

CSC 226

Algorithms and Data Structures: II

Rich Little

rlittle@uvic.ca

ECS 516

Kruskal's algorithm requires an efficient way of testing whether an edge creates a cycle with the edges already selected.

- The union-find data structure helps do this.

Kruskal's Algorithm

Algorithm Kruskal

Input: a weighted connected graph $G = (V, E)$

Output: an MST T for G

Data structure: Disjoint sets (lists or union-find) DS ;
sorted weights priority queue A ; and tree T

for each $v \in V$ **do** $C(v) \leftarrow DS.insert(v)$ **end** // one cluster per vertex

for each $(u,v) \in E$ **do** $A.insert((u,v))$ **end** // sort edges by weight

$T \leftarrow \emptyset$

while T has fewer than $n-1$ edges **do**

$(u, v) \leftarrow A.deleteMin()$ // edge with smallest weight

$C(u) \leftarrow DS.findCluster(u)$;

$C(v) \leftarrow DS.findCluster(v)$;

if $C(u) \neq C(v)$ **then**

 add edge (u, v) to T ;

$DS.insert(DS.union(C(v), C(u)))$; // merge two clusters

end

end

return T

Disjoint-Set Data Structure

- Given a set of elements, it is often useful to break them up or partition them into a number of separate, non-overlapping sets
- A *disjoint-set data structure* is a data structure that keeps track of such a partitioning
- There are three useful operations on such a data structure:
 - *Find*: Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
 - *Union*: Combine or merge two sets into a single set.
 - *MakeSet*: Creates a set containing only a given element

Disjoint Set Data Structure

- The universe consists of n elements, named $1, 2, \dots, n$
- The ADT is a collection of sets of elements
- Each element is in exactly one set
 - Sets are disjoint
 - To start, each set contains one element
- Each set has a name
 - Which is the name of one of its elements
 - Any name of one of its elements will do

Disjoint Set Operations

- **find(elementName)**
 - Returns the name of the unique set that contains the given element
- **union(setName1, setName2)**
 - Merges two sets and replaces them with one
- **Time complexity analysis**
 - Involves analyzing the *amortized* worst-case running time over a sequence of f find and u union operations

Disjoint Set Implementation I

- Create a linked list for each set and choose the element at the head of the list as the representative
- *MakeSet* creates a list of one element
- *Union* simply appends two lists, a constant-time operation
- *Find* requires linear time (i.e., may search entire list)
- A sequence of m union-find operations takes time $O(mn)$. The amortized time per operation is $O(n)$.

How should we represent the sets?

- Each set is a rooted tree
- Each element of a set corresponds to a node in a tree
- *Canonical element* (= name of set) is the root of the tree
- The textbook has 3, increasingly better, ways of implementing this.
 1. quick-find
 2. quick-union
 3. weighted quick-union

Quick-find [eager approach]

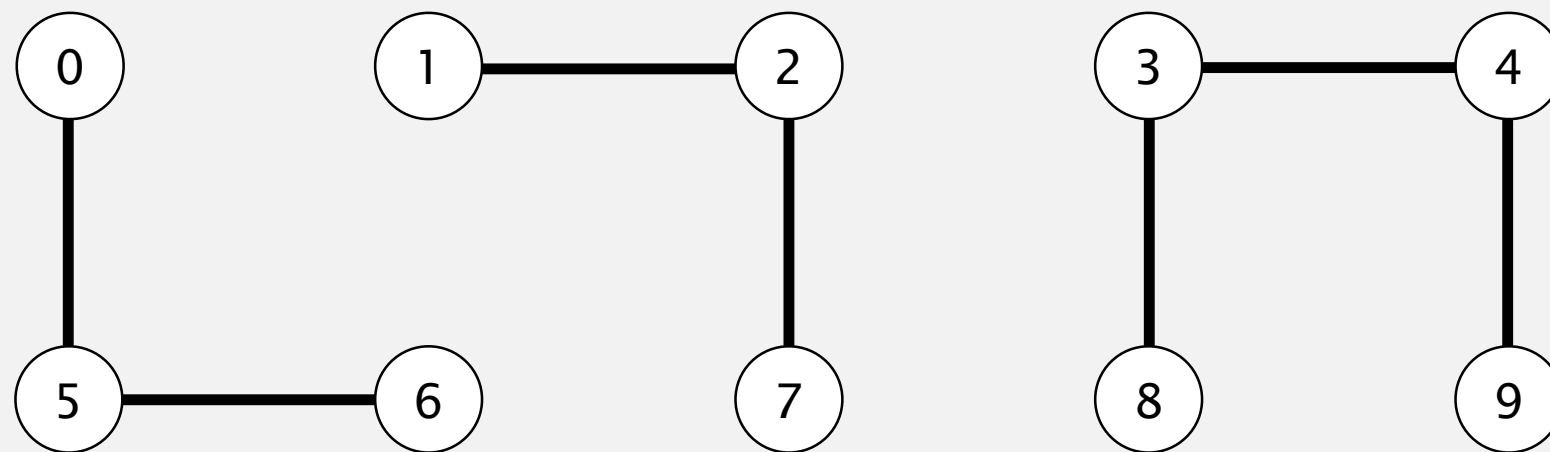
Data structure.

- Integer array `id[]` of length n .
- Interpretation: `id[p]` is the id of the component containing p .

if and only if

	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




Quick-find is too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find	connected
quick-find	N	N	1	1

order of growth of number of array accesses

Union is too expensive. It takes N^2 array accesses to process a sequence of N union operations on N objects.

quadratic


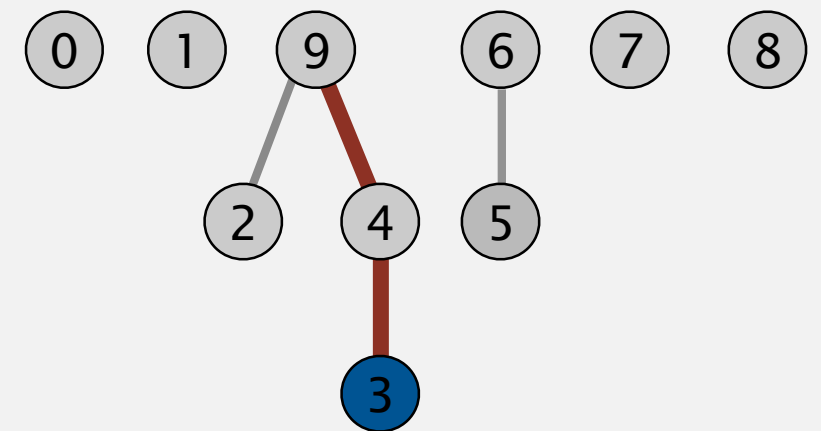
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of length n .
- Interpretation: `id[i]` is parent of i .
- **Root** of i is `id[id[id[...id[i]...]]]`.

	0	1	2	3	4	5	6	7	8	9
<code>id[]</code>	0	1	9	4	9	6	6	7	8	9

keep going until it doesn't change
(algorithm ensures no cycles)



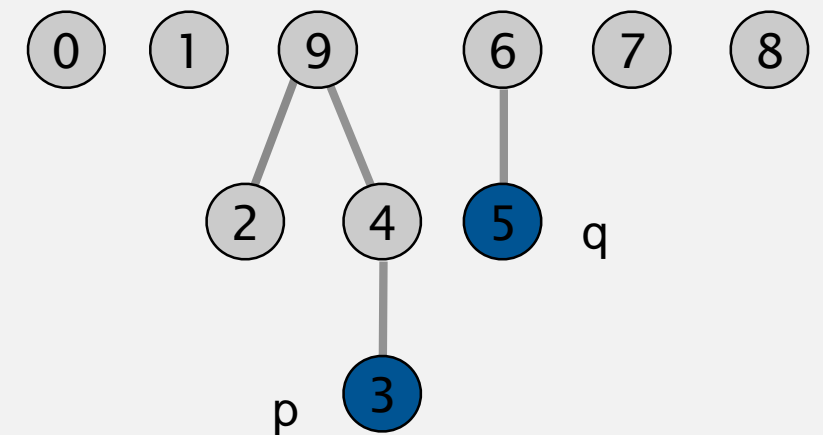
parent of 3 is 4
root of 3 is 9

Quick-union [lazy approach]

Data structure.

- Integer array $id[]$ of length n .
- Interpretation: $id[i]$ is parent of i .
- Root of i is $id[id[id[...id[i]...]]]$.

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



root of 3 is 9

root of 5 is 6

3 and 5 are not connected

Find. What is the root of p ?

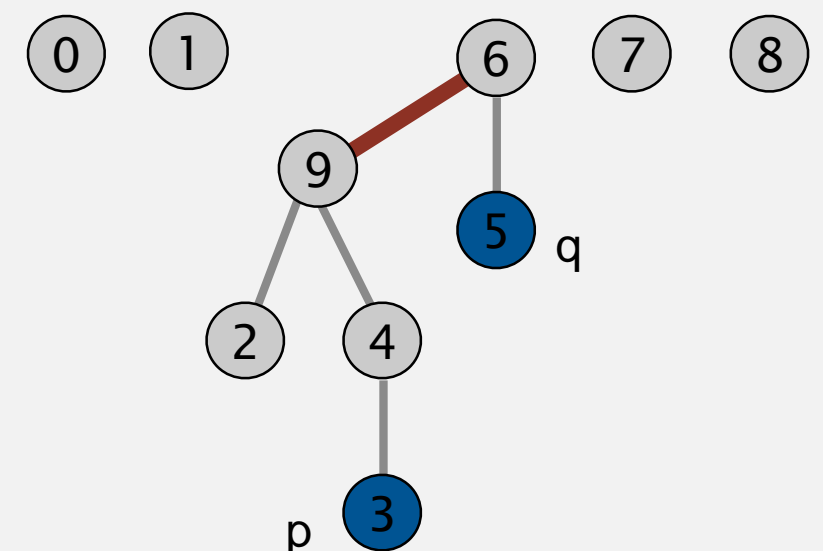
Connected. Do p and q have the same root?

Union. To merge components containing p and q , set the id of p 's root to the id of q 's root.

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



only one value changes



Quick-union is also too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N^\dagger	N	N

← worst case

\dagger includes cost of finding roots

Quick-find defect.

- Union too expensive (N array accesses).

Quick-union defect.

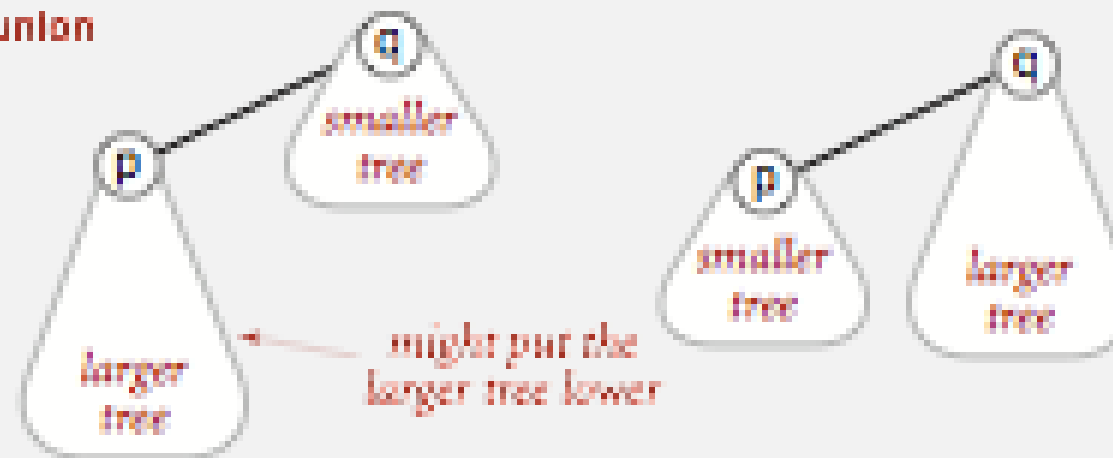
- Trees can get tall.
- Find/connected too expensive (could be N array accesses).

