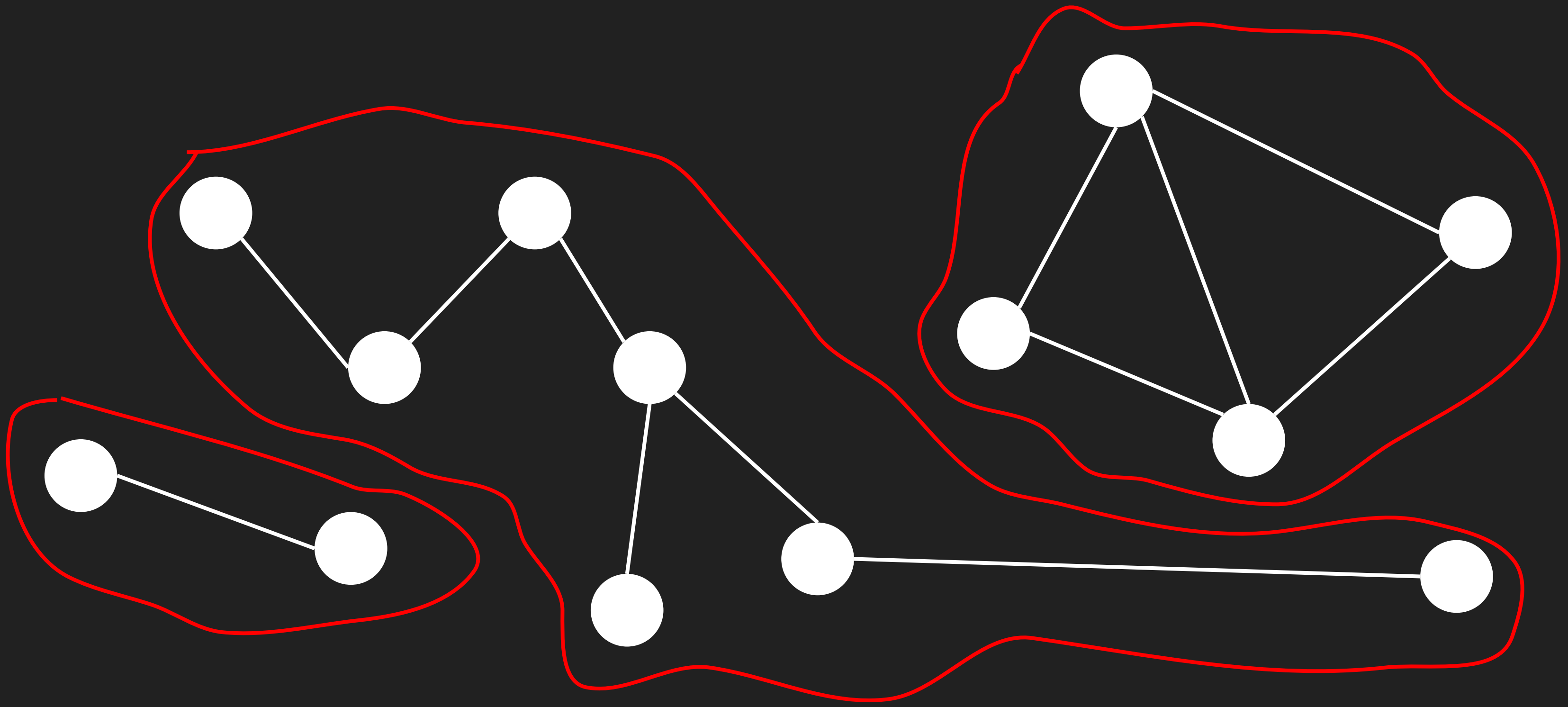
Union-Find

Quick Review: Connected Components

- Essentially answers the question “which nodes are reachable from here”?



Connected Components

- Not always obvious whether two nodes are in the same connected component
    - Not always obvious what the connected components even are!
    - Depends a lot on graph representation

Union-Find

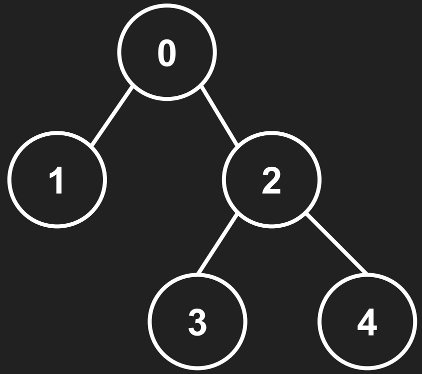
- Given a set of elements *S*, a partition of *S* is a set of nonempty subsets of *S* such that every element of *S* is in exactly one of the subsets
    - If your elements are nodes in a graph, the partitions can correspond to connected components
    - But can be used for other things!

- A Union-Find (or Disjoint-Set) data structure is one that efficiently keeps track of these partitions

- The Union-Find data structure supports two operations:
    - Union
        - Merge two sets of the partition into one
    - Find
        - Identify which partition a given input is a part of

Representing Trees as Arrays

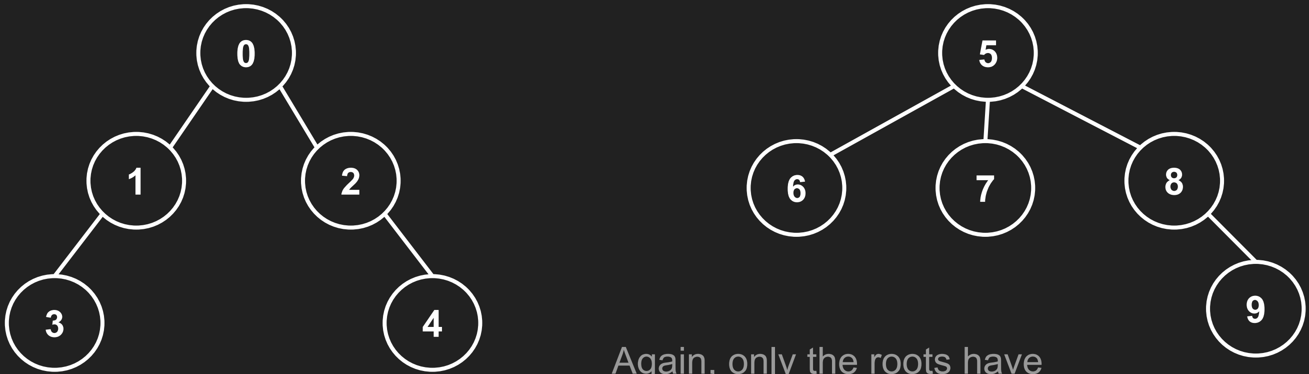
- Key fact: by definition, every tree node has exactly one parent (except the root)
- Big idea: assign each node to an array index, and store the index of the parent



