CSE373: Data Structures & Algorithms
Lecture 10: Implementing Union-Find

Lauren Milne

Summer 2015

Announcements

- Homework 3 due in ONE week…Wednesday July 22nd!
- TA Sessions will remain the same time.
- Midterm on Friday
  - Exam review Thursday 5-6 pm in EEB 125

The plan

Last lecture:

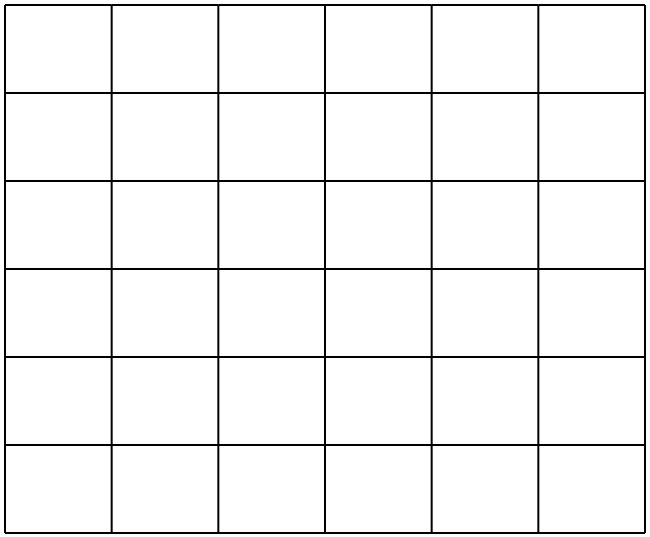
- Disjoint sets
- The union-find ADT for disjoint sets

Today's lecture:

- Finish maze application
- Basic implementation of the union-find ADT with "up trees"
- Optimizations that make the implementation much faster

Example application: maze-building

- Build a random maze by erasing edges



  - Possible to get from anywhere to anywhere
    - Including “start” to “finish”
  - No loops possible without backtracking
    - After a “bad turn” have to “undo”

The algorithm

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

while P has more than one set {

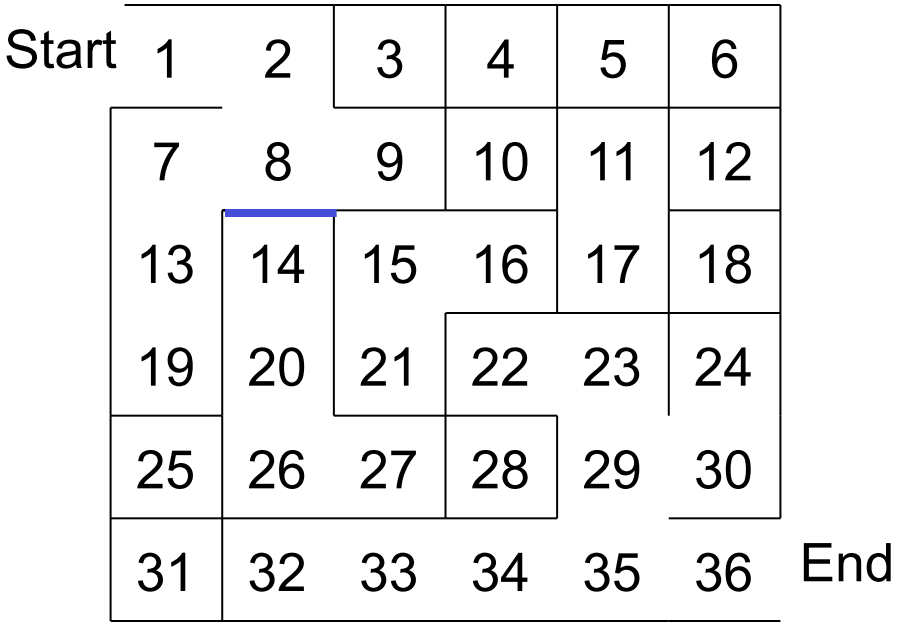
- Pick a random edge (x,y) to remove from E
- u = **`find`**(x)
- v = **`find`**(y)
- if u==v
  add (x,y) to M // same subset, leave edge in maze, do not create cycle
  else
  `union`(u,v) // connect subsets, remove edge from maze

}

Add remaining members of E to M, then output M as the maze

Example

Pick edge (8,14)
Find(8) = 7
Find(14) = 20
Union(7,20)



P
{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

P
{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}

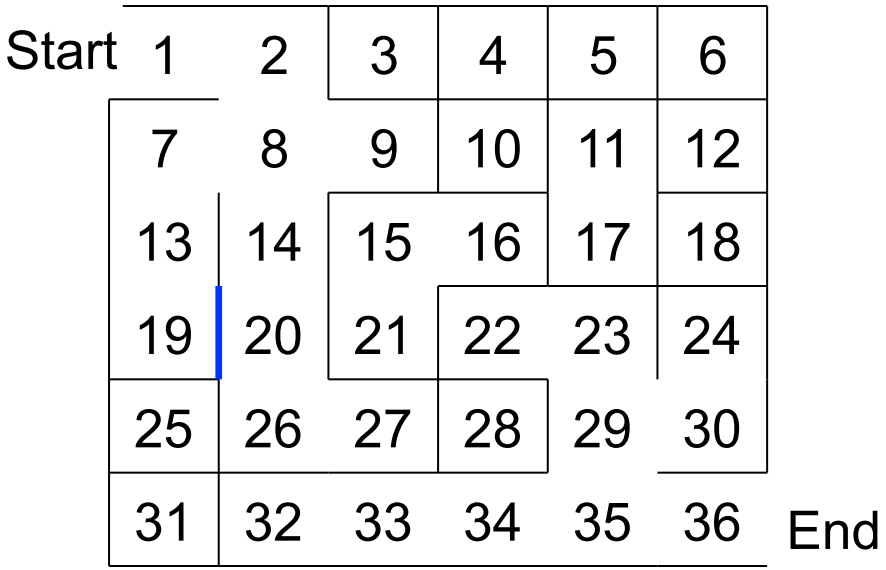
Find(8) = 7
Find(14) = 20

Union(7,20)