Improvement 1: weighting

Weighted quick-union.

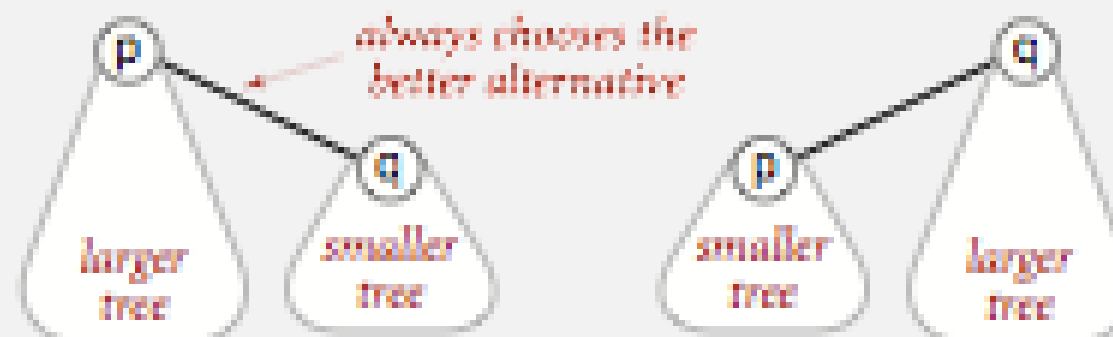
- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.

quick-union



reasonable alternatives:
union by height or "rank"

weighted



Weighted quick-union demo



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

Weighted quick-union demo

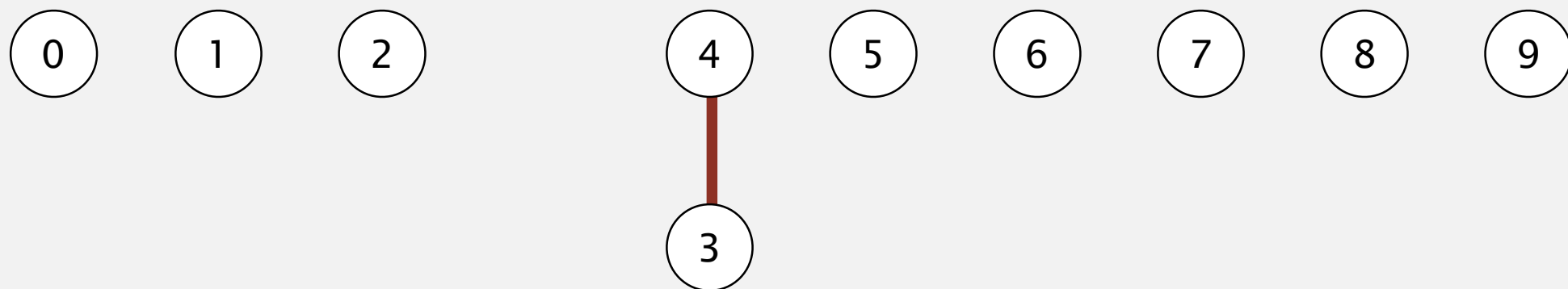
union(4, 3)



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

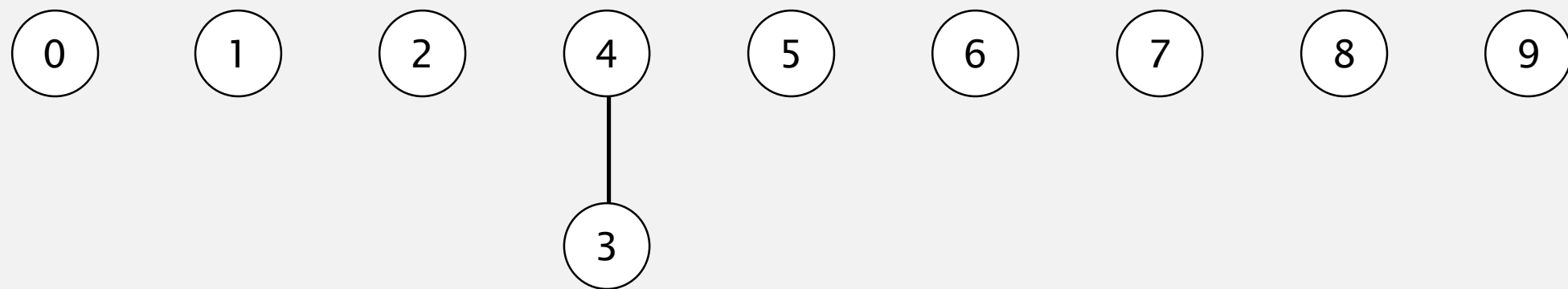
Weighted quick-union demo

union(4, 3)



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

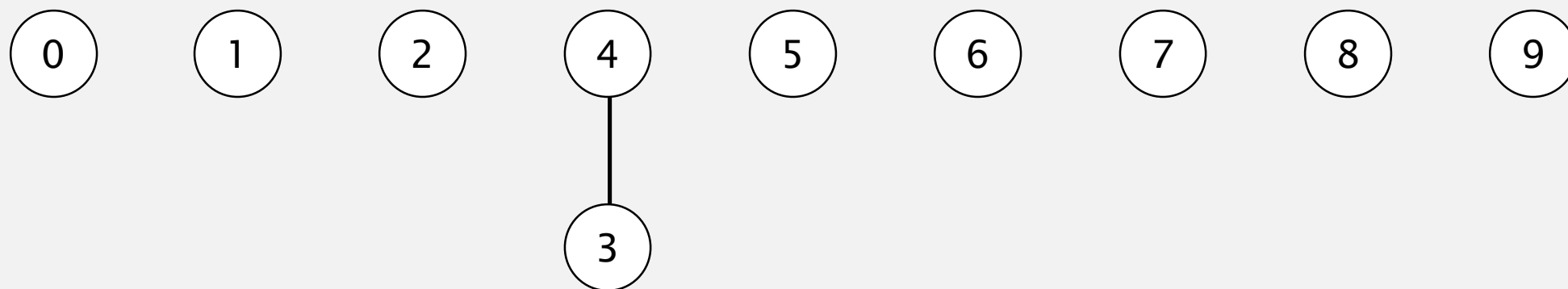
Weighted quick-union demo



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

Weighted quick-union demo

union(3, 8)

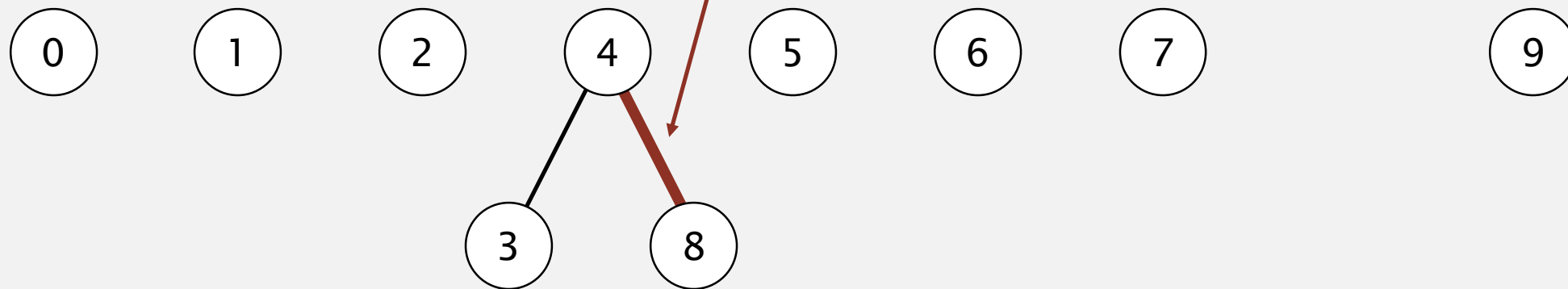


	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	4	4	5	6	7	8	9

Weighted quick-union demo

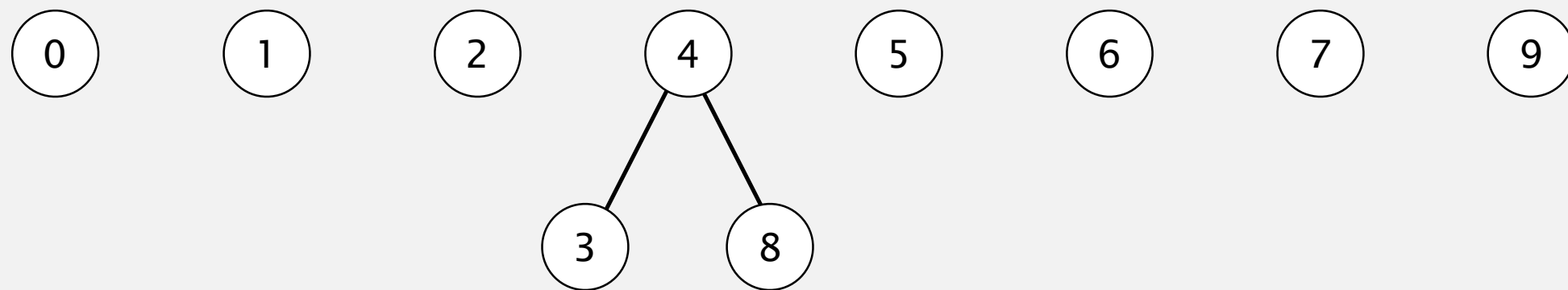
union(3, 8)

weighting: make 8 point to 4 (instead of 4 to 8)



