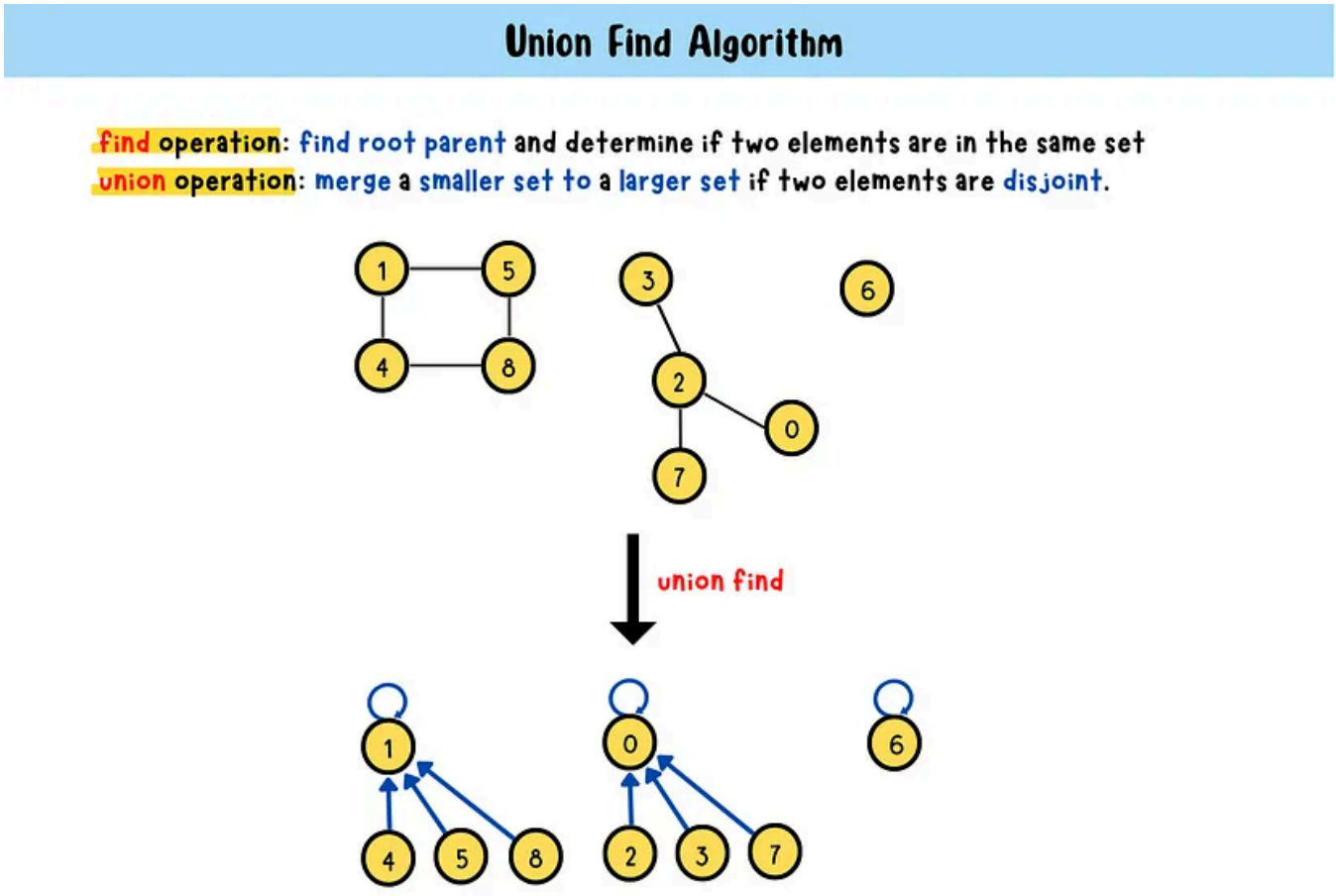
Union Find Algorithm

Claire Lee · Follow
4 min read · Oct 25, 2022

> *An algorithm that implements **find** and **union operations** on a **disjoint set data structure**. It **finds** the **root parent** of an element and determines whether if two elements are in the same set or not. If two elements are at different sets, **merge** the **smaller set to the larger set**. At the end, we can get the **connected components** in a graph.*



