Lecture 12

Disjoint Sets & Union-Find

Problem

Consider this problem. How would you implement a solution?

Given: `friendGroups`, representing groups of friends. You can assume unique names for each person and each person is only in one group.

Example input:

```
[
 ["Riley", "Pascale", "Matthew", "Hunter"],
 [ "Chloe", "Paul", "Zelina"],
 [ "Rebecca", "Raquel", "Trung", "Kyle", "Josh"]
]
```

Problem: Given two Strings "Pascale" and "Raquel" determine if they are in the same group of friends.

Solution Ideas

1. Traverse each Set until you find the Set containing the first name, then see if it also contains the second name.
2. Store a map of people to the set of people they are friends with. Then find the Set of friends for the first name and see if it contains the second name. Note, this works for friends in multiple groups as well.

   ```
   [
   "Riley" → ["Pascale", "Matthew", "Hunter"],
   "Pascale" → ["Riley", "Matthew", "Hunter"],
   ...]
   ```
3. Store friendship in a Graph. A lot like solution 2 actually
4. Disjoint Sets and Union-Find Others?

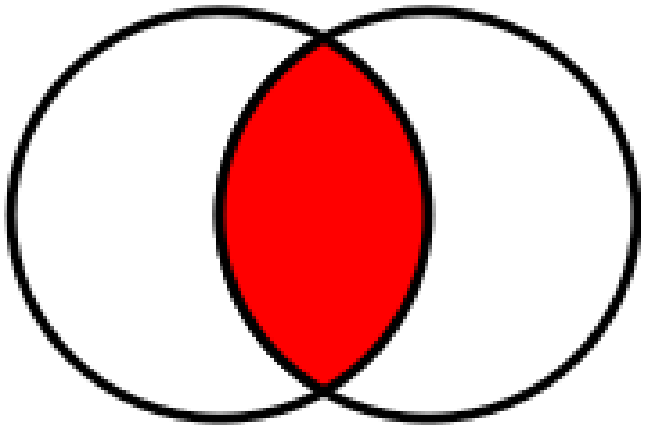
Disjoint Sets and Union Find

- What are sets and *disjoint sets*
- The union-find ADT for disjoint sets
- Basic implementation with "up trees"
- Optimizations that make the implementation much faster

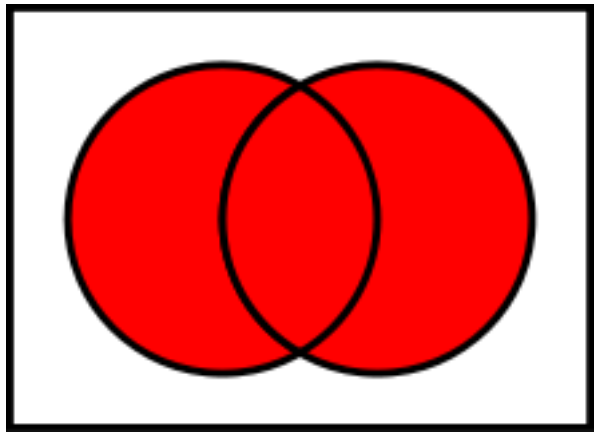
Terminology

Empty set: ∅

Intersection ∩



Union ∪



Notation for elements in a set:

Set S containing e1, e2 and e3: **{e1, e2, el3}**

e1 is an element of S: **e1 ∈ S**

Disjoint sets

- A set is a collection of elements (no-repeats)
- Every set contains the empty set by default
- Two sets are disjoint if they have no elements in common
  - $\mathbf{S_1} \cap \mathbf{S_2} = \varnothing$

- Examples:
  - {a, e, c} and {d, b} Disjoint
  - {x, y, z} and {t, u, x} Not disjoint

Partitions

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

Put another way:

- $S_1 \cup S_2 \cup \ldots \cup S_k = S$
- For all i and j, $i \neq j$ implies $S_i \cap S_j = \varnothing$ (sets are disjoint with each other)

Example: Let *S* be {a,b,c,d,e}

- {a}, {d,e}, {b,c} Partition
- {a,b,c}, ∅, {d}, {e} Partition
- {a,b,c,d,e} Partition
- {a,b,d}, {c,d,e} Not a partition, not disjoint, both sets have d
- {a,b}, {e,c} Not a partition of S (doesn't have d)

Union Find ADT: 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

Subset Find for our problem

- Given an unchanging set *S*, **`create`** an initial partition of a set

```
    "Riley" -> ["Riley", "Pascale", "Matthew", "Hunter"],
    "Chloe" -> [ "Chloe", "Paul", "Zelina"],
    "Rebecca" -> [ "Rebecca", "Raquel", "Trung", "Kyle",
"Josh"]
```

- Operation **`find`** takes an element of *S* and returns the representative element of the subset it is in

```
find("Pascale") returns "Riley"
find("Chloe") returns "Chloe"
```

Not the same subset since not the same representative

Union of two subsets for our problem

- Operation **`union`** takes two subsets and (permanently) makes one larger subset

Chloe and Riley become friends, merging their two groups. Now those to subsets become one subset. We can represent that in two ways:

Merge the sets:

```
"Chloe" -> [ "Chloe", "Paul", "Zelina", "Riley",
        "Pascale", "Matthew", "Hunter"]
```

Or tell Riley that her representative is now Chloe, and on find anyone in Riley's old subset like `find("Pascale")` see what group Riley is in:

```
"Riley" -> ["Pascale", "Matthew", "Hunter"],
"Chloe" -> [ "Chloe", "Paul", "Zelina",
"Riley"]
```

Either way, `find("Pascale")` returns "Chloe"

Another Example

- Let S = {1,2,3,4,5,6,7,8,9}
- Let initial partition be (will highlight representative elements red)

  {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}
- **union**(2,5):

  {1}, {2, 5}, {3}, {4}, {6}, {7}, {8}, {9}
- **find**(4) = 4, **find**(2) = 2, **find**(5) = 2
- **union**(4,6), **union**(2,7)

  {1}, {2, 5, 7}, {3}, {4, 6}, {8}, {9}
- **find**(4) = 6, **find**(2) = 2, **find**(5) = 2
- **union**(2,6)

  {1}, {2, 4, 5, 6, 7}, {3}, {8}, {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 are different
  - ideas? How do we maintain “representative” of a subset?
- Surprisingly useful ADT, but not as common as dictionaries, priority queues / heaps, AVL trees or hashing

Applications / Thoughts on Union-Find

- Many 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

- Our friend group example could be done with Graphs (we'll learn about them later) but we can use Union-Find for a much less storage intense implementation. Cool! ☺
- Union-Find is not as common as dictionaries, queues, and stacks, but valuable because implementations are very fast, so when applicable can provide big improvements

Implementation

- How do you store a subset?
- How do you know what the “representative” is?
- How do you implement union?
- How do you pick a new “representative”?
- What is the cost of find? Of union? Of create?

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

How should we “draw” this data structure?

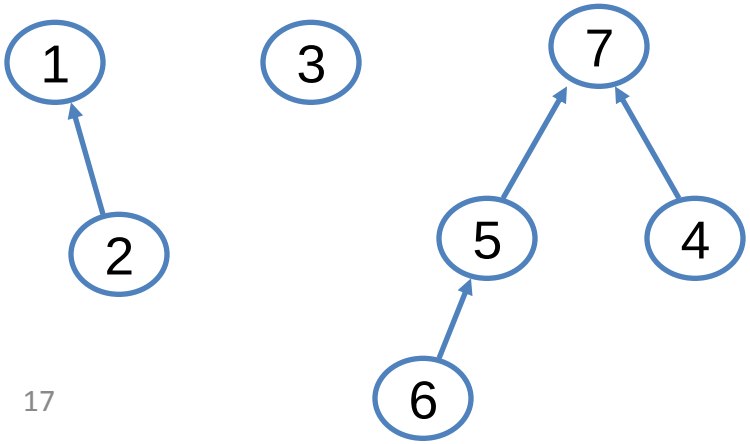
- Saw with heaps that a more intuitive depiction of the data structure can help us better conceptualize the operations.

- We can still implement the code in different ways, just like heaps can be implemented with an array even though we think of them as a tree structure.

Up-tree data structure

- Tree with any number of children at each node
  - References from children to parent (each child knows who it's parent is)
- Start with *forest (collection of trees)* of 1-node trees

- Possible forest after several unions:
  - Will use overall roots for the representative element

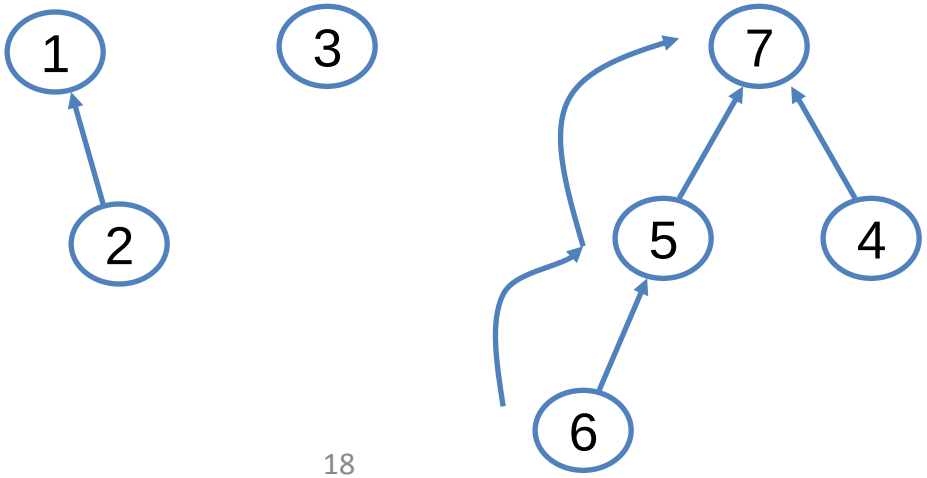


Find

`find`(`x`): (backwards from the tree traversals we've been doing for find so far)

- ***Assume*** we have *O*(1) access to each node
- Start at **`x`** and follow parent pointers to root
- Return the root

`find`(6) = 7

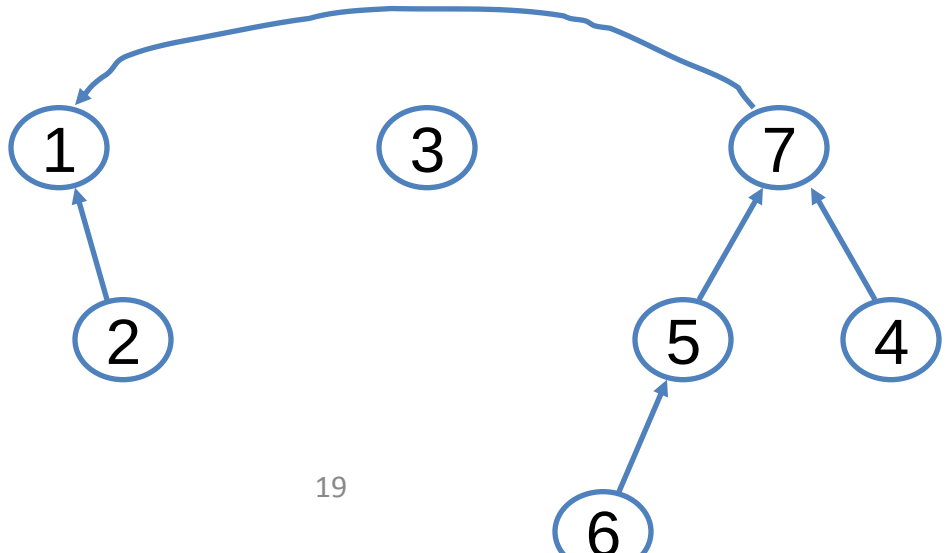


Union

`union(x,y)`:

- Find the roots of **`x`** and **`y`**
- if distinct trees, we merge, if the same tree, do nothing
- Change root of one to have parent be the root of the other

`union`(1,7)



Okay, how can we represent it internally?

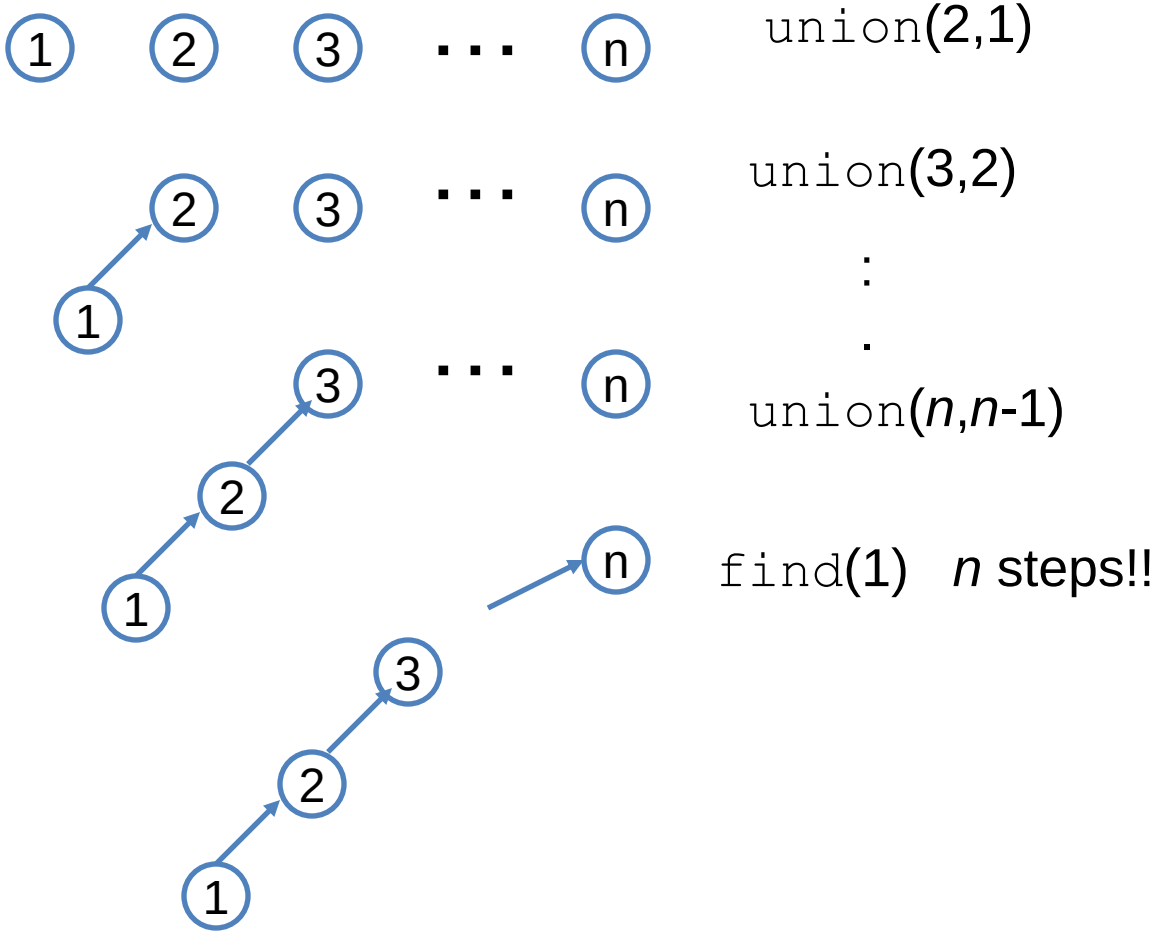
- Important to remember from the operations:
  - We assume O(1) access to *each* node
  - Ideally, we want the traversal from leaf to root of each tree to be as short as possible (the find operation depends on this traversal)
  - We don't want to copy a bunch of nodes to a new tree on each union, we only want to modify one pointer (or a small constant number of them)

Two key optimizations

1. Improve **`union`** so it stays *O(1)* but makes **`find`** *O*(**`log`** *n*)
   - like how we made find faster from BSTs to AVL trees, by doing a little extra work on each insert

2. Improve **`find`** so it becomes even faster
   - path compression: can we get from leaf to root in 1 hop?

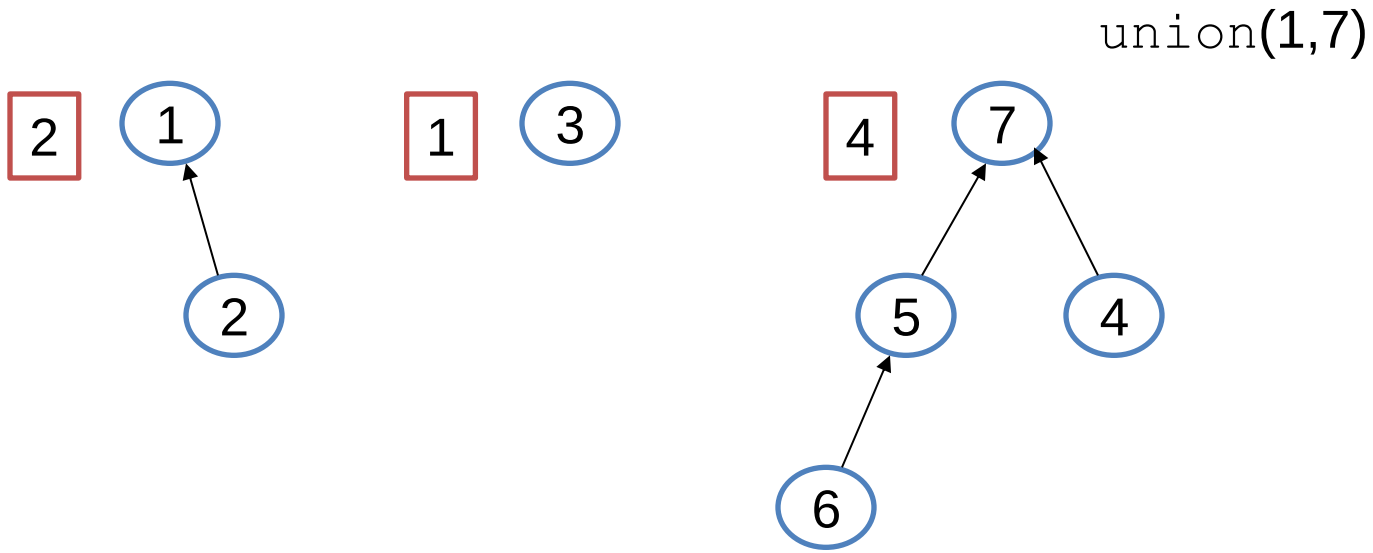
The bad case to avoid



Weighted union

Weighted union:

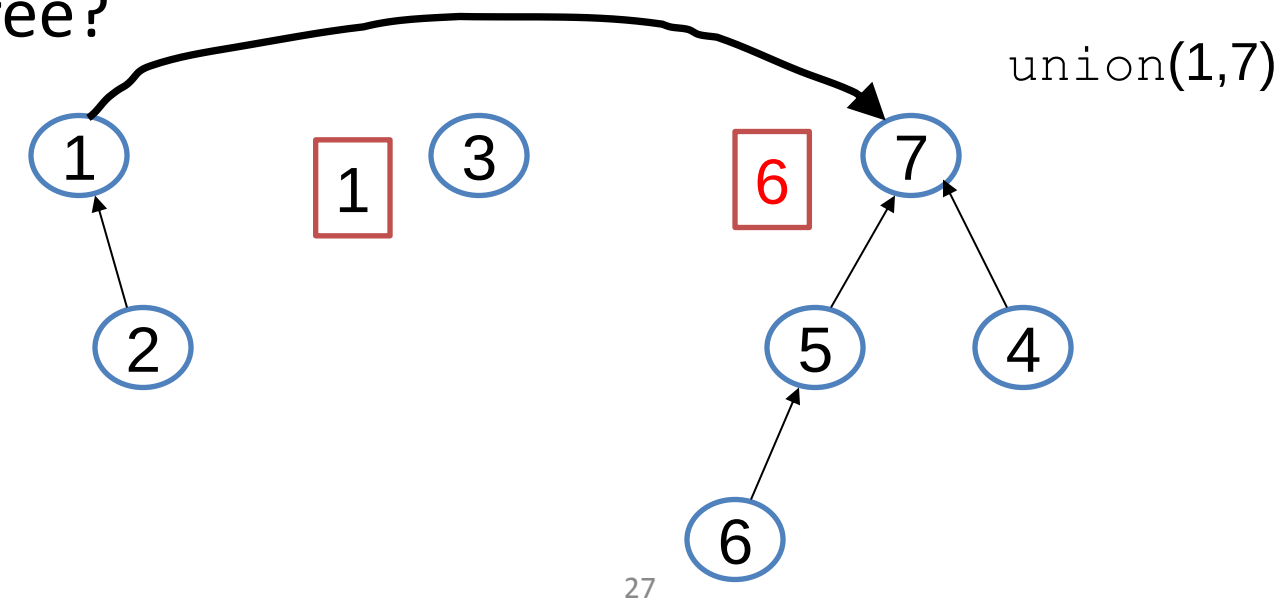
– Always point the *smaller* (total # of nodes) tree to the root of the larger tree



Weighted union

Weighted union:

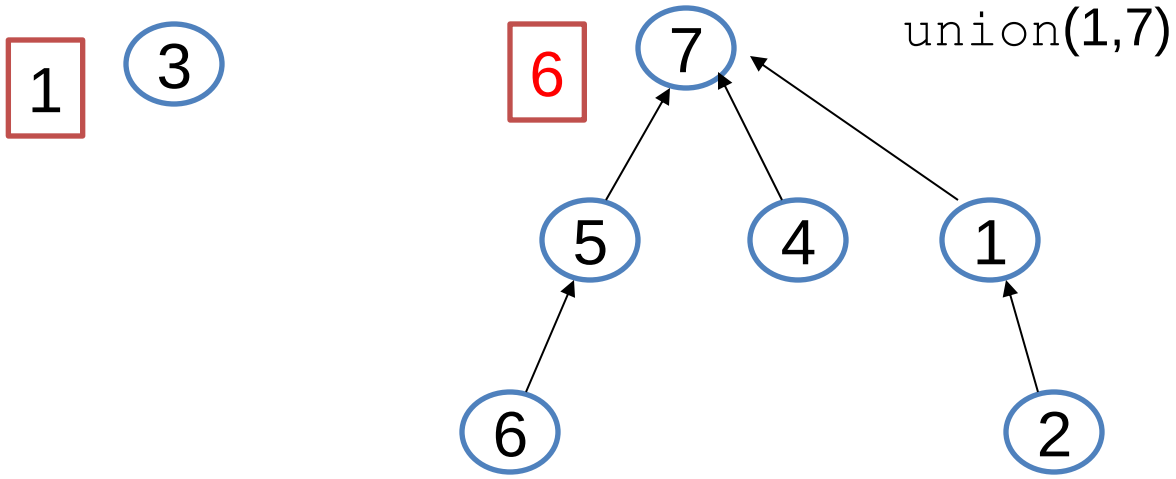
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree
- What just happened to the height of the larger tree?



Weighted union

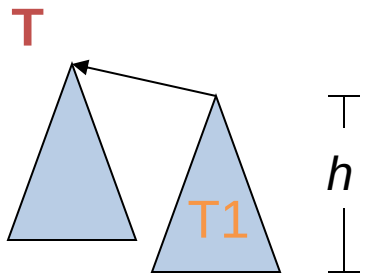
Weighted union:

- Like balancing on an AVL tree, we're trying to keep the traversal from leaf to overall root short



Intuition: The key idea

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

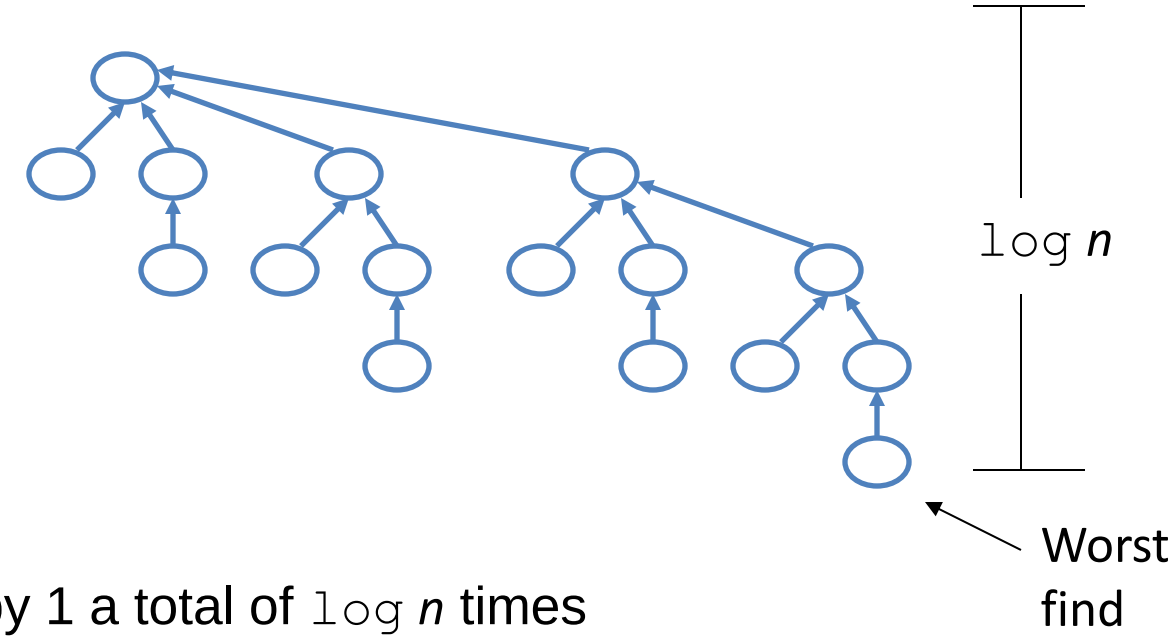


So, as usual, if number of nodes is exponential in height, then height is logarithmic in number of nodes. The height is log(N) where N is the number of nodes.

So **`find`** is *O*(**`log`** *n*)

The new worst case find

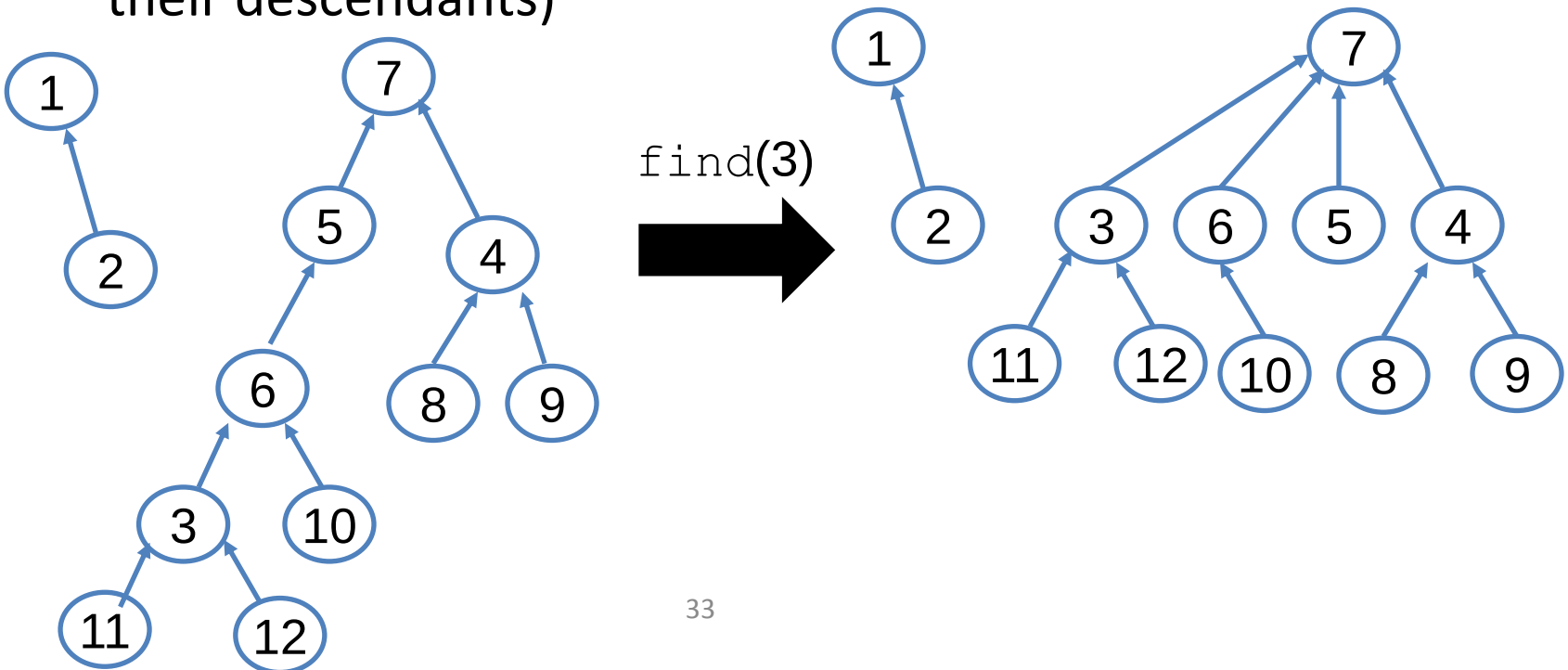
After n/2 + n/4 + …+ 1 Weighted Unions:



Height grows by 1 a total of `log` *n* times

Path compression

- Simple idea: As part of a **`find`**, change each encountered node's parent to point directly to root
  - Faster future **`find`**s 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
- Intuition:
  - How it is *almost* $O(1)$
  - total for m **`find`**s and n-1 **`union`**s is *almost* $O(m+n)$