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

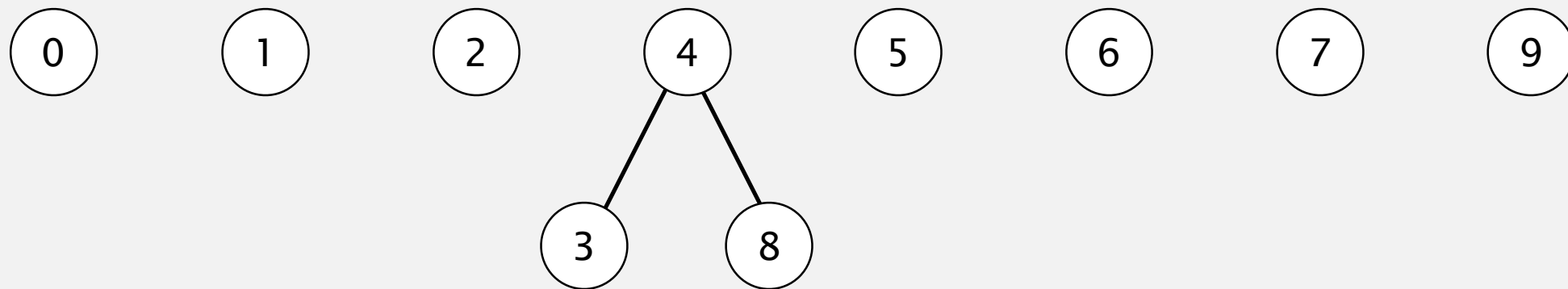
Weighted quick-union demo



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

Weighted quick-union demo

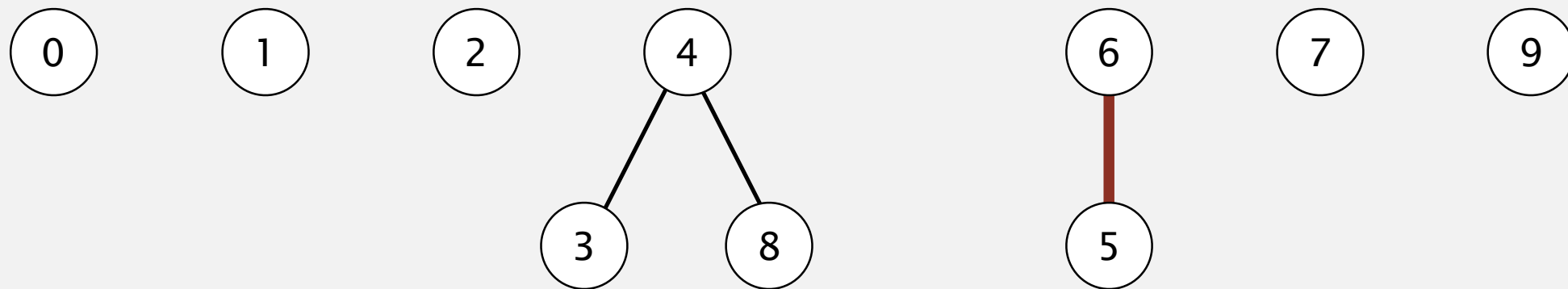
union(6, 5)



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

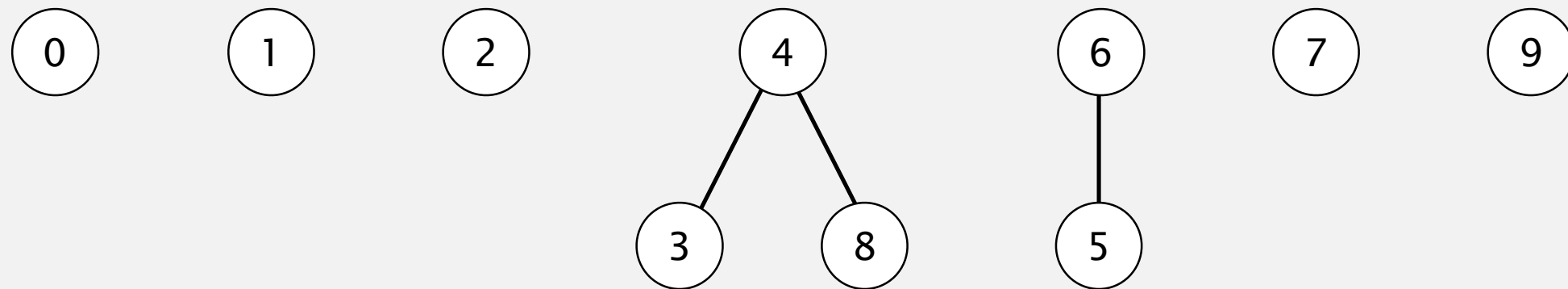
Weighted quick-union demo

union(6, 5)



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

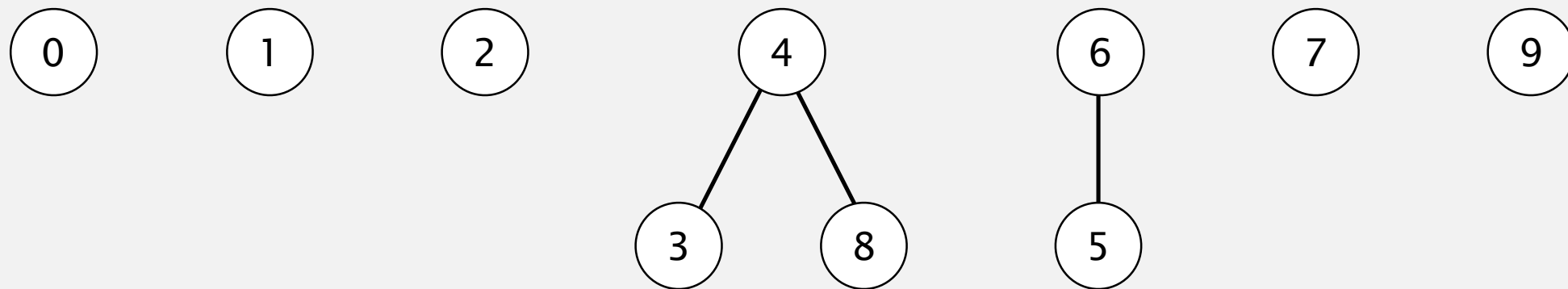
Weighted quick-union demo



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

Weighted quick-union demo

union(9, 4)

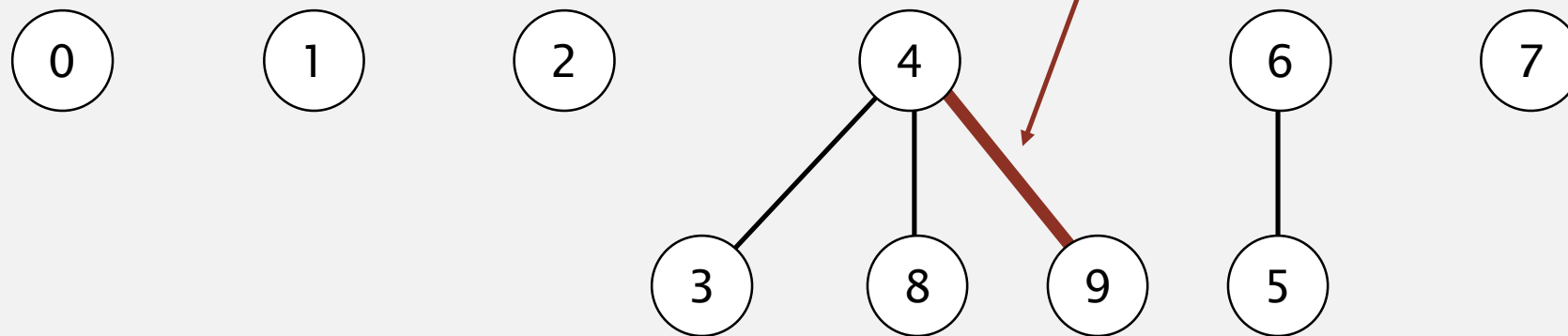


	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	4	4	6	6	7	4	9

Weighted quick-union demo

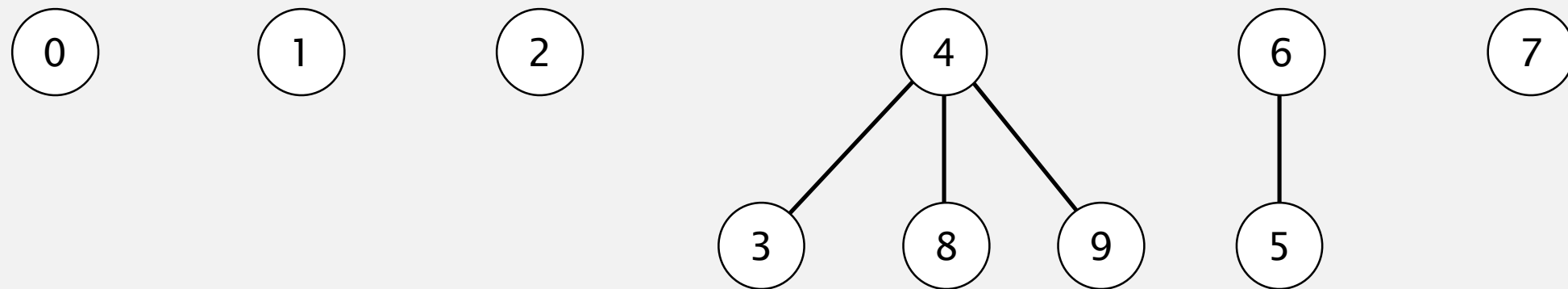
union(9, 4)

weighting: make 9 point to 4



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

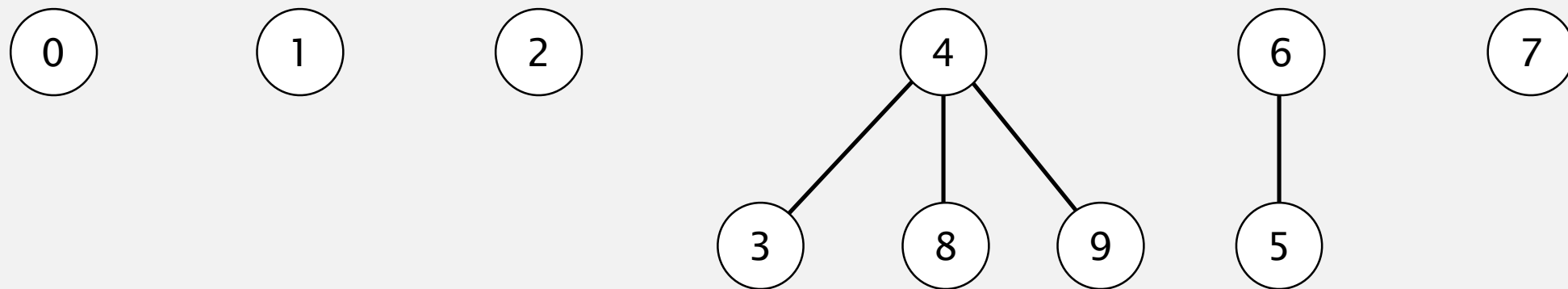
Weighted quick-union demo



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

Weighted quick-union demo

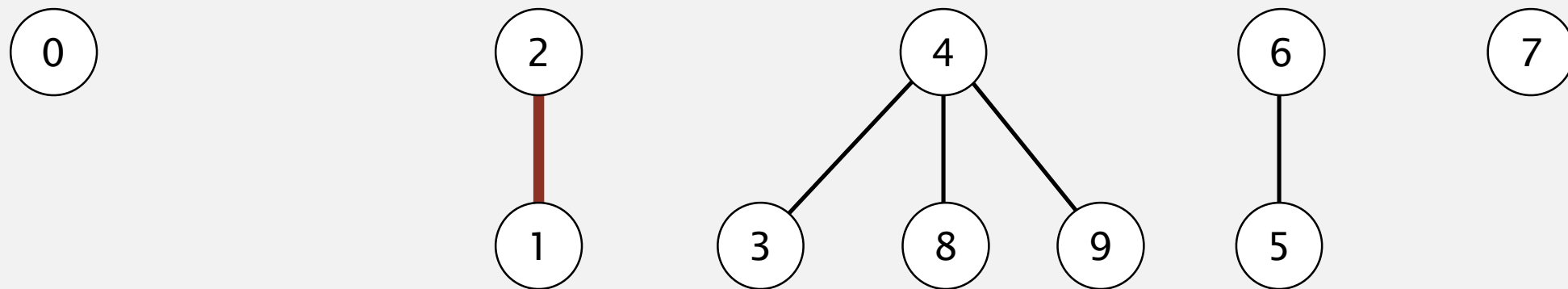
union(2, 1)



