Worst-Case Analysis of Set Union Algorithms

ROBERT E. TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey

AND

JAN VAN LEEUWEN

University of Utrecht, Utrecht, The Netherlands

Abstract. This paper analyzes the asymptotic worst-case running time of a number of variants of the well-known method of path compression for maintaining a collection of disjoint sets under union. We show that two one-pass methods proposed by van Leeuwen and van der Weide are asymptotically optimal, whereas several other methods, including one proposed by Rem and advocated by Dijkstra, are slower than the best methods.

Categories and Subject Descriptors: E.1 [Data Structures]: Trees; F2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G2.1 [Discrete Mathematics]: Combinatorics—*combinatorial algorithms*; G2.2 [Discrete Mathematics]: Graph Theory—*graph algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Equivalence algorithm, set union, inverse Ackermann's function

1. *Introduction*

A well-known problem in data structures is the *set union problem*, defined as follows: Carry out a sequence of intermixed operations of the following three kinds on labeled sets:

make set(e, l): Create a new set with label l containing the single element e. This operation requires that e initially be in no set.

find label(e): Return the label of the set containing element e.

unite(e, f): Combine the sets containing elements e and f into a single set, whose label is the label of the old set containing element e. This operation requires that elements e and f initially be in different sets.

Because of the constraint on *make set*, the sets existing at any time are disjoint and define a partition of the elements into equivalence classes. For this reason the set union problem has been called the *equivalence problem* by some authors. A solution to the set union problem can be used in the compiling of FORTRAN

EQUIVALENCE statements [7] and in finding minimum spanning trees [2]. A generalization of the problem arises in the compiling of FORTRAN COMMON statements [2, 7] and in various graph problems [12].

All algorithms for the set union problem appearing in the literature can be regarded as versions of a general method that we shall call the *canonical element method.* Within each set, we distinguish an arbitrary but unique element called the *canonical element*, which serves to represent the set. We store the label of a set with its canonical element in a field called *label*. We carry out *find label* and *unite* using two lower level operations that manipulate canonical elements:

find(e): Return the canonical element of the set containing element *e*.

link(e, f): Combine the sets whose canonical elements are *e* and *f* into a single set, and make either *e* or *f* the canonical element of the new set. The label of the new set is the label of the old set containing element *e*. This operation requires that $e \neq f$.

The following procedures, written in a version of Dijkstra's guarded command language [4], implement *find label* and *unite*:

function *find label(e)*;
    return *label(find(e))*
end *find label*;

procedure *unite(e, f)*;
    link(find(e), find(f))
end *unite*;

To make finds possible, we represent each set by a rooted tree whose nodes are the elements of the set. The tree has an arbitrary structure except that the root is the canonical element. Each node x contains a pointer $p(x)$ to its parent in the tree; the root points to itself. This *compressed tree* representation (so-called because of the compression operation defined in Section 3) was invented by Galler and Fischer [7]. To carry out *find(e)*, we follow parent pointers from *e* until repeating a node; then we return the repeated node. To carry out *link(e, f)*, we make *f* point to *e*; *e* becomes the canonical element of the new set. The following procedures implement *make set, link,* and *find*:

procedure *make set(e, l)*;
    $p(e) := e$; *label(e)* := *l*
end *make set*;

procedure *link(e, f)*;
    $p(f) := e$
end *link*;

function *find(e)*;
    return if $p(e) = e \rightarrow e$
        [] $p(e) \neq e \rightarrow find(p(e))$
        fi
end *find*;

In analyzing this method (and its more sophisticated variants), we shall regard *make set, link,* and *find* as the fundamental operations. We shall denote by n the number of *make set* operations and by m the number of *find* operations. If k is the number of links, $k \leq n - 1$. We shall assume that $k \geq n/2$. This entails no loss of generality, since there are at most $2k$ elements that are ever in sets containing more than one element, and finds on elements in singleton sets require $O(1)$ time. In a sequence of *make set, link,* and *find* operations that arises from a sequence of

make set, unite, and *find label* operations, there are two finds per link, and $m \geq n$. However, our analysis will be valid for an arbitrary sequence of *make set, link,* and *find* operations, and, we shall not, in general, assume that $m \geq n$.

The naive version of the canonical element method spends most of its time following parent pointers. Each *make set* operation requires $O(1)$ time, as does each link. A find takes time proportional to the number of nodes on the find path, which is at most n. Thus the total time is $O(n + mn)$. The following class of examples shows that this bound is tight. By means of $n - 1$ links, we can build a tree that is a path of n nodes; if we then repeatedly perform a find on this path, we use a total of $\Omega(n + mn)$ time.

THEOREM 1 [5]. *The naive set union algorithm runs in $\Theta(n + mn)$ time in the worst case.*

By changing the structure of the trees to reduce the length of find paths, we can speed up the algorithm considerably. In this paper we analyze several variants of the canonical element method, with the aim of ascertaining which are both easy to implement and efficient in theory and in practice. In Section 2 we study two ways to implement the link operation, called *linking by size* and *linking by rank.* Both methods reduce the maximum length of a find path to $O(\log n)$. In Section 3 we study a way to improve subsequent finds by compressing each find path. In combination with either linking by rank or linking by size, compression gives an asymptotically optimal method (in the sense defined below). However, compression requires two passes over a find path. We discuss two one-pass variants of compression that also are asymptotically optimal. In Section 4 we study an appealing but inferior variant of compression called *reversal.* In Section 5 we study two ways to speed up the canonical element method by doing more work during link and unite operations. Section 6 contains some concluding remarks.

There is a general lower bound for the set union problem that applies to many versions of the canonical element method. The algorithms to which this bound applies are called the *separable algorithms.* Consider an arbitrary sequence of intermixed *make set, link,* and *find* operations. A separable algorithm begins with a list structure L representing the sets defined by the *make set* operations. L contains a distinct node representing each element and may contain an arbitrary number of auxiliary nodes. (We do not distinguish between an element and the node representing it.) Each node contains an arbitrary number of pointers to other nodes. The nodes are partitioned into *accessed nodes* and *unaccessed nodes;* this partition changes as the operations are performed. The algorithm has two kinds of steps:

(i) Follow a pointer $x \rightarrow y$ from an accessed node x to an unaccessed node y. This causes y to become accessed.
(ii) Put into an accessed node x a pointer to another accessed node y.

The algorithm carries out the set operations in the following way. The initiation of an operation *make set(e), find(e),* or *link(e, f)* causes e (and f in the case of *link(e, f)*) to become accessed. The algorithm performs an arbitrary sequence of steps, which in the case of *find(e)* must cause the canonical element of the set containing e to become accessed. The completion of an operation causes all nodes to become unaccessed.

We impose one more restriction on the algorithm, called *separability:* There must be a partition of the nodes of the initial list structure L into n parts, such that

each element is in a different part and no pointer leads from one part to another. Because the algorithm has no global memory (all nodes become unaccessed after each set operation), separability is preserved as the set operations are performed. More precisely, after each set operation the nodes can be partitioned into i parts, where i is the number of currently existing sets, so that each part contains the elements in one set, and no pointer leads from one part to another.

Any correct separable algorithm must perform at least one pointer construction step per link, since otherwise the canonical node of the new set is inaccessible from the nodes in one of the two old sets. (Recall that the set operations are to be performed on-line.) This gives an $\Omega(n)$ lower bound on the number of steps needed by any separable algorithm in the worst case. For $m = \Omega(n)$, Tarjan [11] derived an $\Omega(m\alpha(m, n))$ lower bound, where α is a functional inverse of Ackermann's function [1] defined as follows: For $i, j \geq 1$ let the function $A(i, j)$ be defined by

$$
\begin{array}{lll}
A(1, j) = 2^j & \text{for} & j \geq 1, \\
A(i, 1) = A(i - 1, 2) & \text{for} & i \geq 2, \\
A(i, j) = A(i - 1, A(i, j - 1)) & \text{for} & i, j \geq 2.
\end{array}
$$

Let $\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}$.

Remark 1.   The most important property of Ackermann's function is its explosive growth. In the usual definition of this function $A(1, j) = j + 1$, and the explosion does not occur quite so soon. However, this change only adds a constant to the inverse function α.