Notice: only the root node has its value equal to its index

0	0	0	2	2

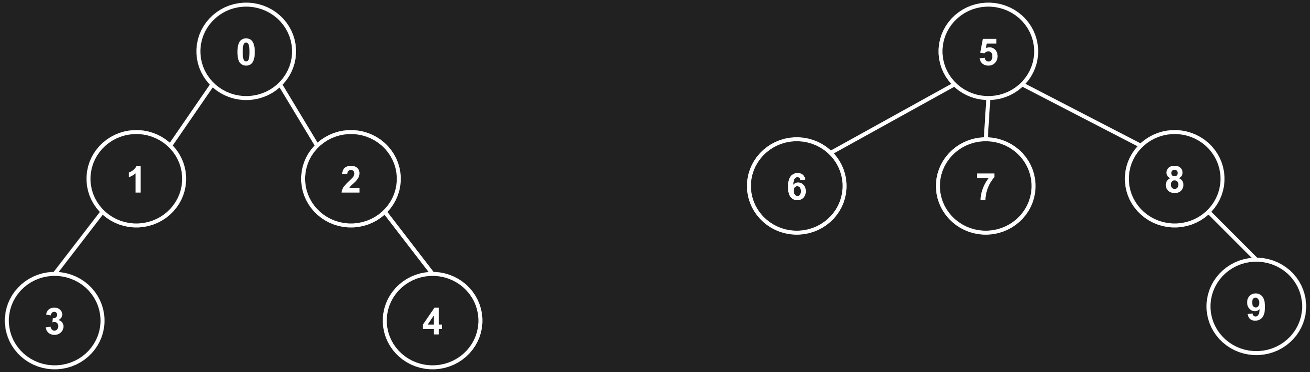
Representing Forests as Arrays



Again, only the roots have values equal to their array indices

0	0	0	1	2	5	5	5	5	8

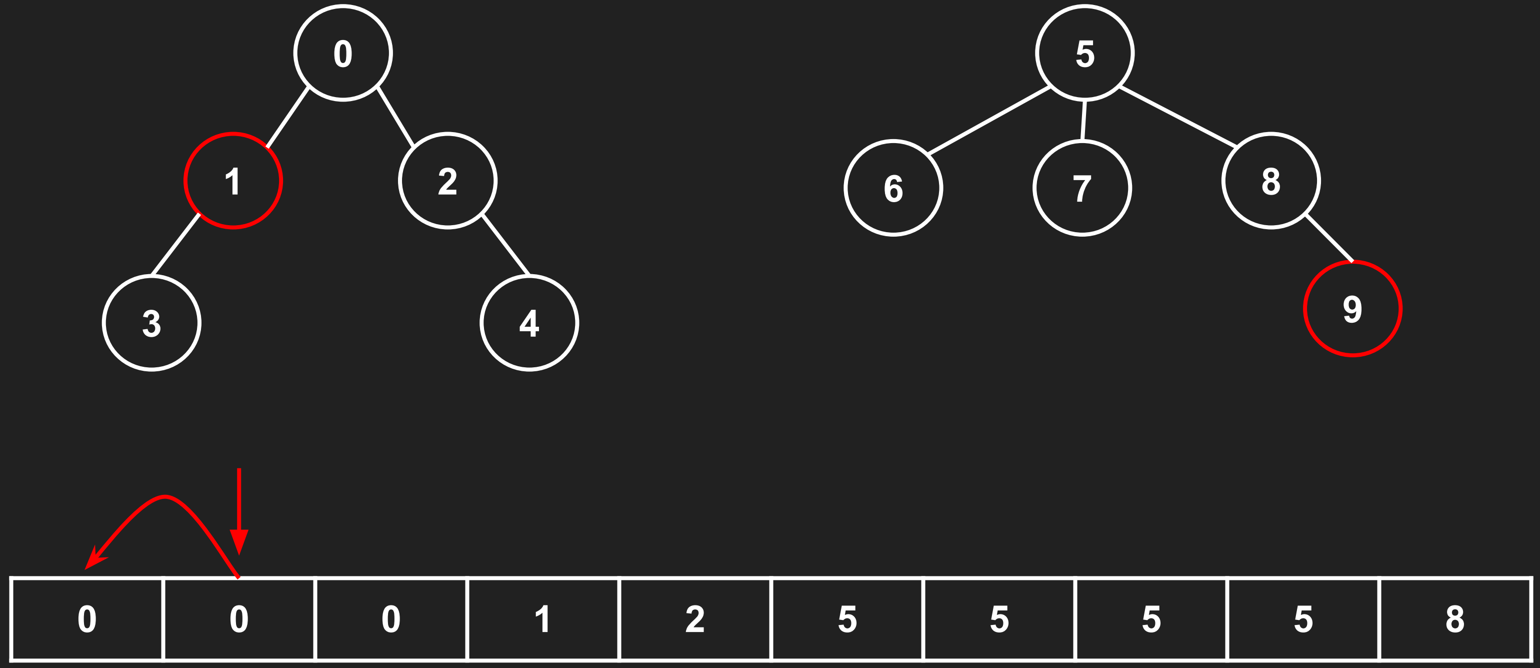
Representing Forests as Arrays



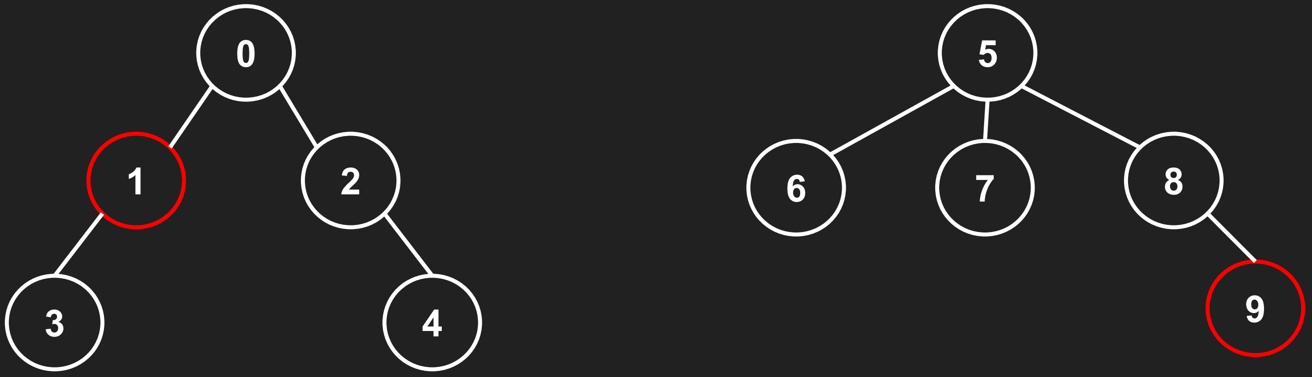
- How does this help with Union-Find?