union find algorithm summary card

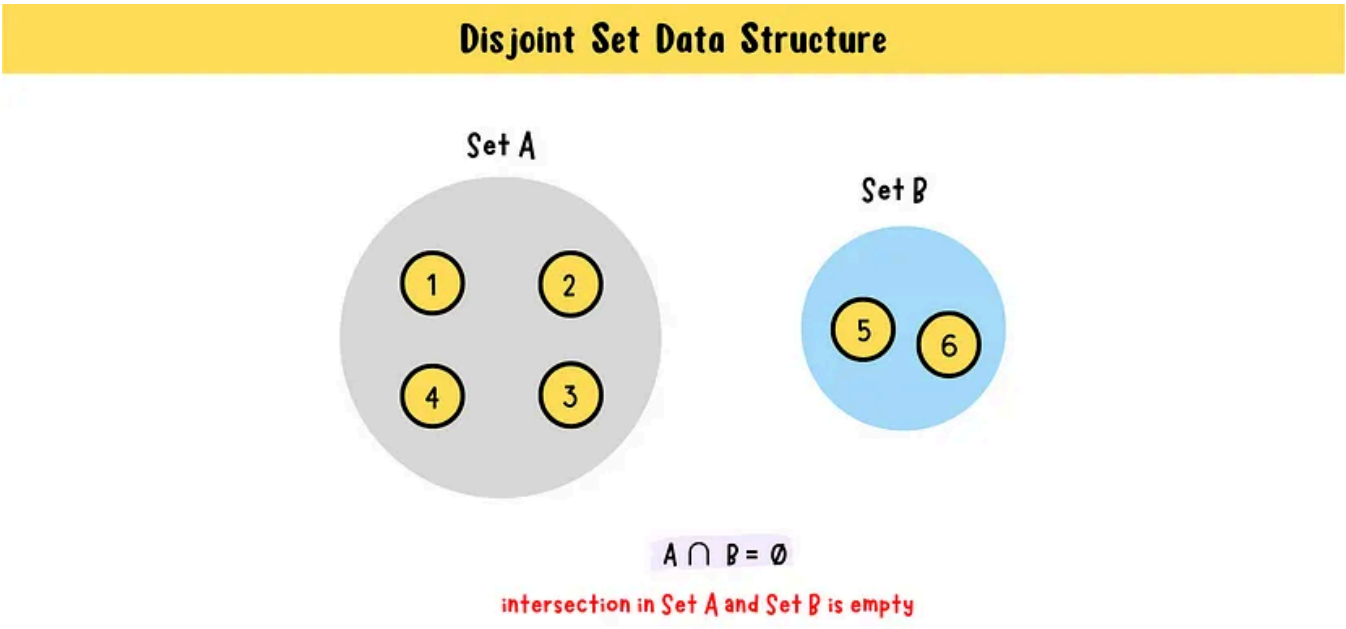
. . .

Table of Contents

. . .

Disjoint Set Data Structure

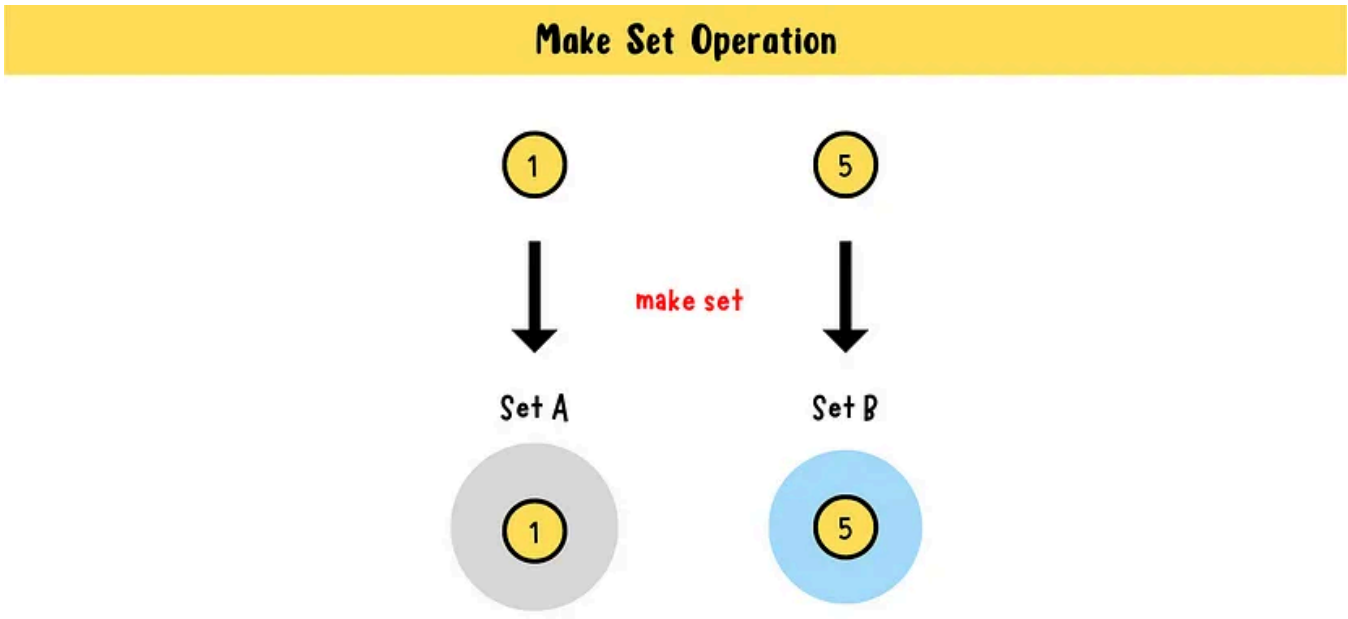
Two or more sets have **no element in common** are called **disjoint sets.** Disjoint set data structure is also referred to as **union find data structure** because of its union and find operations.