Remark 2.   The function α grows very slowly. $A(3, 1) = 16$; thus $\alpha(m, n) \leq 3$ for $n < 2^{16} = 65,536$. $A(4, 1) = A(2, 16)$, which is very large. Thus, for all practical purposes, $\alpha(m, n)$ is a constant no larger than four.

Remark 3.   For fixed n, $\alpha(m, n)$ decreases as m/n increases. In particular, let $a(i, n) = \min\{j \geq 1 \mid A(i, j) > \log n\}$. Then $\lfloor m/n \rfloor \geq a(i, n)$ implies $\alpha(m, n) \leq i$. For instance, $\lfloor m/n \rfloor \geq 1 + \log \log n$ implies $\alpha(m, n) \leq 1$; $\lfloor m/n \rfloor \geq \log^*\log n$ implies $\alpha(m, n) \leq 2$, where $\log^*n$ is defined by

$$
\begin{array}{l}
\log^{(0)}n = n, \qquad \log^{(i+1)}n = \log \log^{(i)}n, \\
\log^*n = \min\{i \mid \log^{(i)}n \leq 1\}.
\end{array}
$$

Tarjan's proof contains an error that was found and corrected by Banachowski [3]. Combining the $\Omega(n)$ and $\Omega(m\alpha(m, n))$ lower bounds, we obtain the following theorem:

THEOREM 2.   *Any separable algorithm for the set union problem requires* $\Omega(n + m\alpha(m + n, n))$ *time in the worst case.*

PROOF.   If $m \geq n$, the theorem follows from Tarjan's bound, since $1 \leq \alpha(m + n, n) \leq \alpha(m, n)$. If $m\alpha(n, n) \leq n$, the theorem follows from the $\Omega(n)$ bound. Finally, if $m\alpha(n, n) > n$ but $m < n$, we obtain from Tarjan's result a lower bound of $\Omega(m\alpha(m, m))$ if we ignore all but m elements. We have $m^2 \geq n$, and since $\alpha(m, m) \leq \alpha(n, n) \leq \alpha(m^2, m^2) \leq \alpha(m, m) + 1$, $\alpha(m, m) \geq \alpha(n, n) - 1 \geq \alpha(m + n, n) - 1$, implying the theorem in this case as well.   □

Remark.   For $m < n$, Theorem 2 improves Banachowski's lower bound [3] of $\Omega(n + m\alpha(n, m))$.

Theorem 2 provides a standard by which we shall judge set union algorithms: We call an algorithm *asymptotically optimal* if its worst-case running time is

$O(n + m\alpha(m + n, n))$. Every algorithm considered in this paper is separable and thus requires $\Omega(n + m\alpha(m + n, n))$ time; an upper bound of $O(n + m\alpha(m + n, n))$ is therefore the best for which we can hope.

2. *Linking by Size or Rank*

One way to make find paths shorter is to use a freedom implicit in the link operation: When performing *link* (e, f), we are free either to make f point to e or to make e point to f. Galler and Fischer [7] proposed *linking by size*: We make the root of the smaller tree point to the root of the larger tree, breaking a tie arbitrarily. The following versions of *make set* and *link* implement linking by size, using a field *size* (x) to store the size of the tree rooted at x:

procedure *make set(e, l)*;
  $p(e) := e$; *label(e)* $:= l$; *size(e)* $:= 1$
end *make set*;
procedure *link(e, f)*;
  if *size(e)* $\geq$ *size(f)* $\rightarrow$
      $p(f) := e$; *size(e)* $:=$ *size(e)* $+$ *size(f)*
  $[\!]$ *size(f)* $>$ *size(e)* $\rightarrow$
      $p(e) := f$; *size(f)* $:=$ *size(e)* $+$ *size(f)*; *label(f)* $:=$ *label(e)*
  fi
end *link*;

If we link by size, no find path has length exceeding log n.[1] We shall prove a stronger result that will be useful later. For any tree node x, we define *rank(x)* to be the height of x.[2]

LEMMA 1. *If x is any node in a forest built from single nodes by a sequence of link-by-size operations, size(x) $\geq 2^{rank(x)}$.*

PROOF. By induction on the number of links. The lemma is true before the first link. The size of a node never decreases as links are performed. The only way to increase the rank of a node x is to perform a link that causes a node y to point to x, in the process making the new rank of x equal to the old rank of y plus one. The new size of x is at least twice the old size of y. Thus, if the lemma holds before the link, it also holds after the link. $\square$

COROLLARY 1. *In a forest built from single nodes by a sequence of links by size, the number of nodes of rank i is at most $n/2^i$.*

PROOF. Ranks strictly increase along any path in the forest. Hence, any two nodes of the same rank are unrelated; that is, they have disjoint sets of descendants. By Lemma 1, any node of rank i has at least 2^i descendants. Thus there are at most $n/2^i$ such nodes. $\square$

COROLLARY 2. *In a forest built from single nodes by a sequence of links by size, no path has length exceeding log n.*

A linking method that achieves the same effect as linking by size while saving space is *linking by rank*: We maintain with each node its rank, rather than its size. When performing *link(e, f)*, we make the node of smaller rank point to the node

---

[1] Throughout this paper we use base-two logarithms.
[2] The height of a node x in a rooted tree is the length of the longest path from a leaf to x. (We adopt the convention that an edge in a rooted tree is directed from child to parent.) Every node is a descendant of itself.
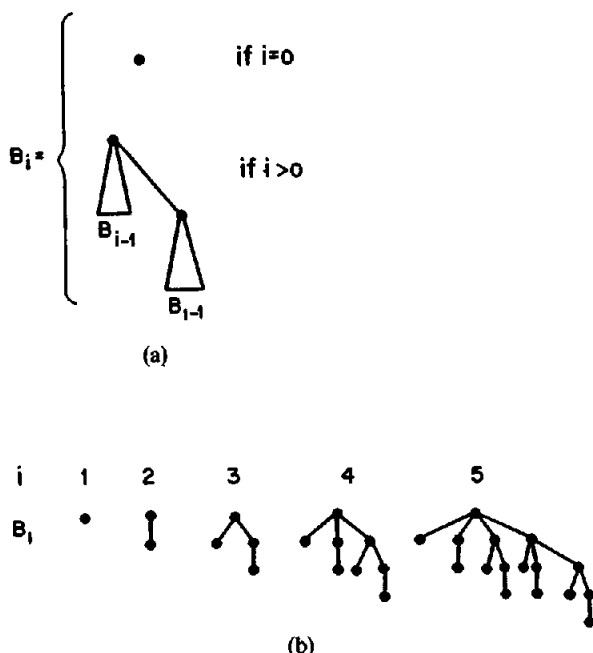
FIG. 1. Binomial trees. (a) Recursive definition. (b) Examples.

of larger rank. The following versions of *make set* and *link* implement linking by rank:

procedure *make set(e, l)*;
 $p(e) := e$; *label(e)* := *l*; *rank(e)* := 0
end *make set*;

procedure *link(e, f)*;
   if *rank(e)* > *rank(f)* → *p(f)* := *e*
     rank(e) = *rank(f)* → *p(f)* := *e*; *rank(e)* := *rank(e)* + 1
     rank(e) < *rank(f)* → *p(e)* := *f*; *label(f)* := *label(e)*
   fi
end *link*;

LEMMA 2. *Lemma* 1 *holds for linking by rank. That is, size(x)* $\geq 2^{rank(x)}$ *for all nodes x. Thus Corollaries* 1 *and* 2 *also hold for linking by rank.*

PROOF. By induction on the number of links. Suppose the lemma is true just before *link(e, f)*. Let *size* and *rank* denote the appropriate functions before the link and let *size'* and *rank'* denote the functions just after the link. Let *x* be any node. There are two cases. If $x \neq e$ or *rank(e)* $\neq$ *rank(f)*, then *size'(x)* ≥ *size(x)* and *rank'(x)* = *rank(x)*, which means the lemma is true after the link. If *x* = *e* and *rank(e)* = *rank(f)*, then *size'(x)* = *size'(e)* = *size(e)* + *size(f)* ≥ $2^{rank(e)} + 2^{rank(f)} = 2^{rank(e)+1} = 2^{rank'(x)}$, and again the lemma is true after the link. □

THEOREM 2 [5]. *With either linking by size or linking by rank, the set union algorithm runs in* $\Theta(n + m \log n)$ *time in the worst case.*

