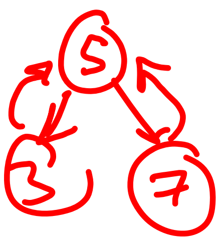
CSE 332: Disjoint Set Union/Find (part 2)

Richard Anderson, Steve Seitz

Winter 2014

Data structure for disjoint sets?

- Represent: {3,5,7} , {4,2,8}, {9}, {1,6}
- Support: find(x), union(x,y)



bunch of balanced trees: find O(log n) union O(n log n)

hash values → trees find O(1) union O(n)

hashtable value → set id

find O(1)

+ array maps id → list

union O(n)

Union/Find Trade-off

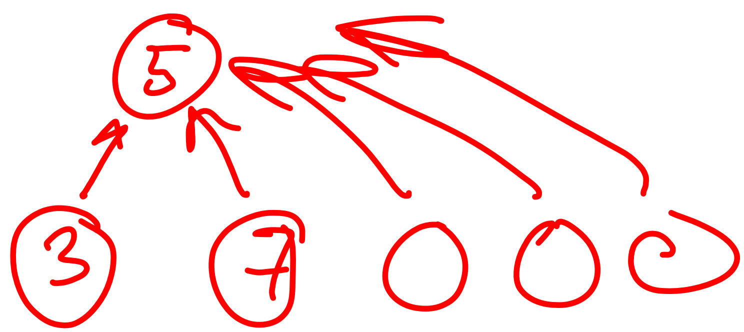
- Known result:
  - Find and Union cannot *both* be done in worst-case $O(1)$ time with any data structure.
- We will instead aim for good *amortized* complexity.
- For m operations on n elements:
  - Target complexity: $O(m)$ *i.e.* $O(1)$ amortized

Tree-based Approach

Each set is a tree

- Root of each tree is the set name.

$\{3, 5, 7\}$



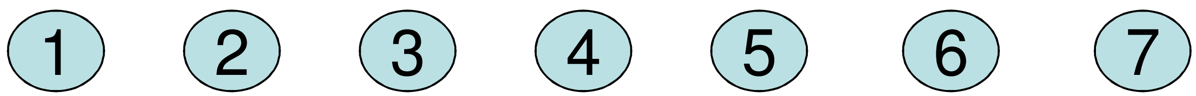
- Allow large fanout (why?)

Up-Tree for DS Union/Find

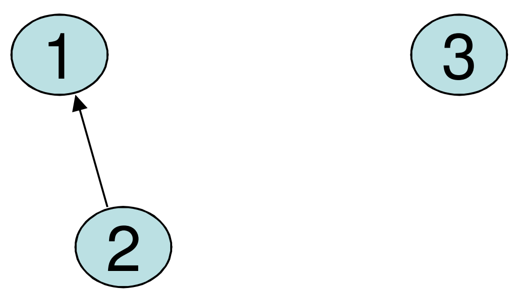
Observation: we will only traverse these trees upward from any given node to find the root.

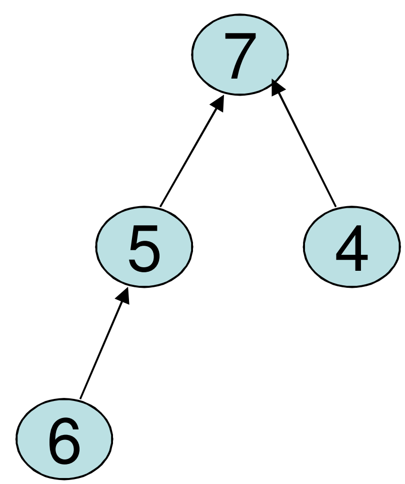
Idea: *reverse* the pointers (make them point up from child to parent). The result is an **up-tree**.

Initial state



Intermediate state





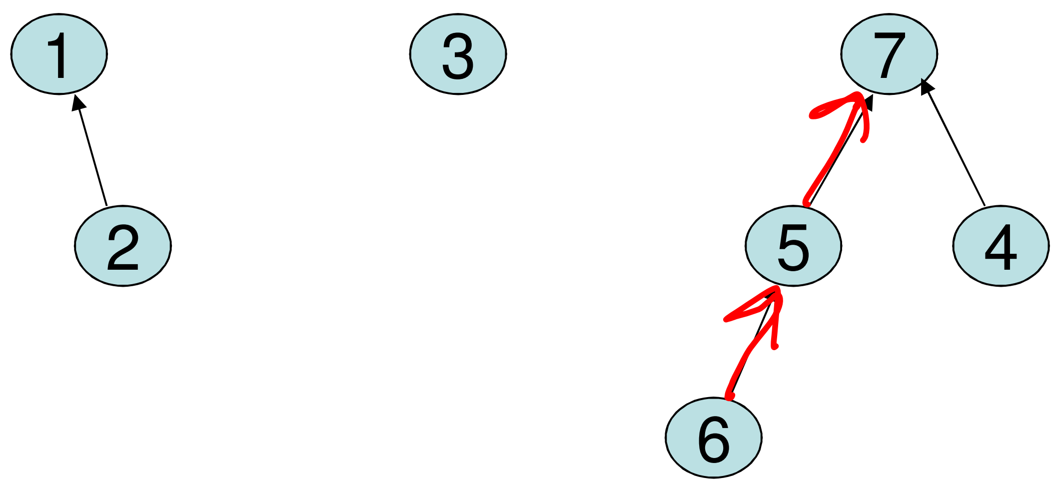
Roots are the names of each set.

