https://algs4.cs.princeton.edu

1.5 Union–Find

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *path compaction*

https://algs4.cs.princeton.edu

1.5 Union–Find

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *path compaction*

Union–find data type

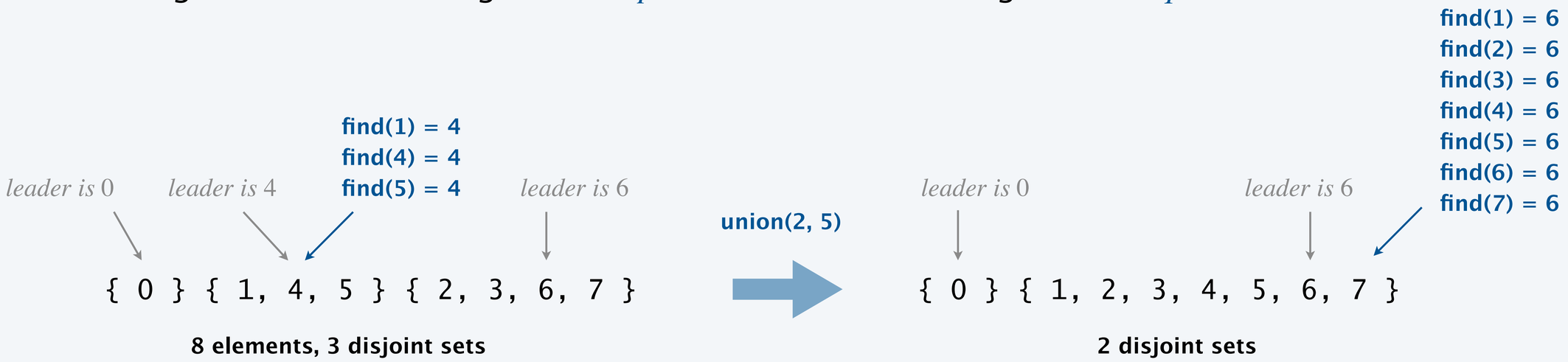
Disjoint sets. A collection of sets containing n elements, with each element in exactly one set.

Leader. Each set designates one of its elements as leader (to uniquely identify it).

no restriction on which element is designated leader
(but leader of a set can't change unless the set changes)

Find. Return the leader of the set containing element p. ← *main use case: are two elements in the same set ?*

Union. Merge the set containing element p with the set containing element q.



Union–find data type: API

Goal. Design an efficient union–find data type.

- Simplifying assumption: the n elements are named $0, 1, 2, \ldots, n-1$.
- The `union()` and `find()` operations can be intermixed.
- Number of elements n can be huge.
- Number of operations m can be huge.

`public class UF`	**description**
`UF(int n)`	*initialize with n singleton sets (each element in its own set)*
`void union(int p, int q)`	*merge sets containing elements p and q*
`int find(int p)`	*return the leader of set containing element p*

Union-find: poll 1

Suppose that immediately before `union(3, 5)`, the sets and leaders are as shown:

$\{ 0, 1, 4, 6 \}$ ⟵ $leader = 0$

$\{ 2, 3, 8 \}$ ⟵ $leader = 3$

$\{ 5, 7, 9 \}$ ⟵ $leader = 9$

Which of the following could be true immediately afterwards?

A. `find(6) == 6`

B. `find(3) == 8`

C. `find(6) == find(7)`

D. `find(5) != find(7)`

E. None of the above

Modeling a percolation system

Elements. Each of the n^2 sites is represented by an element.

Disjoint sets. Each set represents either a cluster of open sites or a singleton blocked site.

$n = 5$

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

open site

blocked site

{ 0, 1 }

{ 3, 8, 13, 14 }

{ 11, 15, 16, 17, 20, 21 }

{ 23, 24 }

{ 2 }

{ 4 }

⋮

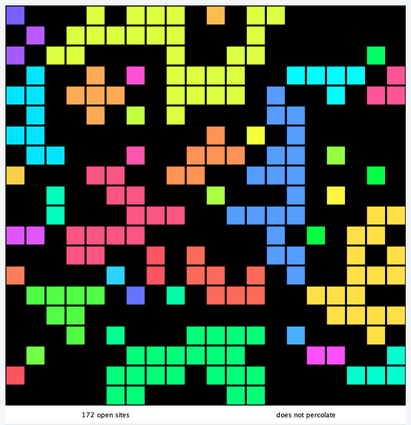
{ 22 }

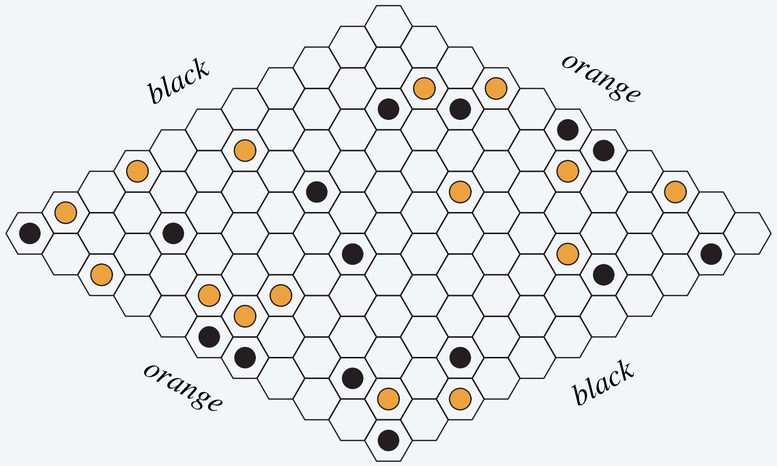
25 elements, 15 disjoint sets

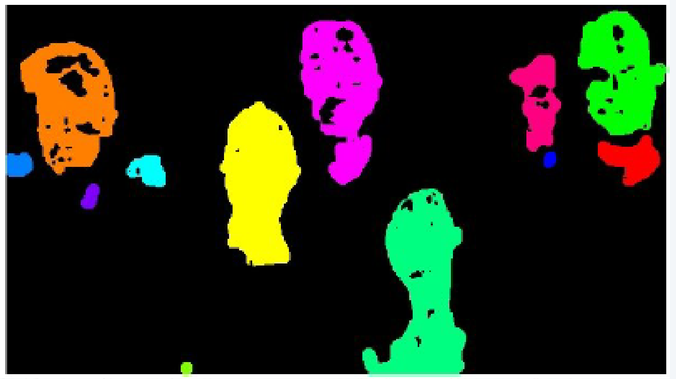
Union–find data type: applications

Disjoint sets can represent:

- Clusters of open sites in a percolation system. ⟵ *see Assignment 1 (Percolation)*
- Connected components in a graph. ⟵ *see Kruskal's algorithm (MST lecture)*
- Interlinked friends in a social network.
- Interconnected devices in a mobile network.
- Equivalent variable names in a Fortran program.
- Stones of the same color that adjoin in the game of Hex.
- Contiguous pixels corresponding to the same feature in a digital image.







https://algs4.cs.princeton.edu

1.5 Union–Find

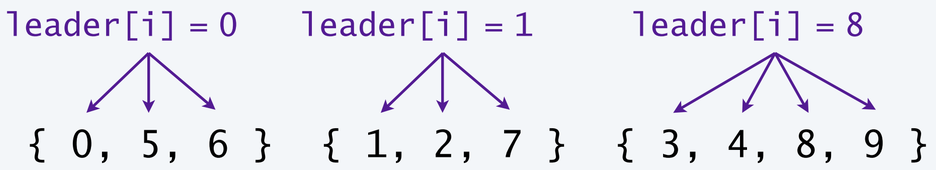
- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *path compaction*

Quick-find

Data structure.

- Integer array `leader[]` of length `n`.
- Interpretation: `leader[i]` is the leader of the set containing element `i`.

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



10 elements, 3 disjoint sets

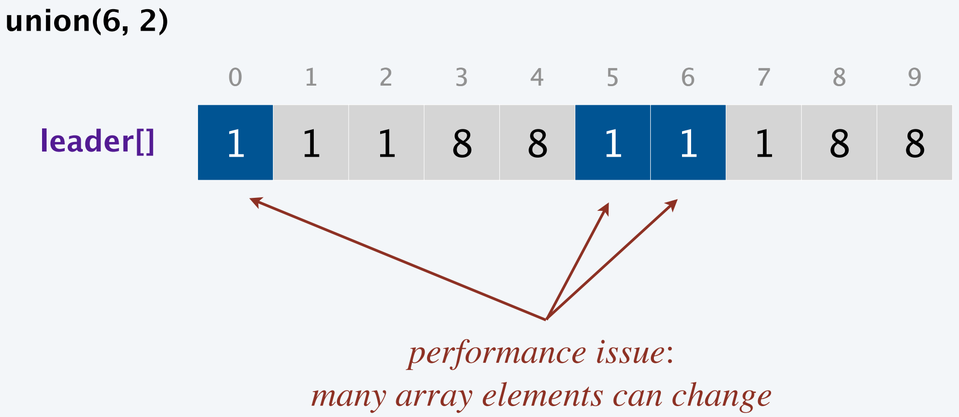
Q. How to implement `find(p)`?

A. Easy, just return `leader[p]`.

Quick-find

Data structure.

- Integer array `leader[]` of length `n`.
- Interpretation: `leader[i]` is the leader of the set containing element `i`.



Q. How to implement `union(p, q)`?

A. Change all array elements equal to `leader[p]` into `leader[q]`. ← *or vice versa*

Quick-find: Java implementation

```
public class QuickFindUF {
   private int[] leader;

   public QuickFindUF(int n) {
      leader = new int[n];
      for (int i = 0; i < n; i++)
         leader[i] = i;
   }

   public int find(int p) {
      return leader[p];
   }

   public void union(int p, int q) {
      int leaderP = leader[p];
      int leaderQ = leader[q];
      for (int i = 0; i < leader.length; i++)
         if (leader[i] == leaderP)
            leader[i] = leaderQ;
   }

}
```

initialize leader of each element to itself
(n array accesses)

return the leader of p
(1 array access)

change all array elements equal to `leader[p]` *into* `leader[q]`
(≥ n array accesses)

https://algs4.cs.princeton.edu/15uf/QuickFindUF.java.html

Quick-find is too slow

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

algorithm	initialize	union	find
quick-find	n	n	1

worst-case number of array accesses (up to constant factors)

Union is too expensive. Processing any sequence of m union operations on n elements takes $\geq mn$ array accesses.

quadratic in input size!

Ex. Performing 10^9 union operations on 10^9 elements might take 30 years.

https://algs4.cs.princeton.edu

1.5 Union–Find

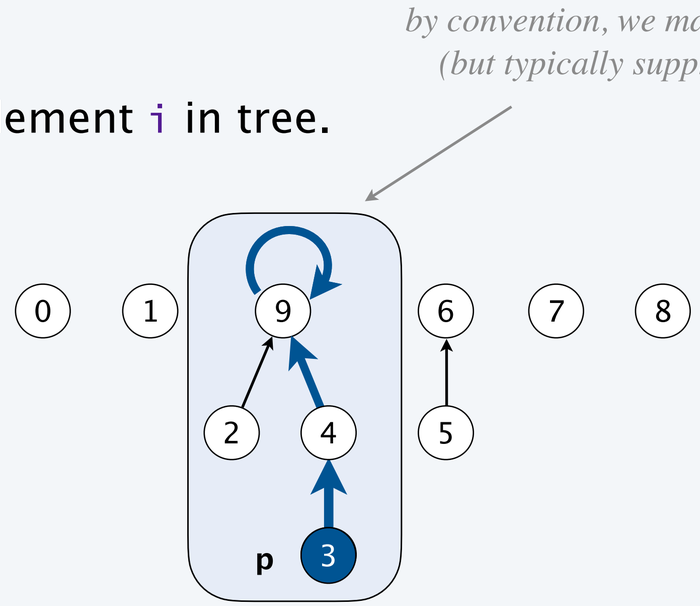
- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *path compaction*

Quick-union

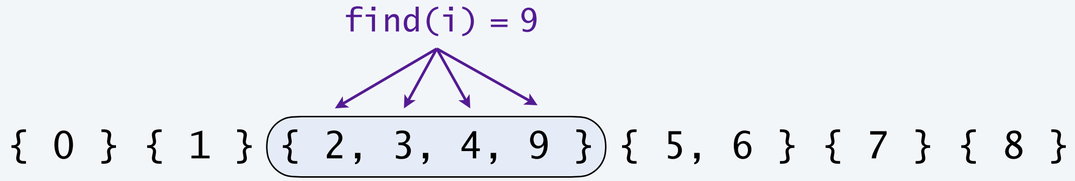
Data structure: Forest of trees.

