Case Study: Union-Find

Dr Nicholas Moore

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 2.2), Prof. Mhaskar's course slides, and `https://www.cs.princeton.edu/~rs/AlgsDS07/04Sorting.pdf`.

A Case Study in Algorithm Development

Things which are true:

- Good algorithms make unsolvable problems solvable.
- Good algorithms can be as simple to code as bad ones.
- **Iterative refinement** can lead to increasingly efficient algorithms.

Over the next few lectures, we will study the **Union-Find** algorithm:

1. To exemplify our approach to the study of algorithms.
2. Because it is a good and useful algorithm.

Problem Introduction: Union-Find

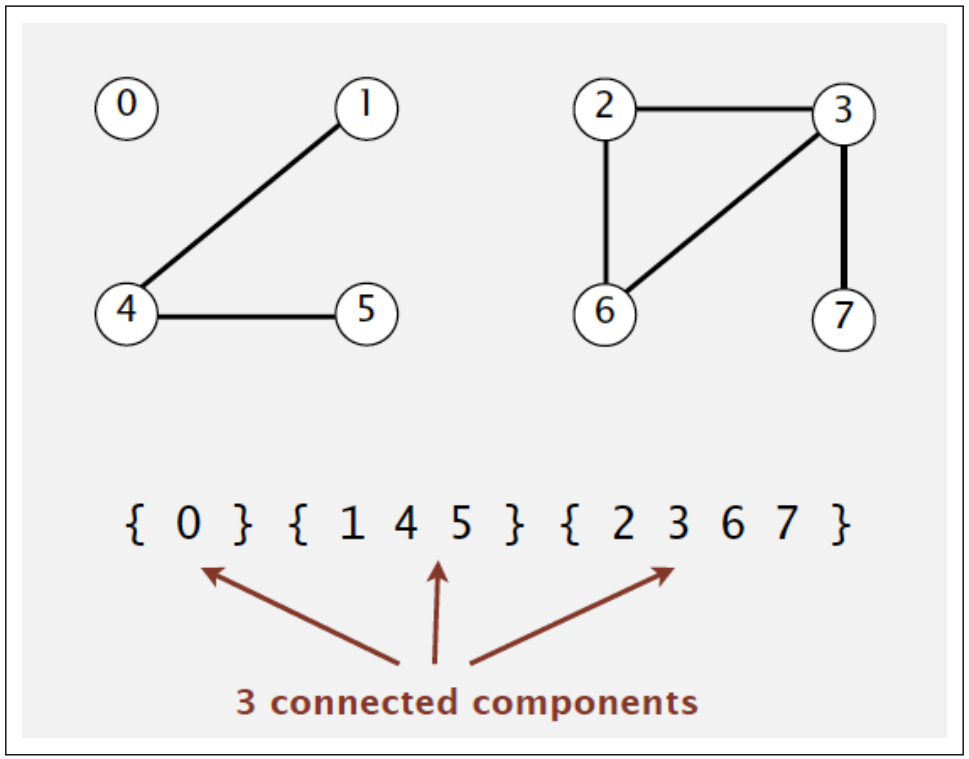
Although we have not formally studied graphs or graph algorithms yet, union-find happens to be a graph algorithm.

Imagine a series of nodes (or **sites**) and edges (or **connections**).

Scenario	Nodes	Edges
Networking	Computers	Network Connections
Programming	Objects	Pointers
Supply Chain	Ports	Ships
Social Media	"People"	"Friendships"

Defining Components

A **connected component** is a set of sites which are mutually connected.



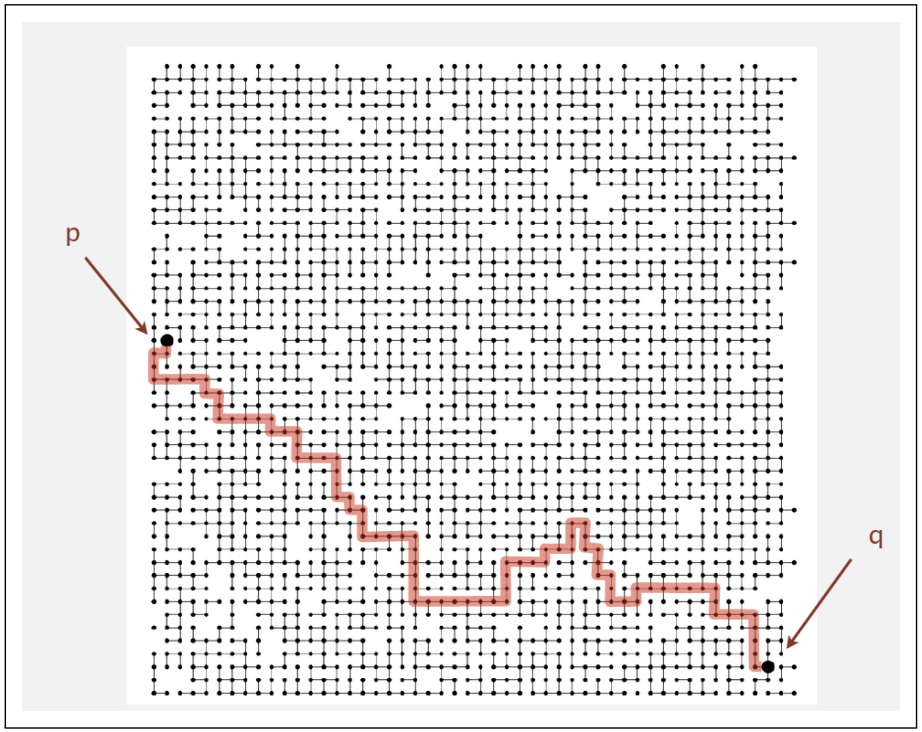
The largest possible number of components in any network is equal to the number of sites! (*why?*)

Union-Finder's Keepers!