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

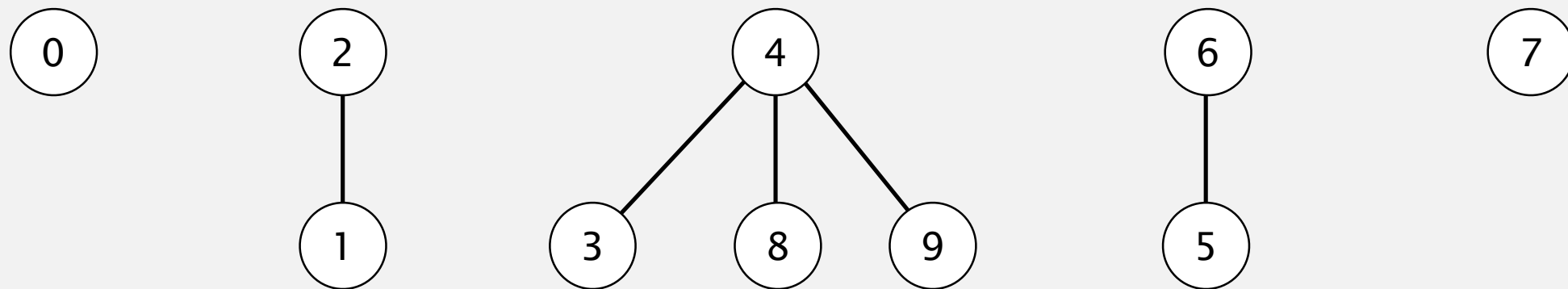
Weighted quick-union demo

union(2, 1)



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

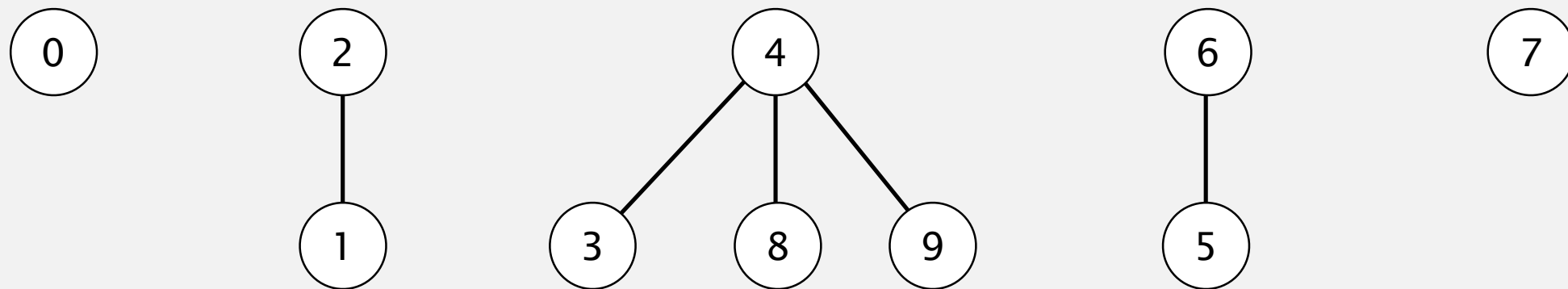
Weighted quick-union demo



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

Weighted quick-union demo

union(5, 0)

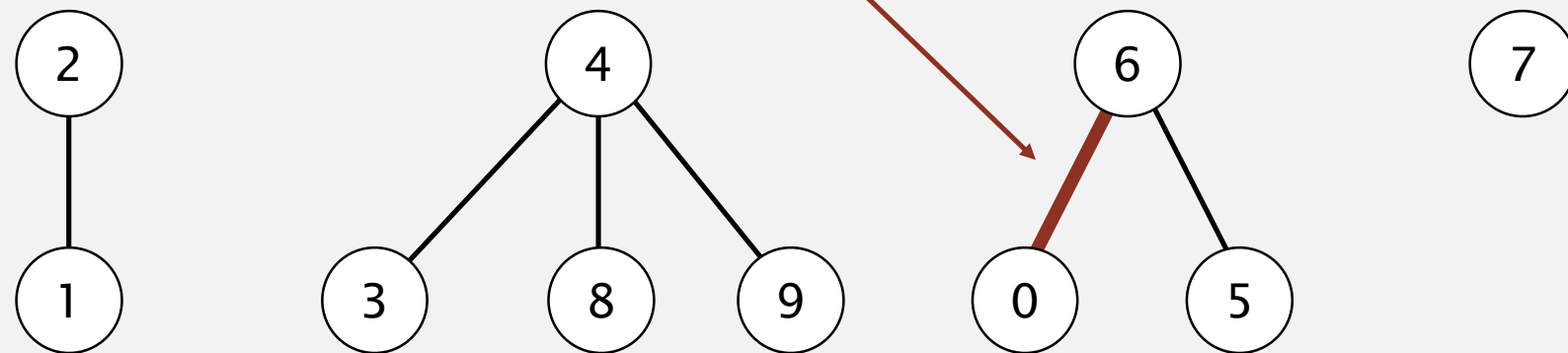


	0	1	2	3	4	5	6	7	8	9
id[]	0	2	2	4	4	6	6	7	4	4

Weighted quick-union demo

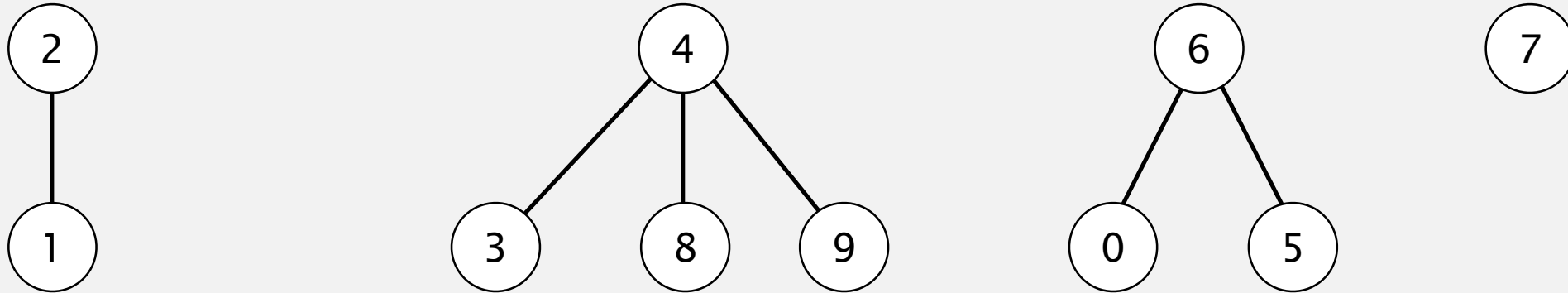
union(5, 0)

weighting: make 0 point to 6 (instead of 6 to 0)



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

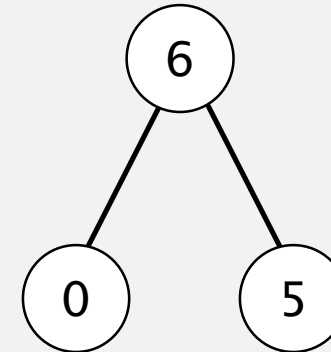
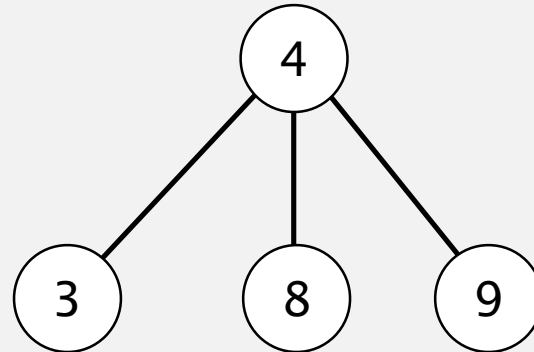
Weighted quick-union demo



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

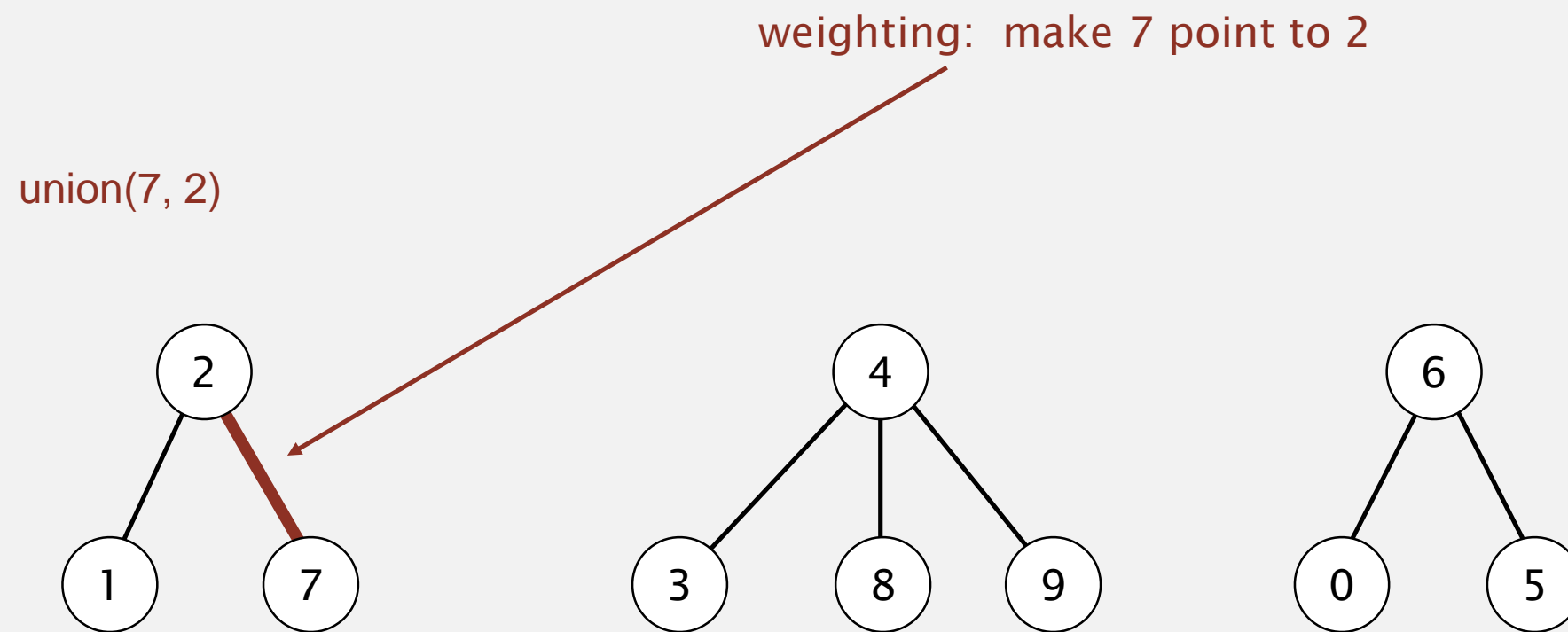
Weighted quick-union demo

union(7, 2)



