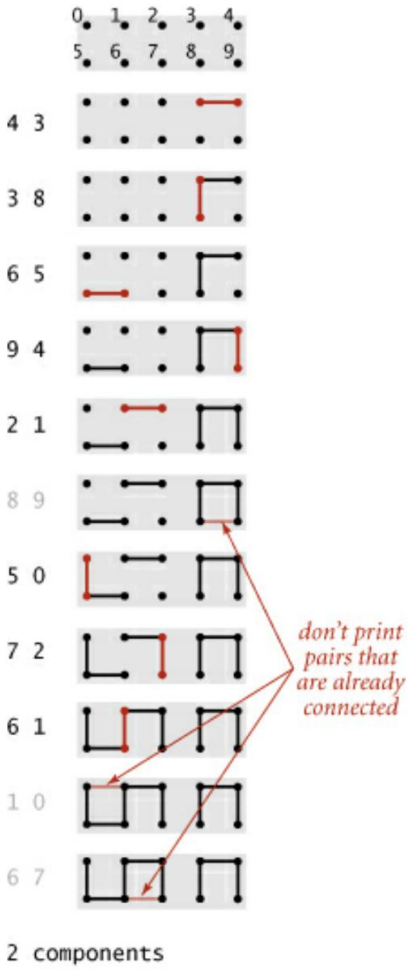
Union Find

Problem: Dynamic Connectivity

- Input: a sequence of pairs of integers
- A pair (p, q) means “p is connected to q” where connectivity is:
  - Reflexive: p is connected to p
  - Symmetric: If p is connected to q, then q is connected to p.
  - Transitive: if p is connected to q and q is connected to r, then p is connected to r.
- So connectivity is an equivalence relation, which can separate objects into equivalence classes (i.e. here, two objects are in the same equivalence class if and only if they are connected.)
- Goal: Filter out extraneous pairs (i.e. ignore any pair (p, q) where p and q are ALREADY in the same equivalence class.

