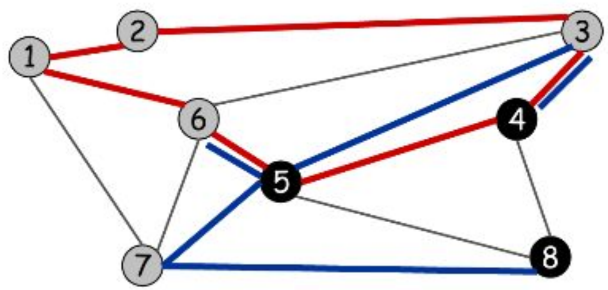
Greedy algorithms

CMPSC 465 - Yana Safonova

The minimum spanning tree

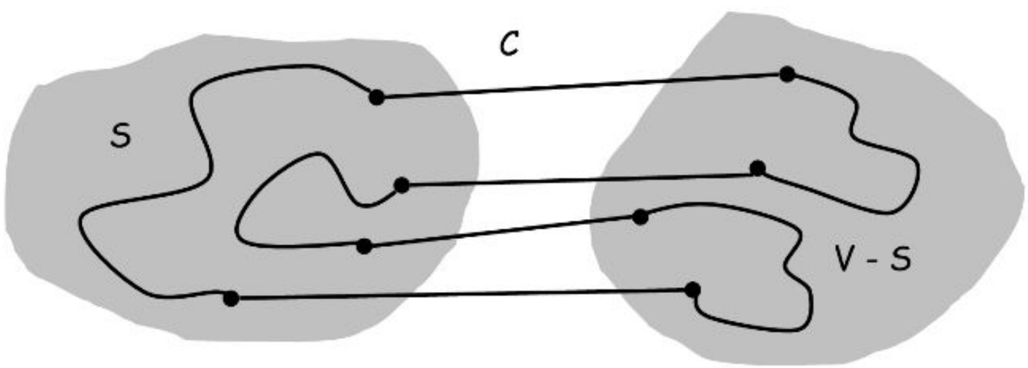
Kruskal's algorithm - proof of correctness

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

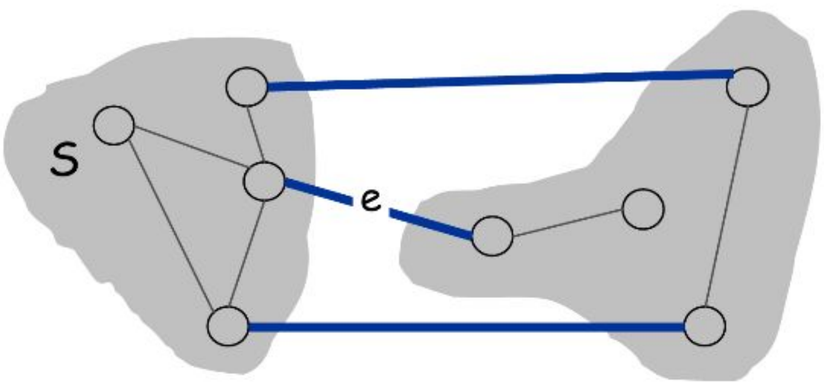
Pf. (by picture)



A cycle has to enter and leave the cut the same amount of times

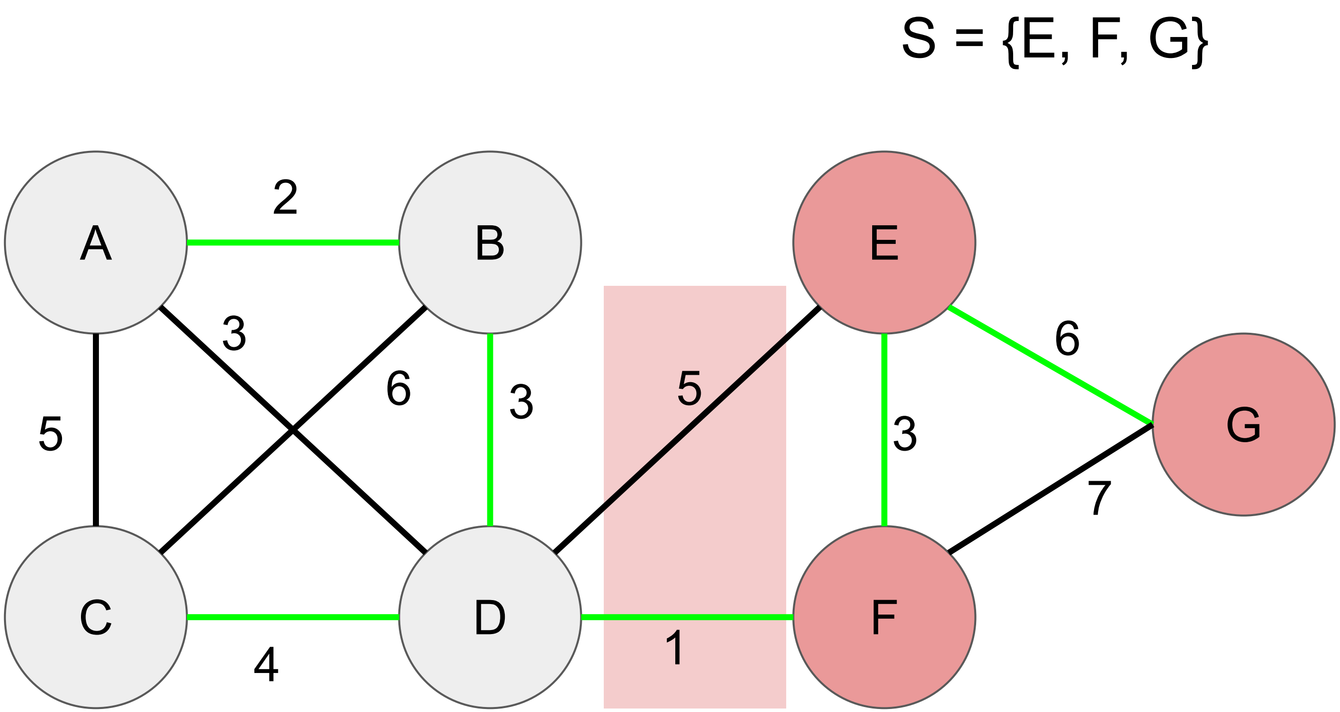
Kruskal’s algorithm - proof of correctness

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.



e is in the MST

The cut property



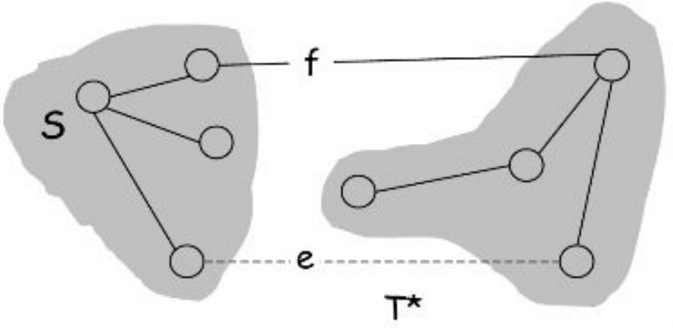
Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Kruskal’s algorithm - proof of correctness

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

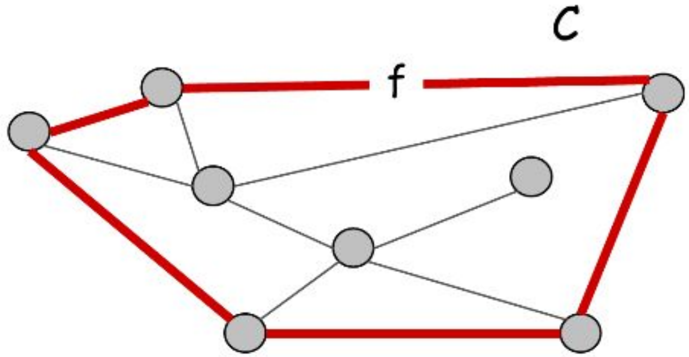
Pf. (exchange argument)

- Suppose e does not belong to T*, and let's see what happens.
- Adding e to T* creates a cycle C in T*.
- Edge e is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say f, that is in both C and D.
- T' = T* $\cup$ { e } - { f } is also a spanning tree.
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction. ▪



Kruskal's algorithm - proof of correctness

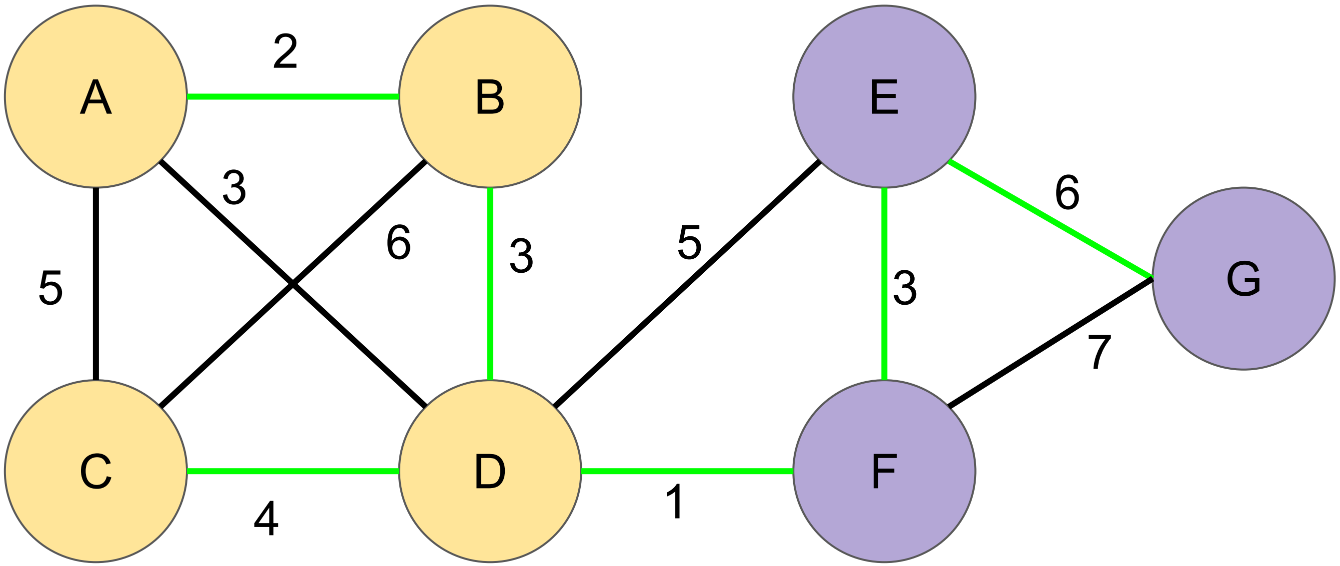
Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



f is not in the MST

The cycle property

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



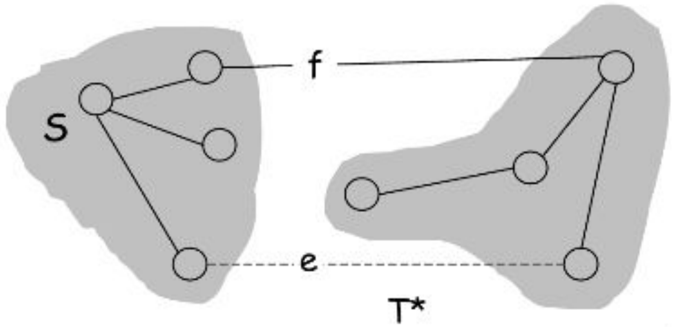
(A, C, D, B)
(A, D, B, C)
(D, E, F)
(E, F, G)

Kruskal’s algorithm - proof of correctness

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T^* does not contain f.

