chapter 24

the disjoint set class

In this chapter we describe an efficient data structure for solving the equivalence problem: the disjoint set class. This data structure is simple to implement, with each routine requiring only a few lines of code. Its implementation is also extremely fast, requiring constant average time per operation. This data structure is also very interesting from a theoretical point of view because its analysis is extremely difficult; the functional form of the worst case is unlike any discussed so far in this text.

In this chapter, we show

- Three simple applications of the disjoint set class
- A way to implement the disjoint set class with minimal coding effort
- A method for increasing the speed of the disjoint set class, using two simple observations
- An analysis of the running time of a fast implementation of the disjoint set class

24.1 equivalence relations

A *relation* is defined on a set if every pair of elements either is related or is not. An *equivalence relation* is reflexive, symmetric, and transitive.

A *relation* R is defined on a set S if for every pair of elements (a, b), $a, b \in S$, $a\ R\ b$ is either true or false. If $a\ R\ b$ is true, we say that a is related to b.

An *equivalence relation* is a relation R that satisfies three properties.

1. *Reflexive:* $a\ R\ a$ is true for all $a \in S$.
2. *Symmetric:* $a\ R\ b$ if and only if $b\ R\ a$.
3. *Transitive:* $a\ R\ b$ and $b\ R\ c$ implies that $a\ R\ c$.

Electrical connectivity, where all connections are by metal wires, is an equivalence relation. The relation is clearly reflexive, as any component is connected to itself. If a is electrically connected to b, then b must be electrically connected to a, so the relation is symmetric. Finally, if a is connected to b and b is connected to c, then a is connected to c.

Likewise, connectivity through a bidirectional network forms equivalence classes of connected components. However, if the connections in the network are directed (i.e., a connection from v to w does not imply one from w to v), we do not have an equivalence relation because the symmetric property does not hold. An example is a relation in which town a is related to town b if traveling from a to b by road is possible. This relationship is an equivalence relation if the roads are two-way.

24.2 dynamic equivalence and applications

For any equivalence relation, denoted ~, the natural problem is to decide for any a and b whether $a \sim b$. If the relation is stored as a two-dimensional array of Boolean variables, equivalence can be tested in constant time. The problem is that the relation is usually implicitly, rather than explicitly, defined.

The *equivalence class* of an element x in set S is the subset of S that contains all the elements related to x. The equivalence classes form *disjoint sets*.

For example, an equivalence relation is defined over the five-element set $\{a_1, a_2, a_3, a_4, a_5\}$. This set yields 25 pairs of elements, each of which either is or is not related. However, the information that $a_1 \sim a_2$, $a_3 \sim a_4$, $a_1 \sim a_5$, and $a_4 \sim a_2$ are all related implies that all pairs are related. We want to be able to infer this condition quickly.

The *equivalence class* of an element $x \in S$ is the subset of S that contains all the elements related to x. Note that the equivalence classes form a partition of S: Every member of S appears in exactly one equivalence class. To decide whether $a \sim b$, we need only check whether a and b are in the same

equivalence class. This information provides the strategy to solve the equivalence problem.

The input is initially a collection of N sets, each with one element. In this initial representation all relations (except reflexive relations) are false. Each set has a different element, so $S_i \cap S_j = \emptyset$ and such sets (in which any two sets contain no common elements) are called *disjoint sets.*

The two basic *disjoint set class operations* are `find`, which returns the name of the set (i.e., the equivalence class) containing a given element, and the `union`, which adds relations. If we want to add the pair (a, b) to the list of relations, we first determine whether a and b are already related. We do so by performing `find` operations on both a and b and finding out whether they are in the same equivalence class; if they are not, we apply `union`. This operation merges the two equivalence classes containing a and b into a new equivalence class. In terms of sets the result is a new set $S_k = S_i \cup S_j$, which we create by simultaneously destroying the originals and preserving the disjointedness of all the sets. The data structure to do this is often called the disjoint set *union/find data structure.* The *union/find algorithm* is executed by processing union/find requests within the disjoint set data structure.

