CSE 373: Data Structures & Algorithms
Union-Find Continued

Riley Porter

Winter 2017

Course Logistics

- HW regrades will be handled by grading head TA Chloe Lathe. Regrade catalyst form up on the website in 3 places.

- AVL Tree topic summary out

- HW3 still out. Don't forget about TA office hours and the catalyst board if you get stuck.

- Midterm a week from today :O :O :O
  - Help come up with a cheat sheet

Review: Union Find Operations

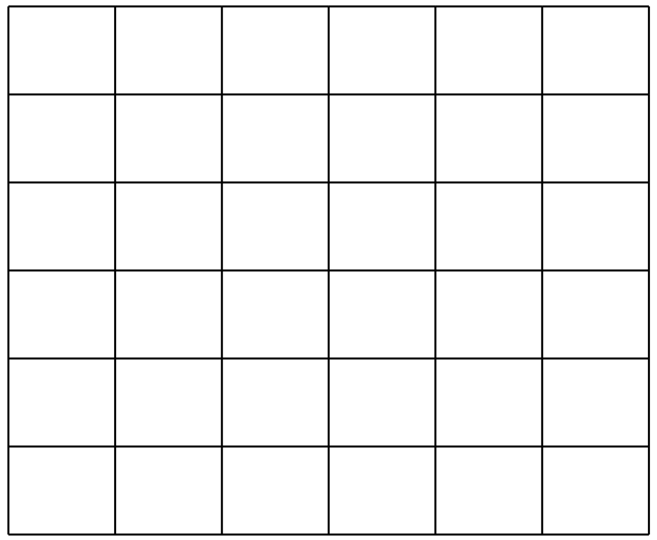
- Given an unchanging set *S*, `create` an initial partition of a set
  - Typically each item in its own subset: {a}, {b}, {c}, …
  - Give each subset a "name" by choosing a *representative element*

- 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 application: maze-building

- Build a random maze by erasing edges



Criteria:

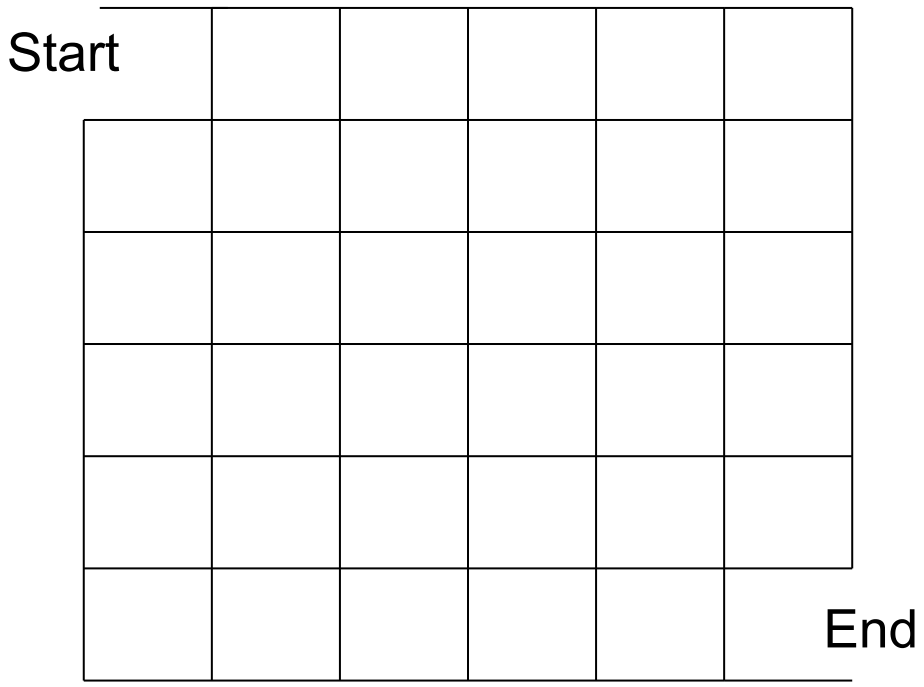
– Possible to get from anywhere to anywhere

– No loops possible without backtracking

  - After a "bad turn" have to "undo"