Find Operation

Find(x) follow x to the root and return the root.

"Forest" of trees

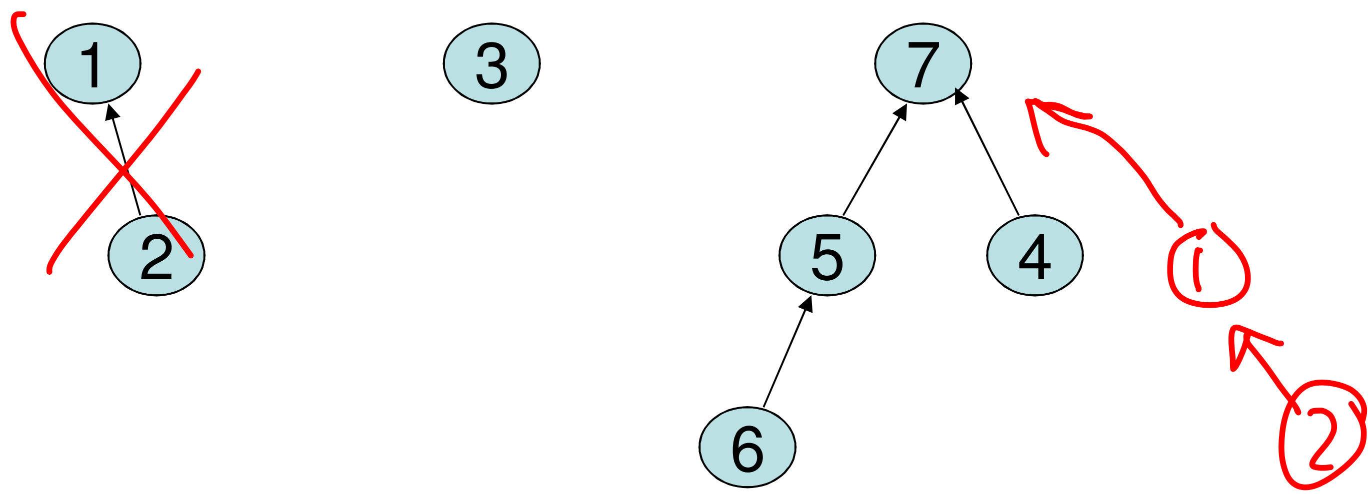
find(6) = 7



Union Operation

Union(i, j) - assuming i and j roots, point i to j.

union(1, 7)

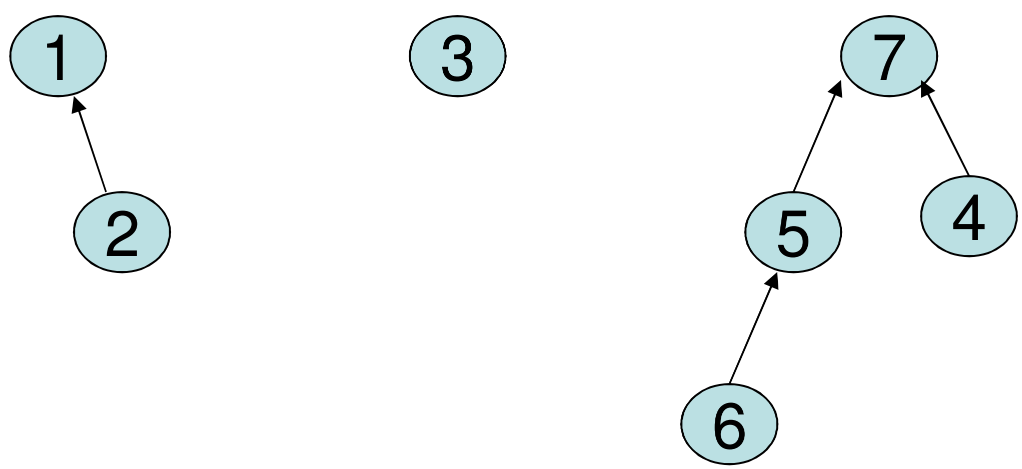


Simple Implementation

- Array of indices

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

up[x] = -1 means x is a root.