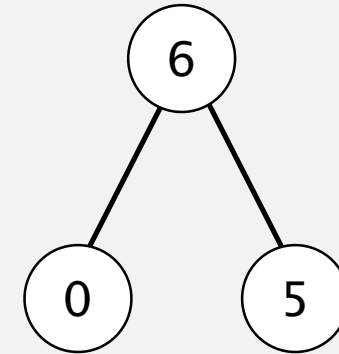
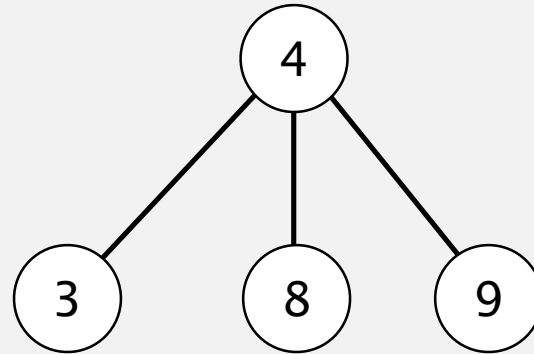
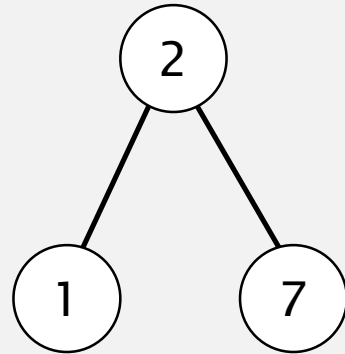
	0	1	2	3	4	5	6	7	8	9
id[]	6	2	2	4	4	6	6	7	4	4

Weighted quick-union demo



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

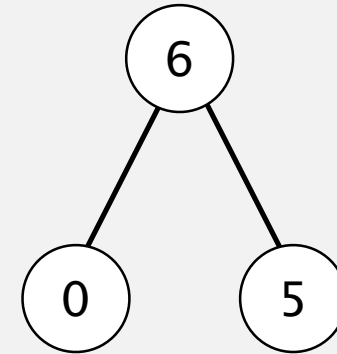
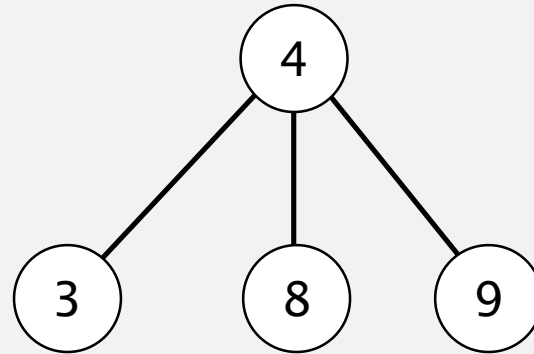
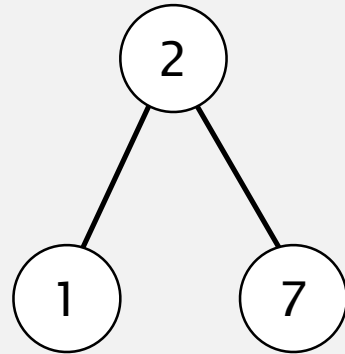
Weighted quick-union demo



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

Weighted quick-union demo

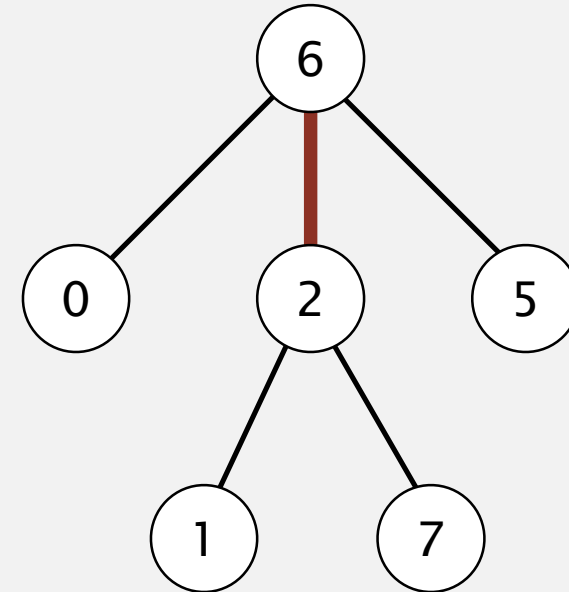
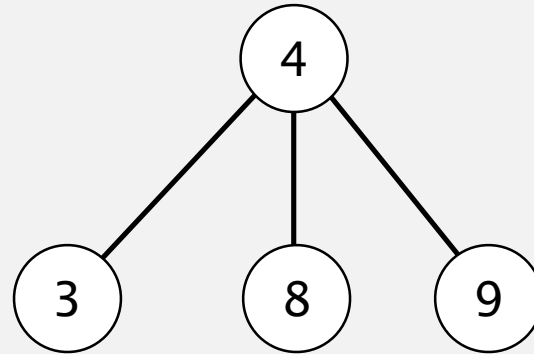
union(6, 1)



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

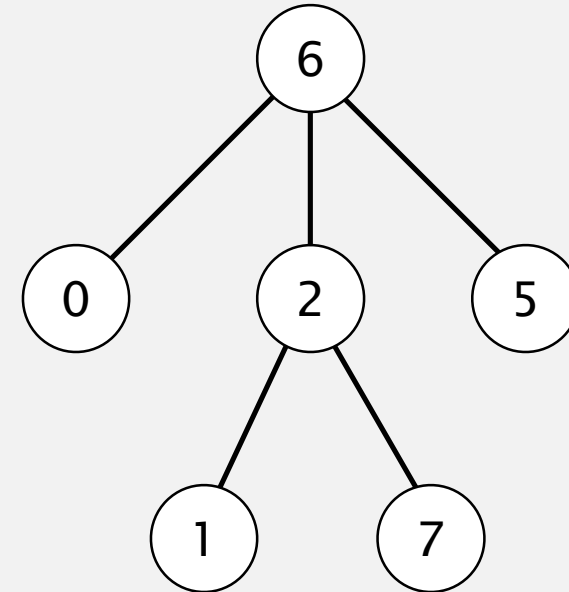
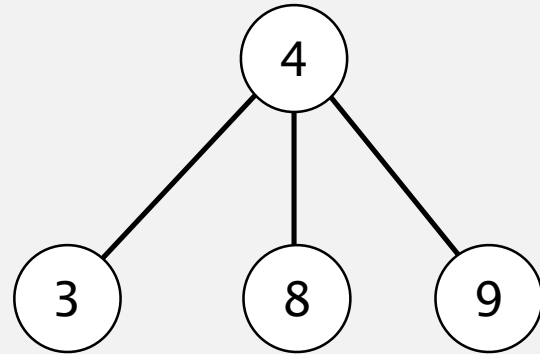
Weighted quick-union demo

union(6, 1)



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

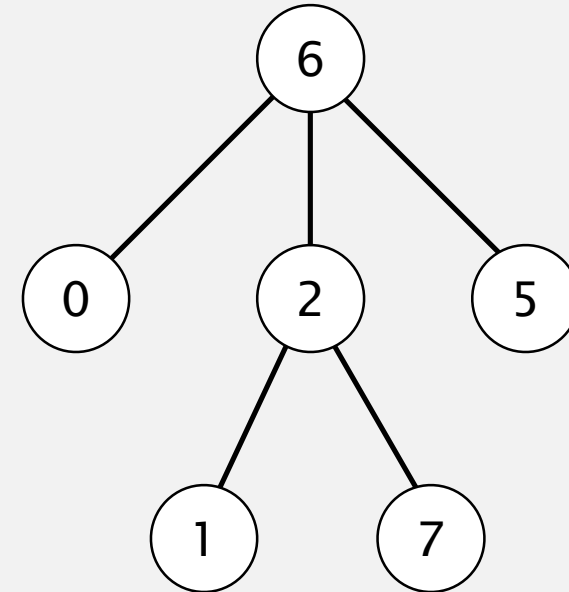
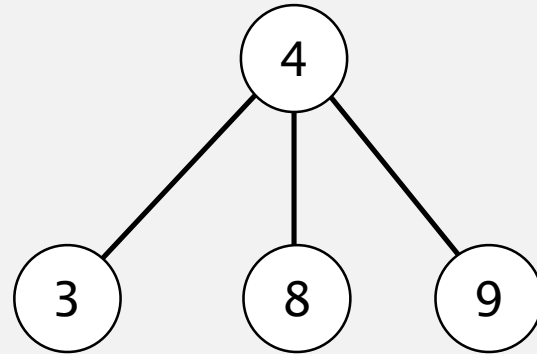
Weighted quick-union demo



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

Weighted quick-union demo

union(7, 3)

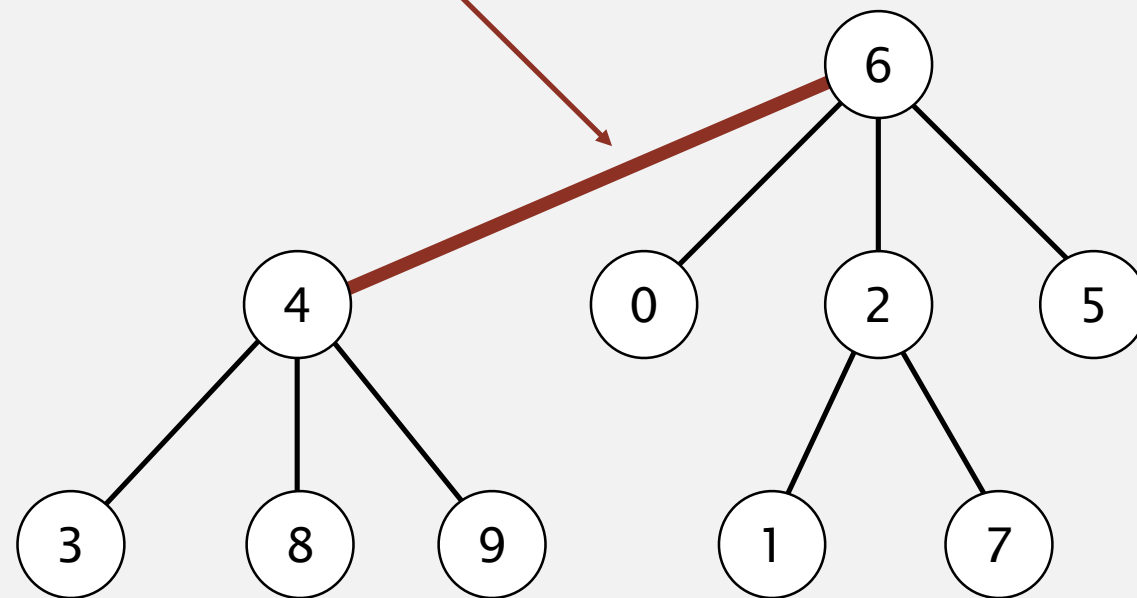


	0	1	2	3	4	5	6	7	8	9
id[]	6	2	6	4	4	6	6	2	4	4

Weighted quick-union demo

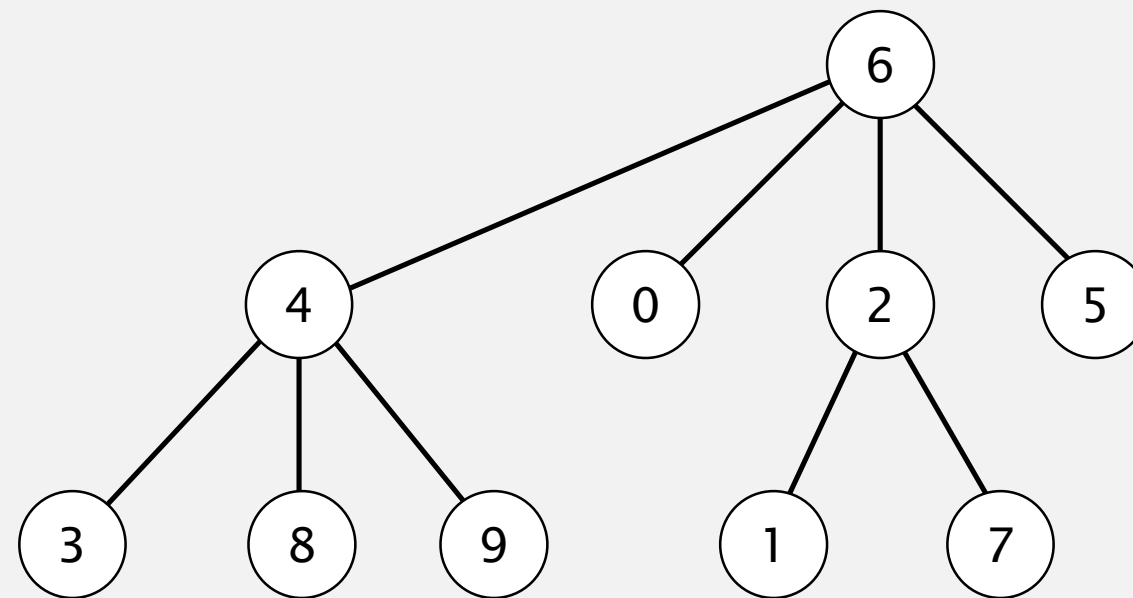
union(7, 3)

weighting: make 4 point to 6 (instead of 6 to 4)



