

Algorithms & Data Structures

Lesson 14: Disjoint Sets & Union-Find

Marc Gaetano

Edition 2017-2018

The plan

- What are *disjoint sets*
 - And how are they “the same thing” as *equivalence relations*
- The union-find ADT for disjoint sets
- Applications of union-find
- Basic implementation of the ADT with “up trees”
- Optimizations that make the implementation much faster

Disjoint sets

- A **set** is a collection of elements (no-repeats)
- Two sets are **disjoint** if they have no elements in common
 - $S_1 \cap S_2 = \emptyset$
- Example: {a, e, c} and {d, b} are disjoint
- Example: {x, y, z} and {t, u, x} are not disjoint

Partitions

A **partition** P of a set S is a set of sets $\{S_1, S_2, \dots, S_n\}$ such that every element of S is in **exactly one** S_i

Put another way:

- $S_1 \cup S_2 \cup \dots \cup S_k = S$
- For all i and j , $i \neq j$ implies $S_i \cap S_j = \emptyset$

Example:

- Let S be $\{a, b, c, d, e\}$
- One partition: $\{a\}, \{d, e\}, \{b, c\}$
- Another partition: $\{a, b, c\}, \emptyset, \{d\}, \{e\}$
- A third: $\{a, b, c, d, e\}$
- Not a partition: $\{a, b, d\}, \{c, d, e\}$
- Not a partition of S : $\{a, b\}, \{e, c\}$

Binary relations

- $S \times S$ is the set of all pairs of elements of S
 - Example: If $S = \{a,b,c\}$
then $S \times S = \{(a,a),(a,b),(a,c),(b,a),(b,b),(b,c),(c,a),(c,b),(c,c)\}$
- A binary relation R on a set S is any subset of $S \times S$
 - Write $R(x,y)$ to mean (x,y) is “in the relation”
 - (Unary, ternary, quaternary, ... relations defined similarly)
- Examples of binary relations for $S = \text{people-in-this-room}$
 - Sitting-next-to-each-other relation
 - First-sitting-right-of-second relation
 - Went-to-same-high-school relation
 - Same-gender-relation
 - First-is-younger-than-second relation

Properties of binary relations

- A binary relation R over set S is **reflexive** if:
 $R(a,a)$ for all a in S
- A binary relation R over set S is **symmetric** if:
 $R(a,b)$ if and only if $R(b,a)$ for all a,b in S
- A binary relation R over set S is **transitive** if:
If $R(a,b)$ and $R(b,c)$ then $R(a,c)$ for all a,b,c in S
- Examples for $S = \text{people-in-this-room}$
 - Sitting-next-to-each-other relation
 - First-sitting-right-of-second relation
 - Went-to-same-high-school relation
 - Same-gender-relation
 - First-is-younger-than-second relation

Equivalence relations

- A binary relation R is an **equivalence relation** if R is reflexive, symmetric, and transitive
- Examples
 - Same gender
 - Connected roads in the world
 - *Graduated* from same high school?
 - ...

Punch-line

- Every partition induces an *equivalence relation*
- Every equivalence relation *induces* a partition
- Suppose $P=\{S_1, S_2, \dots, S_n\}$ be a partition
 - Define $R(x,y)$ to mean x and y are in the same S_i
 - R is an equivalence relation
- Suppose R is an equivalence relation over S
 - Consider a set of sets S_1, S_2, \dots, S_n where
 - (1) x and y are in the same S_i if and only if $R(x,y)$
 - (2) Every x is in some S_i
 - This set of sets is a partition

Example

- Let S be $\{a,b,c,d,e\}$
- One partition: $\{a,b,c\}, \{d\}, \{e\}$
- The corresponding equivalence relation:
 $(a,a), (b,b), (c,c), (a,b), (b,a), (a,c), (c,a), (b,c), (c,b), (d,d), (e,e)$

The operations

- Given an unchanging set S , **create** an initial partition of a set
 - Typically each item in its own subset: $\{a\}$, $\{b\}$, $\{c\}$, ...
 - Give each subset a “name” by choosing a *representative element*
- Operation **find** takes an element of S and returns the representative element of the subset it is in
- Operation **union** takes two subsets and (permanently) makes one larger subset
 - A different partition with one fewer set
 - Affects result of subsequent **find** operations
 - Choice of representative element up to implementation

Example

- Let $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Let initial partition be (will highlight representative elements red)
 $\{\underline{1}\}, \{\underline{2}\}, \{\underline{3}\}, \{\underline{4}\}, \{\underline{5}\}, \{\underline{6}\}, \{\underline{7}\}, \{\underline{8}\}, \{\underline{9}\}$
- `union(2,5):`
 $\{\underline{1}\}, \{\underline{2}, 5\}, \{\underline{3}\}, \{\underline{4}\}, \{\underline{6}\}, \{\underline{7}\}, \{\underline{8}\}, \{\underline{9}\}$
- `find(4) = 4, find(2) = 2, find(5) = 2`
- `union(4,6), union(2,7)`
 $\{\underline{1}\}, \{\underline{2}, 5, 7\}, \{\underline{3}\}, \{4, \underline{6}\}, \{\underline{8}\}, \{\underline{9}\}$
- `find(4) = 6, find(2) = 2, find(5) = 2`
- `union(2,6)`
 $\{\underline{1}\}, \{\underline{2}, 4, 5, 6, 7\}, \{\underline{3}\}, \{\underline{8}\}, \{\underline{9}\}$

No other operations

- All that can “happen” is sets get unioned
 - No “un-union” or “create new set” or ...
- As always: trade-offs – implementations will exploit this small ADT
- Surprisingly useful ADT: list of applications after one example
 - But not as common as dictionaries or priority queues

Applications

- Maze-building is:
 - A surprising use of the union-find ADT
- Many other uses:
 - Road/network/graph connectivity (will see this again)
 - “connected components” e.g., in social network
 - Partition an image by connected-pixels-of-similar-color
 - Type inference in programming languages
- Not as common as dictionaries, queues, and stacks, but valuable because implementations are very fast, so when applicable can provide big improvements

Basic implementation

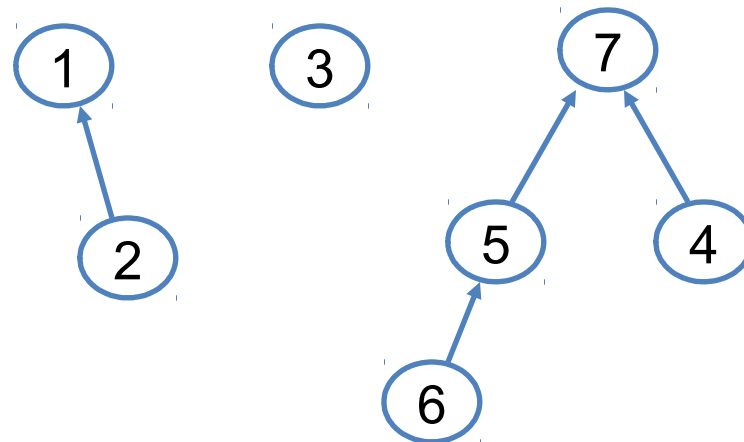
- Start with an initial partition of n subsets
 - Often 1-element sets, e.g., $\{1\}$, $\{2\}$, $\{3\}$, ..., $\{n\}$
- May have m **find** operations and up to $n-1$ **union** operations in any order
 - After $n-1$ **union** operations, every **find** returns same 1 set
- If total for all these operations is $O(m+n)$, then amortized $O(1)$
 - We will get very, very close to this
 - $O(1)$ worst-case is impossible for **find and union**
 - Trivial for one *or* the other

Up-tree data structure

- Tree with:
 - No limit on branching factor
 - References from children to parent
- Start with *forest* of 1-node trees



- Possible forest after several unions:
 - Will use roots for set names

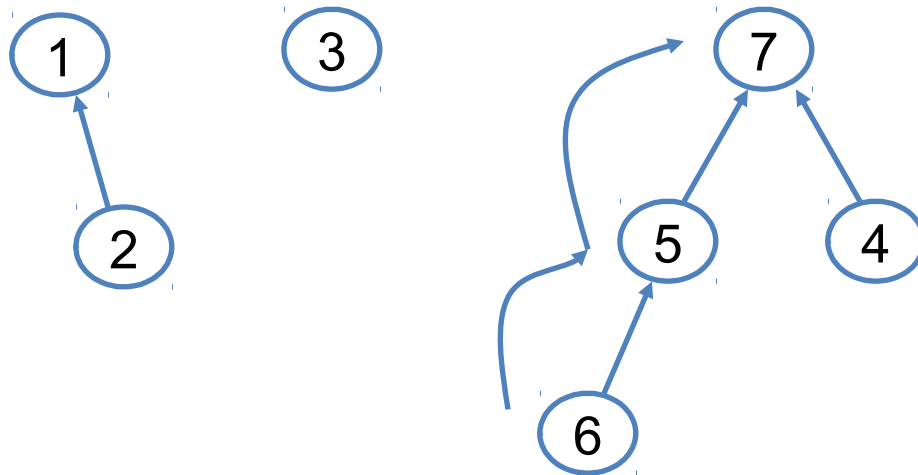


Find

find(x):

- Assume we have $O(1)$ access to each node
 - Will use an array where index i holds node i
- Start at x and follow parent pointers to root
- Return the root

find(6) = 7

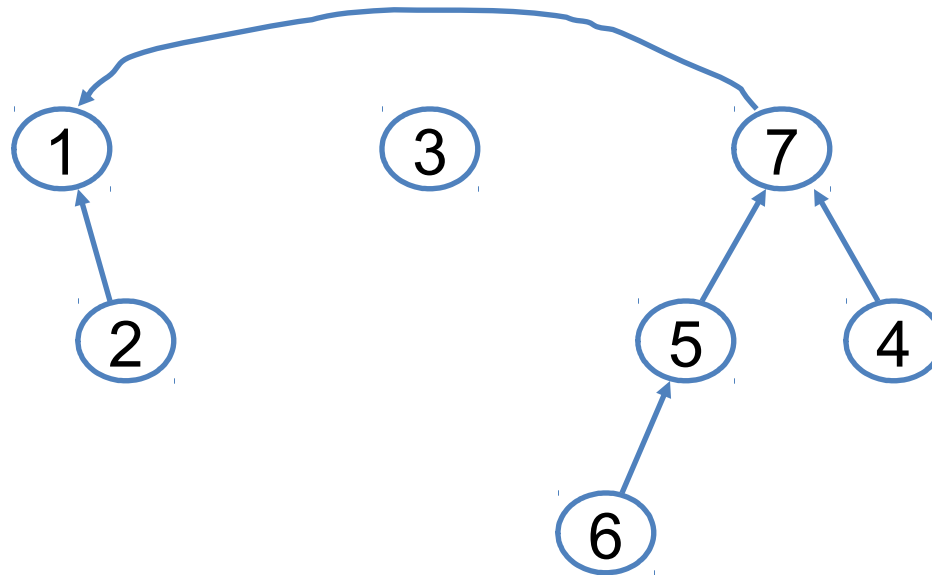


Union

union(x, y):

- Assume **x** and **y** are roots
 - If they are not, just find the roots of their trees
- Assume distinct trees (else do nothing)
- Change root of one to have parent be the root of the other
 - Notice no limit on branching factor

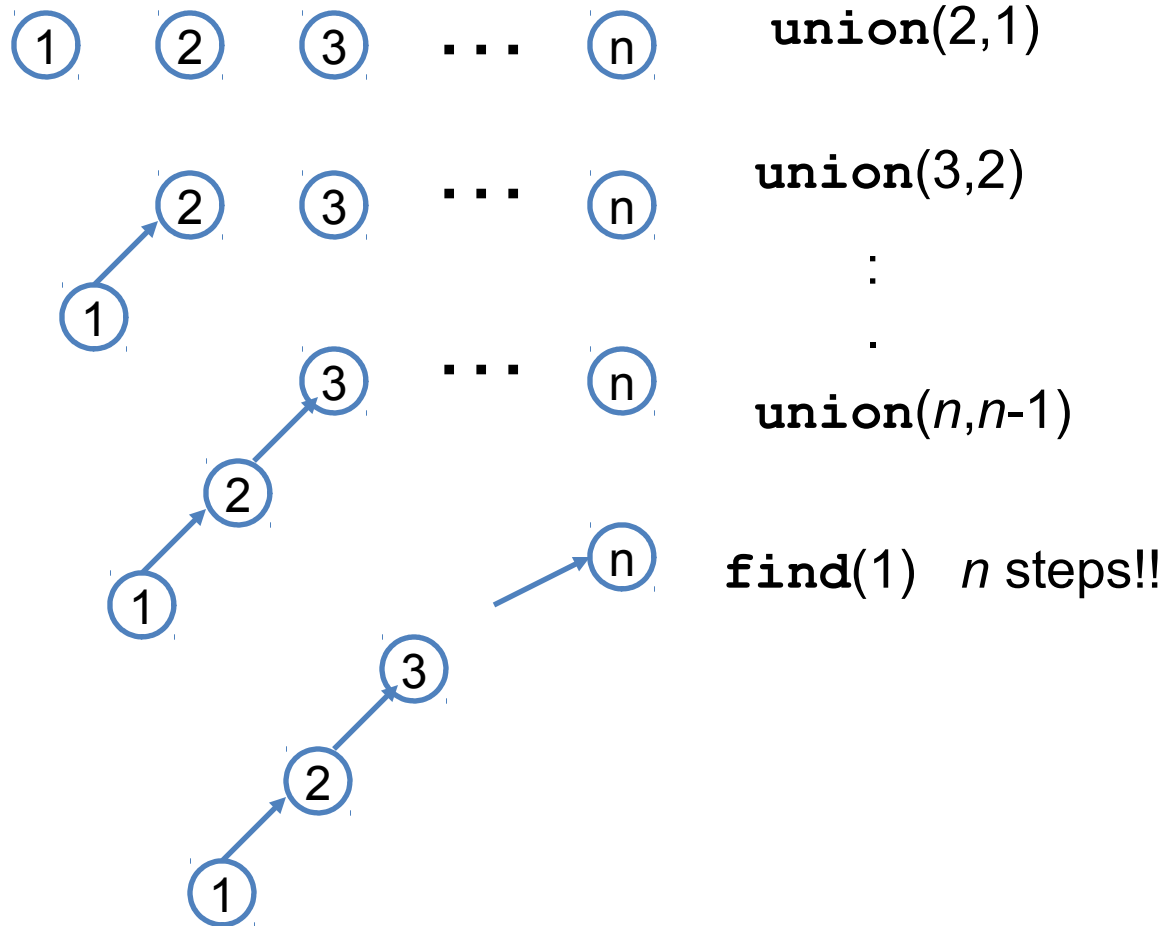
union(1,7)



Two key optimizations

1. Improve **union** so it stays $O(1)$ but makes **find** $O(\log n)$
 - So m **finds** and $n-1$ **unions** is $O(m \log n + n)$
 - *Union-by-size*: connect smaller tree to larger tree
2. Improve **find** so it becomes even faster
 - Make m **finds** and $n-1$ **unions** ***almost*** $O(m + n)$
 - *Path-compression*: connect directly to root during finds

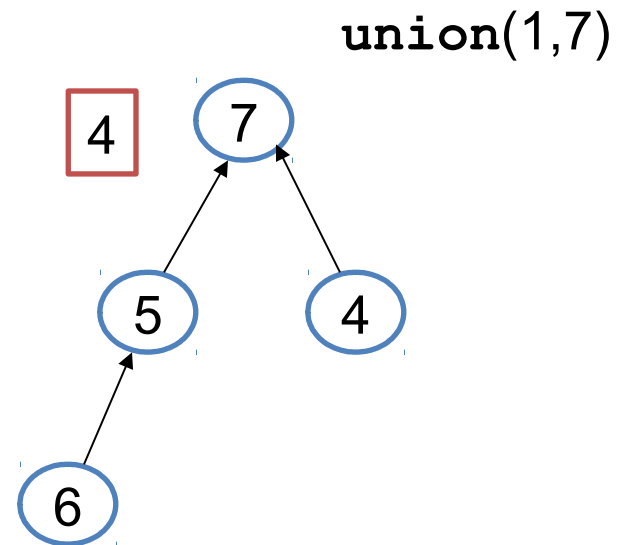
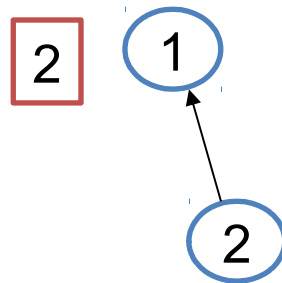
The bad case to avoid



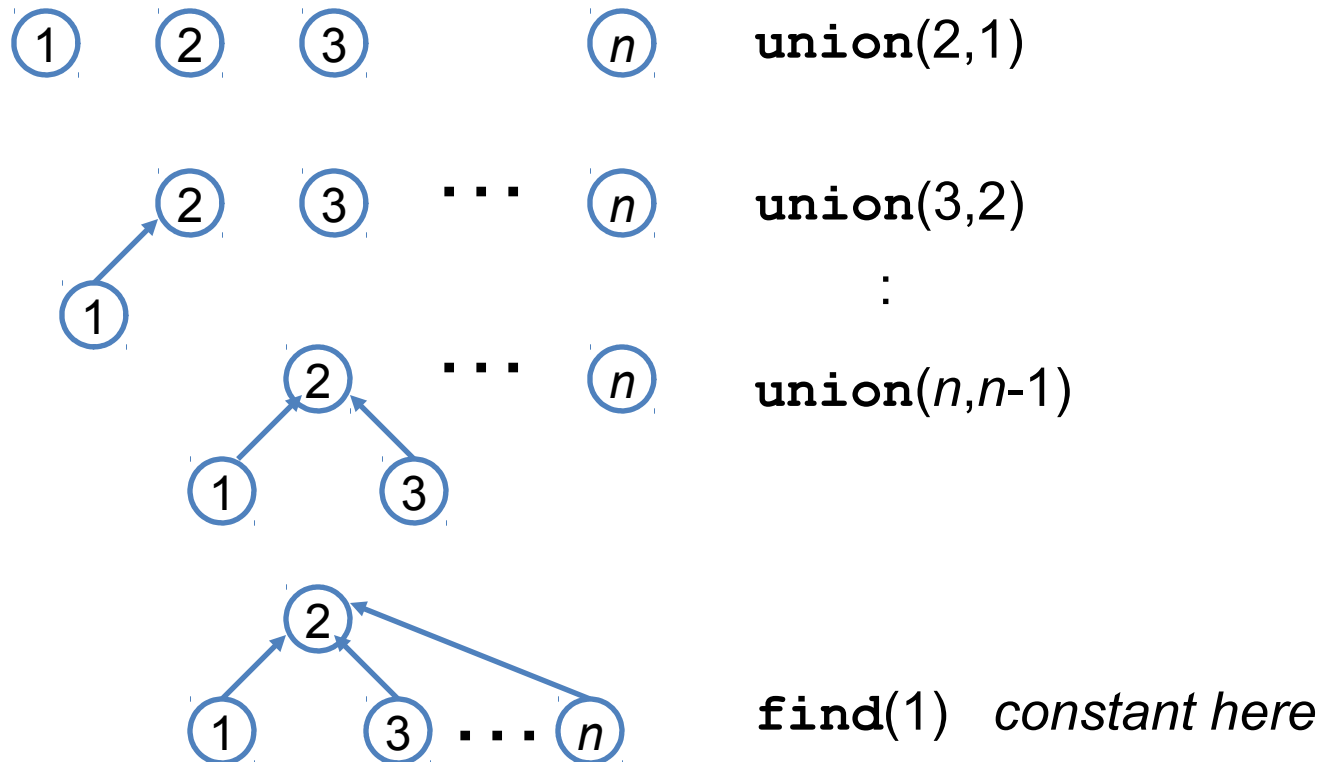
Weighted union

Weighted union:

- Always point the *smaller* (total # of nodes) tree to the root of the larger tree



Bad example? Great example...



General analysis

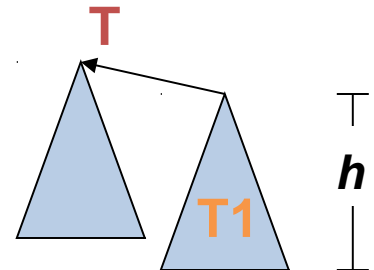
- Showing that one worst-case example is now good is *not* a proof that the worst-case has improved
- So let's prove:
 - **union** is still $O(1)$ – this is fairly easy to show
 - **find** is now $O(\log n)$
- Claim: If we use weighted-union, an up-tree of height h has at least 2^h nodes
 - Proof by induction on h ...

Exponential number of nodes

$P(h)$ = With weighted-union, up-tree of height h has at least 2^h nodes

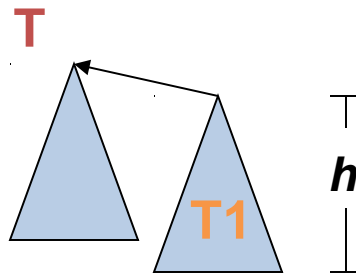
Proof by induction on h ...

- Base case: $h = 0$: The up-tree has 1 node and $2^0 = 1$
- Inductive case: Assume $P(h)$ and show $P(h+1)$
 - A height $h+1$ tree T has at least one height h child $T1$
 - $T1$ has at least 2^h nodes by induction
 - And T has *at least* as many nodes not in $T1$ than in $T1$
 - Else weighted-union would have had T point to $T1$, not $T1$ point to T (!!)
 - So total number of nodes is *at least* $2^h + 2^h = 2^{h+1}$



The key idea

Intuition behind the proof: No one child can have more than half the nodes

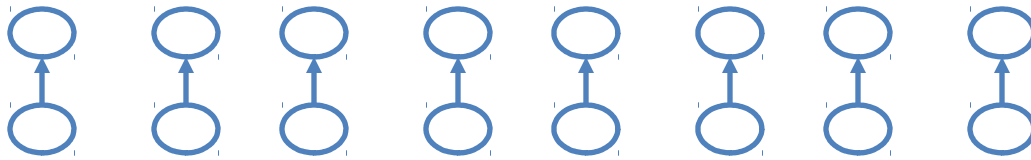


So, as usual, if number of nodes is exponential in height, then height is logarithmic in number of nodes

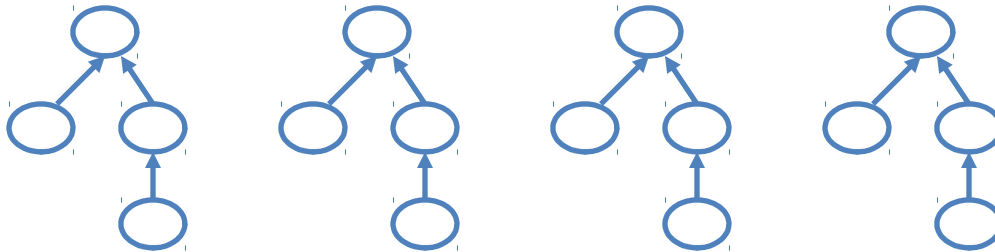
So **find** is $O(\log n)$

The new worst case

$n/2$ Weighted Unions

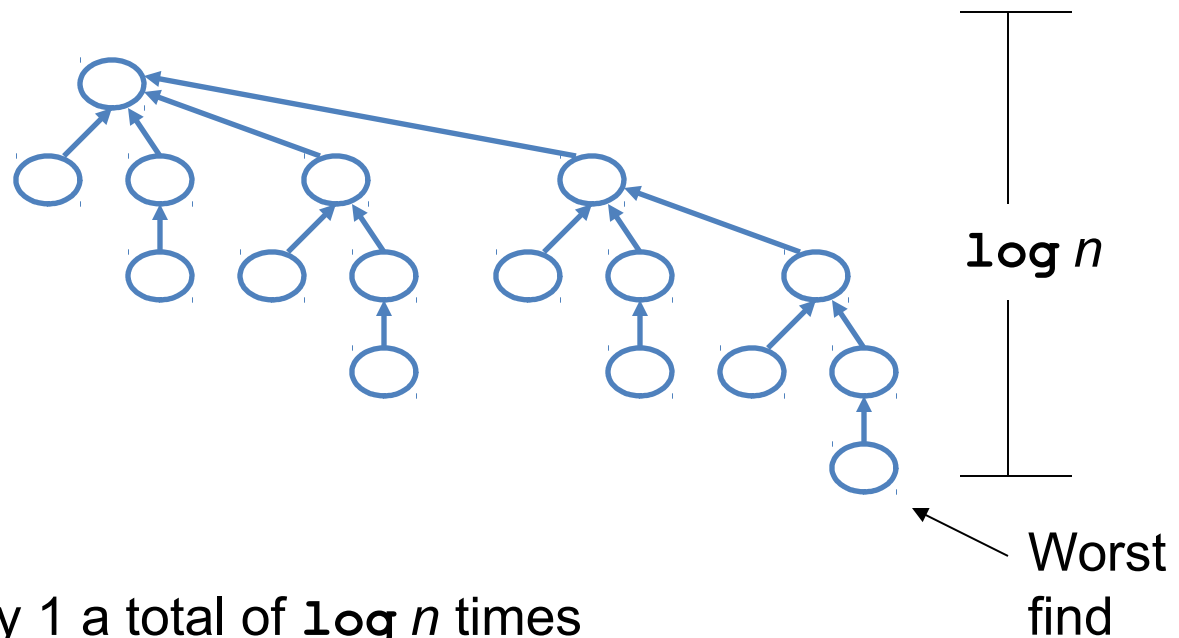


$n/4$ Weighted Unions



The new worst case (continued)

After $n/2 + n/4 + \dots + 1$ Weighted Unions:



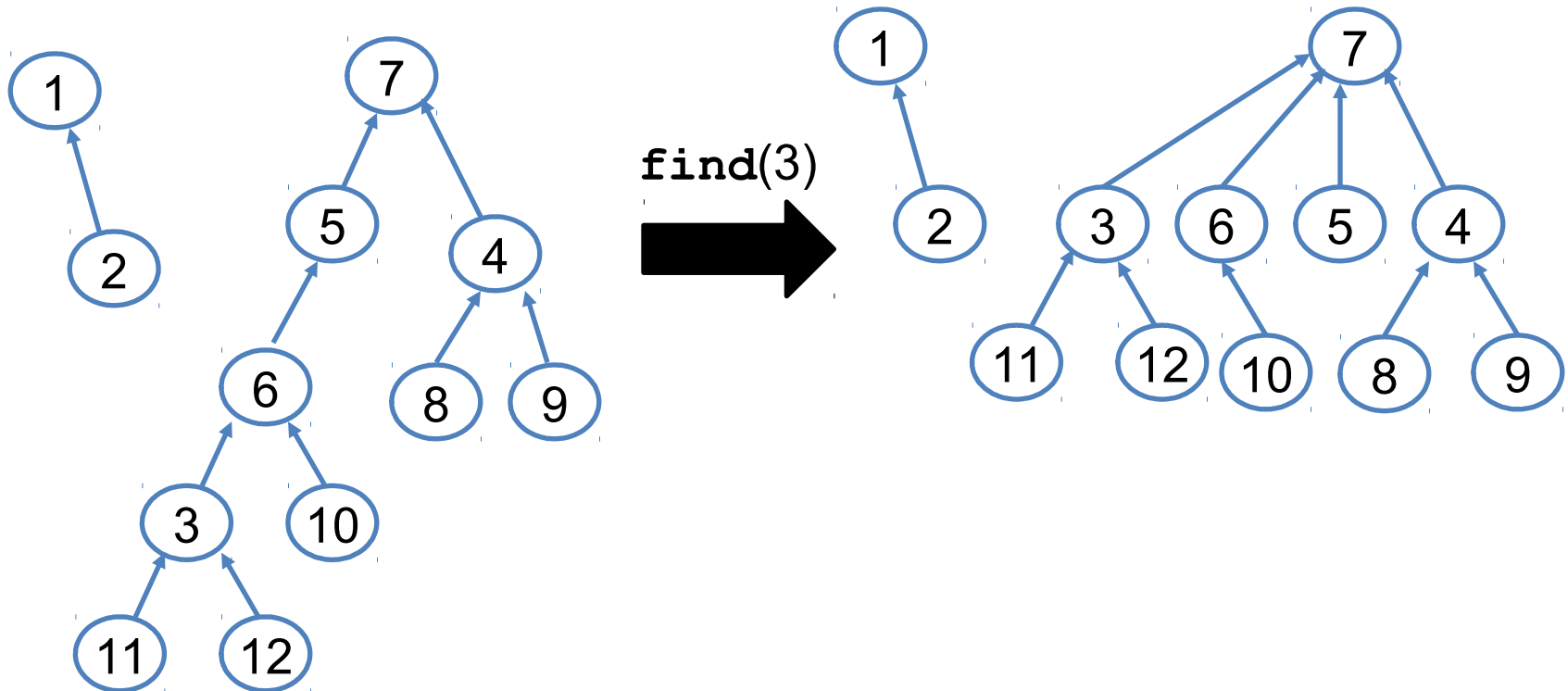
Height grows by 1 a total of $\log n$ times

Two key optimizations

1. Improve **union** so it stays $O(1)$ but makes **find** $O(\log n)$
 - So m **finds** and $n-1$ **unions** is $O(m \log n + n)$
 - *Union-by-size*: connect smaller tree to larger tree
2. Improve **find** so it becomes even faster
 - Make m **finds** and $n-1$ **unions** ***almost*** $O(m + n)$
 - *Path-compression*: connect directly to root during finds

Path compression

- Simple idea: As part of a **find**, change each encountered node's parent to point directly to root
 - Faster future **finds** for everything on the path (and their descendants)



So, how fast is it?

A single worst-case **find** could be $O(\log n)$

- But only if we did a lot of worst-case unions beforehand
- And path compression will make future finds faster

Turns out the amortized worst-case bound is much better than $O(\log n)$

- We won't *prove* it – see text if curious
- Result is that it is *almost* $O(1)$ because total for m **finds** and $n-1$ **unions** is *almost* $O(m+n)$