The two basic *disjoint set class operations* are `union` and `find`.

The algorithm is *dynamic* because, during the course of algorithm execution, the sets can change via the `union` operation. The algorithm must also operate as an *online algorithm* so that, when a `find` is performed, an answer must be given before the next query can be viewed. Another possibility is an *offline algorithm* in which the entire sequence of `union` and `find` requests are made visible. The answer it provides for each `find` must still be consistent with all the `unions` performed before the `find`. However, the algorithm can give all its answers after it has dealt with *all* the questions. This distinction is similar to the difference between taking a written exam (which is generally offline because you only have to give the answers before time expires) and taking an oral exam (which is online because you must answer the current question before proceeding to the next question).

In an *online algorithm*, an answer must be provided for each query before the next query can be viewed.

Note that we do not perform any operations to compare the relative values of elements but merely require knowledge of their location. For this reason, we can assume that all elements have been numbered sequentially, starting from 0, and that the numbering can be determined easily by some hashing scheme.

The set elements are numbered sequentially, starting from 0.

Before describing how to implement the `union` and `find` operations, we provide three applications of the data structure.

24.2.1 application: generating mazes

An example of the use of the union/find data structure is to generate mazes, such as the one shown in Figure 24.1. The starting point is the top-left corner,

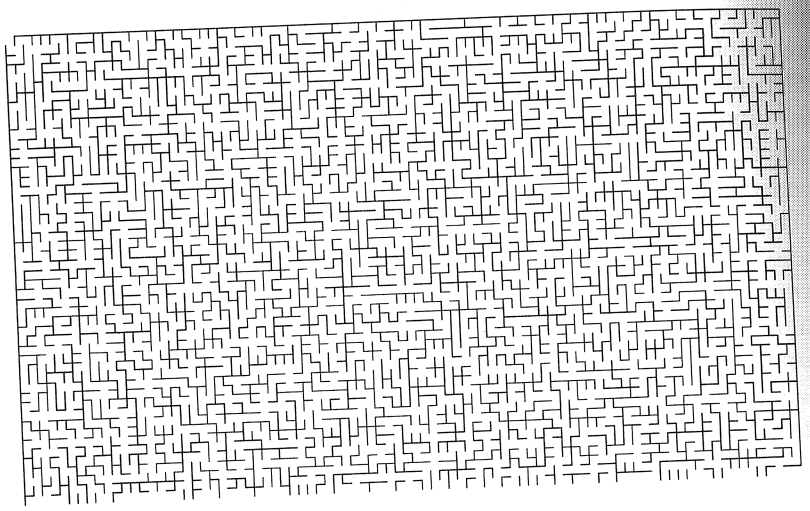


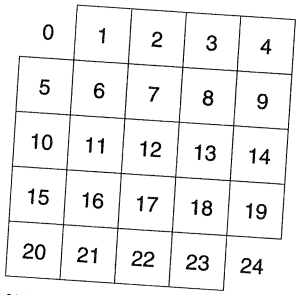
figure 24.1
A 50 × 88 maze

and the ending point is the bottom-right corner. We can view the maze as a 50×88 rectangle of cells in which the top-left cell is connected to the bottom-right cell, and cells are separated from their neighboring cells via walls.

A simple algorithm to generate the maze is to start with walls everywhere (except for the entrance and exit). We then continually choose a wall randomly and knock it down if the cells that the wall separates are not already connected to each other. If we repeat this process until the starting and ending cells are connected, we have a maze. Continuing to knock down walls until every cell is reachable from every other cell is actually better because doing so generates more false leads in the maze.

We illustrate the algorithm with a 5×5 maze, and Figure 24.2 shows the initial configuration. We use the union/find data structure to represent sets of cells that are connected to each other. Initially, walls are everywhere, and each cell is in its own equivalence class.

