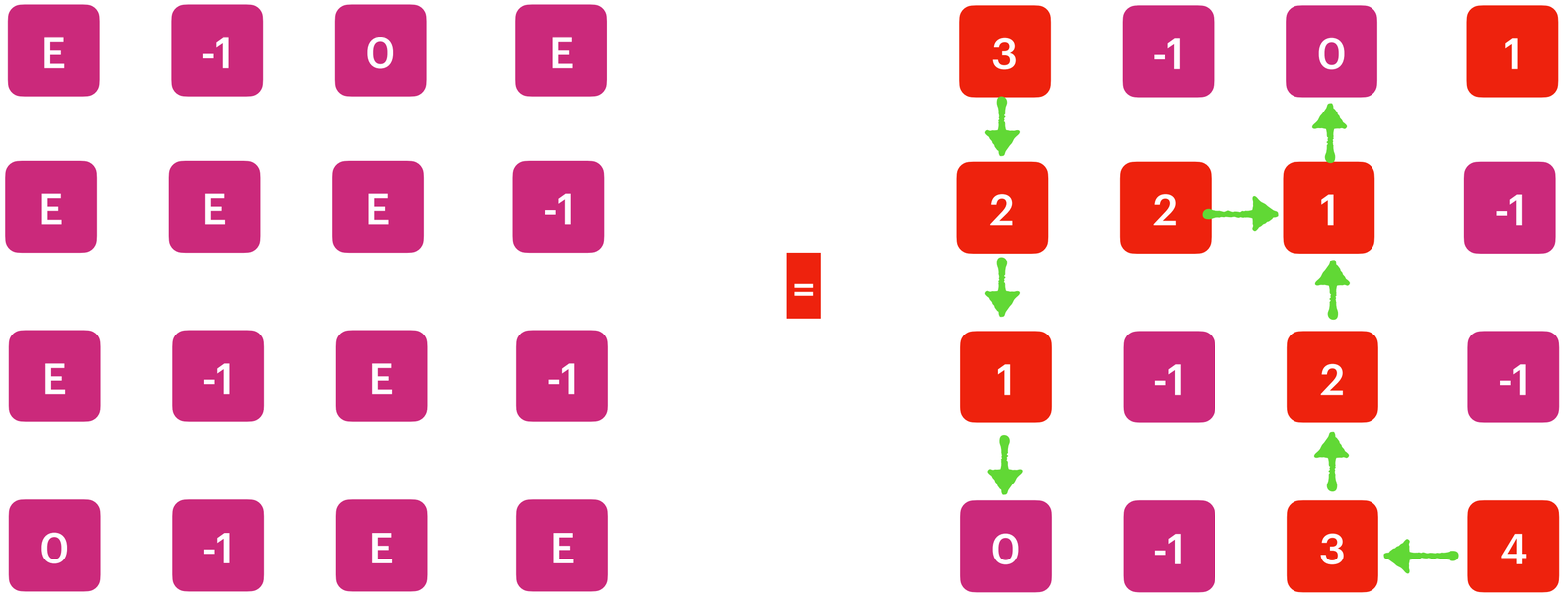
Input: rooms = [[-1]]
Output: [[-1]]

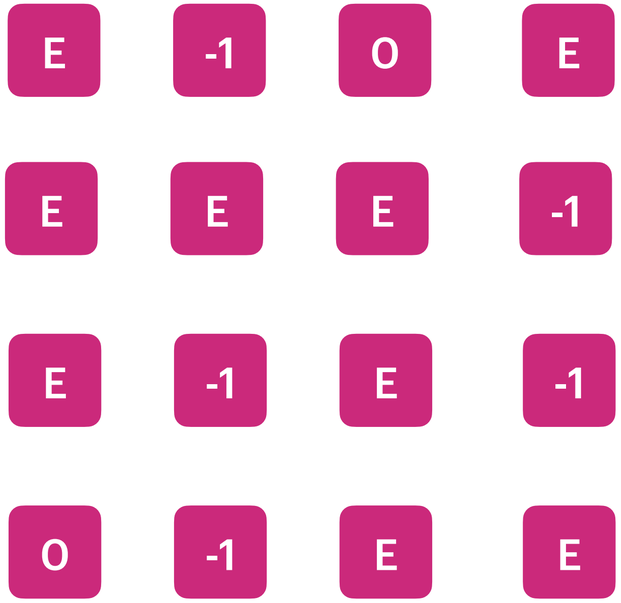
m == rooms.length
n == rooms[i].length
1 <= m, n <= 250
rooms[i][j] is -1, 0, or 231 - 1.

```
[
[E,-1,0,E],
[E,E,E,-1],
[E,-1,E,-1],
[0,-1,E,E]
]
```

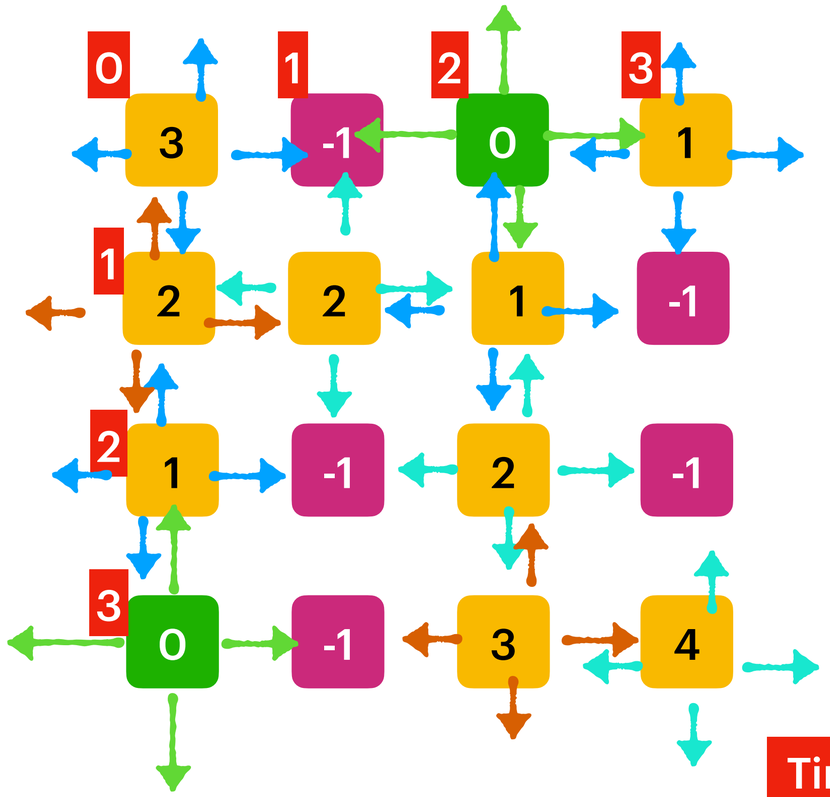
E = Integer.MAX_VALUE = Empty Room
-1 = Wall
0 = Gate

