19 Data Structures for Disjoint Sets

Some applications involve grouping n distinct elements into a collection of disjoint sets—sets with no elements in common. These applications often need to perform two operations in particular: finding the unique set that contains a given element and uniting two sets. This chapter explores methods for maintaining a data structure that supports these operations.

Section 19.1 describes the operations supported by a disjoint-set data structure and presents a simple application. Section 19.2 looks at a simple linked-list implementation for disjoint sets. Section 19.3 presents a more efficient representation using rooted trees. The running time using the tree representation is theoretically superlinear, but for all practical purposes it is linear. Section 19.4 defines and discusses a very quickly growing function and its very slowly growing inverse, which appears in the running time of operations on the tree-based implementation, and then, by a complex amortized analysis, proves an upper bound on the running time that is just barely superlinear.

19.1 Disjoint-set operations

A ***disjoint-set data structure*** maintains a collection $\mathscr{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. To identify each set, choose a ***representative***, which is some member of the set. In some applications, it doesn't matter which member is used as the representative; it matters only that if you ask for the representative of a dynamic set twice without modifying the set between the requests, you get the same answer both times. Other applications may require a prespecified rule for choosing the representative, such as choosing the smallest member in the set (for a set whose elements can be ordered).

As in the other dynamic-set implementations we have studied, each element of a set is represented by an object. Letting x denote an object, we'll see how to support the following operations:

MAKE-SET(x), where x does not already belong to some other set, creates a new set whose only member (and thus representative) is x.

UNION(x, y) unites two disjoint, dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since the sets in the collection must at all times be disjoint, the UNION operation destroys sets S_x and S_y, removing them from the collection $\mathscr{S}$. In practice, implementations often absorb the elements of one of the sets into the other set.

FIND-SET(x) returns a pointer to the representative of the unique set containing x.

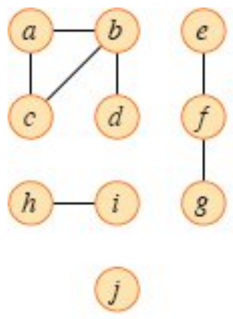
Throughout this chapter, we'll analyze the running times of disjoint-set data structures in terms of two parameters: n, the number of MAKE-SET operations, and m, the total number of MAKE-SET, UNION, and FIND-SET operations. Because the total number of operations m includes the n MAKE-SET operations, $m \geq n$. The first n operations are always MAKE-SET operations, so that after the first n operations, the collection consists of n singleton sets. Since the sets are disjoint at all times, each UNION operation reduces the number of sets by 1. After $n - 1$ UNION operations, therefore, only one set remains, and so at most $n - 1$ UNION operations can occur.

An application of disjoint-set data structures

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph (see Section B.4). Figure 19.1(a), for example, shows a graph with four connected components.

The procedure CONNECTED-COMPONENTS on the following page uses the disjoint-set operations to compute the connected components of a graph. Once the CONNECTED-COMPONENTS procedure has preprocessed the graph, the procedure SAME-COMPONENT answers queries about whether two vertices belong to the same connected component. In pseudocode, we denote the set of vertices of a graph G by $G.V$ and the set of edges by $G.E$.

The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set. Then, for each edge (u, v), it unites the sets containing u and v. By Exercise 19.1-2, after all the edges are processed, two vertices belong to the same connected component if and only if the objects corresponding to the vertices belong to the same set. Thus CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component. Figure 19.1(b) illustrates how CONNECTED-COMPONENTS computes the disjoint sets.



(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, f)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(f, g)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

(b)

Figure 19.1 (a) A graph with four connected components: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, and $\{j\}$. **(b)** The collection of disjoint sets after processing each edge.

CONNECTED-COMPONENTS(G)

```
for each vertex $v \in G.V$
    MAKE-SET($v$)
for each edge $(u, v) \in G.E$
    if FIND-SET($u$) ≠ FIND-SET($v$)
        UNION($u$, $v$)
```

```
SAME-COMPONENT($u$, $v$)
if FIND-SET($u$) == FIND-SET($v$)
    return TRUE
else return FALSE
```

In an actual implementation of this connected-components algorithm, the representations of the graph and the disjoint-set data structure would need to reference each other. That is, an object representing a vertex would contain a pointer to the corresponding disjoint-set object, and vice versa. Since these programming details depend on the implementation language, we do not address them further here.

When the edges of the graph are static—not changing over time—depth-first search can compute the connected components faster (see Exercise 20.3-12 on page 572). Sometimes, however, the edges are added dynamically, with the connected components updated as each edge is added. In this case, the implementation given here can be more efficient than running a new depth-first search for each new edge.

Exercises

19.1-1

The CONNECTED-COMPONENTS procedure is run on the undirected graph $G = (V, E)$, where $V = \{a, b, c, d, e, f, g, h, i, j, k\}$, and the edges of E are processed in the order (d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e). List the vertices in each connected component after each iteration of lines 3–5.

19.1-2

Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices belong to the same connected component if and only if they belong to the same set.

19.1-3

During the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components, how many times is FIND-SET called? How many times is UNION called? Express your answers in terms of $|V|$, $|E|$, and k.

19.2 Linked-list representation of disjoint sets

Figure 19.2(a) shows a simple way to implement a disjoint-set data structure: each set is represented by its own linked list. The object for each set has attributes *head*, pointing to the first object in the list, and *tail*, pointing to the last object. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Within each linked list, the objects may appear in any order. The representative is the set member in the first object in the list.

With this linked-list representation, both MAKE-SET and FIND-SET require only $O(1)$ time. To carry out MAKE-SET(x), create a new linked list whose only object is x. For FIND-SET(x), just follow the pointer from x back to its set object and then return the member in the object that *head* points to. For example, in Figure 19.2(a), the call FIND-SET(g) returns f.