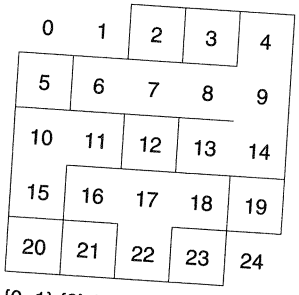
Figure 24.3 shows a later stage of the algorithm, after a few walls have been knocked down. Suppose, at this stage, that we randomly target the wall that connects cells 8 and 13. Because 8 and 13 are already connected (they are in the same set), we would not remove the wall because to do so would simply trivialize the maze. Suppose that we randomly target cells 18 and 13 next. By performing two `find` operations, we determine that these cells are in different sets; thus 18 and 13 are not already connected. Therefore we knock down the wall that separates them, as shown in Figure 24.4. As a



{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14}
{15} {16} {17} {18} {19} {20} {21} {22} {23} {24}

figure 24.2

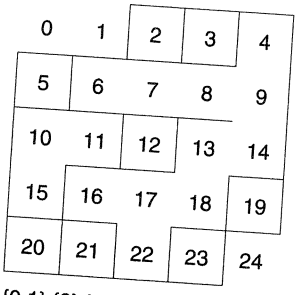
Initial state: All walls are up, and all cells are in their own sets.



{0, 1} {2} {3} {4, 6, 7, 8, 9, 13,14} {5} {10, 11, 15} {12}
{16, 17, 18, 22} {19} {20} {21} {22} {23} {24}

figure 24.3

At some point in the algorithm, several walls have been knocked down and sets have been merged. At this point, if we randomly select the wall between 8 and 13, this wall is not knocked down because 8 and 13 are already connected.



{0,1} {2} {3} {5} {10, 11, 15} {12}
{4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {19} {20} {21} {23} {24}

figure 24.4

We randomly select the wall between squares 18 and 13 in Figure 24.3; this wall has been knocked down because 18 and 13 were not already connected, and their sets have been merged.

result of this operation, the sets containing cells 18 and 13 are combined by a `union` operation. The reason is that all the cells previously connected to 18 are now connected to all the cells previously connected to 13. At the end of the algorithm, as depicted in Figure 24.5, all the cells are connected, and we are done.

The running time of the algorithm is dominated by the union/find costs. The size of the union/find universe is the number of cells. The number of `find` operations is proportional to the number of cells because the number of removed walls is 1 less than the number of cells. If we look carefully, however, we can see that there are only about twice as many walls as cells in the first place. Thus, if N is the number of cells and as there are two `finds` per randomly targeted wall, we get an estimate of between (roughly) $2N$ and $4N$ `find` operations throughout the algorithm. Therefore the algorithm's running time depends on the cost of $O(N)$ `union` and $O(N)$ `find` operations.

24.2.2 application: minimum spanning trees

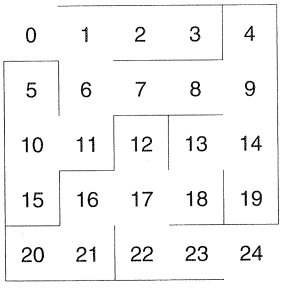
The *minimum spanning tree* is a connected subgraph of G that spans all vertices at minimum total cost.

A *spanning tree* of an undirected graph is a tree formed by graph edges that connect all the vertices of the graph. Unlike the graphs in Chapter 14, an edge (u, v) in a graph G is identical to an edge (v, u). The cost of a spanning tree is the sum of the costs of the edges in the tree. The *minimum spanning tree* is a connected subgraph of G that spans all vertices at minimum cost. A minimum spanning tree exists only if the subgraph of G is connected. As we show shortly, testing a graph's connectivity can be done as part of the minimum spanning tree computation.

In Figure 24.6(b), the graph is a minimum spanning tree of the graph in Figure 24.6(a) (it happens to be unique, which is unusual if the graph has many edges of equal cost). Note that the number of edges in the minimum spanning tree is $|V| - 1$. The minimum spanning tree is a *tree* because it is

figure 24.5

Eventually, 24 walls have been knocked down, and all the elements are in the same set.



