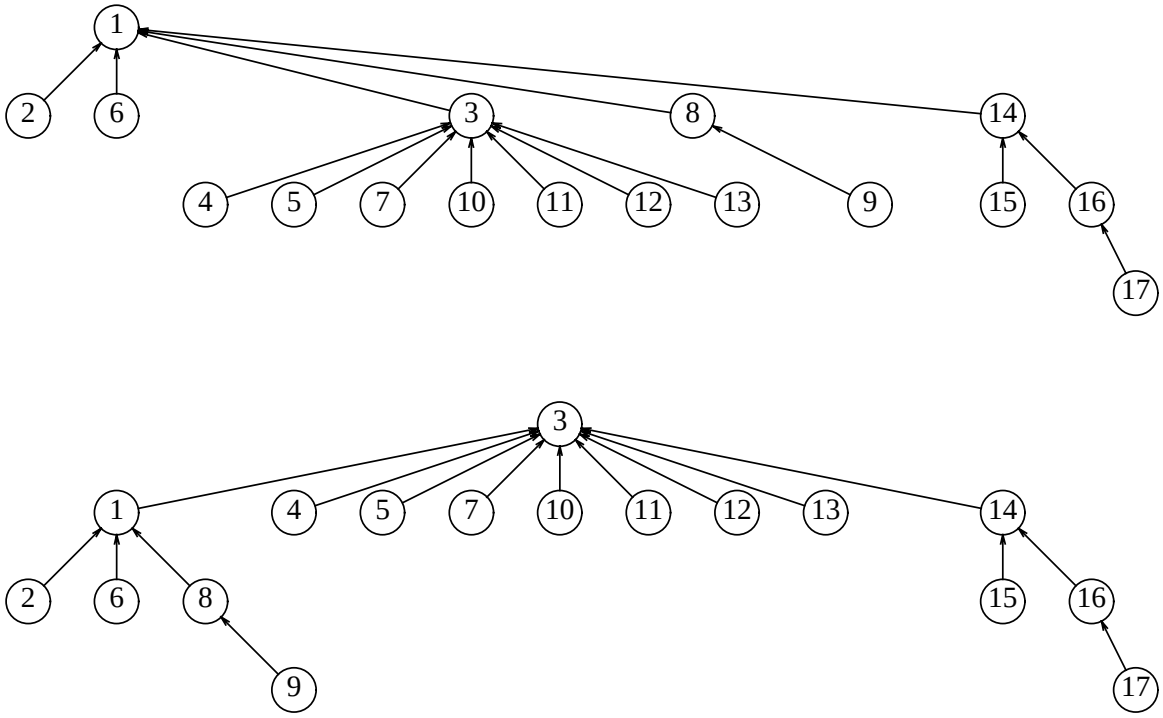
Chapter 8: The Disjoint Set ADT

8.1 We assume that unions operated on the roots of the trees containing the arguments. Also, in case of ties, the second tree is made a child of the first. Arbitrary union and union by height give the same answer (shown as the first tree) for this problem. Union by size gives the second tree.



8.2 In both cases, have nodes 16 and 17 point directly to the root.

8.4 Claim: A tree of height H has at least 2^H nodes. The proof is by induction. A tree of height 0 clearly has at least 1 node, and a tree of height 1 clearly has at least 2. Let T be the tree of height H with fewest nodes. Thus at the time of T's last union, it must have been a tree of height $H-1$, since otherwise T would have been smaller at that time than it is now and still would have been of height H, which is impossible by assumption of T's minimality. Since T's height was updated, it must have been as a result of a union with another tree of height $H-1$. By the induction hypothesis, we know that at the time of the union, T had at least 2^{H-1} nodes, as did the tree attached to it, for a total of 2^H nodes, proving the claim. Thus an N-node tree has depth at most $\lfloor \log N \rfloor$.

