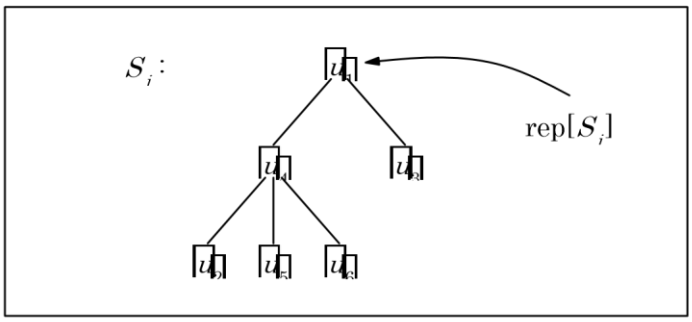
Forest-of-Trees Implementation

In addition to the linked-list implementation, we also saw in Lecture 4 an implementation of the disjoint-set data structure based on trees:



MAKE-SET(u) –	initialize new tree with root node u	$\Theta(1)$
FIND-SET(u)	–walk up tree from u to root	$\Theta(\text{height}) = \Theta(\lg n)$ best-case
UNION(u, v)	–change rep[S_v]'s parent to rep[S_u]	$O(1) + 2T_{\text{FIND-SET}}$

The efficiency of the basic implementation hinges completely on the height of the tree: the shorter the tree, the more efficient the operations. As the implementation currently stands, the trees could

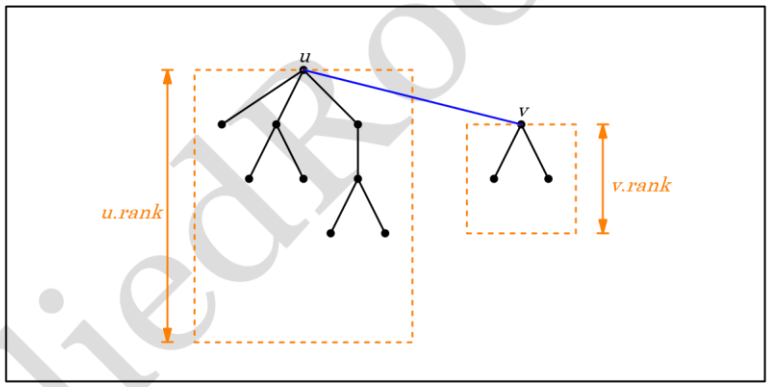


Figure 16.2. Union by rank attempts to always merge the shorter tree into the taller tree, using rank as an estimate (always an overestimate) of height.

be unbalanced and FIND-SET could take as long as $\Theta(n)$ in the worst case. However, this behavior can be dramatically improved, as we will see below.

16.2.1 Union by rank

When we call UNION(u,v), rep[S_v] becomes a child of rep[S_u]. Merging S_v into S_u in this way results in a tree of height

$$\max\{\text{height}[S_u], \text{height}[S_v]+1\} \qquad (16.1)$$

(why?). Thus, the way to keep our trees short is to always merge the shorter tree into the taller tree. (This is analogous to the "smaller into larger" strategy in the linked-list implementation.) To help us do this, we will introduce a new data field called rank. If u is the root of a tree (i.e., if u =

rep[S_u]), then u.rank will be an upper bound on the height of S_u.[1] In light of (16.1), the pseudocode for UNION will be as follows (see Figure 16.2):

```
Algorithm: UNION(u, v)
                          e e
1       u ← FIND-SET(u) e
      ←  2    vFIND-SET(v) e
         3    if u.rank= v.rank then
4       u.rank← u.rank+1
5       v.parent← u
6       else if u.rank> v.rank then
7       v.parent← u
8       else
9       u.parent        v
              ←
```

The following lemma shows that UNION preserves the fact that rank is an upper bound on height.

Lemma 16.1. *Suppose initially the following hold:*

- S_u 6= S_v
- S_u *has height* h_1 *and* S_v *has height* h_2 • rep[S_u].*rank*= r_1 *and* rep[S_v].*rank*= r_2
- $h_1 \le r_1$ *and* $h_2 \le r_2$.