Given a network (a collection of sites and connections), we may be interested in the following:

- CONNECTED: Given two arbitrary sites in the network, are these sites connected by the edges of the network?
  - i.e., are they in the same component?
- UNION: How might we add connections in order to connect and merge components?
- FIND: Given a site, which component is it a part of?
- COUNT: How many components does the network contain?

Diagrammatic Interpretation



To simplify the visuals in this lecture (at least to begin with) sites will be arranged rectangularly, and connections will only occur between adjacent sites. In reality the system is much more general.

- Sites and components will be identified using integer IDs.
- Connections will be pairs of IDs.

Networks, Using Set Theory

Our network is a **set** of sites, tupled with a **connection relation**.

$$N = (S, C) \qquad\qquad C \subseteq S \times S$$

If, for all $s, t \in S$ a pair of sites $(s, t) \in C$, then the connection "exists". C is an **equivalence relation**, so it is:

- **Reflexive**: $\forall s \in S \bullet \ (s, s) \in C$
  - i.e., all sites are connected to themselves.
- **Symmetric**: $\forall s, t \in S \bullet \ (s, t) \in C \implies (t, s) \in C$
  - i.e., connections are bi-directional.
- **Transitive** :
  $\forall s, t, u \in S \bullet \ (s, t) \in C \land (t, u) \in C \implies (s, u) \in C$
  - i.e., sites can be connected via "middlemen".

A-P-I can do that!

public class UF	
UF(int N)	*initialize N sites with integer names (0 to N-1)*
void union(int p, int q)	*add connection between p and q*
int find(int p)	*component identifier for p (0 to N-1)*
boolean connected(int p, int q)	*return true if p and q are in the same component*
int count()	*number of components*

Union-find API

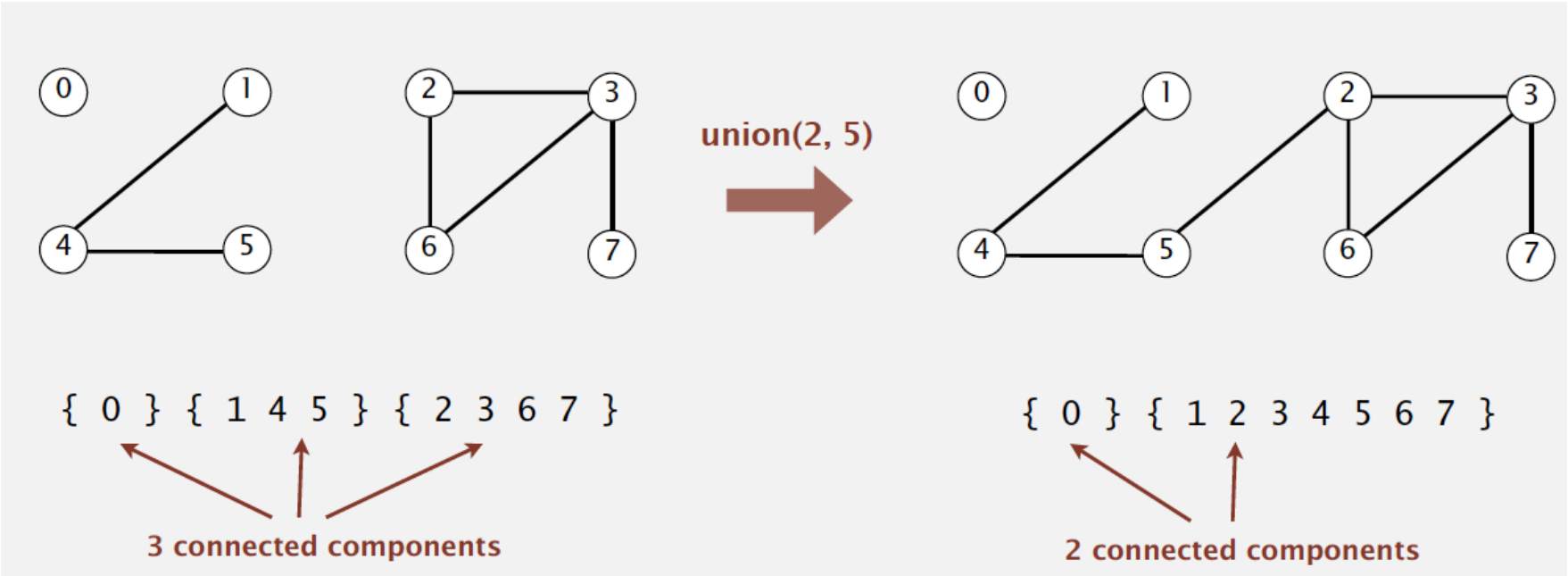
Note that:

- We identify sites p and q using integers
- Although it lacks a delete function, this definition is sufficient to build a network!

Union

The UNION operation creates a new connection between two sites, *if they are not already connected*.

- This has the added effect of merging, or finding the union of, the components wherein these sites dwell.