{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

```
1     // Nearest Common Ancestors algorithm
2     //
3     // Preconditions (and global objects):
4     //  1. union/find structure is initialized
5     //  2. All nodes are initially unmarked
6     //  3. Preorder numbers are already assigned in num field
7     //  4. Each node can store its marked status
8     //  5. List of pairs is globally available
9
10    DisjSets s = new DisjSets( treeSize );  // union/find
11    Node [ ] anchor = new Node[ treeSize ]; // Anchor node for each set
12
13    // main makes the call NCA( root )
14    // after required initializations
15
16    void NCA( Node u )
17    {
18        anchor[ s.find( u.num ) ] = u;
19
20        // Do postorder calls
21        for( each child v of u )
22        {
23            NCA( v );
24            s.union( s.find( u.num ), s.find( v.num ) );
25            anchor[ s.find( u.num ) ] = u;
26        }
27
28        // Do nca calculation for pairs involving u
29        u.marked = true;
30        for( each v such that NCA( u, v ) is required )
31            if( v.marked )
32                System.out.println( "NCA( " + u + ", " + v +
33                        " ) is " + anchor[ s.find( v.num ) ] );
34    }
```

figure 24.11
Pseudocode for the nearest common ancestors problem

24.3 the quick-find algorithm

In this section and Section 24.4 we lay the groundwork for the efficient implementation of the union/find data structure. There are two basic strategies for solving the union/find problem. The first approach, the *quick-find algorithm,* ensures that the `find` instruction can be executed in constant worst-case time. The other approach, the *quick-union algorithm,* ensures

that the union operation can be executed in constant worst-case time. It has been shown that both cannot be done simultaneously in constant worst-case (or even amortized) time.

For the find operation to be fast, in an array we could maintain the name of the equivalence class for each element. Then find is a simple constant-time lookup. Suppose that we want to perform union(a, b). Suppose, too, that a is in equivalence class i and that b is in equivalence class j. Then we can scan down the array, changing all i's to j's. Unfortunately, this scan takes linear time. Thus a sequence of $N - 1$ union operations (the maximum because then everything is in one set) would take quadratic time. In the typical case in which the number of finds is subquadratic, this time is clearly unacceptable.

One possibility is to keep all the elements that are in the same equivalence class in a linked list. This approach saves time when we are updating because we do not have to search the entire array. By itself that does not reduce the asymptotic running time, as performing $\Theta(N^2)$ equivalence class updates over the course of the algorithm is still possible.

If we also keep track of the size of the equivalence classes—and when performing a union change the name of the smaller class to the larger—the total time spent for N unions is $O(N \log N)$. The reason is that each element can have its equivalence class changed at most $\log N$ times because every time its class is changed, its new equivalence class is at least twice as large as its old class (so the repeated doubling principle applies).

This strategy provides that any sequence of at most M find and $N - 1$ union operations take at most $O(M + N \log N)$ time. If M is linear (or slightly nonlinear), this solution is still expensive. It also is a bit messy because we must maintain linked lists. In Section 24.4 we examine a solution to the union/find problem that makes union easy but find hard—the quick-union algorithm. Even so, the running time for any sequence of at most M find and $N - 1$ union operations is only negligibly more than $O(M + N)$ time and, moreover, only a single array of integers is used.

The argument that an equivalence class can change at most $\log N$ times per item is also used in the quick-union algorithm. Quick-find is a simple algorithm, but quick-union is better.

24.4 the quick-union algorithm

Recall that the union/find problem does not require a find operation to return any specific name; it requires just that finds on two elements return the same answer if and only if they are in the same set. One possibility might be to use a tree to represent a set, as each element in a tree has the same root and the root can be used to name the set.

A tree is represented by an array of integers representing parent nodes. The set name of any node in a tree is the root of a tree.

Each set is represented by a tree (recall that a collection of trees is called a *forest*). The name of a set is given by the node at the root. Our trees are not necessarily binary trees, but their representation is easy because the only information we need is the parent. Thus we need only an array of integers: Each entry `p[i]` in the array represents the parent of element `i`, and we can use –1 as a parent to indicate a root. Figure 24.12 shows a forest and the array that represents it.