disjoint set

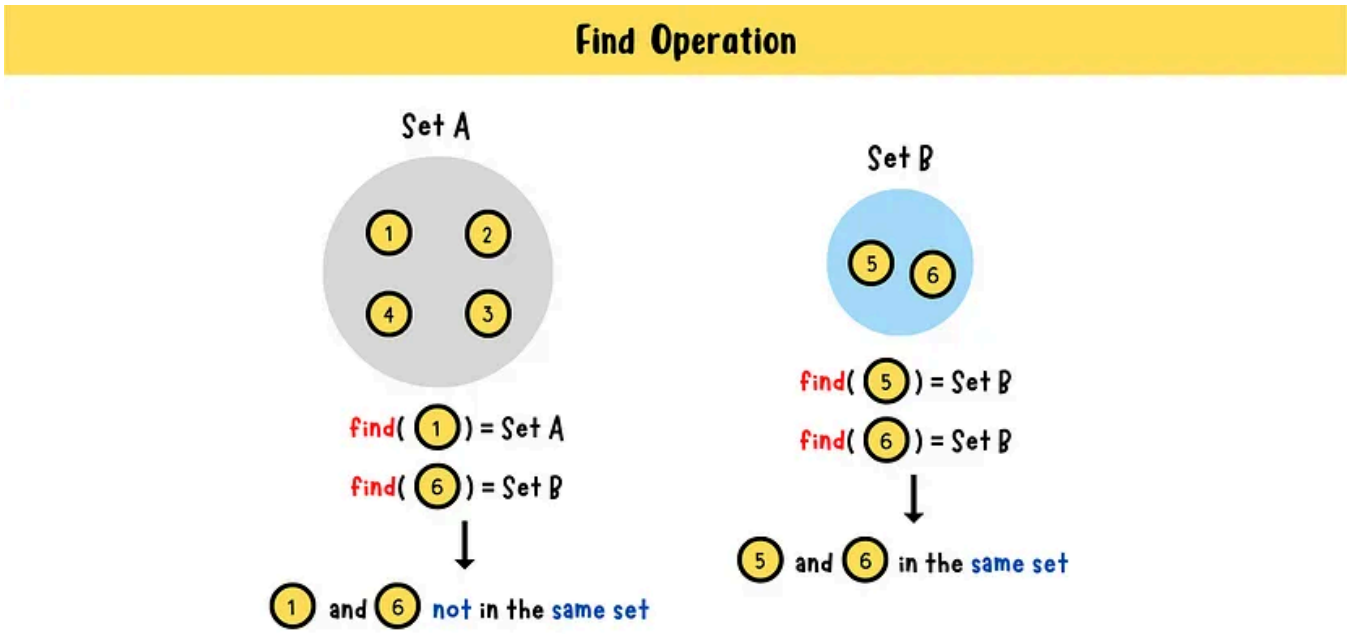
Disjoint set data structure supports three operations:

1. **Make Set:** create a new disjoint set contains only the given element.



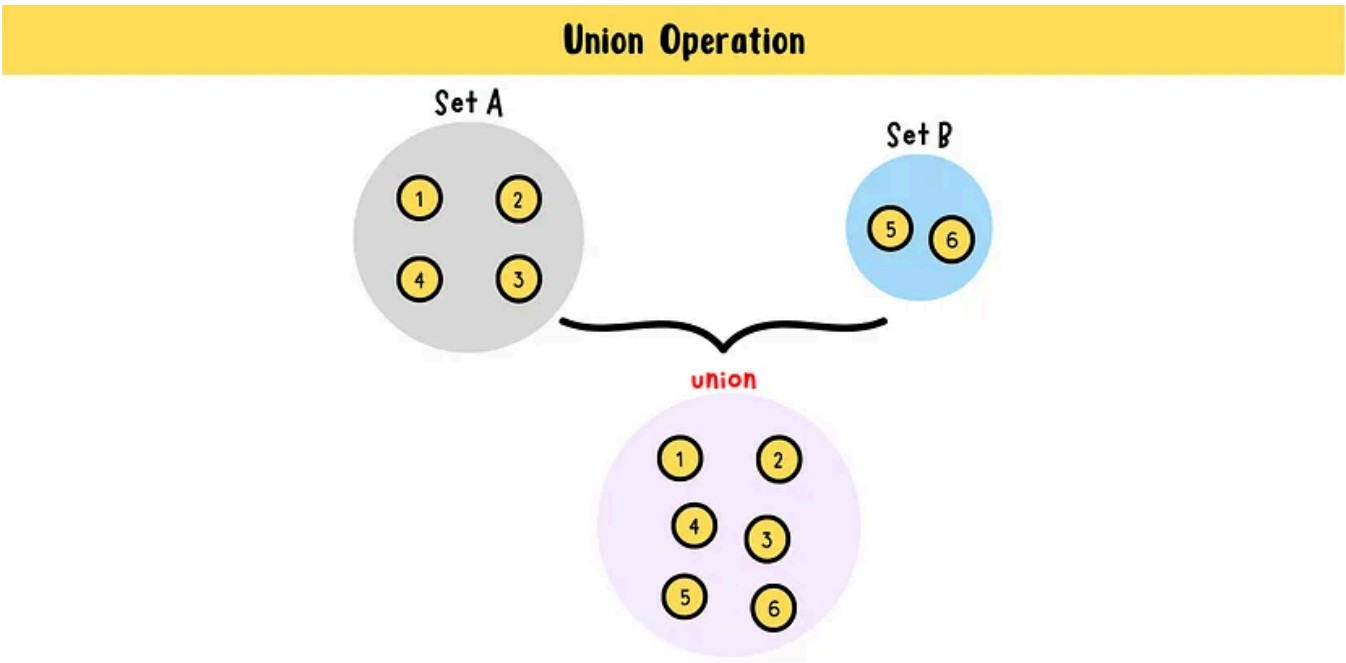
make set

2. Find: determine which subset a given element belongs to. It is used to decide whether if **two elements** are **disjoint or not.**