P
{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}

Example: Add edge to M step

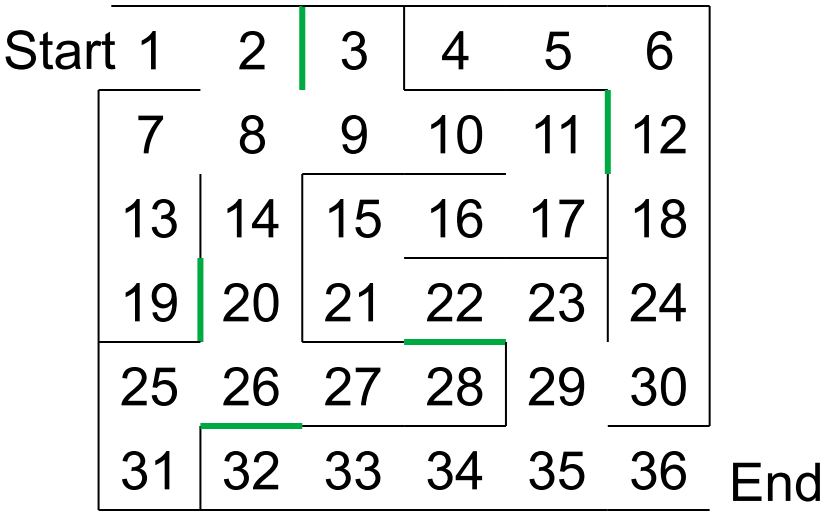
Pick edge (19,20)
Find (19) = 7
Find (20) = 7
Add (19,20) to M



P
{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 of while loop

- Stop when P has one set (i.e. all cells connected)
- Suppose green edges are already in M and black edges were not yet picked
  - Add all black edges to M



P

{1,2,3,4,5,6,7,… 36}

Done! ☺

Union-Find ADT

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

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

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

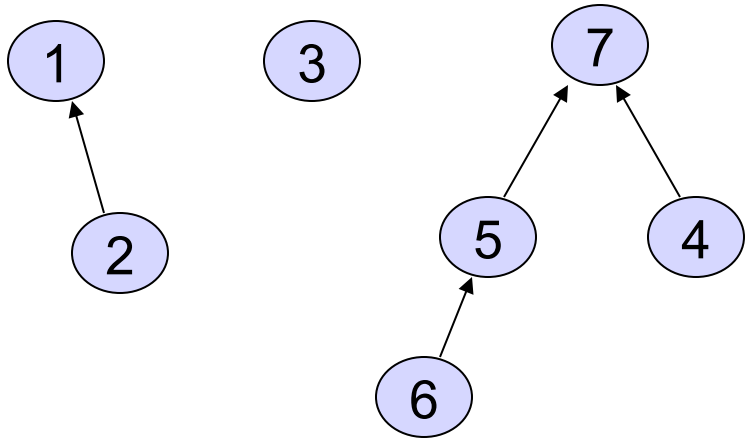
Implementation – our goal

- Start with an initial partition of *n* subsets
  - Often 1-element sets, e.g., {1}, {2}, {3}, …, {*n*}
- May have *m* **`find`** operations
- May have up to *n*-1 **`union`**
  - After *n*-1 **`union`** operations, every **`find`** returns same 1 set