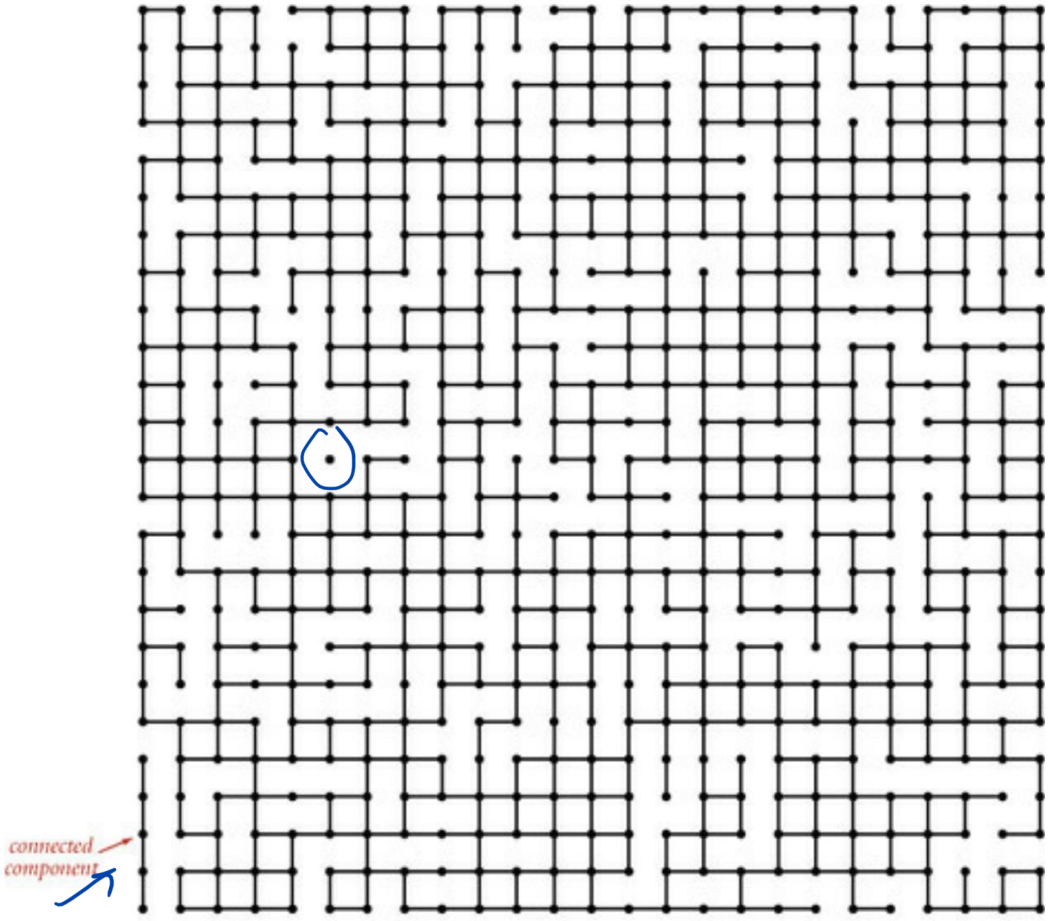


Dynamic connectivity example

Applications of dynamic connectivity...

- Networks: Given two computers that need to be connected, determine if a new direct connection needs to be set up (may not be necessary if they are already connected.)
- Variable name equivalence: An early example from FORTRAN that motivated the development of some of the algorithms. Idea: Determine if two variables refer to the same object.
- Mathematical sets: If p and q are "connected," they are in the same set—to process a new (p, q) pair that are not already in the same set, we UNION the two sets.
- Graph connectivity.

image from [1]



Medium connectivity example (625 sites, 900 edges, 3 connected components)

- 625 sites (vertices)
- 900 edges
- 3 connected components

image from [1]

Union-Find API

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

image from [1]

How do we implement this so the operations are efficient?

How do we implement this so the operations are efficient?

Some general implementation choices…

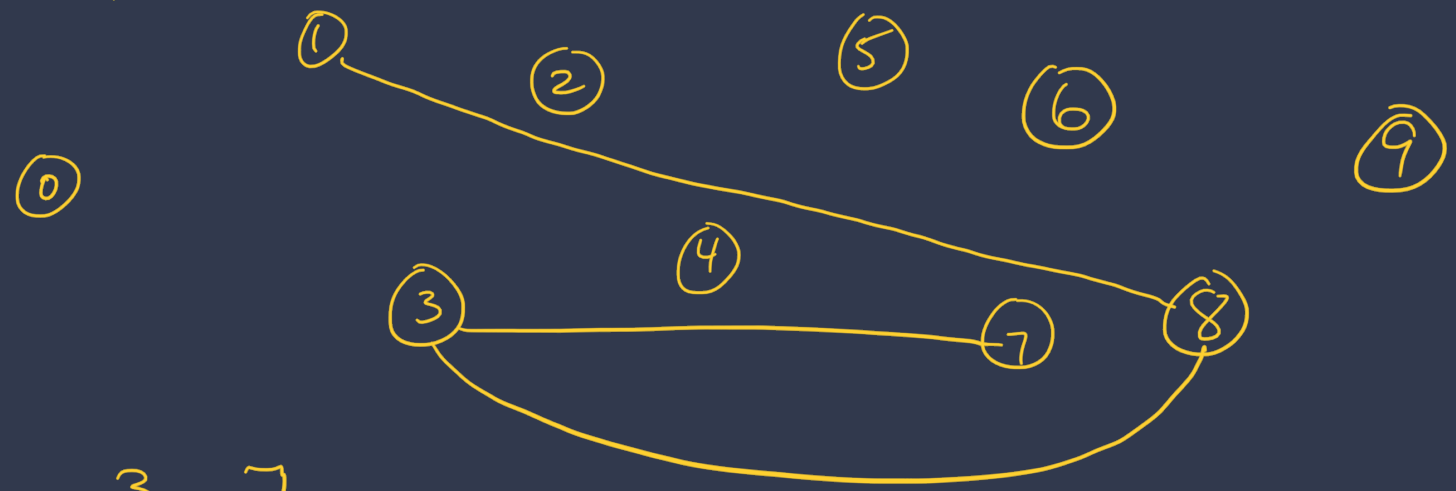
- The Union-Find implementation will maintain an array called **id** that keeps track of the component of each vertex. (i.e. if **id[i]=4**, then vertex *i* is in the component labeled 4.)
- Initially, each vertex is its own component, so **id[i] = i** for all **i**.
- Maintain a count of the number of components. To start with, this is **n** (the number of vertices.)

