CSE 326: Data Structures Disjoint Set Union/Find (part 2)

Brian Curless

Spring 2008

Announcements (5/21/08)

- Homework due beginning of class on Friday.
- Reading for this lecture: Chapter 8.

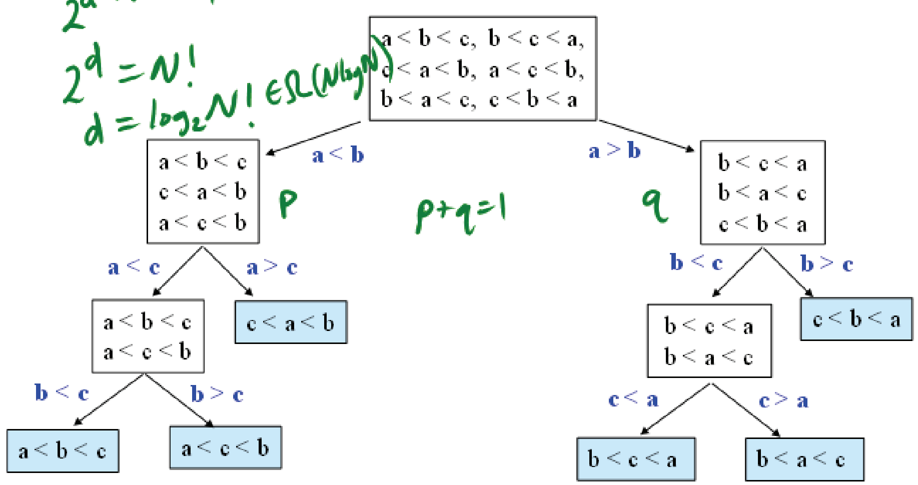
Thurs. office "hour"
10-10:30
11:30 - 12:00

Decision Tree

$\frac{1}{2^d} N! = 1$

$2^d = N!$

$d = \log_2 N! \in \Omega(N \log N)$



The leaves contain all the possible orderings of a, b, c.

Alternate Explanation of the Comparison-based Sorting Bound

At each decision point, one child has ≤ ½ of the options remaining, the other has ≥ ½ remaining.

Worst case: we always end up with ≥ ½ remaining.

Best outcome, in the worst case: we always end up with *exactly* ½ remaining.

Thus, in the worst case, the best we can hope for is halving the space d times (with d comparisons), until we have an answer, i.e., until the space is reduced to size = 1.

The space starts at $N!$ in size, and halving d times means multiplying by $1/2^d$, giving us a lower bound on the worst case:

$$\frac{N!}{2^d} = 1 \quad \Rightarrow \quad N! = 2^d \quad \Rightarrow \quad d = \log_2(N!) \in \Omega(N \log N)$$

Implementation: Take 1

Approach:
- Each set is a doubly-linked list (with pointer to last element).
- Store set name with object.

Find: get set name of object
- Worst case complexity? $O(1)$

Union: put one list on the end of the other, update set names of objects to be all the same
- Worst case complexity? $O(n)$

Implementation: Take 2

Approach:
- Each set is a doubly-linked list (with pointer to last element).
- Front of list is set name identifier.

Find: traverse linked list until reaching the front
- Worst case complexity? $O(n)$

Union: put one list on the end of the other
- Worst case complexity? $O(1)$

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

We'll build on the "fast union" approach (linked list, with head node as set name, no set names explicitly stored in nodes).

Improvements:
- Instead of linked lists, use a forest of trees (one tree per set).
- Root of each tree is the set name.
- Allow large fanout. Why is this good?
  Limits height ⇒ find is faster

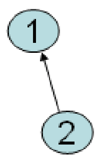
Up-Tree for DS Union/Find

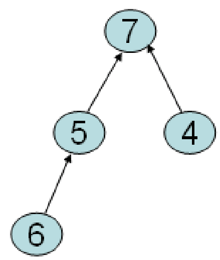
Observation: we will only traverse these trees upward from any given node to find the root.

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

Initial state

Intermediate state

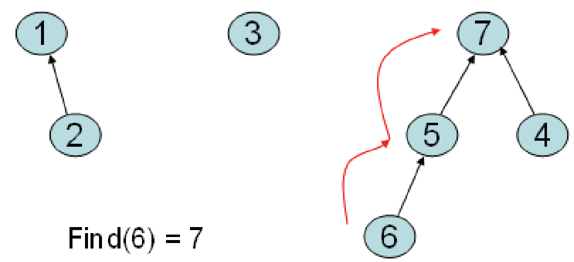




Roots are the names of each set.

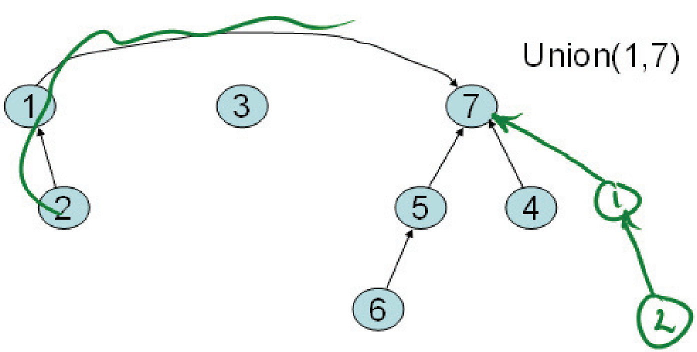
Find Operation

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



Union Operation

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

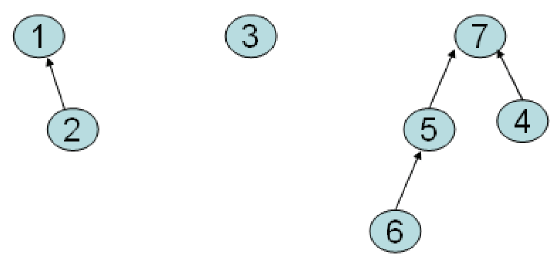


Simple Implementation

- 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

assert up[x]<0 & up[y]<0

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

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

runtime for Union:

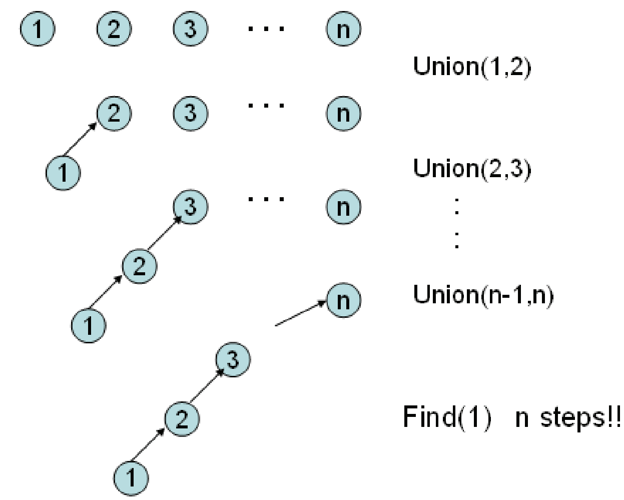
$O(1)$

runtime for Find:

$O(n)$

Amortized complexity is no better.

A Bad Case



Two Big Improvements

Can we do better? *Yes!*

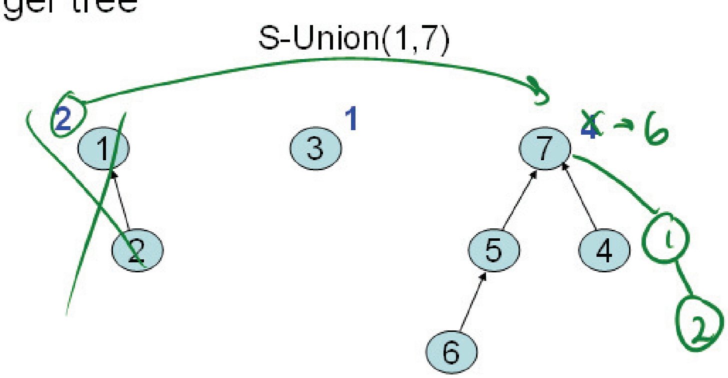
1. Union-by-size
   - Improve Union so that *Find* only takes worst case time of $\Theta(\log n)$.