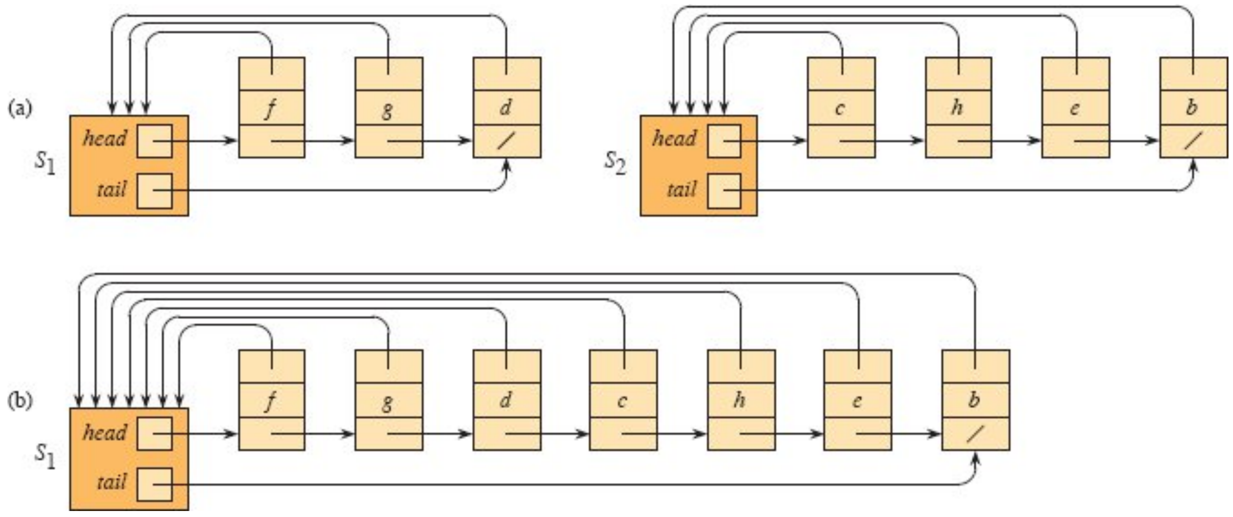
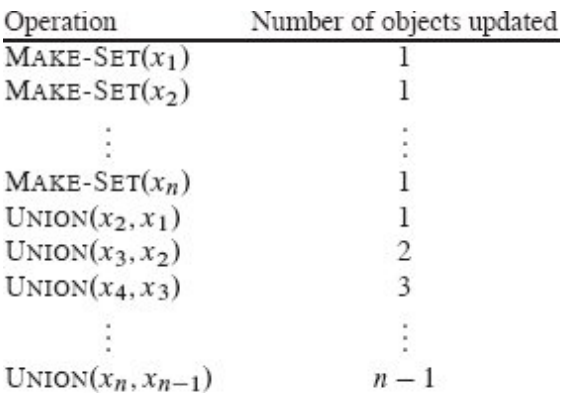


Figure 19.2 (a) Linked-list representations of two sets. Set S_1 contains members *d*, *f*, and *g*, with representative *f*, and set S_2 contains members *b*, *c*, *e*, and *h*, with representative *c*. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. **(b)** The result of UNION(*g*, *e*), which appends the linked list containing *e* to the linked list containing *g*. The representative of the resulting set is *f*. The set object for *e*'s list, S_2, is destroyed.

A simple implementation of union

The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET. As Figure 19.2(b) shows, the operation UNION(*x*, *y*) appends *y*'s list onto the end of *x*'s list. The representative of *x*'s list becomes the representative of the resulting set. To quickly find where to append *y*'s list, use the *tail* pointer for *x*'s list. Because all members of *y*'s list join *x*'s list, the UNION operation destroys the set object for *y*'s list. The UNION operation is where this implementation pays the price for FIND-SET taking constant time: UNION must also update the pointer to the set object for each object originally on *y*'s list, which takes time linear in the length of *y*'s list. In Figure 19.2, for example, the operation UNION(*g*, *e*) causes pointers to be updated in the objects for *b*, *c*, *e*, and *h*.

In fact, we can construct a sequence of m operations on n objects that requires $\Theta(n^2)$ time. Starting with objects $x_1, x_2, \ldots, x_n$, execute the sequence of n MAKE-SET operations followed by $n - 1$ UNION operations shown in Figure 19.3, so that $m = 2n-1$. The n MAKE-SET operations take $\Theta(n)$ time. Because the ith UNION operation updates i objects, the total number of objects updated by all $n-1$ UNION operations forms an arithmetic series:



Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n - 1$

Figure 19.3 A sequence of $2n - 1$ operations on n objects that takes $\Theta(n^2)$ time, or $\Theta(n)$ time per operation on average, using the linked-list set representation and the simple implementation of UNION.

$$\sum_{i=1}^{n-1} i = \Theta(n^2) .$$

The total number of operations is $2n-1$, and so each operation on average requires $\Theta(n)$ time. That is, the amortized time of an operation is $\Theta(n)$.

A weighted-union heuristic

In the worst case, the above implementation of UNION requires an average of $\Theta(n)$ time per call, because it might be appending a longer list onto a shorter list, and the procedure must update the pointer to the set object for each member of the longer list. Suppose instead that each list also includes the length of the list (which can be maintained straightforwardly with constant overhead) and that the UNION procedure always appends the shorter list onto the longer, breaking ties arbitrarily. With this simple ***weighted-union heuristic***, a single UNION operation can still take $\Omega(n)$ time if both sets have $\Omega(n)$ members. As the following theorem shows, however, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Theorem 19.1
Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Proof Because each UNION operation unites two disjoint sets, at most $n - 1$ UNION operations occur over all. We now bound the total time taken by these UNION operations. We start by determining, for each object, an upper bound on the number of times the object's pointer back to its set object is updated. Consider a particular object x. Each time x's pointer is updated, x must have started in the smaller set. The first time x's pointer is updated, therefore, the resulting set must have at least 2 members. Similarly, the next time x's pointer is updated, the resulting set must have had at least 4 members. Continuing on, for any $k \le n$, after x's pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least k members. Since the largest set has at most n members, each object's pointer is updated at most $\lceil \lg n \rceil$ times over all the UNION operations. Thus the total time spent updating object pointers over all UNION operations is $O(n \lg n)$. We must also account for updating the *tail* pointers and the list lengths, which take only $\Theta(1)$ time per UNION operation. The total time spent in all UNION operations is thus $O(n \lg n)$.