YOUR COUNTRY NEEDS YOU
TO STRUCTURE THAT DATA
imgflip.com

Naive Approach

Given our description so far, you may be tempted to do the following.

- Store your set of site IDs in an array or linked list.
- Create a class for the connections with two fields, one for each connection point, and then make an array references to them, or add pointers and make a linked list.

Alternatively...

- Make a struct for nodes, and then use pointers to represent the connections.

BOTH TERRIBLE!

Epiphanies

We can devise a much better system if we realize the following:

- These are not encrypted ID numbers, just the natural numbers between zero and the size of the network.
- While maintaining the connections numerically enables a large number of possible future operations, we do so at the expense of **present functionality**! There is *no need to store them*, so long as we can perform our operations.

Quick-Find: An Eager Approach

Rather than associating sites with **connections**, let's associate sites with **components**.

- Our data structure will be just a single integer array.
  - The Keys will be Site IDs.
  - The Values will be Component IDs.
- We will also maintain a `count` variable.

```
public class UF {
        private int[] id; \\ component IDs (site indexed)
        private int count; \\ number of components
        [...]
}
```

Initialization

The UF class we have described so far is initialized with no connections (recall the API!)

- All we do is create space for the correct number of sites (N).

Therefore, all connections are created by calls to the UF API post-initialization using UNION.

- All sites are connected to themselves.
- Initially all sites are connected to *nothing but* themselves.
- Therefore to initialize `id`, iterate through it and initialize each element to the value of it's index.

Find my Hat!

If we've set things up as above, what is the algorithmic complexity of the FIND operation? Recall:

- FIND takes a site ID, and returns the component ID that site belongs to.

How would this change if we used a linked list instead of an array?

Making Connections

Similarly, to test if two sites are CONNECTED, what do we do?

And what's the complexity?!

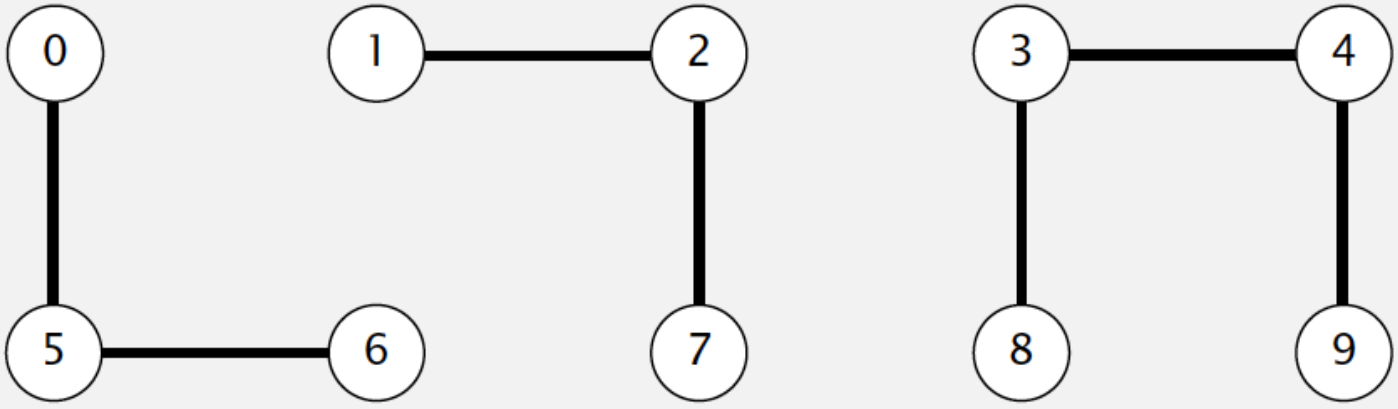
At Your Local Union

Consider the following component mapping.

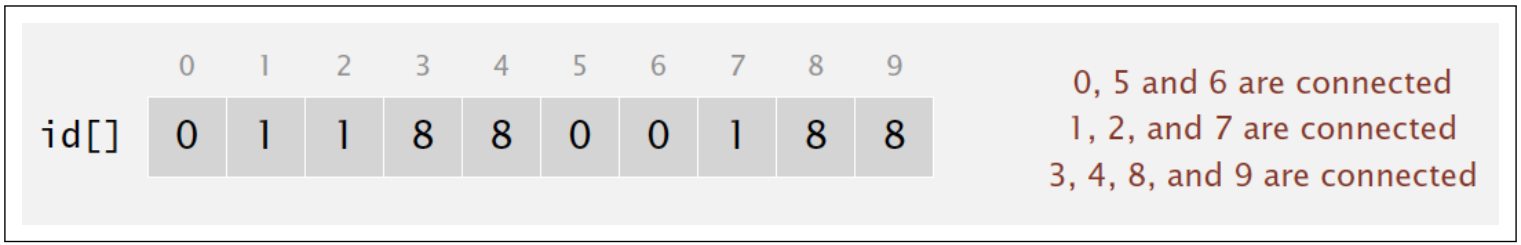
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	1	8	8	0	0	1	8	8

0, 5 and 6 are connected
1, 2, and 7 are connected
3, 4, 8, and 9 are connected

This network produces the following graph:



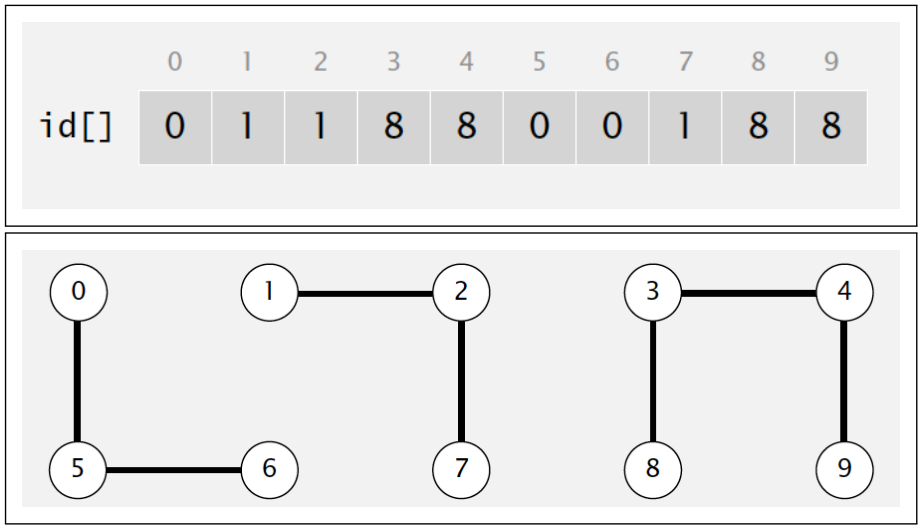
Union Dues



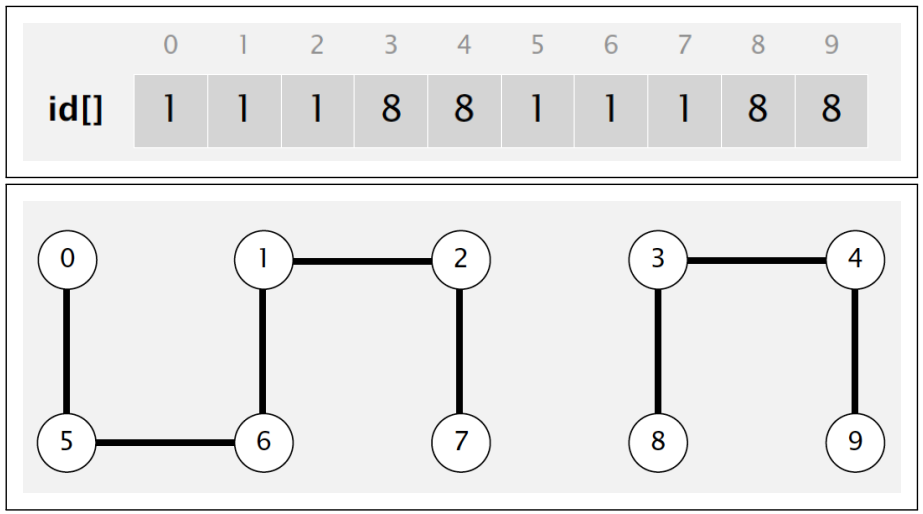
- What is the result of $\textsc{Connected}(1,6)$?
- What would it take for them to be connected?
- Does this suggest a procedure for connecting them?
- What is the time complexity of this procedure?

Union Busting

Before $\textsc{Union}(p = 1, q = 6)$



After



- Simply iterate through `id[]`, changing every occurance of id[p] to id[q], or vice versa.

Time complexity of Quick-Find

For a network of n nodes...

Algorithm	Quick-Find
INITIALIZATION	$O(n)$
FIND	$O(1)$
CONNECTED	$O(1)$
UNION	$O(n)$

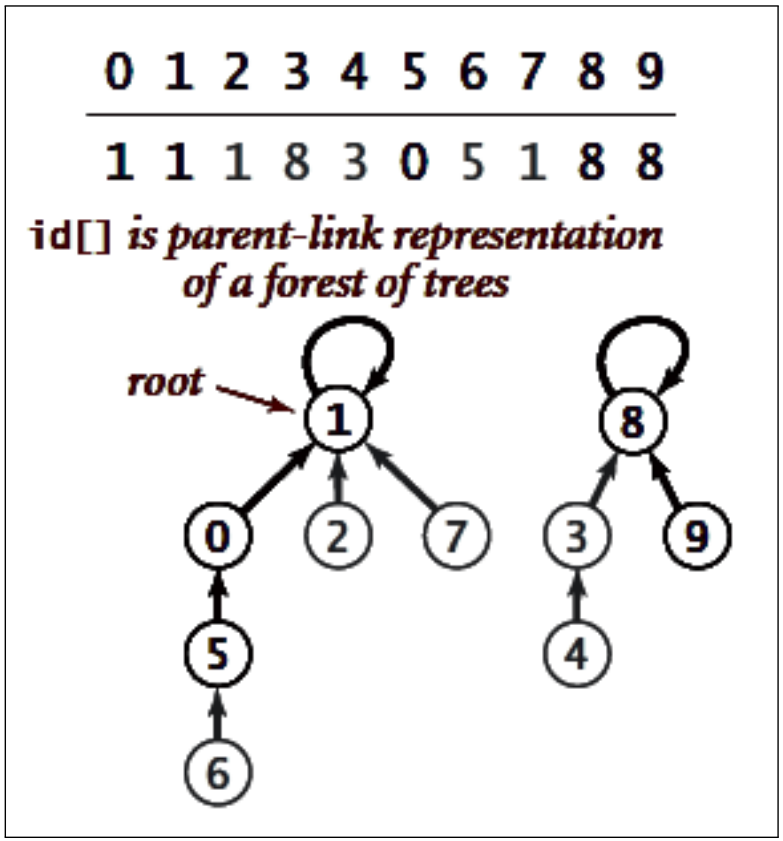
- However, let's also consider that we are adding connections one-at-a-time via UNION.
  - This means that for a network with m connections the time complexity would be O(mn).
- The Quick-Find approach optimizes for FIND at the expense of UNION.

