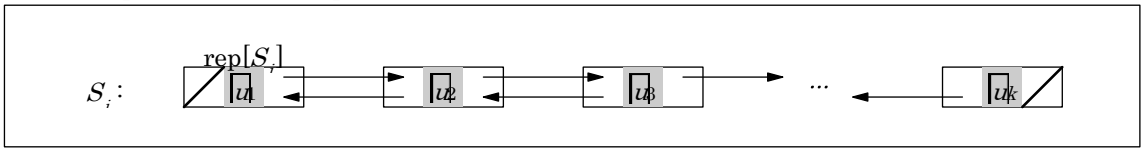
Linked-List Implementation

Recall the simple linked-list implementation of disjoint sets that we saw in Lecture 4:



MAKE-SET(u) – initialize as a lone node — $\Theta(1)$

FIND-SET(u) – walk left from u until you reach the head of S_u — $\Theta(n)$ worst-case

UNION(u, v) – walk right (towards the tail) from u and left (towards the head) from v. Reassign pointers so that the tail of S_u and the head of S_v become neighbors. The representative is updated automatically. — $\Theta(n)$ worst-case

Do we really need to do all that walking? We could save ourselves some time by augmenting each element u with a pointer to rep[S_u], and augmenting rep[S_u] itself with a pointer to the tail. That way, the running time of FIND-SET would be $O(1)$ and all the walking we formerly had to do in UNION would become unnecessary. However, to maintain the new data fields, UNION(u, v) would have to walk through S_v and update each element's "head" field to point to rep[S_u]. Thus, UNION would still take $O(n)$ time in the worst case.

Perhaps amortization can give us a tighter bound on the running time of UNION? At first glance, it doesn't seem to help much. As an example, start with the sets $\{1\},\{2\},\ldots,\{n\}$. Perform UNION(2,1), followed by UNION(3,1), and so on, until finally UNION(n,1), so that $S_1 = \{1,\ldots,n\}$. In each call to UNION, we have to walk to the tail of S_1, which is continually growing. The ith call to UNION has to walk through $i-1$ elements, so that the total running time of the $n-1$ UNION operations is $\sum_{i=1}^{n-1}(i-1) = \Theta(n^2)$. Thus, the amortized cost per call to UNION is $\Theta(n)$.

However, you may have noticed that we could have performed essentially the same operations more efficiently by instead calling UNION(1,2) followed by UNION(1,3), and so on, until finally UNION(1,n). This way, we never have to perform the costly operation of walking to the tail of S_1; we only ever have to walk to the tails of one-element sets. Thus the running time for this smarter sequence of $n-1$ UNION operations is $\Theta(n)$, and the amortized cost per operation is $O(1)$.

The lesson learned from this analysis is that, when performing a UNION operation, it is best to always merge the smaller set into the larger set, i.e., the representative element of the combined set should be chosen equal to the representative of the larger constituent—that way, the least possible amount of walking has to occur. To do this efficiently, we ought to augment each S_i with a "size" field, which we'll call $S_i.weight$ (see Figure 16.1).

It turns out that the "smaller into larger" strategy gives a significant improvement in the amortized worst-case running time of the UNION operation. We'll show that the total running time of any sequence of UNION operations on a disjoint-set data structure with n elements (i.e., in which MAKE-SET is called n times) is $O(n \lg n)$. Thus, the running time of m operations, n of which are MAKE-SET operations, is

$$O(m + n \lg n).$$

To start, focus on a single element u. We'll show that the total amount of time spent updating u's "head" pointer is $O(\lg n)$; thus, the total time spent on all UNION operations is $O(n \lg n)$. When u

is added to the structure via a call to MAKE-SET, we have $S_u.weight = 1$. Then, every time S_u merges with another set S_v, one of the following happens:

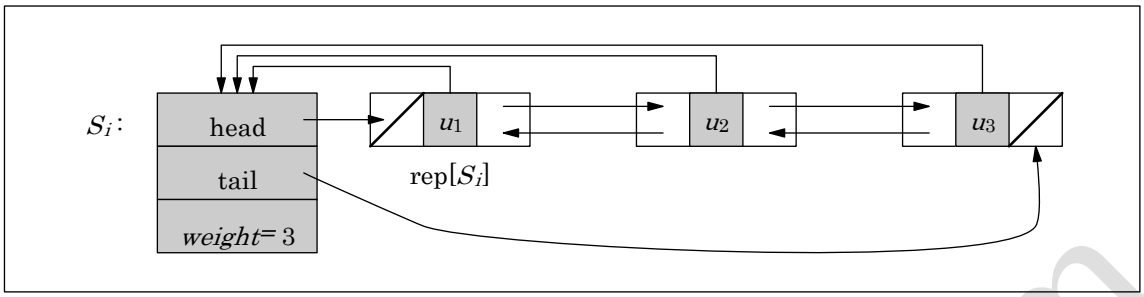


Figure 16.1. A linked list augmented with data fields for the head, the tail, and the size (weight).

- $S_u.weight > S_v.weight$. Then no update to u's "head" pointer is needed.
- $S_v.weight \geq S_u.weight$. Then, we update u's "head" pointer. Also, in this case, the value of $S_u.weight$ at least doubles.

Because $S_u.weight$ at least doubles every time we update u's "head" pointer, and because $S_u.weight$ can only be at most n, it follows that the total number of times we update u's "head" pointer is at most $\lg n$. Thus, as above, the total cost of all UNION operations is $O(n \lg n)$ and the total cost of any sequence of m operations is $O(m + n \lg n)$.