0	0	0	1	2	5	5	5	5	8

Representing Forests as Arrays

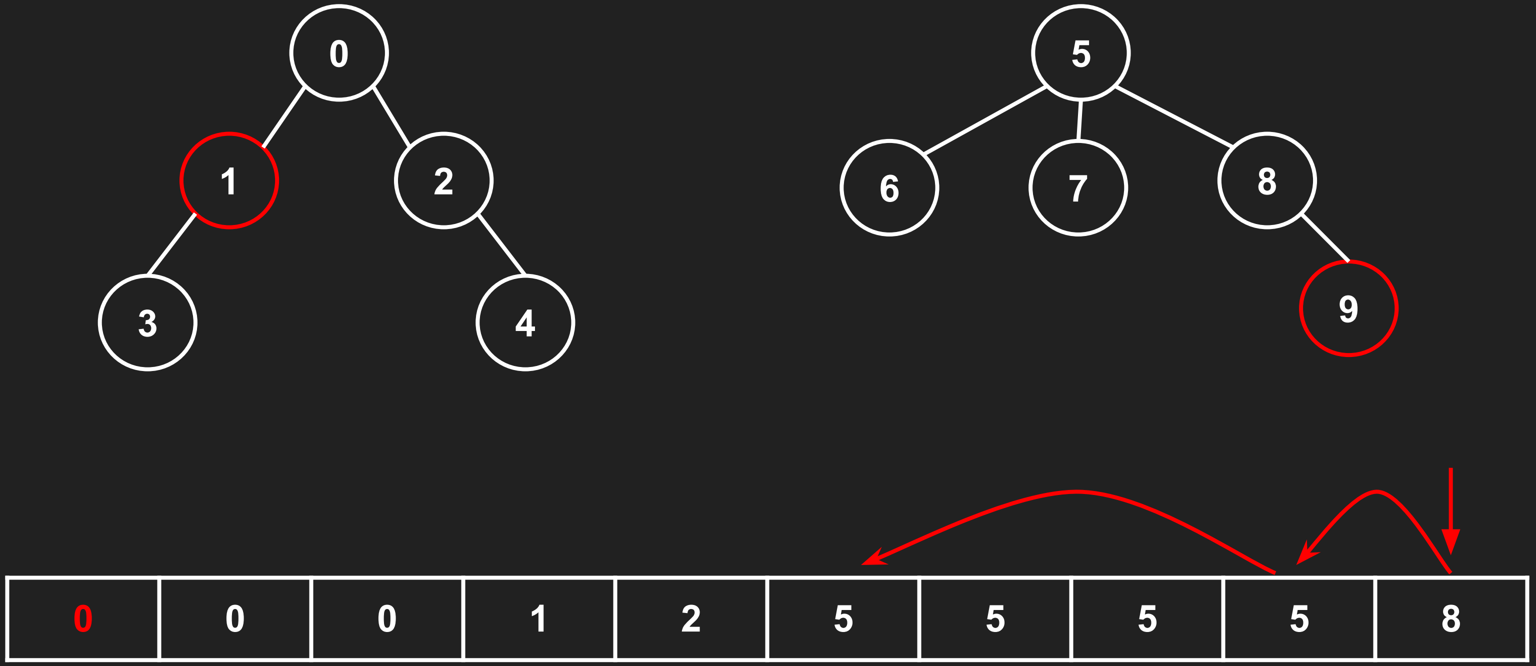


Representing Forests as Arrays

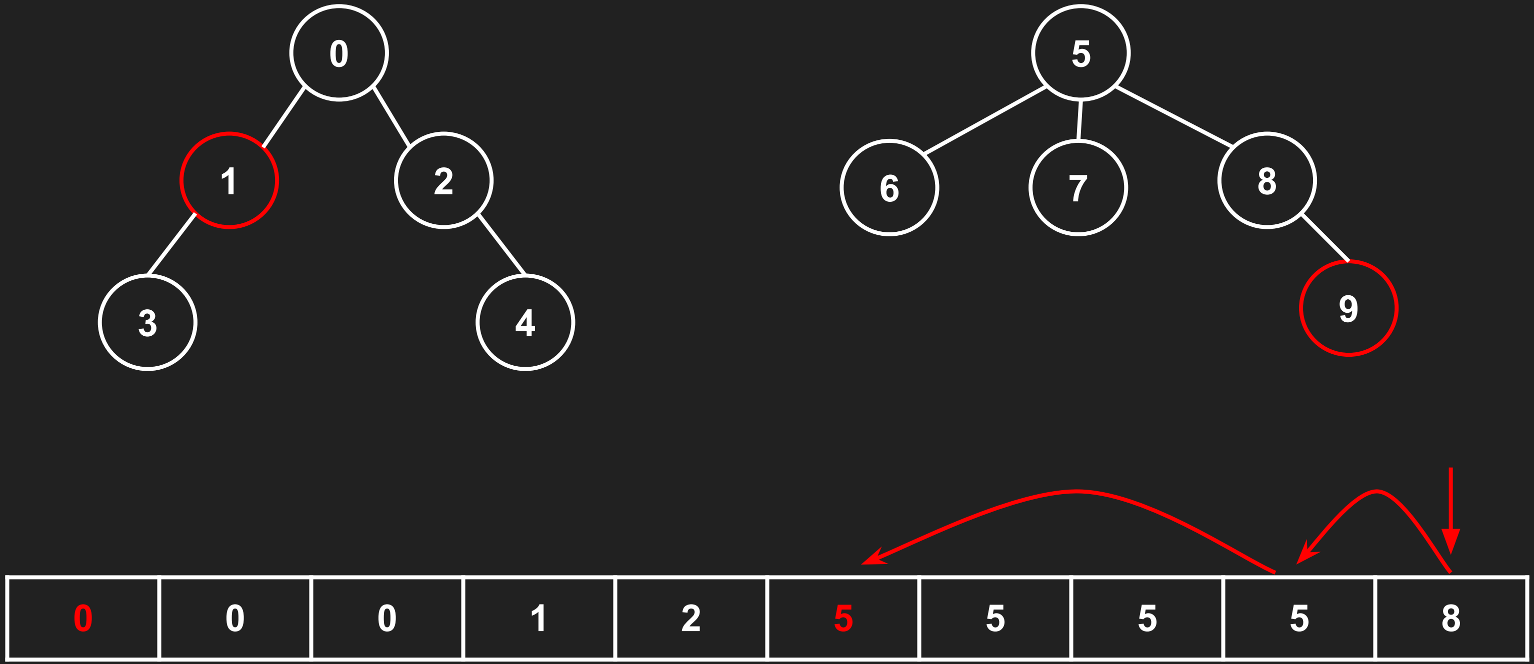


0	0	0	1	2	5	5	5	5	8

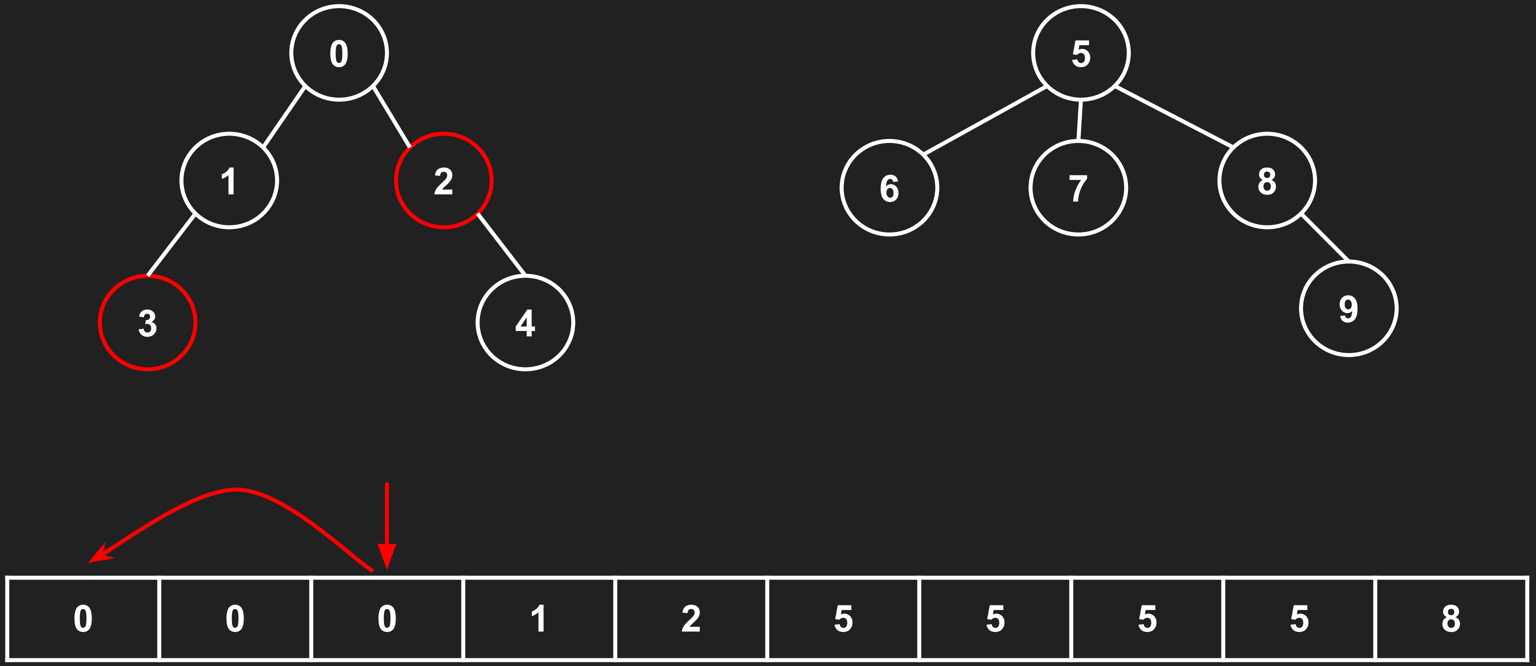
Representing Forests as Arrays



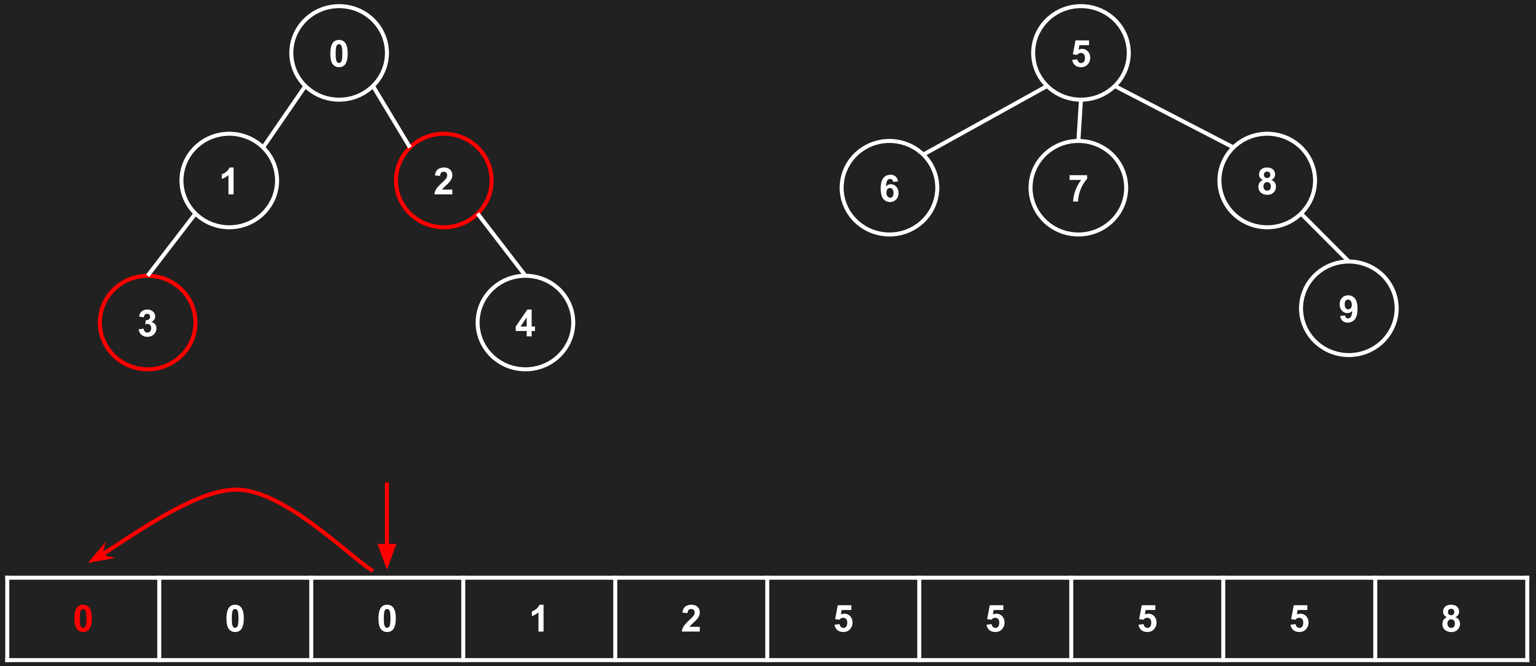
Representing Forests as Arrays



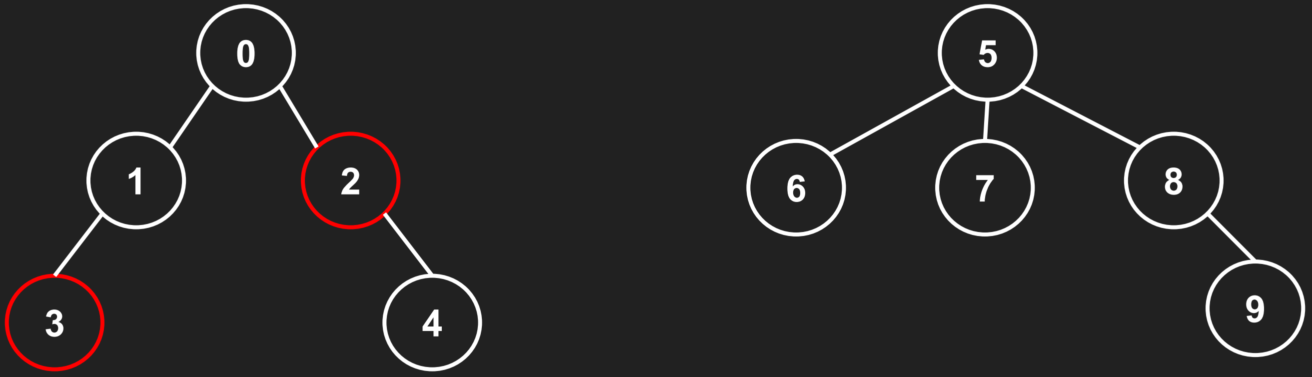
Representing Forests as Arrays