Quick-Union: A Lazy Approach

Let's now consider Quick-Union, which optimizes for UNION at the expense of FIND.

- Quick-Union interprets `id[]` differently, though initialization is identical.
- Rather than containing somewhat arbitrary component numbers, each element of `id[]` is the site ID of another site in the same component.
  - Specifically, each site points to it's **parent node**.
  - The links form a tree structure, where the root node is self-referential.

Can't See the Forest for the Trees



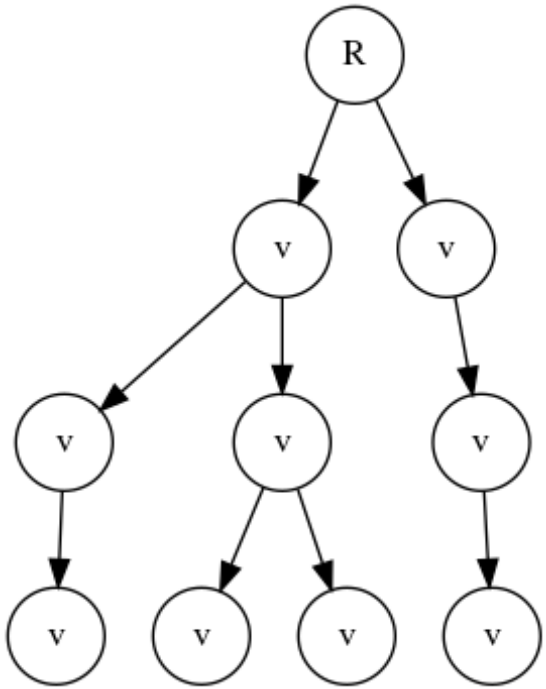
- **Tree** : A hierarchical data structure where each node has at most one parent, and any number of children.
- **Forest** : A disjoint collection of trees.
- **Root** : Each tree has exactly one node that has no parent.

In Quick-Union, root nodes are distinguished by pointing to themselves as their own parent.

Tree Anatomy

- A **leaf node** is any node in the tree which does not point to any other nodes.
- A **stem node** is any node in the tree which does point to other nodes.

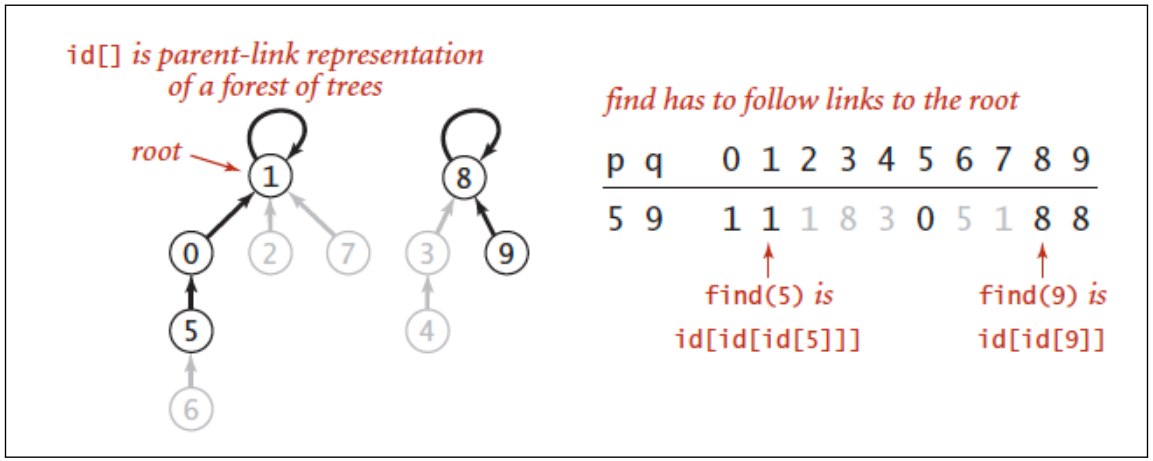
- The **depth** of a tree is number of nodes in the longest branch. (i.e., the distance from the root to the furthest leaf in nodes)
- The nodes in a tree are organized into **ranks** or **levels**.
- The **width** of a level is the number of nodes in that level.



Quick-Union: FIND

In order for FIND to be meaningful each component (tree) must be uniquely identified.

- Since root nodes are self-referential they all have unique values. We can therefore use them as component IDs.
- Given some site ID, FIND therefore need only follow the links of the tree back to the root node, and return that node's value.

