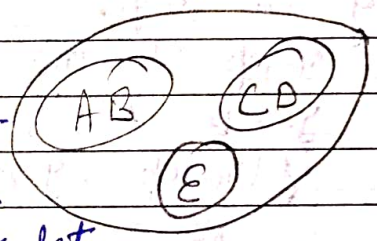
Disjoint-Set

→ A disjoint set data structure can be used in any problem, where we have a big universe (or) set and we have multiple subset of it as disjoint subset (NO two subset have anything in common). So if there is an element x, it belongs to only one subset (or) one partition of your whole set.

e.g.; Consider that there are 5 students in a classroom namely A, B, C, D, E. They will be denoted as 5 different subsets : {A}, {B} {C}, {D}, {E}. After some point of time, A become friends with B and C became friends with D. So A and B will now belong to a same set and C and D will now belong to another same set.



If at any point of time, we want to check that if any two student are friends (or) not, then we can simply check whether they belong to the same set.

There are two types of operations performed on disjoint set Data Structure.

(a) union (A, B) : This operation tells us to merge the sets containing elements A and B respectively by performing a union operation on sets.

(b) Find (A) : This operation tells to find the subset to which the element A belongs.

UNION (0) (1) (2) (3) (4)

(1) union (0, 2) (0,2) (1) (3) (4)
now find (0) & find (2) should return same value.

(2) union (2, 4) (0,2,4) (1) (3)
now find (0), find (2) & find (4) should return same value.

DECEMBER
21
'24
SATURDAY
51st Week
356-010

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

DEC 2024

Worst case: O(n) for both union & find

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

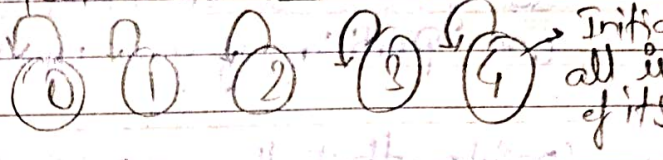
now find(1) and find(3) should return same values.

Implementation

```
n = 5
parent = [i for i in range(n)]
def find(n):
    if parent[n] == n:
        return n
    return find(parent[n])

def union(x,y):
    x_rep = find(x)
    y_rep = find(y)
    if (x_rep == y_rep):
        return
    parent[y_rep] = x_rep
```

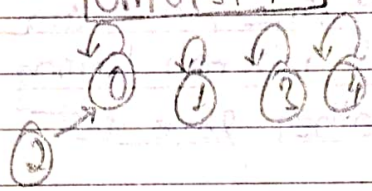
parent [ ] | 0 | 1 | 2 | 3 | 4 |



Initially all is parent of itself.

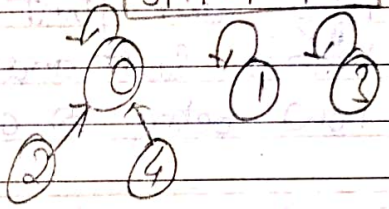
union(0,2):