PROOF. The upper bound is immediate from Corollary 2. Binomial trees provide a class of examples showing that the bound is tight. A binomial tree B_0 consists of a single node. For $i > 0$, a binomial tree B_i is formed from two B_{i-1} trees by making the root of one the parent of the root of the other. (See Figure 1.) B_i has size 2^i and height *i*. For arbitrary *n* we can build a tree B_i containing $2^{\lfloor \log n \rfloor - 1}$
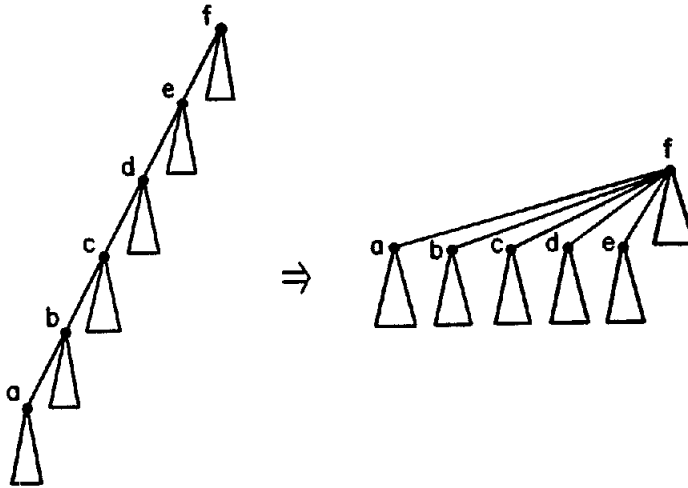
FIG. 2.   Compression of the path $a \to b \to c \to d \to e \to f.$

nodes using any linking rule, since the trees linked in each step are isomorphic. If we then repeatedly perform a find on the path of length $\lfloor \log n \rfloor - 1$, the total time is $\Omega(n + m \log n)$.   □

Linking by rank seems preferable to linking by size since it requires less storage; it needs only log log n bits per node (to store a rank in the range $[0, \lfloor \log n \rfloor]$[3]) instead of log n bits per node (to store a size in the range $[1, n]$). Linking by rank also tends to require less updating than linking by size. All the bounds we shall derive in subsequent sections hold equally for linking by rank and linking by size.

3. *Compression, Splitting, and Halving*

Another way to shorten find paths is to modify the trees during finds. To perform *find(e)*, we first follow parent pointers to the canonical element r of the set containing e; then we make every node on the find path point directly to r. (See Figure 2.) McIlroy and Morris devised this rule, called *compression* [8]. We offer two implementations of find with compression. The first uses recursion and has the advantage of succinctness; the second uses two explicit scans of the find path and is more efficient in practice.

```
function find(e);
    if p(e) = e → return e
    [] p(e) ≠ e → p(e) := find(p(e)); return p(e)
    fi
end find;
function find(e);
    local x, r;
    r := e; do p(r) ≠ r → r := p(r) od;
    x := e; do p(x) ≠ r → x, p(x) := p(x), r od;
    return r
end find;
```

Remark.   In the second version of *find* the statement "$x, p(x) := p(x), r$" denotes parallel assignment: $p(x)$ is assigned to x and r is assigned to $p(x)$ simultaneously.

---

[3] We use the notation $[j, k]$ to denote the set of integers $\{i \mid j \leq i \leq k\}$.

We can use compression with naive linking, with linking by rank, or with linking by size. If we use compression with linking by rank, the value of *rank(x)* computed by the algorithm is, in general, not the height of the compressed tree with root x but only an upper bound on this height. More precisely, the value of *rank(x)* is the height of the tree with root x that would have existed had there been no compression. We shall say more about the properties of *rank* below. Compression is easy to implement but hard to analyze, because the compressions change the forest representing the sets in a complicated way. For the set union algorithm with compression and linking by size, Tarjan [10] derived an $O(m\alpha(m, n))$ time bound, under the assumption that $m \geq n$.

Compression has the disadvantage that it requires two passes over a find path. Van Leeuwen and van der Weide [13, 14] proposed two variants of compression that require only one pass over a find path. The first is *splitting*: During a find we make each node along the find path (except the last and the next-to-last) point to the node two past itself. (See Figure 3.) Splitting breaks a find path into two paths, each about half as long as the original. The following version of find includes splitting:

function *find(e)*;
   local *x*;
   $x := e$; **do** $p(p(x)) \neq p(x) \rightarrow x, p(x) := p(x), p(p(x))$ **od**;
   return $p(x)$
end *find*;

The second variant is *halving*: During a find we make every other node along the find path (except the last and the next-to-last) point to the node two past itself. (See Figure 4.) Halving requires only about half as many pointer updates per find as splitting and intuitively has the advantage that it keeps the nodes on the find path together while it halves the length of the find, so that later finds will produce more compression. The following version of find includes halving:

function *find(e)*;
   local *x*;
   $x := e$; **do** $p(p(x)) \neq p(x) \rightarrow x := p(x) := p(p(x))$ **od**;
   return $p(x)$
end *find*;

Remark 1.   The statement "$x := p(x) := p(p(x))$" denotes sequential assignment: The value of $p(p(x))$ is assigned to $p(x)$ and then to x.

Remark 2.   An optimized version of this procedure uses one test and one pointer extraction for each node on the find path, and one pointer assignment for every other node on the find path.

For the set union algorithm with linking by size and either splitting or halving, van Leeuwen and van der Weide derived an $O(m \log^* n)$ time bound, under the assumption that $m \geq n$. We shall prove that the algorithm with either linking by rank or linking by size and either compression, splitting, or halving runs in $O(n + m\alpha(m + n, n))$ time for arbitrary m and n. Thus these six methods are all asymptotically optimal. To derive this bound, we use the *multiple partition* technique of Tarjan [10]. We extend the technique in two ways. First, we modify it so that it gives a tight bound for $m < n$, as well as for $m \geq n$. Second, we generalize it so that it applies to a large class of find methods. Consider any find path. We perform a *compaction* on the path by making each node on the path (except the last and the next-to-last) point to a node at least two past itself. (See Figure 5.)
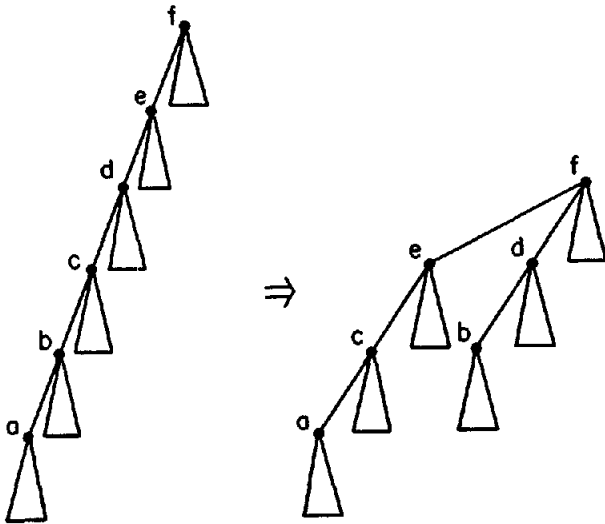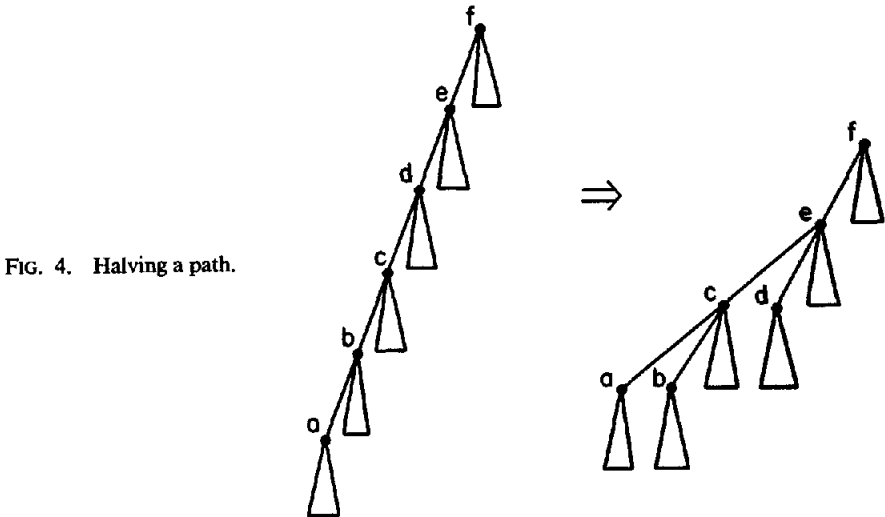
FIG. 3. Splitting a path.