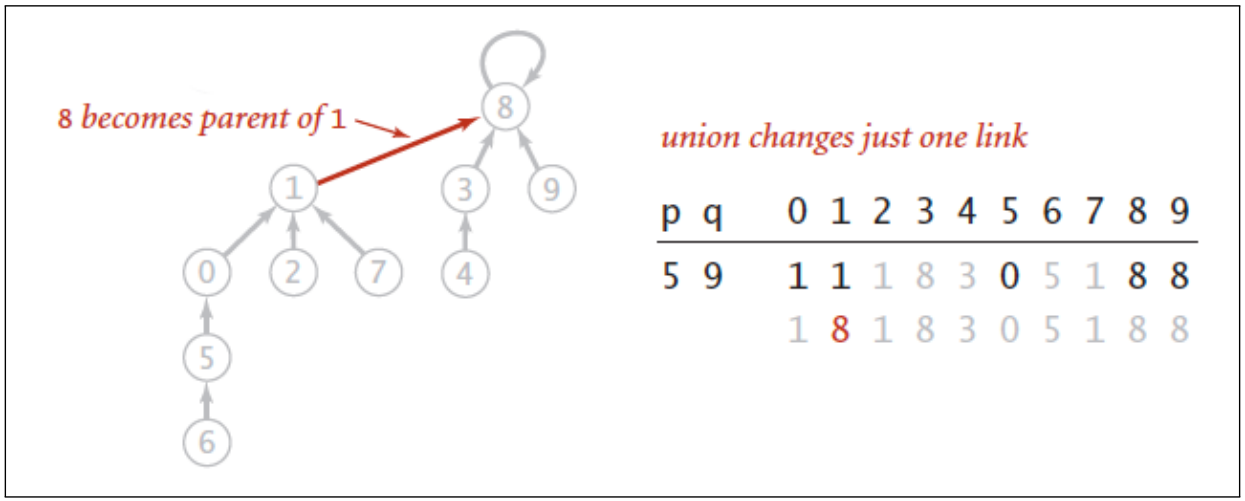


What's the worst case scenario?

Laziness and Unionization

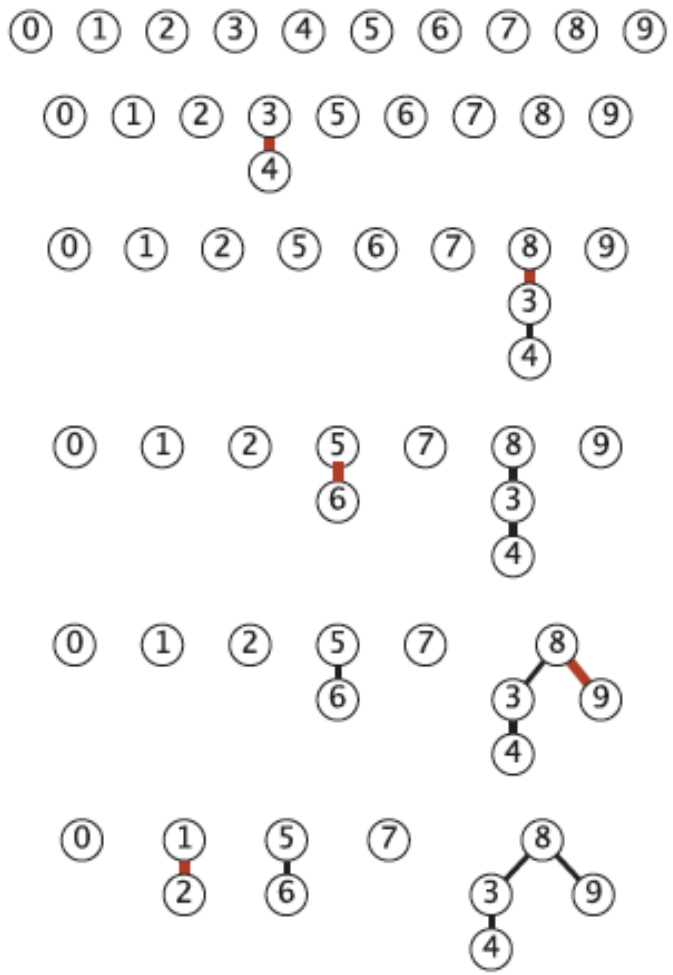
Any tree can become a subtree of another if its root node is directed to some node in another tree.

- Therefore, UNION first finds the roots of both components.
  - If they are the same, nothing further is required!
- Next, one of them is made to point at the other.
- Unification achieved!



I'm Calling My Union!

p	q	id[] 0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9
		0	1	2	3	3	5	6	7	8	9
3	8	0	1	2	3	3	5	6	7	8	9
		0	1	2	8	3	5	6	7	8	9
6	5	0	1	2	8	3	5	6	7	8	9
		0	1	2	8	3	5	5	7	8	9
9	4	0	1	2	8	3	5	5	7	8	9
		0	1	2	8	3	5	5	7	8	8
2	1	0	1	2	8	3	5	5	7	8	8
		0	1	1	8	3	5	5	7	8	8