Implementation

```
void Union(int x, int y) {
  assert(up[x]<0 && up[y]<0);
  up[x] = y;
}
```

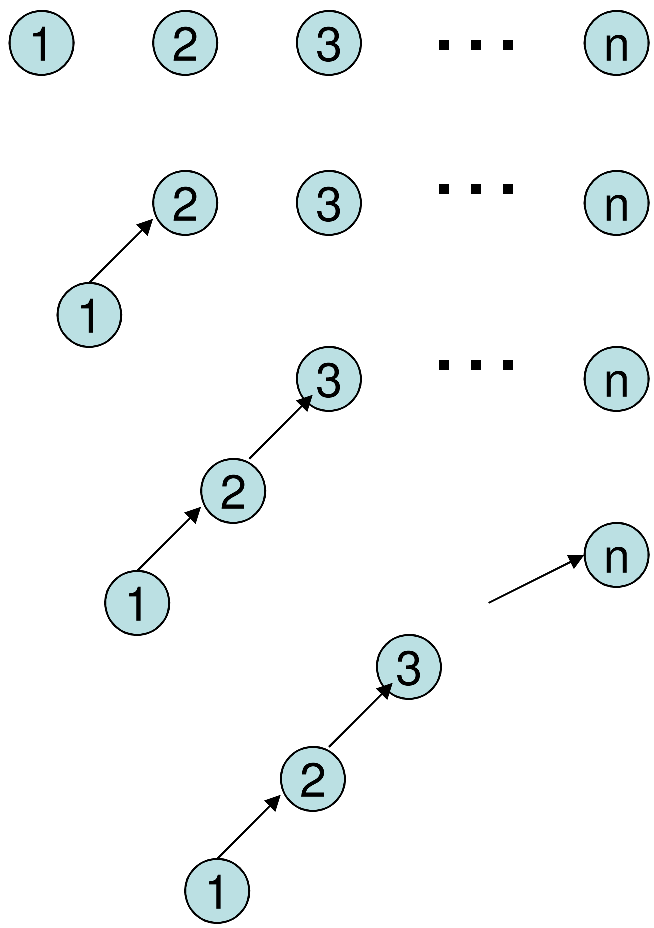
```
int Find(int x) {
  while(up[x] >= 0) {
    x = up[x];
  }
  return x;
}
```

runtime for Union: O(1)

runtime for Find: O(n)

Amortized complexity is no better.

A Bad Case



Union(1,2)

Union(2,3)

:

:

Union(n-1,n)

Find(1) n steps!!

after find: make found node point to root

union → point to top

make shorter tree point to taller root

Two Big Improvements

Can we do better? *Yes!*

1. Union-by-size
   - Improve Union so that *Find* only takes worst case time of $\Theta(\log n)$.

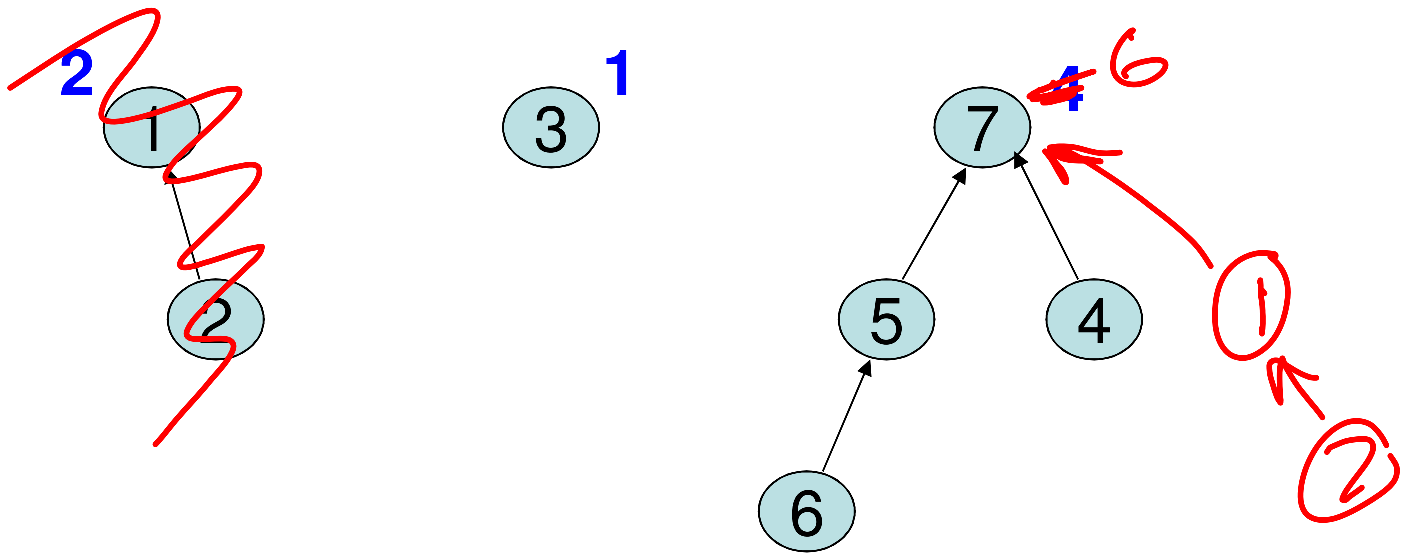
2. Path compression
   - Improve Find so that, with Union-by-size, Find takes amortized time of <u>almost</u> $\Theta(1)$.