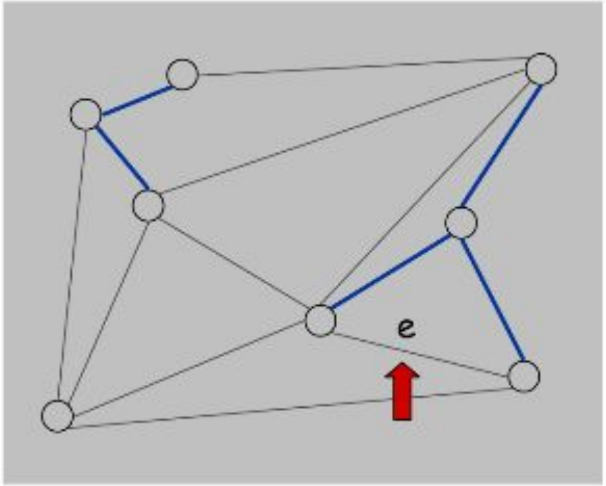
Pf. (exchange argument)

- Suppose f belongs to T^*, and let's see what happens.
- Deleting f from T^* creates a cut S in T^*.
- Edge f is both in the cycle C and in the cutset D corresponding to S
  $\Rightarrow$ there exists another edge, say e, that is in both C and D.
- $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction. ▪

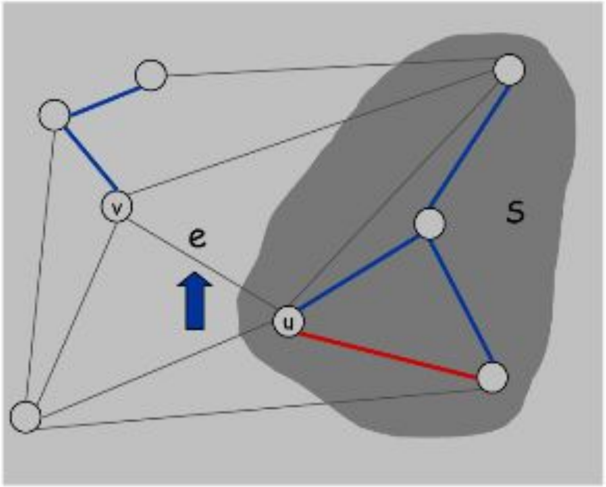


Kruskal's algorithm - proof of correctness

- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1



Case 2

Kruskal's algorithm - implementation

Implementation. Use the union-find data structure.

- Build set T of edges in the MST.
- Maintain set for each connected component.