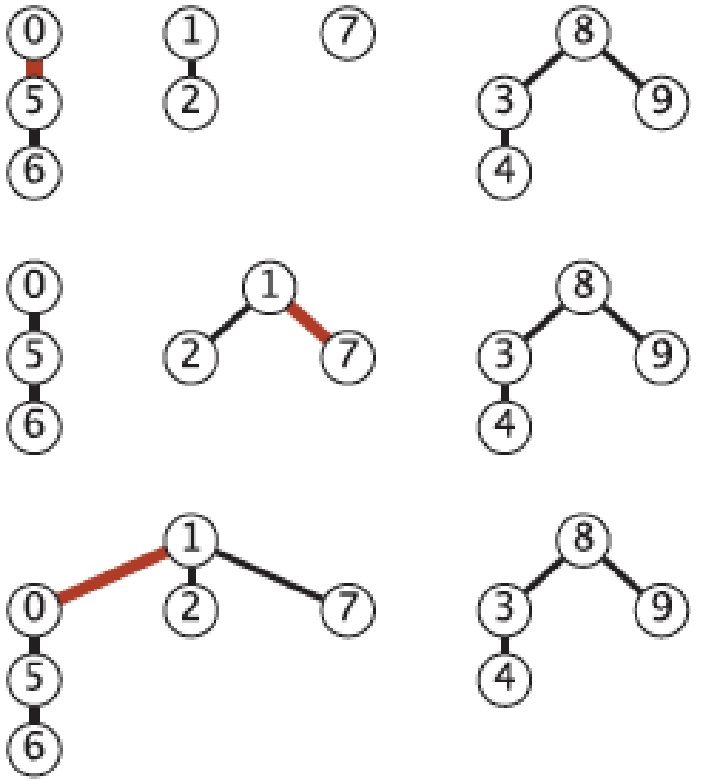
The Union's Going to Call Me Back...

```
8 9    0 1 1 8 3 5 5 7 8 8
5 0    0 1 1 8 3 5 5 7 8 8
       0 1 1 8 3 0 5 7 8 8

7 2    0 1 1 8 3 0 5 7 8 8
       0 1 1 8 3 0 5 1 8 8

6 1    0 1 1 8 3 0 5 1 8 8
       1 1 1 8 3 0 5 1 8 8

1 0    1 1 1 8 3 0 5 1 8 8
6 7    1 1 1 8 3 0 5 1 8 8
```



Time complexity of Quick-Find

For a network of n nodes...

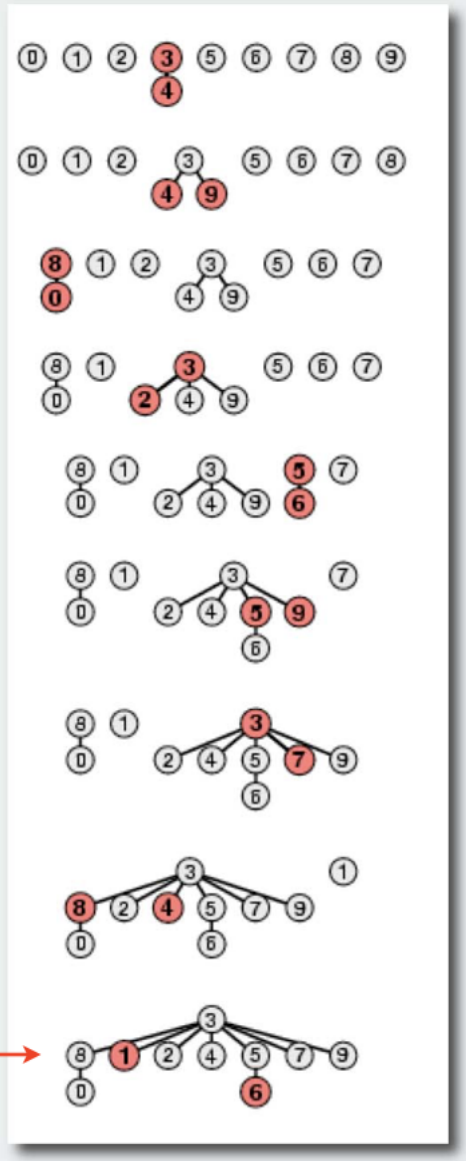
Algorithm	Quick-Find	Quick-Union
INITIALIZATION	$O(n)$	$O(n)$
FIND	$O(1)$	$O(n)$
CONNECTED	$O(1)$	$O(n)$
UNION	$O(n)$	$O(n)$

- Note that Quick-Union's UNION requires two calls to FIND, thus having at least as much time complexity.
- Although in the worst case Quick-Union performs worse than Quick-Find, in the average case it's quite a bit faster.
- If only there was some way of preventing the worst case scenario!

Weighted Quick-Union

- In addition to `id[]`, we maintain an extra array, `sz[i]`, to keep a count of the number of objects in the tree rooted at i.
- UNION is modified to link the root of the *smaller* tree to root of *larger*.
  - `sz[]` must also be updated accordingly

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



no problem: trees stay flat →

Weighted Quick-Union Runtime

The worst-case depth of a tree thus constructed is $\lg(n)$ (Note: $\lg(n) = \log_2(n)$)

- The worst case weighted merge is with two trees of equal size.
- This leads to the tree size doubling with each merger.
- The depth of a binary tree with 2^k nodes is k.

Since everything depends on the runtime of FIND, we have acheived logarithmic time across the board!

