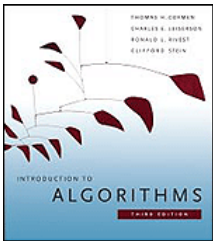

Chapter 21 - Data Structures for Disjoint Sets
Introduction to Algorithms, Third Edition
by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
The MIT Press © 2009 *Citation*

Previous Next

21.3 Disjoint-Set Forests

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest***, illustrated in Figure 21.4(a), each member points only to its parent. The root of each tree contains the representative and is its own parent. As we shall see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, by introducing two heuristics—"union by rank" and "path compression"—we can achieve an asymptotically optimal disjoint-set data structure.

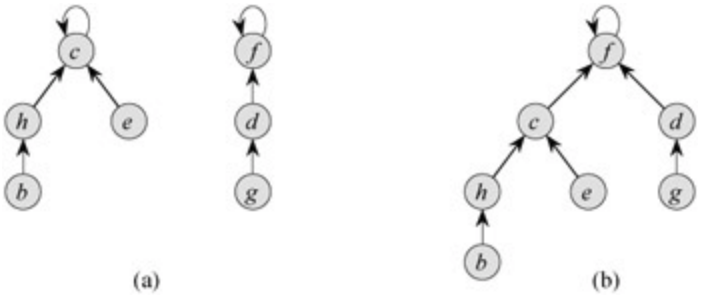

Figure 21.4: A disjoint-set forest. **(a)** Two trees representing the two sets of Figure 21.2. The tree on the left represents the set {*b*, *c*, *e*, *h*}, with c as the representative, and the tree on the right represents the set {*d*, *f*, *g*}, with *f* as the representative. **(b)** The result of UNION(*e*, *g*).

We perform the three disjoint-set operations as follows. A MAKE-SET operation simply creates a tree with just one node. We perform a FIND-SET operation by following parent pointers until we find the root of the tree. The nodes visited on this simple path toward the root constitute the ***find path***. A UNION operation, shown in Figure 21.4(b), causes the root of one tree to point to the root of the other.

Heuristics to Improve the Running Time

So far, we have not improved on the linked-list implementation. A sequence of n - 1 UNION operations may create a tree that is just a linear chain of n nodes. By using two heuristics, however, we can achieve a running time that is almost linear in the total number of operations m.

The first heuristic, ***union by rank***, is similar to the weighted-union heuristic we used with the linked-list representation. The obvious approach would be to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, we shall use an approach that eases the analysis. For each node, we maintain a ***rank***, which is an upper bound on the height of the node. In union by rank, we make the root with smaller rank point to the root with larger rank during a UNION operation.

The second heuristic, ***path compression***, is also quite simple and highly effective. As shown in Figure 21.5, we use it during FIND-SET operations to make each node on the find path point directly to the root. Path compression does not change any ranks.

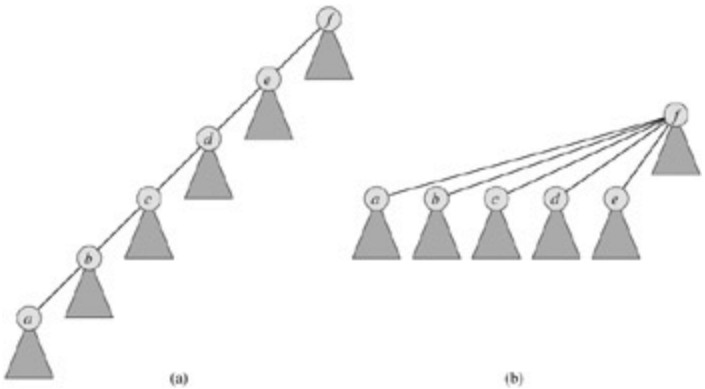

Figure 21.5: Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(*a*). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(*a*). Each node on the find path now points directly to the root.

Pseudocode for Disjoint-Set Forests

To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks. With each node *x*, we maintain the integer value *x.rank*, which is an upper bound on the height of *x* (the number of edges in the longest simple path from a descendant leaf to *x*). When MAKE-SET creates a singleton set, the single node in the corresponding tree has an initial rank of 0. Each FIND-SET operation leaves all ranks unchanged. The UNION operation has two cases, depending on whether the roots of the trees have equal rank. If the roots have unequal rank, we make the root with higher rank the parent of the root with lower rank, but the ranks themselves remain unchanged. If, instead, the roots have equal ranks, we arbitrarily choose one of the roots as the parent and increment its rank.

Let us put this method into pseudocode. We designate the parent of node *x* by *x.p*. The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs.

```
MAKE-SET(x)
1 x.p = x
2 x.rank = 0

UNION(x, y)
1 LINK(FIND-SET(x), FIND-SET(y)

LINK(x, y)
1 if x.rank > y.rank
2     y.p = x
3 else x.p = y
4     if x.rank == y.rank
5         y.rank = y.rank + 1
```

The FIND-SET procedure with path compression is quite simple:

```
FIND-SET(x)
1 if x ≠ x.p
2     x.p = FIND-SET(x.p)
3 return x.p
```

The FIND-SET procedure is a ***two-pass method***: as it recurses, it makes one pass up the find path to find the root, and as the recursion unwinds, it makes a second pass back down the find path to update each node to point directly to the root. Each call of FIND-SET(*x*) returns *x.p* in line 3. If *x* is the root, then FIND-SET skips line 2 and instead returns *x.p*, which is *x*, this is the case in which the recursion bottoms out. Otherwise, line 2 executes, and the recursive call with parameter *x.p* returns a pointer to the root. Line 2 updates node *x* to point directly to the root, and line 3 returns this pointer.