Suppose we then call UNION(u, v)*, producing a new set* $S = S_u \cup S_v$*. Let* h *be the height of* S *and let* r = rep[S].*rank. Then* $h \le r$.

Proof. First, suppose $r_1 > r_2$. Then S_v has been merged into S_u and ^{r} = r_1. By (16.1), we have

$$h = \max^{\copyright} h_1,\ h_2 + 1^{a}$$
$$\le \max^{\copyright} r_1,\ r_2 + 1^{a}$$
$$=$$
$$r_1$$
$$=$$

[1] If union by rank is the only improvement we use, then u.rank will actually be the exact height of Su. But in general, we wish to allow other improvements (such as path compression) to decrease the height of Su without having to worry about updating ranks. In such cases, the upper bound provided by u.rank may not be tight.

r.

A similar argument shows that $h \leq r$ in the case that $r_2 > r_1$. Finally, suppose $r_1 = r_2$. Then S_v has been merged into S_u and $r = r_1 + 1 = r_2 + 1$, so

$$h = \max(h_1, h_2) + 1$$

$$\leq \max\ r_1, r_2 + 1$$

$$= r_2 + 1 = r. \quad \square$$

It turns out that the rank of a tree with k elements is always at most $\lg k$. Thus, the worst-case performance of a disjoint-set forest with union by rank having n elements is

MAKE-SET	$O(1)$
FIND-SET	$\Theta(\lg n)$
UNION	$\Theta(\lg n)$.

Exercise 16.2. *Amortization does not help this analysis. Given sufficiently large n and given m which is sufficiently large compared to n, produce a sequence of m operations, n of which are* MAKE-SET *operations (so the structure ultimately contains n elements), whose running time is* $\Theta(m \lg n)$.

Exercise 16.3. *Above we claimed that the rank of any tree with k elements is at most* $\lg k$. *Use induction to prove this claim. (You may assume that* UNION *is the only procedure that modifies ranks. However, you should not assume anything about the height of a tree except that it is less than the rank.) What is the base case?*

16.2.2 Path compression

The easiest kind of tree to walk up is a *flat* tree, where all non-root nodes are direct children of the root (see Figure 16.3). The idea of path compression is that, every time we invoke FIND-SET and walk up the tree, we should reassign parent pointers to make each node we pass a direct child of the root (see Figure 16.4). This locally flattens the tree. With path compression, the pseudocode for FIND-SET is as follows:

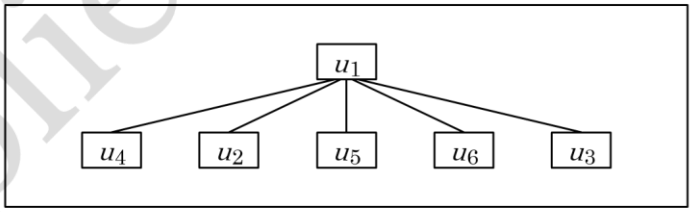


Figure 16.3. In a flat tree, each FIND-SET operation requires us to traverse only one edge.