The time for the entire sequence of m operations follows. Each MAKE-SET and FIND-SET operation takes $O(1)$ time, and there are $O(m)$ of them. The total time for the entire sequence is thus $O(m + n \lg n)$.

■

Exercises

19.2-1

Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic. Make sure to specify the attributes that you assume for set objects and list objects.

19.2-2

Show the data structure that results and the answers returned by the FIND-SET operations in the following program. Use the linked-list representation with the weighted-union heuristic. Assume that if the sets containing x_i and x_j have the same size, then the operation UNION(x_i, x_j) appends x_j's list onto x_i's list.

```
for $i = 1$ to 16
    MAKE-SET($x_i$)
for $i = 1$ to 15 by 2
    UNION($x_i$, $x_{i+1}$)
for $i = 1$ to 13 by 4
    UNION($x_i$, $x_{i+2}$)
UNION($x_1$, $x_5$)
UNION($x_{11}$, $x_{13}$)
UNION($x_1$, $x_{10}$)
FIND-SET($x_2$)
FIND-SET($x_9$)
```

19.2-3

Adapt the aggregate proof of Theorem 19.1 to obtain amortized time bounds of $O(1)$ for MAKE-SET and FIND-SET and $O(\lg n)$ for UNION using the linked-list representation and the weighted-union heuristic.

19.2-4

Give a tight asymptotic bound on the running time of the sequence of operations in Figure 19.3 assuming the linked-list representation and the weighted-union heuristic.

19.2-5

Professor Gompers suspects that it might be possible to keep just one pointer in each set object, rather than two (*head* and *tail*), while keeping the number of pointers in each list element at two. Show that the professor's suspicion is well founded by describing how to represent each set by a linked list such that each operation has the same running time as the operations described in this section. Describe also how the operations work. Your scheme should allow for the weighted-union heuristic, with the same effect as described in this section. (*Hint*: Use the tail of a linked list as its set's representative.)

19.2-6

Suggest a simple change to the UNION procedure for the linked-list representation that removes the need to keep the *tail* pointer to the last object in each list. Regardless of whether the weighted-union heuristic is used, your change should not change the asymptotic running time of the UNION procedure. (*Hint*: Rather than appending one list to another, splice them together.)

19.3 Disjoint-set forests

A faster implementation of disjoint sets represents sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest***, illustrated in Figure 19.4(a), each member points only to its parent. The root of each tree contains the representative and is its own parent. As we'll see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, two heuristics—"union by rank" and "path compression"—yield an asymptotically optimal disjoint-set data structure.

The three disjoint-set operations have simple implementations. A MAKE-SET operation simply creates a tree with just one node. A FIND-SET operation follows parent pointers until it reaches 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 19.4(b), simply causes the root of one tree to point to the root of the other.

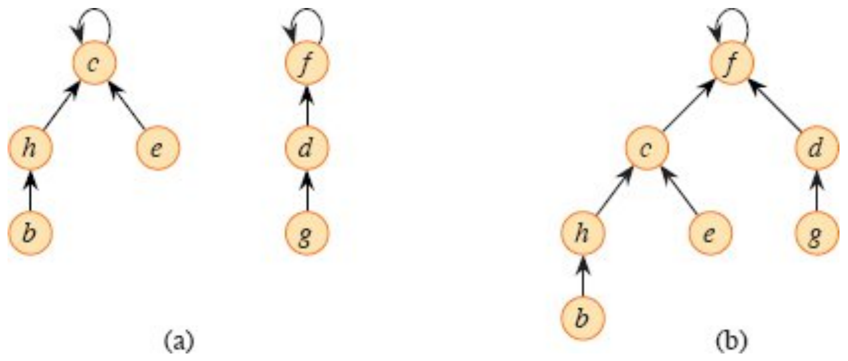


Figure 19.4 A disjoint-set forest. **(a)** Trees representing the two sets of Figure 19.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of UNION (e, g).

Heuristics to improve the running time

So far, disjoint-set forests have not improved on the linked-list implementation. A sequence of $n - 1$ UNION operations could 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 m of operations.

The first heuristic, ***union by rank***, is similar to the weighted-union heuristic we used with the linked-list representation. The common-sense approach is 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, however, we'll adopt an approach that eases the analysis. For each node, maintain a ***rank***, which is an upper bound on the height of the node. Union by rank makes 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 19.5, FIND-SET operations use it to make each node on the find path point directly to the root. Path compression does not change any ranks.

Pseudocode for disjoint-set forests

The union-by-rank heuristic requires its implementation to keep track of ranks. With each node x, 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 ranks, make the root with higher rank the parent of the root with lower rank, but don't change the ranks themselves. If the roots have equal ranks, arbitrarily choose one of the roots as the parent and increment its rank.