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

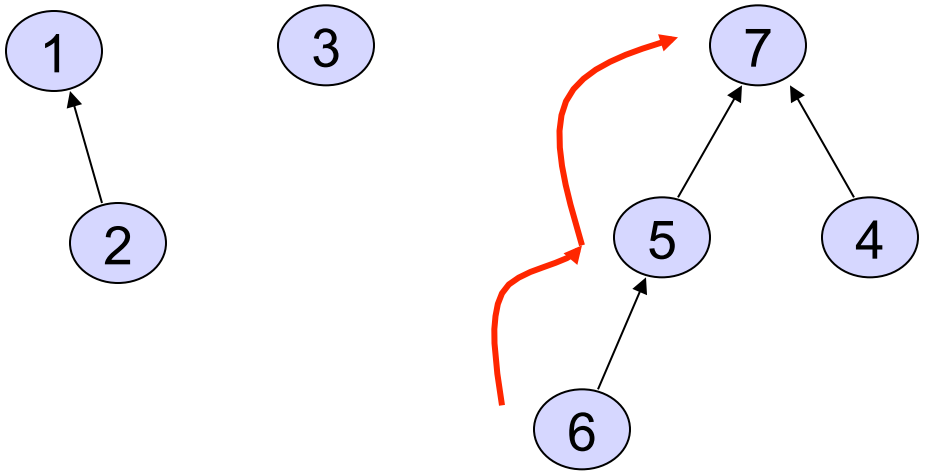


Find

`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

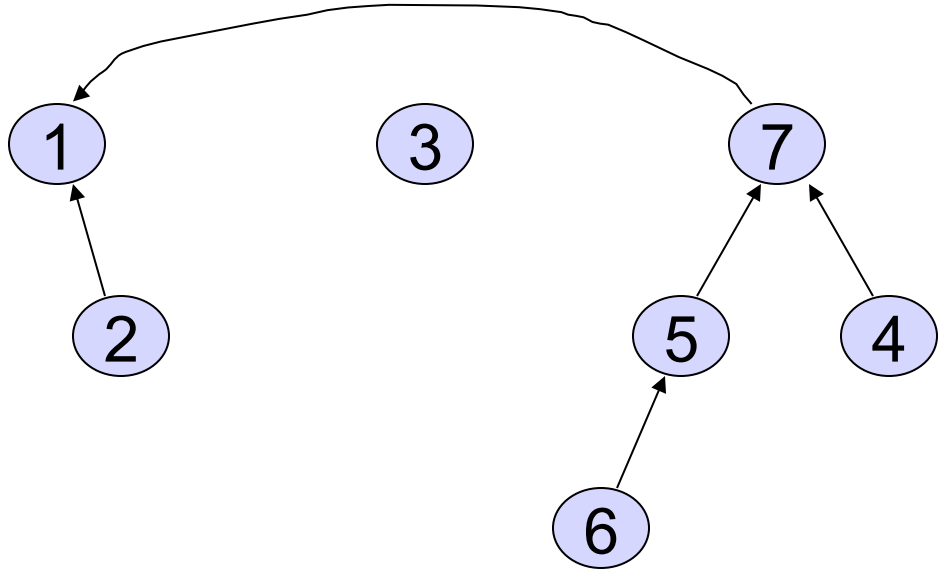


Union

`union(x,y)`:

- Assume **`x`** and **`y`** are roots
  - Else **`find`** the roots of their trees
- Change root of one to have parent be the root of the other
  - Notice no limit on branching factor

`union`(1,7)

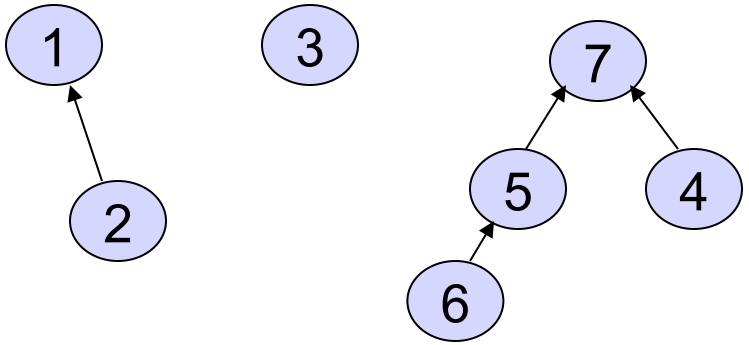


Simple implementation

- If set elements are contiguous numbers (e.g., 1,2,…,*n*), use 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

- Example:

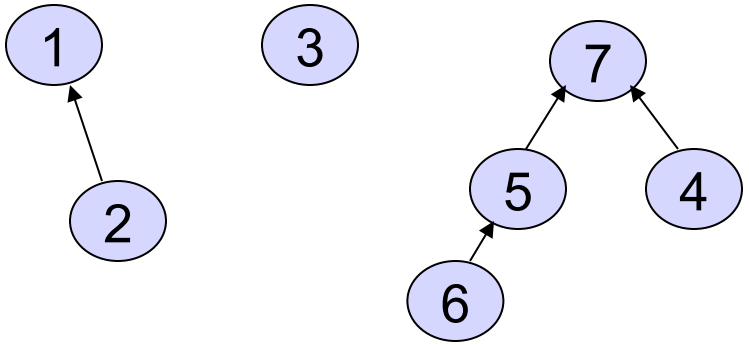


	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0

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){
   up[y] = x;
}
```



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

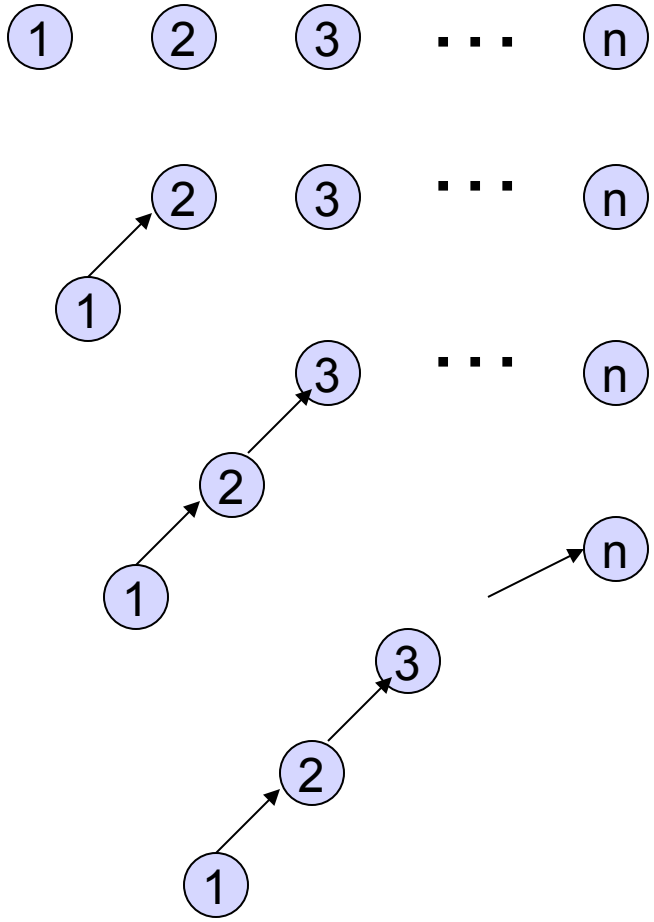
- Worst-case run-time for **`union`**? $O(1)$
- Worst-case run-time for **`find`**? $O(n)$
- Worst-case run-time for *m* **`finds`** and *n*-1 **`unions`**? $O(m*n)$

Two key optimizations

1. Improve **`union`** so it stays *O(1)* but makes **`find`** *O*(**`log`** *n*)
   - So *m* **`find`**s and *n*-1 **`union`**s 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* **`find`**s and *n*-1 **`union`**s ***almost*** *O*(*m* + *n*)
   - *Path-compression:* connect directly to root during finds

The bad case to avoid



union(2,1)

union(3,2)

:

.

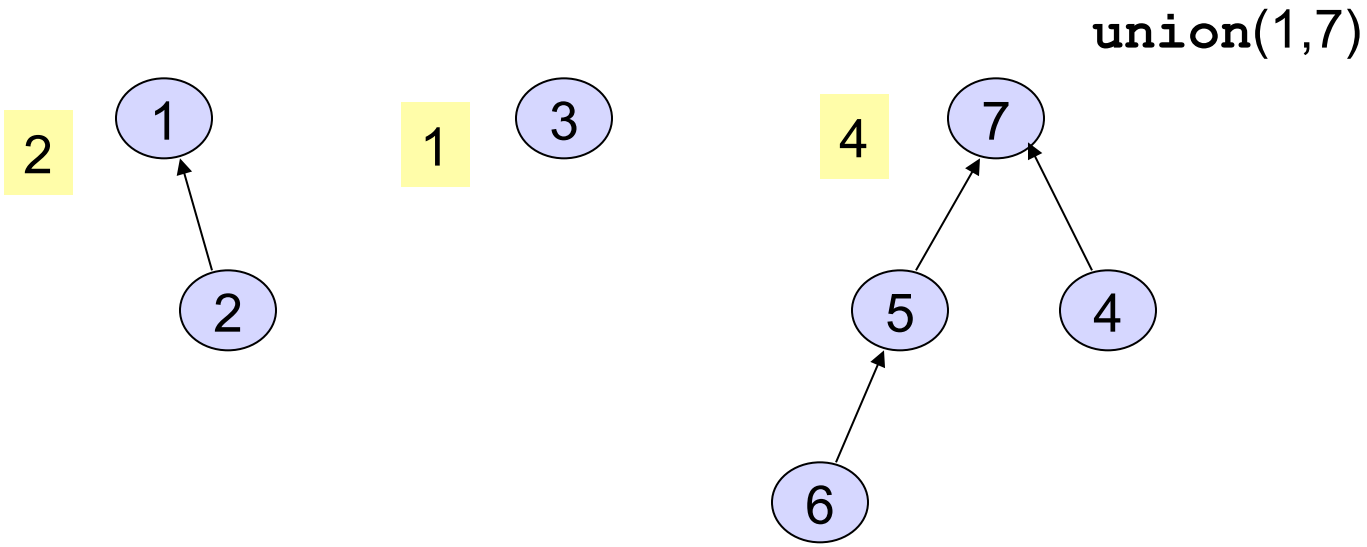
union(*n*,*n*-1)

find(1) = *n* steps!!