Representing Forests as Arrays



Representing Forests as Arrays

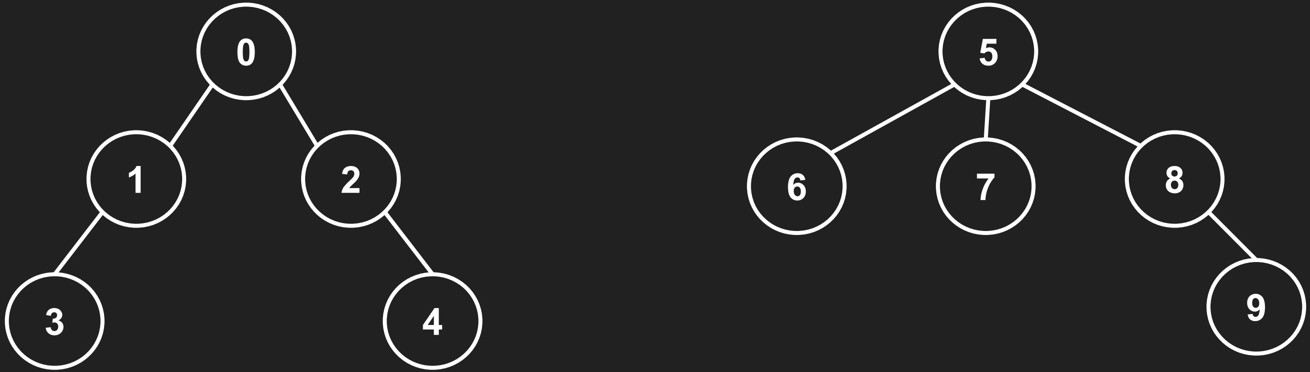


0	0	0	1	2	5	5	5	5	8

Union-Find

- Can take advantage of the tree-as-array representation to map each node to a unique “component ID”
  - The ID is the root of the tree
- The Find method can be implemented by walking up the tree
  - Using Find on two different nodes can tell you if they’re in the same partition