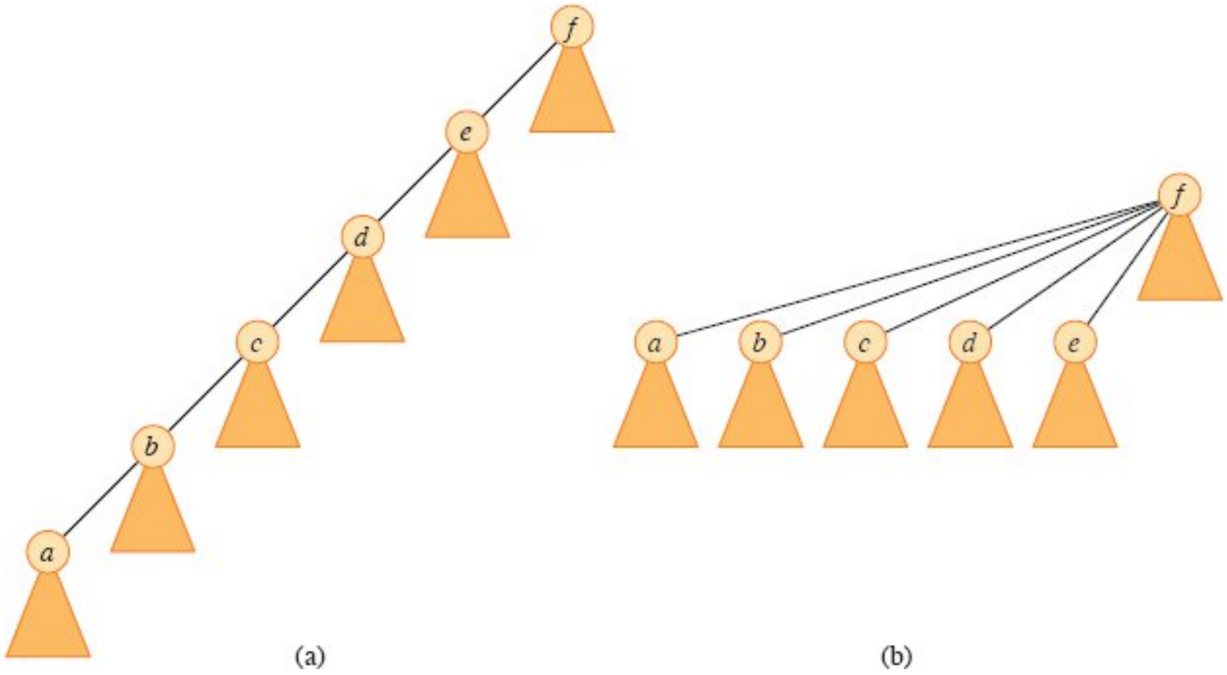


Figure 19.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

Let's put this method into pseudocode, appearing on the next page. The parent of node x is denoted by $x.p$. The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs. The FIND-SET procedure with path compression, implemented recursively, turns out to be quite simple.

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 just returns $x.p$, which is x. In this case 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.

```
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          $y.rank = y.rank + 1$

FIND-SET($x$)
1  if $x \neq x.p$                   // not the root?
2      $x.p$ = FIND-SET($x.p$)       // the root becomes the parent
3  return $x.p$                      // return the root
```

Effect of the heuristics on the running time

Separately, either union by rank or path compression improves the running time of the operations on disjoint-set forests, and combining the two heuristics yields an even greater improvement. Alone, union by rank yields a running time of $O(m \lg n)$ for a sequence of m operations, n of which are MAKE-SET (see Exercise 19.4-4), and this bound is tight (see

Exercise 19.3-3). Although we won't prove it here, for a sequence of n MAKE-SET operations (and hence at most $n - 1$ UNION operations) and f FIND-SET operations, the worst-case running time using only the path-compression heuristic is $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$.

Combining union by rank and path compression gives a worst-case running time of $O(m\,\alpha(n))$, where $\alpha(n)$ is a *very* slowly growing function, defined in Section 19.4. In any conceivable application of a disjoint-set data structure, $\alpha(n) \le 4$, and thus, its running time is as good as linear in m for all practical purposes. Mathematically speaking, however, it is superlinear. Section 19.4 proves this $O(m\alpha(n))$ upper bound.

Exercises

19.3-1
Redo Exercise 19.2-2 using a disjoint-set forest with union by rank and path compression. Show the resulting forest with each node including its x_i and rank.

19.3-2
Write a nonrecursive version of FIND-SET with path compression.

19.3-3
Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when using only union by rank and not path compression.

19.3-4
Consider the operation PRINT-SET(x), which is given a node x and prints all the members of x's set, in any order. Show how to add just a single attribute to each node in a disjoint-set forest so that PRINT-SET(x) takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that you can print each member of the set in $O(1)$ time.

★ 19.3-5
Show that any sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations appear before any of the

FIND-SET operations, takes only $O(m)$ time when using both path compression and union by rank. You may assume that the arguments to LINK are roots within the disjoint-set forest. What happens in the same situation when using only path compression and not union by rank?

★ 19.4 Analysis of union by rank with path compression

As noted in Section 19.3, the combined union-by-rank and path-compression heuristic runs in $O(m\ \alpha(n))$ time for m disjoint-set operations on n elements. In this section, we'll explore the function α to see just how slowly it grows. Then we'll analyze the running time using the potential method of amortized analysis.

A very quickly growing function and its very slowly growing inverse

For integers $j, k \geq 0$, we define the function $A_k(j)$ as

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0\,, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1\,, \end{cases} \tag{19.1}$$

where the expression $A_{k-1}^{(j+1)}(j)$ uses the functional-iteration notation defined in equation (3.30) on page 68. Specifically, equation (3.30) gives $A_{k-1}^{(0)}(j) = j$ and $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ for $i \geq 1$. We call the parameter k the ***level*** of the function A.

The function $A_k(j)$ strictly increases with both j and k. To see just how quickly this function grows, we first obtain closed-form expressions for $A_1(j)$ and $A_2(j)$.

Lemma 19.2
For any integer $j \geq 1$, we have $A_1(j) = 2j + 1$.

Proof We first use induction on i to show that $A_0^{(i)}(j) = j+i$. For the base case, $A_0^{(0)}(j) = j = j+0$. For the inductive step, assume that

$A_0^{(i-1)}(j) = j + (i-1)$. Then $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j + (i-1)) + 1 = j + i$. Finally, we note that $A_1(j) = A_0^{(j+1)}(j) = j + (j+1) = 2j + 1$.

■

Lemma 19.3

For any integer $j \geq 1$, we have $A_2(j) = 2^{j+1}(j+1) - 1$.

Proof We first use induction on i to show that $A_1^{(i)}(j) = 2^i(j+1) - 1$. For the base case, we have $A_1^{(0)}(j) = j = 2^0(j+1) - 1$. For the inductive step, assume that $A_0^{(i-1)}(j) = j + (i-1)$. Then $A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j+1) - 1) = 2\cdot(2^{i-1}(j+1)-1)+1 = 2^i(j+1)-2+1 = 2^i(j+1)-1$. Finally, we note that $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1$.