The `union` operation is constant time.

To perform a `union` of two sets, we merge the two trees by making the root of one tree a child of the root of the other. This operation clearly takes constant time. Figures 24.13–24.15 represent the forest after each of `union`(4, 5), `union`(6, 7), and `union`(4, 6), where we have adopted the convention that the new root after `union`(*x*, *y*) is *x*.

The cost of a `find` depends on the depth of the accessed node and could be linear.

A `find` operation on element *x* is performed by returning the root of the tree containing *x*. The time for performing this operation is proportional to the number of nodes on the path from *x* to the root. The `union` strategy outlined previously enables us to create a tree whose every node is on the path to *x*, resulting in a worst-case running time of $\Theta(N)$ per `find`. Typically (as shown in the preceding

figure 24.12
A forest and its eight elements, initially in different sets

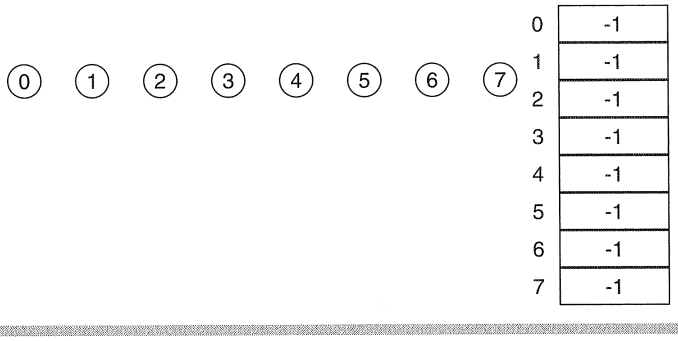
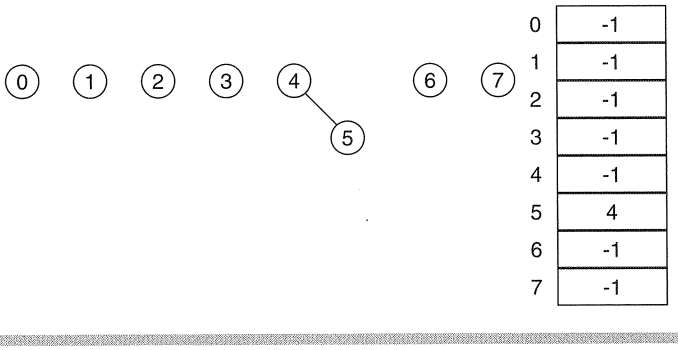