- Note: this representation isn’t perfect
  - Doesn’t easily let you walk down the tree
  - Not all graphs/connected components can neatly map into a forest without loss of edges
  - In other words, good for Union-Find but not a silver bullet for graph representation

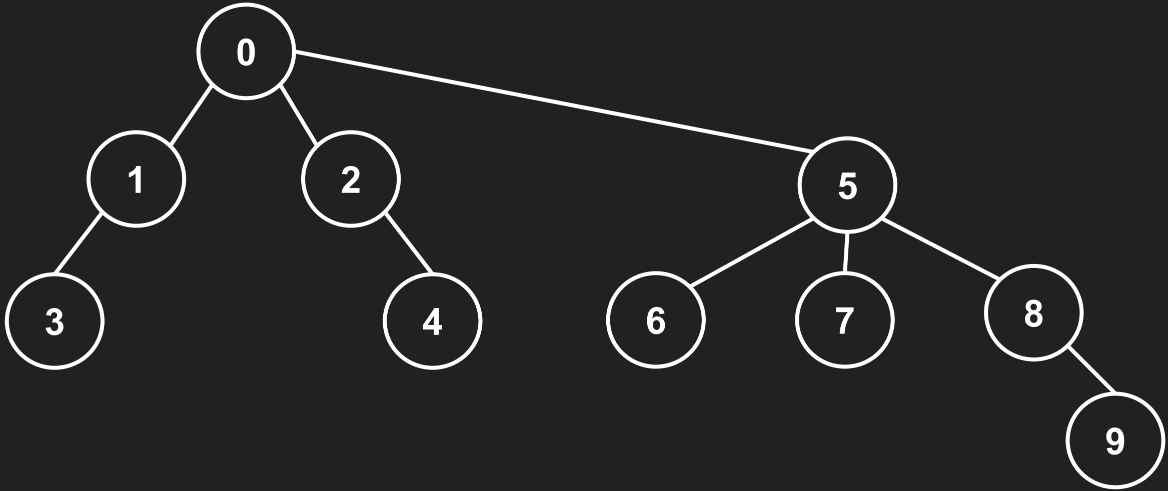
Representing Forests as Arrays



- What about union?