FIG. 4. Halving a path.

Compression is locally the best kind of compaction, since it moves nodes as close to the root as possible, whereas splitting is locally the worst. Halving is not a form of compaction, but with minor changes our method gives a tight bound for halving as well. Although the analysis below is self-contained, some familiarity with the technique of [10] will serve the reader well.

Consider any sequence of intermixed *make set*, *find*, and *link* operations, implemented so that every find compacts the find path. The time required to carry out the sequence is bounded by a constant times the sum of n and the total number of nodes on find paths, assuming that the compaction of a path takes time linear in the number of nodes on the path. To analyze the total number of nodes on find paths, we need some tools. For the moment we shall not specify the linking method, since we can use the same proof framework to analyze compaction with any of the three linking methods.

We measure the pointer changes caused by the compactions with respect to a fixed forest called the *reference forest*. The reference forest of a sequence of *make*
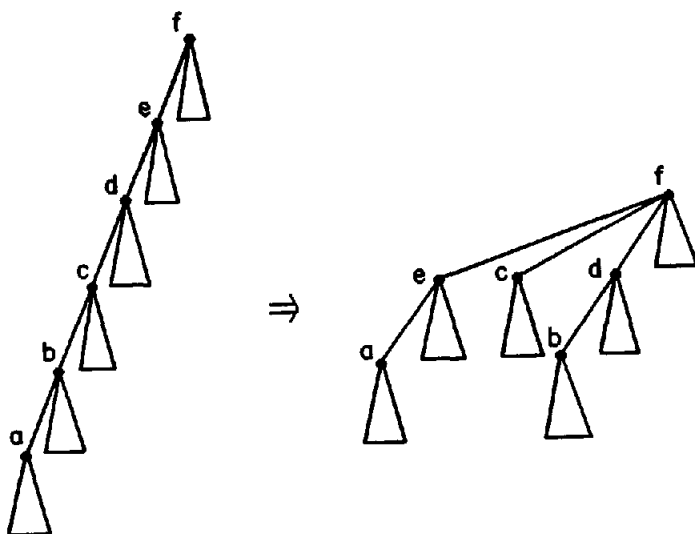
FIG. 5. Compaction of a path. Candidates for the new parent of node a are c, d, e, and f; for node b. d, e, and f; for node c: e and f; and for node d: f.

set, find, and *link* operations is the forest produced by carrying out all the *make set* and *link* operations while ignoring all the finds. Thus no compaction takes place. (The parent of a node x in the reference forest is the *first* value other than x assigned to $p(x)$ by the algorithm.) For the duration of this section we define the rank of a node to be its height in the reference forest. The rank of a node is fixed throughout the running of the algorithm; if linking by rank is used, the rank of a node x is the *last* value assigned to $rank(x)$ by the algorithm.

The following properties hold for any sequence of *make set, find,* and *link* operations with compaction. For any node x, $p(x)$ is always a proper ancestor of x in the reference forest. Thus ranks strictly increase along any find path. Let p denote the parent function just before a find, and let p' denote the parent function just after the find. If x is any node on the find path other than the last and the next-to-last, then compaction ensures that $p'(x)$ is an ancestor of $p(p(x))$ in the reference forest. More important, compression causes the rank of the parent of x to increase from $rank(p(x))$ to at least $rank(p(p(x)))$. (Note that ranks never change, but parents do.) By analyzing these rank increases, we can bound the total number of nodes on find paths. To get the best bounds, we must group the rank changes into levels and account separately for each level of change.

To group rank changes, we define a collection of partitions on the integers from zero to the maximum rank of a node. There is one such partition for each level $i \in [0, k]$, where k is a parameter to be chosen later. The blocks of the level i partition are intervals defined by

$$block(i, j) = [B(i, j), B(i, j + 1) - 1] \qquad \text{for} \quad j \in [0, l_i - 1],$$

where the interval boundaries $B(i, j)$ and the number of intervals l_i in level i are also parameters to be chosen later.

For this definition to make sense, we require that the boundary function $B(i, j)$, which is defined for $i \in [0, k]$, $j \in [0, l_i]$, have the following properties, whose meanings are explained below.

(a) $B(0, j) = j$ for $j \in [0, l_0]$;

(b) $B(i, 0) = 0$ for $i \in [1, k]$;

(c) $B(i, j) < B(i, j + 1)$ for $i \in [1, k]$, $j \in [0, l_i - 1]$;

(d) $B(i, l_i) > h$ for $i \in [0, k]$, where h is the maximum rank of a node;

(e) $l_k = 1$.

Property (c) implies that the blocks of the level i partition are nonempty disjoint intervals. Properties (b) and (d) imply that every integer in the range $[0, h]$ is in some block of the level i partition. Property (a) implies that each block of the level zero partition is a singleton. Property (e) implies that the level k partition consists of a single block.

Each level of blocks partitions the nodes by rank. For $i \in [1, k]$, $j \in [0, l_i - 1]$, let n_{ij} be the number of nodes with rank in $block(i, j) - block(i - 1, 0)$. Then, for any i,

$$\sum_{j=0}^{l_i-1} n_{ij} \le n.$$

Our intention is that the partition become coarser as the level increases. As a measure of this coarsening we use the function b_{ij}, which for $i \in [1, k]$, $j \in [0, l_i - 1]$ is the number of level $i - 1$ blocks whose intersection with $block(i, j)$ is nonempty.

As the algorithm proceeds, each node x has a *level*, defined to be the minimum value of i such that $rank(x)$ and $rank(p(x))$ are in the same block of the level i partition. Since the level zero partition consists of singletons and the level k partition consists of a single block, $level(x) \in [1, k]$ unless x is a tree root, in which case $level(x) = 0$. Whereas the rank of a node is fixed, the level of a node can increase, but not decrease, as the algorithm proceeds.

To bound the number of nodes on find paths, we assign a *charge* for each find. The charge is allocated among the nodes on the find path and the find itself, in a way that depends upon the levels of the nodes just before the find takes place. The charge assigned to a given node is further allocated among levels. The charging rules are somewhat complicated because of the generality of the results we are trying to obtain. We use two rules to assign charge:

Find Charging. Charge $3k$ to the find itself.

Node Charging. Let x be any node on the find path other than the first and the last. Let i be the maximum level of any node preceding x on the path. If

$$\min\{i, level(p(x))\} \ge level(x),$$

charge

$$\min\{i, level(p(x))\} - level(x) + 1$$

to x. Of this amount, charge one to each level in the range

$$[level(x), \min\{i, level(p(x))\}].$$

Note. The node charging rule does not charge the next-to-last node on a find path.

LEMMA 3. *The amount charged for a find is at least the number of nodes on the find path.*

PROOF. Let $x_0 \to x_1 \to \cdots \to x_h$ be a find path that starts at node x_0 and ends at node x_h. Let *level* be the level function just before the find, and let

$$P = \{i \mid i \in [0, h - 1] \text{ and } level(x_i) \leq level(x_{i+1})\};$$
$$N = \{i \mid i \in [0, h - 1] \text{ and } level(x_i) > level(x_{i+1})\}.$$

Then

$$\sum_{i=0}^{h-1} (level(x_{i+1}) - level(x_i)) = level(x_h) - level(x_0) \geq -level(x_0),$$

which implies

$$\sum_{i \in P} (level(x_{i+1}) - level(x_i)) \geq -level(x_0)$$
$$+ \sum_{i \in N} (level(x_i) - level(x_{i+1})) \geq -level(x_0) + |N|,$$

and

$$\sum_{i \in P} (level(x_{i+1}) - level(x_i) + 1) \geq |P| + |N| - level(x_0) = h - level(x_0).$$