- Interpretation: each rooted tree represents one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of element `i` in tree.

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



parent of 3 *is* 4
parent of 4 *is* 9
parent of 9 *is* 9
root of 3 *is* 9



10 elements, 6 disjoint sets (6 trees)

Q. How to implement `find()`?

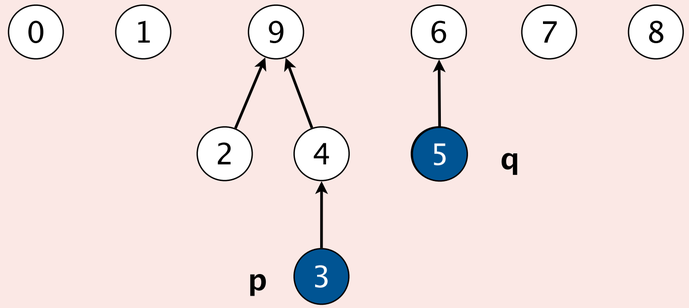
A. Use tree roots as leaders. Follow parent pointers until you reach the root.

Union–find: poll 2

Data structure: Forest of trees.

- Interpretation: each rooted tree represents one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of element `i` in tree.

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



Which option is not a valid way to implement `union(3, 5)` ?

A. Set `parent[6] = 9`

B. Set `parent[9] = 6`

C. Set `parent[3] = 5`

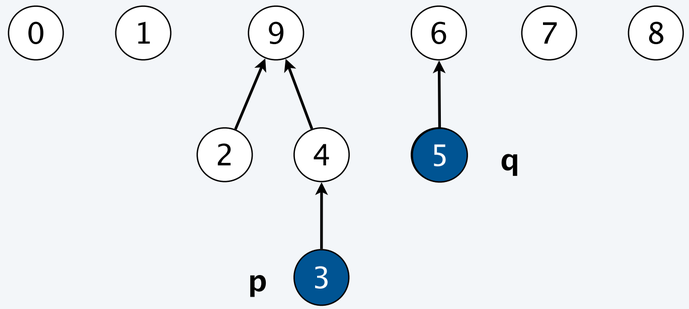
D. Set `parent[3] = parent[4] = parent[9] = 6`

Quick-union

Data structure: Forest of trees.

- Interpretation: each rooted tree represents one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of element `i` in tree.

	0	1	2	3	4	5	6	7	8	9
union(3, 5)	0	1	9	4	9	6	6	7	8	9



Q. How to implement `union(p, q)`?

A. Set the root of `p`'s tree to point to the root of `q`'s tree. ⟵ *or vice versa*

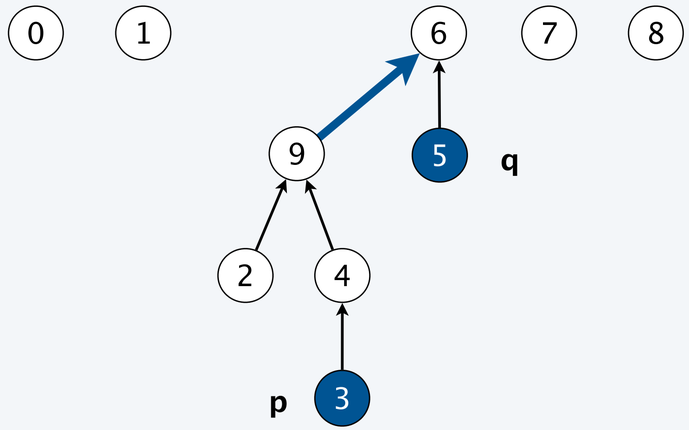
Quick-union

Data structure: Forest of trees.

- Interpretation: each rooted tree represents one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of element `i` in tree.

	0	1	2	3	4	5	6	7	8	9
union(3, 5)	0	1	9	4	9	6	6	7	8	6

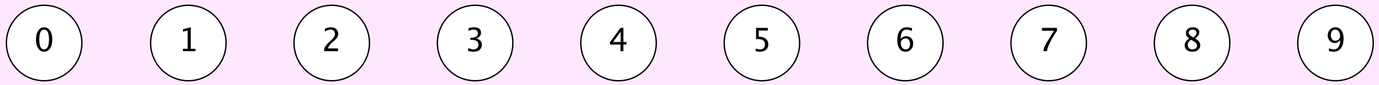
only one array element changes



Q. How to implement `union(p, q)`?

A. Set the root of `p`'s tree to point to the root of `q`'s tree. ⟵ *or vice versa*

Quick-union demo



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

Quick-union: Java implementation

```
public class QuickUnionUF {
   private int[] parent;

   public QuickUnionUF(int n) {
      parent = new int[n];
      for (int i = 0; i < n; i++)
          parent[i] = i;
   }

   public int find(int p) {
      while (p != parent[p])
          p = parent[p];
      return p;
   }

   public void union(int p, int q) {
      int rootP = find(p);
      int rootQ = find(q);
      parent[rootP] = rootQ;
   }
}
```

set parent of each element to itself (to create forest of n singleton trees)

follow parent pointers until reach root; return resulting root

link root of p to root of q

https://algs4.cs.princeton.edu/15uf/QuickUnionUF.java.html

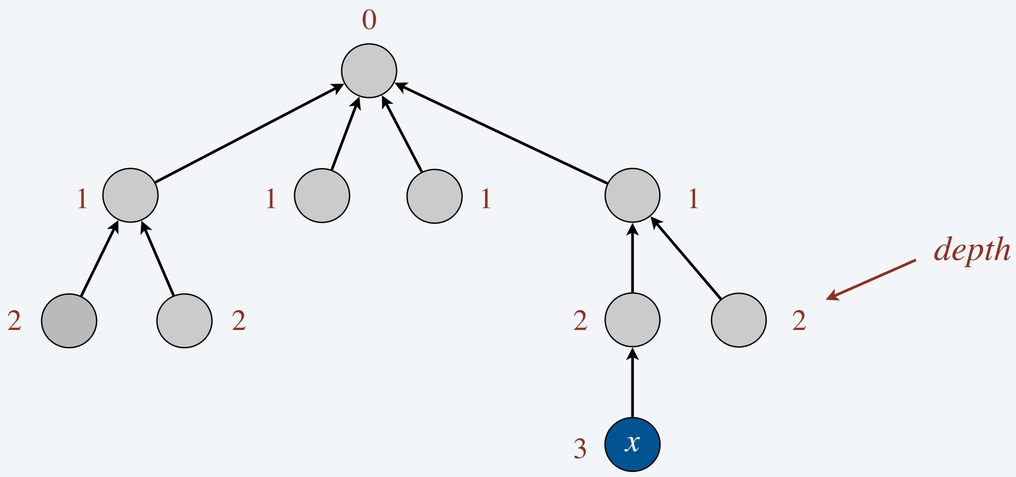
Quick-union analysis

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

Running time.

- `union(p, q)` takes constant time, once you know the two roots.
- `find(p)` takes time proportional to the depth of node `p` in tree.

 # links on path from node to root



depth(x) = 3

Quick-union analysis

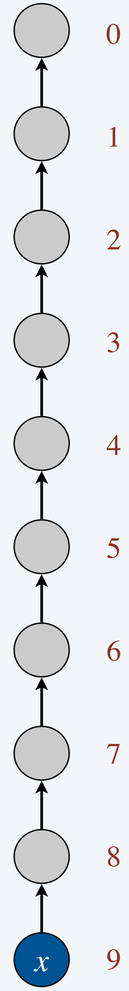
Cost model. Number of array accesses (for read or write).

Running time.

- `union(p, q)` takes constant time, once you know the two roots.
- `find(p)` takes time proportional to the depth of node `p` in tree.

algorithm	initialize	union	find
quick-find	n	n	1
quick-union	n	n	n

worst-case number of array accesses (up to constant factors)



worst-case depth = n-1

Union and find become expensive when trees get tall. Processing certain sequences of m union and find operations on n elements takes $\geq mn$ array accesses.

quadratic in input size !

https://algs4.cs.princeton.edu

1.5 Union–Find

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *path compaction*

Union-find: poll 3

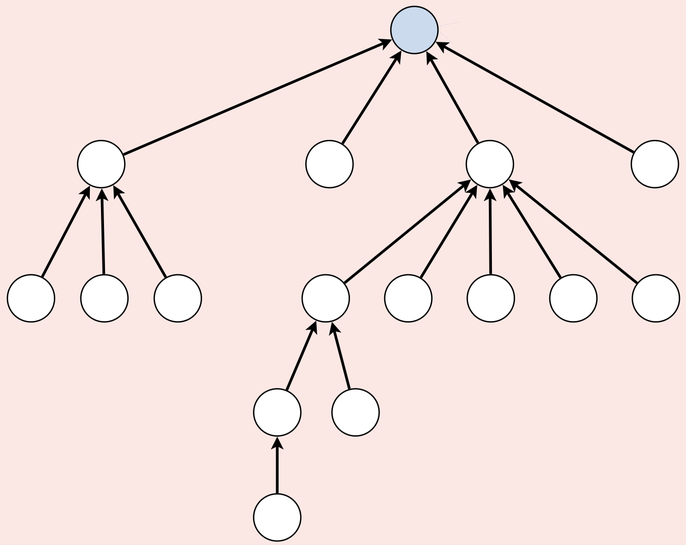
When linking two trees, which of these strategies is most effective?

A. Link the root of the smaller tree to the root of the larger tree

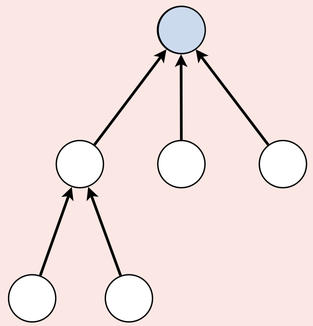
B. Link the root of the larger tree to the root of the smaller tree

C. Flip a coin; randomly choose between A and B

D. All of the above



larger tree
(size = 16, height = 4)



smaller tree
(size = 6, height = 2)

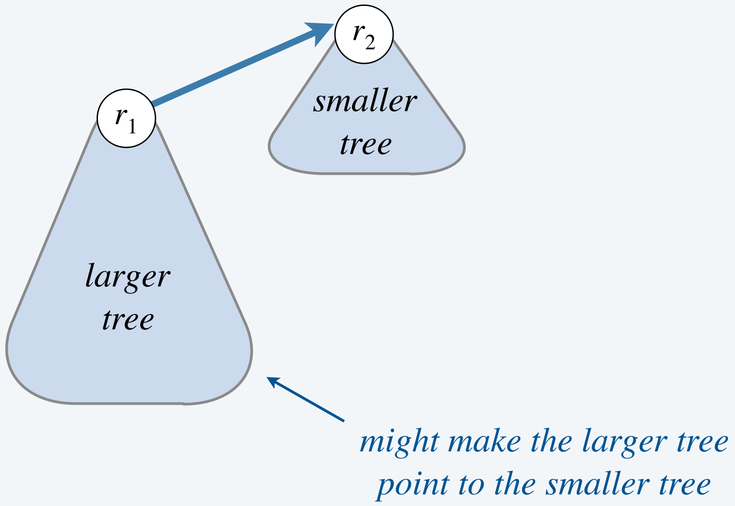
Weighted quick-union (link-by-size)

Link-by-size. Modify quick-union to avoid tall trees.

- Keep track of each tree's size = number of elements.
- Always make the root of the smaller tree point to the root of the larger tree.

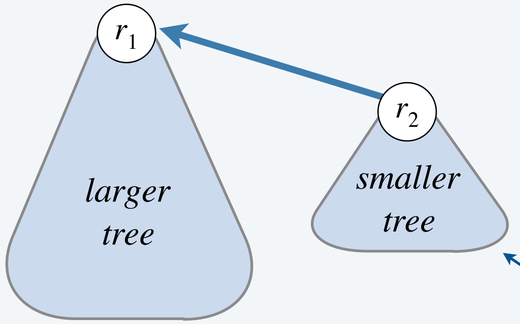
fine alternative: link-by-height (minimize worst-case depth vs. average depth)

quick-union



might make the larger tree point to the smaller tree

weighted quick-union



always makes the smaller tree point to the larger tree

Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain an extra array `size[i]` to count the number of elements in the tree rooted at `i` (initially `1`).

- `find()`: same as quick-union (follow parent points to root).
- `union()`: make the root of smaller tree point to root of the larger tree; update `size[]`.

```
public void union(int p, int q) {
   int rootP = find(p);
   int rootQ = find(q);
   if (rootP == rootQ) return;

   if (size[rootP] < size[rootQ]) {
      parent[rootP] = rootQ;
      size[rootQ] += size[rootP];
   }
   else {
      parent[rootQ] = rootP;
      size[rootP] += size[rootQ];
   }

}
```

p and q already in the same set

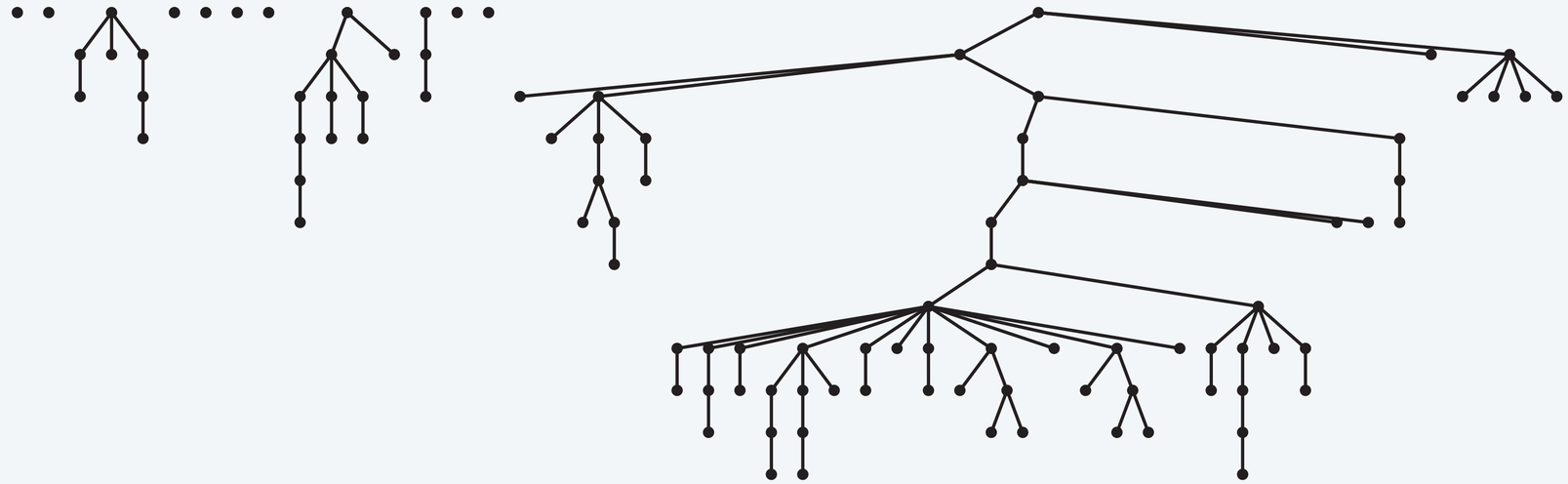
link root of smaller tree to root of larger tree (and update size)

new tree size is now the sum of tree sizes

https://algs4.cs.princeton.edu/15uf/WeightedQuickUnionUF.java.html

Quick-union vs. weighted quick-union: larger example

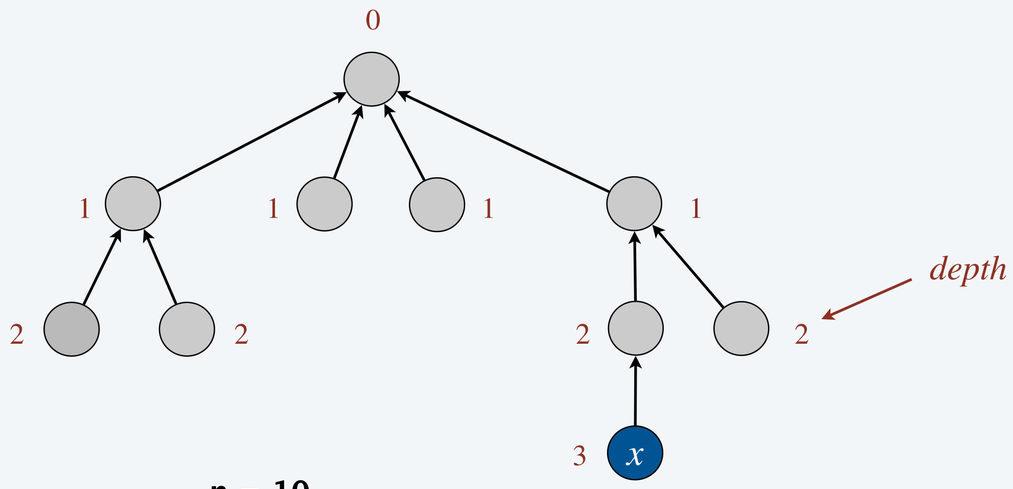
quick-union



weighted

Weighted quick-union analysis

Proposition. Every node has depth $\leq \log_2 n$.



n = 10

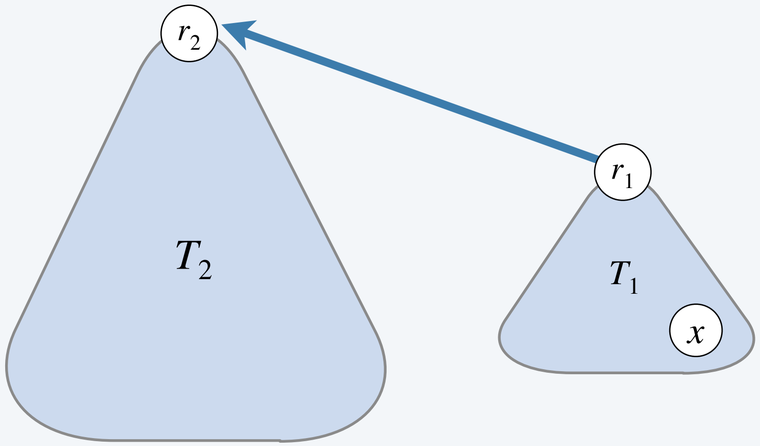
depth(x) = 3 ≤ $\log_2$ n

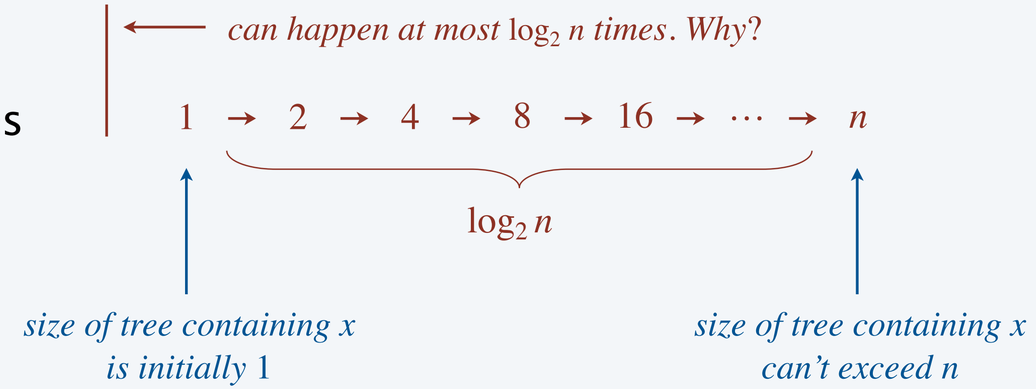
Weighted quick-union analysis

Proposition. Every node has depth $\leq \log_2 n$.

Pf.

- The depth of a node x increases only when the root of its tree T_1 is linked to the root of a larger tree T_2, forming a new tree T_3.
- When this happens:
  - the depth of x increases by 1, and
  - the size of the tree containing x at least doubles because $size(T_3) = size(T_1) + size(T_2) \geq 2 \times size(T_1)$.





Weighted quick-union analysis

Proposition. Every node has depth $\leq \log_2 n$.

Running time.

- `union(p, q)` takes constant time, once you know the two roots.
- `find(p)` takes time proportional to the depth of node `p` in tree.

algorithm	initialize	union	find
quick-find	n	n	1
quick-union	n	n	n
weighted quick-union	n	$\log n$	$\log n$

worst-case number of array accesses (up to constant factors)

← *in this course,* log *mean logarithm for some constant base*

SAVE MEMORY IN WEIGHTED QUICK-UNION

Challenge. Implement weighted quick-union using only one integer array of length n.

```
public class WeightedQuickUnionUF {
   private int[] parent;
   private int[] size;
   …

}
```

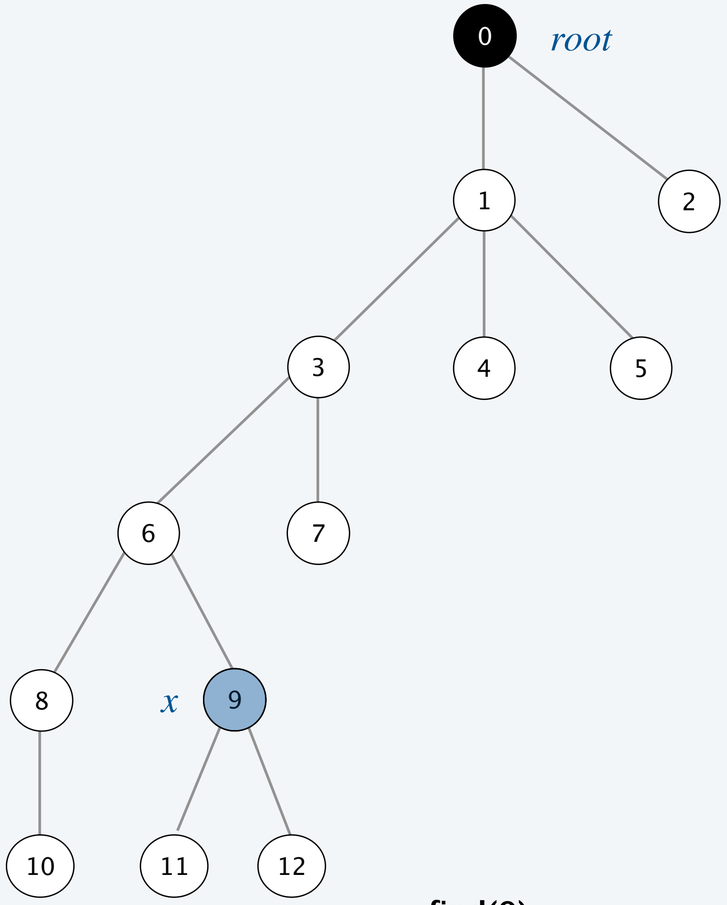
https://algs4.cs.princeton.edu

1.5 Union–Find

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *path compaction*

Path compression

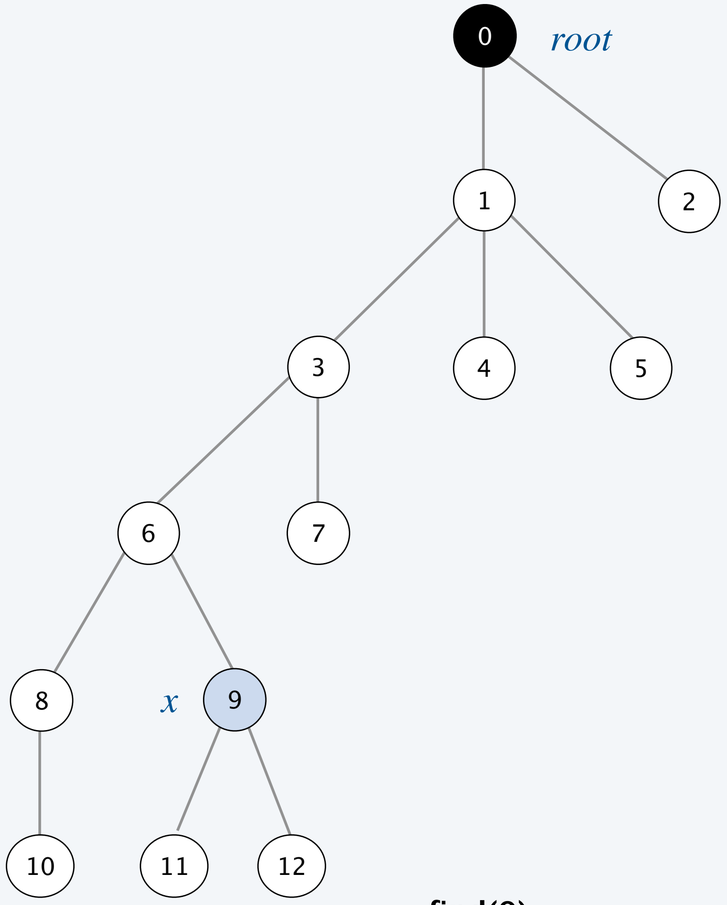
Path compression. Make every examined node point directly to its root.



find(9)

Path compression

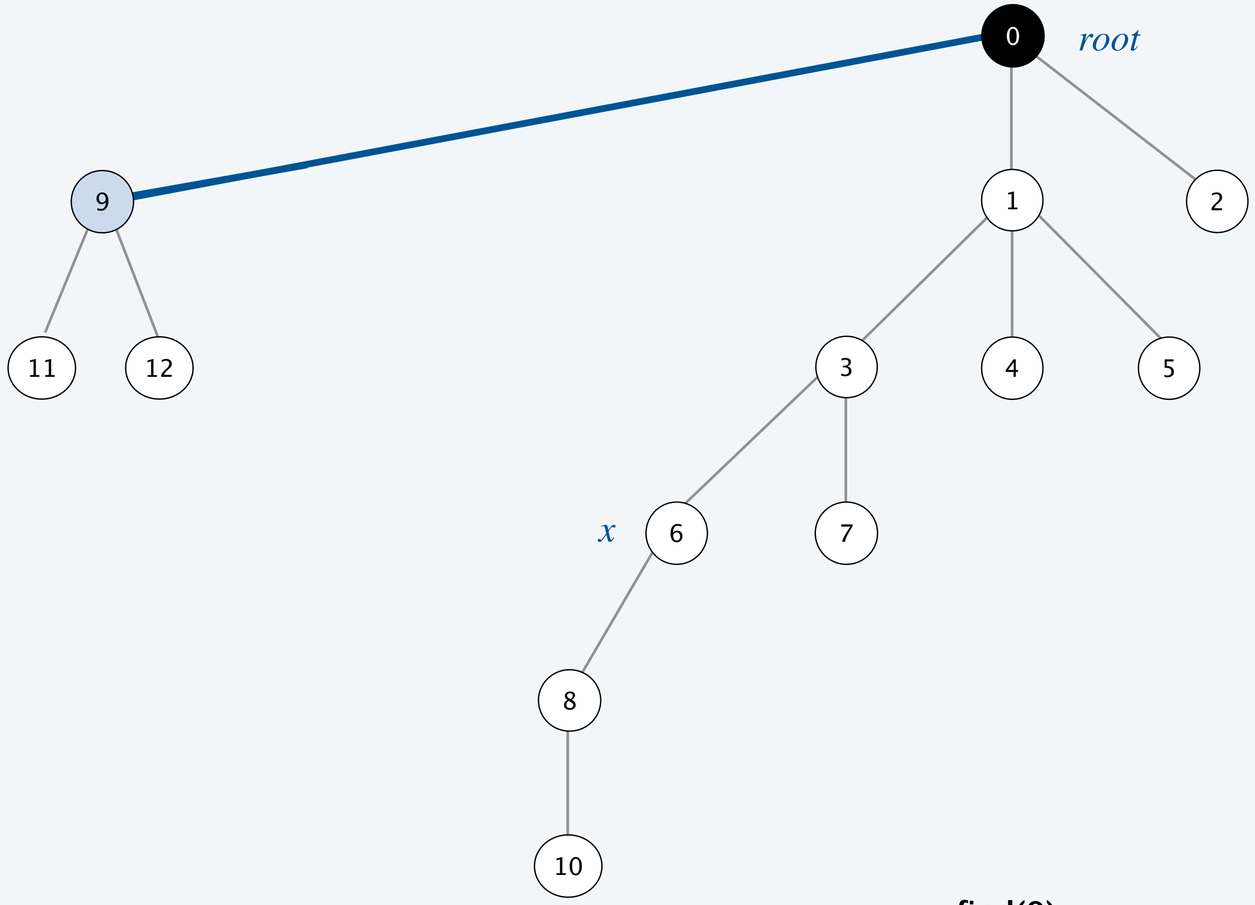
Path compression. Make every examined node point directly to its root.



find(9)

Path compression

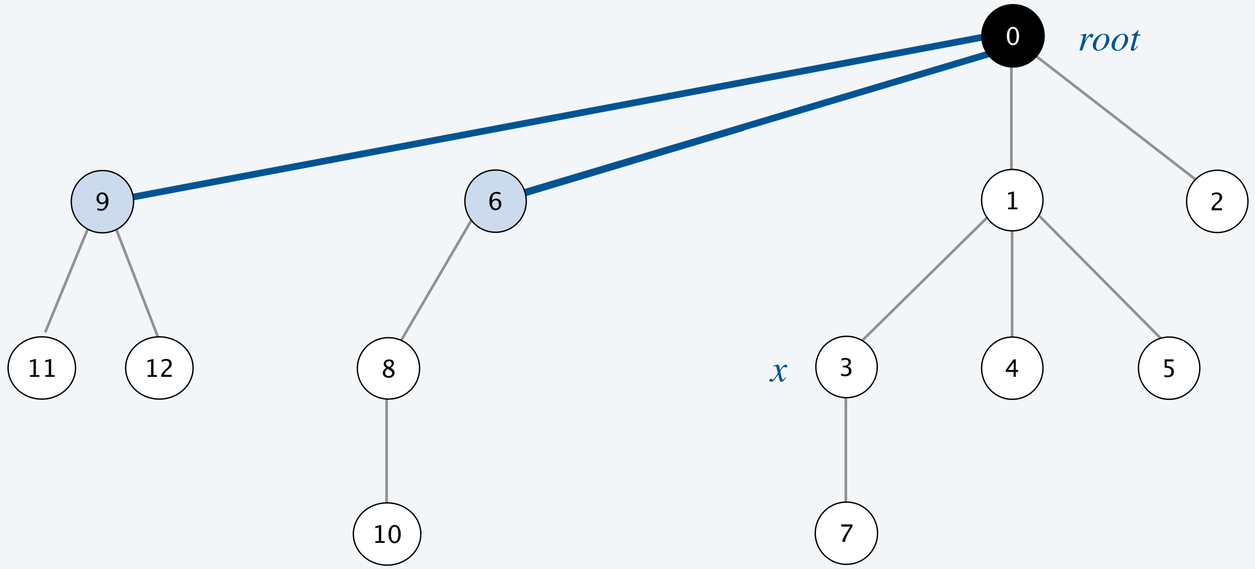
Path compression. Make every examined node point directly to its root.



find(9)

Path compression

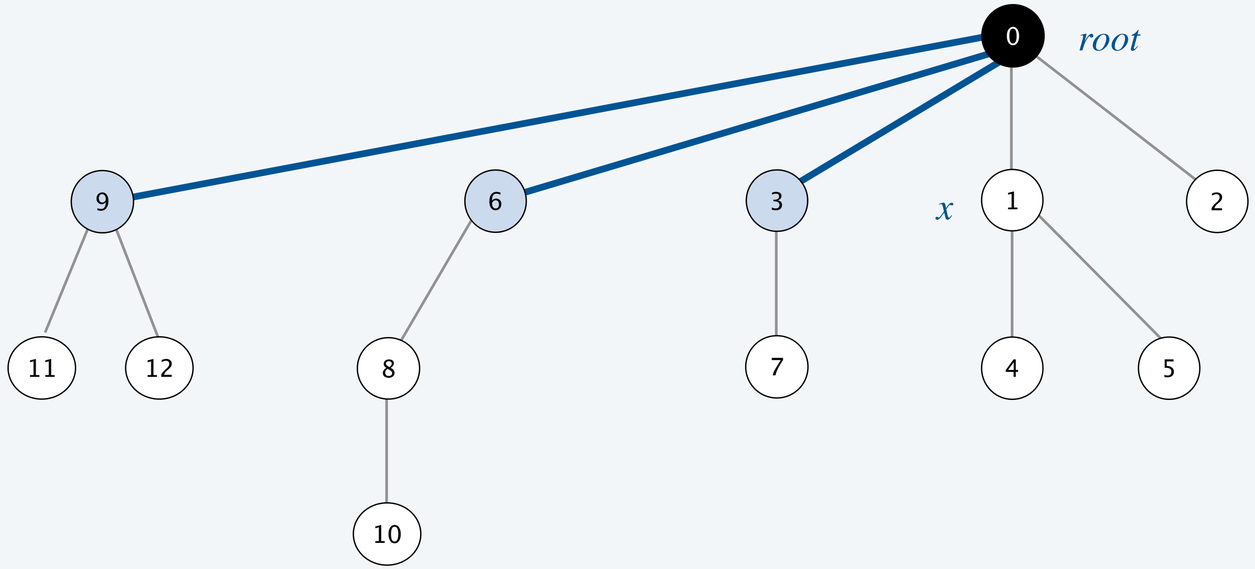
Path compression. Make every examined node point directly to its root.



find(9)

Path compression

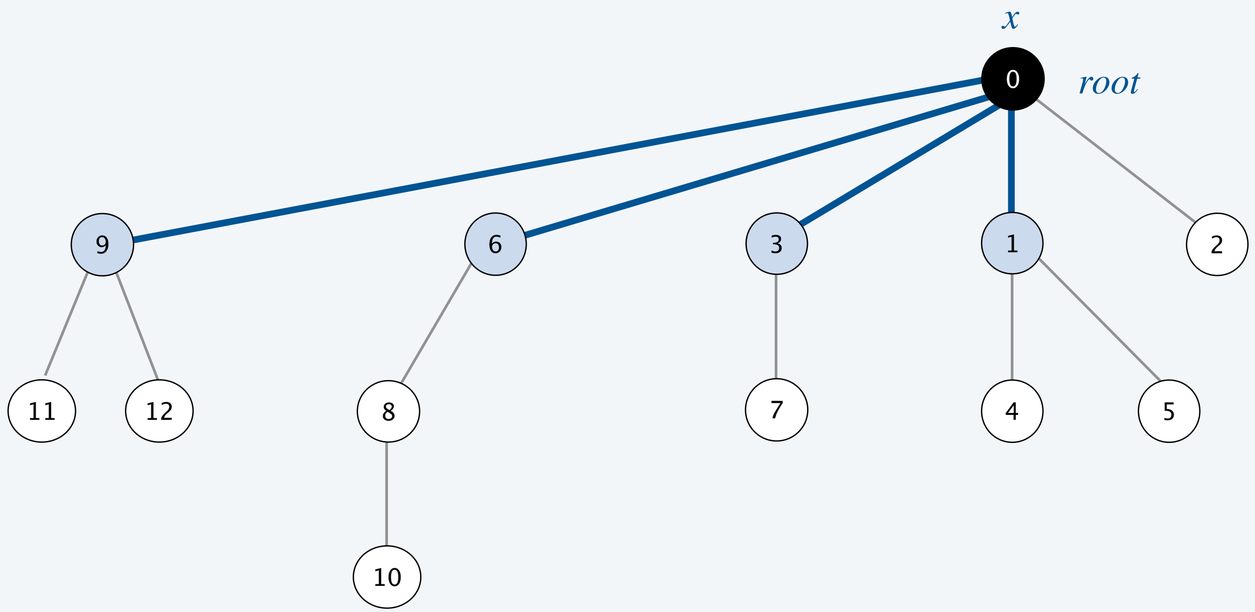
Path compression. Make every examined node point directly to its root.



find(9)

Path compression

Path compression. Make every examined node point directly to its root.



Bottom line. Now, `union()` and `find()` have the side effect of compressing the tree.

Path compaction: Java implementation

Path compression. Make every examined node point directly to its root.

Implementation. Add second loop to `find()` to update `parent[]` of each examined node.

Path halving. Make every other node on the find-path point to its grandparent.

Implementation. No need for a second loop.

```
public int find(int p) {
   while (p != parent[p]) {
      // path halving
      parent[p] = parent[parent[p]];
      p = parent[p];
   }
   return p;
}
```

only one extra line of code !

https://algs4.cs.princeton.edu/15uf/QuickUnionPathHalvingUF.java.html

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

Union-find: poll 4

What is the worst-case running time of a single `union()` or `find()` operation in quick-union with path compaction on n elements?

A. $\Theta(1)$

B. $\Theta(\log n)$

C. $\Theta(\sqrt{n})$

D. $\Theta(n)$

Summary

Key point. Weighted quick-union (and/or path compaction) makes it possible to solve problems that would otherwise take prohibitively long.

algorithm	worst-case time
quick-find	$m\ n$
quick-union	$m\ n$
weighted quick-union	$m \log n$
quick-union + path compaction	$m \log n$
weighted quick-union + path compaction	$m\ \alpha(m, n)$

see P05

inverse "Ackermann" function (*ask Tarjan and see* COS 423)

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

Ex. [10^9 union-find operations on 10^9 elements]

- Efficient algorithm reduces time from 30 years to 6 seconds.
- Supercomputer won't help much.

Credits

image	source
Game of Hex	Wolfram MathWorld
Cluster Labeling	Tiberiu Marita
Bob Tarjan	Princeton University
Computer and Supercomputer	New York Times

A final thought

> *The goal is to come up with algorithms that you can apply in practice that run fast, as well as being simple, beautiful, and analyzable.*

— Robert Tarjan