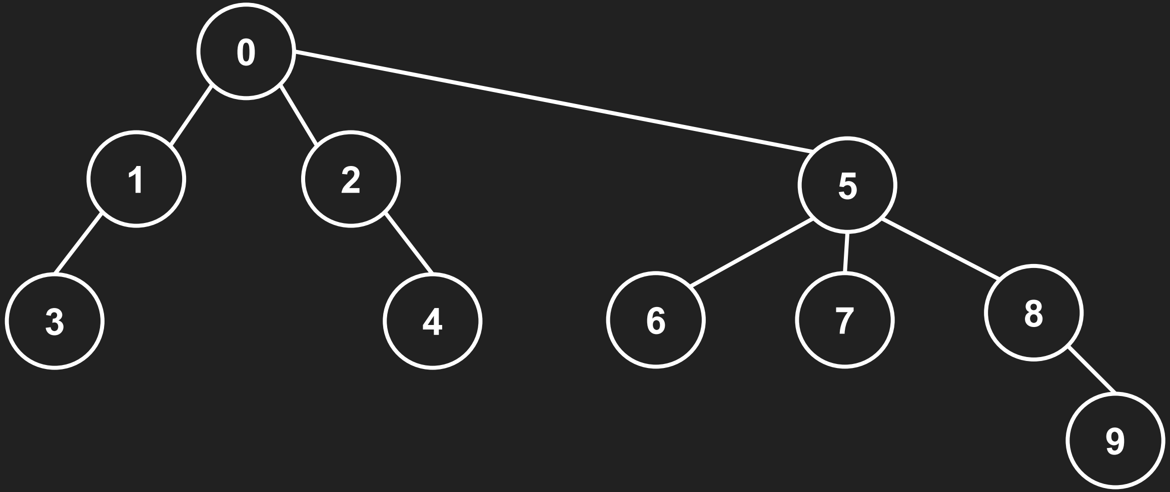
0	0	0	1	2	5	5	5	5	8

Representing Forests as Arrays



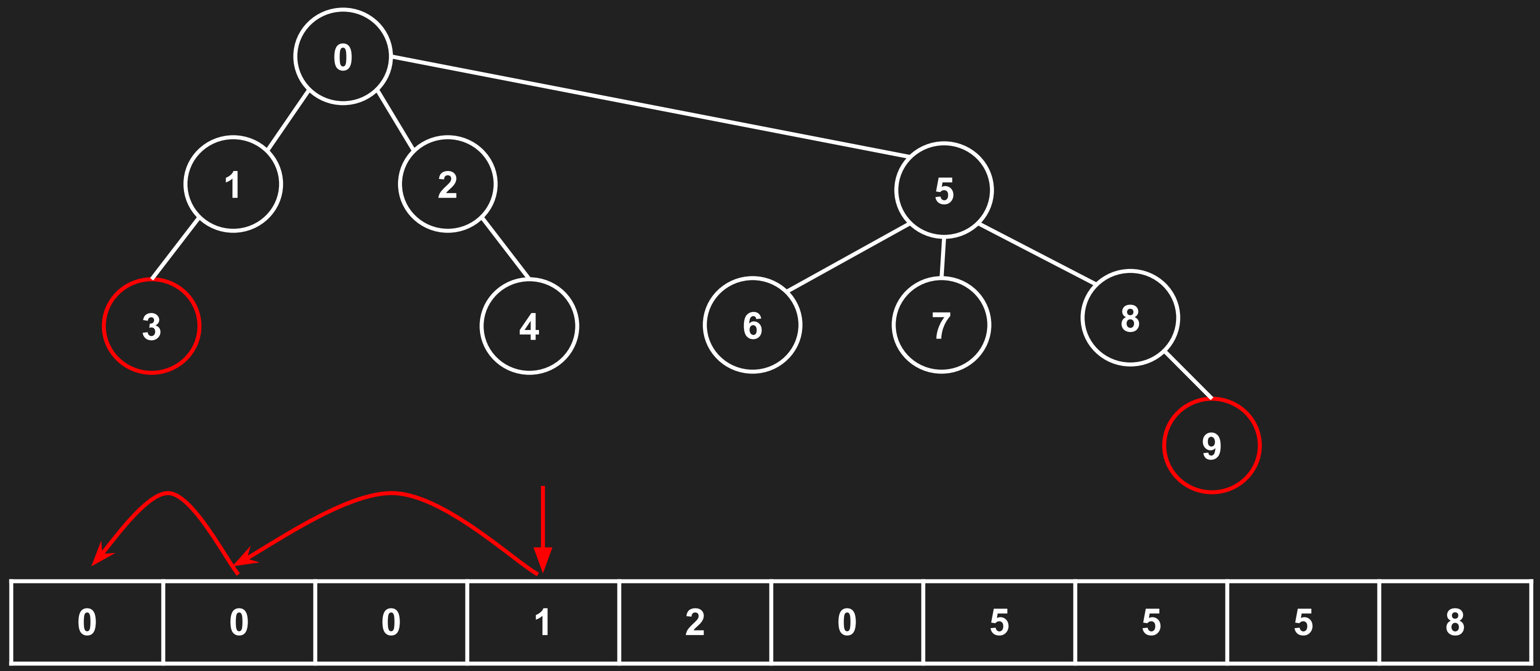
0	0	0	1	2	5	5	5	5	8

Representing Forests as Arrays

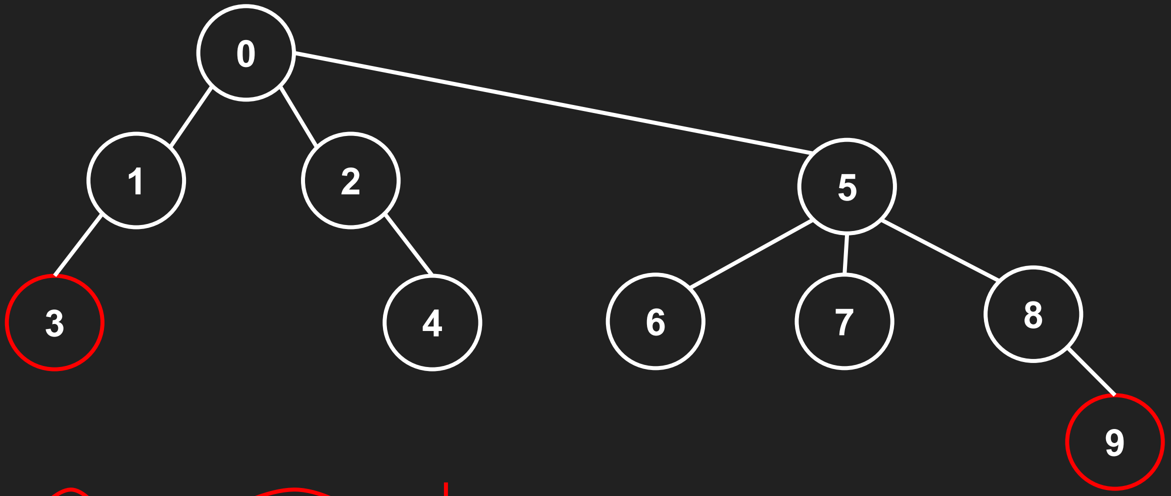


0	0	0	1	2	0	5	5	5	8

Representing Forests as Arrays

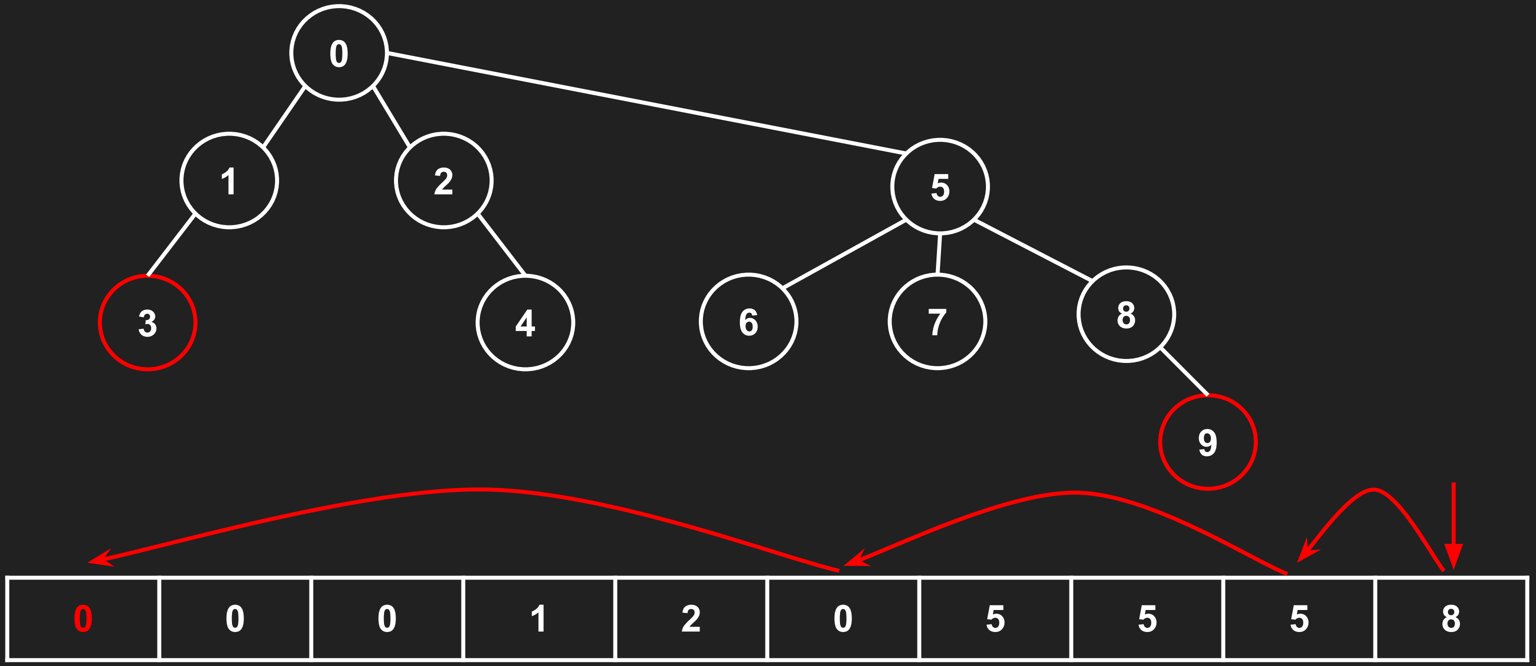


Representing Forests as Arrays



0	0	0	1	2	0	5	5	5	8

Representing Forests as Arrays



Union-Find

- The Union method is as easy as making the root of one tree a child of the root of another tree
  - In our data structure, this just means changing a single value in the array
  - Union might not be called on the roots, so generally Union requires calls to Find

Union-Find: Implementation

```
#`s` is a list representing the partitions