find

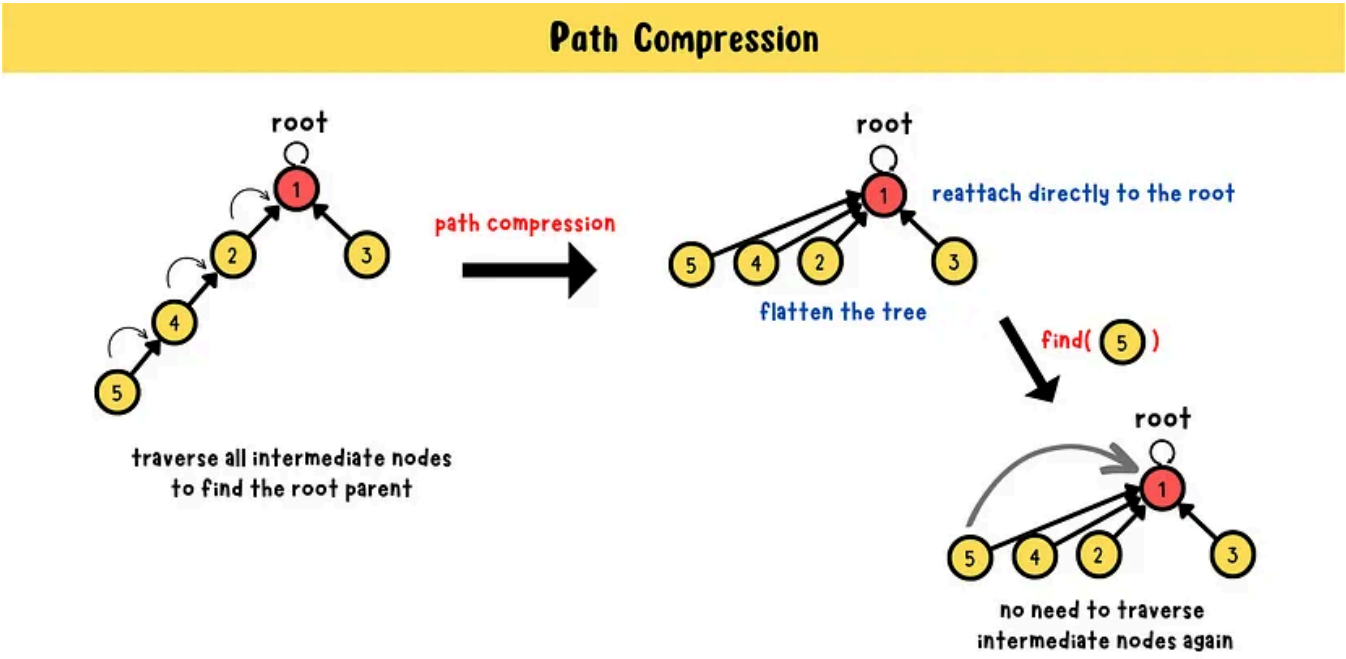
3. Union: merge two disjoint sets to a single disjoint set.



union

Path Compression

Path compression is a way to **flatten** the structure of the **tree** to make find operation more efficient. Find operation is used to find the **root parent** for a node. Without path compression, we have to travel upward the tree toward the root. The beauty of path compression is that we can find directly the root parent by **reattaching visited nodes to the root.**

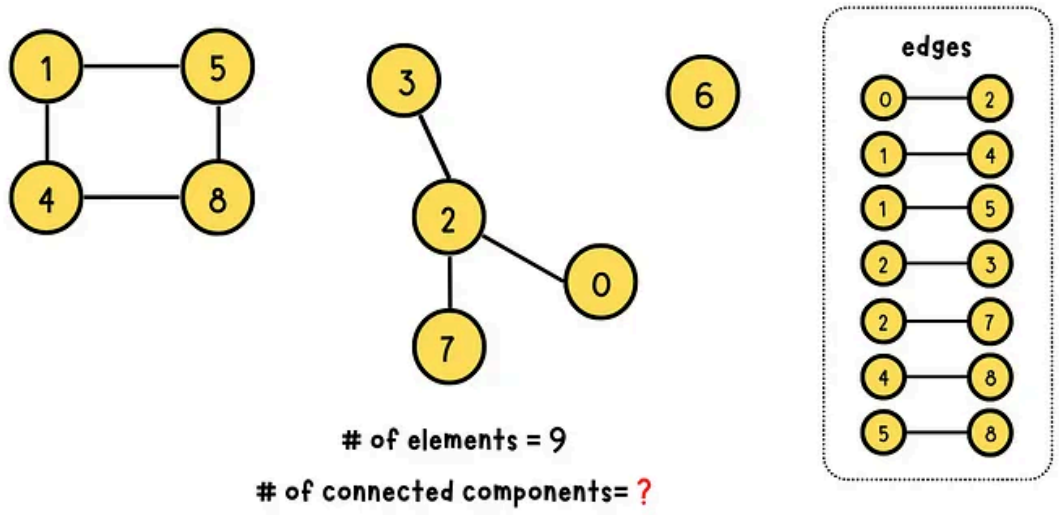


path compression

How Does Union Find Algorithm Work?

Union find algorithm performs the **find operation** to find a given element's root parent and determine whether if two elements are in the same subset or not. If the two elements are in the same subset, they are already connected. Otherwise, they belong to different sets. Implement the **union operation** to merge the two disjoint sets to a signal set.