figure 24.13
The forest after the `union` of trees with roots 4 and 5



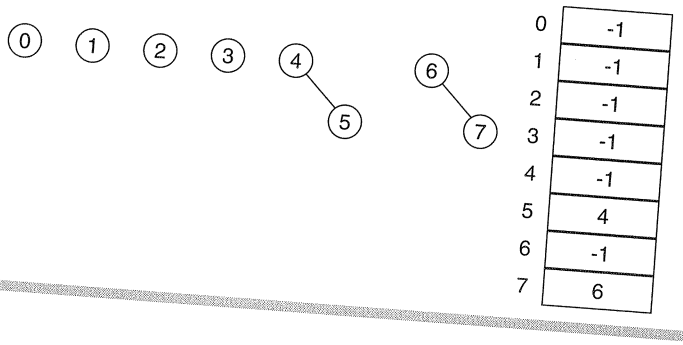


figure 24.14
The forest after the `union` of trees with roots 6 and 7

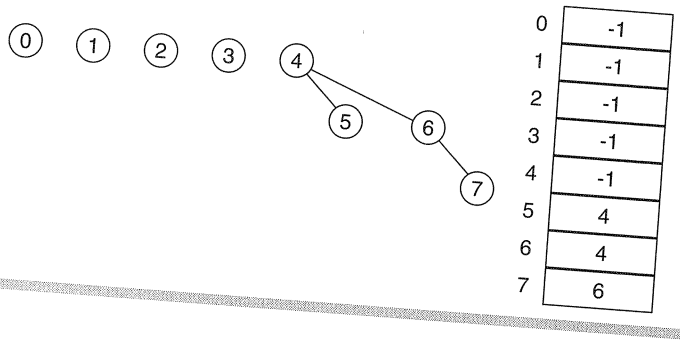


figure 24.15
The forest after the `union` of trees with roots 4 and 6

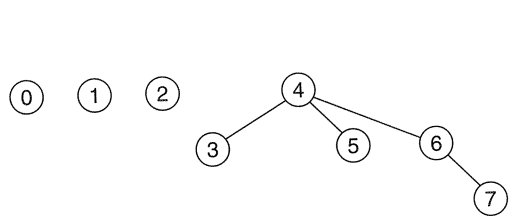
applications), the running time is computed for a sequence of M intermixed instructions. In the worst case, M consecutive operations could take $\Theta(MN)$ time.

Quadratic running time for a sequence of operations is generally unacceptable. Fortunately, there are several ways to easily ensure that this running time does not occur.

24.4.1 smart union algorithms

We performed the previous `unions` rather arbitrarily by making the second tree a subtree of the first. A simple improvement is always to make the smaller tree a subtree of the larger, breaking ties by any method, an approach called *union-by-size*. The preceding three `union` operations were all ties, so we can consider that they were performed by size. If the next operation is `union`(3, 4), the forest shown in Figure 24.16 forms. Had the size heuristic not been used, a deeper forest would have been formed (three nodes rather than one would have been one level deeper).

figure 24.16
The forest formed by union-by-size, with the sizes encoded as negative numbers



0	-1
1	-1
2	-1
3	4
4	-5
5	4
6	4
7	6

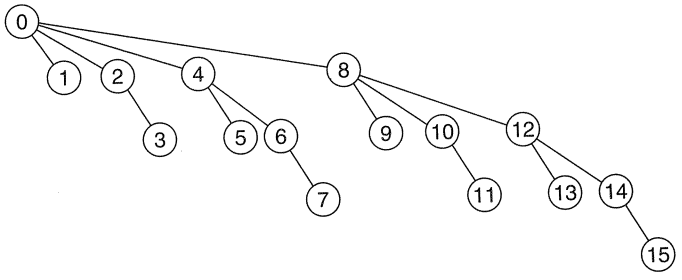
Union-by-size guarantees logarithmic finds.

If the `union` operation is done by size, the depth of any node is never more than log N. A node is initially at depth 0, and when its depth increases as a result of a `union`, it is placed in a tree that is at least twice as large as before. Thus its depth can be increased at most log N times. (We used this argument in the quick-find algorithm in Section 24.3.) This outcome implies that the running time for a `find` operation is $O(\log N)$ and that a sequence of M operations takes at most $O(M \log N)$ time. The tree shown in Figure 24.17 illustrates the worst tree possible after 15 `union` operations and is obtained if all the `unions` are between trees of equal size. (The worst-case tree is called a *binomial tree*. Binomial trees have other applications in advanced data structures.)

Instead of −1 being stored for roots, the negative of the size is stored.

To implement this strategy, we need to keep track of the size of each tree. Since we are just using an array, we can have the array entry of the root contain the *negative* of the size of the tree, as shown in Figure 24.16. Thus the initial representation of the tree with all −1s is reasonable. When a `union` operation is performed, we check the sizes; the new size is the sum of the old. Thus union-by-size is not at all difficult to implement and requires no extra space. It is also fast on average because, when random `union` operations are

figure 24.17
Worst-case tree for $N = 16$



performed, generally very small (usually one-element) sets are merged with large sets throughout the algorithm. Mathematical analysis of this process is quite complex; the references at the end of the chapter provide some pointers to the literature.