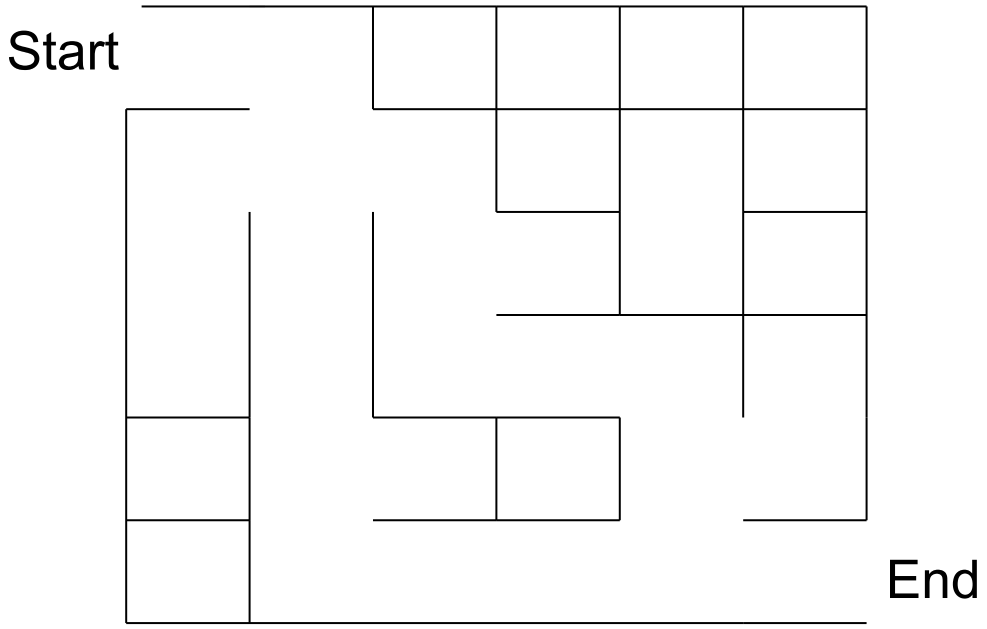
Maze building

Pick start edge and end edge



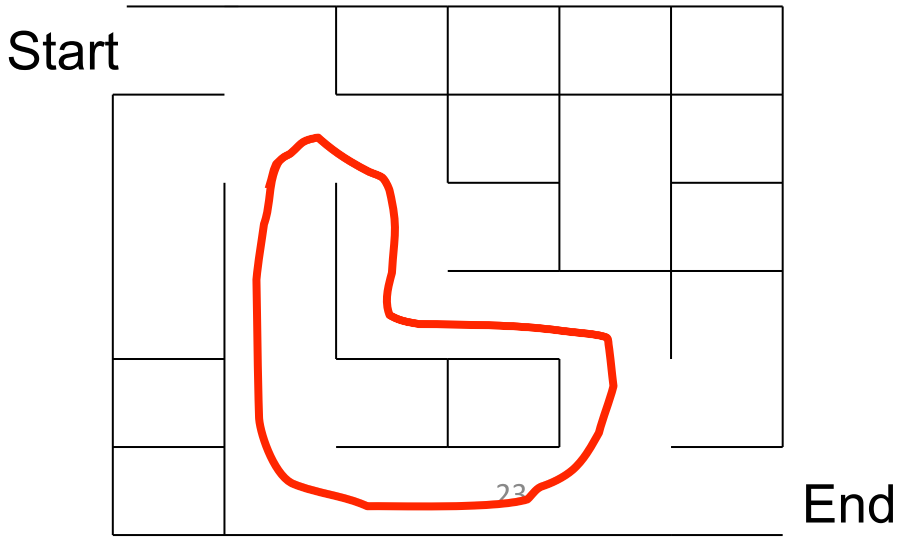
Repeatedly pick random edges to delete

One approach: just keep deleting random edges until you can get from start to finish



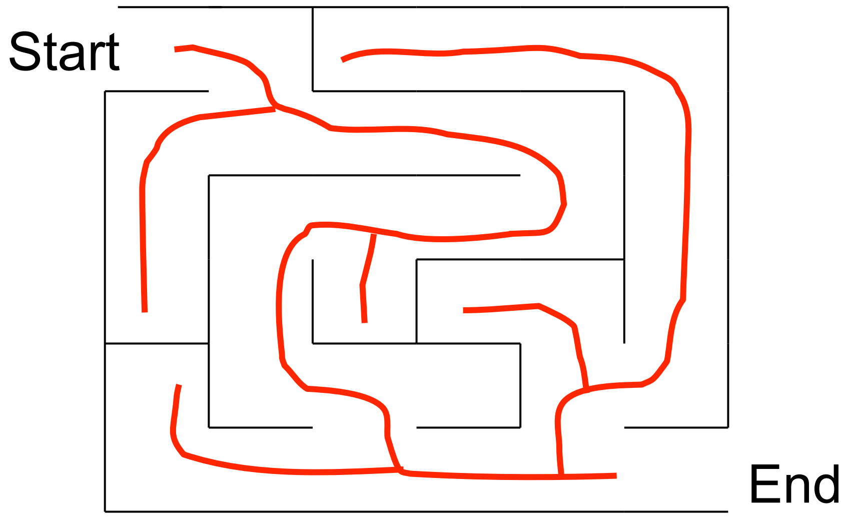
Problems with this approach

1. How can you tell when there is a path from start to finish?
   – We do not really have an algorithm yet (Graphs)
2. We have *cycles*, which a "good" maze avoids
3. We can't get from anywhere to anywhere else



Revised approach

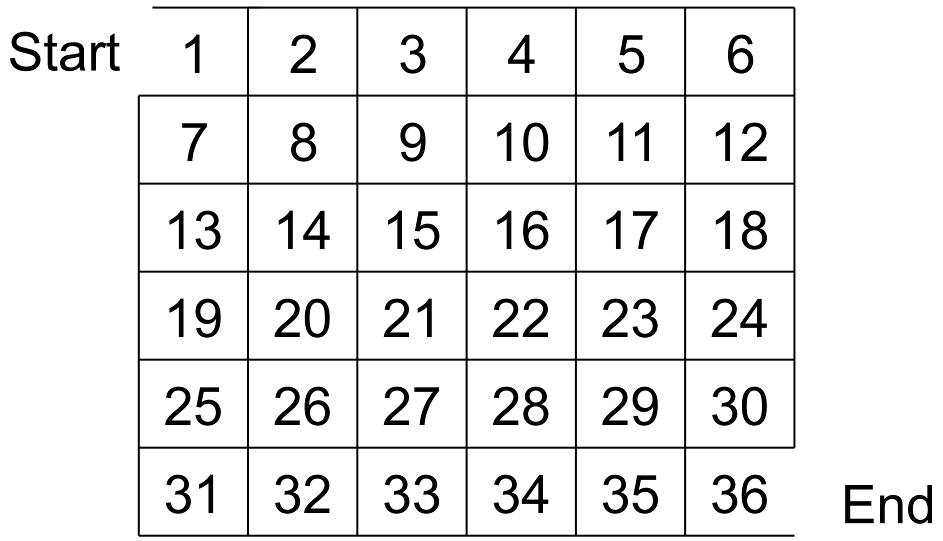
- Consider edges in random order
- But only delete them if they introduce no cycles (how? TBD)
- When done, will have one way to get from any place to any other place (assuming no backtracking)