Fill the Empty Room with nearest possible distance





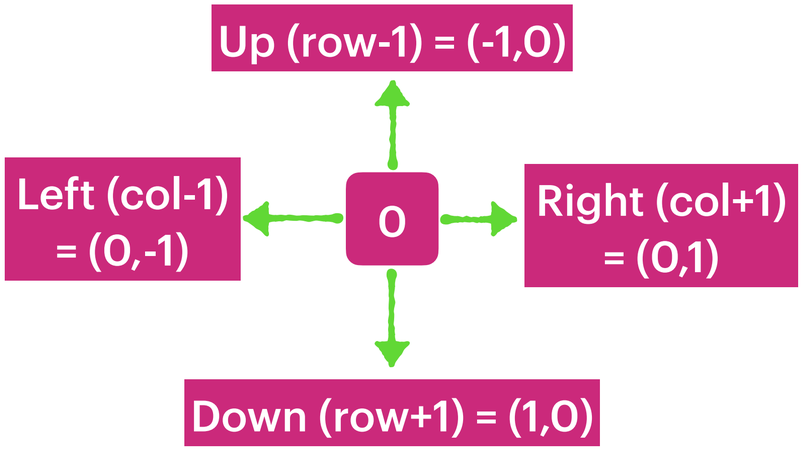
=



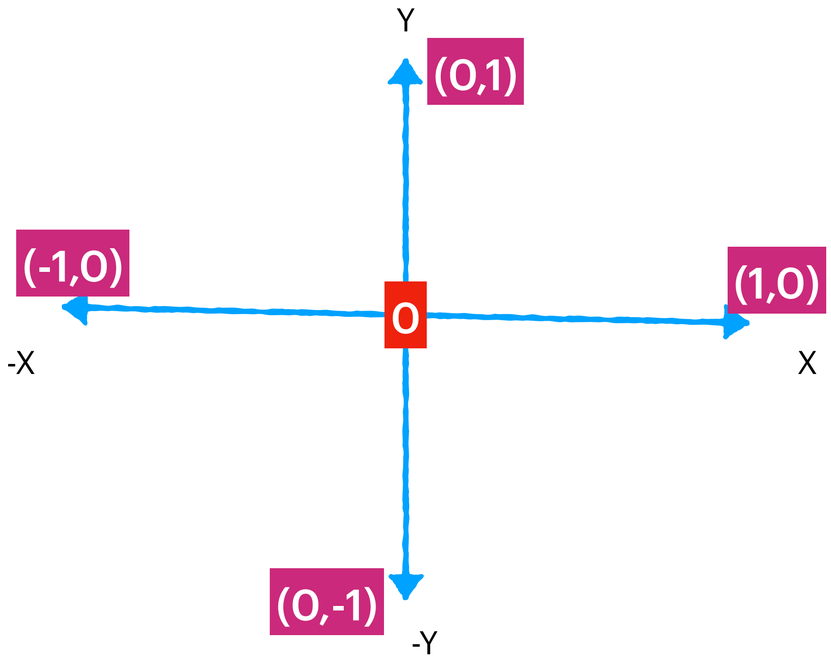

```
[
[3,-1,0,1],
[2,2,1,-1],
[1,-1,2,-1],
[0,-1,3,4]
]
```

Queue : []

Time Complexity : O(MN)
Space Complexity : O(MN)



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


Y
(0,1)
(-1,0)
0
(1,0)
-X
X
(0,-1)
-Y

01 Matrix

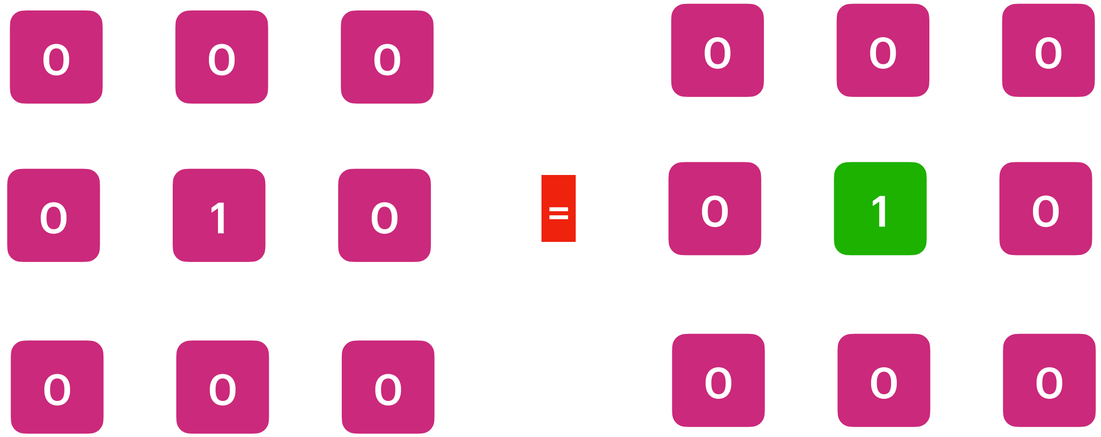
Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.
The distance between two adjacent cells is 1.

Input: mat = [
[0,0,0],
[0,1,0],
[0,0,0]
]
Output: [
[0,0,0],
[0,1,0],
[0,0,0]
]

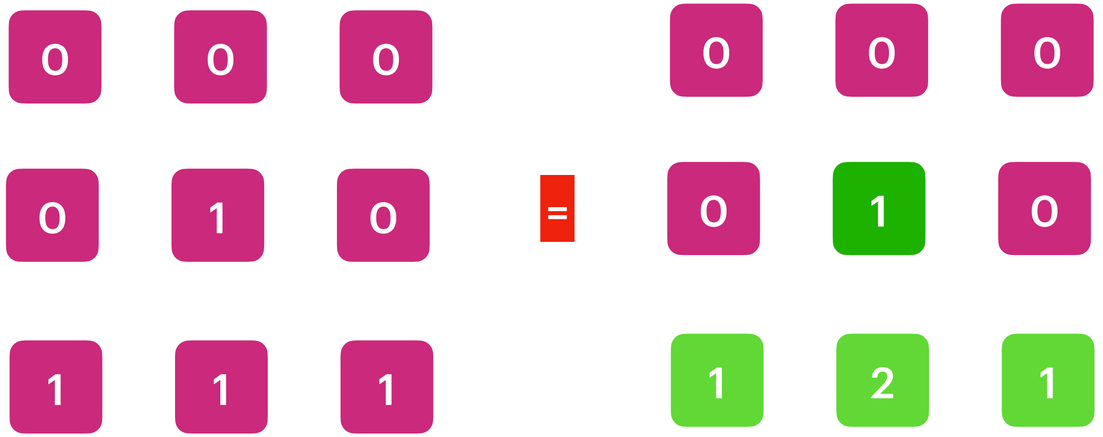
Input: mat = [
[0,0,0],
[0,1,0],
[1,1,1]
]
Output: [
[0,0,0],
[0,1,0],
[1,2,1]
]

Constraints:

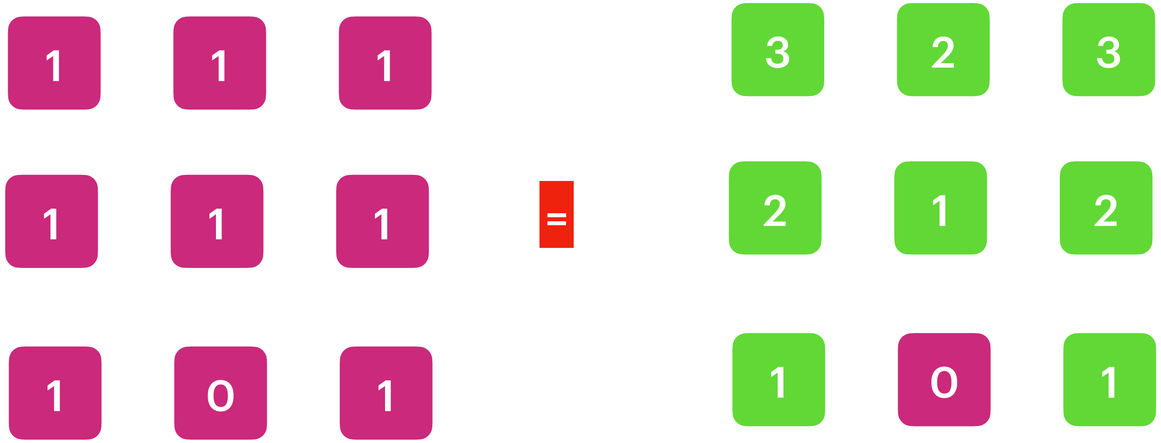
m == mat.length
n == mat[i].length
1 <= m, n <= 104
1 <= m * n <= 104
mat[i][j] is either 0 or 1.
There is at least one 0 in mat.