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

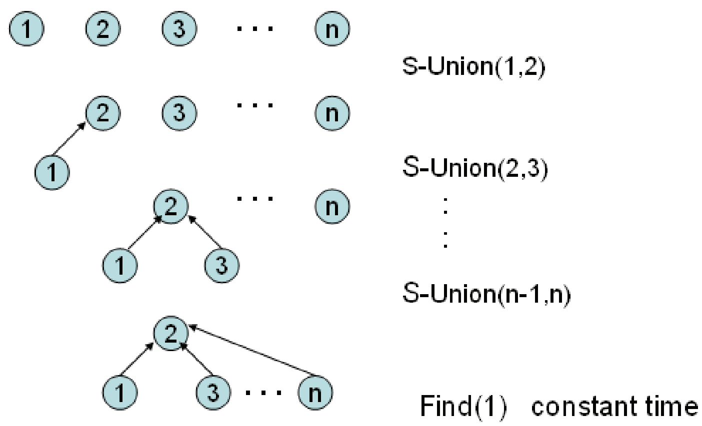
Union-by-Size

Union-by-size

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

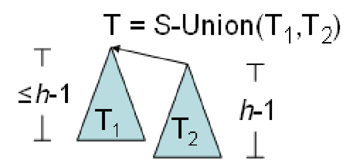


Example Again



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
  - Inductive step: Then true for h.
  - Observation: tree gets taller only as a result of a union.



$S(T_1) \geq S(T_2)$ $\quad S(T_2) \geq 2^{h-1}$

$S(T_1) \geq 2^{h-1}$

$S(T) = S(T_1) + S(T_2)$

$\geq 2^{h-1} + 2^{h-1} = 2^h$

$\Rightarrow S(T) \geq 2^h$

Analysis of Union-by-Size

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

All nodes in one set of height h.

$n \geq 2^h$

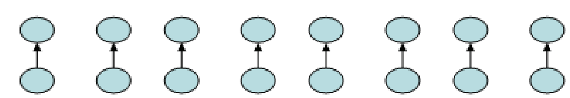
$\log_2 n \geq h$

$h \leq \log_2 n \in \Theta(\log_2 n)$

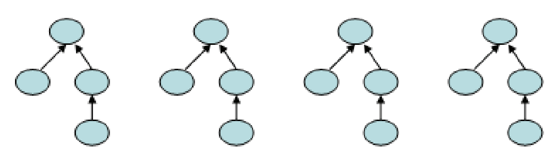
- (Amortized complexity is no better.)

Worst Case for Union-by-Size

n/2 Unions-by-size

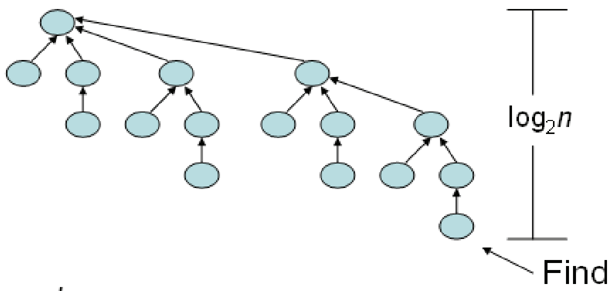


n/4 Unions-by-size



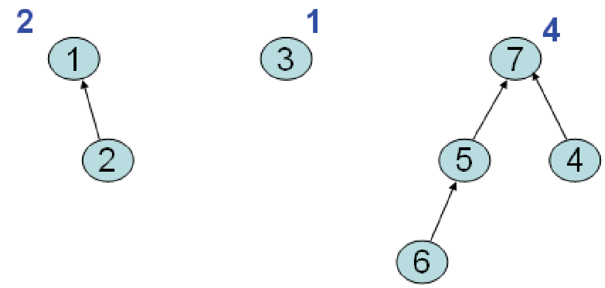
Example of Worst Cast (cont')

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.