- Notice the funny-looking *tree* in red

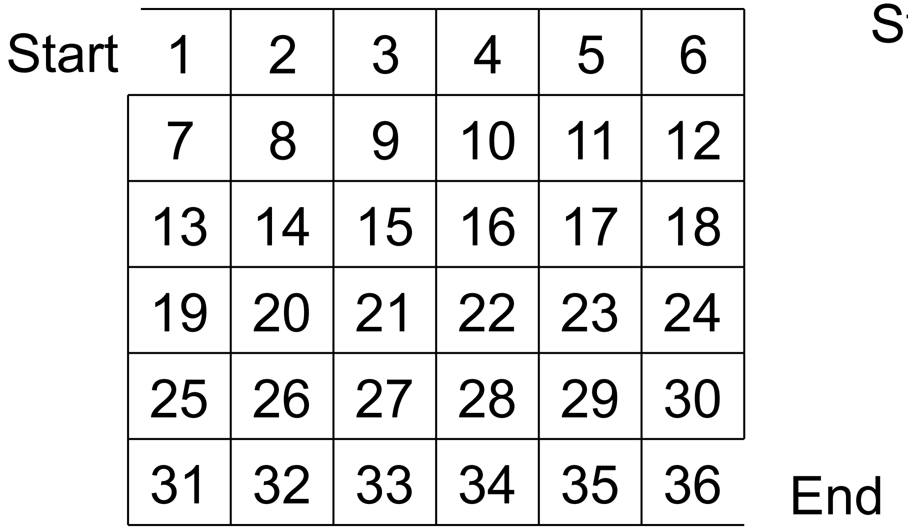
Cells and edges

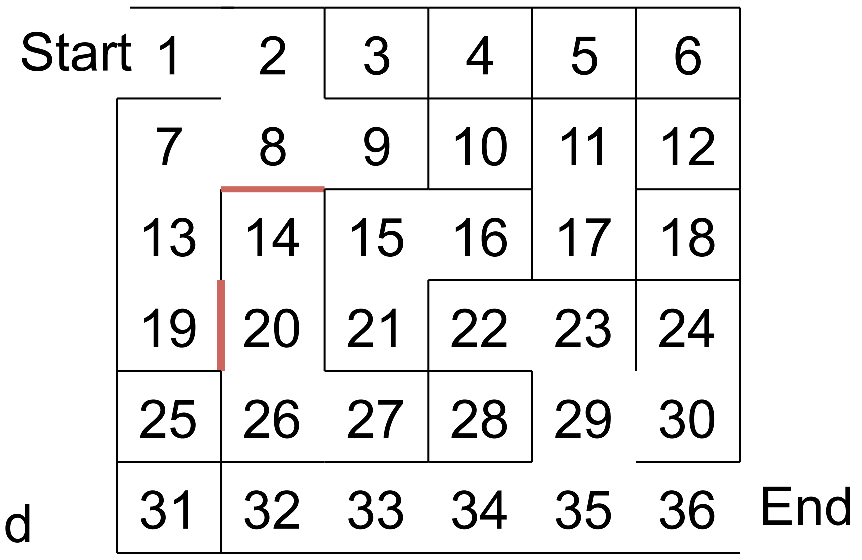
- Let's number each cell
  - 36 total for 6 x 6
- An (internal) edge (x,y) is the line between cells x and y
  - 60 total for 6x6: (1,2), (2,3), …, (1,7), (2,8), …



The trick

- Partition the cells into **disjoint sets**: "are they connected"
  - Initially every cell is in its own subset
- If an edge would connect two different subsets:
  - then remove the edge and **union** the subsets
  - else leave the edge because removing it makes a cycle





Pseudocode of the algorithm

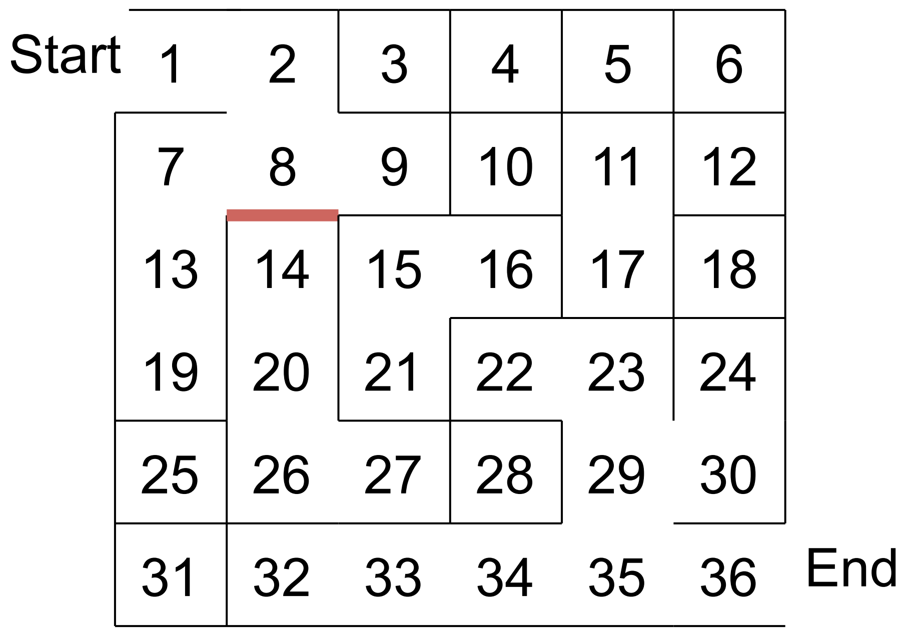
- Partition = **disjoint sets** of connected cells, initially each cell in its own 1-element set
- Edges = **set** of edges not yet processed, initially all (internal) edges
- Maze = **set** of edges kept in maze (initially empty)

```
// stop when possible to get anywhere
while Partition has more than one set {
    pick a random edge (cell_1,cell_2) to remove from
    Edges
    set_1 = find(cell_1)
    set_2 = find(cell_2)
    if set_1 == set_2:
        // same subset, do not create cycle
         add (cell_1, cell_2) to Maze
    else:
        // do not put edge in Maze, connect subsets
        union(set_1, set_2)
 }
```

Add remaining members of Edges to Maze, then output Maze

pick random Edge step

Pick (8,14)



{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
{18}
{25}
{28}
{31}
{22,23,24,29,30,32
33,34,35,36}

Example pick random Edge step

Partition:
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
{18}
{25}
{28}
{31}
{22,23,24,29,30,32
33,34,35,36}

Chosen
Edge: (8, 14)

Find(8) = 7
Find(14) = 20

Union(7,20)

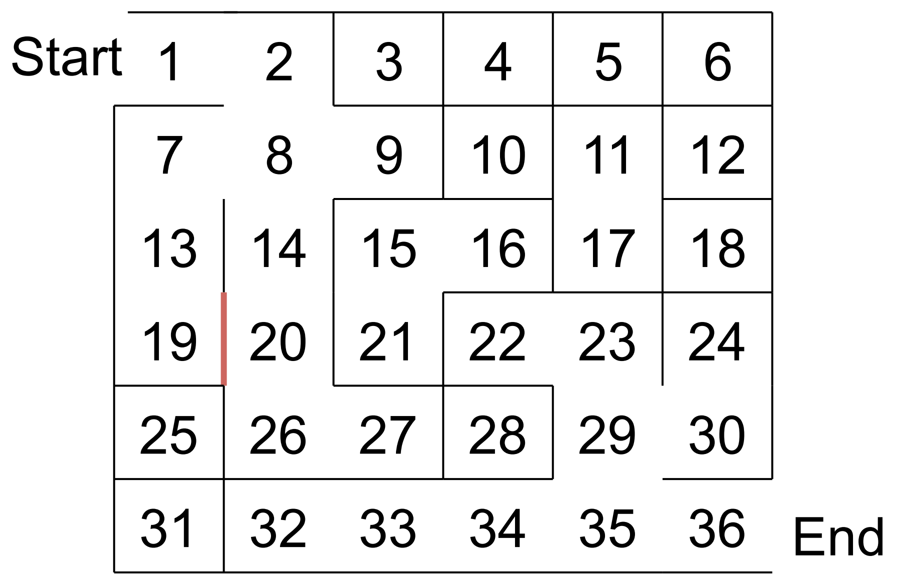
$\Rightarrow$

Since we unioned the two sets, we “deleted” the edge and don’t add the edge to our Maze