Union-by-size

Union-by-size:

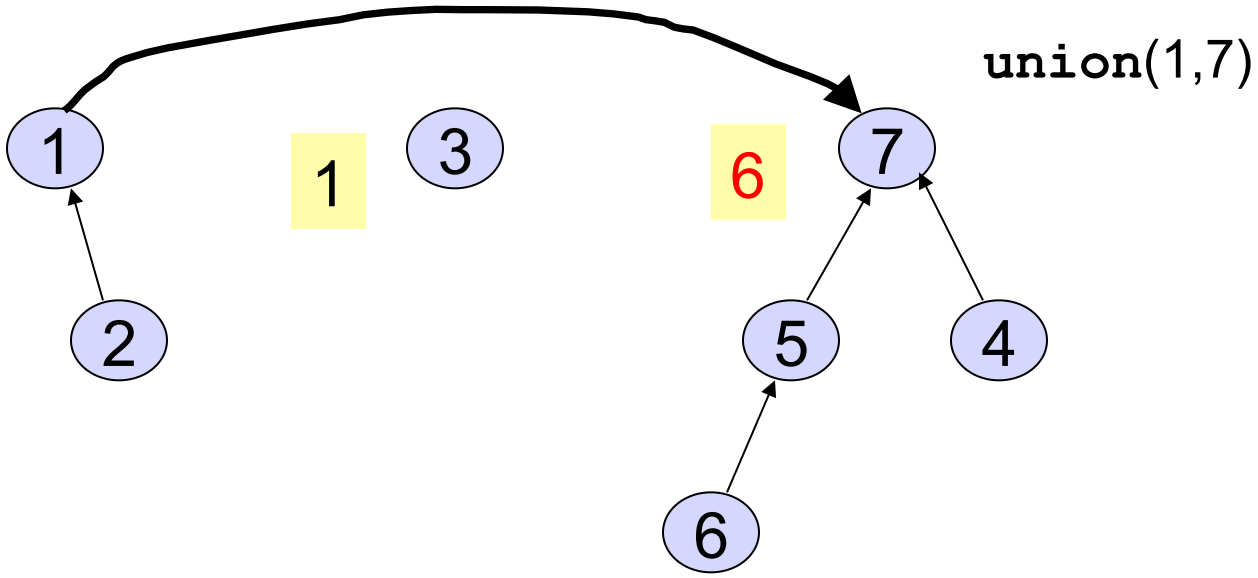
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree



Union-by-size

Union-by-size:

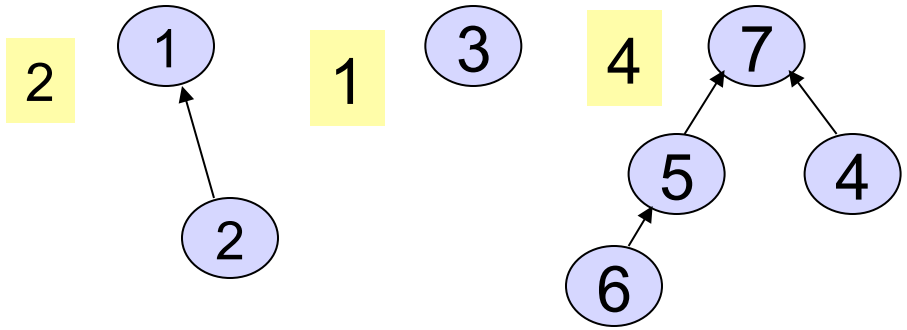
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree



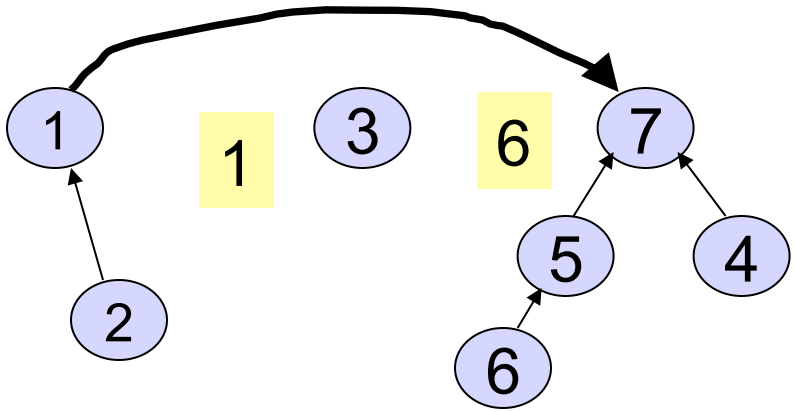
Array implementation

Keep the size (number of nodes in a second array)

- Or have one array of objects with two fields



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

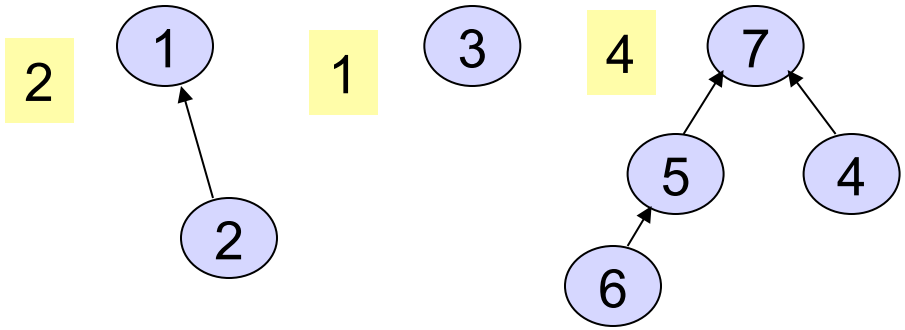


	1	2	3	4	5	6	7
up	7	1	0	7	7	5	0
weight			1				6

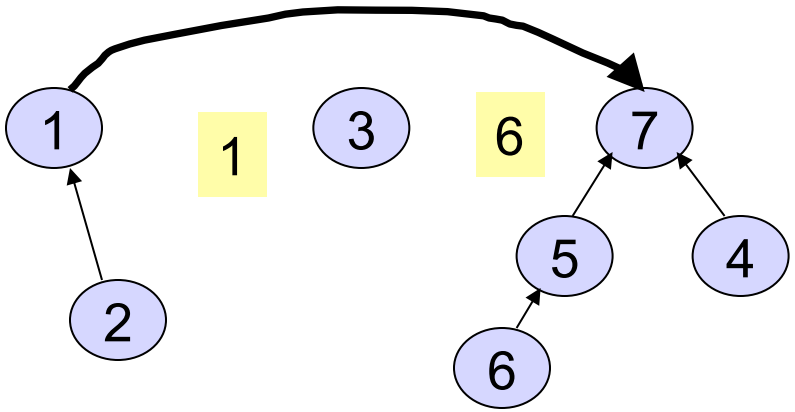
Nifty trick

Actually we do not need a second array…

- Instead of storing 0 for a root, store negation of size
- So up value < 0 means a root

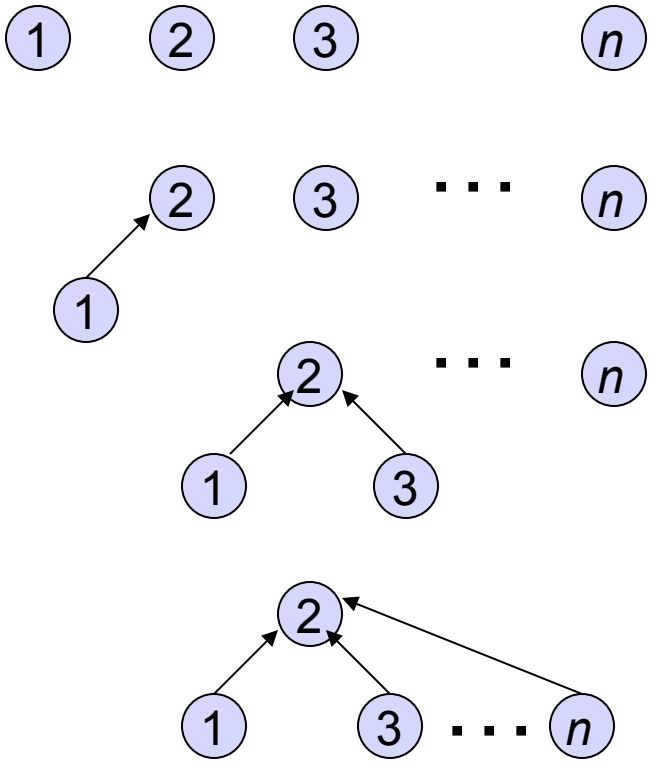


	1	2	3	4	5	6	7
up	-2	1	-1	7	7	5	-4



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

The Bad case? Now a Great case…