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m          are u and v in different connected components?
      (u,v) = ei
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }                    merge two components
   return T
}
```

Disjoint sets or union-find sets or merge-find sets

What is a disjoint set?

A data structure that is also known as the **disjoint set** or the **merge-find set**

Example:

There are 5 people: A, B, C, D, and E.

A is B's friend, B is C's friend, and D is E's friend, therefore, the following is true:

- A, B, and C are connected to each other
- D and E are connected to each other

{A, B, C} and {D, E} are two disjoint sets

How can we quickly compute if two people are in the same friend group?

Disjoint set - description

Operations:

Union(A, B): Connect two elements A and B

Find(A, B): Find whether the two elements A and B are connected

Example:

A set of elements S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

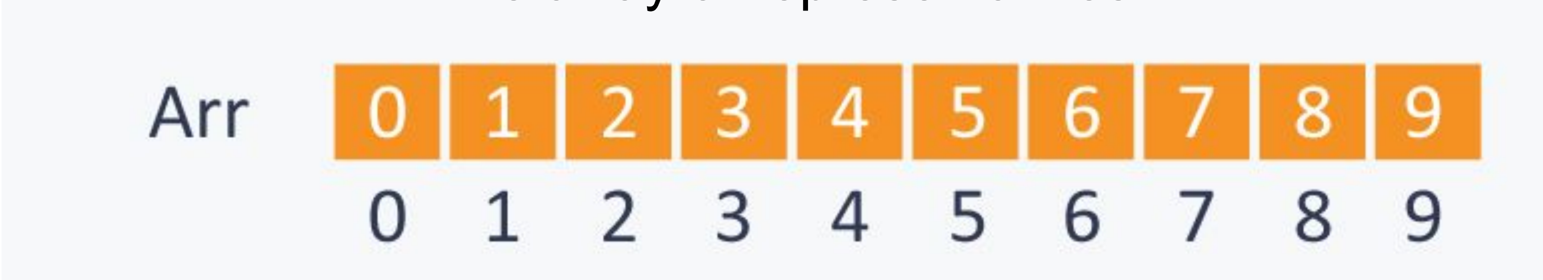
Arr[ ] that is indexed by elements of sets, which are of size N (N = 10) is used for the operations of *union* and *find*.

Assumption:

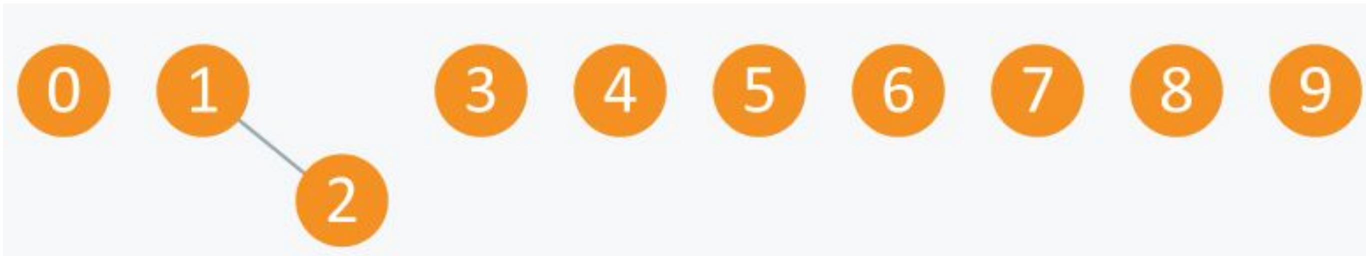
A and B objects are connected only if arr[A] = arr[B].

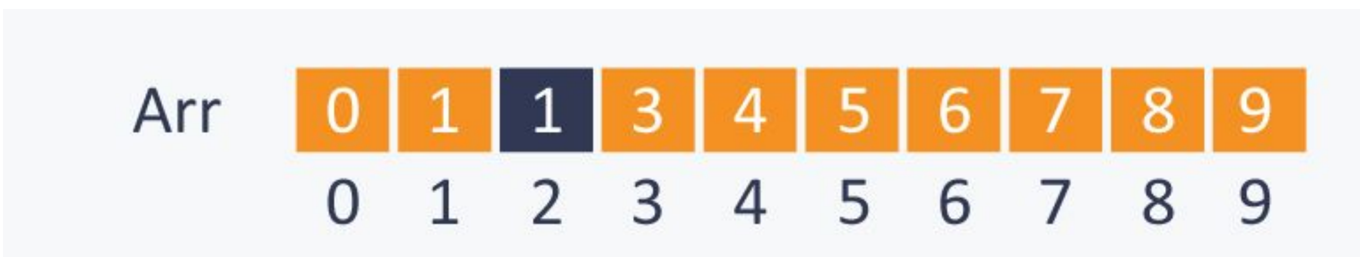
Disjoint set

The array of representatives

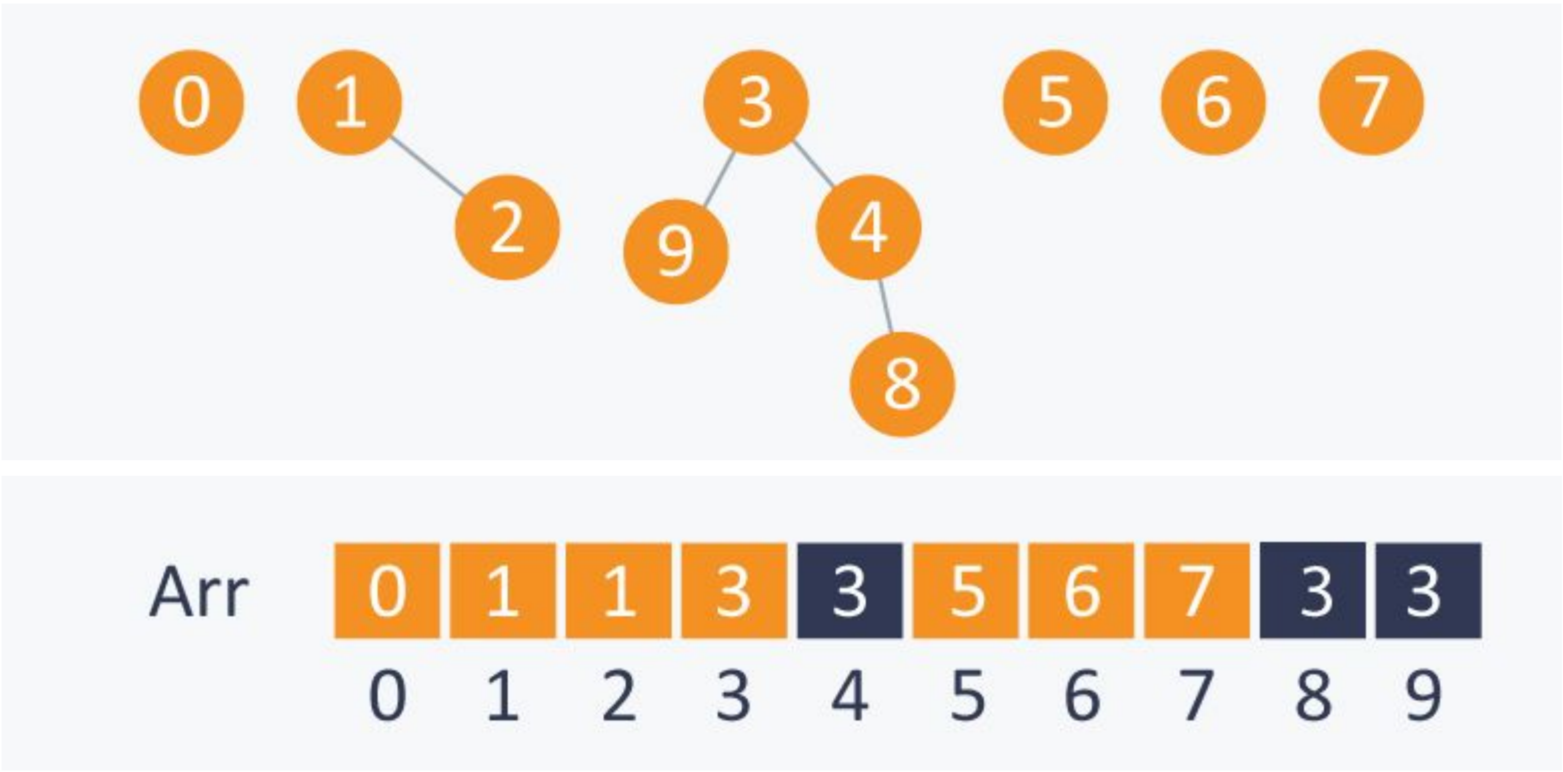


Disjoint set - Union(2, 1)

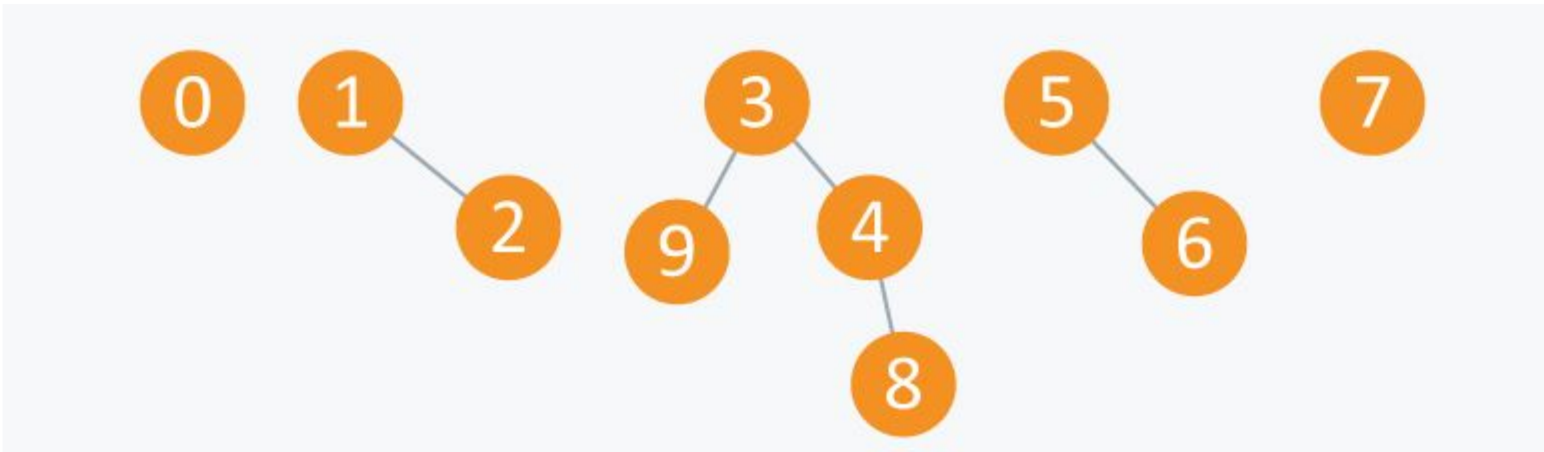


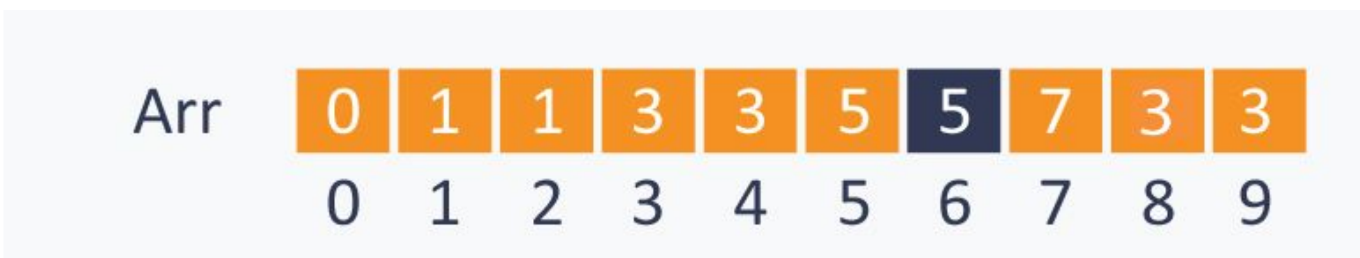


Disjoint set - Union(4, 3), Union(8, 4), Union(9, 3)

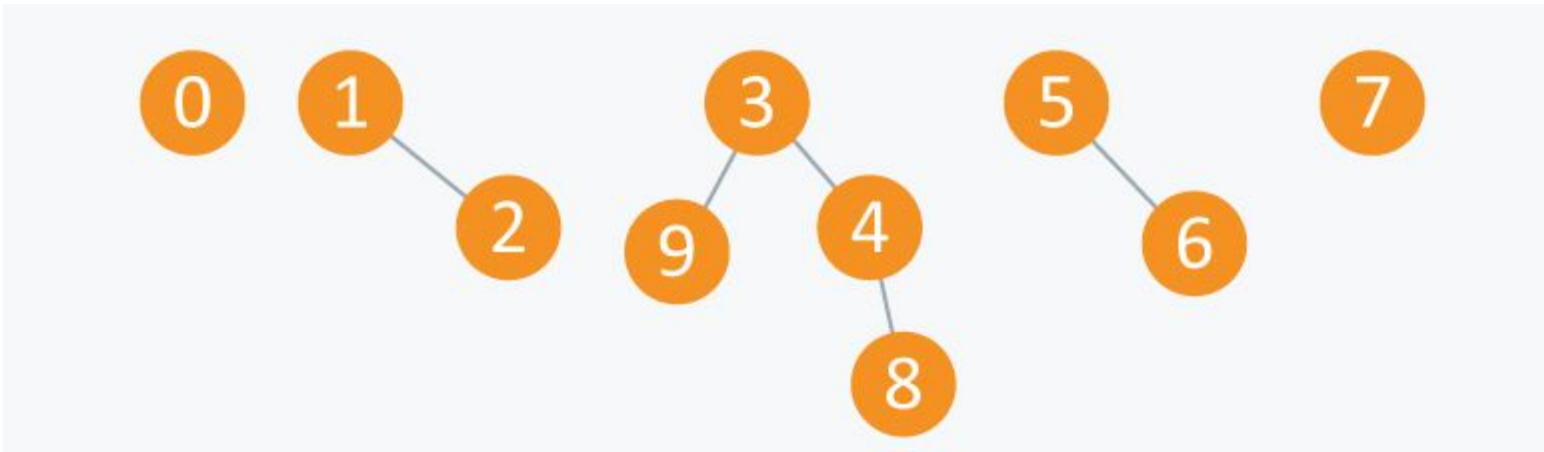


Disjoint set - Union(6, 5)





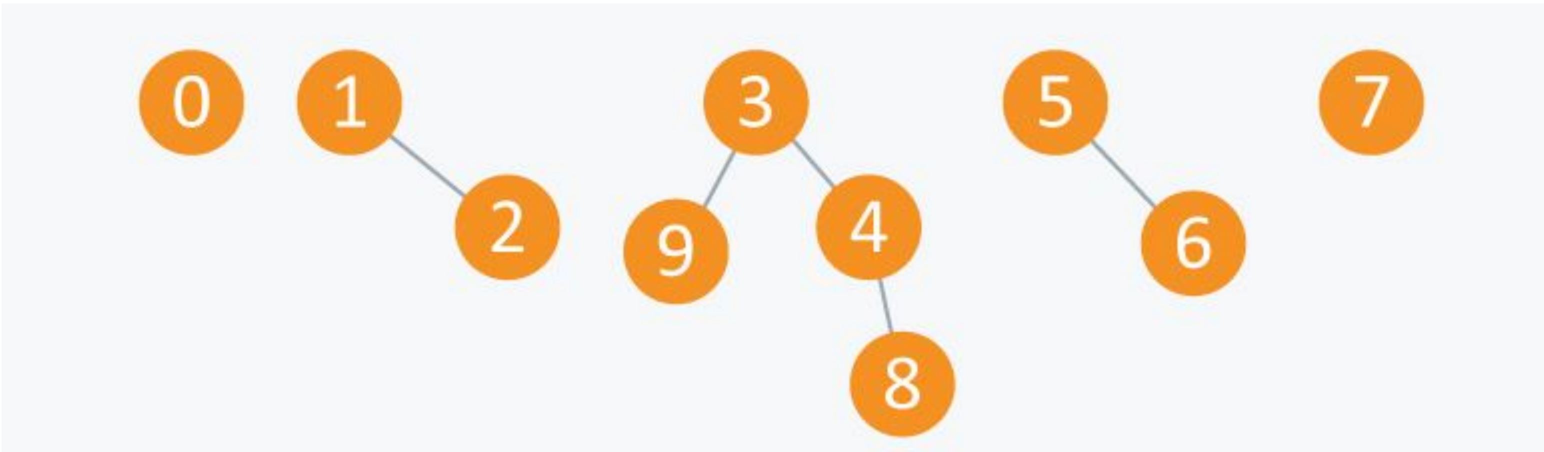
Disjoint set - 5 disjoint sets

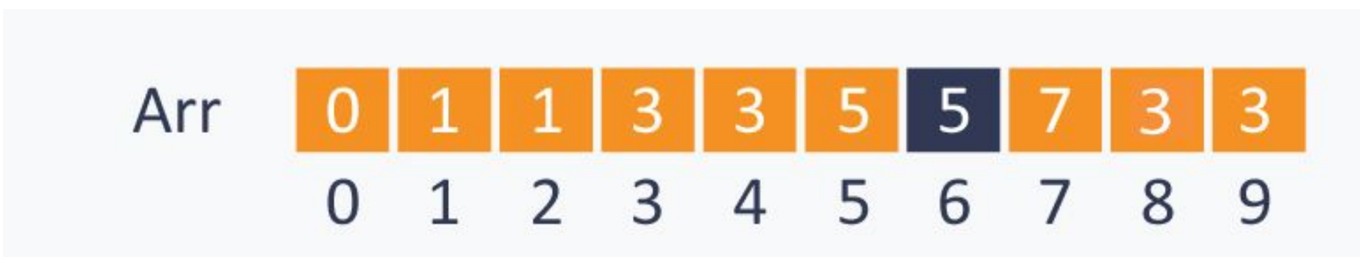


Arr	0	1	1	3	3	5	5	7	3	3
	0	1	2	3	4	5	6	7	8	9

- First subset comprises the elements {3, 4, 8, 9}
- Second subset comprises the elements {1, 2}
- Third subset comprises the elements {5, 6}
- Fourth subset comprises the elements {0}
- Fifth subset comprises the elements {7}

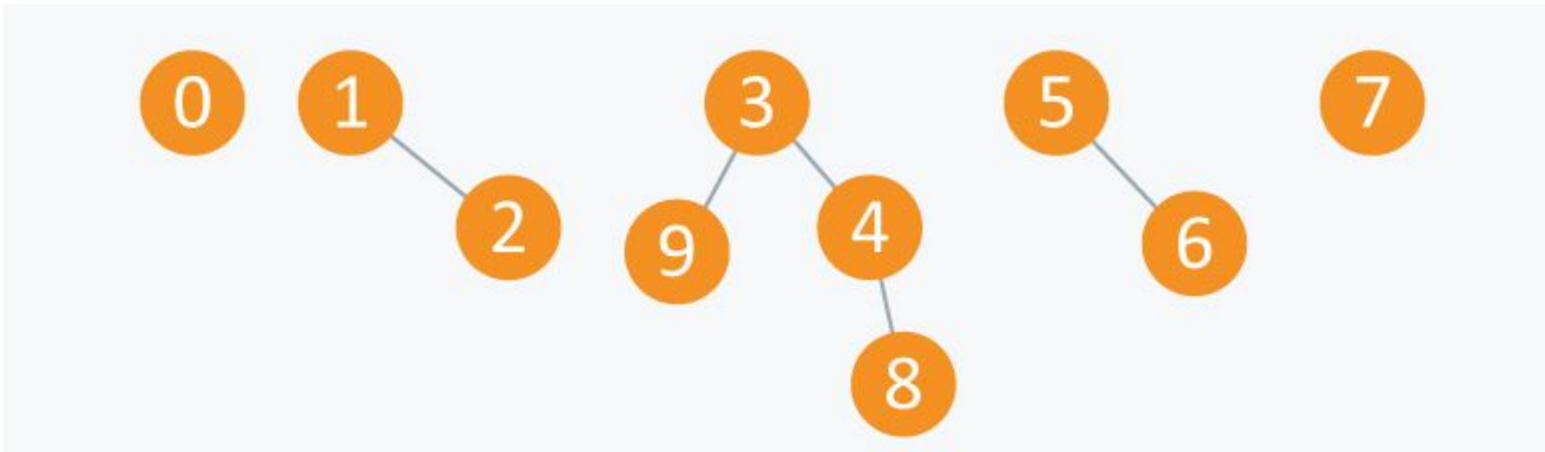
Disjoint set - find operations





- Find(0, 7): 0 and 7 are disconnected: FALSE
- Find(8, 9): Although 8 and 9 are not connected directly, there is a path that connects them: TRUE

Disjoint set - final representatives



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

Disjoint set - naive implementation