Now we can see how quickly $A_k(j)$ grows by simply examining $A_k(1)$ for levels $k = 0, 1, 2, 3, 4$. From the definition of $A_0(j)$ and the above lemmas, we have $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$, and $A_2(1) = 2^{1+1} \cdot (1+1) - 1 = 7$. We also have

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^8 \cdot 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047 \end{aligned}$$

and

$$\begin{aligned} A_4(1) &= A_3^{(2)}(1) \\ &= A_3(A_3(1)) \\ &= A_3(2047) \\ &= A_2^{(2048)}(2047) \end{aligned}$$

$$
\begin{aligned}
&\gg A_2(2047)\\
&= 2^{2048} \cdot 2048 - 1\\
&= 2^{2059} - 1\\
&> 2^{2056}\\
&= (2^4)^{514}\\
&= 16^{514}\\
&\gg 10^{80},
\end{aligned}
$$

which is the estimated number of atoms in the observable universe. (The symbol "$\gg$" denotes the "much-greater-than" relation.)

We define the inverse of the function $A_k(n)$, for integer $n \geq 0$, by

$$\alpha(n) = \min\{k : A_k(1) \geq n\} . \tag{19.2}$$

In words, $\alpha(n)$ is the lowest level k for which $A_k(1)$ is at least n. From the above values of $A_k(1)$, we see that

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 , \\ 1 & \text{for } n = 3 , \\ 2 & \text{for } 4 \leq n \leq 7 , \\ 3 & \text{for } 8 \leq n \leq 2047 , \\ 4 & \text{for } 2048 \leq n \leq A_4(1) . \end{cases}$$

It is only for values of n so large that the term "astronomical" understates them (greater than $A_4(1)$, a huge number) that $\alpha(n) > 4$, and so $\alpha(n) \leq 4$ for all practical purposes.

Properties of ranks

In the remainder of this section, we prove an $O(m\alpha(n))$ bound on the running time of the disjoint-set operations with union by rank and path compression. In order to prove this bound, we first prove some simple properties of ranks.

Lemma 19.4

For all nodes x, we have $x.rank \le x.p.rank$, with strict inequality if $x \ne x.p$ (x is not a root). The value of $x.rank$ is initially 0, increases through time until $x \ne x.p$, and from then on, $x.rank$ does not change. The value of $x.p.rank$ monotonically increases over time.

Proof The proof is a straightforward induction on the number of operations, using the implementations of MAKE-SET, UNION, and FIND-SET that appear on page 530, and is left as Exercise 19.4-1.

■

Corollary 19.5
On the simple path from any node going up toward a root, node ranks strictly increase.

■

Lemma 19.6
Every node has rank at most $n - 1$.

Proof Each node's rank starts at 0, and it increases only upon LINK operations. Because there are at most $n - 1$ UNION operations, there are also at most $n - 1$ LINK operations. Because each LINK operation either leaves all ranks alone or increases some node's rank by 1, all ranks are at most $n - 1$.

■

Lemma 19.6 provides a weak bound on ranks. In fact, every node has rank at most $\lfloor \lg n \rfloor$ (see Exercise 19.4-2). The looser bound of Lemma 19.6 suffices for our purposes, however.

Proving the time bound

In order to prove the $O(m\alpha(n))$ time bound, we'll use the potential method of amortized analysis from Section 16.3. In performing the amortized analysis, it will be convenient to assume that we invoke the LINK operation rather than the UNION operation. That is, since the parameters of the LINK procedure are pointers to two roots, we act as though we perform the appropriate FIND-SET operations separately. The following lemma shows that even if we count the extra FIND-SET operations induced by UNION calls, the asymptotic running time remains unchanged.

Lemma 19.7
Suppose that we convert a sequence S' of m' MAKE-SET, UNION, and FIND-SET operations into a sequence S of m MAKE-SET, LINK, and FIND-SET operations by turning each UNION into two FIND-SET operations followed by one LINK. Then, if sequence S runs in $O(m\alpha(n))$ time, sequence S' runs in $O(m' \alpha(n))$ time.

Proof Since each UNION operation in sequence S' is converted into three operations in S, we have $m' \le m \le 3m'$, so that $m = \Theta(m')$, Thus, an $O(m\ \alpha(n))$ time bound for the converted sequence S implies an $O(m'\ \alpha(n))$ time bound for the original sequence S'.

■

From now on, we assume that the initial sequence of m' MAKE-SET, UNION, and FIND-SET operations has been converted to a sequence of m MAKE-SET, LINK, and FIND-SET operations. We now prove an $O(m\ \alpha(n))$ time bound for the converted sequence and appeal to Lemma 19.7 to prove the $O(m'\ \alpha(n))$ running time of the original sequence of m' operations.

Potential function

The potential function we use assigns a potential $\phi_q(x)$ to each node x in the disjoint-set forest after q operations. For the potential Φ_q of the entire forest after q operations, sum the individual node potentials: $\Phi_q = \sum_x \phi_q(x)$. Because the forest is empty before the first operation, the sum is taken over an empty set, and so $\Phi_0 = 0$. No potential Φ_q is ever negative.

The value of $\phi_q(x)$ depends on whether x is a tree root after the qth operation. If it is, or if $x.rank = 0$, then $\phi_q(x) = \alpha(n) \cdot x.rank$.

Now suppose that after the qth operation, x is not a root and that $x.rank \ge 1$. We need to define two auxiliary functions on x before we can define $\phi_q(x)$. First we define

$$\text{level}(x) = \max\{k : x.p.rank \geq A_k(x.rank)\} \ . \tag{19.3}$$

That is, level(x) is the greatest level k for which A_k, applied to x's rank, is no greater than x's parent's rank.

We claim that

$$0 \leq \text{level}(x) < \alpha(n) \ , \tag{19.4}$$

which we see as follows. We have

$$\begin{aligned} x.p.rank &\geq x.rank + 1 && \text{(by Lemma 19.4 because } x \text{ is not a root)} \\ &= A_0(x.rank) && \text{(by the definition (19.1) of } A_0(j)\text{)}, \end{aligned}$$

which implies that level(x) ≥ 0, and

$$\begin{aligned} A_{\alpha(n)}(x.rank) &\geq A_{\alpha(n)}(1) && \text{(because } A_k(j) \text{ is strictly increasing)} \\ &\geq n && \text{(by the definition (19.2) of } \alpha(n)\text{)} \\ &> x.p.rank && \text{(by Lemma 19.6)}, \end{aligned}$$

which implies that level(x) < $\alpha(n)$.