Graphical Explanation



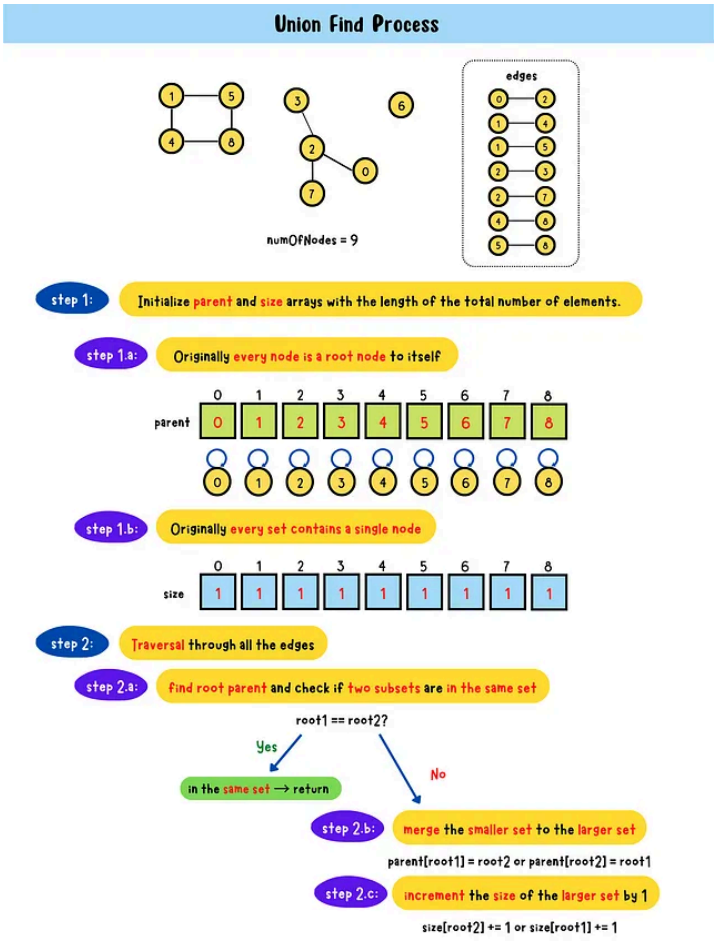
- *step1*: Initialize **parent** and **size** arrays with the length of the total number of elements.

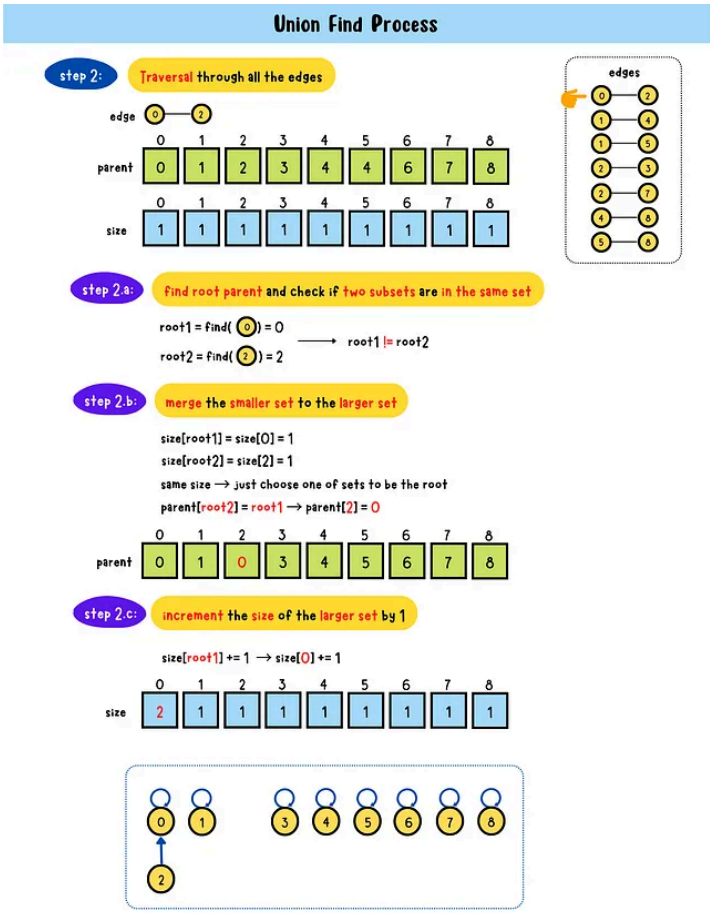
```
parent: store a node's root parent
size: store the total number of elements in a subset

step1.a: originally every node is a root node to itself
         parent[i] = i
step1.b: originally every set contains a single node
         size[i] = 1
```