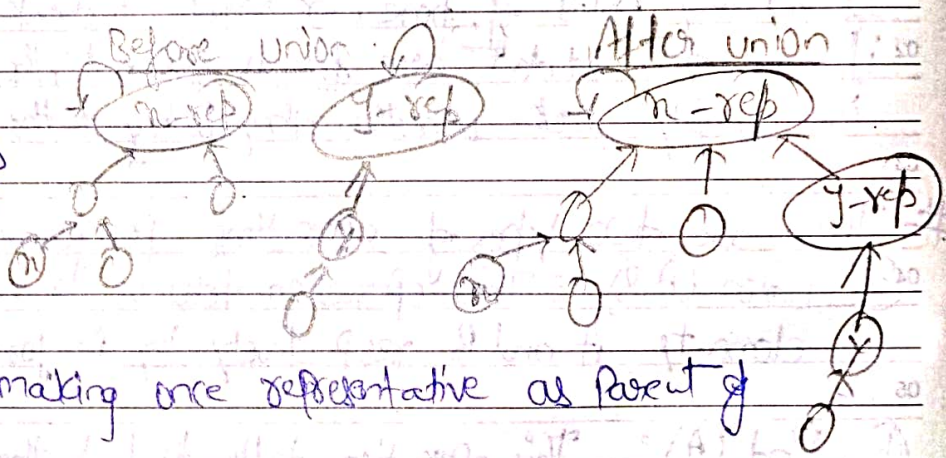
0	1	3	4



union(2,4):

0	1	0	3	0





So, union is simply making one representative as parent of other representative.

→ The implementation of union() & find() is naive and takes O(n) time in worst case. These method can be improved to O(logn) using union by Rank (or) Height.

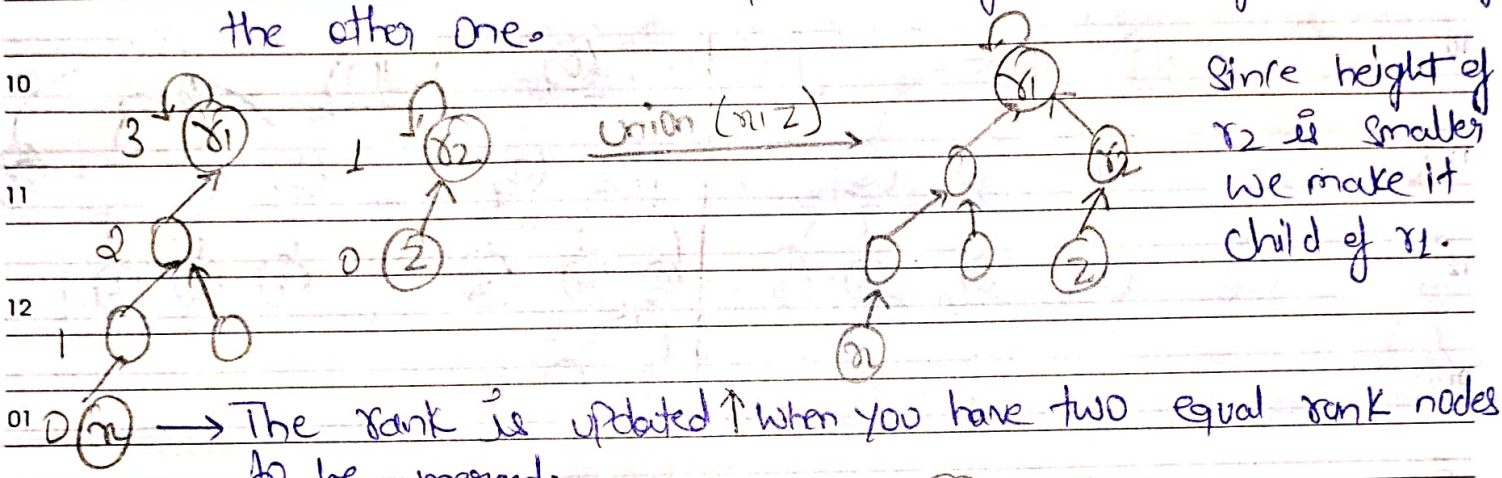
Applications

(1) Detecting a cycle in a Graph. [It can be easily solved using the disjoint set & union-find algorithms.

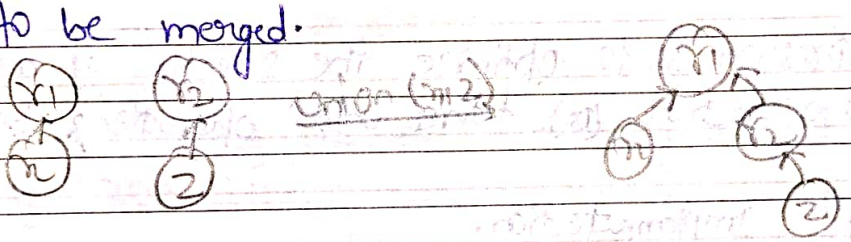
Union by Rank

array

(a) We use an extra array, rank in the union operation.
(b) It typically stores height.
(c) The idea is to make representative of smaller height as child of the other one.



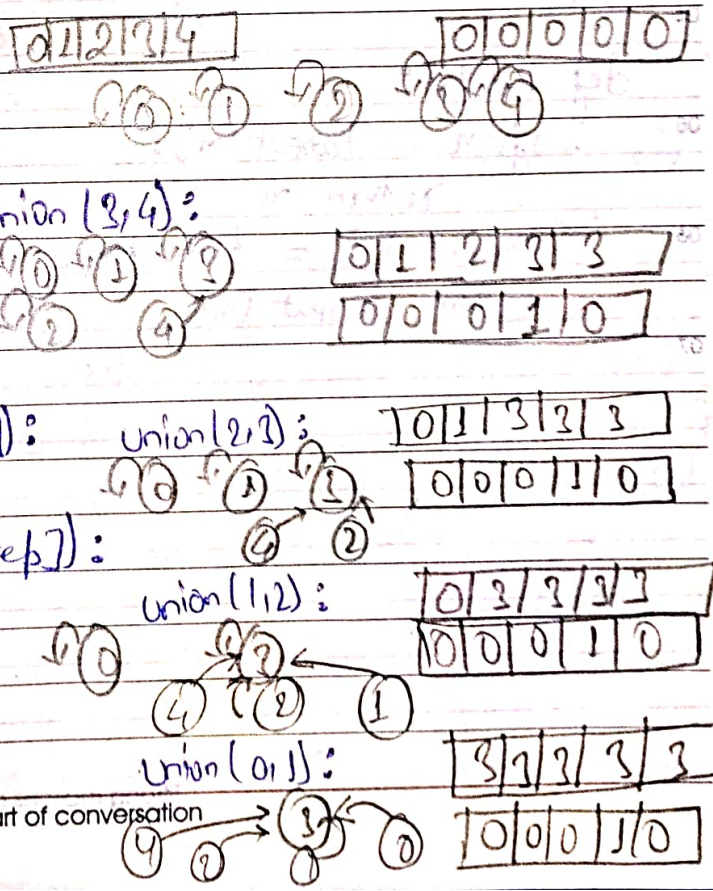
→ The rank is updated ↑ when you have two equal rank nodes to be merged.



Implementation

```
n = 5
Parent = [i for i range(n)]
rank = [0 for i range(n)]
def union(x,y):
    x-rep = find(x)
    y-rep = find(y)
    if (x-rep == y-rep): return
    if (rank[x-rep] < rank[y-rep]):
        Parent[x-rep] = y-rep
    elif (rank[x-rep] > rank[y-rep]):
        Parent[y-rep] = x-rep
    else:
        Parent[y-rep] = x-rep
        rank[x-rep] += 1
```

TC = O(log n)



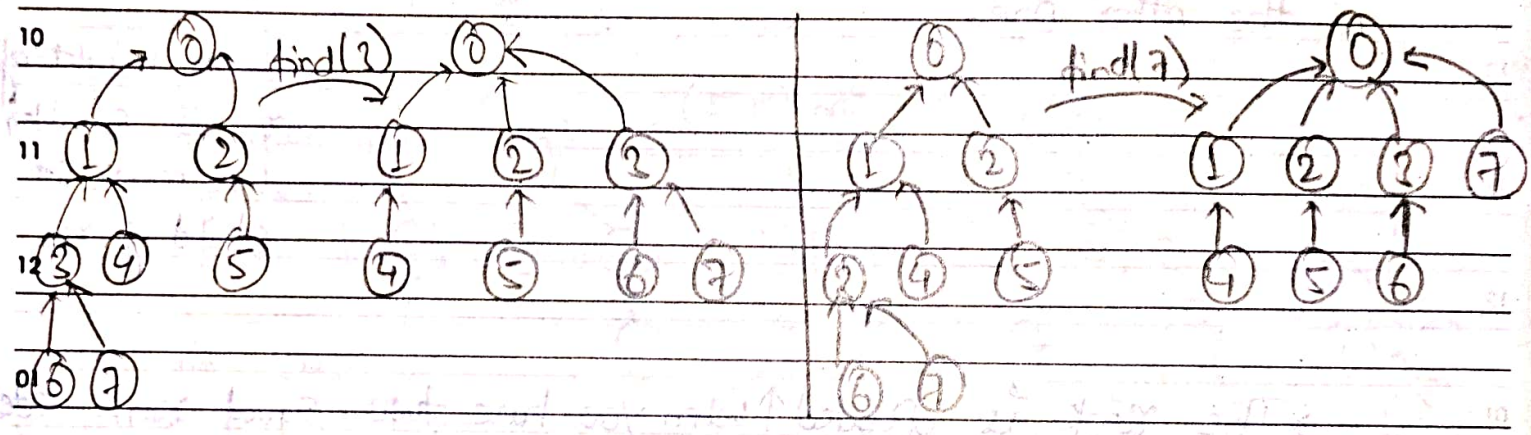
DECEMBER
23
'24
MONDAY
52nd Week
358-008

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

DEC 2024

Path Compression

→ The idea is to modify and optimize the tree in the find().
→ We make parent of all nodes (on the path from given node to root) as root.



Idea of path compression is to optimize the structure of a tree so that the future find operation (or) future union operations become faster

Modified find fn Implementation.

```
def find(x):
    if x == parent[x]:
        return x
    parent[x] = find(parent[x])
    return parent[x]
```

Time for m operation on n elements
$O(m \alpha(n))$
Where $\alpha(n) \leq 4$
$\alpha(n)$ is inverse ackerman fn.

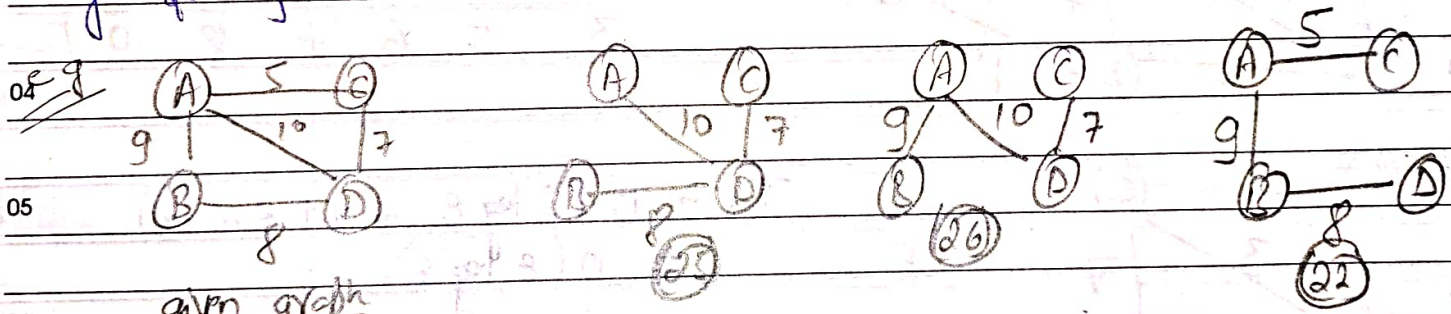
Kruskal's Algorithms

Spanning tree :- Spanning tree of a graph is a tree which connects all the vertices in the spanning tree.
There should be still a path from every vertex to any other vertex and it should be a tree which means there should not be any cycles.