For a given nonroot node x, the value of level(x) monotonically increases over time. Why? Because x is not a root, its rank does not change. The rank of $x.p$ monotonically increases over time, since if $x.p$ is not a root then its rank does not change, and if $x.p$ is a root then its rank can never decrease. Thus, the difference between $x.rank$ and $x.p.rank$ monotonically increases over time. Therefore, the value of k needed for $A_k(x.rank)$ to overtake $x.p.rank$ monotonically increases over time as well.

The second auxiliary function applies when $x.rank \geq 1$: iter(x) = max

$$\text{iter}(x) = \max\left\{i : x.p.rank \geq A^{(i)}_{\text{level}(x)}(x.rank)\right\} \ . \tag{19.5}$$

That is, iter(x) is the largest number of times we can iteratively apply $A_{\text{level}(x)}$, applied initially to x's rank, before exceeding x's parent's rank.

We claim that when $x.rank \geq 1$, we have

$$1 \leq \text{iter}(x) \leq x.rank \ , \tag{19.6}$$

which we see as follows. We have

$$\begin{aligned} x.p.rank &\geq A_{\text{level}(x)}(x.rank) && \text{(by the definition (19.3) of level}(x)) \\ &= A^{(1)}_{\text{level}(x)}(x.rank) && \text{(by the definition (3.30) of functional iteration),} \end{aligned}$$

which implies that iter(x) ≥ 1. We also have

$$\begin{aligned} A^{(x.rank+1)}_{\text{level}(x)}(x.rank) &= A_{\text{level}(x)+1}(x.rank) && \text{(by the definition (19.1) of } A_k(j)) \\ &> x.p.rank && \text{(by the definition (19.3) of level}(x)), \end{aligned}$$

which implies that iter(x) ≤ *x.rank*. Note that because *x.p.rank* monotonically increases over time, in order for iter(x) to decrease, level(x) must increase. As long as level(x) remains unchanged, iter(x) must either increase or remain unchanged.

With these auxiliary functions in place, we are ready to define the potential of node x after q operations:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{if } x \text{ is a root or } x.rank = 0\ , \\ (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) & \\ & \text{if } x \text{ is not a root and } x.rank \geq 1\ . \end{cases} \tag{19.7}$$

We next investigate some useful properties of node potentials.

Lemma 19.8

For every node x, and for all operation counts q, we have

$0 \leq \phi_q(x) \leq \alpha(n) \cdot x.rank.$

Proof If x is a root or *x.rank* = 0, then $\phi_q(x) = \alpha(n) \cdot x.rank$ by definition. Now suppose that x is not a root and that *x.rank* ≥ 1. We can obtain a lower bound on $\phi_q(x)$ by maximizing level(x) and iter(x). The bounds (19.4) and (19.6) give $\alpha(n)$ − level(x) ≥ 1 and iter(x) ≤ *x.rank*. Thus, we have

$\phi_q(x) = (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x)$

$$\begin{aligned} &\geq x.rank - x.rank \\ &= 0. \end{aligned}$$

Similarly, minimizing level(x) and iter(x) provides an upper bound on $\phi_q(x)$. By the bound (19.4), level(x) ≥ 0, and by the bound (19.6), iter(x) ≥ 1. Thus, we have

$$\begin{aligned} \phi_q(x) &\leq (\alpha(n) - 0) \cdot x.rank - 1 \\ &= \alpha(n) \cdot x.rank - 1 \\ &< \alpha(n) \cdot x.rank. \end{aligned}$$

■

Corollary 19.9

If node x is not a root and $x.rank > 0$, then $\phi_q(x) < \alpha(n) \cdot x.rank$.

Potential changes and amortized costs of operations

We are now ready to examine how the disjoint-set operations affect node potentials. Once we understand how each operation can change the potential, we can determine the amortized costs.

Lemma 19.10

Let x be a node that is not a root, and suppose that the qth operation is either a LINK or a FIND-SET. Then after the qth operation, $\phi_q(x) \leq \phi_{q-1}(x)$. Moreover, if $x.rank \geq 1$ and either level(x) or iter(x) changes due to the qth operation, then $\phi_q(x) \leq \phi_{q-1}(x) - 1$. That is, x's potential cannot increase, and if it has positive rank and either level(x) or iter(x) changes, then x's potential drops by at least 1.

Proof Because x is not a root, the qth operation does not change $x.rank$, and because n does not change after the initial n MAKE-SET operations, $\alpha(n)$ remains unchanged as well. Hence, these components of the formula for x's potential remain the same after the qth operation. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$.

Now assume that $x.rank \geq 1$. Recall that level(x) monotonically increases over time. If the qth operation leaves level(x) unchanged, then iter(x) either increases or remains unchanged. If both level(x) and iter(x) are unchanged, then $\phi_q(x) = \phi_{q-1}(x)$. If level($x$) is unchanged and iter($x$) increases, then it increases by at least 1, and so $\phi_q(x) \leq \phi_{q-1}(x) - 1$.

Finally, if the qth operation increases level(x), it increases by at least 1, so that the value of the term $(\alpha(n) - \text{level}(x)) \cdot x.rank$ drops by at least $x.rank$. Because level(x) increased, the value of iter(x) might drop, but according to the bound (19.6), the drop is by at most $x.rank - 1$. Thus, the increase in potential due to the change in iter(x) is less than the decrease in potential due to the change in level(x), yielding $\phi_q(x) \leq \phi_{q-1}(x) - 1$.

■

Our final three lemmas show that the amortized cost of each MAKE-SET, LINK, and FIND-SET operation is $O(\alpha(n))$. Recall from equation (16.2) on page 456 that the amortized cost of each operation is its actual cost plus the change in potential due to the operation.

Lemma 19.11
The amortized cost of each MAKE-SET operation is $O(1)$.

Proof Suppose that the qth operation is MAKE-SET(x). This operation creates node x with rank 0, so that $\phi_q(x) = 0$. No other ranks or potentials change, and so $\Phi_q = \Phi_{q-1}$. Noting that the actual cost of the MAKE-SET operation is $O(1)$ completes the proof.

■

Lemma 19.12
The amortized cost of each LINK operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is LINK(x, y). The actual cost of the LINK operation is $O(1)$. Without loss of generality, suppose that the LINK makes y the parent of x.