- *step2*: **Traversal** through **all edges**

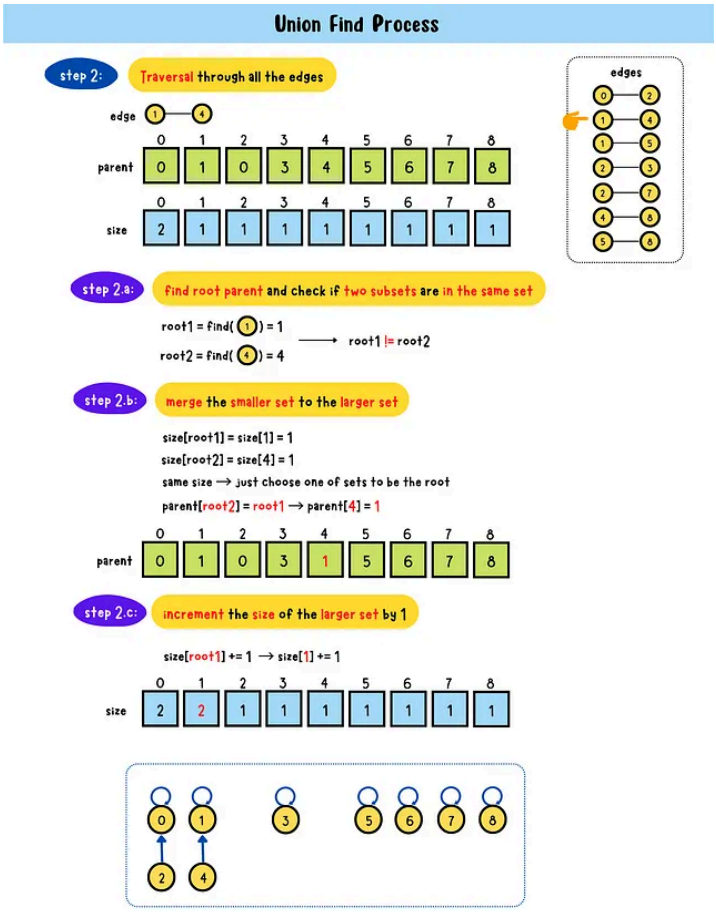
```
step2.a: find root parent and check if two subsets are in the same
set

root1 == root2?
- Yes → in the same set → return
- No → step2.b

step2.b: merge the smaller set to the larger set
parent[root1] = root2 or parent[root2] = root1
```

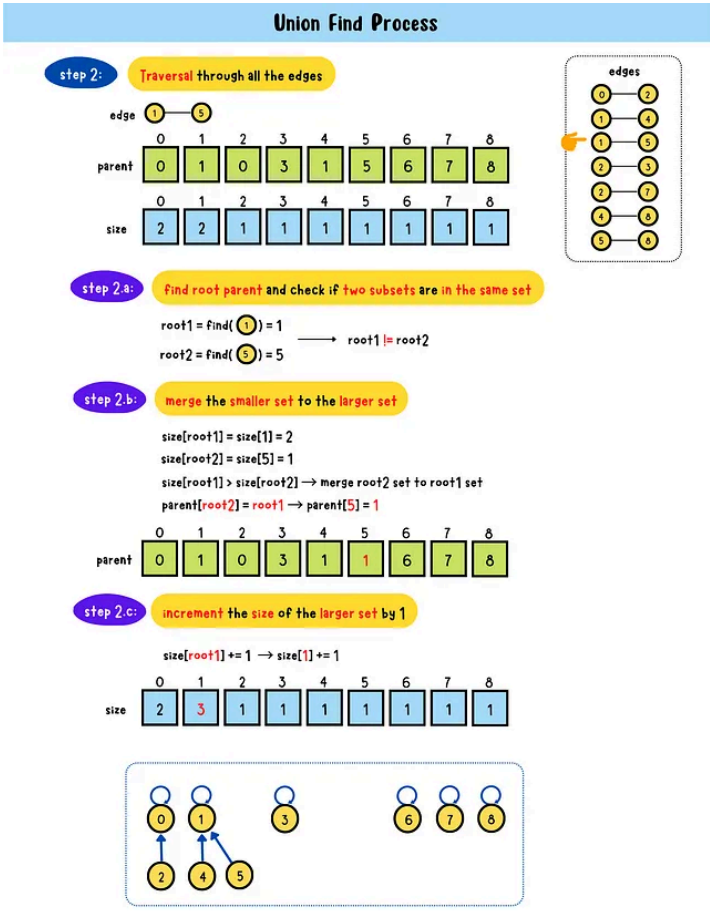
```
step2.c: increment the size of the larger set by 1
size[root2] += 1 or size[root1] += 1
```