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

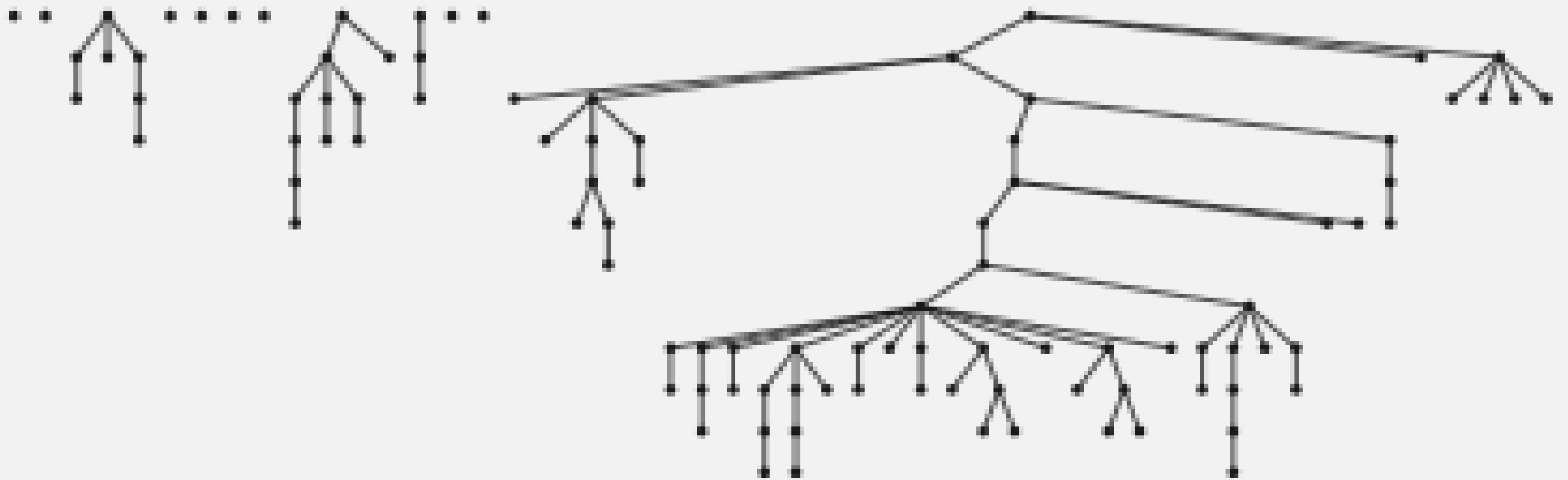
Weighted quick-union demo



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

Quick-union and weighted quick-union example

quick-union



average distance to root: 5.11

weighted



average distance to root: 1.51

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

Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array `sz[i]` to count number of objects in the tree rooted at `i`.

Find/connected. Identical to quick-union.

Union. Modify quick-union to:

- Link root of smaller tree to root of larger tree.
- Update the `sz[]` array.

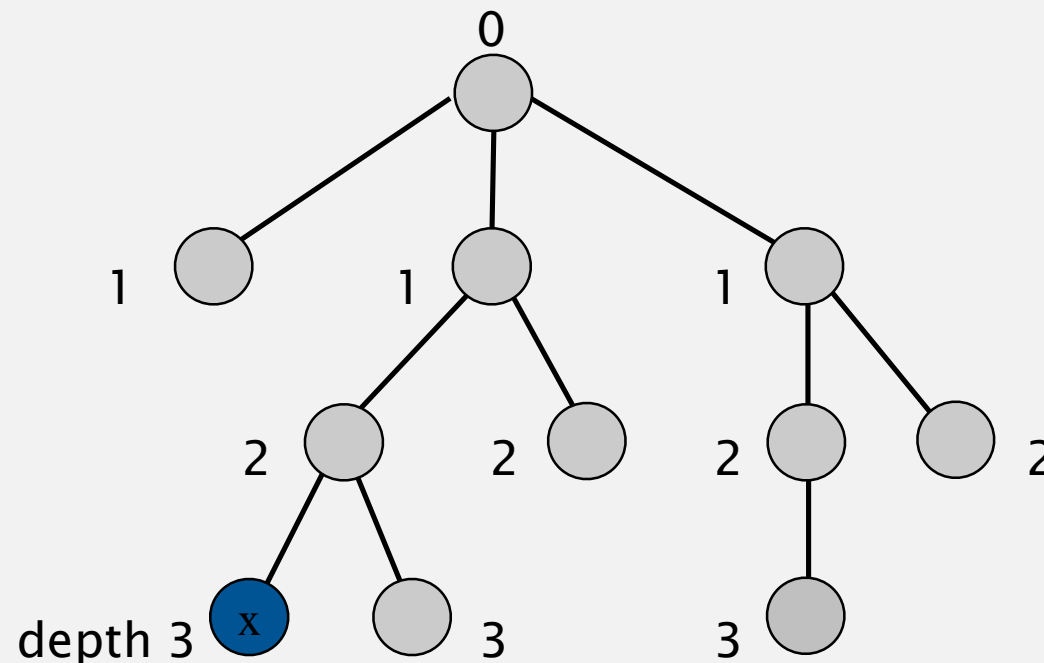
```
int i = find(p);
int j = find(q);
if (i == j) return;
if (sz[i] < sz[j])
    { id[i] = j; sz[j] += sz[i]; }
else
    { id[j] = i; sz[i] += sz[j]; }
```

Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p .
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most $\log n$. ↙ $\log = \text{base-2 logarithm}$



$N = 11$
 $\text{depth}(x) = 3 \leq \log n$

Weighted quick-union analysis

Running time.

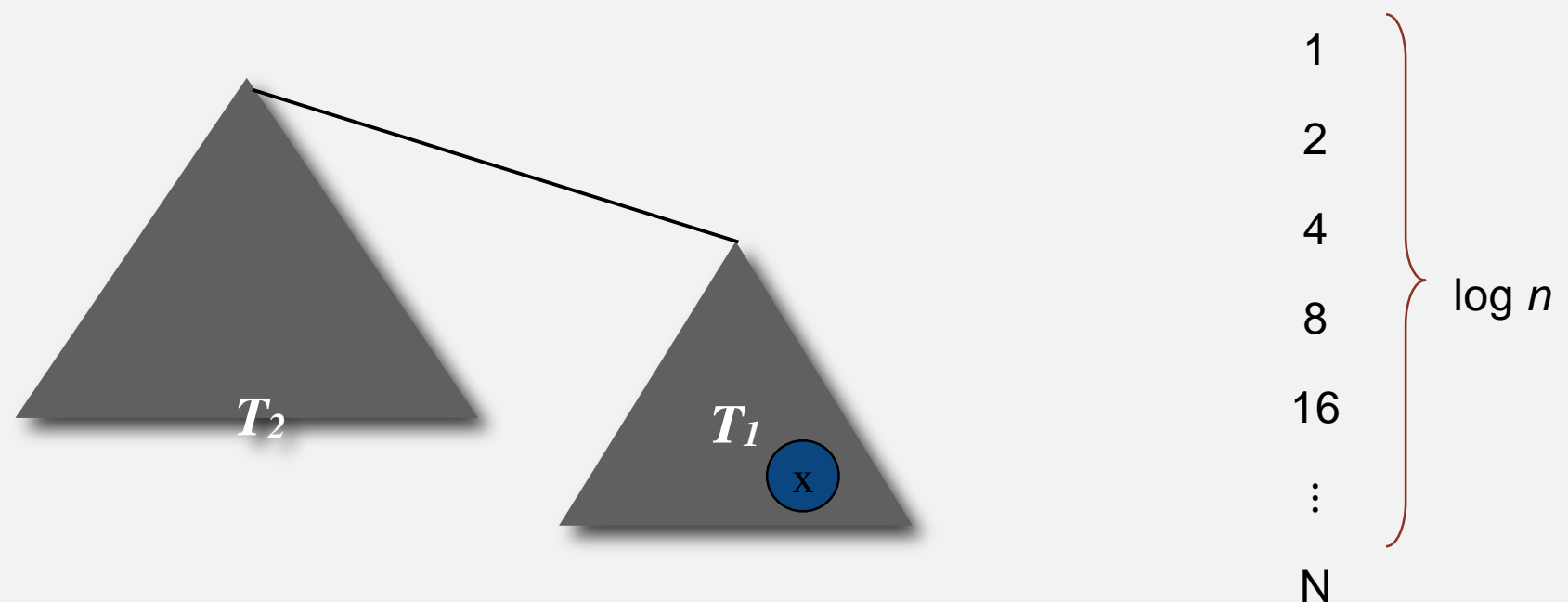
- Find: takes time proportional to depth of p .
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most $\log n$. ↙ $\log = \text{base-2 logarithm}$

Pf. What causes the depth of object x to increase?

Increases by 1 when tree T_1 containing x is merged into another tree T_2 .

- The size of the tree containing x at least doubles since $|T_2| \geq |T_1|$.
- Size of tree containing x can double at most $\log n$ times. Why?



Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p .
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most $\log n$.

algorithm	initialize	union	find	connected
quick-find	n	n	1	1
quick-union	n	n^\dagger	n	n
weighted QU	n	$\log n^\dagger$	$\log n$	$\log n$

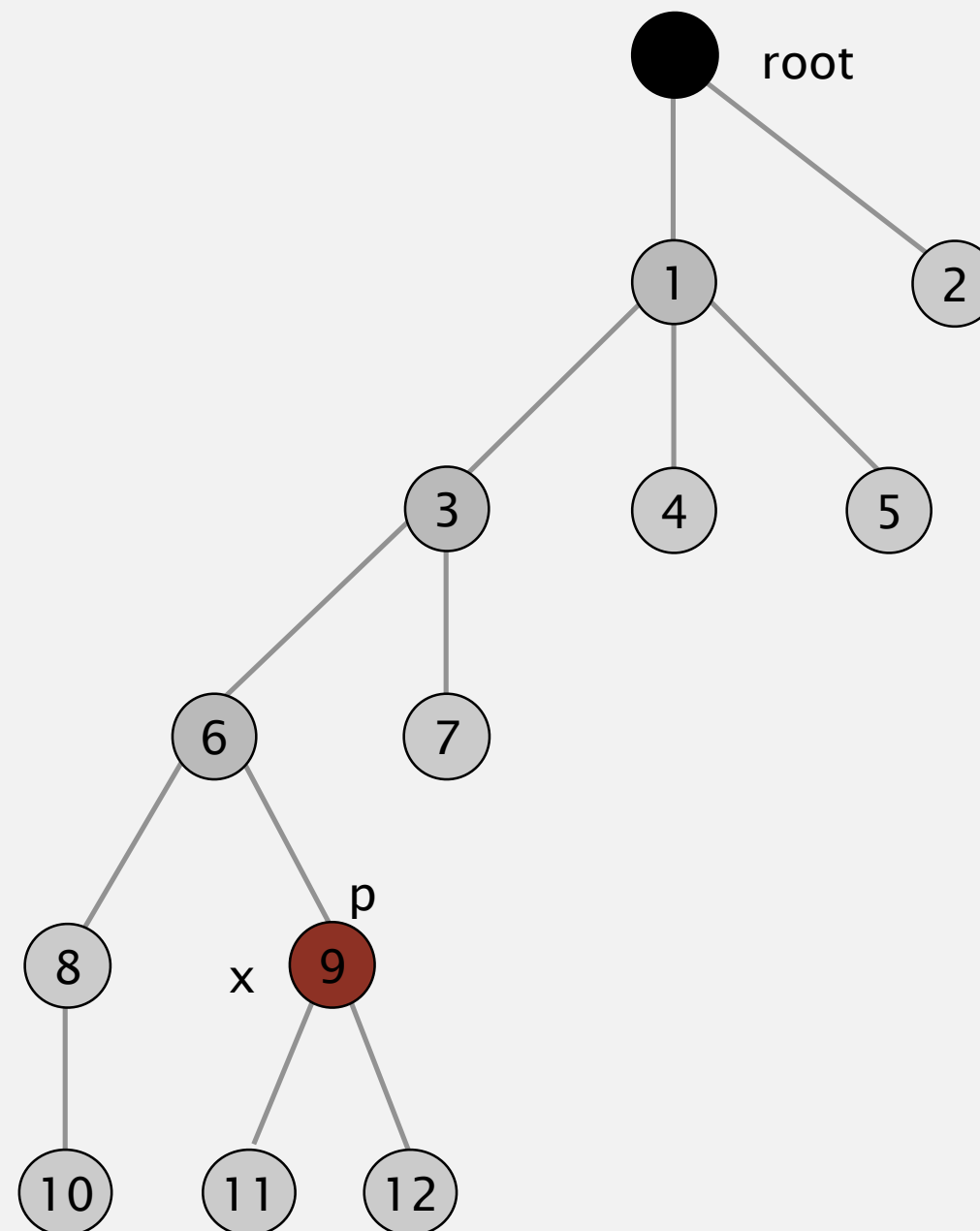
\dagger includes cost of finding roots

Q. Stop at guaranteed acceptable performance?

A. No, easy to improve further.

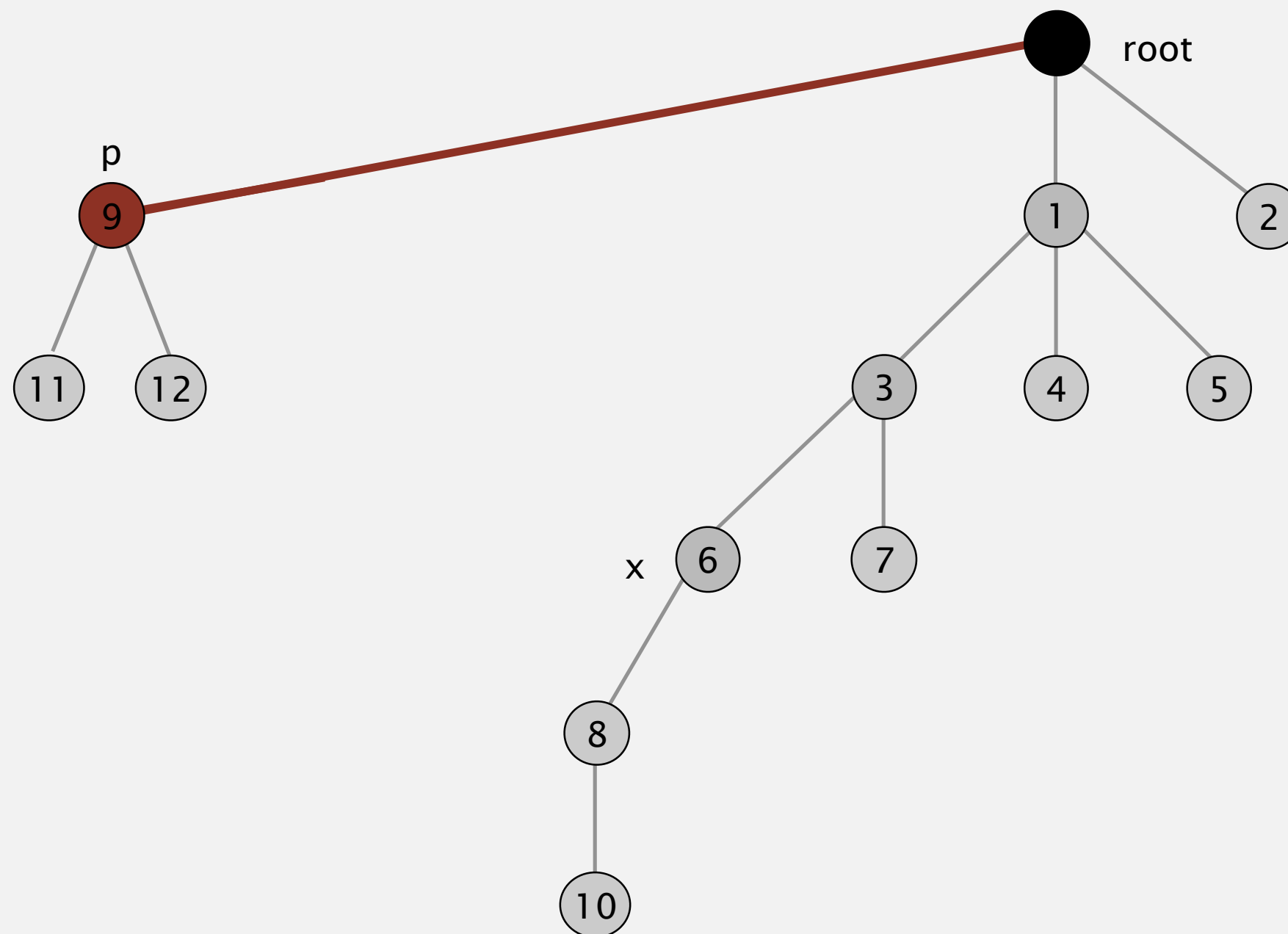
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the $\text{id}[]$ of each examined node to point to that root.



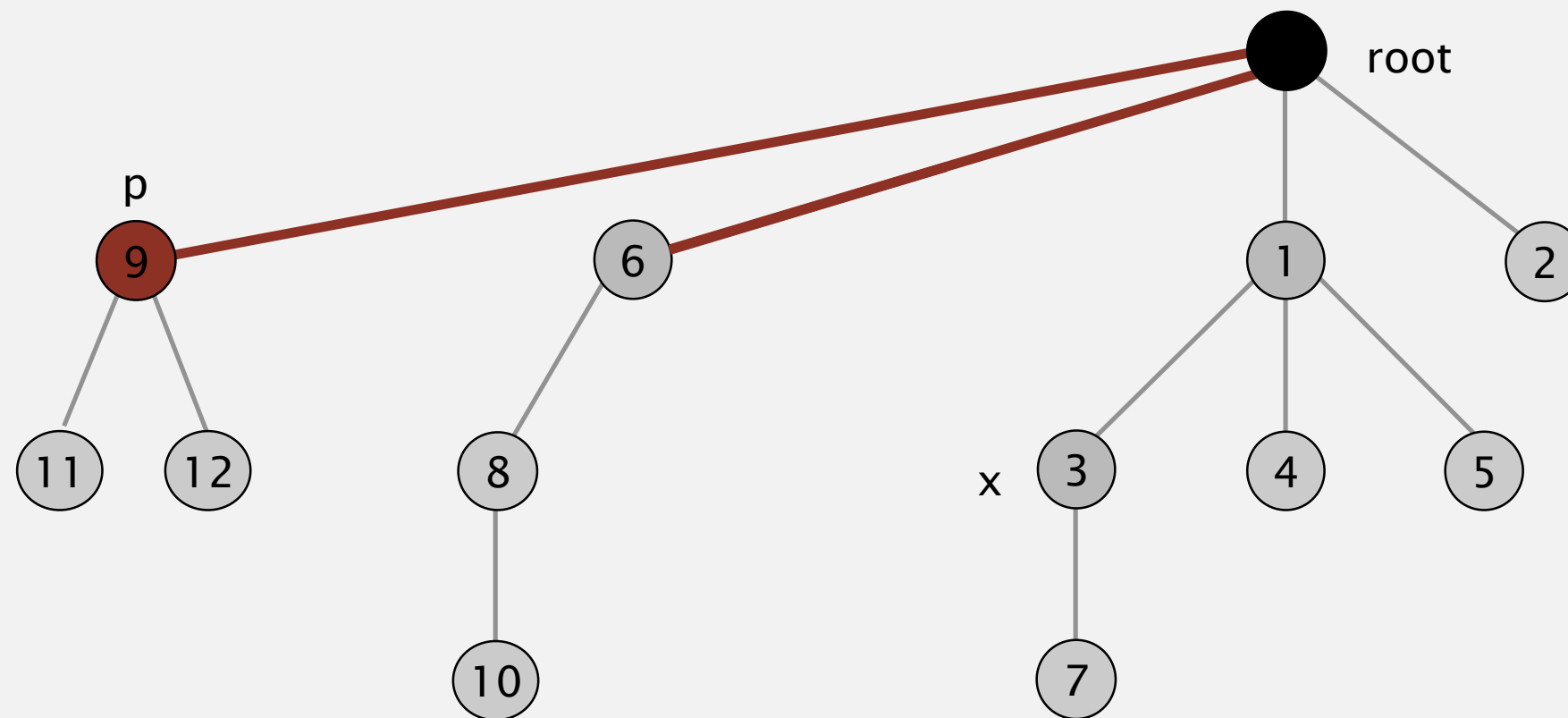
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the `id[]` of each examined node to point to that root.