Let $y_0, y_1, \ldots, y_g$ be the subsequence of $x_0, x_1, \ldots, x_h$ consisting of those nodes x_i whose level exceeds the level of all previous nodes on the path. Note that $y_0 = x_0$, none of the nodes y_j is charged for the find, and $g \leq k - 1$. Consider the amount charged for the find by the node charging rule. If $x_i \in P$ but x_{i+1} is not in the y-subsequence, then a charge of $level(x_{i+1}) - level(x_i) + 1$ is assigned to x_i. (In this case, x_i cannot be in the y-subsequence.) If $x_i \in P$ and $x_{i+1} = y_j$ for some j, a charge of at least

$$level(y_{j-1}) - level(x_i) = level(y_j) - level(x_i) + 1 - (level(y_j) - level(y_{j-1}) + 1)$$

is assigned to x_i. (This includes the possibility that $y_{j-1} = x_i$, in which case the charge assigned to x_i is zero.) Thus the total charge (including the charge to the find itself) is at least

$$\sum_{i \in P} (level(x_{i+1}) - level(x_i) + 1) - \sum_{j=1}^{g} (level(y_j) - level(y_{j-1}) + 1) + 3k$$
$$\geq h - level(x_0) - level(y_g) + level(y_0) - (k - 1) + 3k \geq h + 1,$$

since $x_0 = y_0$. □

The following lemma gives a formula bounding the total charge for all finds:

LEMMA 4. *The total charge for all finds, and hence the total number of nodes on find paths, is at most*

$$3km + \sum_{i=1}^{k} \sum_{j=0}^{l_i-1} b_{ij} n_{ij}.$$

PROOF. If a node x is charged at level i, then $1 \leq level(x) \leq i$ before the find and $level(x) \geq i$ after the find. Let $rank(x) \in block(i, j)$ and suppose $level(x) = i$ after the find. This implies that $level(p(x)) = i$ before the find, for if $level(p(x)) > i$ before the find, then $level(x) > i$ after the find. To say that $level(p(x)) = i$ means that $rank(p(x))$ and $rank(p(p(x)))$ are in different level $i - 1$ blocks. The find thus causes $p(x)$ to change so that $rank(p(x))$ is in a new level $i - 1$ block. This can happen at most $b_{ij} - 1$ times without increasing the level

$$0 \quad 2^1 \quad 2^2 \quad 2^3 \quad 2^4 \quad 2^5 \quad 2^6 \quad 2^7 \quad 2^8 \quad 2^9 \quad 2^{10} \quad 2^{11} \quad 2^{12} \quad 2^{13} \quad 2^{14} \quad 2^{15} \quad 2^{16}$$
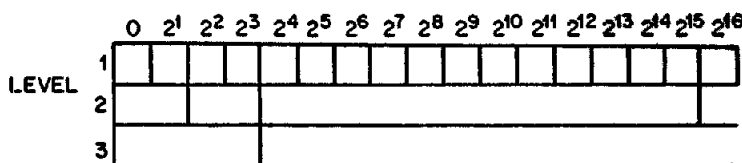
LEVEL 1
2
3

FIG. 6. Multiple partition for compaction with linking by rank or size. Level zero is omitted and a logarithmic scale is used.

of x, since $block(i, j)$ intersects only b_{ij} level $i - 1$ blocks. Thus, after x is changed b_{ij} times at level i, its level is at least $i + 1$, and it is never again charged at level i.

For x to be charged at level i, there must be a predecessor x' of x on the find path such that $level(x') \geq i$ before the find. By the definition of level, $rank(x')$ and $rank(p'(x))$ are in different level $i - 1$ blocks before the find. Since $rank(x) \geq rank(p(x'))$, $rank(x) \notin block(i - 1, 0)$. This implies that the number of nodes whose rank is in $block(i, j)$ that can be charged at level i is at most n_{ij}.

Summing all the charges, we obtain a total charge of at most

$$3km + \sum_{i=1}^{k} \sum_{j=0}^{l_i-1} b_{ij} n_{ij}. \qquad \square$$

Now we are ready to focus on a particular version of the set union algorithm. Suppose we use either linking by rank or linking by size. The next lemma bounds n_{ij}.

LEMMA 5. *With linking by rank or linking by size,*

$$n_{ij} \leq \frac{n}{2^{\max\{B(i,j),B(i-1,1)\}-1}}.$$

PROOF. By Corollary 1,

$$n_{ij} \leq \sum_{h=\max\{B(i,j),B(i-1,1)\}}^{B(i,j+1)-1} \frac{n}{2^h} \leq \sum_{h=\max\{B(i,j),B(i-1,1)\}}^{\infty} \frac{n}{2^h}$$

$$= \frac{n}{2^{\max\{B(i,j),B(i-1,1)\}-1}}. \qquad \square$$

LEMMA 6. *In any sequence of set operations implemented using any form of compaction and either linking by rank or linking by size, the total number of nodes on find paths is at most* $3m\alpha(m + n, n) + 4m + 13n$.

PROOF. Choose $k = \alpha(m + n, n) + 1$, $l_i = \min\{j \mid A(i, j) > \log n\}$ for $i \in [1, \alpha(m + n, n)]$, $l_k = 1$, and

$$B(i, j) = A(i, j) \quad \text{for} \quad i \in [1, \alpha(m + n, n)], \quad j \in [1, l_i];$$

$$B(k, 1) = \lfloor \log n \rfloor + 1.$$

(See Figure 6.)

With this definition, it is easy to see that the boundary function $B(i, j)$ has properties (c), (d), and (e). (By Corollary 1, no node has rank exceeding $\log n$.) We estimate b_{ij} as follows:

(i) $b_{i0} = 2$ for $i \in [1, \alpha(m + n, n)]$: For $i \in [1, \alpha(m + n, n)]$,

$$block(i, 0) = block(i - 1, 0) \cup block(i - 1, 1)$$

since $A(1, 1) = 2$ and $A(i, 1) = A(i - 1, 2)$ for $i \geq 2$.

(ii) $b_{ij} \le A(i, j)$ for $i \in [1, \alpha(m + n, n)]$, $j \in [1, l_i - 1]$: For $j \in [1, l_1 - 1]$,

$$block(1, j) = [A(1, j), A(1, j + 1) - 1] = [2^j, 2^{j+1} - 1].$$

Thus $b_{1j} = 2^j = A(1, j)$. For $i \in [2, \alpha(m + n, n)]$, $j \in [1, l_i - 1]$,

$$
\begin{aligned}
block(i, j) &= [A(i, j), A(i, j + 1) - 1] \\
&= [A(i, j), A(i - 1, A(i, j)) - 1] \\
&\subseteq [0, A(i - 1, A(i, j)) - 1] = \bigcup_{h=0}^{A(i,j)-1} block(i - 1, h).
\end{aligned}
$$

Thus $b_{ij} \le A(i, j)$.

(iii) $b_{k0} = \lfloor(m + n)/n\rfloor$: We have $b_{k0} = l_{\alpha(m+n,n)} = \min\{j \mid A(\alpha(m + n, n), j) > \log n\}$ $\le \lfloor(m + n)/n\rfloor$ by the definition of α.

To estimate the bound given in Lemma 4, we break the sum

$$\sum_{i=1}^{k} \sum_{j=0}^{l_i-1} b_{ij} n_{ij}$$

into three parts: First,

$$\sum_{j=0}^{l_k-1} n_{kj} b_{kj} = n_{k0} b_{k0} \le n \lfloor(m + n)/n\rfloor \le m + n.$$

Second,

$$\sum_{i=1}^{k-1} b_{i0} n_{i0} \le \sum_{i=1}^{k-1} \frac{2n}{2^{B(i-1,1)-1}} \quad \text{by Lemma 5}$$

$$\le 4n \sum_{i=1}^{k-1} \frac{1}{2^{B(i-1,1)}} \le 4n.$$

Third, for $i \in [1, \alpha(m + n, n)]$,

$$\sum_{j=1}^{l_i-1} b_{ij} n_{ij} \le \sum_{j=1}^{l_i-1} \frac{A(i, j)n}{2^{A(i,j)-1}} \quad \text{by Lemma 5}$$

$$\le \sum_{h=A(i,1)}^{\infty} \frac{hn}{2^{h-1}} = \frac{n(A(i, 1) + 1)}{2^{A(i,1)-2}},$$

which implies

$$\sum_{i=1}^{k-1} \sum_{j=1}^{l_i-1} b_{ij} n_{ij} \le \sum_{i=1}^{k-1} \frac{n(A(i, 1) + 1)}{2^{A(i,1)-2}}$$

$$\le n \sum_{h=2}^{\infty} \frac{(h + 1)}{2^{h-2}} = 8n.$$

Combining estimates, we discover that the total charge for all finds is at most $3m(\alpha(m + n, n) + 1) + m + 13n$. $\square$

THEOREM 3. *The set union algorithm with either linking by rank or linking by size and either compression, splitting, or halving runs in $O(n + m\alpha(m + n, n))$ time and thus is asymptotically optimal.*

PROOF. For compression and splitting, the theorem is immediate from Lemma 6. For halving, we must change the multiple partition method slightly. We call every other node on a find path, starting with the first, *essential*. The essential

nodes on the path (except the last) are exactly the nodes whose parents change when the path is halved. For any node x, we define $level(x)$ to be the minimum value of i such that $rank(x)$ and $rank(p^2(x))$ are in the same block of the level i partition. We assign charge to nodes as follows:

Node Charging. Let x be any essential node on the find path other than the first and the last. Let i be the maximum level of any essential node preceding x on the path. If

$$\min\{i, level(p^2(x))\} \geq level(x),$$

charge

$$\min\{i, level(p^2(x))\} - level(x) + 1$$

to x.