Union-by-Size

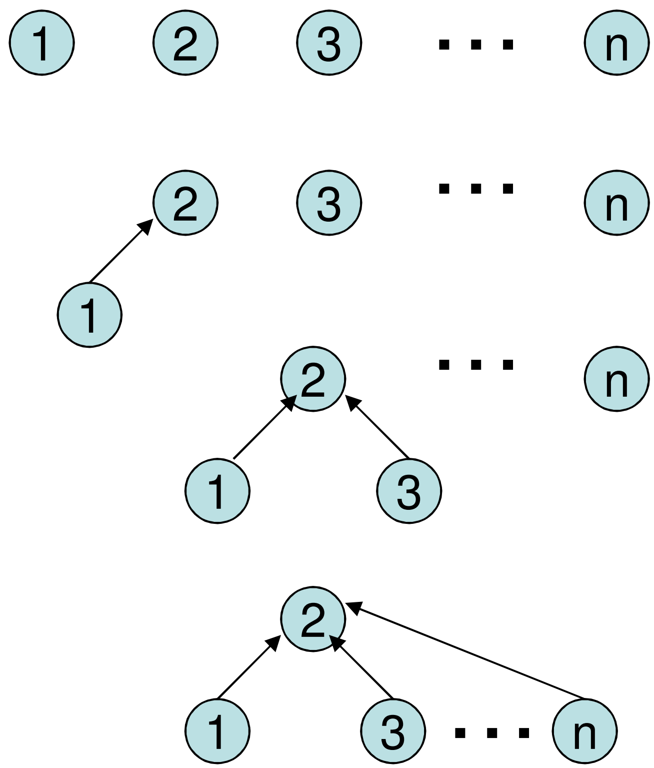
Union-by-size

- Always point the smaller tree to the root of the larger tree

S-Union(7,1)



Example Again



S-Union(1,2)

S-Union(2,3)

:

:

S-Union(n-1,n)

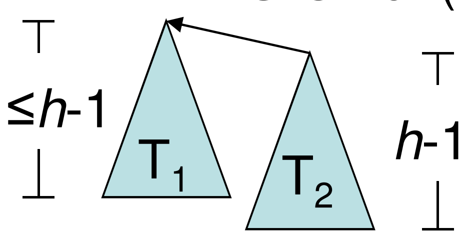
Find(1)   constant time

Analysis of Union-by-Size

- Theorem: With union-by-size an up-tree of height h has size at least 2^h.
- Proof by induction
  - Base case: $h = 0$. The up-tree has one node, $2^0 = 1$
  - Inductive hypothesis: Assume true for h-1
  - Observation: tree gets taller only as a result of a union.

$$S(T) = S(T_1) + S(T_2)$$
$$\geq 2^{h-1} + 2^{h-1}$$
$$= 2^h$$

T = S-Union(T_1,T_2)



Analysis of Union-by-Size

- What is worst case complexity of Find(x) in an up-tree forest of *n* nodes?

worst: all nodes in one tall tree of height h

Thm: $n \geq 2^h$

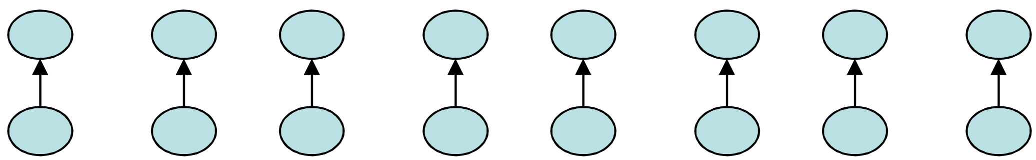
$\Rightarrow \log_2 n \geq h$

Find $O(\log n)$

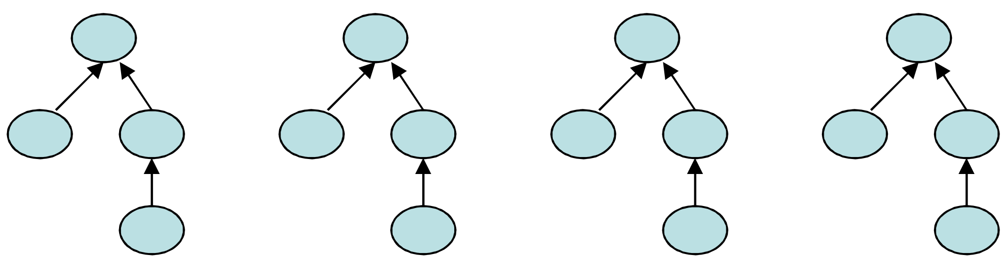
- (Amortized complexity is no better.)