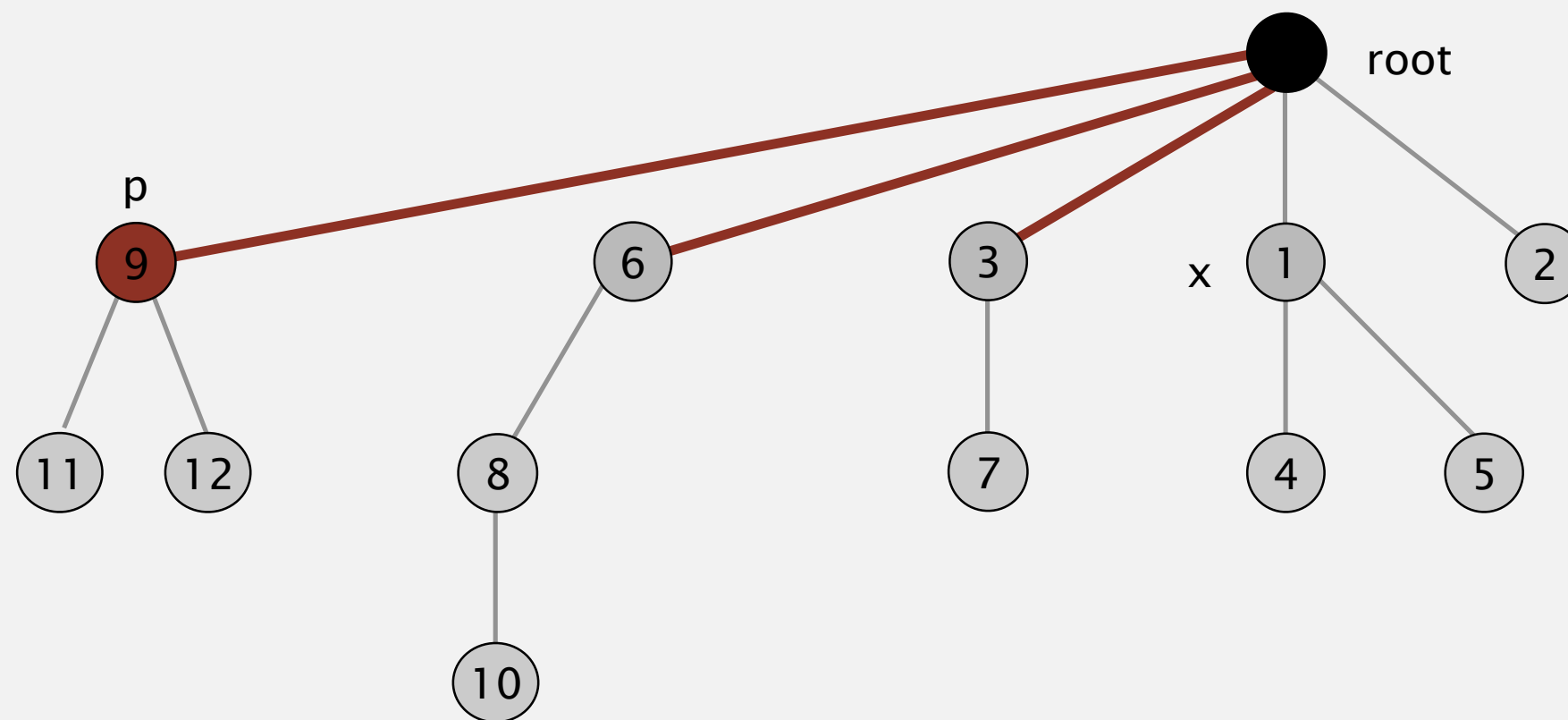
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the $\text{id}[]$ of each examined node to point to that root.



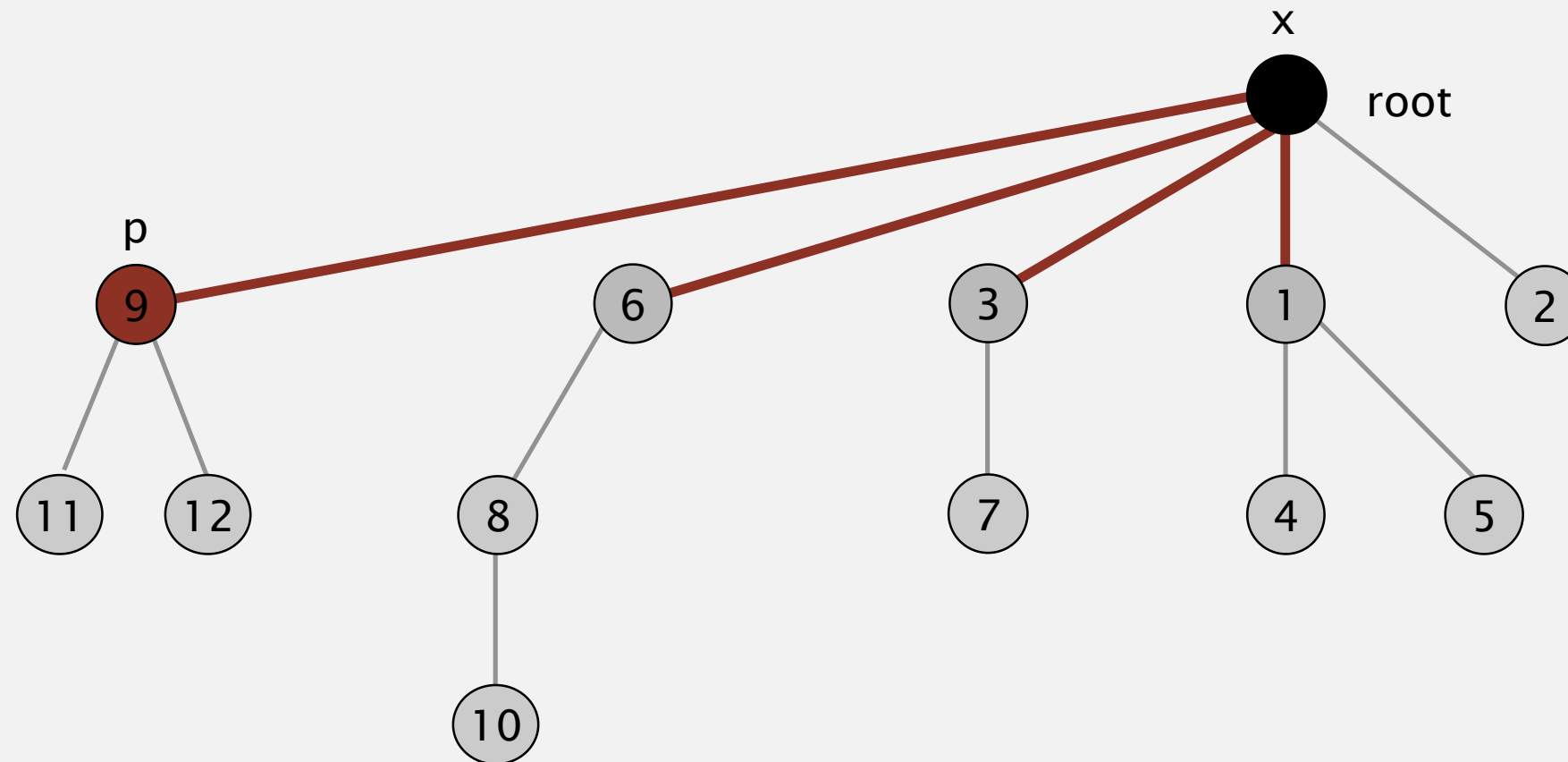
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the $\text{id}[]$ of each examined node to point to that root.



Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the `id[]` of each examined node to point to that root.



Bottom line. Now, `find()` has the side effect of compressing the tree.

Path compression: Java implementation

Two-pass implementation: add second loop to find() to set the id[] of each examined node to the root.

Simpler one-pass variant (path halving): Make every other node in path point to its grandparent.

```
public int find(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]];
        i = id[i];
    }
    return i;
}
```

← only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression: amortized analysis

Proposition. [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of m union–find ops on n objects takes $O(n + m \log^* n)$ array accesses.

- Analysis can be improved to $n + m \alpha(n)$.
- Simple algorithm with fascinating mathematics.

n	$\log^* n$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

iterated log function

Linear-time algorithm for m union-find ops on n objects?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

Amazing fact. [Fredman-Saks] No linear-time algorithm exists.

 in "cell-probe" model of computation

$\log^*(n)$: The iterated logarithm

- $\log^*(n) = \min\{i : t(i) \geq n\}$ is the inverse of the *tower-of-twos function* $t(i)$ where

$$t(i) = \begin{cases} 1 & \text{if } i = 0 \\ 2^{t(i-1)} & \text{if } i \geq 1 \end{cases}$$

$\alpha(n)$: inverse of Ackerman function $A(n)$

$\alpha(n) = \min\{m : A(m) \geq n\}$ where $A(n) = A_n(n)$ and

$$A_i(n) = \begin{cases} 2n & \text{for } i = 0 \text{ and } n \geq 0 \\ A_{i-1}(2) & \text{for } i \geq 1 \text{ and } n = 1 \\ A_{i-1}(A_i(n-1)) & \text{for } i \geq 1 \text{ and } n \geq 2 \end{cases}$$

$\alpha(n)$: inverse Ackerman function

$$\begin{aligned} A(2) &= A_2(2) = A_1(A_2(1)) = A_1(A_1(2)) \\ &= A_1(A_0(A_1(1))) = A_1(A_0(2^2)) \\ &= A_1(2 \cdot 2^2) = A_1(2^3) = A_0(A_1(2^3 - 1)) \\ &= A_0(A_0(A_1(2^3 - 2))) = A_0(A_0(A_0(A_1(2^3 - 3)))) \\ &= A_0(A_0(A_0(A_0(A_1(2^3 - 4))))) \\ &= A_0(A_0(A_0(A_0(A_0(A_1(2^3 - 5)))))) \\ &= A_0(A_0(A_0(A_0(A_0(A_0(A_1(2^3 - 6))))))) \\ &= A_0(A_0(A_0(A_0(A_0(A_0(A_0(A_1(2^3 - 7)))))))) \end{aligned}$$

$\alpha(n)$: inverse Ackerman function

$$= A_0(A_0(A_0(A_0(A_0(A_0(A_1(1)))))))$$

$$= A_0(A_0(A_0(A_0(A_0(A_0(2^2))))))$$

$$= A_0(A_0(A_0(A_0(A_0(2^3)))))$$

$$= A_0(A_0(A_0(A_0(2^4))))$$

$$= A_0(A_0(A_0(2^5)))$$

$$= A_0(A_0(2^6)) = A_0(2^7)$$

$$= A_0(2^8) = 2^9$$

Summary

Key point. Weighted quick union (and/or path compression) makes it possible to solve problems that could not otherwise be addressed.

algorithm	worst-case time
quick-find	$O(m\ n)$
quick-union	$O(m\ n)$
weighted QU	$O(n + m \log n)$
QU + path compression	$O(n + m \log n)$
weighted QU + path compression	$O(n + m \log^* n)$

order of growth for m union-find operations on a set of n objects

Ex. [10^9 unions and finds with 10^9 objects]

- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

Kruskal's algorithm: Java implementation

```
public class KruskalMST
{
    private Queue<Edge> mst = new Queue<Edge>();

    public KruskalMST(EdgeWeightedGraph G)
    {
        MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());

        UF uf = new UF(G.V());
        while (!pq.isEmpty() && mst.size() < G.V()-1)
        {
            Edge e = pq.delMin();
            int v = e.either(), w = e.other(v);
            if (!uf.connected(v, w))
            {
                uf.union(v, w);
                mst.enqueue(e);
            }
        }
    }

    public Iterable<Edge> edges()
    { return mst; }
}
```

← build priority queue
(or sort)

← greedily add edges to MST

← edge v-w does not create cycle

← merge sets

← add edge to MST