Partition:
{1,2,7,8,9,13,19,14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
{18}
{25}
{28}
{31}
{22,23,24,29,30,32
33,34,35,36}

Add edge to Maze step

Pick (19,20)

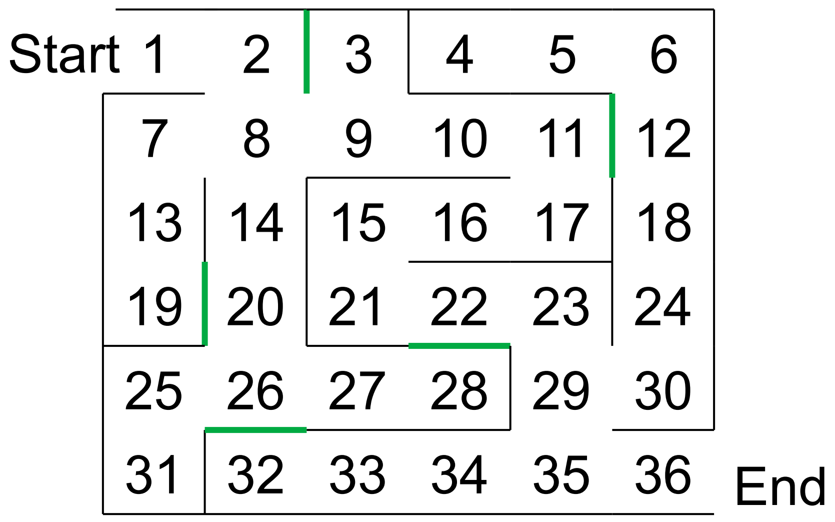


Since we didn't union the sets together, we don't want to delete this edge (it would introduce a cycle). We add the edge (19,20) to our Maze.

Partition:
{1,2,7,8,9,13,19,14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
{18}
{25}
{28}
{31}
{22,23,24,29,30,32
33,34,35,36}

At the end

- Stop when Partition has one set
- Suppose green edges are already in Maze and black edges were not yet picked
  - Add all black edges to Maze



Partition:
{1,2,3,4,5,6,7,… 36}

Applications / Thoughts on Union-Find

- Maze-building is cute ☺ and 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
- 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

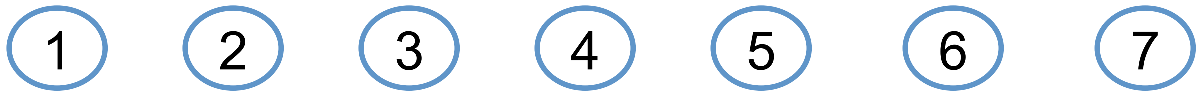
How should we “draw” this data structure?

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

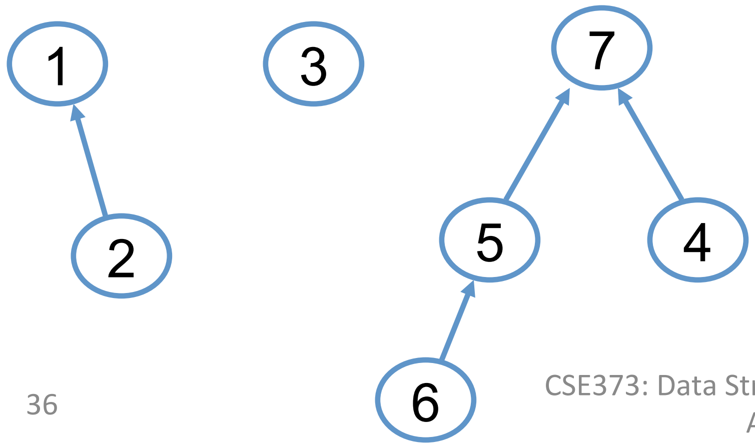
- We can still implement the code in different ways, just like heaps can be implemented with an array even though we think of them as a tree structure.