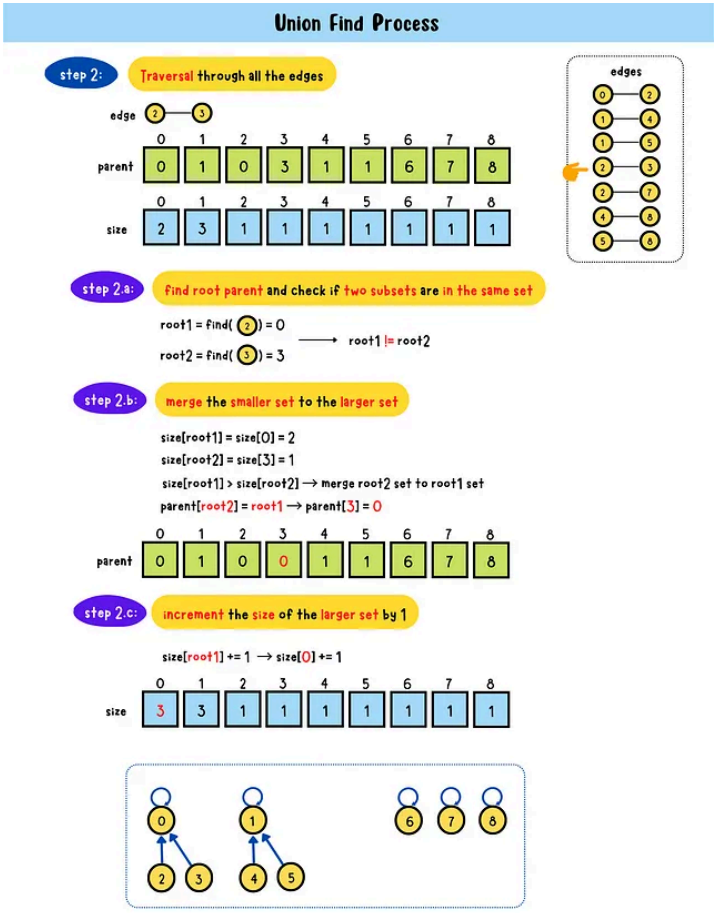


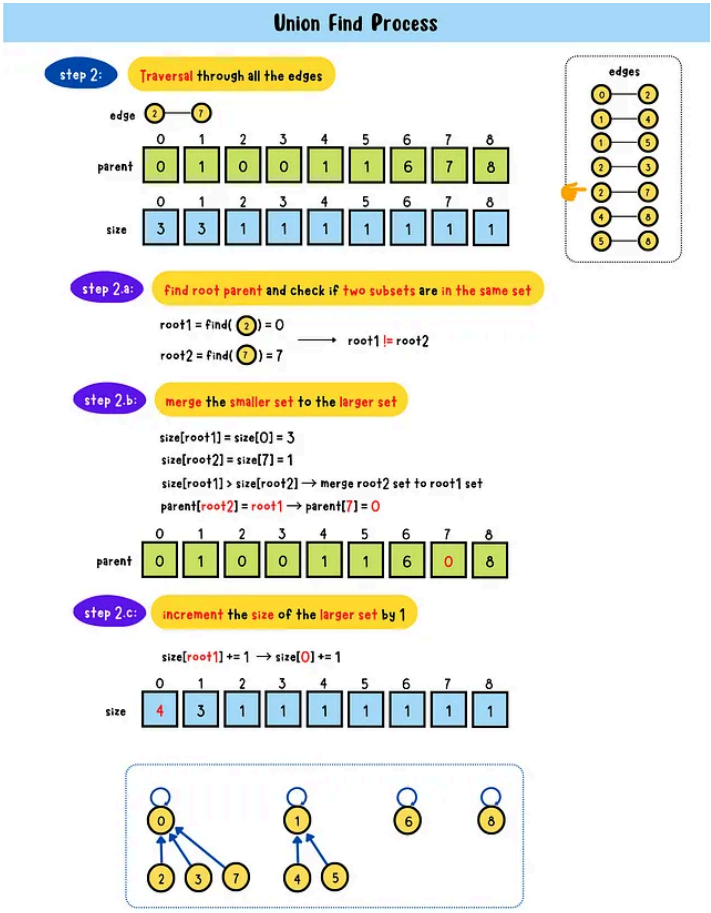
1, 2


Union Find Process
step 2: Traversal through all the edges
edge (1) (4)
parent 0 1 0 3 4 5 6 7 8
size 2 1 1 1 1 1 1 1 1
edges
step 2.a: find root parent and check if two subsets are in the same set
root1 = find(1) = 1
root2 = find(4) = 4
root1 != root2
step 2.b: merge the smaller set to the larger set
size[root1] = size[1] = 1
size[root2] = size[4] = 1
same size → just choose one of sets to be the root
parent[root2] = root1 → parent[4] = 1
parent 0 1 0 3 1 5 6 7 8
step 2.c: increment the size of the larger set by 1
size[root1] += 1 → size[1] += 1
size 2 2 1 1 1 1 1 1 1