Algorithm	Quick-Find	Quick-Union	Weighted-QU
INITIALIZATION	$O(n)$	$O(n)$	$O(n)$
FIND	$O(1)$	$O(n)$	$O(\lg(n))$
CONNECTED	$O(1)$	$O(n)$	$O(\lg(n))$
UNION	$O(n)$	$O(n)$	$O(\lg(n))$

Weighted Quick-union with Path Compression

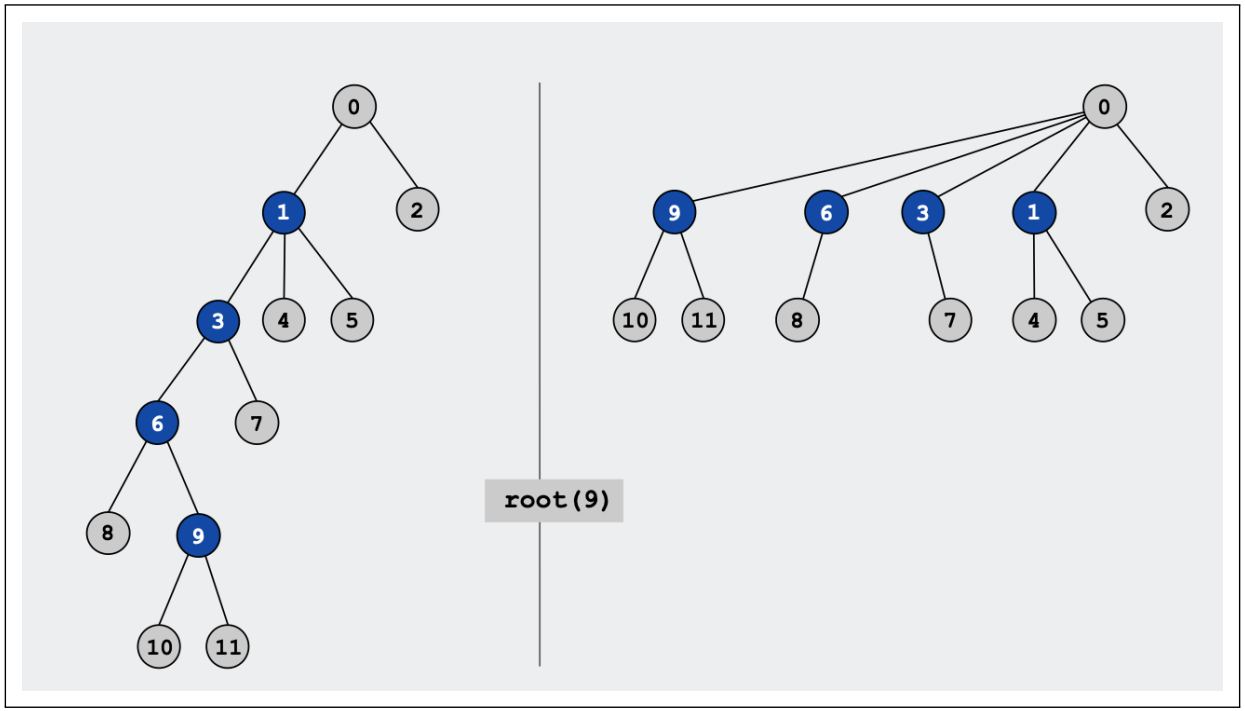
Doing something about the worst case scenario worked pretty well last time, so let's try it again!

- The worst case scenario given on the previous slide is where nodes form a **binary tree**.
- There is no restriction on how many child nodes a parent can have, however.
- Flatter trees improve find time with no consequences!
- The question becomes: How do we compress the path?

Finders Keepers

The least expensive thing to do is piggy-back path compression on an operation that is already traversing the trees.

- This leaves only FIND.
- Technically, we are adding a **side-effect** to FIND.



Compressed Java = Espresso?

Approach 1: Two-Pass
Keep track of all nodes visited, and loop through them at the end, redirecting them to the found root node.

- Adds a loop, but does not increase complexity.

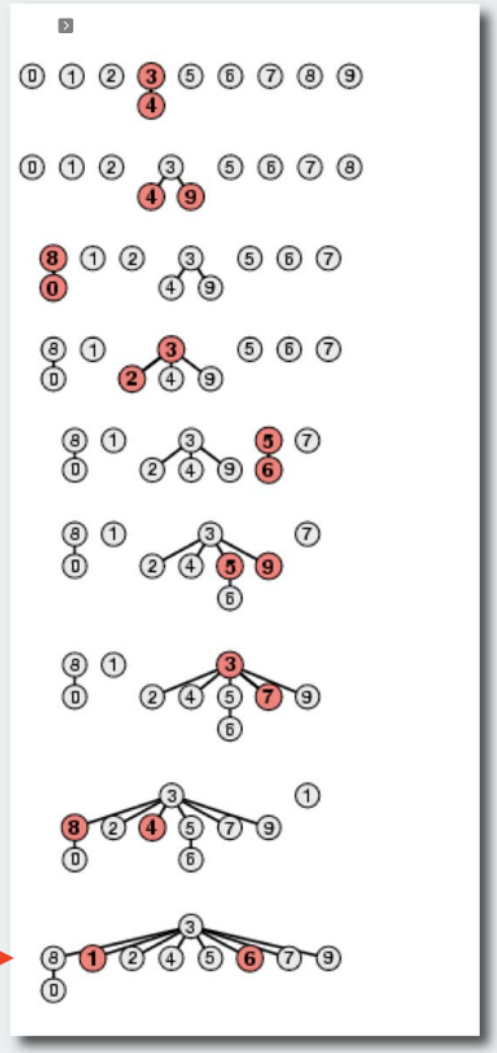
Approach 2: Single-Pass
Change each node to point at it's gradparent.

- Adds only one line of code!
- Works because root nodes are their own parents.

Successive Unions with Path Compression

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

no problem: trees stay VERY flat




Disjoint Tree
Collection Path
Compression in Java
Forestpresso
imgflip.com