Disjoint set - Data structure

(*) Two sets are said to be disjoint if they have no elements in common.

$S_1 = \{e_3, e_5, e_7\}$

$S_2 = \{e_4, e_2, e_8\}$

$S_1 \cap S_2 = \emptyset$

So, S_1, S_2 are disjoint sets.

* Union(x, y) – Combine or merge two sets x and y into a single set.

- Before Union:

$\{\{e_3, e_5, e_7\}, \{e_4, e_2, e_8\}, \{e_9\}, \{e_1, e_6\}\}$

(Representative: e_5, Representative: e_1)

- After Union(e_5, e_1):

$\{\{e_3, e_5, e_7, e_1, e_6\}, \{e_4, e_2, e_8\}, \{e_9\}\}$

After union

* If find(a) == find(b) is true only if a and b in the same set.

* Find(x) – Return the name of the set containing x.

- Example: $\{\{e_3, e_5, e_7, e_1, e_6\}, \{e_4, e_2, e_8\}, \{e_9\}\}$

- Find(e_1) = e_5

- Find(e_4) = e_8

Representative

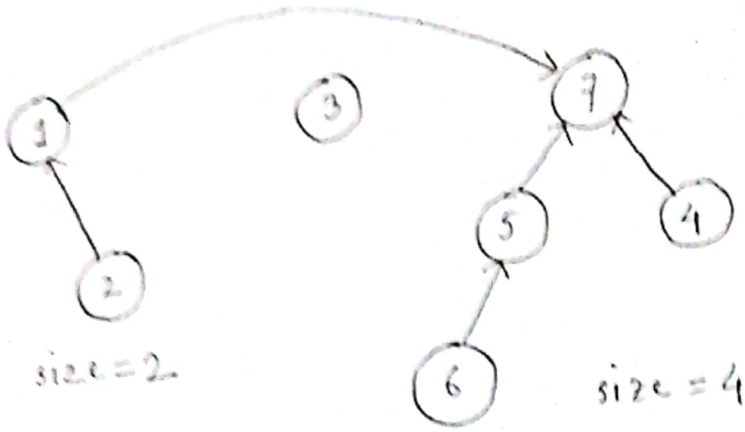
Since Find(e_1) ≠ Find(e_4)

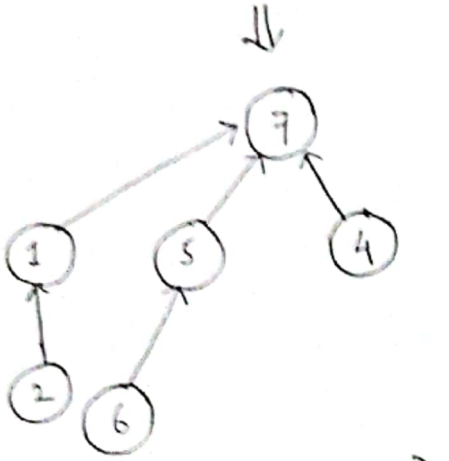
so, e_1 and e_4 are in the different sets.

Union Operation in Tree

Union(x, y) - Assuming x and y roots, point x to y

x = 3
y = 7
Union(3, 7)



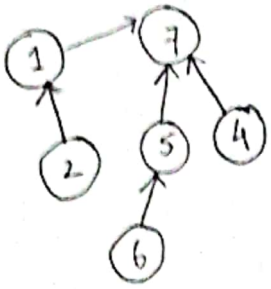


After Union(1, 7)

Union by rank

(Also called Union by size)

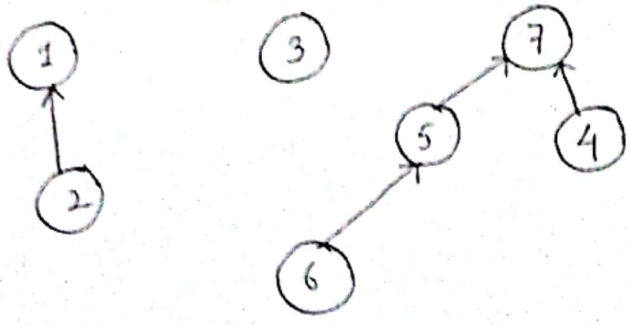
- Always point the smaller tree to the root of the larger tree



since 4 > 2, so, 7 will be root

Find Operation in Tree

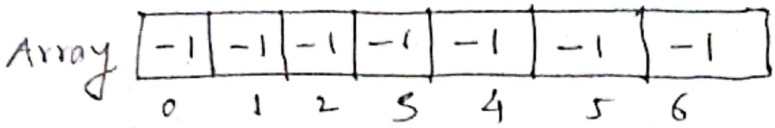
Find(x): Follow x to root and return root