General analysis

- Showing 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 "obvious"
  - **`find`** is now $O($**`log`** $n)$

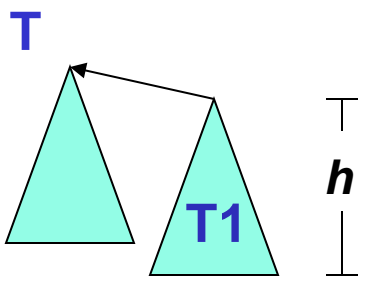
- Claim: If we use union-by-size, an up-tree of height h has at least 2^h nodes
  - Proof by induction on h…

Exponential number of nodes

P(*h*)= With union-by-size, up-tree of height *h* has at least 2^h nodes

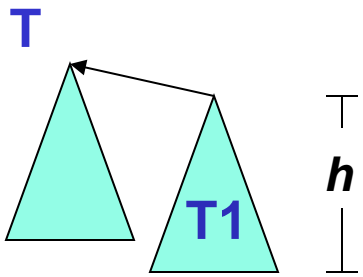
Proof by induction on *h*…

- 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 (assumption)
  - And T has *at least* as many nodes not in T1 than in T1
    - Else union-by-size 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

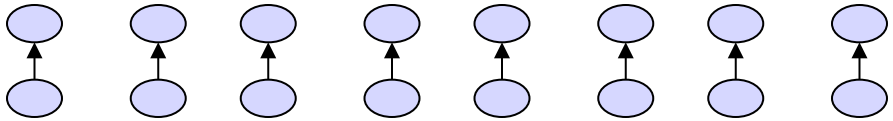


As usual, if number of nodes is exponential in height,
then height is logarithmic in number of nodes

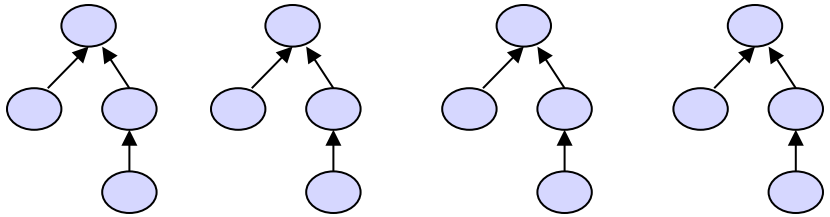
So **`find`** is $O($**`log`** $n)$

The new worst case