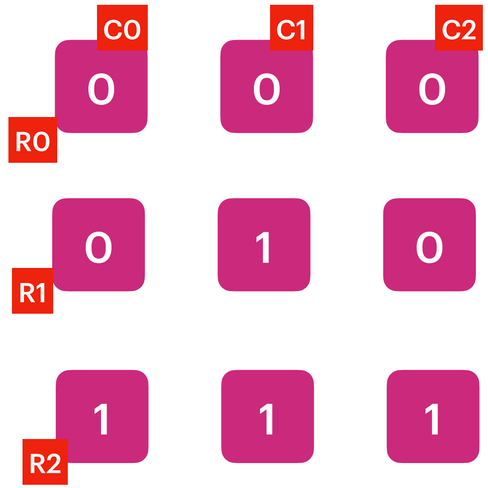


Input: mat = [
[0,0,0],
[0,1,0],
[0,0,0]
]
Output: [
[0,0,0],
[0,1,0],
[0,0,0]
]



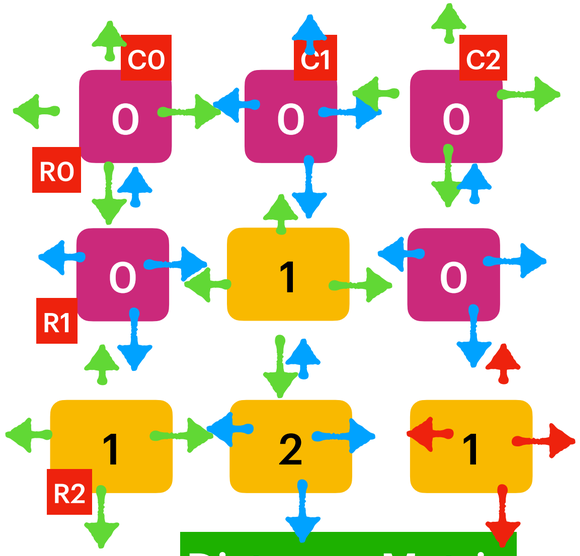
Input: mat = [
[0,0,0],
[0,1,0],
[1,1,1]
]
Output: [
[0,0,0],
[0,1,0],
[0,0,0]
]


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



Input-Matrix

=

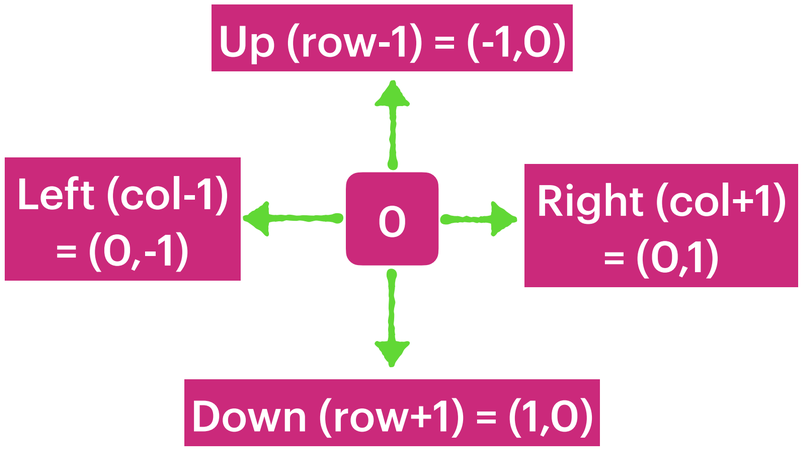


Distance-Matrix

dir_Cell > currentCell_Value + 1

Time Complexity : O(M*N)
Space Complexity : O(M*N)

Queue []



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

01 Matrix :: Level Order

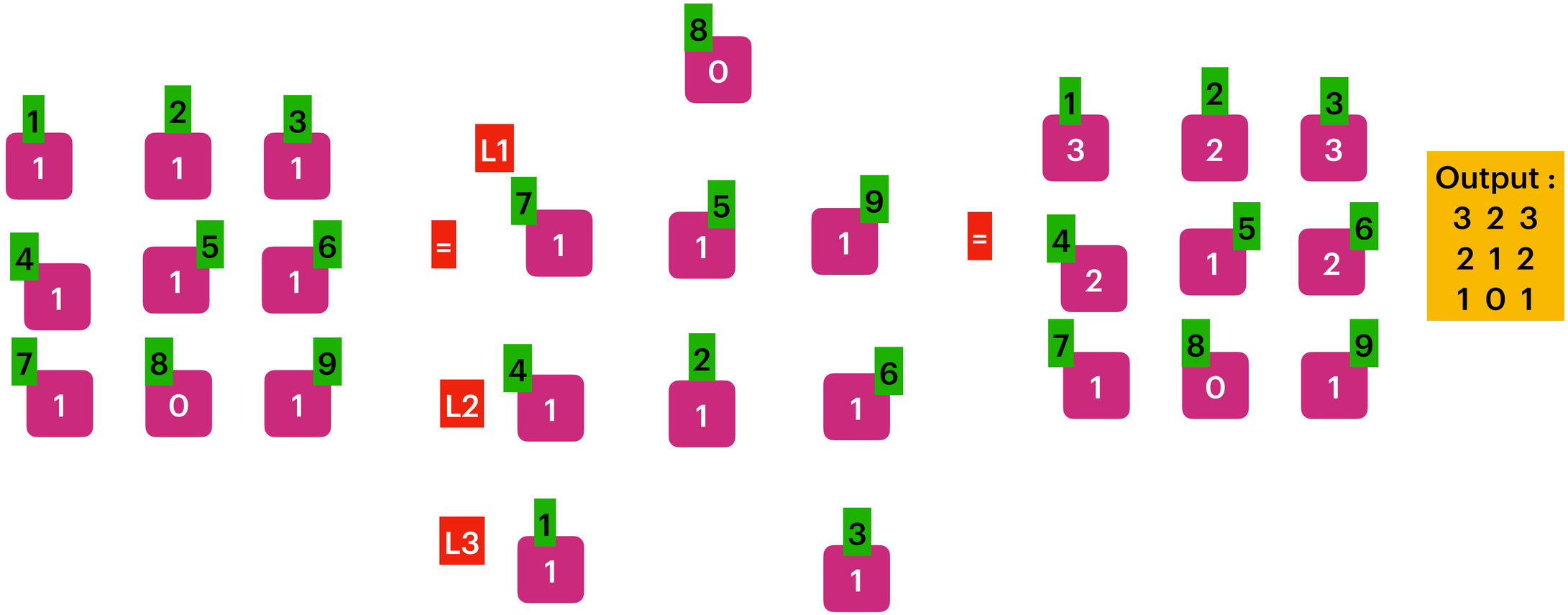
Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.
The distance between two adjacent cells is 1.