Of this amount, charge one to each level in the range $[level(x), \min\{i, level(p^2(x))\}]$.

The proofs of Lemmas 3, 4, and 6 now apply and serve to bound the total count of essential nodes on find paths. (We must replace $p(x)$ by $p^2(x)$ throughout the proof of Lemma 4.) Since at least half the nodes on each find path are essential, twice the bound of Lemma 6 holds as a bound for halving. $\square$

Theorem 3 gives us six different asymptotically optimal set union algorithms. Perhaps the best is linking by rank with halving, since it saves space over linking by size and uses only one pass over each find path.

In the remainder of this paper, we analyze a number of other set union algorithms. This analysis will give us insight into the behavior of different variants of compression and into the benefits of linking by rank or size instead of linking naively. We begin by analyzing compression, splitting, and halving with naive linking. This is not just a theoretical exercise, since these techniques can be used to solve a generalization of the set union problem in which naive linking is the only linking method possible [12].

Our results are as follows: For the case $m \geq n$, all three methods have the same asymptotic running time: $\Theta(m \log_{(1+m/n)} n)$. For $m < n$, compression with naive linking runs in $\Theta(n + m \log n)$ time and splitting with naive linking runs more slowly, in $\Theta(n \log m)$ time. We have not been able to obtain a tight bound for halving with naive linking in the case $m < n$; our best upper bound is $O(n \log m)$ and our best lower bound is $\Omega(n + m \log n)$.

We begin by deriving the upper bounds.

LEMMA 7. *Suppose $m \geq n$. In any sequence of set operations implemented using any form of compaction and naive linking, the total number of nodes on find paths is at most $(4m + n) \lceil \log_{(1+m/n)} n \rceil$. With halving and naive linking, the total number of nodes on find paths is at most $(8m + 2n) \lceil \log_{(1+m/n)} n \rceil$.*

PROOF. We apply the multiple partition method. Consider any form of compaction with naive linking. Choose $k = \lceil \log_{(1+m/n)} n \rceil$, $l_i = \lceil n / \lfloor 1+m/n \rfloor^i \rceil$ for $i \in [1, k]$, and $B(i, j) = j \lfloor 1 + m/n \rfloor^i$ for $i \in [1, k]$, $j \in [1, l_i]$. (See Figure 7.) Properties (c)–(e) are immediate. (Since there are only n nodes, no rank exceeds $n - 1$.) For $i \in [1, k]$, $j \in [0, l_i - 1]$, $b_{ij} = \lfloor 1 + m/n \rfloor$. By Lemma 4, the total number of nodes on find paths is at most

$$3km + \sum_{i=1}^{k} \sum_{j=0}^{l_i-1} \lfloor 1 + m/n \rfloor n_{ij}$$

$$\leq 3km + k(n + m) = (4m + n)k = (4m + n)\lceil \log_{(1+m/n)} n \rceil.$$
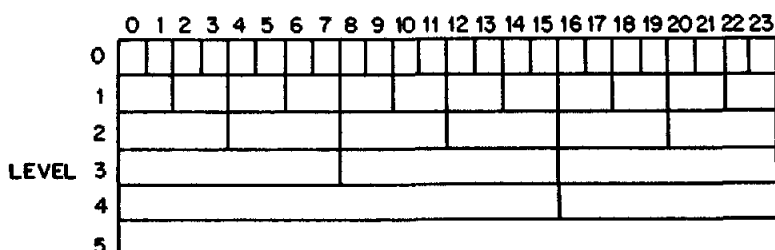
FIG. 7.   Multiple partition for compaction with naive linking if $\lfloor 1 + m/n \rfloor = 2$.

The bound for halving with naive linking follows as in the proof of Theorem 3. $\square$

LEMMA 8.   *Suppose $m < n$. In any sequence of set operations implemented using any form of compaction and naive linking, the total number of nodes on find paths is at most $(3m + 2n)\lceil \log m \rceil + 2(n + m)$. With halving and naive linking, the total number of nodes on find paths is at most $(6m + 2n)\lceil \log m \rceil + 4(n + m)$.*

PROOF.   We apply the multiple partition method, modified to estimate the charge at level k (the highest level) in a different way. In order for a node x to be charged at level k, $p(x)$ must be at level k; that is, $rank(p(x))$ and $rank(p^2(x))$ must be in different blocks of the level $k - 1$ partition. This means that for a given find at most $b_{k0}-1$ nodes can be charged at level k. Incorporating this estimate into the proof of Lemma 4, we obtain an upper bound of

$$(3k + b_{k0}-1)m + \sum_{i=1}^{k-1} \sum_{j=0}^{l_i-1} b_{ij}n_{ij}$$

on the total number of nodes on find paths.

Consider any form of compaction with naive linking. Choose $k = \lceil \log m \rceil + 1$, $l_i = \lceil n/2^i \rceil$ for $i \in [1, k - 1]$, $l_k = 1$, $B(i, j) = j2^i$ for $i \in [1, k - 1]$, $j \in [1, l_i]$, and $B(k, 1) = n$. As usual, properties (c)–(e) are immediate. We estimate b_{ij} as follows: for $i \in [1, k - 1]$, $j \in [0, l_i - 1]$, $b_{ij} = 2$; $b_{k0} \leq \lceil n/m \rceil$. Plugging into the estimate above, we find that the total number of nodes on find paths is at most

$$(3\lceil \log m \rceil + 2 + \lceil n/m \rceil)m + \lceil \log m \rceil 2n = (3m + 2n)\lceil \log m \rceil + 2(n + m).$$

The bound for halving with naive linking follows as in the proof of Theorem 3. $\square$

LEMMA 9.   *Suppose $m < n$. In any sequence of set operations implemented using compression and naive linking, the total number of nodes on find paths is at most $n + 2m\lceil \log n \rceil + m$.*

PROOF.   We use a modified form of the multiple partition method. Since the method is simpler in this case, we shall describe it more or less from scratch. For a node x such that $p(x) \neq x$, let the *level* of x be the minimum value of i such that $rank(p(x)) - rank(x) \geq 2^{i-1}$; for a node x such that $p(x) = x$, let the level of x be zero. Call a node *active* if it is returned by at least one of the m finds, *passive* otherwise. There are at most m active nodes. Furthermore, after a node x is on at least one find path its parent $p(x)$ is active, and although $p(x)$ may change, it remains an active node.

We charge for a find as follows:

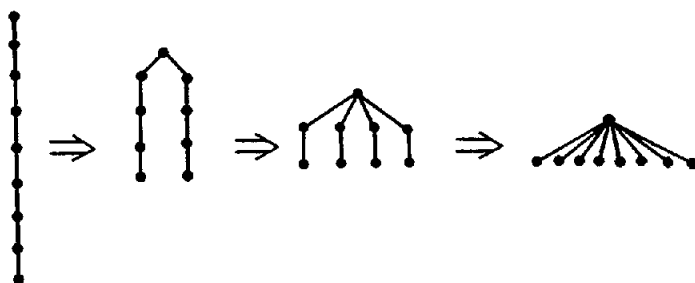*Find Charging.*   Charge $\lceil \log n \rceil + 2$ to the find.

FIG. 8. Repeated splitting of a long path.

Active Node Charging. If x is an active node on the path of level i that is followed somewhere on the path by another node of level i, charge one to x.