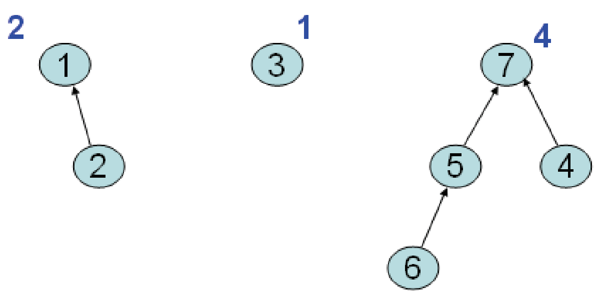
Array Implementation



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

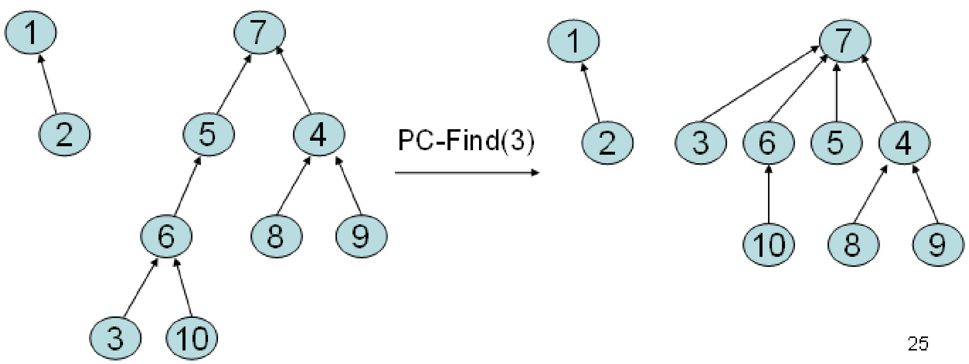
Negative up-values correspond to sizes of roots.

Code for Union-by-Size

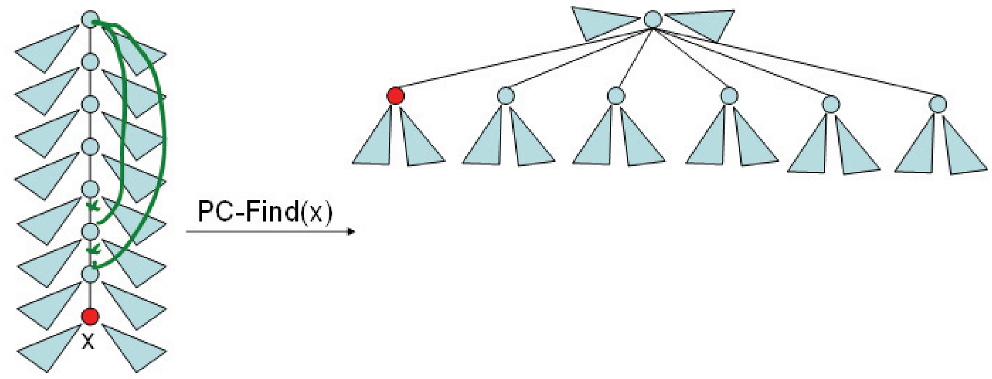
```
S-Union(i,j){
  // Collect sizes
  si = -up[i];
  sj = -up[j];

  // 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 the amortized complexity, we'll borrow an idea from splay trees:
  - 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."

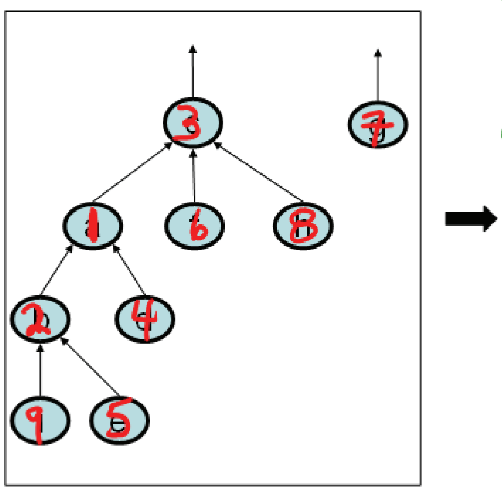


Self-Adjustment Works



Your turn

Draw the result of Find(~~e~~ 5):

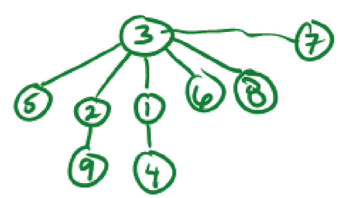


write up array before

Draw tree for after

write up array for after

1	2	3	4	5	6	7	8	9
3	1	-8	1	2	3	-1	3	2



1	2	3	4	5	6	7	8	9
3	3	~~-8~~ -9	1	3	3	3	3	2

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)

$\log^* 2^{65536} = \ldots\ldots\ldots\ldots \approx \log^* (2 \times 10^{19,728}) = 5$

$\log^* n \leq 5$ 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!