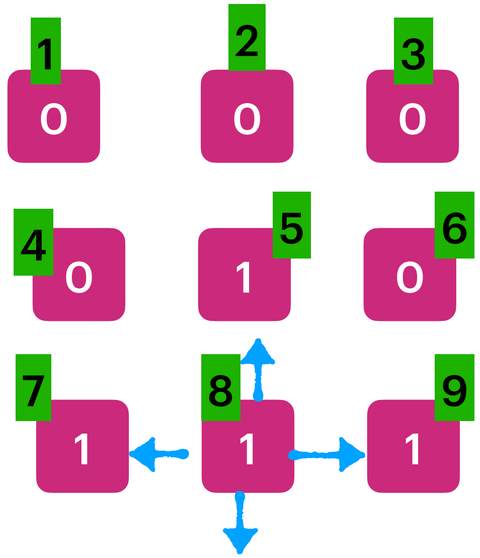
Input: mat = [
[0,0,0],
[0,1,0],
[0,0,0]
]
Output: [
[0,0,0],
[0,1,0],
[0,0,0]
]

Input: mat = [
[0,0,0],
[0,1,0],
[1,1,1]
]
Output: [
[0,0,0],
[0,1,0],
[1,2,1]
]

Constraints:

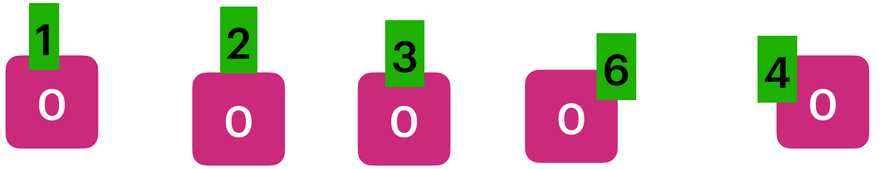
m == mat.length
n == mat[i].length
1 <= m, n <= 104
1 <= m * n <= 104
mat[i][j] is either 0 or 1.
There is at least one 0 in mat.


1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 0
9 1
8 0
L1
7 1
5 1
9 1
=
L2
4 1
2 1
6 1
L3
1 1
3 1
=
1 3
2 2
3 3
4 2
5 1
6 2
7 1
8 0
9 1
Output :
3 2 3
2 1 2
1 0 1


1
0
2
0
3
0
4
0
5
1
6
0
7
1
8
1
9
1

=

L0

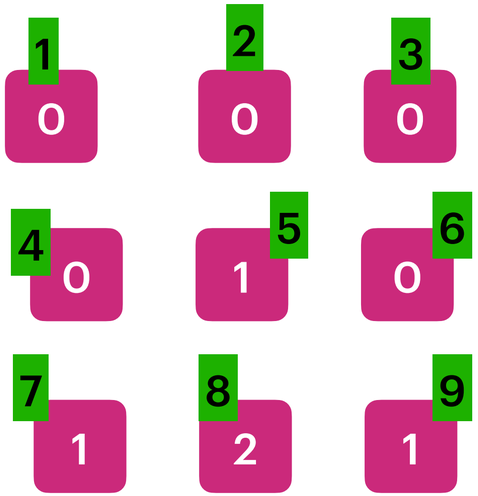

1
0
2
0
3
0
6
0
4
0

L1

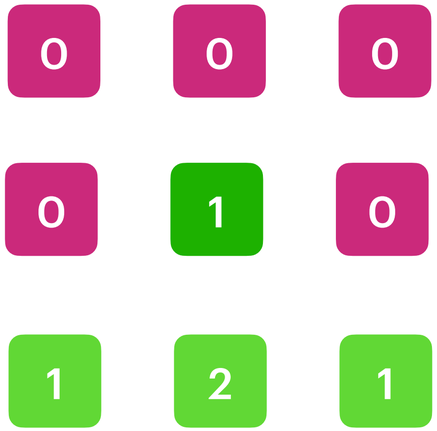

5
1
9
1
7
1

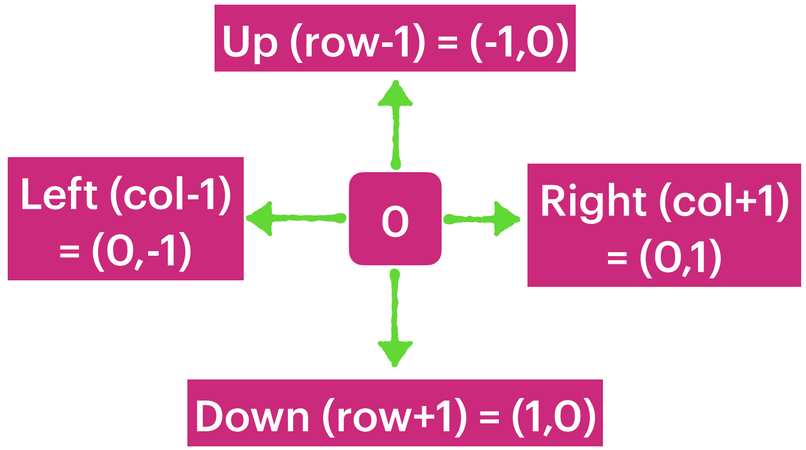
L2


8
2


1
0
2
0
3
0
4
0
5
1
6
0
7
1
8
2
9
1

=


0
0
0
0
1
0
1
2
1


Up (row-1) = (-1,0)
Left (col-1) = (0,-1)
0
Right (col+1) = (0,1)
Down (row+1) = (1,0)

Rotting Oranges

You are given an m x n grid where each cell can have one of three values:

0 representing an empty cell,
1 representing a fresh orange, or
2 representing a rotten orange.
Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange.
If this is impossible, return -1.

Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4

Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

Input: grid = [[1]]
Output: -1
Explanation: Since the orange can never rotate.

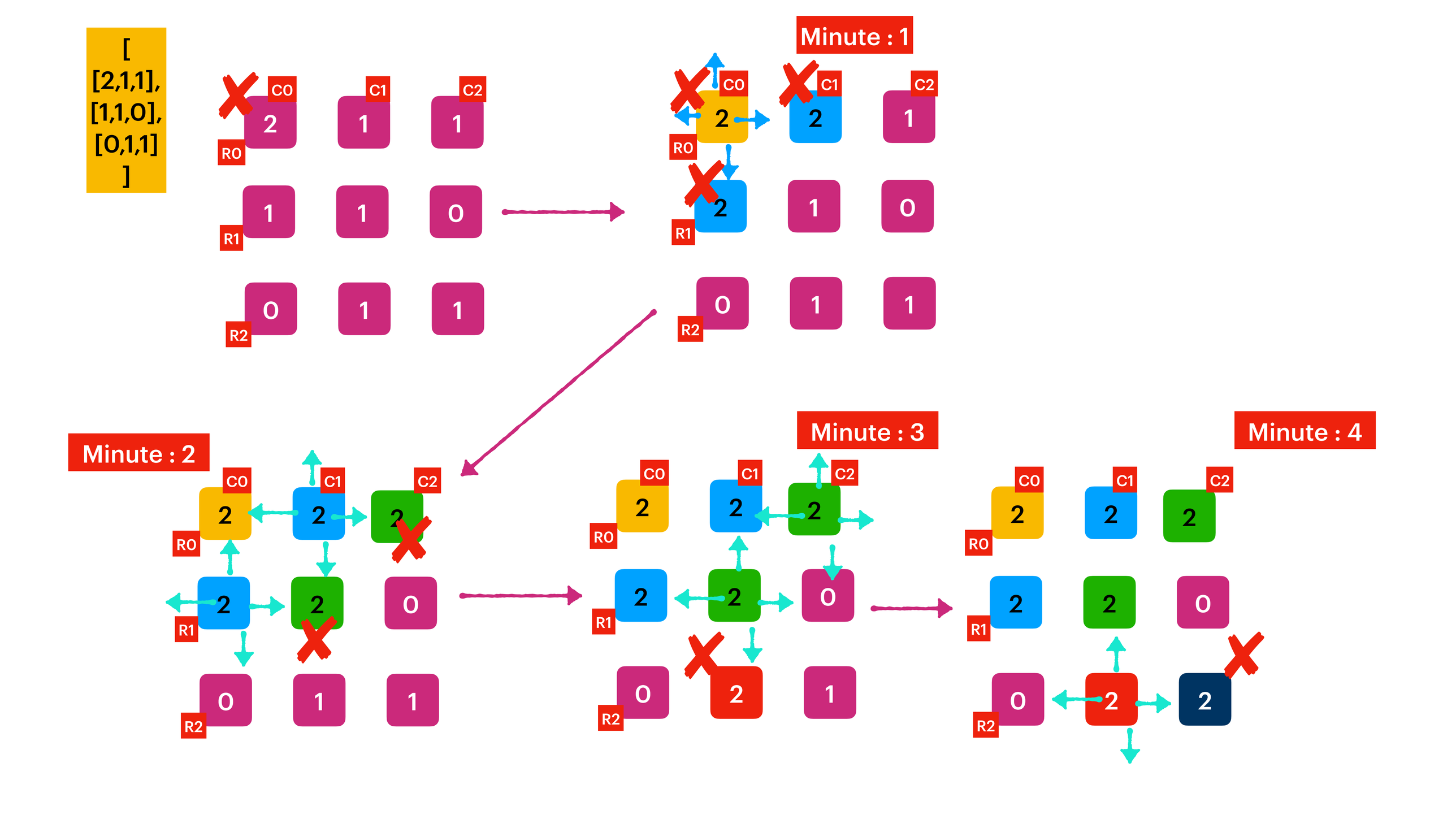
Input: grid = [[0,0,0,0,0]]
Output: 0
Explanation: Since there are no oranges to rotate.