#`e` is the ID of a particular element
def find(s, e):
    p = s[e]
    if e == p:
        return p
    return find(s, p)
```

Union-Find: Implementation

```
#`s` is a list representing the partitions
#`e1` and `e2` are element IDs in partitions we wish to merge
def union(s, e1, e2):
    r1 = find(s, e1)
    r2 = find(s, e2)
    s[r2] = r1
```

Union-Find: Complexity Analysis

- Space: *O(n)*
  - Since we need this new array to store the partitions
- Find worst-case runtime: *O(n)*
  - Need to walk up tree
  - No guarantee the tree is balanced
- Union worst-case runtime: *O(n)*
  - Relies on Find, so can't be any better

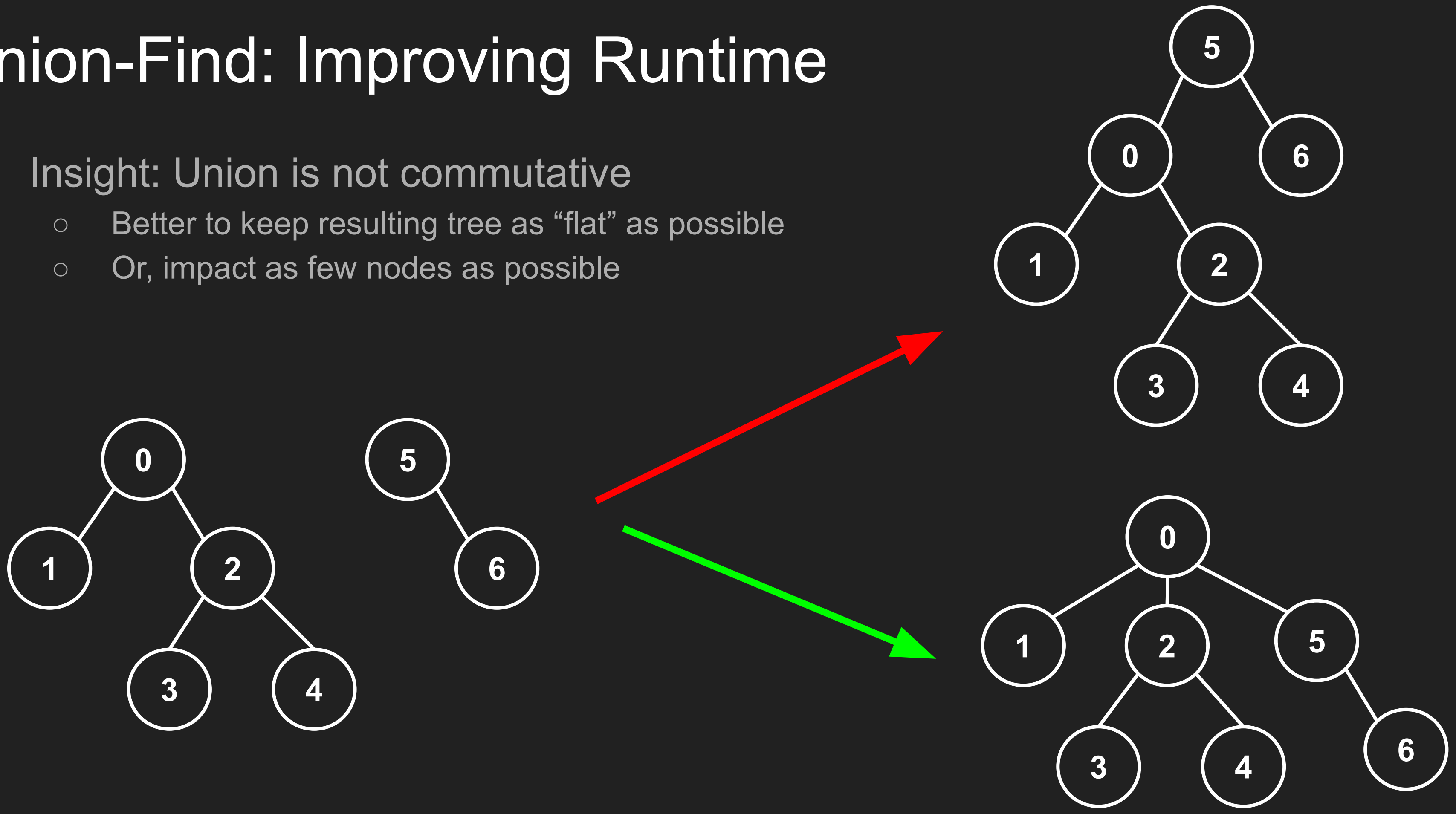
- Can we do better?
  - For space, no
  - For runtime, yes!