```
    def initialize(Arr, N):
        for i in range(N):
            Arr[ i ] = i

    # returns true if A and B are connected, else returns
false
    def find(Arr, A, B):
        return Arr[ A ] == Arr[ B ]
        #if Arr[ A ] == Arr[ B ]:
        #    return True
        #else:
        #    return False

    #change all entries from Arr[ A ] to Arr[ B ]
    def union(Arr, N, A, B):
        TEMP = Arr[ A ]
        for i in range(N):
            if Arr[ i ] == TEMP:
                Arr[ i ] = Arr[ B ]
```

Disjoint set - naive implementation

```
    def initialize(Arr, N):
        for i in range(N):
            Arr[ i ] = i

    # returns true if A and B are connected, else returns
false
    def find(Arr, A, B):
        return Arr[ A ] == Arr[ B ]
        #if Arr[ A ] == Arr[ B ]:
        #    return True
        #else
        #    return False

    #change all entries from Arr[ A ] to Arr[ B ]
    def union(Arr, N, A, B):
        TEMP = Arr[ A ]
        for i in range(N):
            if Arr[ i ] == TEMP:
                Arr[ i ] = Arr[ B ]
```

O(N) and $O(N^2)$ if applied to all elements

Kruskal’s algorithm + naive disjoint sets

- M is the number of edges
- N is the number of vertices
- The maximum number of M is N(N - 1) / 2 or $O(N^2)$
- $O(M) * O(N) = O(N^3)$ - too slow!

```
Kruskal(G, c) {
   Sort edges weights so that c₁ ≤ c₂ ≤ ... ≤ cₘ.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m        are u and v in different connected components?
      (u,v) = eᵢ
      if (u and v are in different sets) {
         T ← T ∪ {eᵢ}
         merge the sets containing u and v
      }                  merge two components
   return T
}
```

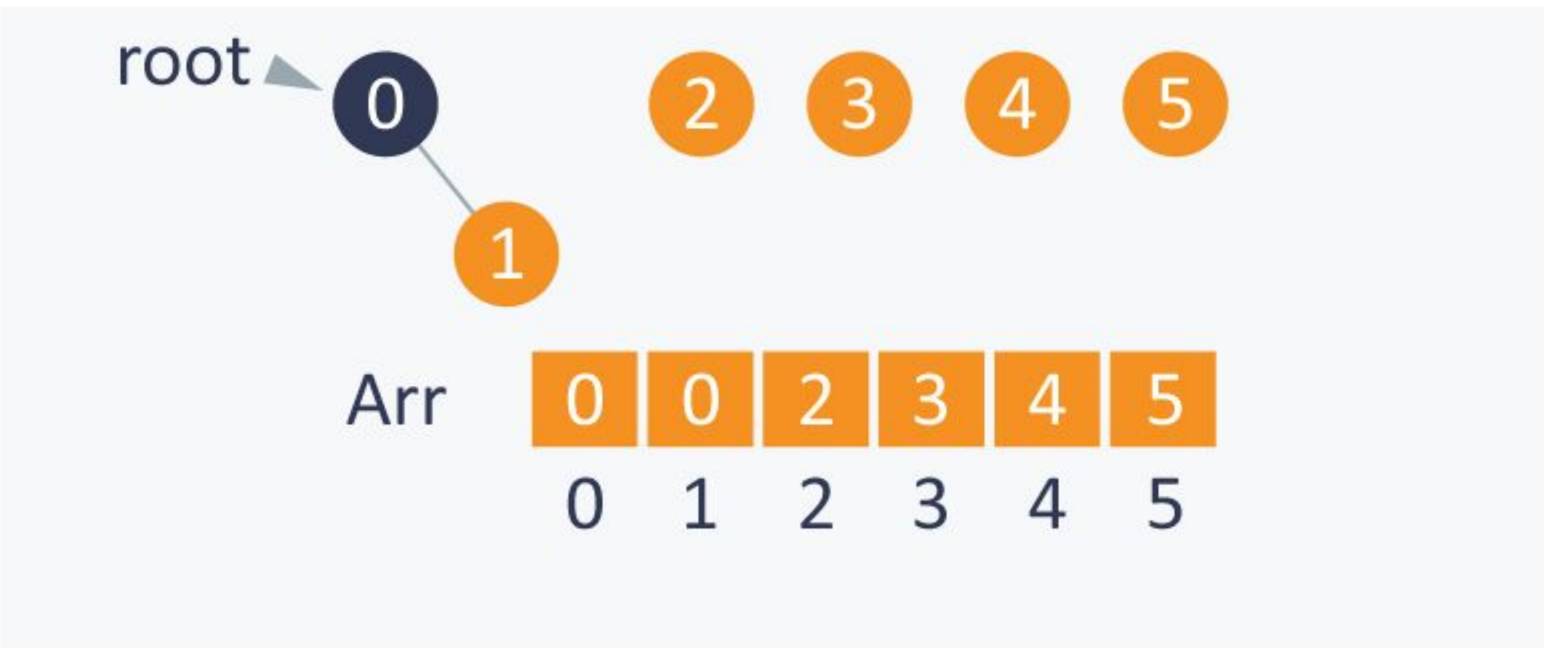
Disjoint sets

- Stores non-overlapping sets
- Union(a, b) = merges sets where a and b are located
- Find(a, b) = tells whether or not a and b are located in the same set

	Naive implementation
Find	O(1)
Union	O(N)
Union for all elements	O(N^2)

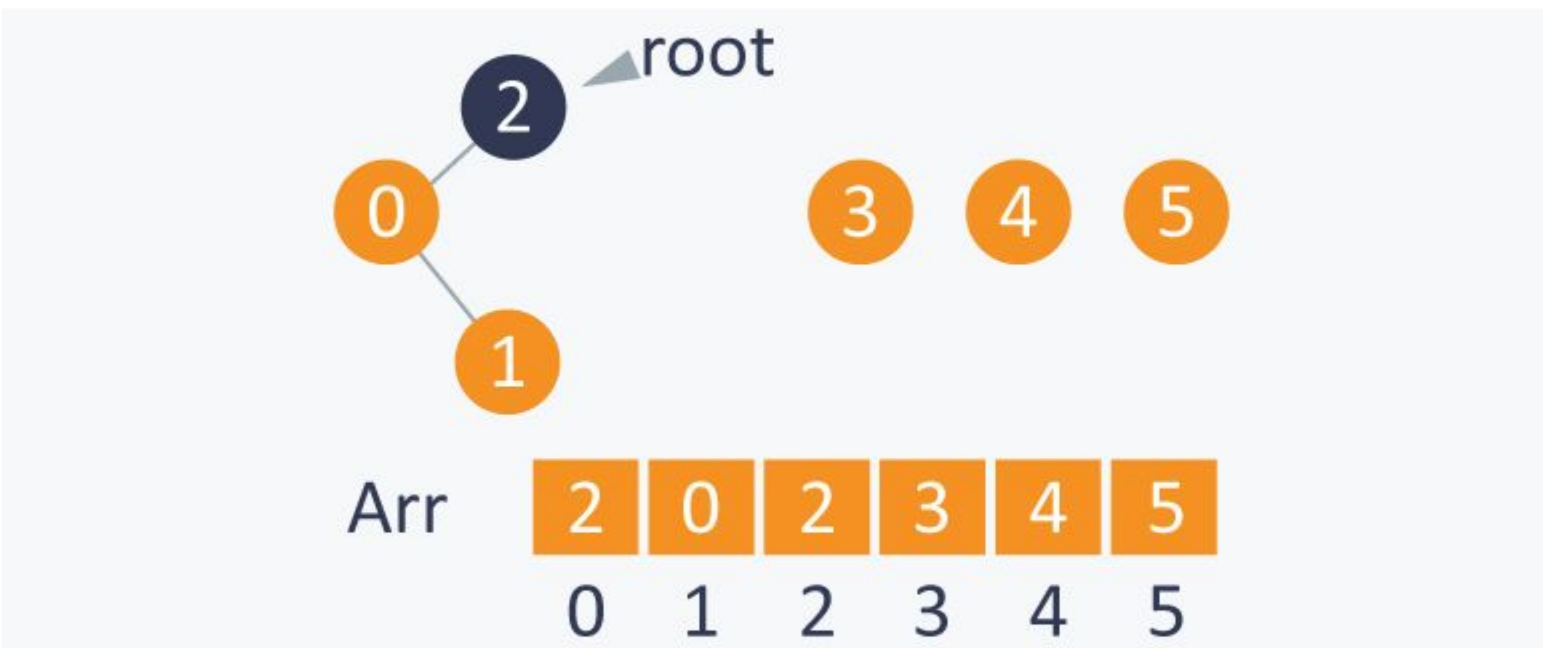
Optimized disjoint set

Optimized disjoint set - union(0, 1)



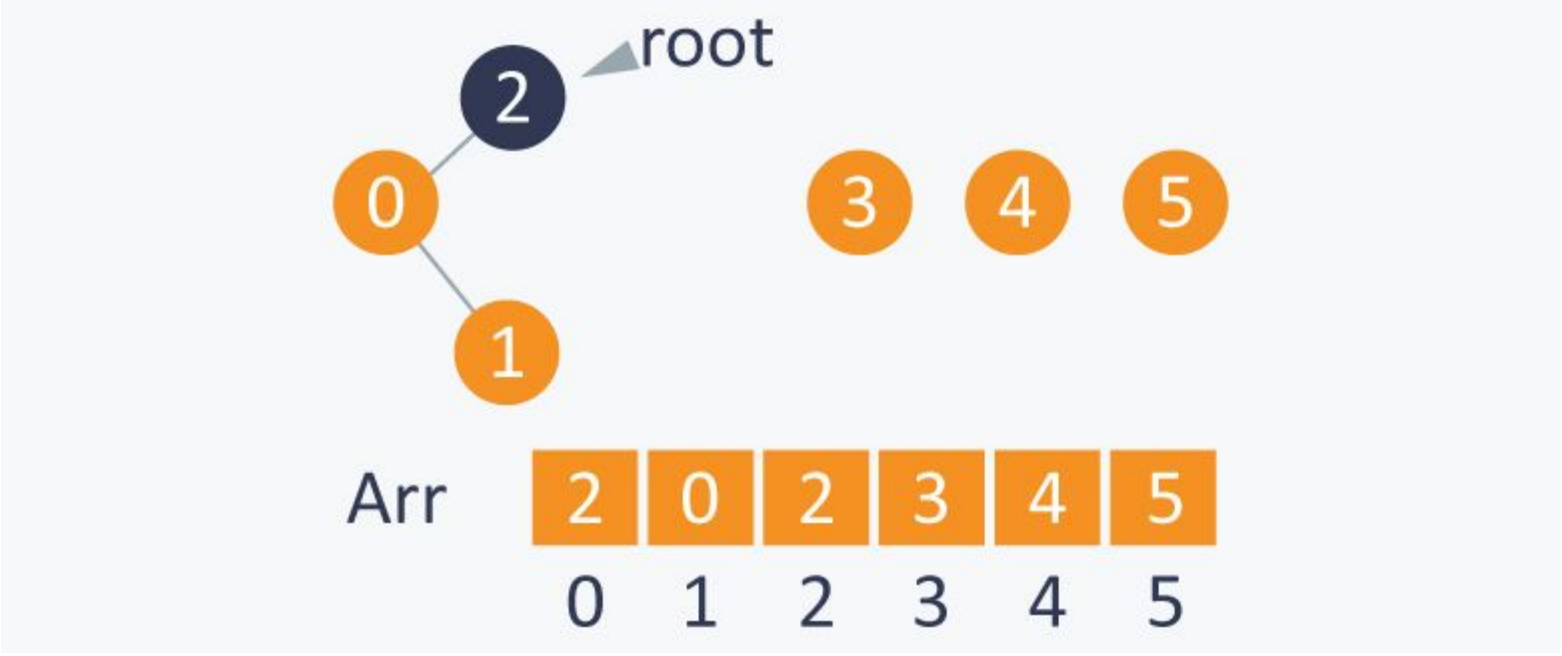
Optimized disjoint set - union(0, 2)

We only change the representative for the old root!

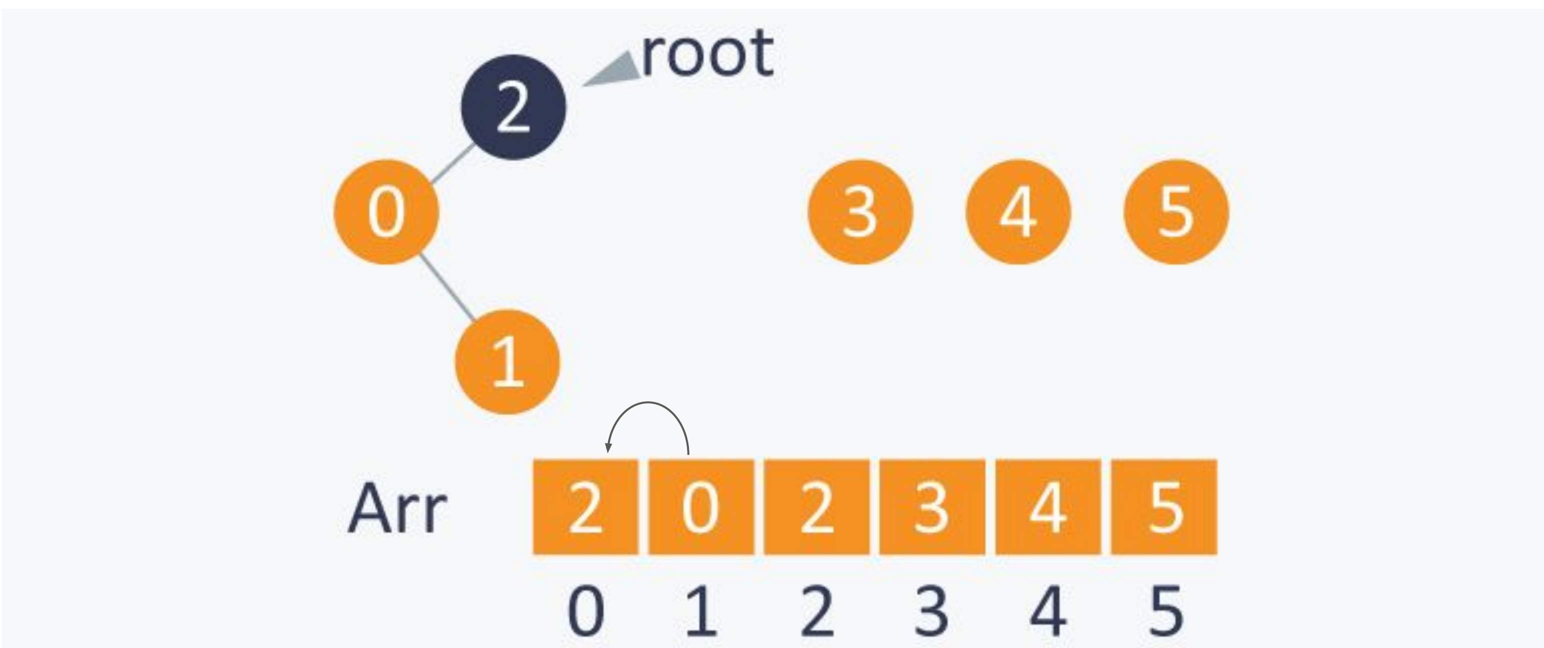


Optimized disjoint set - union(0, 2)

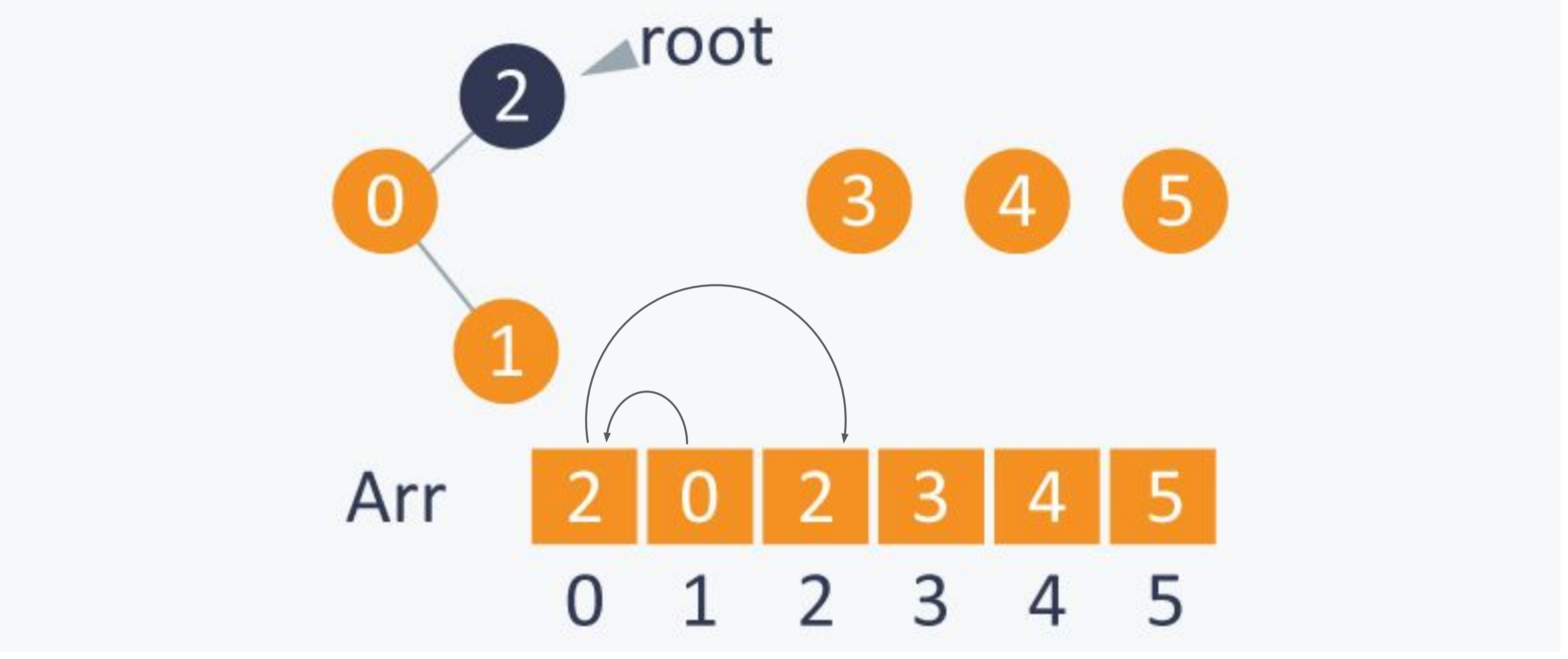
In the previous implementation,
Arr would be [2, 2, 2, 3 ,4 5]



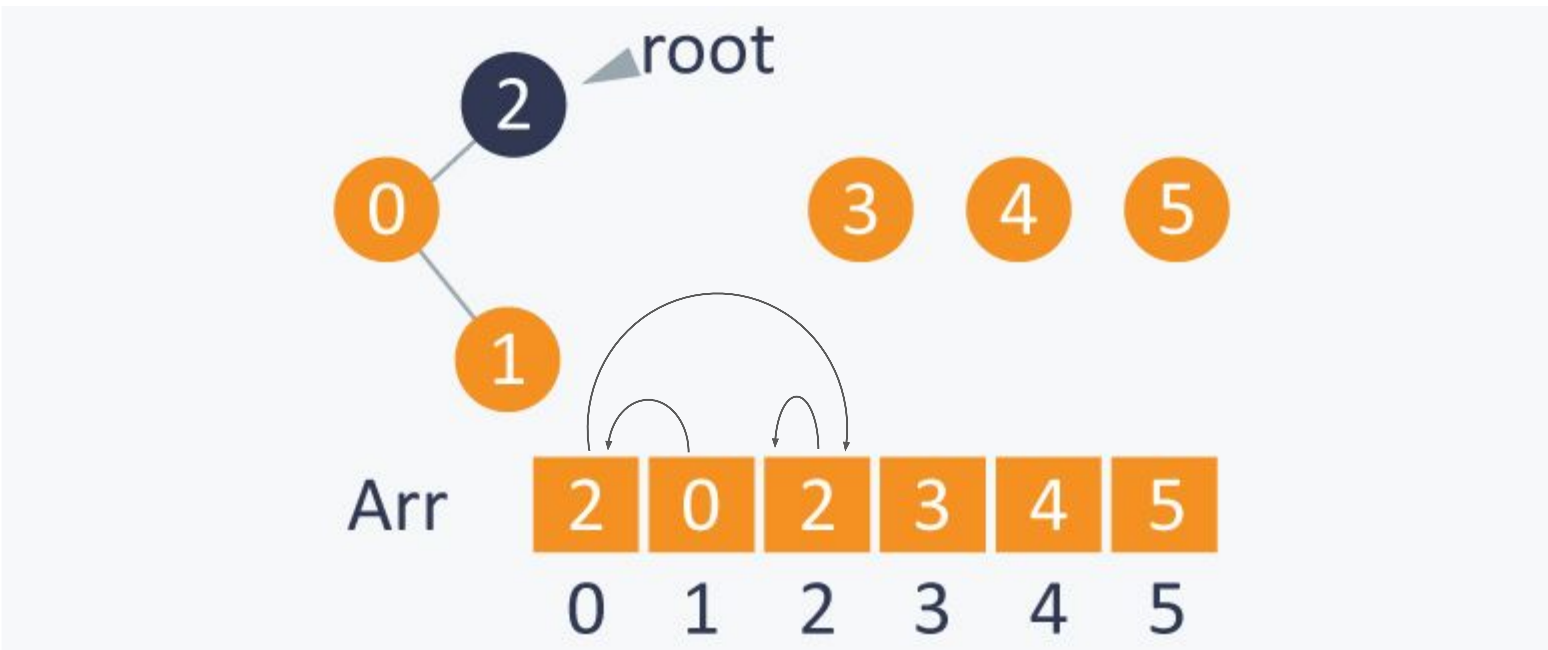
Optimized disjoint set - how to compute the representative of 1?



Optimized disjoint set - how to compute the representative of 1?

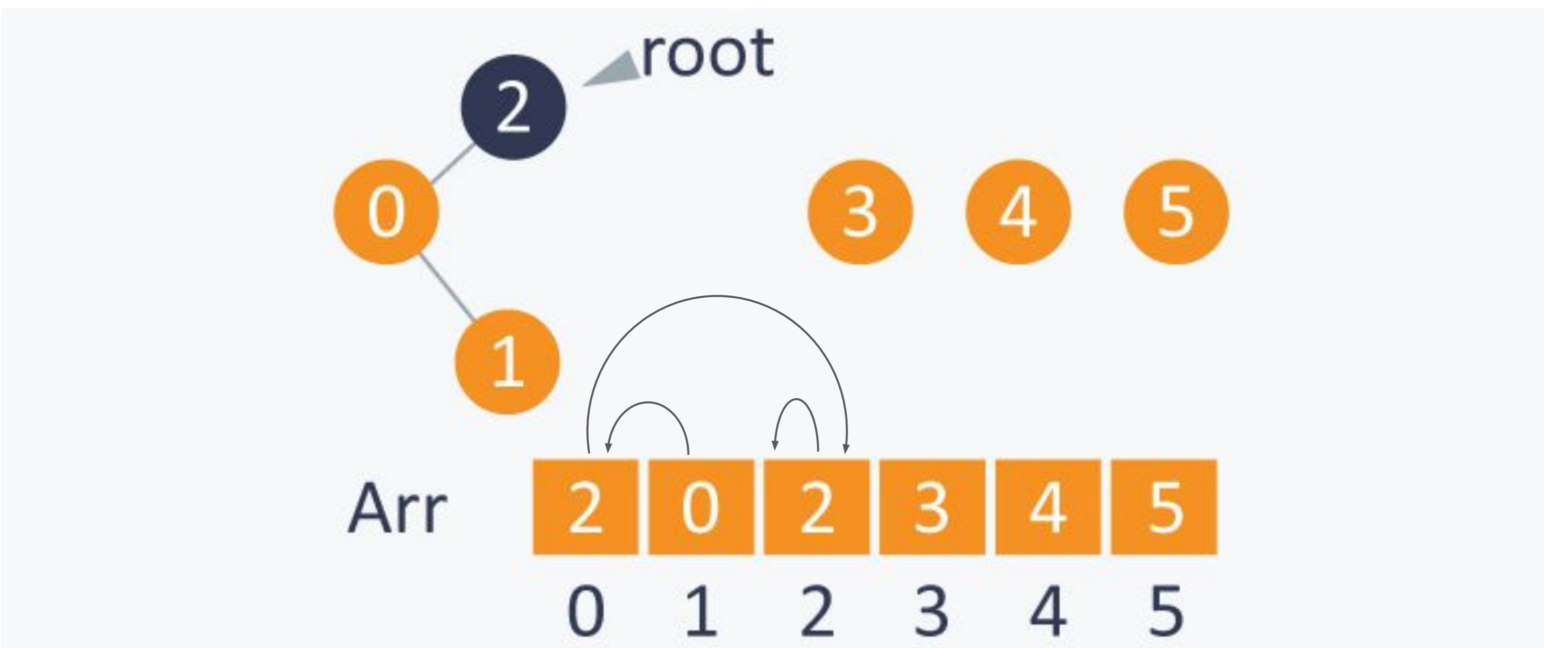


Optimized disjoint set - how to compute the representative of 1?