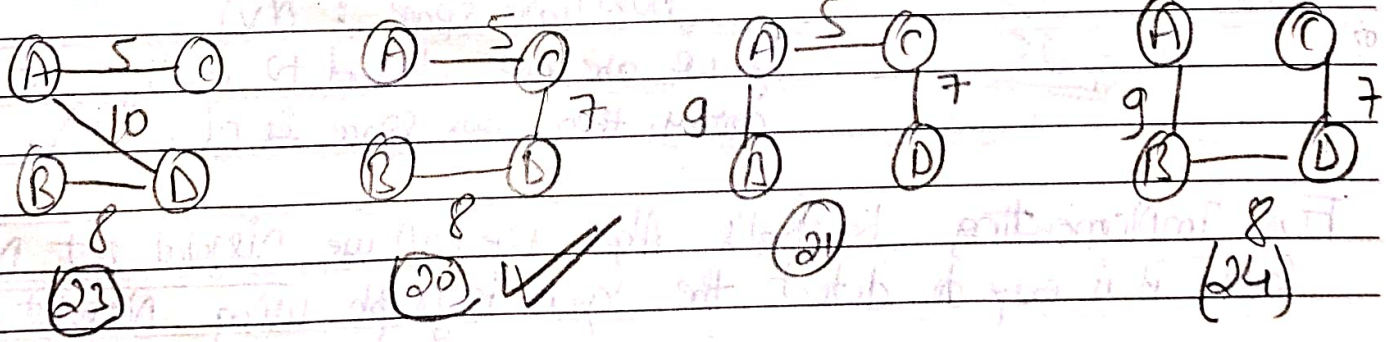
Spanning tree of a graph with N vertices has (N-1) Edges.

Minimum Spanning tree :- A MST for a weighted, connected, undirected graph is a spanning tree with a weight less than (or) equal to the weight of ever other spanning tree.

The weight of spanning tree is the sum of weight given to each Edge of spanning tree.



given graph



Different Spanning tree

Sometimes the best gain is to lose

DECEMBER
25 '24
WEDNESDAY
52nd Week
360-006

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

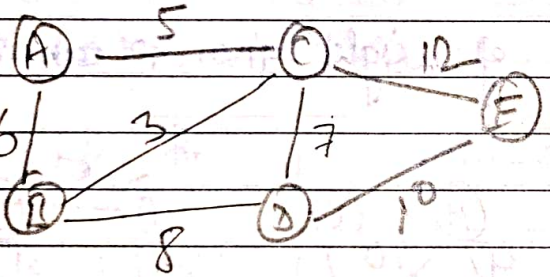
DEC 2024

Kruskal's Algo is used to find the minimum cost spanning tree uses the greedy Approach.
The Greedy approach is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

Steps:

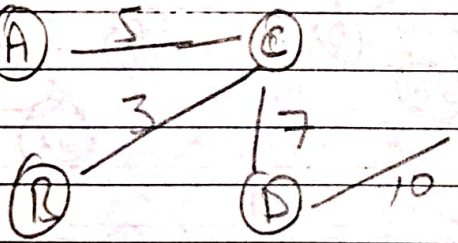
1. Sort all Edges in increasing Order.
2. Initialize : MST = [], res = 0.
3. Do following for every edge e', while MST size doesn't become V-1.
   (a) If adding e to MST does not cause a cycle
   MST = MST ∪ {e}
   res = res + e.weight.
4. Return res.

eg



Sorted Edge

BC	AC	AB	CD	BD	DE	CE
3	5	6	7	8	10	12



res = 25

$TC = O(E \log E + V + E\alpha(V))$
$= O(E \log E)$

Auxiliary space : O(V)
If we are not allowed to modify the input array, then aux space is O(V+E).

For Implementing Kruskal's Algo we will use Disjoint set DS. Cause it is easy to detect the cycle in graph using Disjoint set.

For Implementation see chat GPT.