8.5 All answers are $O(M)$ because in all cases $\alpha(M, N) = 1$.

8.6 Assuming that the graph has only nine vertices, then the union/find tree that is formed is shown here. The edge (4,6) does not result in a union because at the time it is examined, 4 and 6 are already in the same component. The connected components are {1,2,3,4,6} and

{5,7,8,9}.

8.8 (a) When we perform a union, we push onto a stack the two roots and the old values of their parents. To implement a *Deunion,* we only have to pop the stack and restore the values. This strategy works fine in the absence of path compression.

(b) If path compression is implemented, the strategy described in part (a) does not work because path compression moves elements out of subtrees. For instance, the sequence *Union(1,2), Union(3,4), Union(1,3), Find(4), Deunion(1,3)* will leave 4 in set 1 if path compression is implemented.

8.9 We assume that the tree is implemented with pointers instead of a simple array. Thus *Find* will return a pointer instead of an actual set name. We will keep an array to map set numbers to their tree nodes. *Union* and *Find* are implemented in the standard manner. To perform *Remove(X),* first perform a *Find(X)* with path compression. Then mark the node containing X as vacant. Create a new one-node tree with X and have it pointed to by the appropriate array entry. The time to perform a *Remove* is the same as the time to perform a *Find,* except that there potentially could be a large number of vacant nodes. To take care of this, after N *Remove*s are performed, perform a *Find* on every node, with path compression. If a *Find*(X) returns a vacant root, then place X in the root node, and make the old node containing X vacant. The results of Exercise 8.11 guarantee that this will take linear time, which can be charged to the N *Remove*s. At this point, all vacant nodes (indeed all nonroot nodes) are children of a root, and vacant nodes can be disposed (if an array of pointers to them has been kept). This also guarantees that there are never more than $2N$ nodes in the forest and preserves the $M\alpha(M, N)$ asymptotic time bound.

8.11 Suppose there are u *Union*s and f *Find*s. Each union costs constant time, for a total of u. A *Find* costs one unit per vertex visited. We charge, as in the text, under the following slightly modified rules:

(A) the vertex is a root or child of the root

(B) otherwise

Essentially, all vertices are in one rank group. During any *Find,* there can be at most two rule (A) charges, for a total of $2f$. Each vertex can be charged at most once under rule (B) because after path compression it will be a child of the root. The number of vertices that are not roots or children of roots is clearly bounded by u, independent of the unioning strategy, because each *Union* changes exactly one vertex from root to nonroot status, and this bounds the number of type (B) nodes. Thus the total rule (B) charges are at most u. Adding all charges gives a bound of $2f + 2u$, which is linear in the number of operations.

8.13 For each vertex v, let the pseudorank R_v be defined as $\lfloor \log S_v \rfloor$, where S_v is the number of descendents (including itself) of v in the final tree, after all *Union*s are performed, ignoring

path compression.

Although the pseudorank is not maintained by the algorithm, it is not hard to show that the pseudorank satisfies the same properties as the ranks do in union-by-rank. Clearly, a vertex with pseudorank R_v has at least 2^{R_v} descendents (by its definition), and the number of vertices of pseudorank R is at most $N/2^R$. The union-by-size rule ensures that the parent of a node has twice as many descendents as the node, so the pseudoranks monotonically increase on the path toward the root if there is no path compression. The argument in Lemma 8.3 tells us that path compression does not destroy this property.

If we partition the vertices by pseudoranks and assign the charges in the same manner as in the text proof for union-by-rank, the same steps follow, and the identical bound is obtained.

8.14 This is most conveniently implemented without recursion and is faster because, even if full path compression is implemented nonrecursively, it requires two passes up the tree. This requires only one. We leave the coding to the reader since comparing the various *Union* and *Find* strategies is a reasonable programming project. The worst-case running time remains the same because the properties of the ranks are unchanged. Instead of charging one unit to each vertex on the path to the root, we can charge two units to alternating vertices (namely, the vertices whose parents are altered by path halving). These vertices get parents of higher rank, as before, and the same kind of analysis bounds the total charges.