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

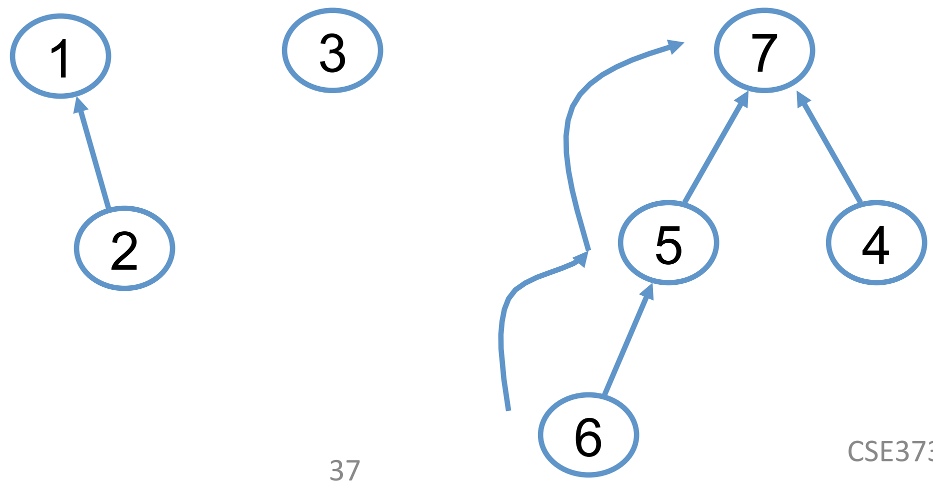


Find

`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

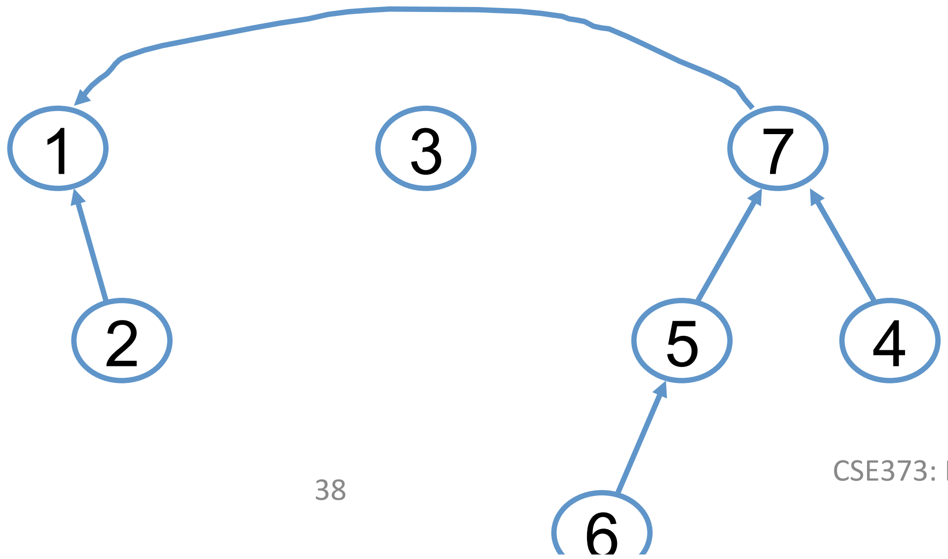


Union

`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)

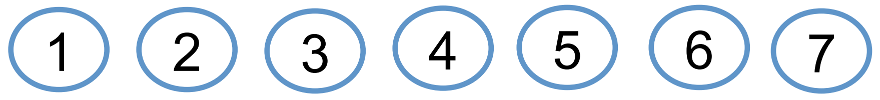


Okay, how can we represent it internally?

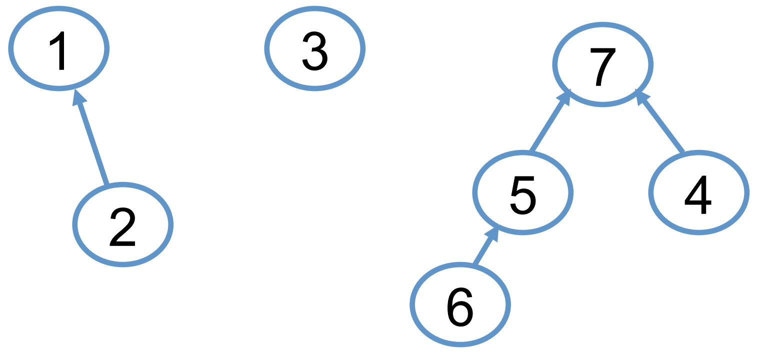
- Important to remember from the operations:
  - We assume O(1) access to *each* node
  - Ideally, we want the traversal from leaf to root of each tree to be as short as possible (the find operation depends on this traversal)
  - We don't want to copy a bunch of nodes to a new tree on each union, we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

Simple implementation

- If set elements are contiguous numbers (e.g., 1,2,…,*n*), use an array of length *n* called **`up`**
  - Starting at index 1 on slides
  - Put in array index of parent, with 0 (or -1, etc.) for a root
- Example:



	1	2	3	4	5	6	7
up	0	0	0	0	0	0	0



	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0

- If set elements are not contiguous numbers, could have a separate dictionary hash map to map elements (keys) to numbers (values)

Implement operations

```
// assumes x in range 1,n
int find(int x) {
  while(up[x] != 0) {
     x = up[x];
  }
  return x;
}
```

```
// assumes x,y are roots
void union(int x, int y){
   // y = find(y)
   // x = find(x)
   up[y] = x;
}
```

- Worst-case run-time for **`union`**?
- Worst-case run-time for **`find`**?
- Worst-case run-time for *m* **`find`**s and *n*-1 **`union`**s?

Implement operations

```
// assumes x in range 1,n
int find(int x) {
  while(up[x] != 0) {
     x = up[x];
  }
  return x;
}
```

```
// assumes x,y are roots
void union(int x, int y){
   // y = find(y)
   // x = find(x)
   up[y] = x;
}
```

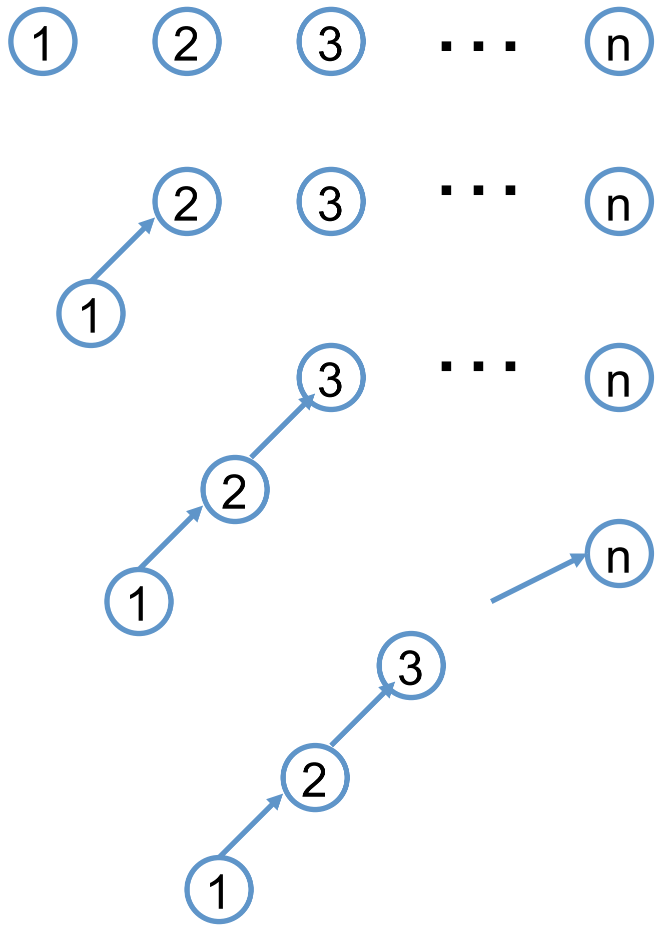
- Worst-case run-time for **`union`**? *O*(1) (with our assumption…)
- Worst-case run-time for **`find`**? *O(n)*
- Worst-case run-time for *m* **`find`**s and *n*-1 **`union`**s? *O(m*n)*