Passive Node Charging. Charge one to every passive node on the path other than the first.

We first observe that the amount charged for a find is at least the number of nodes on the find path: We charge one for every passive node except possibly one, and one for every active node except possibly one per level. Since all ranks are in the range $[0, n - 1]$, all levels are in the range $[0, \lceil \log n \rceil]$. Thus the amount charged to the find is at least as large as the number of uncharged nodes.

A passive node that is charged has a child whose parent changes from passive to active because of the find. This can happen at most n times, for a total charge of n. When an active node is charged, its level increases by at least one. Thus an active node is charged at most $\lceil \log n \rceil - 1$ times, for a total charge of $m(\lceil \log n \rceil - 1)$. The total charge to finds is $m(\lceil \log n \rceil + 2)$, giving a grand total of $n + 2m\lceil \log n \rceil + m$. □

To obtain lower bounds for compression, splitting, and halving with naive linking, we must construct time-consuming sequences of set operations. We begin by considering splitting for $m < n$, since this is by far the easiest case. Suppose we build up a path containing $2^i + 1$ nodes (for some $i \geq 1$) and then split it. The result is two paths of $2^{i-1} + 1$ nodes sharing a common final vertex. We can split each of these to obtain four paths of length $2^{i-2} + 1$ with common final vertex, and repeat this process until we obtain 2^i paths of length two. (See Figure 8.)

To make this example more precise, assume $n > m \geq 3$. Let $i = \lfloor \log(n - 1) \rfloor$ and $j = \lfloor \log m \rfloor$. Build a path of $2^i + 1 \leq n$ nodes. Then perform one split of length $2^i + 1$, two splits of length $2^{i-1} + 1, \ldots, 2^{j-1}$ splits of length $2^{i-j+1} + 1$. The total number of splits is $2^j - 1 \leq m$. The total length of the find paths is at least $j2^i = \Omega(n \log m)$. Thus, the bound in Lemma 8 is tight for splitting.

Let us turn to the case $m \geq n$. For each of the three versions of find, we shall construct sequences of set operations requiring $\Omega(m \log_{(1+m/n)} n)$ time, thus showing that the bound in Lemma 7 is tight. We begin with compression. Fischer [5] noted that if we link a tree containing a single node with a binomial tree B_i and then compress the longest path, the result is a new binomial tree B_i with an extra node. That is, binomial trees are self-reproducing under compression and linking with a single node. (See Figure 9.) Fischer thus obtained a lower bound of $\Omega(n \log n)$ for compression with naive linking. This class of examples gives a more general lower bound of $\Omega(n + m \log n)$ for arbitrary $m < n$, implying that the bound in Lemma 9 is tight.
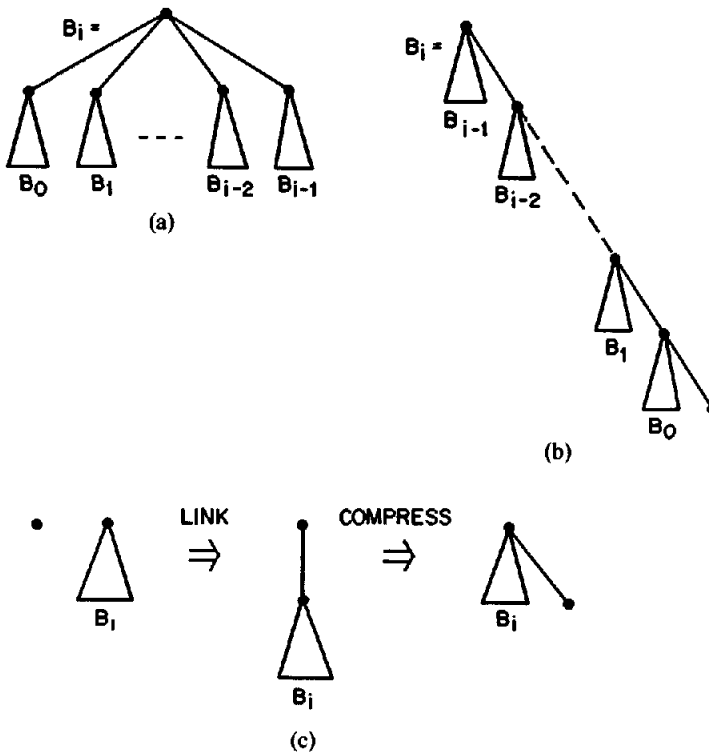
FIG. 9.   Self-reproduction of B_i. (a) Horizontal unrolling of B_i using recursive definition. (b) Vertical unrolling of B_i. (c) Self-reproduction.

We shall generalize Fischer's idea. Let j be any positive integer. For $k \geq 1$ we define the class of trees T_k recursively as follows: For $k \leq j$, T_k is a tree with a single node. For $k > j$, T_k is formed from T_{k-1} and T_{k-j} by making the root of T_{k-1} the parent of the root of T_{k-j}. (See Figure 10.) Note that if $j = 1$, we obtain the binomial trees.

T_k has the following property: If we link a tree containing a single node with a T_k tree and then perform j compressions, we obtain a new T_k tree with an extra node. To demonstrate this self-reproduction, we use the recursive definition of T_k. Suppose $k \geq 2j$. By applying the definition j times, we can unroll T_k horizontally, into a tree consisting of T_{k-j} linked with $T_{k-2j+1}, T_{k-2j+2}, \ldots$, and T_{k-j}. (See Figure 11a.) By continuing to expand the tree at the root, we eventually unroll T_k into a tree consisting of a root and subtrees $T_1, T_2, \ldots, T_{k-j}$. (See Figure 11b.)

We can, on the other hand, continue the expansion by unrolling each of the trees $T_{k-2j+1}, T_{k-2j+2}, \ldots, T_{k-j}$ vertically. For ease of description, let us assume that k is a multiple of j, say $k = ij$. Figure 12 illustrates the unrolled tree T_k, which consists of T_{k-j} linked with j subtrees. The hth subtree is a path whose nodes are the roots of trees $T_h, T_{j+h-1}, T_{2j+h-1}, \ldots, T_{k-2j+h-1}$.

The vertically unrolled tree T_k contains one copy of T_h for each h in the interval $[1, k - j]$ and an extra copy of T_j. But T_j is a tree with a single node. The horizontally unrolled T_k also contains one copy of T_h for each h in the interval $[1, k - j]$. Thus, if we link a tree containing a single node with T_k and perform compressions on the j vertically unrolled paths, we obtain T_k with an extra node. (See Figure 13.) Each of the j paths compressed contains $i + 1$ nodes. An easy induction shows that T_k contains at most $(j+1)^{i-1}$ nodes.
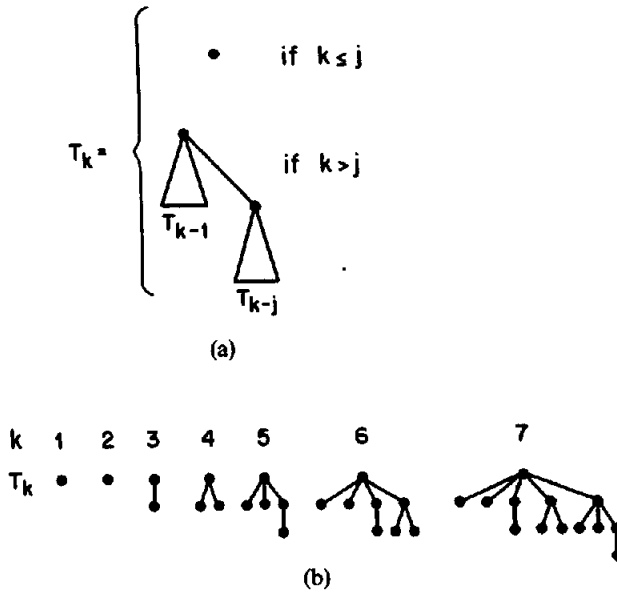
FIG. 10. T_k trees. (a) Recursive definition of T_k. (b) Examples of T_k for $j = 2$.
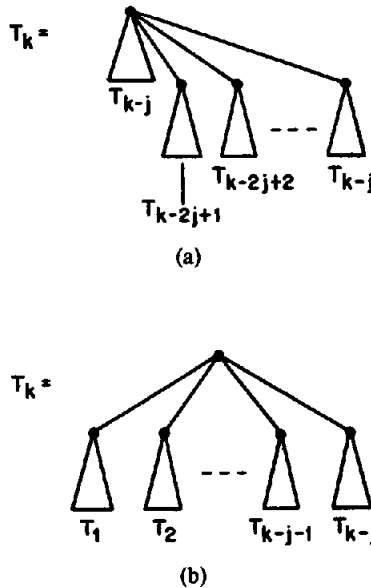

FIG. 11. T_k after horizontal unrolling. (a) After j-fold unrolling. (b) After complete unrolling.