Worst Case for Union-by-Size

n/2 Unions-by-size

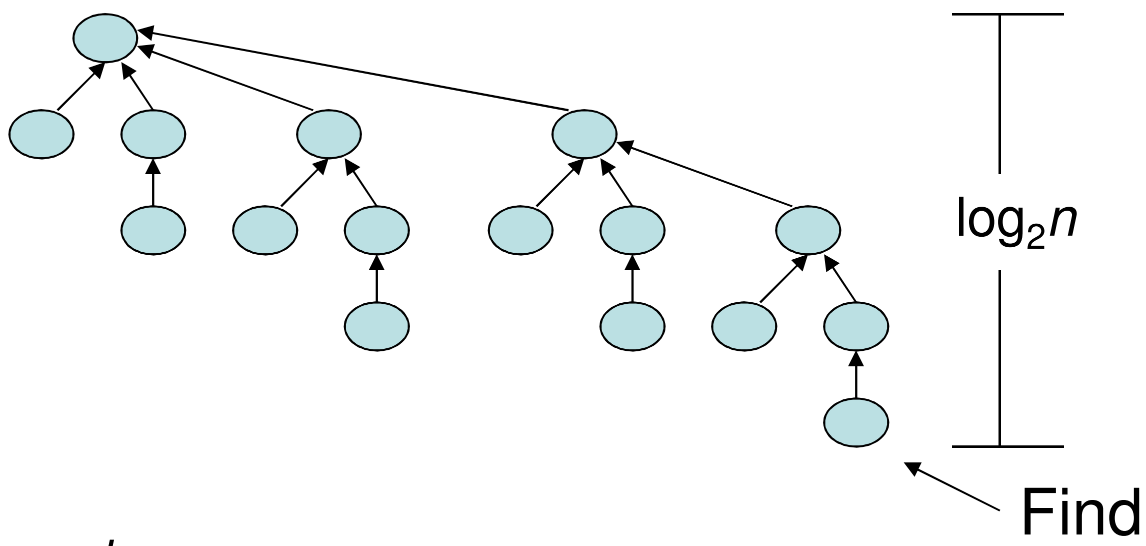


n/4 Unions-by-size



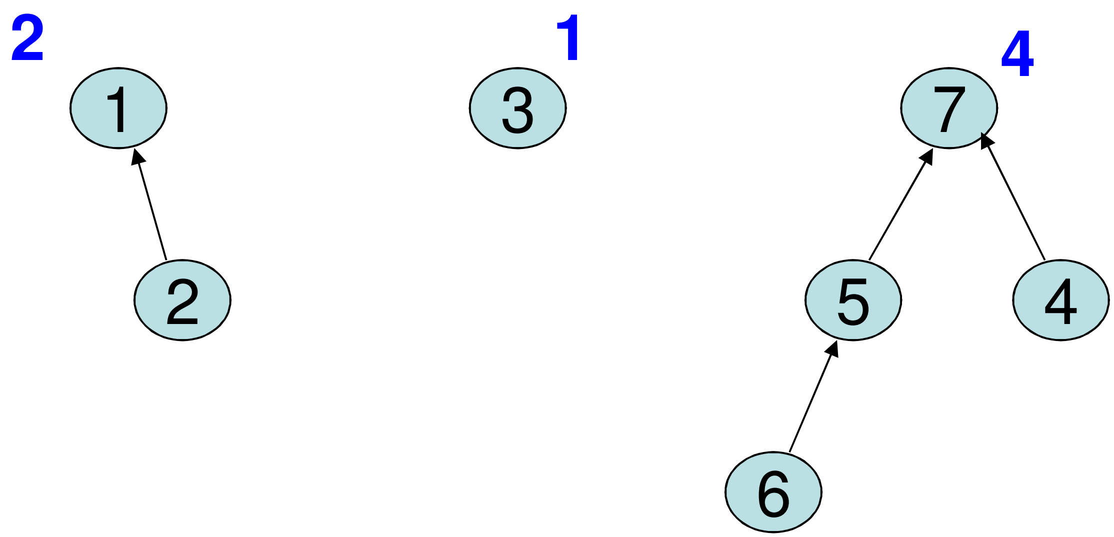
Example of Worst Cast (cont’)

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



If there are $n = 2^k$ nodes then the longest path from leaf to root has length k.