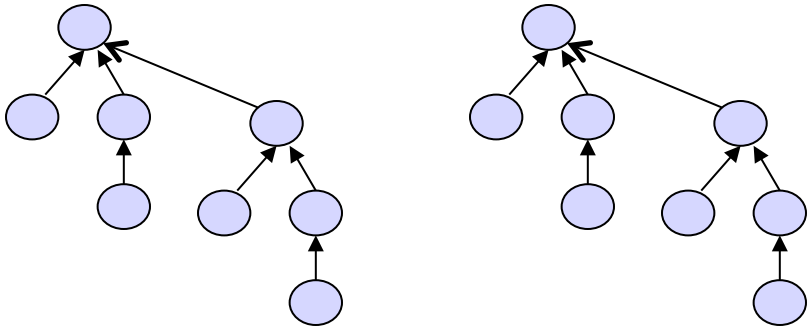
n/2 Unions-by-size



n/4 Unions-by-size

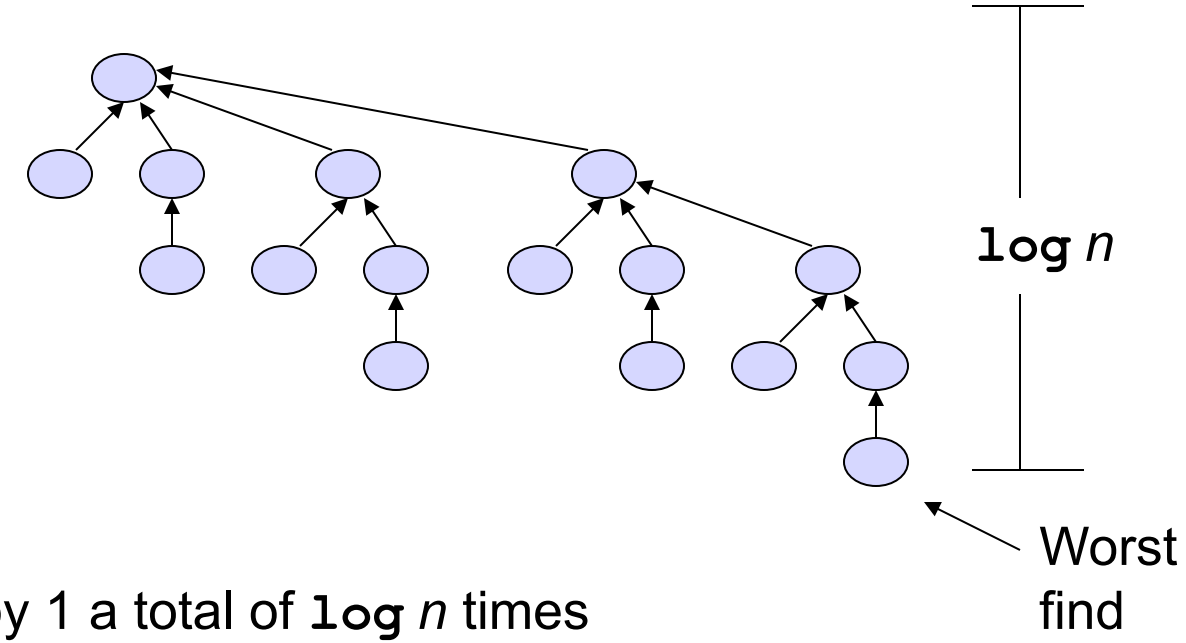


n/8 Unions-by-size



The new worst case (continued)

After n/2 + n/4 + …+ 1 Unions-by-size:



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

What about union-by-height

We could store the height of each root rather than size

- Still guarantees logarithmic worst-case find
  - Proof left as an exercise if interested

- But does not work well with our next optimization

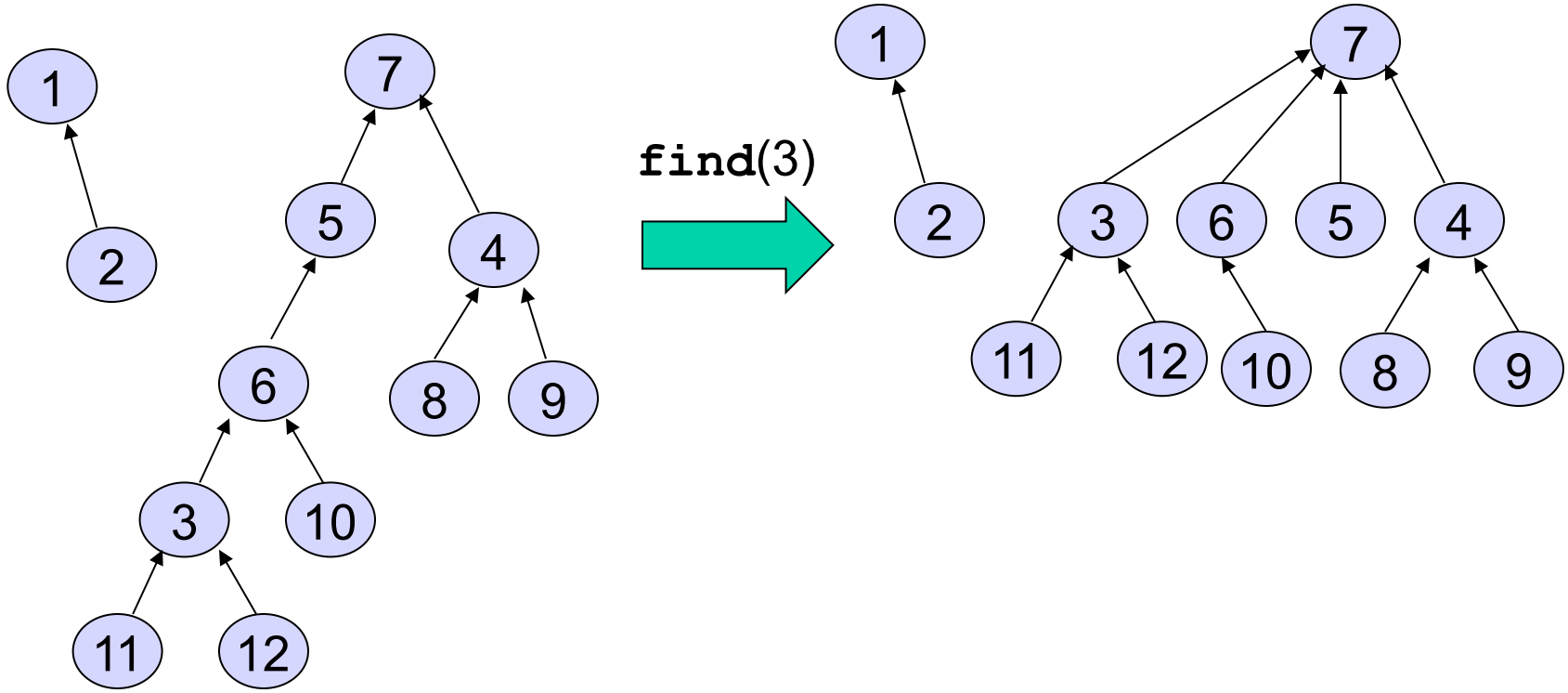
Two key optimizations

1. Improve **`union`** so it stays *O(1)* but makes **`find`** *O*(**`log`** *n*)
   - So *m* **`find`**s and *n*-1 **`union`**s 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* **`find`**s and *n*-1 **`union`**s ***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 **`find`**s for everything on the path (and their descendants)



Pseudocode