Option 1: Quick-find

- **UF(int n):** initialize **id** values--**O(n)**
- **union(int p, int q):**
  - get **id[p]** and **id[q]**
  - **if** (id[p] == id[q]) do nothing
  - **else**
    - scan **id** and set any element that is equal to id[p] to id[q]--**O(n)**
    - decrement component count
- **find(int p):** return **id[p]--O(1)**
- **connected(int p, int q):** return id[p] == id[q]--**O(1)**
- **count():** return count--**O(1)**

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

Count= ~~10~~ ~~9~~ ~~8~~ 7



union 3 7

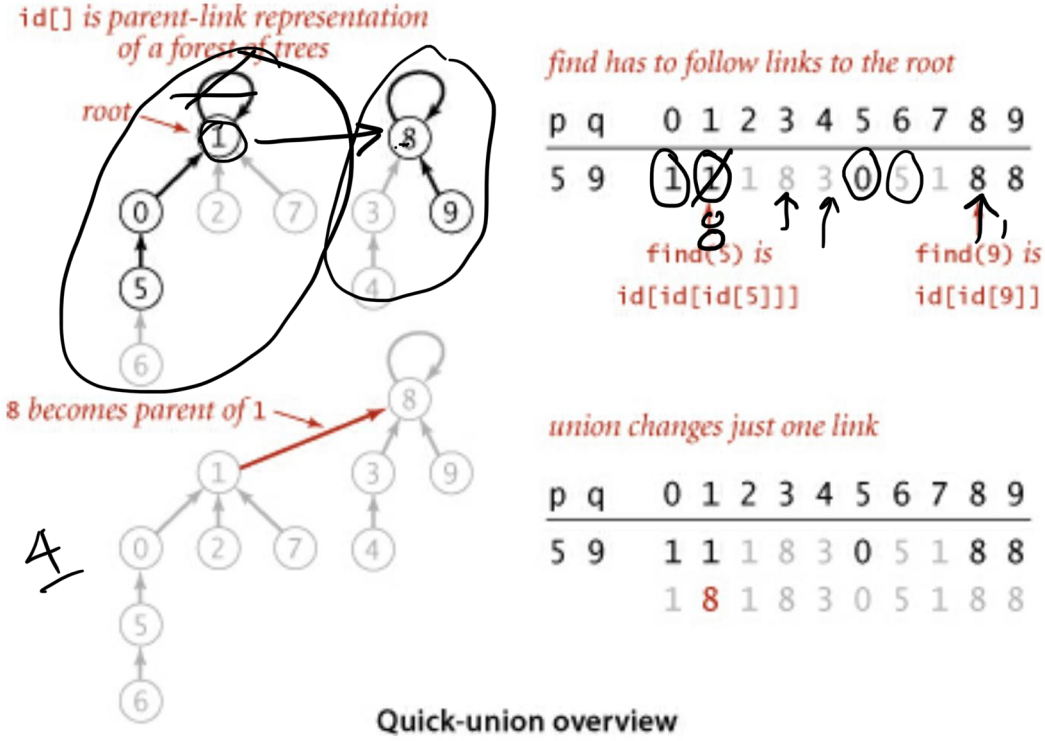
union 3 8

union 8 1

union 7 8

Option 2: Quick-union

- **id** is set up like a tree so that **id[i]** gives you its parent, and so on, until you get to a value that points to itself (the root)
- only one update is needed to union two sets but how many items are checked to find the root?