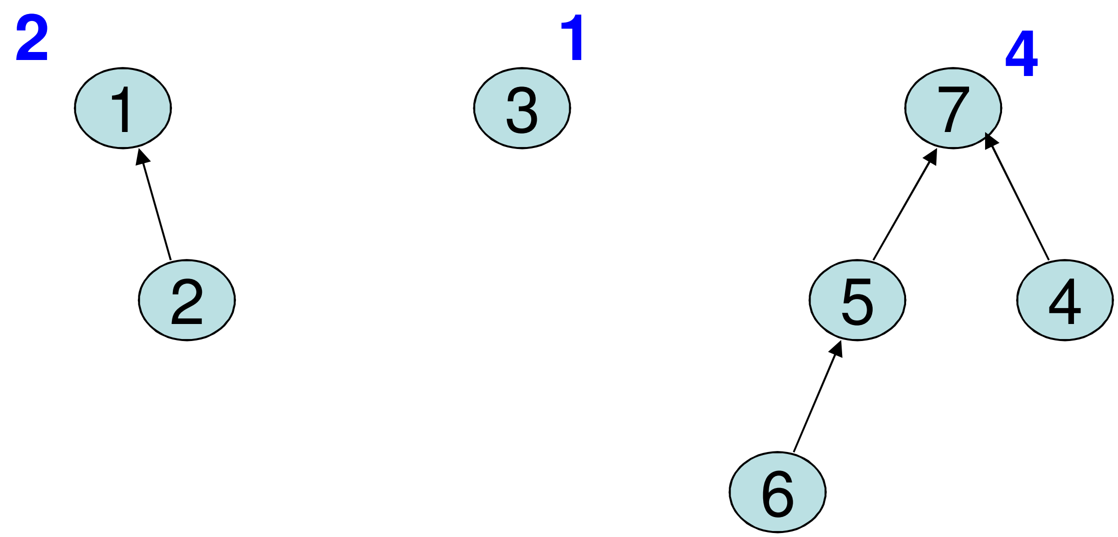
Array Implementation



Can store separate size array:

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

Elegant Array Implementation



Better, store sizes in the up array:

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

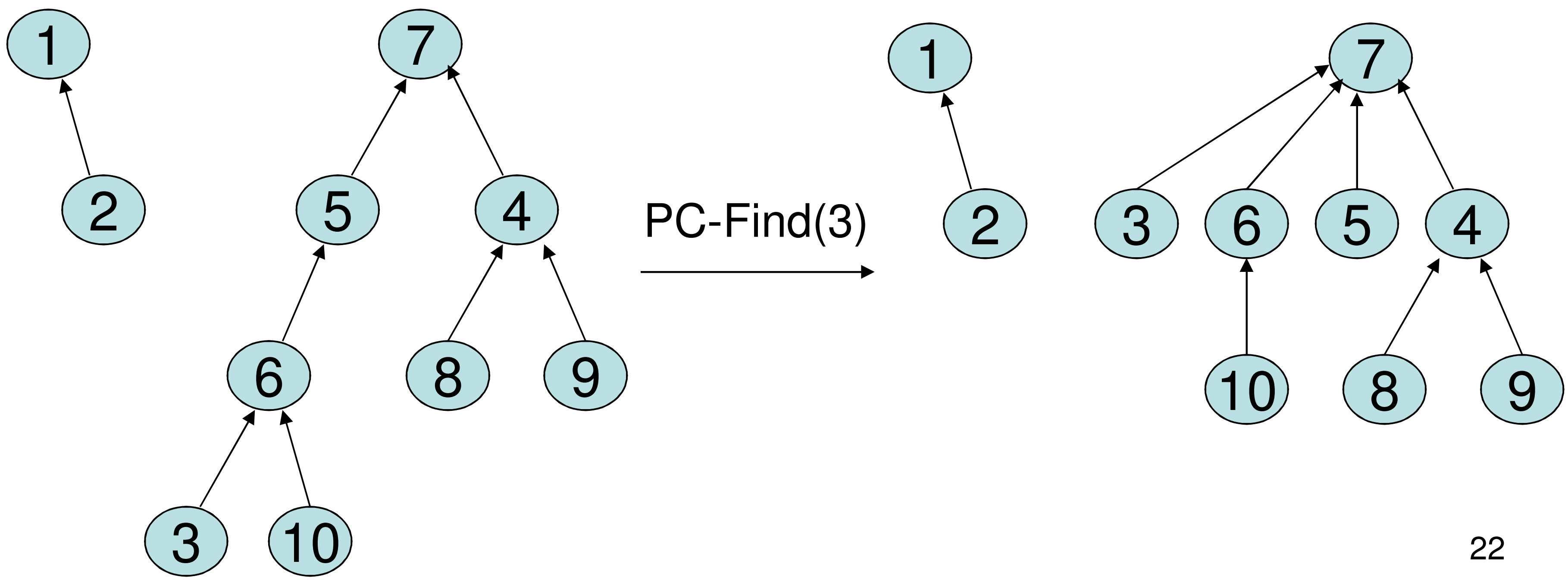
Negative up-values correspond to sizes of roots.