Union-Find: Improving Runtime

- Insight: for any given node, all we really care about is the root of its tree
  - In other words, the nodes in between don’t matter
  - Best-case scenario is a very “flat” tree
  - Fewer “hops” during Find

Union-Find: Improving Runtime

- Insight: Union is not commutative
  - Better to keep resulting tree as “flat” as possible
  - Or, impact as few nodes as possible



Union-Find: Improving Runtime

- Using first insight, we can improve Find
  - As we walk up the tree, we can be rewriting parents of visited nodes to point directly to root
  - Won’t improve first Find, but will improve all future ones
  - “Improve what you use”, “improve as you go”
  - Known as *path-compression*

- Using second insight, we can improve Union
  - Store either *size* or *rank* along with nodes, so you can compare subtrees
  - Choose the root of new tree to be the bigger/deeper tree
  - Known as *union-by-size* or *union-by-rank*
  - Both are reasonable, we’ll be looking at *union-by-size*
    - Note: this means we now need to store both *parent* and *size* in each array cell

Union-Find: Improved Implementation

```
#`s` is a list representation the partitions
#`e` is the ID of a particular element
def find(s, e):
    v = s[e]
    if e != v.parent:
        v.parent = find(s, v.parent)
    return v.parent
```

Union-Find: Improved Implementation

```
#`s` is a list representation the partitions
#`e1` and `e2` are element IDs in partitions we wish to merge
def union(s, e1, e2):
    r1 = find(s, e1)
    r2 = find(s, e2)
    if r1 == r2:
        return
    if s[r1].size > s[r2].size:
        s[r2].parent = r1
        s[r1].size += s[r2].size
    else:
        s[r1].parent = r2
        s[r2].size += s[r1].size
```

Union-Find: Revised Complexity Analysis

- Space: still $O(n)$
  - Storing size now, but still just an extra $O(n)$ integers
- Runtime of Union is still dependent on runtime of Find
- So what's the new runtime of Find/Union?
- Answer: almost $O(1)$, amortized
  - Note: "amortized" essentially means "smoothed out over many operations"
- Actual answer: $O(\alpha(n))$, amortized

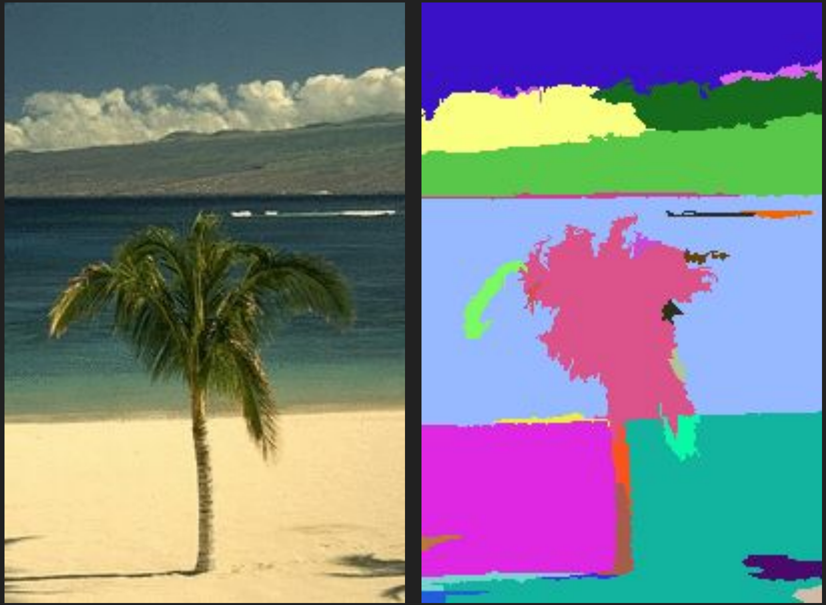
Digression: Ackermann Function

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

- This function grows REALLY FAST
  - Example: $A(4, 2)$ is 19,729 digits long
- If $f(n) = A(n, n)$, then $\alpha(n) = f^{-1}(n)$
  - Known as the “inverse Ackermann function”
- $\alpha(n)$ grows REALLY SLOW
  - Example: $\alpha(n) < 5$ for literally any n that can be written in this physical universe

Union-Find: Applications

- Image segmentation
  - Used for self-driving cars - seriously!
  - Cornell’s 2007 DARPA Urban Challenge car used this



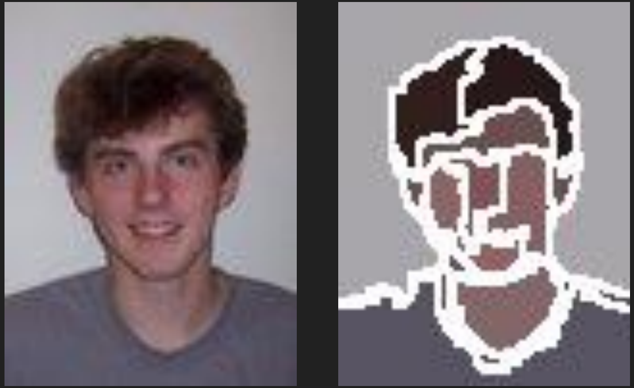


Image Segmentation

- Every pixel is a node, every node has an edge to its eight neighbors
- Edge weights are distance in RGB space
  - $\mathrm{sqrt}((r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2)$
- Start with each node in its own partition
- Define *Int(C)* to be the edge of greatest weight in connected component *C*
  - Called the “internal difference”
- Define *T(C)* to be *k* / |*C*|, where *k* is a constant
  - The “threshold”
- Iterate through edge weights from least to greatest
- For edge (v_1, v_2):
  - If v_1 and v_2 are already in the same connected component, remove the edge
  - Union connected components if $w(v_1, v_2) < min(Int(C_1) + T(C_1), Int(C_2) + T(C_2))$

Union-Find: Applications

- Optical Character Recognition (OCR)
  - Similar to image segmentation: can find similarly colored components to be the characters
  - Run character shapes through machine learning pipeline to match known character
    - Or, potentially just use a lookup table if you know the font!
  - Glossing over some details:
    - Dealing with letters like “i” which are not connected components
    - Dealing with “ligatures”