Quick-union overview

image from [1]

Option 2: Quick-union

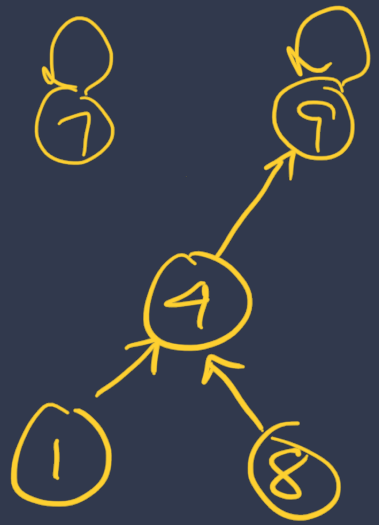
- **UF(int n):** initialize **id** values--**O(n)**
- **union(int p, int q):**
  - follow the “path” to find the root of the components of **p** and **q--O(tree height)**
  - reassign one root to point to the other
  - decrement component count
- **find(int p):** follow the “path” to find the root--**O(tree height)**
- **connected(int p, int q):** return find(p) == find(q)--**O(tree height)**
- **count():** return count--**O(1)**

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

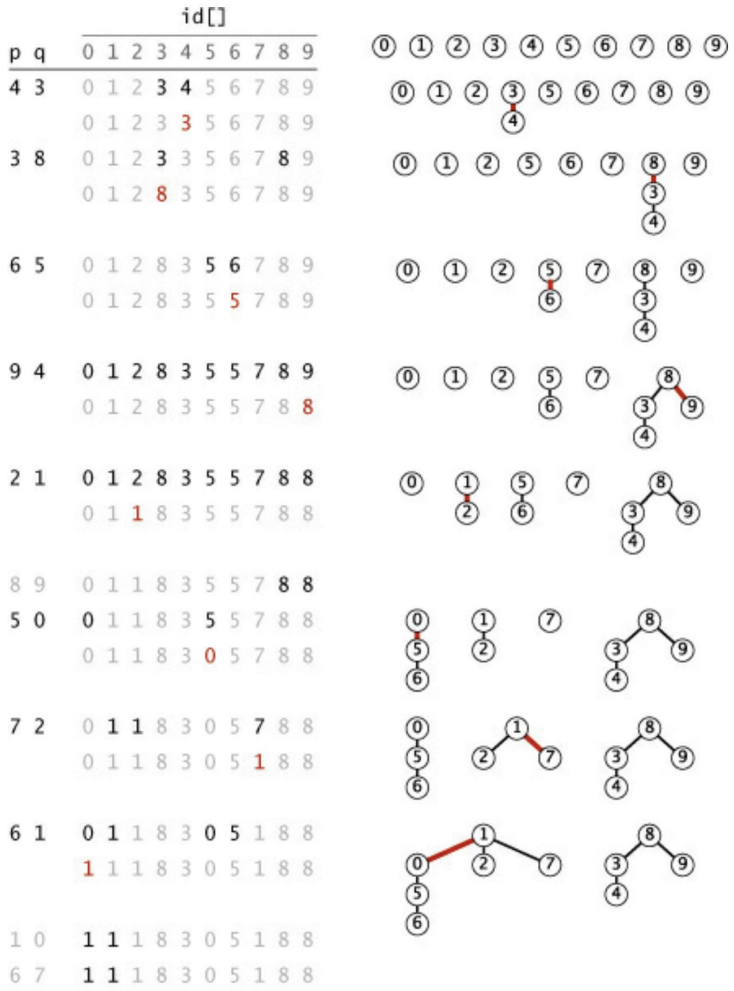
union 1 4

union 8 1

union 1 9



count: ~~10~~ ~~9~~ ~~8~~ 7



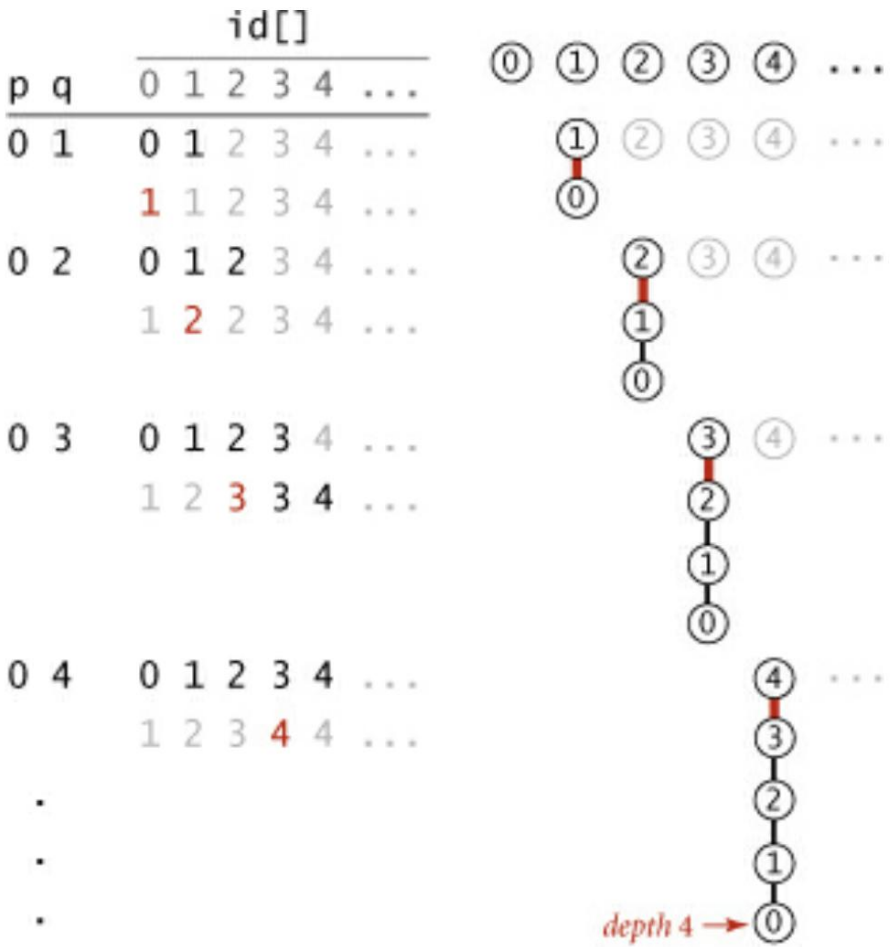
Quick-union trace (with corresponding forests of trees)

Quick-union trace with representation as a forest of trees.

As connections are added, you get fewer but larger trees (correspond to components).

If the runtime of key operations depends on the height of the tree, what is the worst case?

image from [1]



Quick-union worst case

Worst Case: O(N)

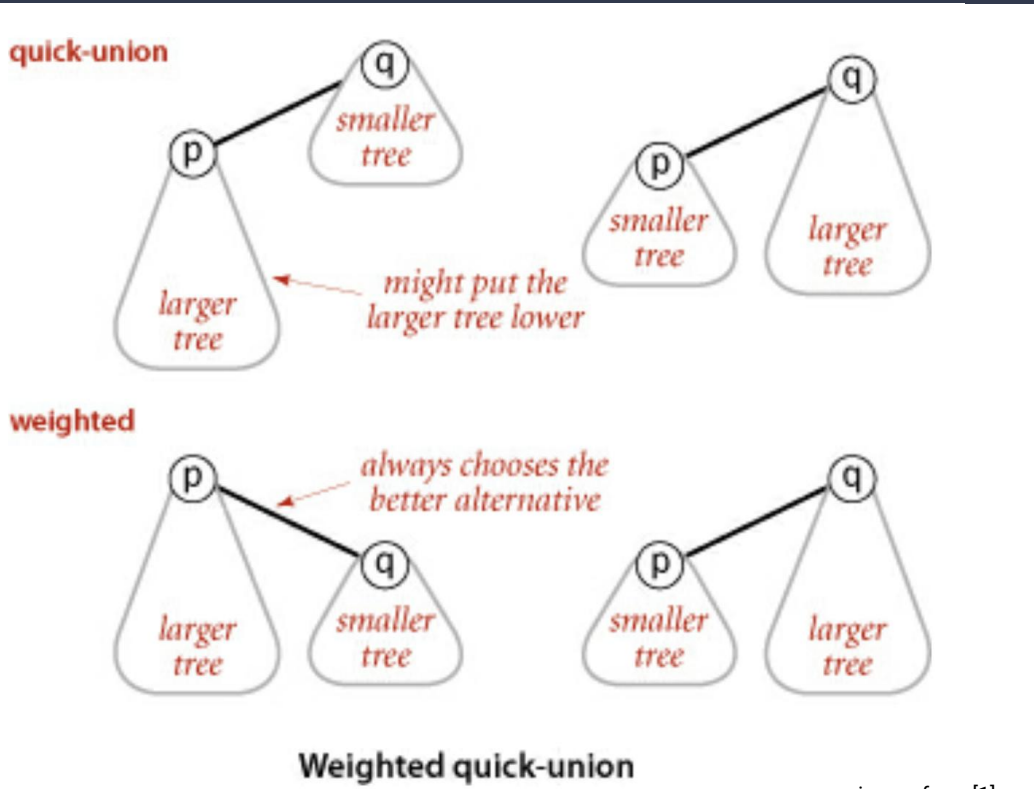
Can we improve upon this?