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



`union`(2,1)

`union`(3,2)

⋮

`union`(*n*,*n*-1)

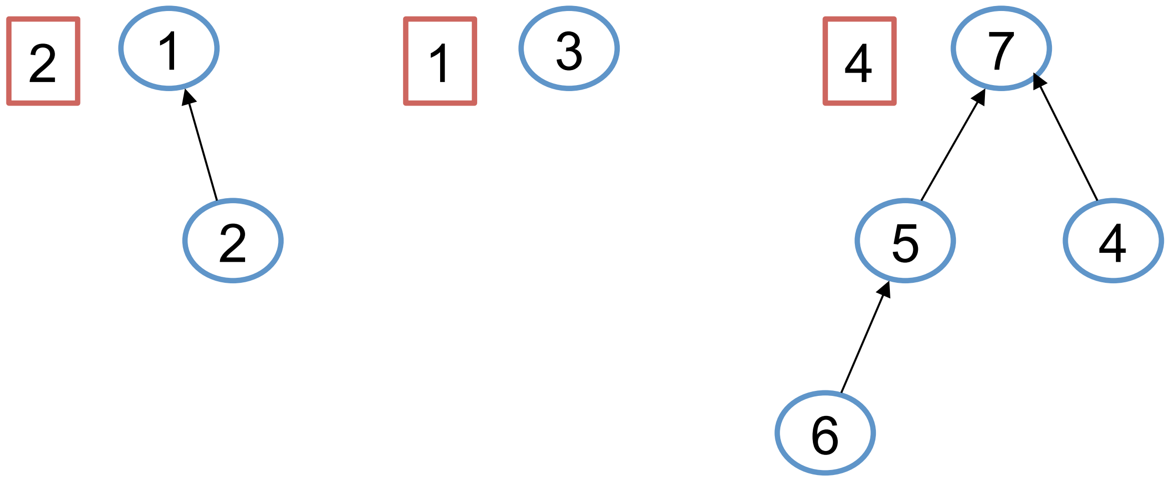
`find`(1)   *n* steps!!

Weighted union

Weighted union:

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

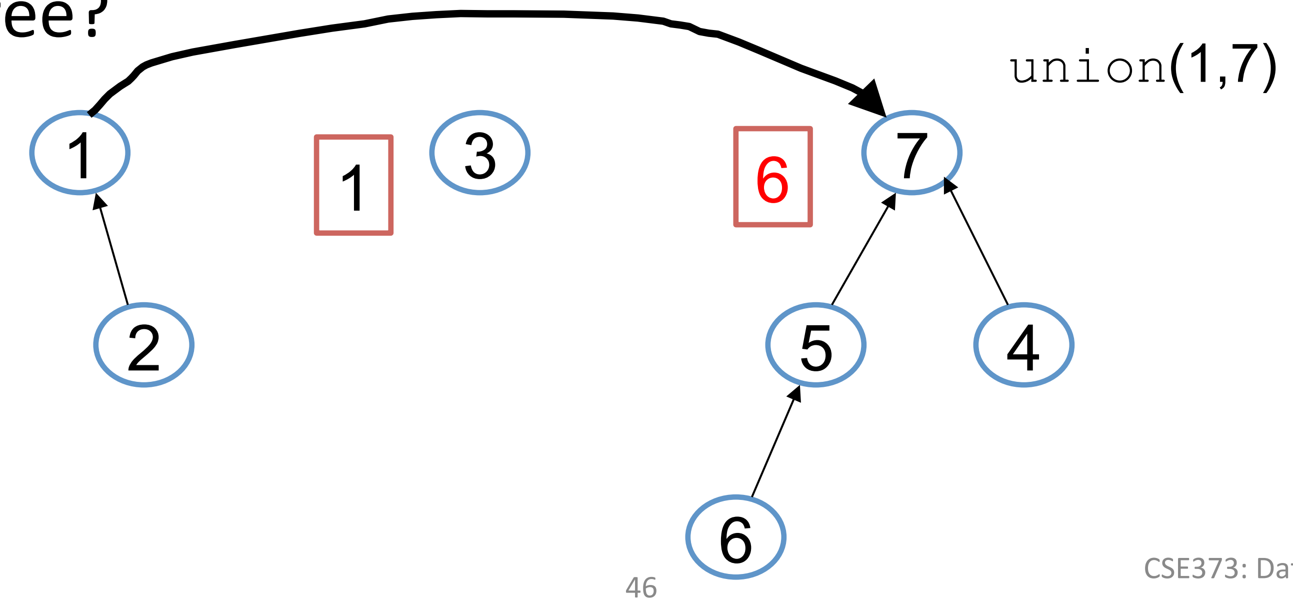
`union`(1,7)



Weighted union

Weighted union:

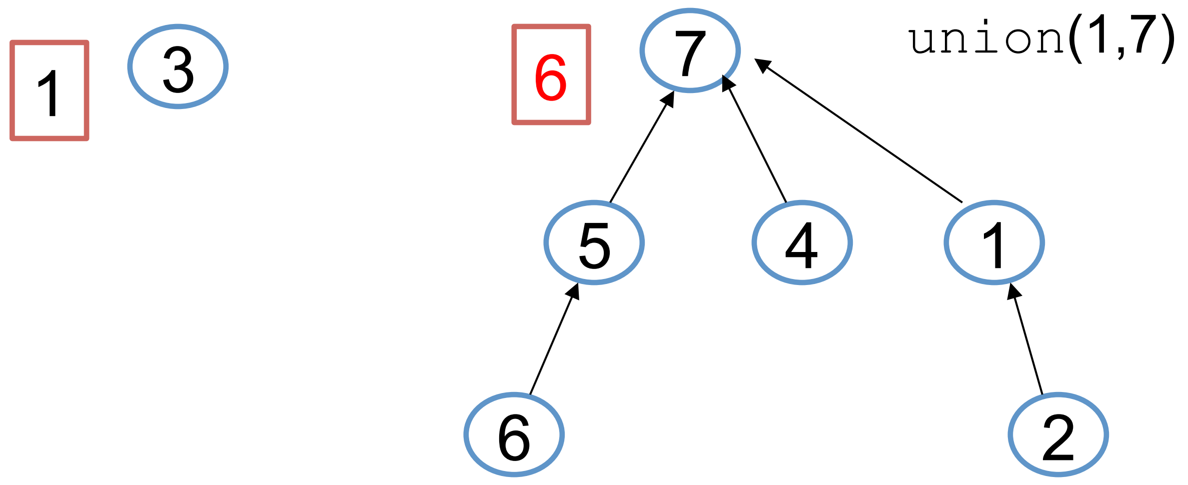
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree
- What just happened to the height of the larger tree?