Find(6) = 7

(2)

Initiaze:



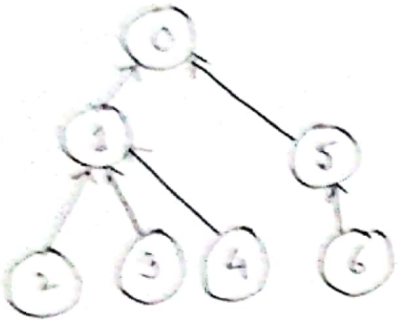

```
for (i=0; i<size; i++){
    Array[i] = -1;
}
```

Merge two subtrees if they are different

```
void UNION (int a, int b) {
    int root1 = FIND(a);
    int root2 = FIND(b);
    if (root1 != root2) {
        Array [root1] = root2;
    }
}

int FIND( int curr) {
    while ( Array[curr] != -1) {
        curr = Array[curr];
    }
    return curr;
}
```

Find/Union Operation Example:



Array

-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6

Value of a & b	Find(a)	Find(b)	Array[root1] = root2
~~a=1, b=1~~	~~root1 = 1~~	~~root2 = 1~~	~~Array[1] = 1~~
a=2, b=1	root1 = 2	root2 = 1	Array[2] = 1
a=3, b=1	root1 = 3	root2 = 1	Array[3] = 1
a=4, b=1	root1 = 4	root2 = 1	Array[4] = 1
a=3, b=4	root1 = -1	root2 = -1	×
a=6, b=5	root1 = 6	root2 = 5	Array[6] = 5
a=1, b=0	root1 = 1	root2 = 0	Array[1] = 0
a=5, b=0	root1 = 5	root2 = 0	Array[5] = 0

Array

Array

-1	-1	1	-1	-1	-1	-1
0	1	2	3	4	5	6

Array

-1	-1	1	1	-1	-1	-1
0	1	2	3	4	5	6

Array

-1	-1	1	1	1	-1	-1
0	1	2	3	4	5	6

×

Array

-1	-1	1	1	1	-1	5
0	1	2	3	4	5	6

Array

-1	0	1	1	1	-1	5
0	1	2	3	4	5	6

Array

-1	0	1	1	1	0	5
0	1	2	3	4	5	6

Kruskal's Algorithm

Minimum Spanning Tree

Kruskals():

Sort edges in increasing order of length $\{e_1, e_2, e_3, \ldots, e_m\}$

$T = \{\}$

for i = 1 to m

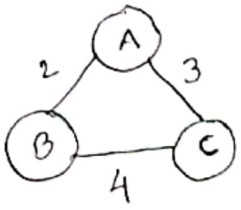
  if e_i does not add a cycle:

    add e_i to T

return T.

* But how can we determine that adding e_i to T won't add a cycle?

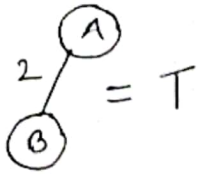
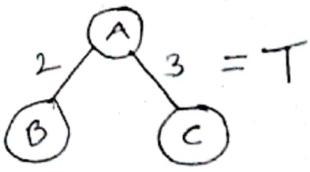
Ans:



$T = \{\}$

$E = \text{edges} = \{AB, AC, BC\}$

$\text{cost} = \{2, 3, 4\}$

Selected Edge	Accept / Reject	Total cost	Tree
$e_1 =$ "AB"	Accept	2	 = T
$e_2 =$ "AC"	Accept	2+3	 = T
$e_3 =$ "BC"	Reject (Addition will form cycle)	X	X

Minimum Spanning Tree: