THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS

Alfred V. Aho
Bell Laboratories

John E. Hopcroft
Cornell University

Jeffrey D. Ullman
Princeton University

Addison-Wesley Publishing Company
Reading, Massachusetts · Menlo Park, California
London · Amsterdam · Don Mills, Ontario · Sydney

4.6 A SIMPLE DISJOINT-SET UNION ALGORITHM

Consider the handling of vertices in the spanning tree algorithm of Example 4.1. The set-processing problem that arose had the following three characteristics.

1. Whenever two sets were merged, the two sets were disjoint.
2. The elements of the sets could be treated as integers in the range 1 through n.
3. The operations were UNION and FIND.

In this section and the next we shall consider data structures for problems of this nature. Assume we are given n elements, which we shall take to be the integers $1, 2, \ldots, n$. Initially, each element is assumed to be in a set by itself. A sequence of UNION and FIND instructions is to be executed. A UNION instruction, we recall, is of the form UNION(A, B, C), indicating that two disjoint sets named A and B are to be replaced by their union, and their union is to be named C. In applications it is often unimportant what we choose to name a set, so we shall assume that sets can be named by integers in the range 1 to n. Moreover, we shall assume no two sets are ever given the same name.

There are several interesting data structures to handle this problem. In this section we shall present a data structure that is capable of processing a sequence containing up to $n - 1$ UNION instructions and $O(n \log n)$ FIND instructions in time $O(n \log n)$. In the next section we shall describe a data structure that will handle a sequence of $O(n)$ UNION and FIND instructions with a worst-case time that is almost linear in n. These data structures are also capable of handling sequences of INSERT, DELETE, and MEMBER instructions with the same computational complexity.

Note that the searching algorithms we considered in Sections 4.2–4.5 assumed that the elements were drawn from a universal set that was much larger than the number of instructions to be executed. In this section we are assuming that the universal set is approximately the same size as the length of the sequence of instructions to be executed.

Perhaps the simplest data structure for the UNION–FIND problem is to use an array to represent the collection of sets present at any one time. Let R be an array of size n such that $R[i]$ is the name of the set containing element i. Since the names of sets are unimportant, we may initially take $R[i] = i$, $1 \leq i \leq n$, to denote the fact that at the start the collection of sets is $\{\{1\}, \{2\}, \ldots, \{n\}\}$ and set $\{i\}$ is named i.

The instruction FIND(i) is executed by printing the current value of $R[i]$. Thus the cost of executing a FIND instruction is a constant, which is the best we could hope for.

To execute the instruction UNION(A, B, C), we sequentially scan the array R, and each entry containing either A or B is set to C. Thus the cost of executing a UNION instruction is $O(n)$. A sequence of n UNION instructions could require $O(n^2)$ time, which is undesirable.

This simple-minded algorithm can be improved in several ways. One improvement would take advantage of linked lists. Another would recognize that it is more efficient always to merge a smaller set into a larger one. To do so, we need to distinguish between "internal names," which are used to identify sets in the R array, and "external names," the ones mentioned in the UNION instructions. Both are presumably integers between 1 and n, but not necessarily the same.

Let us consider the following data structure for this problem. As before, we use an array R such that $R[i]$ contains the "internal" name of the set containing the element i. But now, for each set A we construct a linked list LIST[A] containing the elements of the set. Two arrays, LIST and NEXT, are used to implement this linked list. LIST[A] is an integer j indicating that j is the first element in the set whose internal name is A. NEXT[j] gives the next element in A, NEXT[NEXT[j]] the next element, and so on.

In addition, we shall use an array called SIZE, where SIZE[A] is the number of elements in set A. Also, sets are renamed internally. Two arrays INTERNAL_NAME and EXTERNAL_NAME associate internal and external names. That is, EXTERNAL_NAME[A] gives the real name (the name dictated by the UNION instructions) of the set with internal name A. INTERNAL_NAME[j] is the internal name of the set with external name j. The internal names are the ones used in the array R.

Example 4.6. Suppose $n = 8$ and we have the collection of three sets {1, 3, 5, 7}, {2, 4, 8}, and {6} with external names 1, 2, and 3, respectively. The data structures for these three sets are shown in Fig. 4.13, where we assume 1, 2, and 3 have the internal names 2, 3, and 1, respectively. □

The instruction FIND(i) is executed as before, by consulting $R[i]$ to determine the internal name of the set currently containing element i. Then EXTERNAL_NAME[$R[i]$] gives the real name of the set containing i.

A union instruction of the form UNION(I, J, K) is executed as follows. (The line numbers refer to Fig. 4.14.)

1. We determine the internal names for sets I and J (lines 1–2).
2. We compare the relative sizes of sets I and J by consulting the array SIZE (lines 3–4).
3. We traverse the list of elements of the smaller set and change the corresponding entries in the array R to the internal name of the larger set (lines 5–9).

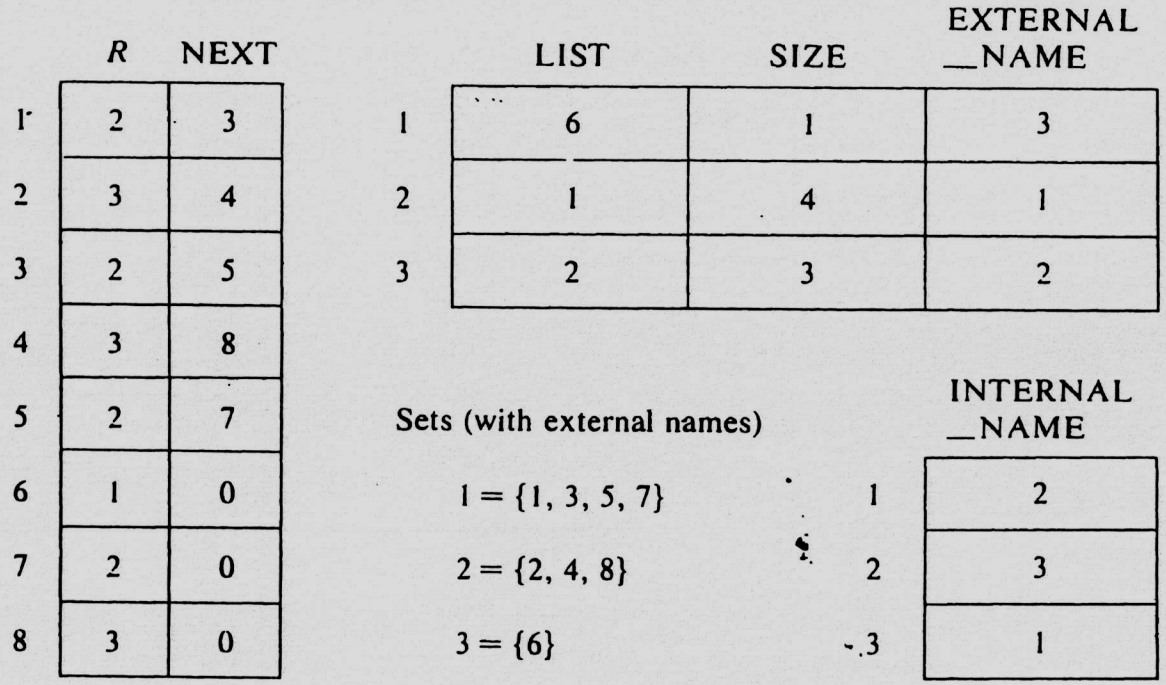

Fig. 4.13. Data structures for UNION–FIND algorithm.

```
        procedure UNION(I, J, K):
        begin
1.          A ← INTERNAL_NAME[I];
2.          B ← INTERNAL_NAME[J];
3.          wlg assume SIZE[A] ≤ SIZE[B]
4.            otherwise interchange roles of A and B in
                begin
5.                  ELEMENT ← LIST[A];
6.                  while ELEMENT ≠ 0 do
                        begin
7.                          R[ELEMENT] ← B;
8.                          LAST ← ELEMENT;
9.                          ELEMENT ← NEXT[ELEMENT]
                        end;
10.                 NEXT[LAST] ← LIST[B];
11.                 LIST[B] ← LIST[A];
12.                 SIZE[B] ← SIZE[A] + SIZE[B];
13.                 INTERNAL_NAME[K] ← B;
14.                 EXTERNAL_NAME[B] ← K
                end
        end
```

Fig. 4.14. Implementation of UNION instruction.

4. We merge the smaller set into the larger by appending the list of elements of the smaller set to the beginning of the list for the larger set (lines 10–12).
5. We give the combined set the external name K (lines 13–14).

By merging the smaller set into the larger, the UNION instruction can be executed in time proportional to the cardinality of the smaller set. The complete details are given in the procedure in Fig. 4.14.

Example 4.7. After execution of the instruction UNION(1, 2, 4), the data structures in Fig. 4.13 would become as shown in Fig. 4.15. □

Theorem 4.3. Using the algorithm of Fig. 4.14 we can execute $n - 1$ UNION operations (the maximum possible number) in $O(n \log n)$ steps.

Proof. Since the cost of each UNION is proportional to the number of elements moved, apportioning the cost of each UNION instruction uniformly among the elements moved results in a fixed charge to an element each time it is moved. The key observation is that each time an element is moved from a list, it finds itself on a list which is at least twice as long as before. Thus no element can be moved more than $\log n$ times and hence the total cost charged

	R	NEXT
1	2	3
2	2	4
3	2	5
4	2	8
5	2	7
6	1	0
7	2	0
8	2	1

	LIST	SIZE	EXTERNAL _NAME
1	6	1	3
2	2	7	4
3	–	–	–
4	–	–	–

Sets (with external names)

$3 = \{6\}$

$4 = \{1, 2, 3, 4, 5, 7, 8\}$

	INTERNAL _NAME
1	–
2	–
3	1
4	2

Fig. 4.15. Data structures after UNION instruction.

to any element is $O(\log n)$. The total cost is obtained by summing the costs charged to the elements. Thus the total cost is $O(n \log n)$. □

It follows from Theorem 4.3 that if m FIND and up to $n - 1$ UNION instructions are executed, then the total time spent is $O(\text{MAX}(m, n \log n))$. If m is on the order of $n \log n$ or greater, then this algorithm is actually optimal to within a constant factor. However, in many situations we shall find that m is $O(n)$, and in this case, we can do better than $O(\text{MAX}(m, n \log n))$, as we shall see in the next section.

4.7 TREE STRUCTURES FOR THE UNION–FIND PROBLEM

In the last section we presented a data structure for the UNION–FIND problem that would allow the processing of $n - 1$ UNION instructions and $O(n \log n)$ FIND instructions in time $O(n \log n)$. In this section we shall present a data structure consisting of a forest of trees to represent the collection of sets. This data structure will allow the processing of $O(n)$ UNION and FIND instructions in almost linear time.

Suppose we represent each set A by a rooted undirected tree T_A, where the elements of A correspond to the vertices of T_A. The name of the set is attached to the root of the tree. An instruction of the form UNION(A, B, C) can be executed by making the root of T_A a son of the root of T_B and changing the name at the root of T_B to C. An instruction of the form FIND(i) can be executed by locating the vertex representing element i in some tree T in the forest, and traversing the path from this vertex to the root of T, where we find the name of the set containing i.

With such a scheme, the cost of merging two trees is a constant. However, the cost of a FIND(i) instruction is on the order of the length of the path from vertex i to its root. This path could have length $n - 1$. Thus the cost of executing $n - 1$ UNION instructions followed by n FIND instructions could be as high as $O(n^2)$. For example, consider the cost of the following sequence:

UNION(1, 2, 2)
UNION(2, 3, 3)
.
.
.
UNION($n - 1$, n, n)
FIND(1)
FIND(2)
.
.
.
FIND(n)

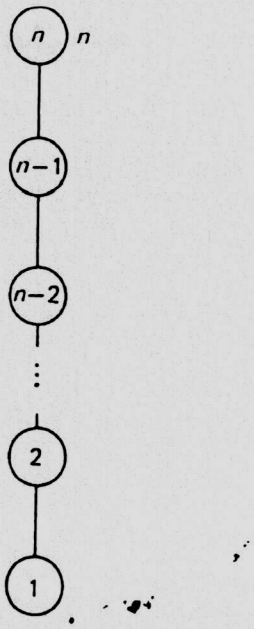

Fig. 4.16 Tree after UNION instructions.

The $n-1$ UNION instructions result in the tree shown in Fig. 4.16. The cost of the n FIND instructions is proportional to

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

However, the cost can be reduced if the trees can be kept balanced. One way to accomplish this is to keep count of the number of vertices in each tree and, when merging two sets, always to attach the smaller tree to the root of the larger. This technique is analogous to the technique of merging smaller sets into larger, which we used in the last section.

Lemma 4.1. If in executing each UNION instruction the root of the tree with fewer vertices (ties are broken arbitrarily) is made a son of the root of the larger, then no tree in the forest will have height greater than or equal to h unless it has at least 2^h vertices.

Proof. The proof is by induction on h. For $h = 0$, the hypothesis is true since every tree has at least one vertex. Assume the induction hypothesis true for all values less than $h \geq 1$. Let T be a tree of height h with fewest vertices. Then T must have been obtained by merging two trees T_1 and T_2, where T_1 has height $h-1$ and T_1 has no more vertices than T_2. By the induction hypothesis T_1 has at least 2^{h-1} vertices and hence T_2 has at least 2^{h-1} vertices, implying that T has at least 2^h vertices. □

Consider the worst-case execution time for a sequence of n UNION and FIND instructions using the forest data structure, with the modification that

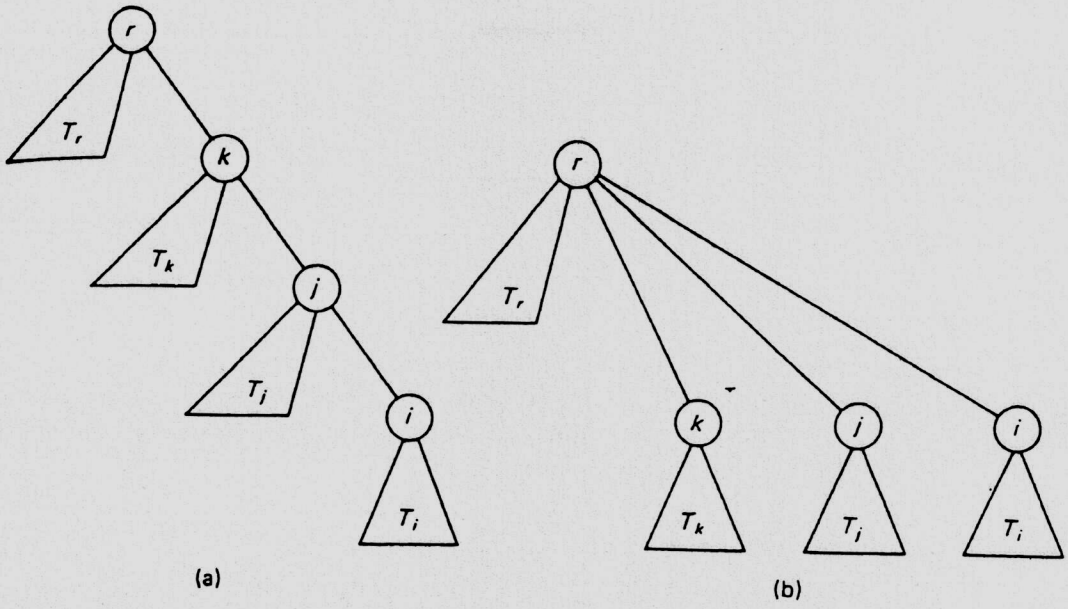

Fig. 4.17 Effect of path compression.

in a UNION the root of the smaller tree becomes a son of the root of the larger tree. No tree can have height greater than log n. Hence the execution of $O(n)$ UNION and FIND instructions costs at most $O(n \log n)$ units of time. This bound is tight, in that there are sequences of n instructions that will take time proportional to $n \log n$.

We now introduce another modification to this algorithm, called *path compression*. Since the cost of the FIND's appears to dominate the total cost, we shall try to reduce the cost of the FIND's. Each time a FIND(i) instruction is executed we traverse the path from vertex i to its root r. Let i, $v_1, v_2, \ldots, v_n, r$ be the vertices on this path. We then make each of $i, v_1, v_2, \ldots, v_{n-1}$ a son of the root. Figure 4.17(b) illustrates the effect of the instruction FIND(i) on the tree of Fig. 4.17(a).

The complete tree-merging algorithm for the UNION–FIND problem, including path compression, is expressed by the following algorithm.

Algorithm 4.3. Fast disjoint-set union algorithm.

Input. A sequence σ of UNION and FIND instructions on a collection of sets whose elements consist of integers from 1 through n. The set names are also assumed to be integers from 1 to n, and initially, element i is by itself in a set named i.

Output. The sequence of responses to the FIND instructions in σ. The response to each FIND instruction is to be produced before looking at the next instruction in σ.

Method. We describe the algorithm in three parts—the initialization, the response to a FIND, and the response to a UNION.

1. *Initialization.* For each element i, $1 \le i \le n$, we create a vertex v_i. We set $\text{COUNT}[v_i] = 1$, $\text{NAME}[v_i] = i$, and $\text{FATHER}[v_i] = 0$. Initially, each vertex v_i is a tree by itself. In order to locate the root of set i, we create an array ROOT with $\text{ROOT}[i]$ pointing to v_i. To locate the vertex for element i, we create an array ELEMENT, initially with $\text{ELEMENT}[i] = v_i$.
2. *Executing* FIND(i). The program is shown in Fig. 4.18. Starting at vertex $\text{ELEMENT}[i]$ we follow the path to the root of the tree, making

```
begin
    make LIST empty;
    v ← ELEMENT[i];
    while FATHER[v] ≠ 0 do
        begin
            add v to LIST;
            v ← FATHER[v]
        end;
    comment v is now the root;
    print NAME[v];
    for each w on LIST do FATHER[w] ← v
end
```

Fig. 4.18. Executing instruction FIND(i).

```
begin
  wlg assume COUNT[ROOT[i]] ≤ COUNT[ROOT[j]]
    otherwise interchange i and j in
    begin
        LARGE ← ROOT[j];
        SMALL ← ROOT[i];
        FATHER[SMALL] ← LARGE;
        COUNT[LARGE] ← COUNT[LARGE] + COUNT[SMALL];
        NAME[LARGE] ← k;
        ROOT[k] ← LARGE
    end
end
```

Fig. 4.19. Executing instruction UNION(i, j, k).

a list of vertices encountered. At the root, the name of the set is printed, and each vertex on the path traversed is made a son of the root.

3. *Executing* UNION(i, j, k). Via the array ROOT, we find the roots of the trees representing sets i and j. We then make the root of the smaller tree a son of the root of the larger. See Fig. 4.19. □

We shall show that path compression speeds up the algorithm considerably. To calculate the improvement we introduce two functions F and G. Let

$$F(0) = 1,$$

$$F(i) = 2^{F(i-1)}, \qquad \text{for } i > 0.$$

The function F grows extremely fast, as the table in Fig. 4.20 shows. The function $G(n)$ is defined to be smallest integer k such that $F(k) \geq n$. The function G grows extremely slowly. In fact, $G(n) \leq 5$ for all "practical" values of n, i.e., for all $n \leq 2^{65536}$.

We shall now prove that Algorithm 4.3 will execute a sequence σ of cn UNION and FIND instructions in at most $c'nG(n)$ time, where c and c' are constants, c' depending on c. For simplicity, we assume the execution of a UNION instruction takes one "time unit" and the execution of the instruction FIND(i) takes a number of time units proportional to the number of vertices on the path from the vertex labeled i to the root of the tree containing this vertex.†

n	$F(n)$
0	1
1	2
2	4
3	16
4	65536
5	2^{65536}

Fig. 4.20. Some values of F.

† Thus one "time unit" in the sense used here requires some constant number of steps on a RAM. Since we neglect constant factors, order-of-magnitude results can as well be expressed in terms of "time units."

Definition. It is convenient to define the *rank* of a vertex with respect to the sequence σ of UNION and FIND instructions as follows:

1. Delete the FIND instructions from σ.
2. Execute the resulting sequence σ' of UNION instructions.
3. The rank of a vertex v is the height of v in the resulting forest.

We shall now derive some important properties of the rank of a vertex.

Lemma 4.2. There are at most $n/2^r$ vertices of rank r.

Proof. By Lemma 4.1 each vertex of rank r has at least 2^r descendants in the forest which results from executing σ'. Since the sets of descendants of any two distinct vertices of the same height in a forest are disjoint and since there are at most $n/2^r$ disjoint sets of 2^r or more vertices, there can be at most $n/2^r$ vertices of rank r. □

Corollary. No vertex has rank greater than $\log n$.

Lemma 4.3. If at some time during the execution of σ, w is a proper descendant of v, then the rank of w is less than the rank of v.

Proof. If at some time during the execution of σ, w is made a descendant of v, then w will be a descendant of v in the forest resulting from the execution of the sequence σ'. Thus the height of w must be less than the height of v, so the rank of w is less than the rank of v. □

We now partition the ranks into *groups*. We put rank r in group $G(r)$. For example, ranks 0 and 1 are in group 0, rank 2 is in group 1, ranks 3 and 4 are in group 2, ranks 5 through 16 are in group 3. For $n > 1$, the largest possible rank, $\lfloor \log n \rfloor$, is in rank group $G(\lfloor \log n \rfloor) \leq G(n) - 1$.

Consider the cost of executing a sequence σ of cn UNION and FIND instructions. Since each UNION instruction can be executed at the cost of one time unit, all UNION instructions in σ can be executed in $O(n)$ time. In order to bound the cost of executing all FIND instructions we use an important "bookkeeping" trick. The cost of executing a single FIND is apportioned between the FIND instruction itself and certain vertices on the path in the forest data structure which are actually moved. The total cost is computed by summing over all FIND instructions the cost apportioned to them, and then summing the cost assigned to the vertices, over all vertices in the forest.

We charge for the instruction FIND(i) as follows. Let v be a vertex on the path from the vertex representing i to the root of tree containing i.

1. If v is the root, or if FATHER[v] is in a different rank group from v, then charge one time unit to the FIND instruction itself.
2. If both v and its father are in the same rank group, then charge one time unit to v.

By Lemma 4.3 the vertices going up a path are monotonically increasing in rank, and since there are at most $G(n)$ different rank groups, no FIND instruction is charged more than $G(n)$ time units under rule 1. If rule 2 applies, vertex v will be moved and made the son of a vertex of higher rank than its previous father. If vertex v is in rank group $g > 0$, then v can be moved and charged at most $F(g) - F(g-1)$ times before it acquires a father in a higher rank group. In rank group 0, a vertex can be moved at most once before obtaining a father in a higher group. From then on, the cost of moving v will be charged to the FIND instructions by rule 1.

To obtain an upper bound on the charges made to the vertices themselves, we multiply the maximum possible charge to any vertex in a rank group by the number of vertices in that rank group, and sum over all rank groups. Let $N(g)$ be the number of vertices in rank group $g > 0$. Then by Lemma 4.2:

$$\begin{aligned} N(g) &\le \sum_{r=F(g-1)+1}^{F(g)} n/2^r \\ &\le (n/2^{F(g-1)+1})[1 + \tfrac{1}{2} + \tfrac{1}{4} + \cdots] \\ &\le n/2^{F(g-1)} \\ &\le n/F(g). \end{aligned}$$

The maximum charge to any vertex in rank group $g > 0$ is less than or equal to $F(g) - F(g-1)$. Thus the maximum charge to all vertices in rank group g is bounded by n. The same statement clearly applies for $g = 0$ as well. Since there are at most $G(n)$ rank groups, the maximum charge to all vertices is $nG(n)$. Therefore, the total amount of time required to process cn FIND instructions is at most $cnG(n)$ charged to the FIND's and at most $nG(n)$ charged to the vertices. Thus we have the following theorem.

Theorem 4.4. Let c be any constant. Then there exists another constant c' depending on c such that Algorithm 4.3 will execute a sequence σ of cn UNION and FIND instructions on n elements in at most $c'nG(n)$ time units.

Proof. By the above discussion. □

It is left as an exercise to show that if the primitive operations INSERT and DELETE, as well as UNION and FIND, are permitted in the sequence σ, then σ can still be executed in $O(nG(n))$ time.

It is not known whether Theorem 4.4 provides a tight bound on the running time of Algorithm 4.3. However, as a matter of theoretical interest, in the remainder of this section we shall prove that the running time of Algorithm 4.3 is not linear in n. To do this, we shall construct a particular sequence of UNION and FIND instructions, which Algorithm 4.3 takes more than linear time to process.

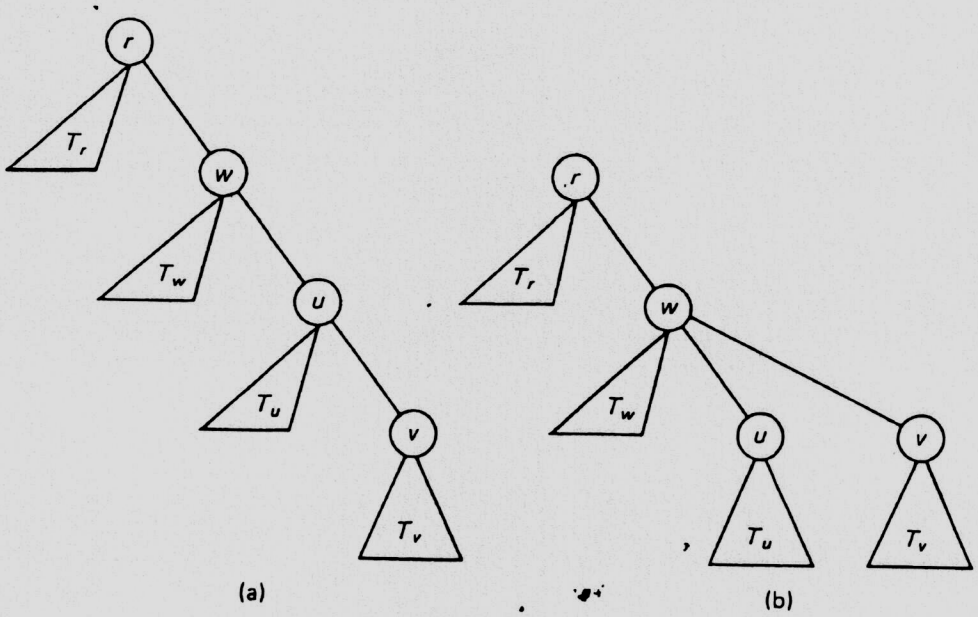

Fig. 4.21 Effect of partial FIND operation.

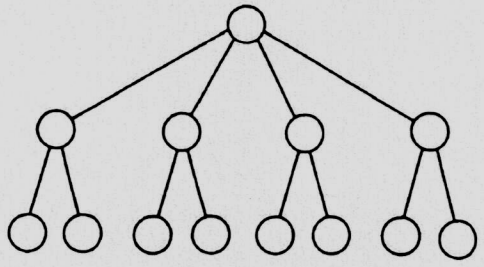

Fig. 4.22 The tree $T(2)$.

It is convenient to introduce a new operation on trees which we shall call *partial FIND*, or PF for short. Let T be a tree in which $v, v_1, v_2, \ldots, v_m, w$ is a path from a vertex v to an ancestor w. (w is not necessarily the root.) The operation PF(v, w) makes each of $v, v_1, v_2, \ldots, v_{m-1}$ sons of vertex w. We say this partial FIND is of *length* $m + 1$ (if $v = w$, the length is 0). Figure 4.21(b) illustrates the effect of PF(v, w) on the tree of Fig. 4.21(a).

Suppose we are given a sequence σ of UNION and FIND instructions. When we execute a given FIND instruction in σ we locate a vertex v in some tree T and follow the path from v to the root w of T. Now suppose we execute only the UNION instructions in σ, ignoring the FIND's. This will result in a forest F of trees. We can still capture the effect of a given FIND instruction in σ by locating in F the vertices v and w used by the original FIND instruction and then executing PF(v, w). Note that the vertex w may no longer be a root in F.

In deriving a lower bound on the running time of Algorithm 4.3, we consider the behavior of the algorithm on a sequence of UNION's followed by PF's which can be replaced by a sequence of UNION's and FIND's whose execution time is the same. From the following special trees we shall derive particular sequences of UNION's and PF's on which Algorithm 4.3 takes more than linear time.

Definition. For $k \geq 0$, let $T(k)$ be the tree such that

1. each leaf of $T(k)$ has depth k.
2. each vertex of height h has 2^h sons, $h \geq 1$.

Thus the root of $T(k)$ has 2^k sons, each of which is a root of a copy of $T(k-1)$. Figure 4.22 shows $T(2)$.

Lemma 4.4. With a sequence of UNION instructions we can create, for any $k \geq 0$, a tree $T'(k)$ that contains as a subgraph the tree $T(k)$. Furthermore, at least one-quarter of the vertices in $T'(k)$ are leaves of $T(k)$.

Proof. The proof proceeds by induction on k. The lemma is trivial for $k = 0$, since $T(0)$ consists of a single vertex. To construct $T'(k)$ for $k > 0$, first construct $2^k + 1$ copies of $T'(k-1)$. Form the tree $T'(k)$ by selecting one copy of $T'(k-1)$ and then merging into it, one by one, each of the remaining copies. The root of the resulting tree has (among others) 2^k sons, each of which is a root of $T'(k-1)$.

Let $N'(k)$ be the total number of vertices in $T'(k)$ and let $L(k)$ be the number of leaves in $T(k)$. Then

$$N'(0) = 1$$
$$N'(k) = (2^k + 1)N'(k-1), \qquad \text{for } k \geq 1,$$

and

$$L(0) = 1$$
$$L(k) = 2^k L(k-1), \qquad \text{for } k \geq 1;$$

so

$$\frac{L(k)}{N'(k)} = \frac{\prod_{i=1}^{k} 2^i}{\prod_{i=1}^{k} (2^i + 1)} = \frac{2}{3} \prod_{i=2}^{k} \frac{1}{1 + 2^{-i}}, \qquad \text{for } k \geq 1. \tag{4.3}$$

We note that for $i \geq 2$, $\log_e (1 + 2^{-i}) < 2^{-i}$, so

$$\log_e \left(\prod_{i=2}^{k} \frac{1}{1 + 2^{-i}} \right) \geq -\sum_{i=2}^{k} 2^{-i} \geq -\tfrac{1}{2}. \tag{4.4}$$

Using (4.3) and (4.4) together we have

$$\frac{L(k)}{N'(k)} \geq \tfrac{2}{3} e^{-1/2} \geq \tfrac{1}{4},$$

thus proving the lemma. □

We shall construct a sequence of UNION and PF instructions that will first build the tree $T'(k)$ and then perform PF's on the leaves of the subgraph $T(k)$. We shall now show that for every $l > 0$, there exists a k such that we can perform a PF of length l in succession on every leaf of $T(k)$.

Definition. Let $D(c, l, h)$ be the smallest value of k such that if we replace every subtree in $T(k)$ whose root has height h by any tree having l leaves and height at least 1, then we may perform a PF of length c on each leaf in the resulting tree.

Lemma 4.5. $D(c, l, h)$ is defined (i.e., finite) for all c, l, and h greater than zero.

Proof. The proof involves a double induction. We wish to prove the result by induction on c. But in order to prove the result for c given the result for $c - 1$, we must also do an induction on l.

The basis, $c = 1$, is easy. $D(1, l, h) = h$ for all l and h, since a PF of length 1 does not move any vertices.

Now for the induction on c, suppose that for all l and h, $D(c - 1, l, h)$ is defined. We must show that $D(c, l, h)$ is defined for all l and h. This is done by induction on l.

For the basis of this induction, we show

$$D(c, 1, h) \leq D(c - 1, 2^{h+1}, h + 1).$$

Note that when $l = 1$, we have substituted trees with a single leaf for subtrees with roots at the vertices of height h in $T(k)$ for some k. Let H be the set of vertices of height h in this $T(k)$. Clearly, in the modified tree each leaf is the proper descendant of a unique member of H. Therefore, if we could do PF's of length $c - 1$ on all the members of H, we could certainly do PF's of length c on all the leaves.

Let $k = D(c - 1, 2^{h+1}, h + 1)$. By the hypothesis for the induction on c, we know that k exists. If we consider the vertices of height $h + 1$ in $T(k)$, we see that each has 2^{h+1} sons, all of which are members of H. If we delete all proper descendants of the vertices in H from $T(k)$, we have in effect substituted trees of height 1 with 2^{h+1} leaves for each subtree having roots at height $h + 1$. By the definition of D, $k = D(c - 1, 2^{h+1}, h + 1)$ is sufficiently large so that PF's of length $c - 1$ can be done on all its leaves, i.e., the members of H.

Now, to complete the induction on c, we must do the inductive step for l. In particular, we shall show:

$$D(c, l, h) \leq D(c - 1, 2^{D(c,l-1,h)(1+D(c,l-1,h))/2}, D(c, l - 1, h)) \qquad \text{for } l > 1. \tag{4.5}$$

To prove (4.5), let $k = D(c, l - 1, h)$ and let k' be the right side of (4.5). We must find a way to substitute a tree of l leaves for each vertex of height h in