Weighted union

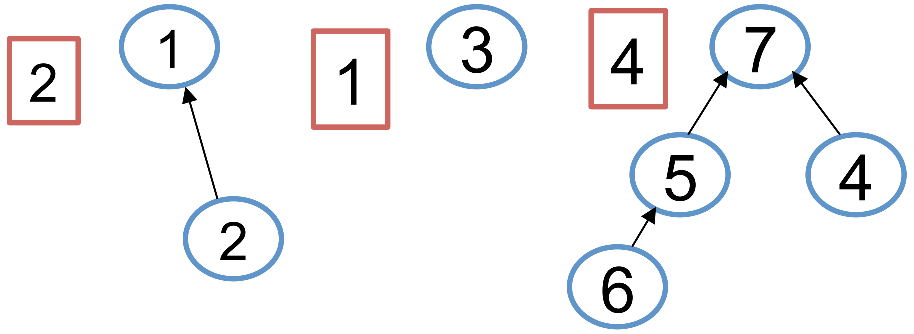
Weighted union:

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

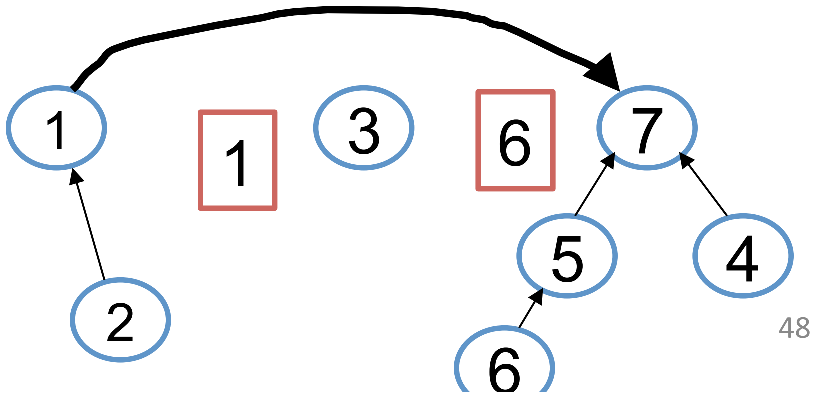


Array implementation

Keep the *weight* (number of nodes in a second array). Or have one array of objects with two fields. Could keep track of *height*, but that's harder. *Weight* gives us an approximation.



	1	2	3	4	5	6	7
parent	0	1	0	7	7	5	0
weight	2		1				4

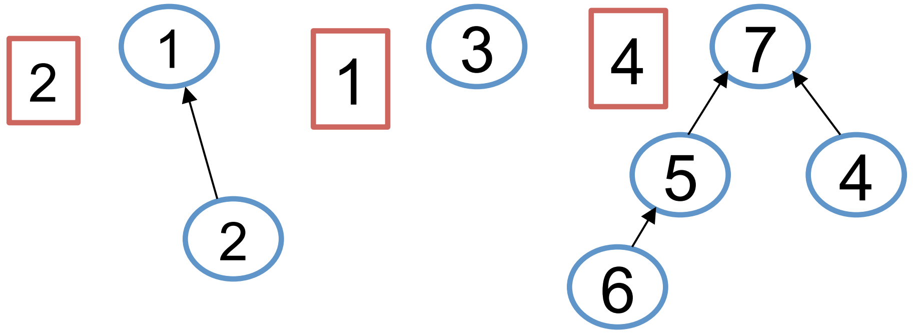


	1	2	3	4	5	6	7
parent	7	1	0	7	7	5	0
weight	2		1				6

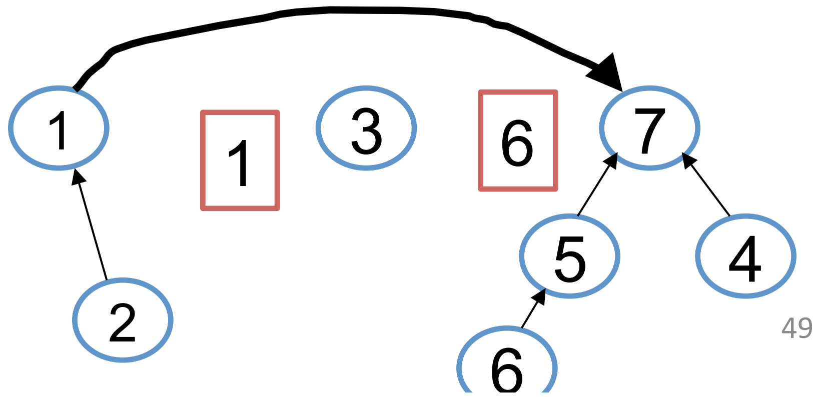
Nifty trick

Actually we do not need a second array…

- Instead of storing 0 for a root, store negation of weight. So parent value < 0 means a root.



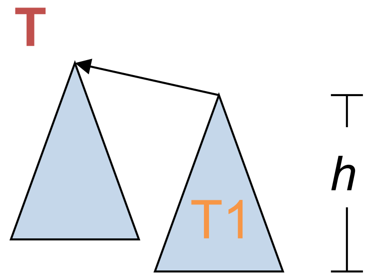
	1	2	3	4	5	6	7
parent or weight	-2	1	-1	7	7	5	-4