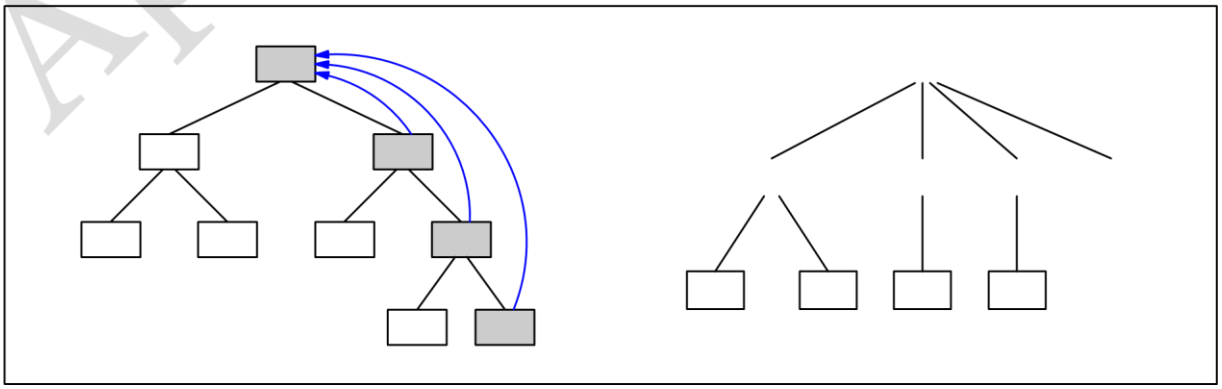


Figure 16.4. With path compression, calling FIND-SET(u8) will have the side-effect of making u8 and all of its ancestors direct children of the root.

```
Algorithm: FIND-SET(u)
1 A ←;
2       n← u
3       while n is not the root do
4       A ← A∪{n}
5       n← n.parent
6       for each x ∈ A do
7       x.parent← n
8       return n
```

What data structure should we use for A? In an ideal world, where n can truly be arbitrarily large, we would probably want A to be a dynamically doubled array of the kind discussed in Lecture 10. In real life, however, some assumptions can be made. For example, if you have less than a petabyte (1024 TB, or 2^{53} bits) of available memory, then the rank (and therefore the height) of any tree is at most $\lg^{\lfloor} 2^{53}^{\rfloor} = 53$, and it would be slightly more efficient to maintain A as a static array of size 53 (with an end-marking sentinel value, perhaps).

It can be shown that, with path compression (but not union by rank), the running time of any sequence of n MAKE-SET operations, f FIND-SET operations, and up to $n-1$ UNION operations is

$$\Theta(n + f(1+\log_{2+f/n} n)).$$

16.2.3 Both improvements together

The punch-line of this lecture is that, taken together, union by rank and path compression produce a spectacularly efficient implementation of the disjoint-set data structure.

Theorem 16.2. *On a disjoint-set forest with union by rank and path compression, any sequence of m operations, n of which are* MAKE-SET *operations, has worst-case running time*

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

where α is the inverse Ackermann function. Thus, the amortized worst-case running time of each operation is $\Theta(\alpha(n))$. If one makes the approximation $\alpha(n) = O(1)$, which is valid for literally all conceivable purposes, then the operations on a disjoint-set forest have $O(1)$ amortized running time.

The proof of this theorem is in §21.4 of CLRS. You can read it if you like; it is not essential. You might also be interested to know that, in a 1989 paper, Fredman and Saks proved that $\Theta(\alpha(n))$ is the fastest possible amortized running time per operation for any implementation of the disjointset data structure.

The inverse Ackermann function α is defined by

$$\alpha(n) = \min\{k : A_k(1) \geq n\},$$

where $(k, j) \mapsto A_k(j)$ is the Ackermann function. Because the Ackermann function is an extremely rapidly growing function, the inverse Ackermann function α is an extremely slowly growing function (though it is true that $\lim_{n\to\infty}\alpha(n) = \infty$).

The Ackermann function A (at least, one version of it) is defined by

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ (A_{k-1})^{(j+1)}(j) & \text{(that is, } A_{k-1} \text{ iterated } j+1 \text{ times)} \quad \text{for } k \geq 1. \end{cases}$$

Some sample values of the Ackermann function are

$A_1(1) = 3$ $\qquad$ $A_1(j) = 2j+1$

$A_2(1) = 7$ $\qquad$ $A_2(j) = 2^{j+1}(j+1)$

$A_3(1) = 2047$

$A_4(1) \approx 10^{80}$.

By current estimates, 10^{80} is roughly the same order of magnitude as the number of particles in the observable universe. Thus, even if you are a theoretical computer scientist or mathematician, you will still most likely never end up considering a number n so large that $\alpha(n) > 4$.