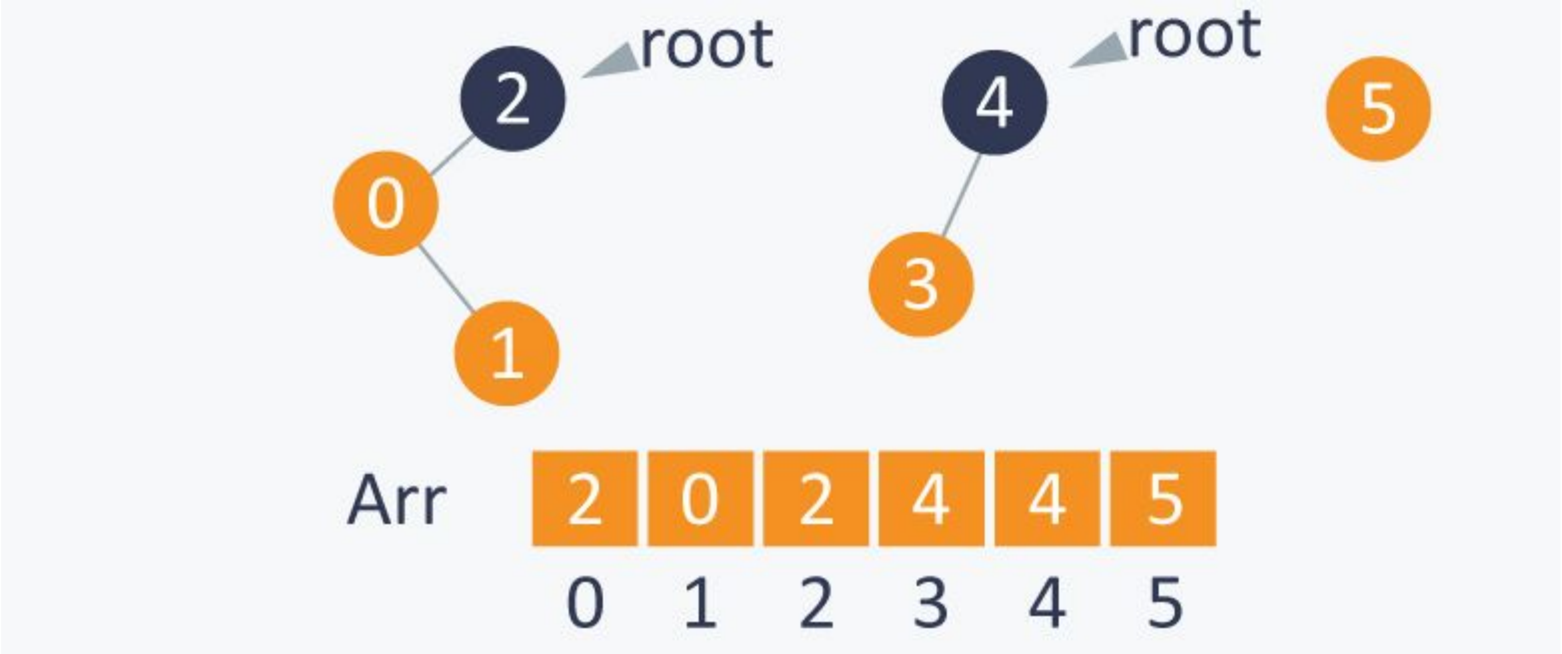


Optimized disjoint set - how to compute the representative of 1?

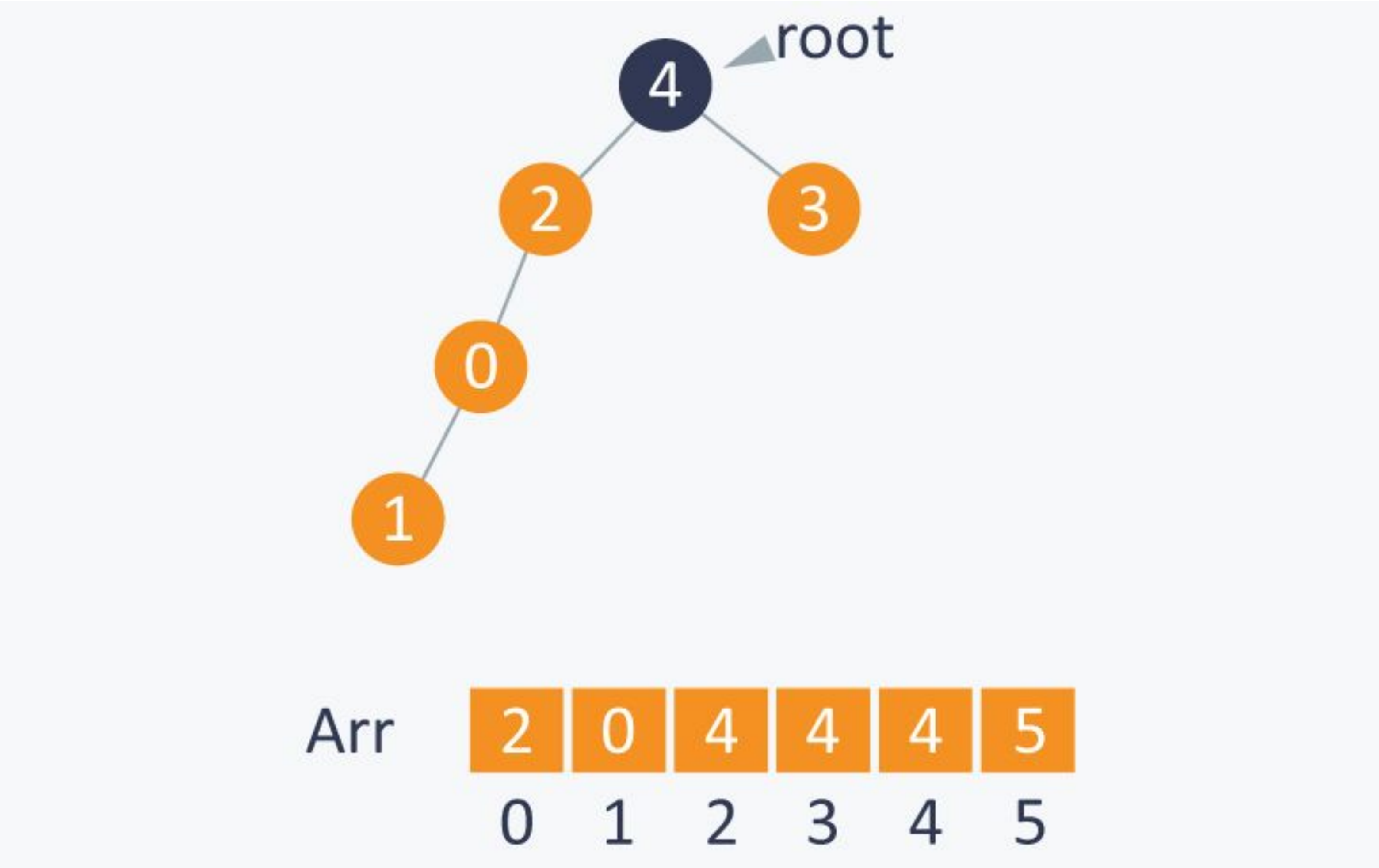
UNION becomes faster
FIND becomes longer



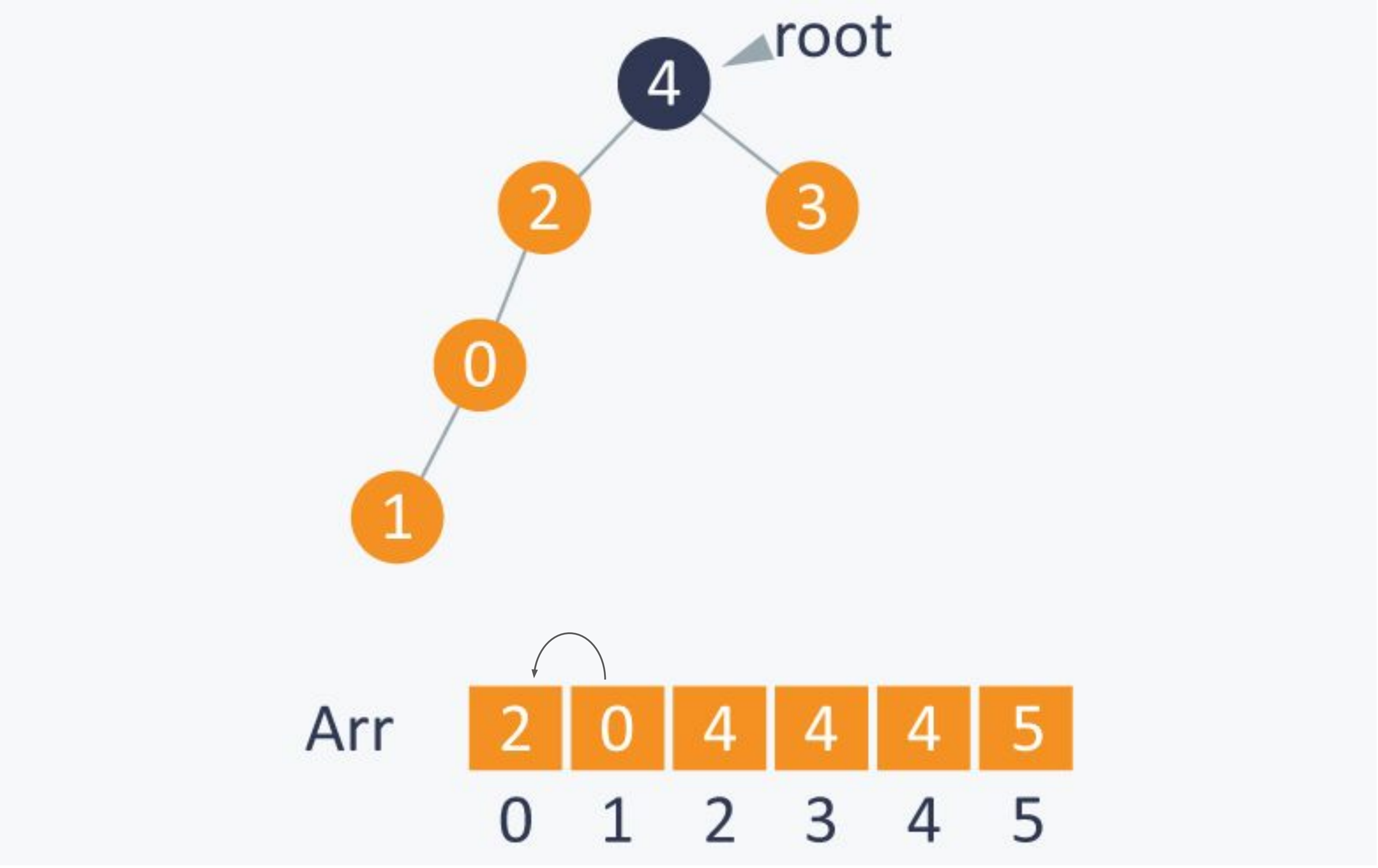
Optimized disjoint set - union(3, 4)



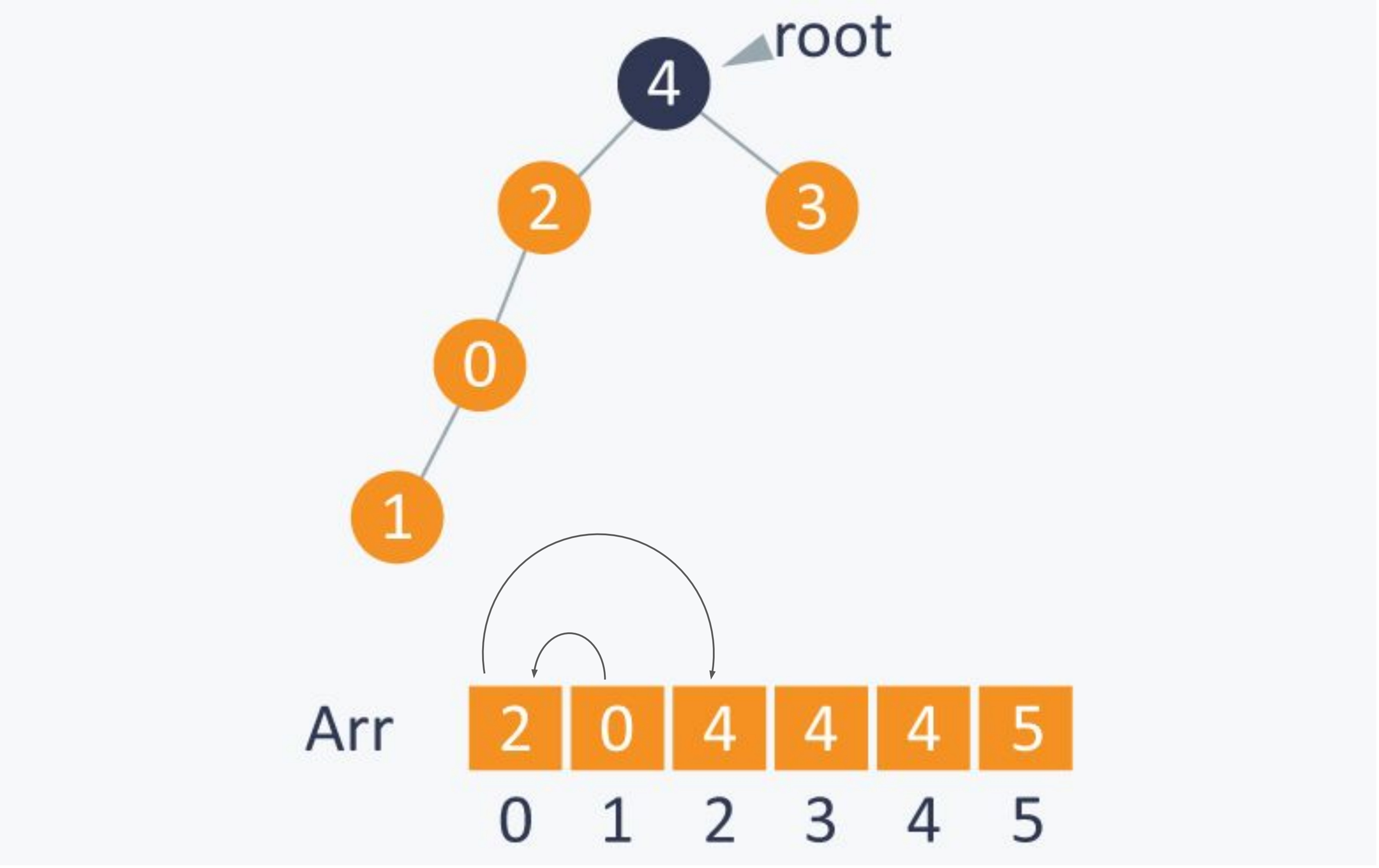
Optimized disjoint set - union(1, 4)



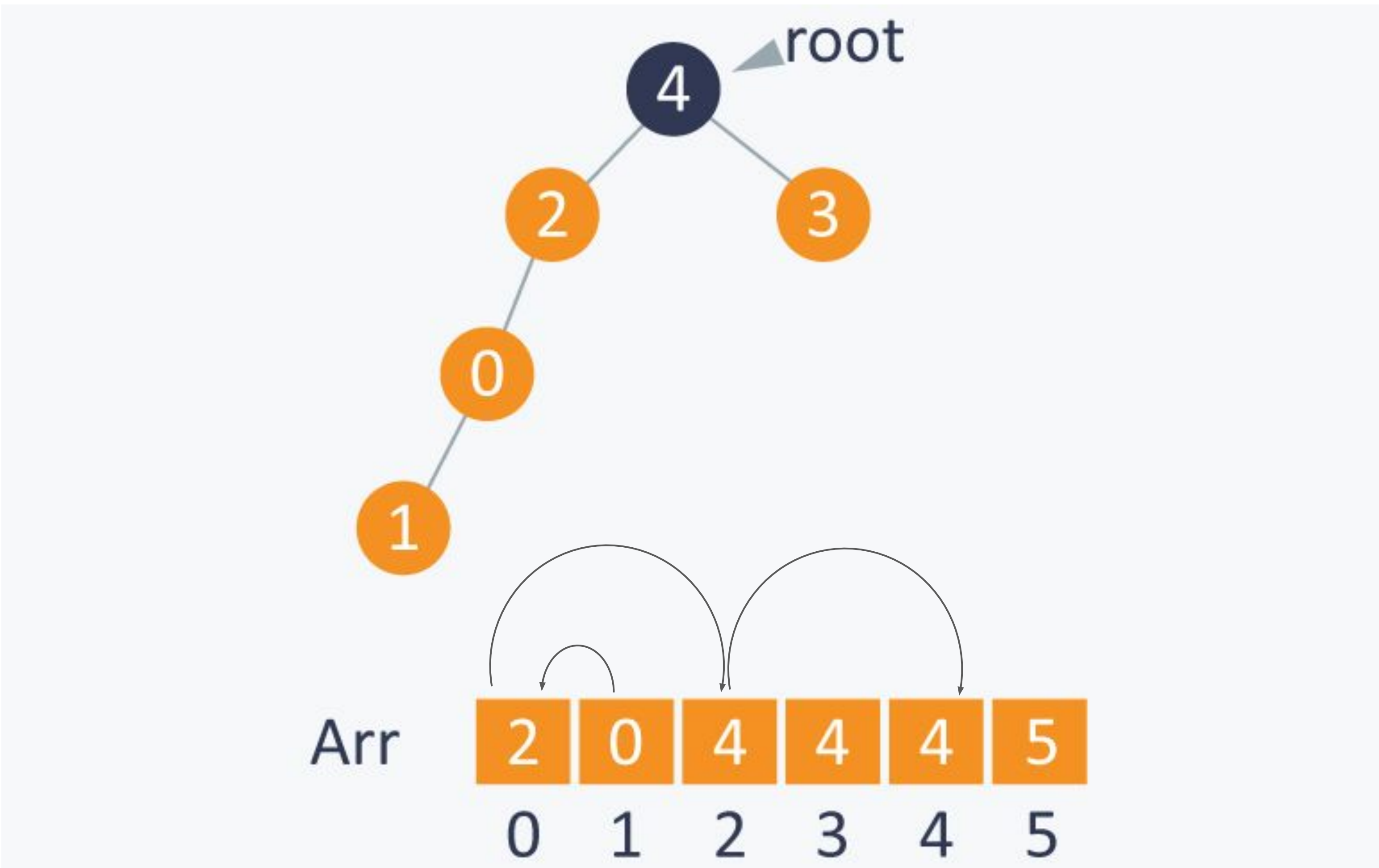
Optimized disjoint set - how to compute the representative of 1?



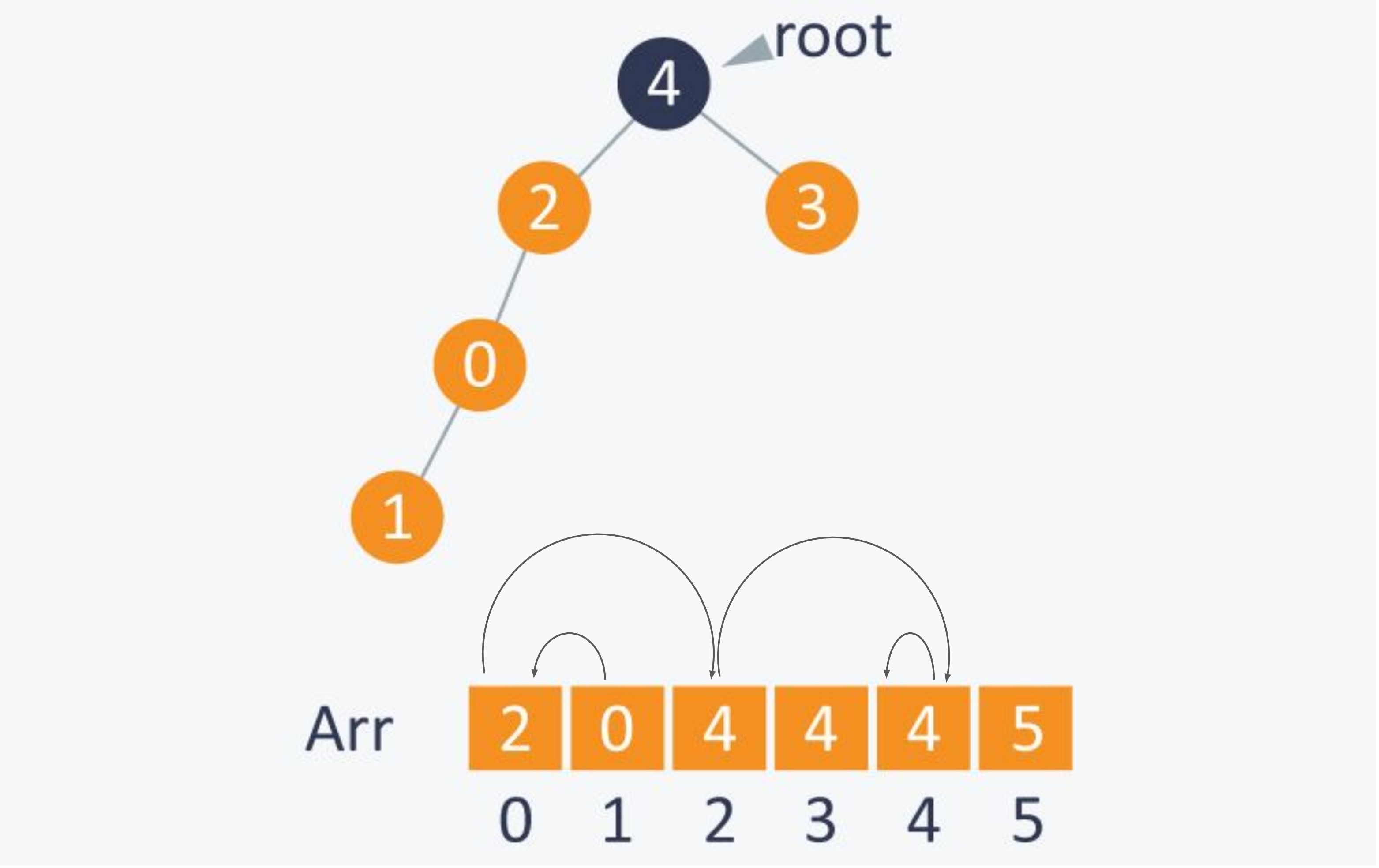
Optimized disjoint set - how to compute the representative of 1?



Optimized disjoint set - how to compute the representative of 1?



Optimized disjoint set - how to compute the representative of 1?



Optimized disjoint set - implementation

```
#finding root of an element
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i

def union(Arr, A, B):
    root_A = root(Arr, A)
    root_B = root(Arr, B)
    Arr[ root_A ] = root_B


def find(Arr, A, B):
    return root(Arr, A)==root(Arr, B)
```

Optimized disjoint set - only a half-way through!

A loop is hidden here

```
#finding root of an element
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i
```

We don’t have a loop here anymore

```
def union(Arr, A, B):
    root_A = root(Arr, A)
    root_B = root(Arr, B)
    Arr[ root_A ] = root_B
```

```
def find(Arr, A, B):
    return root(Arr, A)==root(Arr, B)
```