	1	2	3	4	5	6	7
parent or weight	7	1	-1	7	7	5	-6

Intuition: The key idea

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

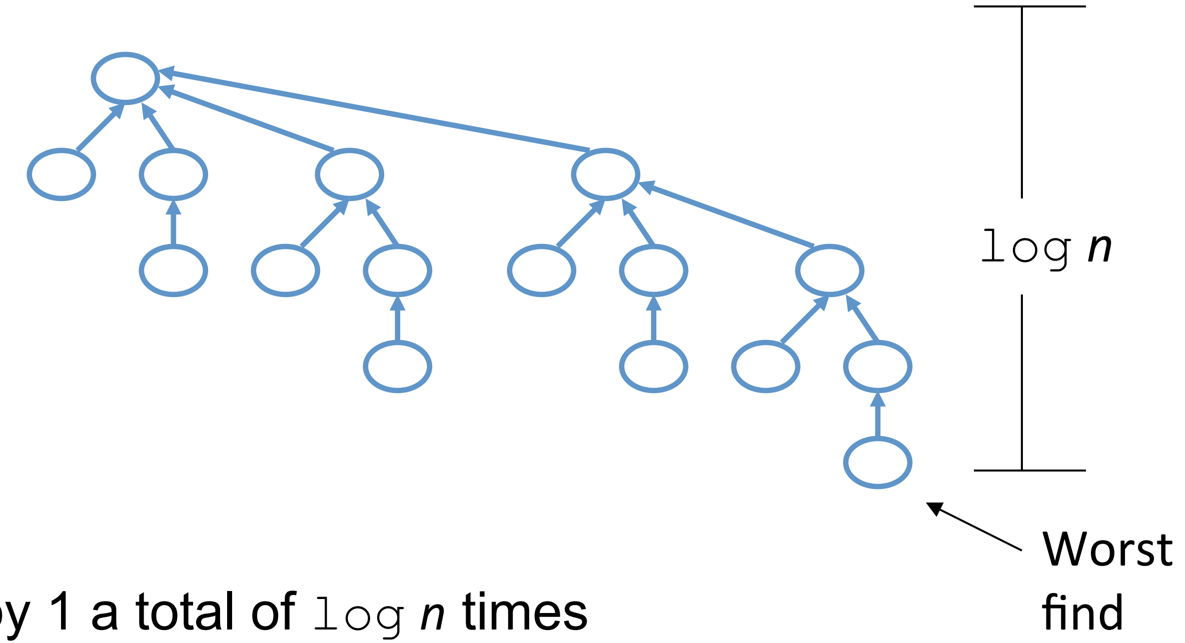


So, as usual, if number of nodes is exponential in height, then height is logarithmic in number of nodes. The height is log(N) where N is the number of nodes.

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

The new worst case find

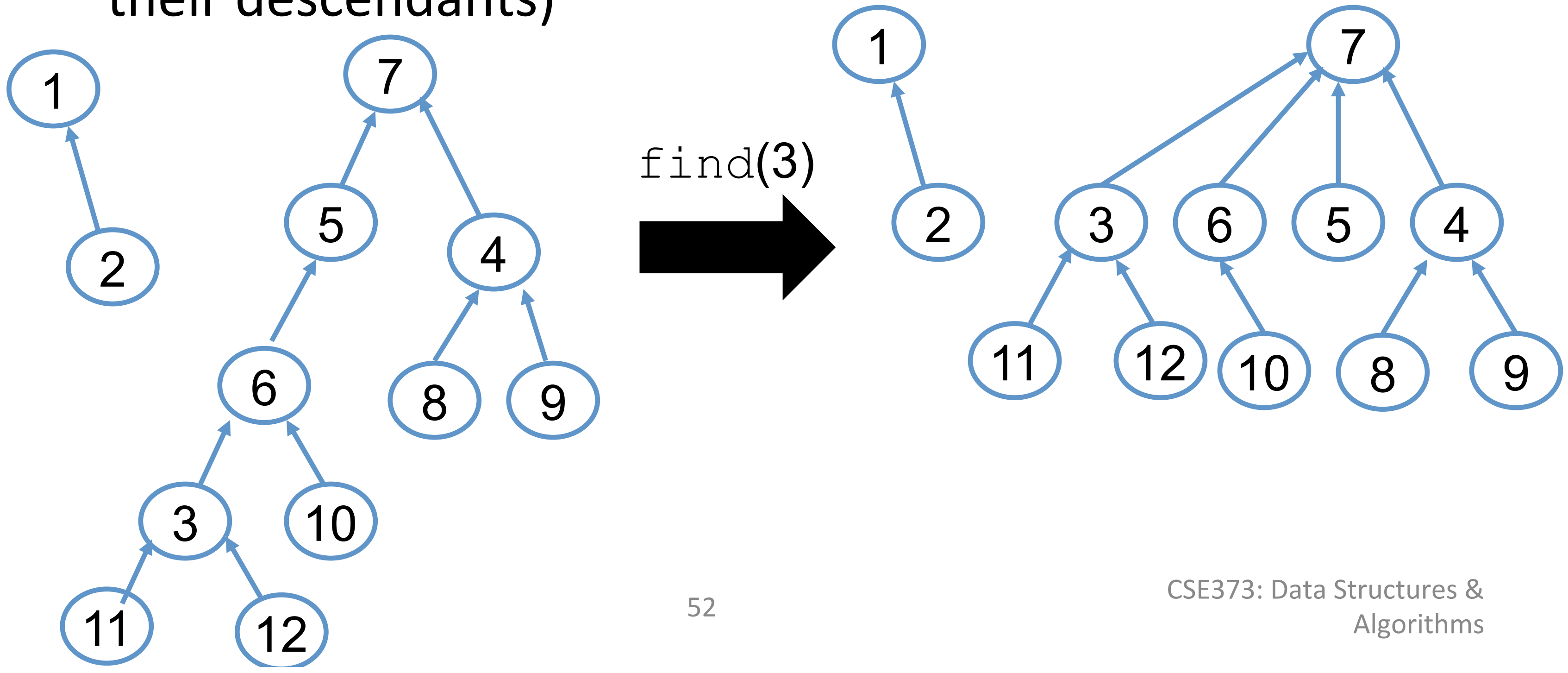
After n/2 + n/4 + …+ 1 Weighted Unions:



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)



Pseudocode

```
// performs path compression
find(element)
    // find root
    root = element
    while parent[root] > 0 // not overall root yet
        root = parent[root]

   if element == root
        // element already the top of a tree
        return root
    else // let's compress the path
        old_parent = parent[element]
        while (old_parent != root)
            parent[element] = root
            element = old_parent
            old_parent = parent[element]
        return root
```

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* $O(m+n)$

Today's Takeaways

- How to implement the Union-Find ADT and the create, union, and find operations
- The optimizations that can help make it faster
- The intuitions on runtime analysis for the different operations and why the optimizations help make it faster