To determine the change in potential due to the LINK, note that the only nodes whose potentials may change are x, y, and the children of y just prior to the operation. We'll show that the only node whose potential can increase due to the LINK is y, and that its increase is at most $\alpha(n)$:

- By Lemma 19.10, any node that is y's child just before the LINK cannot have its potential increase due to the LINK.
- From the definition (19.7) of $\phi_q(x)$, note that, since x was a root just before the qth operation, $\phi_{q-1}(x) = \alpha(n) \cdot x.rank$ at that time. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$. Otherwise,

  $$\begin{aligned}\phi_q(x) &< \alpha(n) \cdot x.rank \text{ (by Corollary 19.9)} \\ &= \phi_{q-1}(x),\end{aligned}$$

  and so x's potential decreases.
- Because y is a root prior to the LINK, $\phi_{q-1}(y) = \alpha(n) \cdot y.rank$. After the LINK operation, y remains a root, so that y's potential still equals $\alpha(n)$ times its rank after the operation. The LINK operation either leaves y's rank alone or increases y's rank by 1. Therefore, either $\phi_q(y) = \phi_{q-1}(y)$ or $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

The increase in potential due to the LINK operation, therefore, is at most $\alpha(n)$. The amortized cost of the LINK operation is $O(1) + \alpha(n) = O(\alpha(n))$.

■

Lemma 19.13
The amortized cost of each FIND-SET operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is a FIND-SET and that the find path contains s nodes. The actual cost of the FIND-SET operation is $O(s)$. We will show that no node's potential increases due to the FIND-SET and that at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes on the find path have their potential decrease by at least 1.

We first show that no node's potential increases. Lemma 19.10 takes care of all nodes other than the root. If x is the root, then its potential is $\alpha(n) \cdot x.rank$, which does not change due to the FIND-SET operation.

Now we show that at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes have their potential decrease by at least 1. Let x be a node on the find path such that $x.rank > 0$ and x is followed somewhere on the find path by another node y that is not a root, where level(y) = level(x) just before the FIND-SET operation. (Node y need not *immediately* follow x on the find path.) All but at most $\alpha(n) + 2$ nodes on the find path satisfy these constraints on x. Those that do not satisfy them are the first node on the find path (if it has rank 0), the last node on the path (i.e., the root), and the last node w on the path for which level(w) = k, for each $k = 0, 1, 2, \ldots, \alpha(n) -1$.

Consider such a node x. It has positive rank and is followed somewhere on the find path by nonroot node y such that level(y) = level(x) before the path compression occurs. We claim that the path compression decreases x's potential by at least 1. To prove this claim, let k = level(x) = level(y) and i = iter(x) before the path compression occurs. Just prior to the path compression caused by the FIND-SET, we have

$x.p.rank \geq A_k^{(i)}(x.rank)$ (by the definition (19.5) of iter(x)),

$y.p.rank \geq A_k(y.rank)$ (by the definition (19.3) of level(y)),

$y.rank \geq x.p.rank$ (by Corollary 19.5 and because y follows x on the find path).

Putting these inequalities together gives

$$\begin{aligned} y.p.rank &\geq A_k(y.rank) \\ &\geq A_k(x.p.rank) && \text{(because } A_k(j) \text{ is strictly increasing)} \\ &\geq A_k(A_k^{(i)} \\ &= A_k^{(i+1)}(x.rank) && \text{(by the definition (3.30) of functional iteration).} \end{aligned}$$

Because path compression makes x and y have the same parent, after path compression we have $x.p.rank = y.p.rank$. The parent of y might change due to the path compression, but if it does, the rank of y's new parent compared with the rank of *y*'s parent before path compression is either the same or greater. Since *x.rank* does not change, $x.p.rank = y.p.rank \geq A_k^{(i+1)}(x.rank)$ after path compression. By the definition (19.5) of the iter function, the value of iter(x) increases from i to at least $i + 1$. By Lemma 19.10, $\phi_q(x) \leq \phi_{q-1}(x) - 1$, so that x's potential decreases by at least 1.

The amortized cost of the FIND-SET operation is the actual cost plus the change in potential. The actual cost is $O(s)$, and we have shown that the total potential decreases by at least $\max\{0, s - (\alpha(n) + 2)\}$. The amortized cost, therefore, is at most $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, since we can scale up the units of potential to dominate the constant hidden in $O(s)$. (See Exercise 19.4-6.)

■

Putting the preceding lemmas together yields the following theorem.

Theorem 19.14
A sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, can be performed on a disjoint-set forest with union by rank and path compression in $O(m\,\alpha(n))$ time.

Proof Immediate from Lemmas 19.7, 19.11, 19.12, and 19.13.

■

Exercises

19.4-1
Prove Lemma 19.4.

19.4-2
Prove that every node has rank at most $\lfloor \lg n \rfloor$.

19.4-3
In light of Exercise 19.4-2, how many bits are necessary to store *x.rank* for each node x?

19.4-4

Using Exercise 19.4-2, give a simple proof that operations on a disjoint-set forest with union by rank but without path compression run in $O(m \lg n)$ time.

19.4-5

Professor Dante reasons that because node ranks increase strictly along a simple path to the root, node levels must monotonically increase along the path. In other words, if $x.rank > 0$ and $x.p$ is not a root, then $\text{level}(x) \le \text{level}(x.p)$. Is the professor correct?

19.4-6

The proof of Lemma 19.13 ends with scaling the units of potential to dominate the constant hidden in the $O(s)$ term. To be more precise in the proof, you need to change the definition (19.7) of the potential function to multiply each of the two cases by a constant, say c, that dominates the constant in the $O(s)$ term. How must the rest of the analysis change to accommodate this updated potential function?

★ 19.4-7

Consider the function $\alpha'(n) = \min\{k : A_k(1) \ge \lg(n + 1)\}$. Show that $\alpha'(n) \le 3$ for all practical values of n and, using Exercise 19.4-2, show how to modify the potential-function argument to prove that performing a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, on a disjoint-set forest with union by rank and path compression takes $O(m\alpha'(n))$ time.