An alternative implementation that also guarantees logarithmic depth is *union-by-height* in which we keep track of the height of the trees instead of the size and perform `union` operations by making a shallower tree a subtree of the deeper tree. This algorithm is easy to write and use because the height of a tree increases only when two equally deep trees are joined (and then the height goes up by 1). Thus union-by-height is a trivial modification of union-by-size. As heights start at 0, we store the negative of the number of nodes rather than the height on the deepest path, as shown in Figure 24.18.

Union-by-height also guarantees logarithmic `find` operations.

24.4.2 path compression

The union/find algorithm, as described so far, is quite acceptable for most cases. It is very simple and linear on average for a sequence of M instructions. However, the worst case is still unappealing. The reason is that a sequence of `union` operations occurring in some particular application (such as the NCA problem) is not obviously random (in fact, for certain trees, it is far from random). Hence we have to seek a better bound for the worst case of a sequence of M operations. Seemingly, no more improvements to the union algorithm are possible because the worst case is achievable when identical trees are merged. The only way to speed up the algorithm then, without reworking the data structure entirely, is to do something clever with the `find` operation.

That something clever is *path compression*. Clearly, after we perform a `find` on x, changing x's parent to the root would make sense. In that way, a second `find` on x or any item in x's subtree becomes easier. There is no need to stop there, however. We might as well change the parents for all the nodes on the access path. In path compression *every* node on the path from x to the root

Path compression makes every accessed node a child of the root until another `union` occurs.

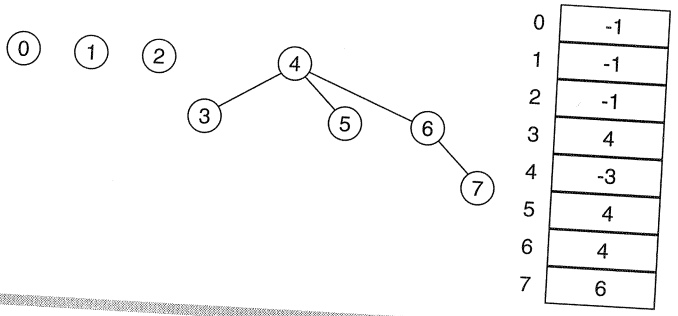


figure 24.18
A forest formed by union-by-height, with the height encoded as a negative number

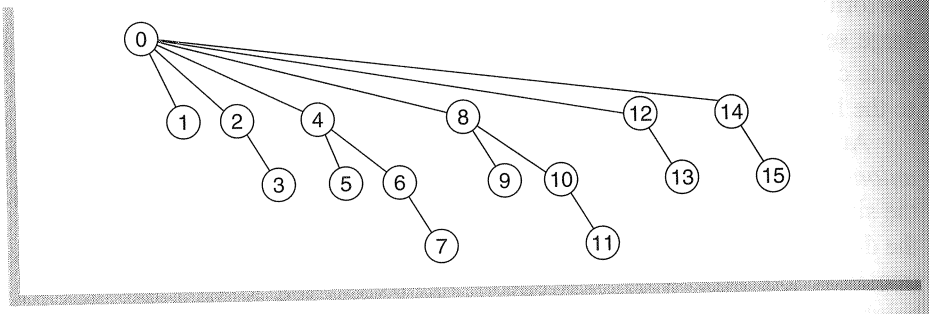


figure 24.19
Path compression resulting from a `find(14)` on the tree shown in Figure 24.17

has its parent changed to the root. Figure 24.19 shows the effect of path compression after `find(14)` on the generic worst tree shown in Figure 24.17. With an extra two parent changes, nodes 12 and 13 are now one position closer to the root and nodes 14 and 15 are now two positions closer. The fast future accesses on the nodes pay (we hope) for the extra work to do the path compression. Note that subsequent `unions` push the nodes deeper.

Path compression guarantees logarithmic amortized cost for the `find` operation.

When `unions` are done arbitrarily, path compression is a good idea because of the abundance of deep nodes; they are brought near the root by path compression. It has been proved that when path compression is done in this case, a sequence of M operations requires at most $O(M \log N)$ time, so path compression by itself guarantees logarithmic amortized cost for the `find` operation.

Path compression and a smart union rule guarantee essentially constant amortized cost per operation (i.e., a long sequence can be executed in almost linear time).

Path compression is perfectly compatible with union-by-size. Thus both routines can be implemented at the same time. However, path compression is not entirely compatible with union-by-height because path compression can change the heights of the trees. We do not know how to recompute them efficiently, so we do not attempt to do so. Then the heights stored for each tree become estimated heights, called *ranks,* which is not a problem. The resulting algorithm, *union-by-rank,* is thus obtained from union-by-height when compression is performed. As we show in Section 24.6, the combination of a smart union rule and path compression gives an almost linear guarantee on the running time for a sequence of M operations.

24.5 java implementation

Disjoint sets are relatively simple to implement.

The class skeleton for a disjoint sets class is given in Figure 24.20, and the implementation is completed in Figure 24.21. The entire algorithm is amazingly short.

```
1  package weiss.nonstandard;
2
3  // DisjointSets class
4  //
5  // CONSTRUCTION: with int representing initial number of sets
6  //
7  // ******************PUBLIC OPERATIONS*********************
8  // void union( root1, root2 ) --> Merge two sets
9  // int find( x )              --> Return set containing x
10 // ******************ERRORS********************************
11 // Error checking or parameters is performed
12
13 public class DisjointSets
14 {
15     public DisjointSets( int numElements )
16       { /* Figure 24.21 */ }
17
18     public void union( int root1, int root2 )
19       { /* Figure 24.21 */ }
20
21     public int find( int x )
22       { /* Figure 24.21 */ }
23
24     private int [ ] s;
25
26
27     private void assertIsRoot( int root )
28     {
29         assertIsItem( root );
30         if( s[ root ] >= 0 )
31             throw new IllegalArgumentException( );
32     }
33
34     private void assertIsItem( int x )
35     {
36         if( x < 0 || x >= s.length )
37             throw new IllegalArgumentException( );
38     }
39 }
```

figure 24.20
The disjoint sets class skeleton

In our routine, `union` is performed on the roots of the trees. Sometimes the operation is implemented by passing any two elements and having `union` perform the `find` operation to determine the roots.

The interesting procedure is `find`. After the `find` has been performed recursively, `array[x]` is set to the root and then is returned. Because this procedure is recursive, all nodes on the path have their entries set to the root.

figure 24.21
Implementation of a disjoint sets class

```
1     /**
2      * Construct the disjoint sets object.
3      * @param numElements the initial number of disjoint sets.
4      */
5     public DisjointSets( int numElements )
6     {
7         s = new int[ numElements ];
8         for( int i = 0; i < s.length; i++ )
9             s[ i ] = -1;
10    }
11
12    /**
13     * Union two disjoint sets using the height heuristic.
14     * root1 and root2 are distinct and represent set names.
15     * @param root1 the root of set 1.
16     * @param root2 the root of set 2.
17     * @throws IllegalArgumentException if root1 or root2
18     * are not distinct roots.
19     */
20    public void union( int root1, int root2 )
21    {
22        assertIsRoot( root1 );
23        assertIsRoot( root2 );
24        if( root1 == root2 )
25            throw new IllegalArgumentException( );
26
27        if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
28            s[ root1 ] = root2;        // Make root2 new root
29        else
30        {
31            if( s[ root1 ] == s[ root2 ] )
32                s[ root1 ]--;          // Update height if same
33            s[ root2 ] = root1;        // Make root1 new root
34        }
35    }
36
37    /**
38     * Perform a find with path compression.
39     * @param x the element being searched for.
40     * @return the set containing x.
41     * @throws IllegalArgumentException if x is not valid.
42     */
43    public int find( int x )
44    {
45        assertIsItem( x );
46        if( s[ x ] < 0 )
47            return x;
48        else
49            return s[ x ] = find( s[ x ] );
50    }
```