```
// performs path compression
int find(i) {
  // find root
  int r = i
  while(up[r] > 0)
     r = up[r]

  // compress path
  if i==r
    return r;
  int old_parent = up[i]
  while(old_parent != r) {
    up[i] = r
    i = old_parent;
    old_parent = up[i]
  }
  return r;
}
```

Example

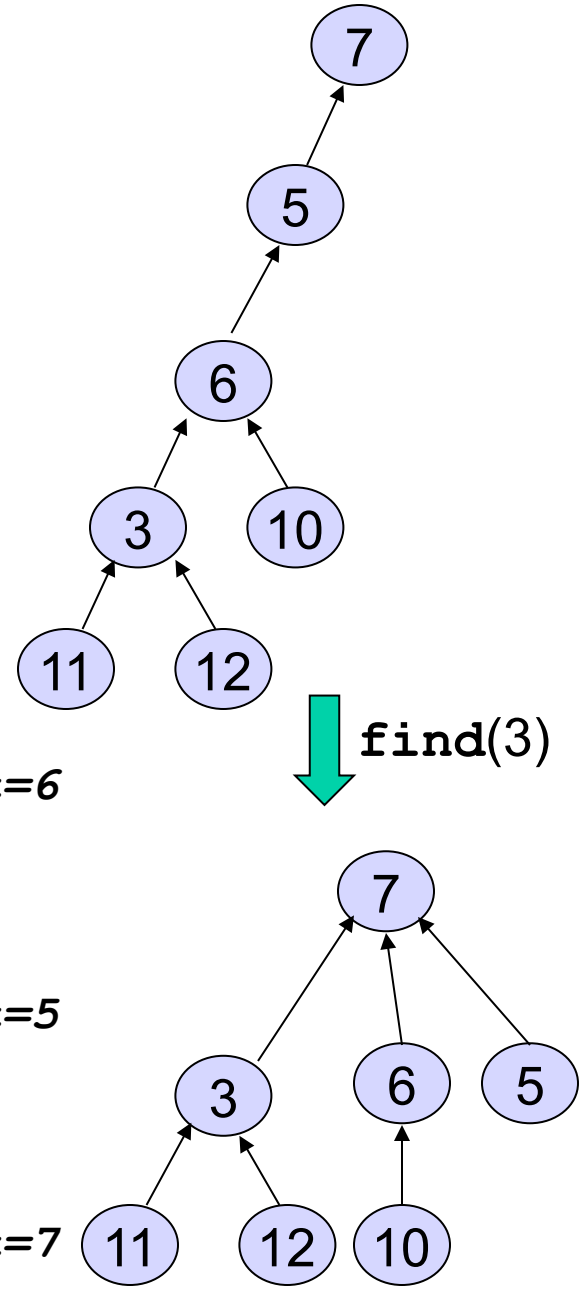
i=3
r=3

r=6
r=5
r=7

old_parent=6

up[3]=7
i=6
old_parent=5

up[6]=7
i=5
old_parent=7



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
- But we will *understand* it:
  - How it is *almost* $O(1)$
  - Because total for m **`find`**s and n-1 **`union`**s is *almost* $O(m+n)$

A really slow-growing function

`log*` *x* is the minimum number of times you need to apply "**`log`** of **`log`** of **`log`** of" to go from *x* to a number <= 1

For just about every number we care about, **`log*`** *x* is less than or equal to *5* (!)

If $x <= 2^{65536}$ then **`log*`** $x <= 5$

- $\log^* 2 = 1$
- $\log^* 4 = \log^* 2^2 = 2$
- $\log^* 16 = \log^* 2^{(2^2)} = 3$ (log log log 16 = 1)
- $\log^* 65536 = \log^* 2^{((2^2)^2)} = 4$ (log log log log 65536 = 1)
- $\log^* 2^{65536} = \ldots\ldots\ldots\ldots = 5$

Almost linear

- Turns out total time for m **`find`**s and n-1 **`union`**s is

  $O((m+n)*(\textbf{log*}\ (m+n))$

  – Remember, if $m+n < 2^{65536}$ then **`log*`** $(m+n) < 5$

  so effectively we have $O(m+n)$
- Because **`log*`** grows soooo slowly
  – For all practical purposes, amortized bound is constant, i.e., cost of find is constant, total cost for m finds is linear
  – We say “near linear” or “effectively linear”
- Need union-by-size and path-compression for this bound
  – Path-compression changes height but not weight, so they interact well
- As always, asymptotic analysis is separate from “coding it up”