Code for Union-by-Size

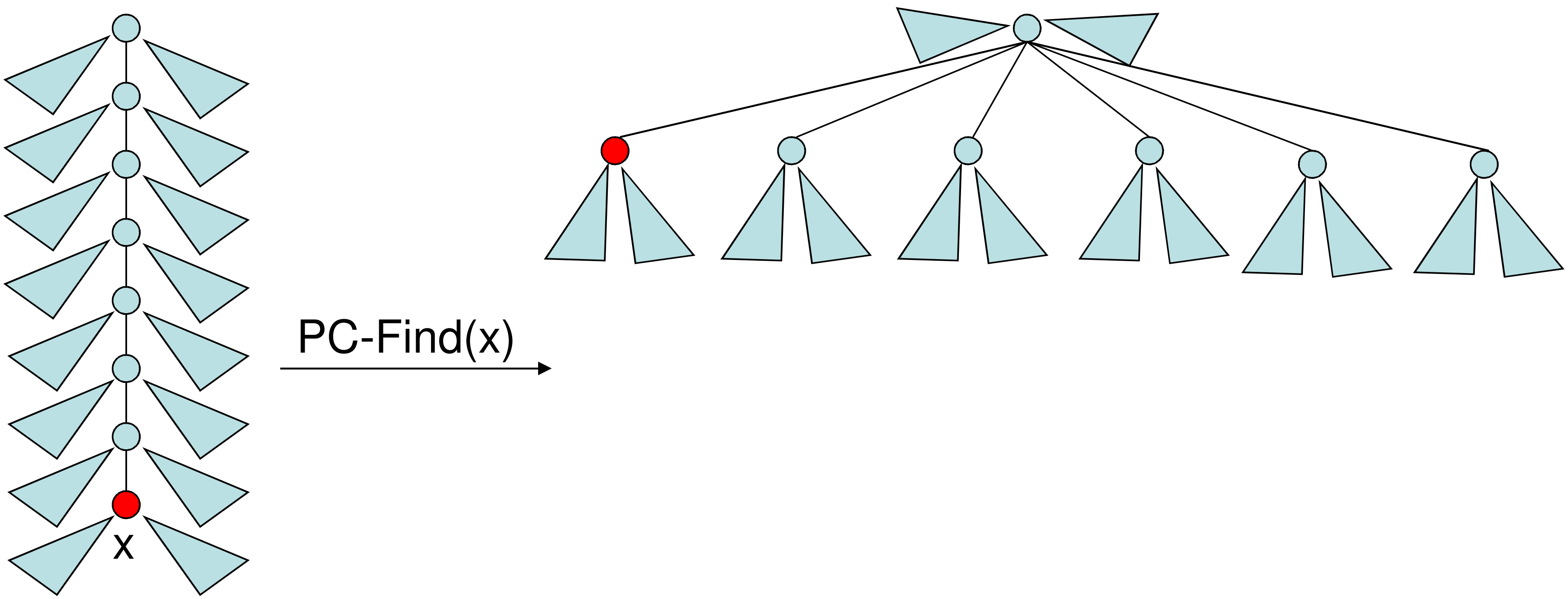
```
S-Union(i,j){
  // Collect sizes
  si = -up[i];
  sj = -up[j];

  // verify i and j are roots
  assert(si >=0 && sj >=0)
  // point smaller sized tree to
  // root of larger, update size
  if (si < sj) {
    up[i] = j;
    up[j] = -(si + sj);
  else {
    up[j] = i;
    up[i] = -(si + sj);
  }
}
```