Problems

19-1 Offline minimum

In the ***offline minimum problem***, you maintain a dynamic set T of elements from the domain $\{1, 2, \ldots, n\}$ under the operations INSERT and EXTRACT-MIN. The input is a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \ldots, n\}$ is inserted exactly once. Your goal is to determine which key is returned by each EXTRACT-MIN call. Specifically, you must fill in an array $extracted[1:m]$, where for $i = 1, 2, \ldots, m$, $extracted[i]$ is the key returned by the ith

EXTRACT-MIN call. The problem is “offline” in the sense that you are allowed to process the entire sequence S before determining any of the returned keys.

a. Consider the following instance of the offline minimum problem, in which each operation INSERT(i) is represented by the value of i and each EXTRACT-MIN is represented by the letter E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.

Fill in the correct values in the *extracted* array.

To develop an algorithm for this problem, break the sequence S into homogeneous subsequences. That is, represent S by

I_1, E, I_2, E, I_3, … , I_m, E, I_{m+1},

where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j, initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. Then execute the OFFLINE-MINIMUM procedure.

```
OFFLINE-MINIMUM($m$, $n$)
1  for $i = 1$ to $n$
2      determine $j$ such that $i \in K_j$
3      if $j \neq m + 1$
4          $extracted[j] = i$
5          let $l$ be the smallest value greater than $j$ for which set $K_l$ exists
6          $K_l = K_j \cup K_l$, destroying $K_j$
7  return extracted
```

b. Argue that the array *extracted* returned by OFFLINE-MINIMUM is correct.

c. Describe how to implement OFFLINE-MINIMUM efficiently with a disjoint-set data structure. Give as tight a bound as you can on the worst-case running time of your implementation.

19-2 Depth determination

In the ***depth-determination problem***, you maintain a forest $\mathcal{F} = \{T_i\}$ of rooted trees under three operations:

MAKE-TREE(v) creates a tree whose only node is v.

FIND-DEPTH(v) returns the depth of node v within its tree.

GRAFT(r, v) makes node r, which is assumed to be the root of a tree, become the child of node v, which is assumed to be in a different tree from r but may or may not itself be a root.

a. Suppose that you use a tree representation similar to a disjoint-set forest: $v.p$ is the parent of node v, except that $v.p = v$ if v is a root. Suppose further that you implement GRAFT(r, v) by setting $r.p = v$ and FIND-DEPTH(v) by following the find path from v up to the root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations is $\Theta(m^2)$.

By using the union-by-rank and path-compression heuristics, you can reduce the worst-case running time. Use the disjoint-set forest $\mathcal{S} = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest $\mathcal{F}$. The tree structure within a set S_i, however, does not necessarily correspond to that of T_i. In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows you to determine any node's depth in T_i.

The key idea is to maintain in each node v a "pseudodistance" $v.d$, which is defined so that the sum of the pseudodistances along the simple path from v to the root of its set S_i equals the depth of v in T_i. That is, if the simple path from v to its root in S_i is $v_0, v_1, \ldots, v_k$, where $v_0 = v$ and v_k is S_i's root, then the depth of v in T_i is $\sum_{j=0}^{k} v_j.d$.

b. Give an implementation of MAKE-TREE.

c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudodistances correctly.

d. Show how to implement GRAFT(r, v), which combines the sets containing r and v, by modifying the UNION and LINK procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.

e. Give a tight bound on the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations, n of which are MAKE-TREE operations.

19-3 Tarjan's offline lowest-common-ancestors algorithm

The ***lowest common ancestor*** of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T. In the ***offline lowest-common-ancestors problem***, you are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T, and you wish to determine the lowest common ancestor of each pair in P.

To solve the offline lowest-common-ancestors problem, the LCA procedure on the following page performs a tree walk of T with the initial call LCA($T.root$). Assume that each node is colored WHITE prior to the walk.

a. Argue that line 10 executes exactly once for each pair $\{u, v\} \in P$.

b. Argue that at the time of the call LCA(u), the number of sets in the disjoint-set data structure equals the depth of u in T.

```
LCA($u$)
1 MAKE-SET($u$)
2 FIND-SET($u$).$ancestor$ = $u$
3 for each child $v$ of $u$ in $T$
4     LCA($v$)
5     UNION($u$, $v$)
6     FIND-SET($u$).$ancestor$ = $u$
```

```
$u.color$ = BLACK
for each node $v$ such that $\{u, v\} \in P$
    if $v.color$ == BLACK
        print "The lowest common ancestor of"
            $u$ "and" $v$ "is" FIND-SET($v$).$ancestor$
```

c. Prove that LCA correctly prints the lowest common ancestor of u and v for each pair $\{u, v\} \in P$.

d. Analyze the running time of LCA, assuming that you use the implementation of the disjoint-set data structure in Section 19.3.

Chapter notes

Many of the important results for disjoint-set data structures are due at least in part to R. E. Tarjan. Using aggregate analysis, Tarjan [427, 429] gave the first tight upper bound in terms of the very slowly growing inverse $\widehat{\alpha}(m, n)$ of Ackermann's function. (The function $A_k(j)$ given in Section 19.4 is similar to Ackermann's function, and the function $\alpha(n)$ is similar to $\widehat{\alpha}(m, n)$. Both $\alpha(n)$ and $\widehat{\alpha}(m, n)$ are at most 4 for all conceivable values of m and n.) An upper bound of $O(m \lg^* n)$ was proven earlier by Hopcroft and Ullman [5, 227]. The treatment in Section 19.4 is adapted from a later analysis by Tarjan [431], which is based on an analysis by Kozen [270]. Harfst and Reingold [209] give a potential-based version of Tarjan's earlier bound.

Tarjan and van Leeuwen [432] discuss variants on the path-compression heuristic, including "one-pass methods," which sometimes offer better constant factors in their performance than do two-pass methods. As with Tarjan's earlier analyses of the basic path-compression heuristic, the analyses by Tarjan and van Leeuwen are aggregate. Harfst and Reingold [209] later showed how to make a small change to the potential function to adapt their path-compression analysis to these one-pass variants. Goel et al. [182] prove that linking disjoint-set trees randomly yields the same asymptotic running time as union by rank. Gabow and Tarjan [166] show that in certain applications, the disjoint-set operations can be made to run in $O(m)$ time.