image from [1]

Option 3: Weighted Quick-union

Analysis:

- In the worst case, the trees are of equal size and combining them results in a tree of twice the size whose height increases by 1.
- Consider two trees with 2^k nodes. Then combining those trees will produce a new tree of size 2^{k+1} where the height has increased by 1.
- If we produce all trees this way, we guarantee that the height of the tree is **O(logN).**

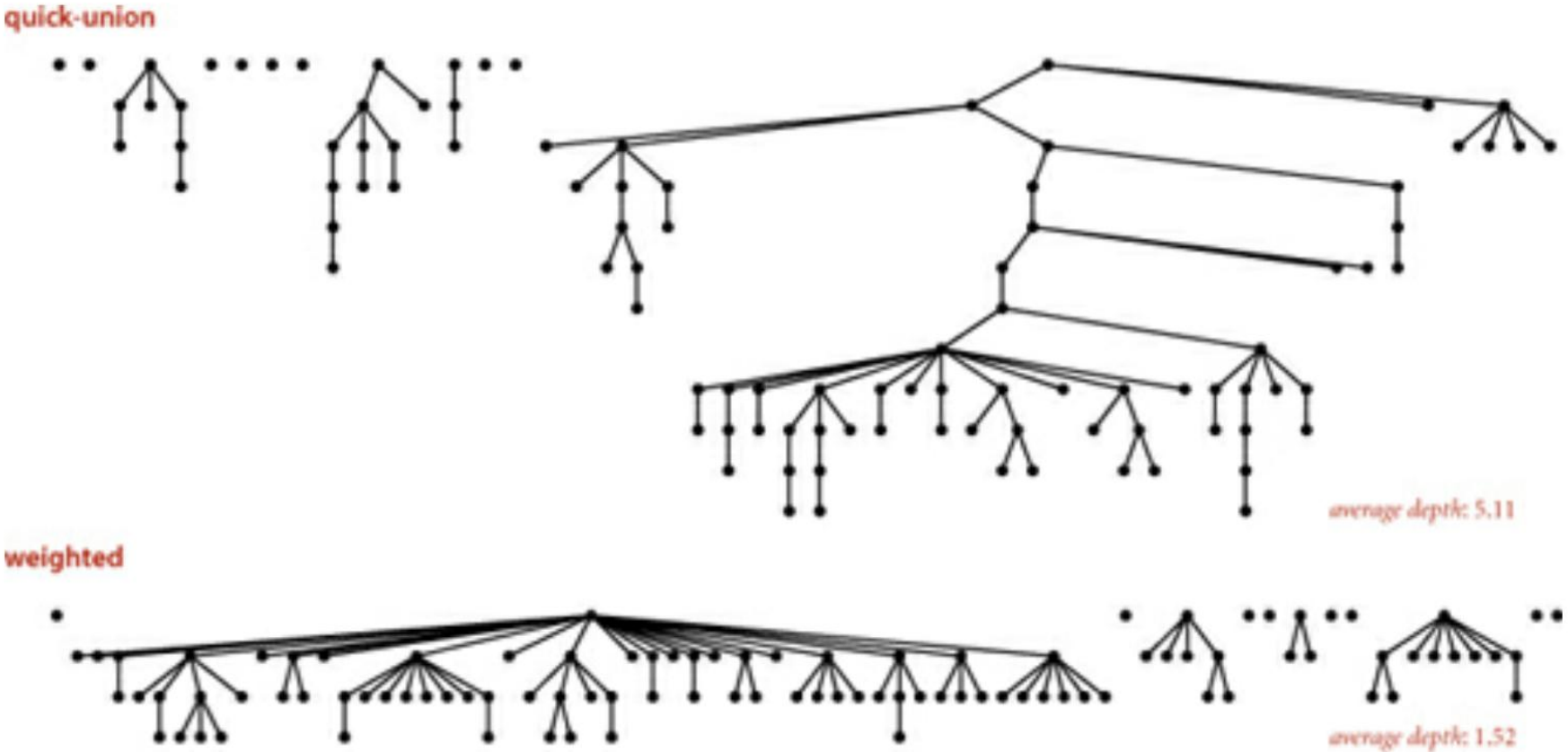


Weighted quick-union

image from [1]