Path Compression

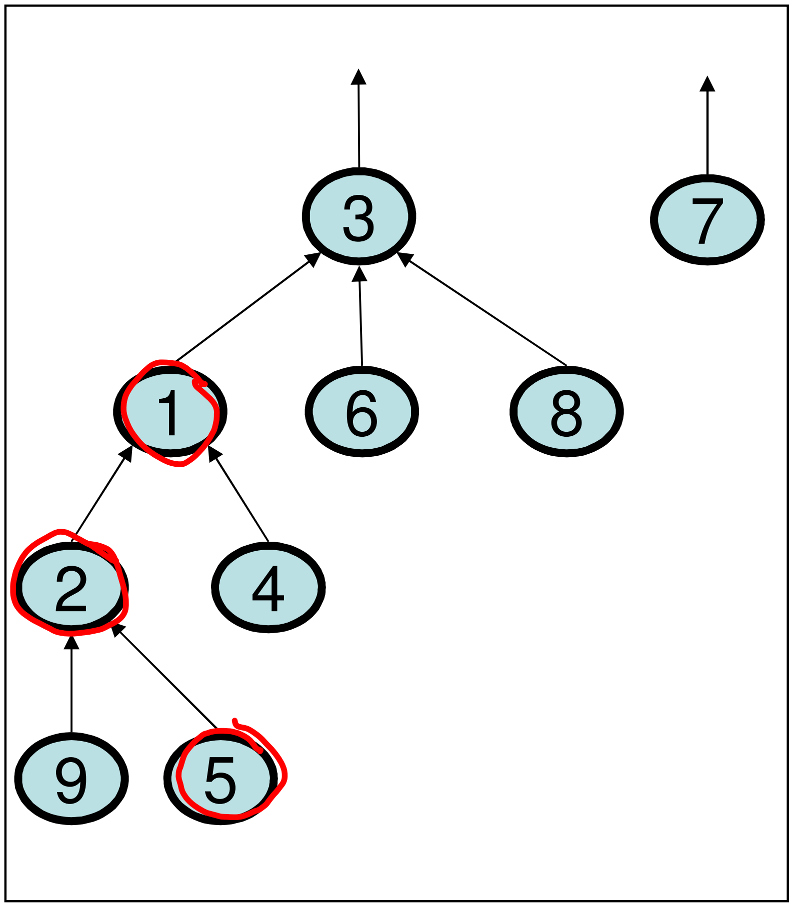
- To improve (amortized) complexity:
  - when going up the tree, *improve nodes on the path*!
- On a Find operation point all the nodes on the search path directly to the root. This is called "path compression."

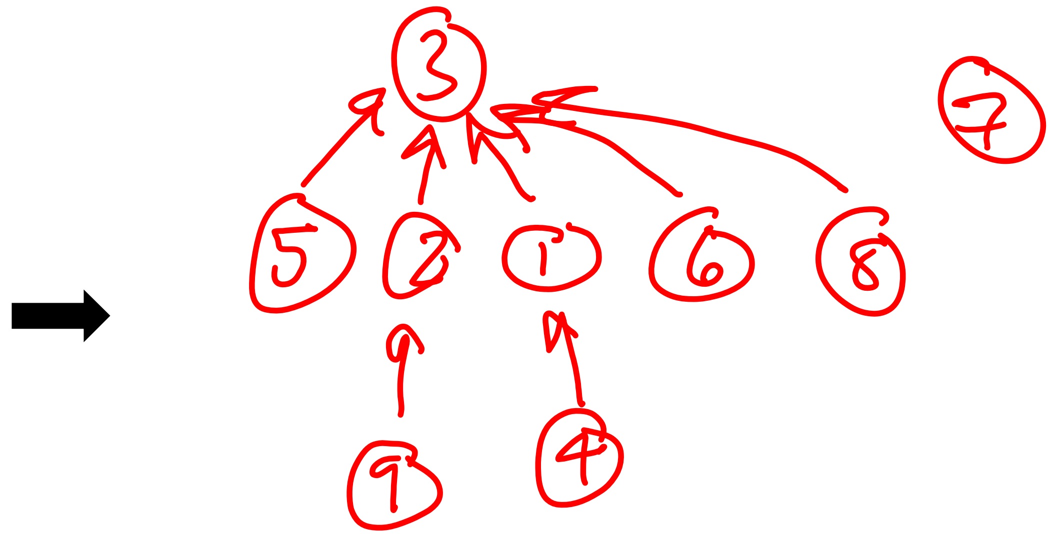


Self-Adjustment Works



Draw the result of Find(5):





Code for Path Compression Find

```
PC-Find(i) {
  //find root
  j = i;
  while (up[j] >= 0) {
    j = up[j];
  root = j;

  //compress path
  if (i != root) {
    parent = up[i];
    while (parent != root) {
      up[i] = root;
      i = parent;
      parent = up[parent];
    }
  }
  return(root)
}
```

Complexity of Union-by-Size + Path Compression

- Worst case time complexity for…
  - …a single Union-by-size is: $O(1)$
  - …a single PC-Find is: $O(\log n)$
- Time complexity for $m \geq n$ operations on n elements has been shown to be $O(m \log^* n)$.

  [See Weiss for proof.]
  - Amortized complexity is then $O(\log^* n)$
  - What is $\log^*$?

log* *n*

log* *n* = number of times you need to apply log to bring value down to at most 1

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} = …………… ≈ log* (2 x $10^{19,728}$) = 5

log * *n* ≤ 5 for all reasonable *n*.

The Tight Bound

In fact, Tarjan showed the time complexity for $m \geq n$ operations on n elements is:

$$\Theta(m\ \alpha(m, n))$$

Amortized complexity is then $\Theta(\alpha(m, n))$.

What is $\alpha(m, n)$?

– Inverse of Ackermann's function.

– For reasonable values of *m, n*, grows even slower than log * *n.* So, it's even "more constant."

Proof is beyond scope of this class. A simple algorithm can lead to incredibly hardcore analysis!