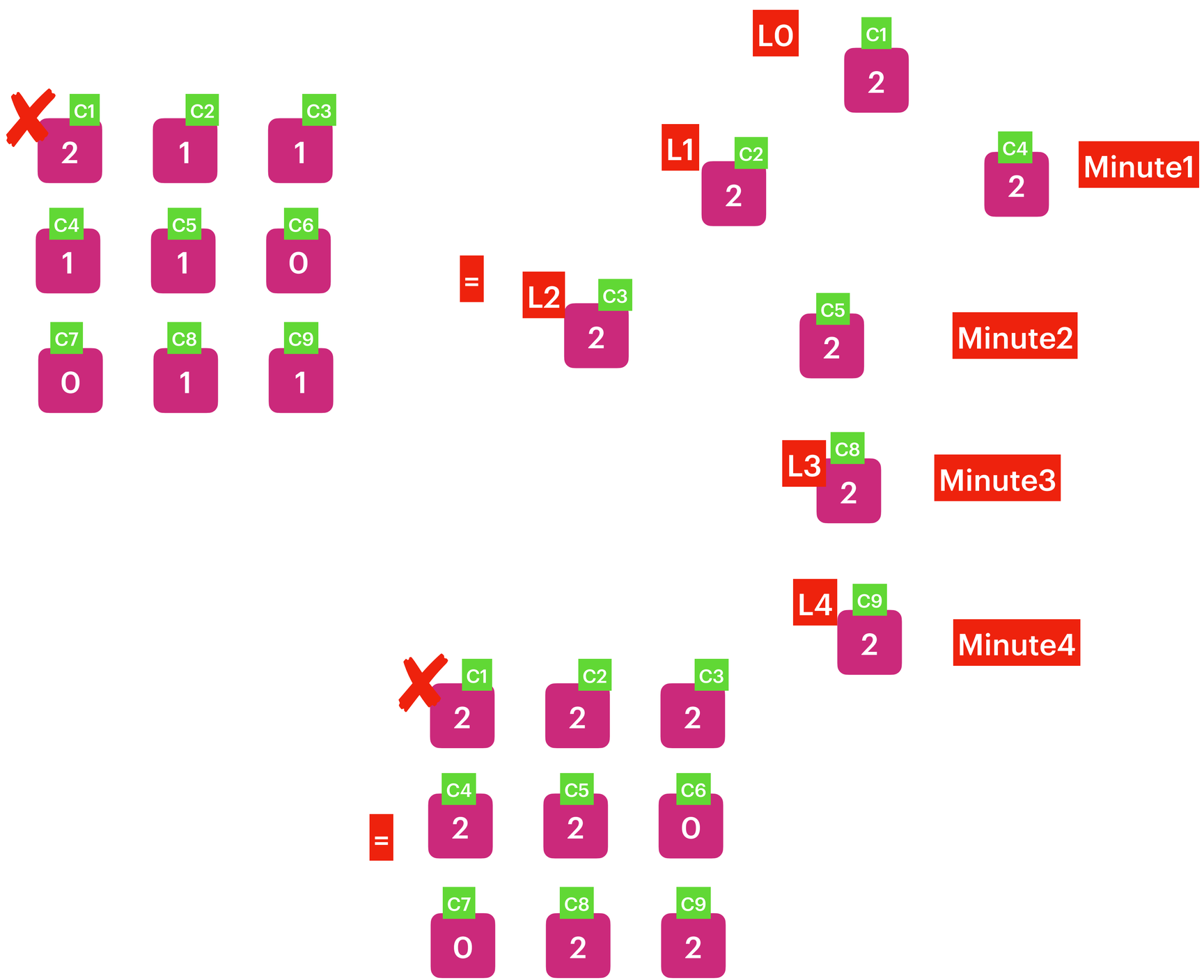
Input: grid = [[0]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.

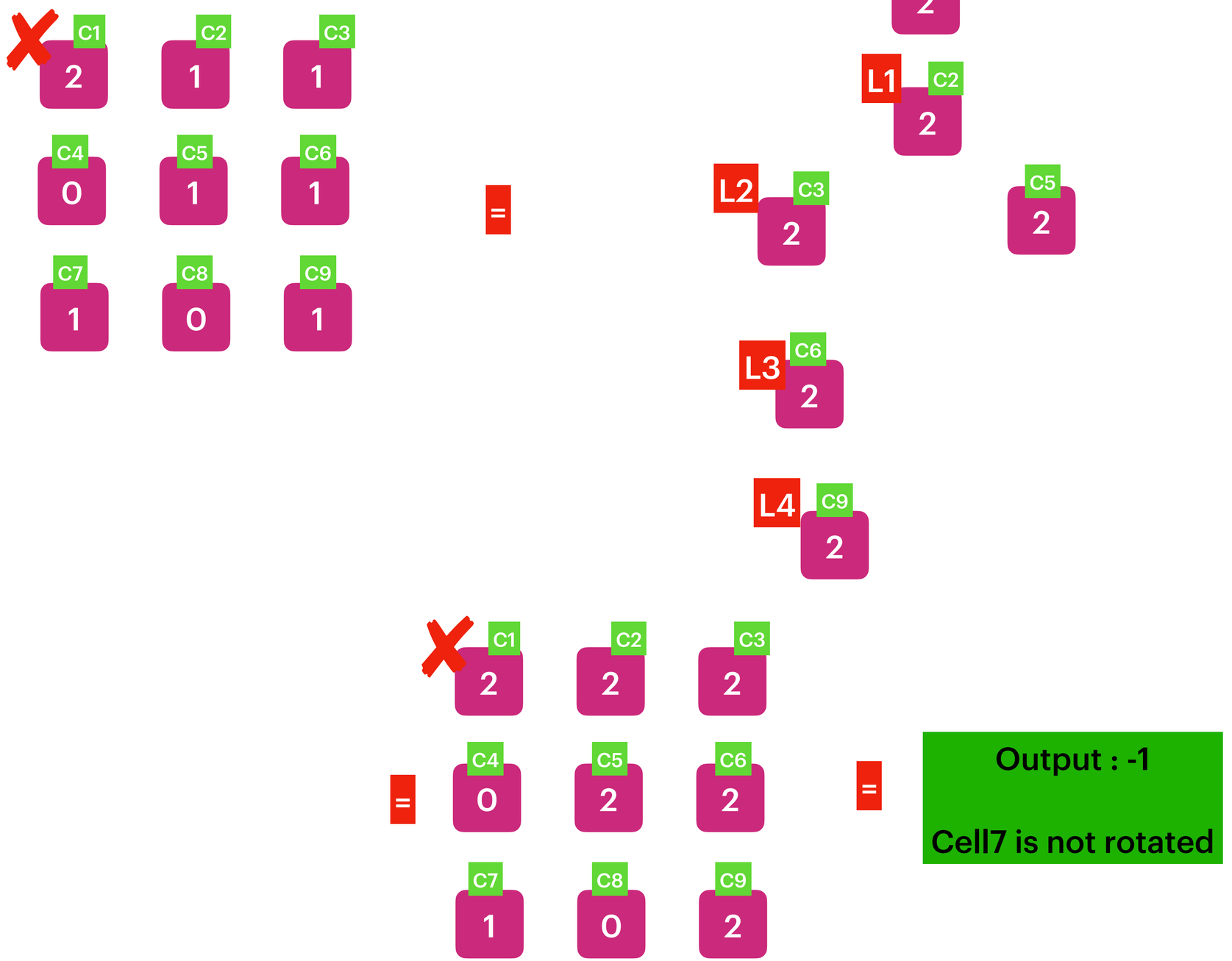
Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.