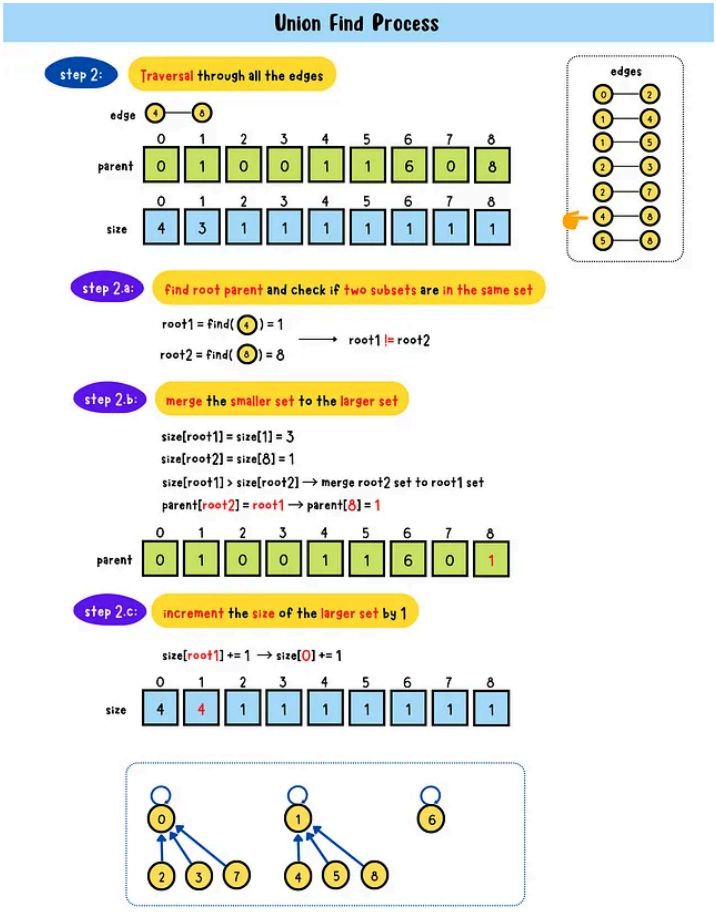

Union Find Process
step 2: Traversal through all the edges
edge (1) (5)
parent 0 1 0 3 1 5 6 7 8
size 2 2 1 1 1 1 1 1 1
edges
step 2.a: find root parent and check if two subsets are in the same set
root1 = find(1) = 1
root2 = find(5) = 5
root1 != root2
step 2.b: merge the smaller set to the larger set
size[root1] = size[1] = 2
size[root2] = size[5] = 1
size[root1] > size[root2] → merge root2 set to root1 set
parent[root2] = root1 → parent[5] = 1
parent 0 1 0 3 1 1 6 7 8
step 2.c: increment the size of the larger set by 1
size[root1] += 1 → size[1] += 1
size 2 3 1 1 1 1 1 1 1

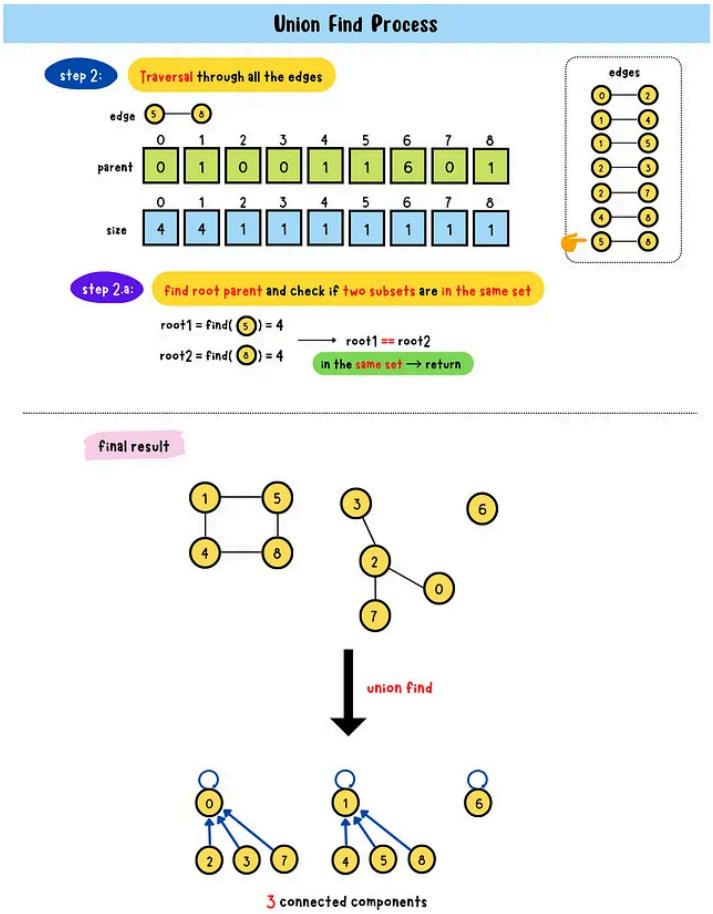
3, 4


Union Find Process
step 2: Traversal through all the edges
edge (2) (3)
parent 0 1 0 3 1 1 6 7 8
size 2 3 1 1 1 1 1 1 1
edges
step 2.a: find root parent and check if two subsets are in the same set
root1 = find(2) = 0
root2 = find(3) = 3
root1 != root2
step 2.b: merge the smaller set to the larger set
size[root1] = size[0] = 2
size[root2] = size[3] = 1
size[root1] > size[root2] → merge root2 set to root1 set
parent[root2] = root1 → parent[3] = 0
parent 0 1 0 0 1 1 6 7 8
step 2.c: increment the size of the larger set by 1
size[root1] += 1 → size[0] += 1
size 3 3 1 1 1 1 1 1 1


Union Find Process
step 2: Traversal through all the edges
edge (2) (7)
parent 0 1 0 0 1 1 6 7 8
size 3 3 1 1 1 1 1 1 1
edges
step 2.a: find root parent and check if two subsets are in the same set
root1 = find(2) = 0
root2 = find(7) = 7
root1 != root2
step 2.b: merge the smaller set to the larger set
size[root1] = size[0] = 3
size[root2] = size[7] = 1
size[root1] > size[root2] → merge root2 set to root1 set
parent[root2] = root1 → parent[7] = 0
parent 0 1 0 0 1 1 6 0 8
step 2.c: increment the size of the larger set by 1
size[root1] += 1 → size[0] += 1
size 4 3 1 1 1 1 1 1 1

5, 6





7, 8

Code Implementation

Complexity

Time: O(nlog(n))

Space: O(n) parent and size array

n: the total number of nodes in the given graph

- **Find operation**
  Time: O(log(n)) log(n) is the height of the tree
  Space: O(1)

- **Union operation**
  Time: O(1)
  Space: O(1)

Golang