Option 3: Weighted Quick-union

- **UF(int n):** initialize **id** values--**O(n)**
- **union(int p, int q):**
  - follow the “path” to find the root of the components of **p** and **q--O(logN)**
  - reassign the root of the smaller tree to point to the other
  - decrement component count
- **find(int p):** follow the “path” to find the root--**O(logN)**
- **connected(int p, int q):** return find(p) == find(q)--**O(logN)**
- **count():** return count--**O(1)**



Quick-union and weighted quick-union (100 sites, 88 union() operations)

image from [1]

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

union 3 8

union 3 1

count = ~~10~~ ~~9~~ 8

Option 4: Weighted Quick-union with Path Compression

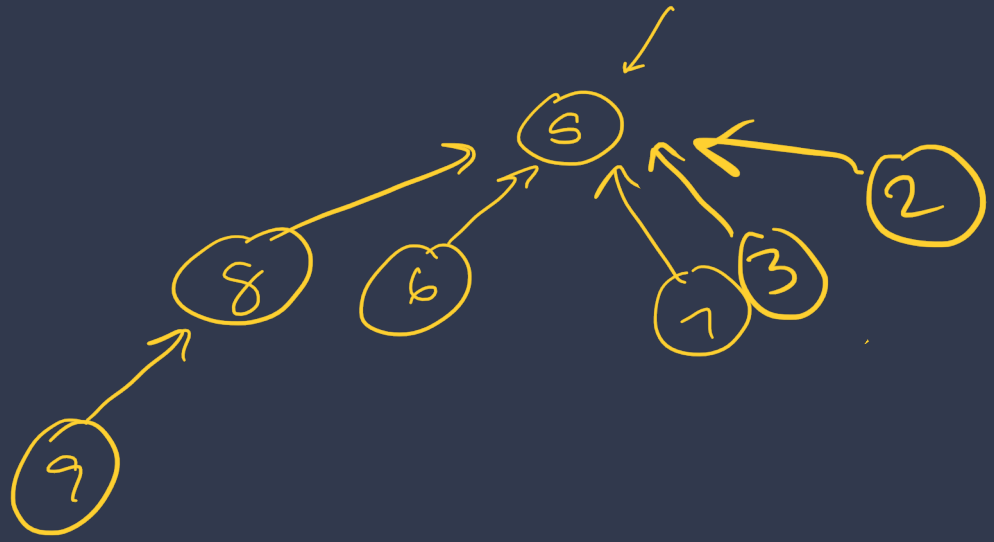
- Weighted Quick-Union with an extra optimization.
- **Main Idea:** Ideally, we want every node to link directly to its root.
- **But**...it's expensive to change all the nodes (remember Quick-Find?)
- **Solution**: Change the ones you are already examining as you look for the root--this just requires another loop to the *find* operation that sets the id[] value of each node on the path to the root. What's the runtime?

Option 4: Weighted Quick-union with Path Compression

What's the runtime?

- The worst-case for *find* and *union* may still be *O(logN)* but theoretical analysis shows that the amortized time for both is **close to constant**.
- However, it should be noted that you're unlikely to see much of a difference in practical situations between this and simple weighted quick-union.

0	1	2	3	4	5	6	7	8	9



union 2, 9

algorithm	order of growth for N sites (worst case)		
	constructor	union	find
quick-find	N	N	1
quick-union	N	*tree height*	*tree height*
weighted quick-union	N	$\lg N$	$\lg N$
weighted quick-union with path compresson	N	*very, very nearly, but not quite* 1 *(amortized)* (see EXERCISE 1.5.13)	
impossible	N	1	1

Performance characteristics of union-find algorithms

image from [1]

References

[1] *Algorithms, Fourth Edition;* Robert Sedgewick and Kevin Wayne