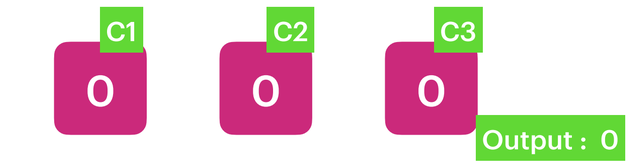
Constraints:

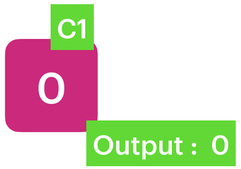
m == grid.length
n == grid[i].length
1 <= m, n <= 10
grid[i][j] is 0, 1, or 2.

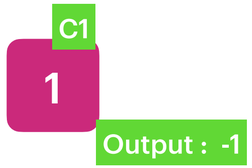

[
[2,1,1],
[1,1,0],
[0,1,1]
]
Minute : 1
Minute : 2
Minute : 3
Minute : 4
C0
C1
C2
R0
R1
R2

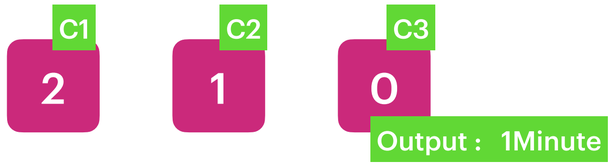

C1
2
C2
1
C3
1
C4
1
C5
1
C6
0
C7
0
C8
1
C9
1
=
L0
C1
2
L1
C2
2
C4
2
Minute1
L2
C3
2
C5
2
Minute2
L3
C8
2
Minute3
L4
C9
2
Minute4
C1
2
C2
2
C3
2
=
C4
2
C5
2
C6
0
C7
0
C8
2
C9
2


C1
2
C2
1
C3
1
C4
0
C5
1
C6
1
C7
1
C8
0
C9
1
=
C1
2
L1
C2
2
L2
C3
2
C5
2
L3
C6
2
L4
C9
2
C1
2
C2
2
C3
2
=
C4
0
C5
2
C6
2
C7
1
C8
0
C9
2
=
Output : -1
Cell7 is not rotated

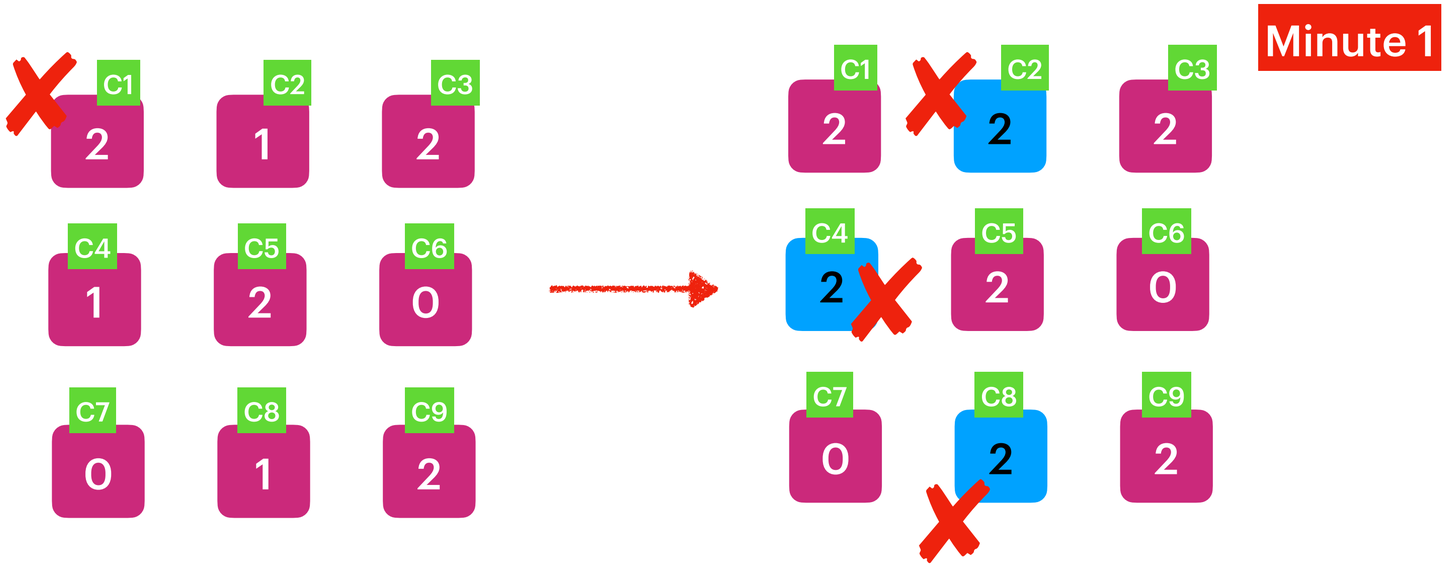

C1
0
C2
0
C3
0
Output : 0

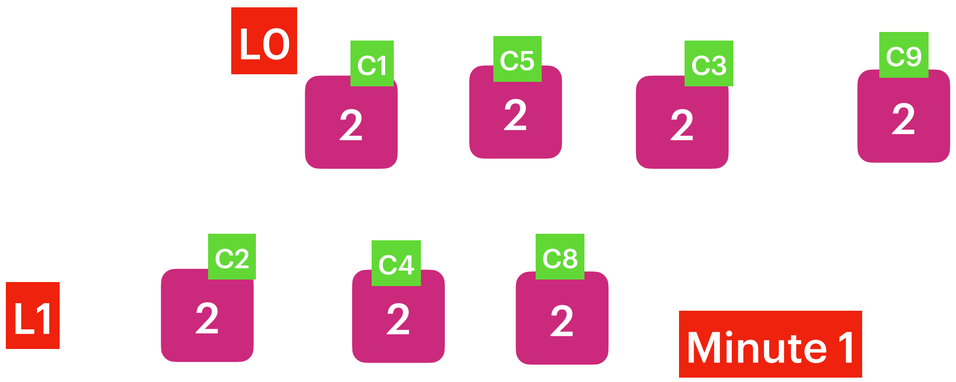

C1
0
Output : 0


C1
1
Output : -1


C1
2
C2
1
C3
0
Output : 1Minute


C1
0
C2
1
C3
0
Output : -1


C1
2
C2
1
C3
2
C4
1
C5
2
C6
0
C7
0
C8
1
C9
2
Minute 1
C1
2
C2
2
C3
2
C4
2
C5
2
C6
0
C7
0
C8
2
C9
2


LO
C1
2
C5
2
C3
2
C9
2
L1
C2
2
C4
2
C8
2
Minute 1

Number of Islands

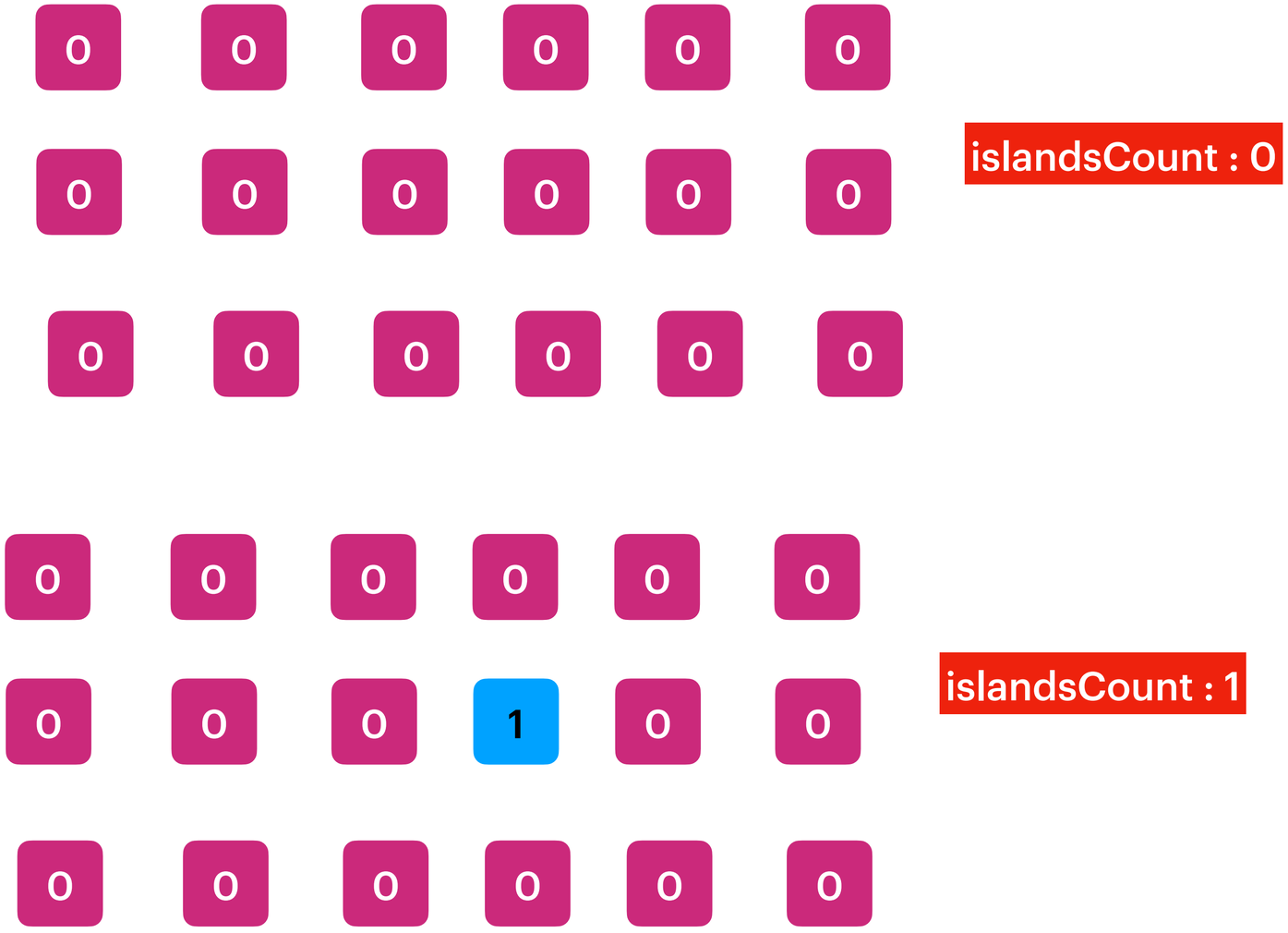
Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

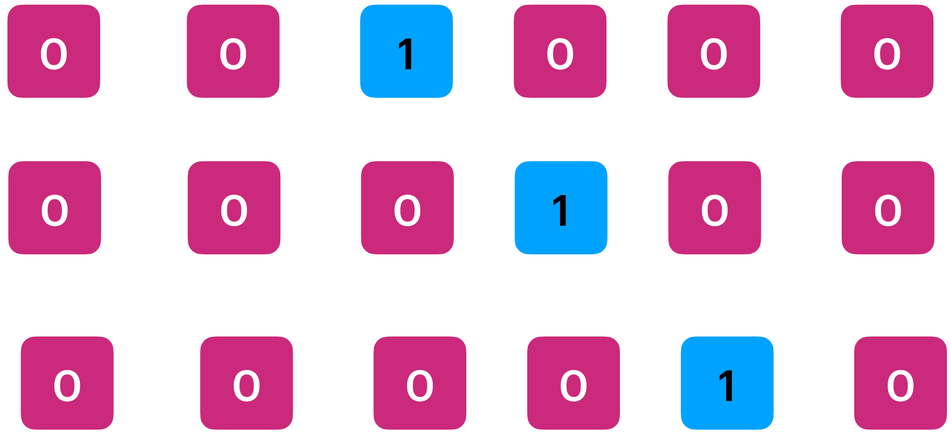
An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically.
You may assume all four edges of the grid are all surrounded by water.

Input: grid = [
["1","1","1","1","0"],
["1","1","0","1","0"],
["1","1","0","0","0"],
["0","0","0","0","0"]
]
Output: 1

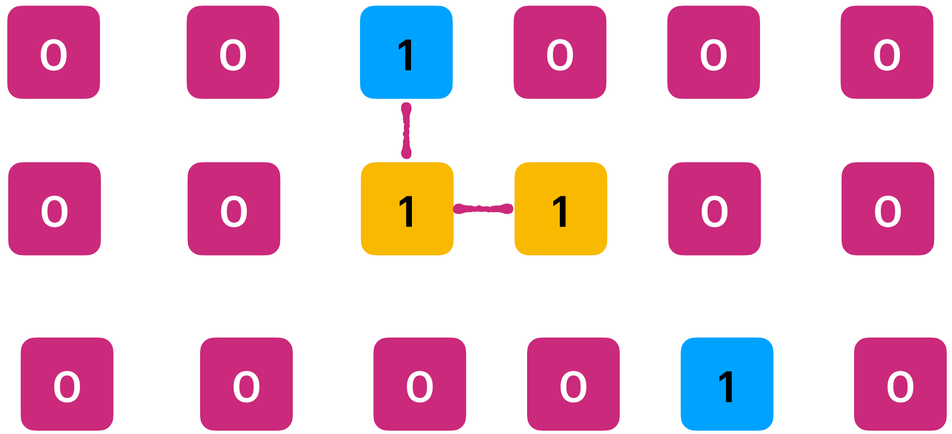
Input: grid = [
["1","1","0","0","0"],
["1","1","0","0","0"],
["0","0","1","0","0"],
["0","0","0","1","1"]
]
Output: 3

m == grid.length
n == grid[i].length
1 <= m, n <= 300
grid[i][j] is '0' or '1'.

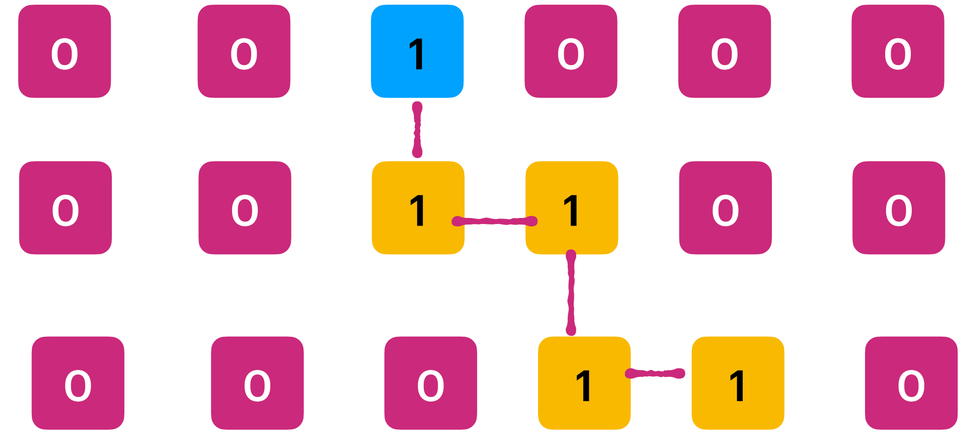

0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
islandsCount : 0
0 0 0 0 0 0
0 0 0 1 0 0
0 0 0 0 0 0
islandsCount : 1



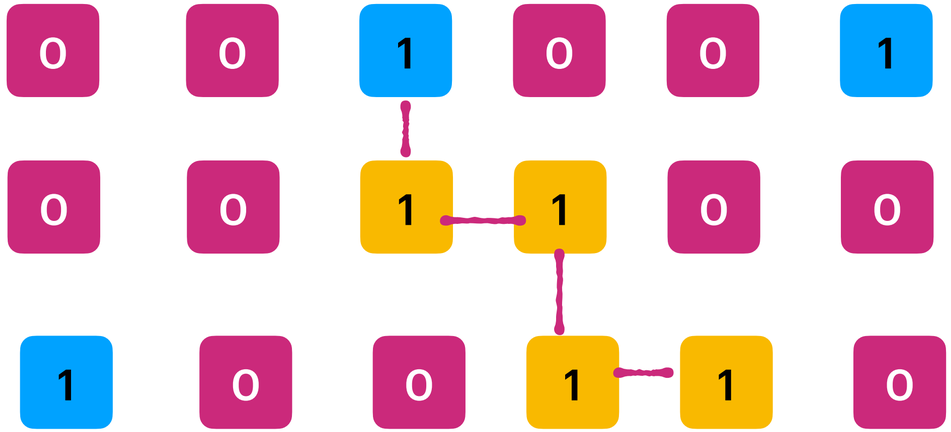
islandsCount : 3



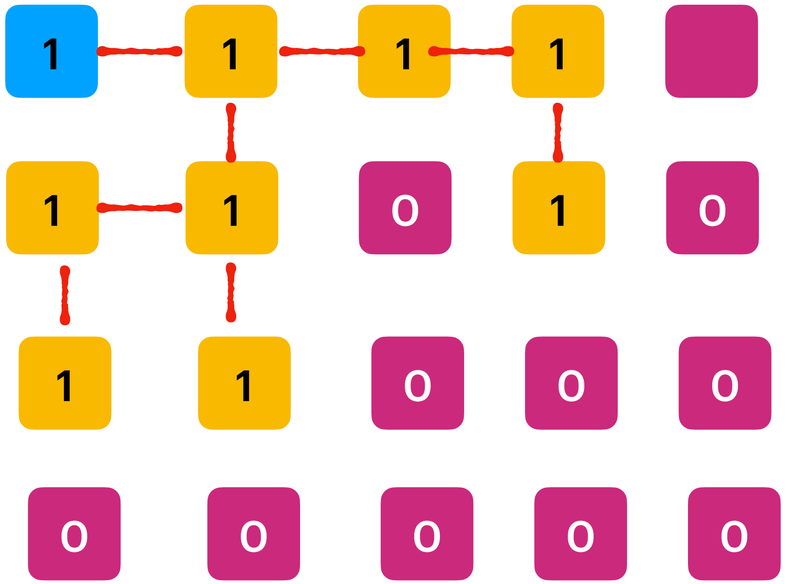
islandsCount : 2



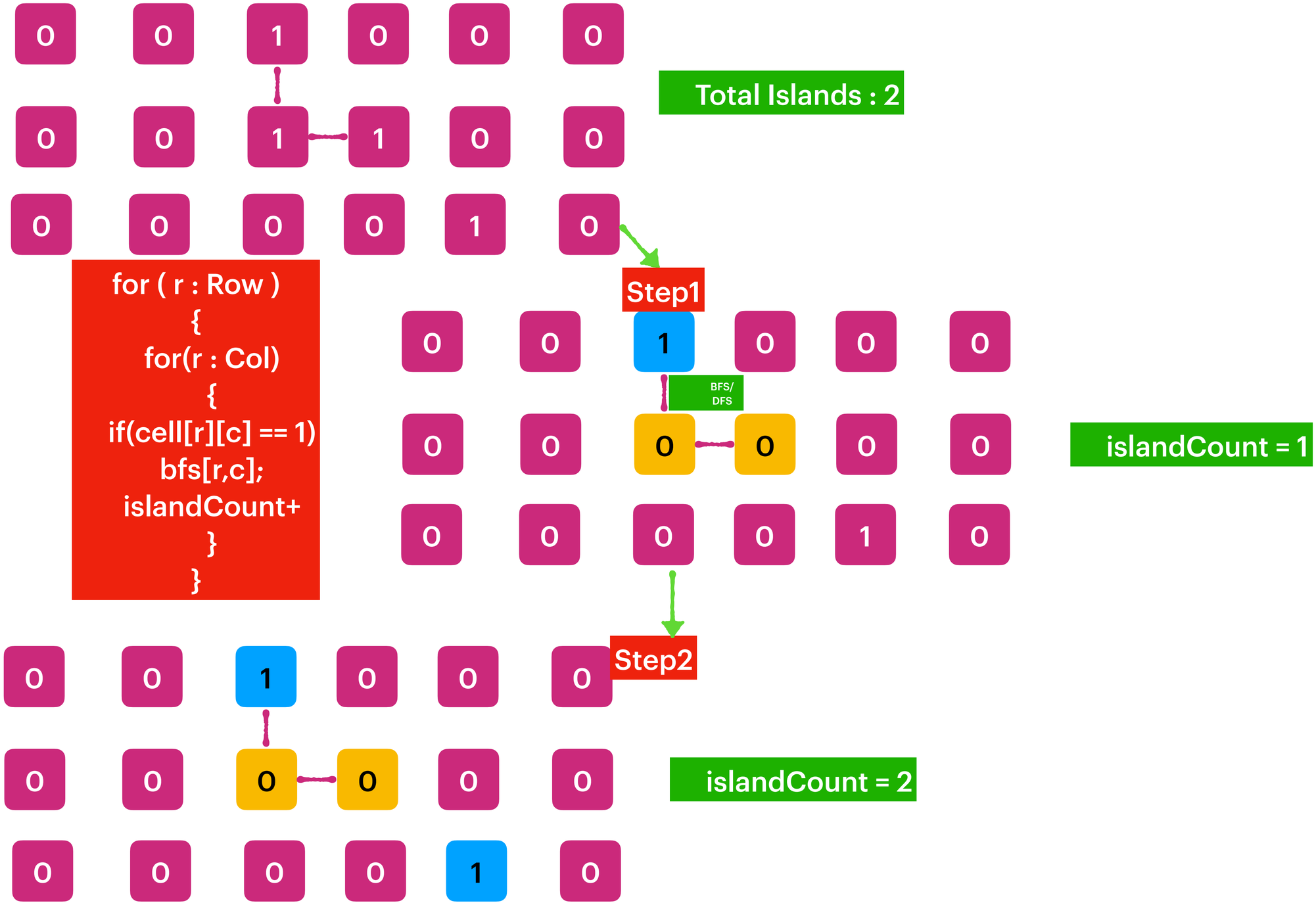
islandsCount : 1



islandsCount : 3



islandsCount : 1


0 0 1 0 0 0
0 0 1 1 0 0
0 0 0 0 1 0
Total Islands : 2
for (r : Row)
{
for(r : Col)
{
if(cell[r][c] == 1)
bfs[r,c];
islandCount+
}
}
Step1
0 0 1 0 0 0
BFS/
DFS
0 0 0 0 0 0
islandCount = 1
0 0 0 0 1 0
Step2
0 0 1 0 0 0
0 0 0 0 0 0
islandCount = 2
0 0 0 0 1 0