We obtain bad examples for compression with naive linking in the case $m \geq n$ as follows: Suppose $m \geq n \geq 2$. Let $j = \lfloor m/n \rfloor$, $i = \lfloor \log_{j+1}(n/2) \rfloor + 1$, and $k = ij$. Build a T_k tree. Note that $|T_k| \leq (j + 1)^{i-1} \leq n/2$. Repeat the following operations $\lfloor n/2 \rfloor$ times: Link a single-node tree with the existing tree, which consists of T_k with some extra nodes. Then perform j finds, each traversing a path of $i + 1$ nodes, to reproduce T_k with some extra nodes. There are at most m finds, and the total number of nodes on find paths is at least $j\lfloor n/2 \rfloor (i + 1) = \Omega(m \log_{(1+m/n)} n)$.
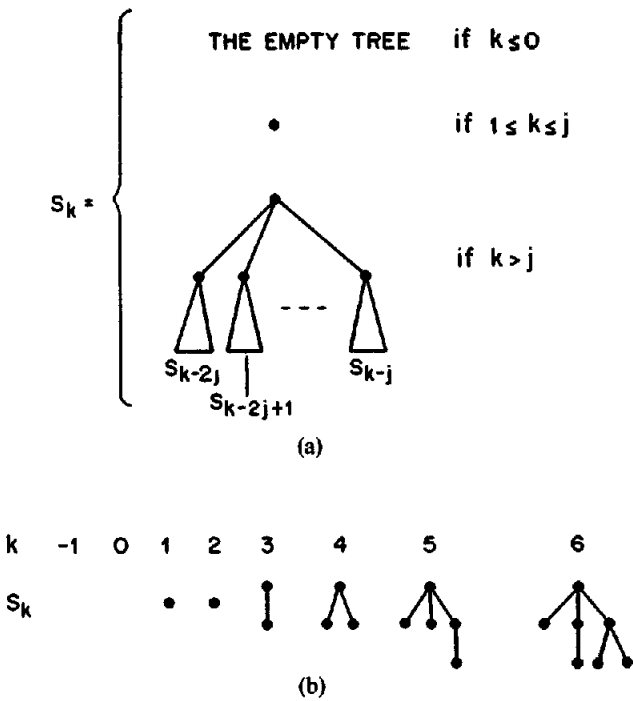
FIG. 12. T_k for $k = ij$ after j-fold horizontal and complete vertical unrolling.

FIG. 13.   Self-reproduction of T_k.

Summarizing our results for compression with naive linking, we have the following theorem (note that

$$\log_{(2+m/n)} n = \Theta(\log_{(1+m/n)} n) \qquad \text{if} \quad m \geq n,$$
$$\log_{(2+m/n)} n = \Theta(\log n) \qquad \text{if} \quad m < n).$$

THEOREM 4.   *The set union algorithm with compression and naive linking runs in $\Theta(n + m \log_{(2+m/n)} n)$ time.*

For splitting, we can define a similar class of self-reproducing trees, called S_k trees. Let $j \geq 1$. For each integer k, define S_k as follows: If $k \leq 0$, S_k is an empty tree. If $k \in [1, j]$, S_k is a single-node tree. If $k > j$, S_k consists of a root and $j + 1$ subtrees, $S_{k-2j}, S_{k-2j+1}, \ldots, S_{k-j}$. (See Figure 14.) Note that S_k for $k \in [j, 2j]$ consists of a root with $k - j$ children.

For $k = ij$, S_k has the following property: If we unite a single-node tree with S_k and then perform j splittings, each on a path of $i + 1$ nodes, the result is a new S_k tree with an extra node. To demonstrate this self-reproduction, we introduce an auxiliary class of trees R_k. For $k \geq j$, R_k consists of a root and j subtrees, S_{k-2j+1}, $S_{k-2j+2}, \ldots, S_{k-j}$. (See Figure 15a.) Note that $R_k = S_k$ for $k \in [j, 2j-1]$. For $k > j$, we can represent S_k as R_{k-1} linked with S_{k-j} or alternatively as R_k linked with S_{k-2j}. (See Figure 15b.)

Suppose $k = ij$ with $i \geq 1$. We can unroll the j subtrees $S_{k-2j+1}, S_{k-2j+2}, \ldots, S_{k-j}$ of S_k into paths of i nodes using the first expansion of Figure 15b. (See Figure 16.)

$$S_k =$$

THE EMPTY TREE   if $k \leq 0$

•   if $1 \leq k \leq j$

if $k > j$

S_{k-2j} S_{k-j}

S_{k-2j+1}

(a)

k   -1 0 1 2 3   4   5   6

S_k • • ┃ ∧ ∧┃   ∧┃∧

(b)

FIG. 14. S_k trees. (a) Recursive definition of S_k. (b) Examples for $j = 2$.

$$R_k =$$

S_{k-2j+1} S_{k-2j+2} S_{k-j}

(a)

$$S_k =$$ $=$

R_{k-1} R_k

S_{k-j} S_{k-2j}

(b)
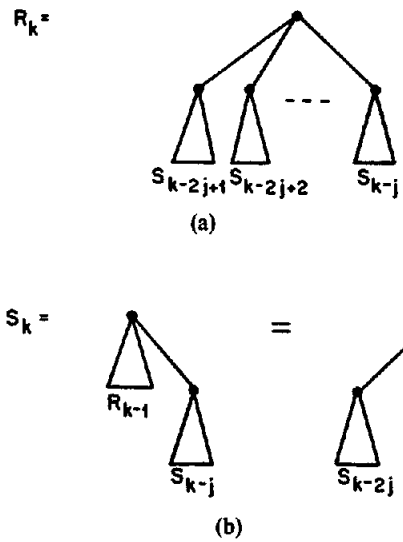
FIG. 15. R_k trees. (a) Definition. (b) Alternative expansions of S_k for $k > j$.

If we unite a single-node tree with S_k and then split the unrolled paths, we can roll up the resulting tree into S_k with an extra node, using the second expansion of Figure 15b and the equalities $S_k = S_{k-1}$ for $k \in [2, j]$ and $R_k = S_k$ for $k \in [j, 2j - 1]$. Figure 17 illustrates the case of even i; the case of odd i is similar.

Each of the j paths split contains $i + 1$ nodes. An easy induction shows that S_k contains at most $(j + 1)^{i-1}$ nodes for $k \geq 1$. We can construct bad examples for splitting with naive linking using S_k trees just as we did for compression with naive
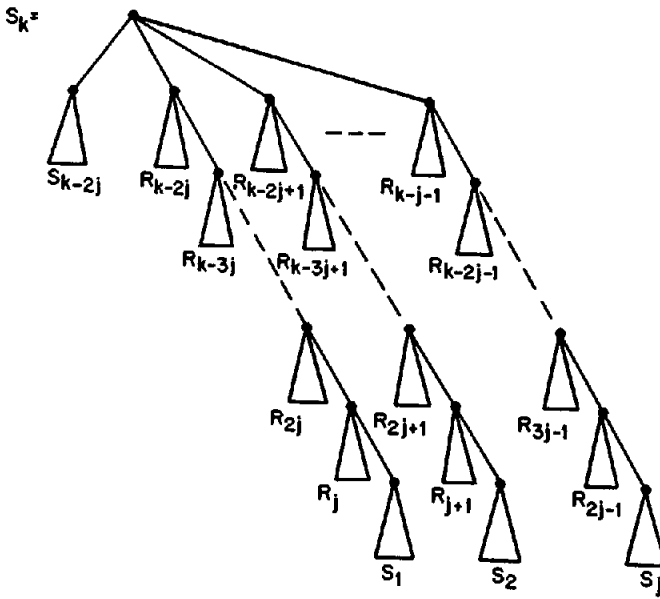
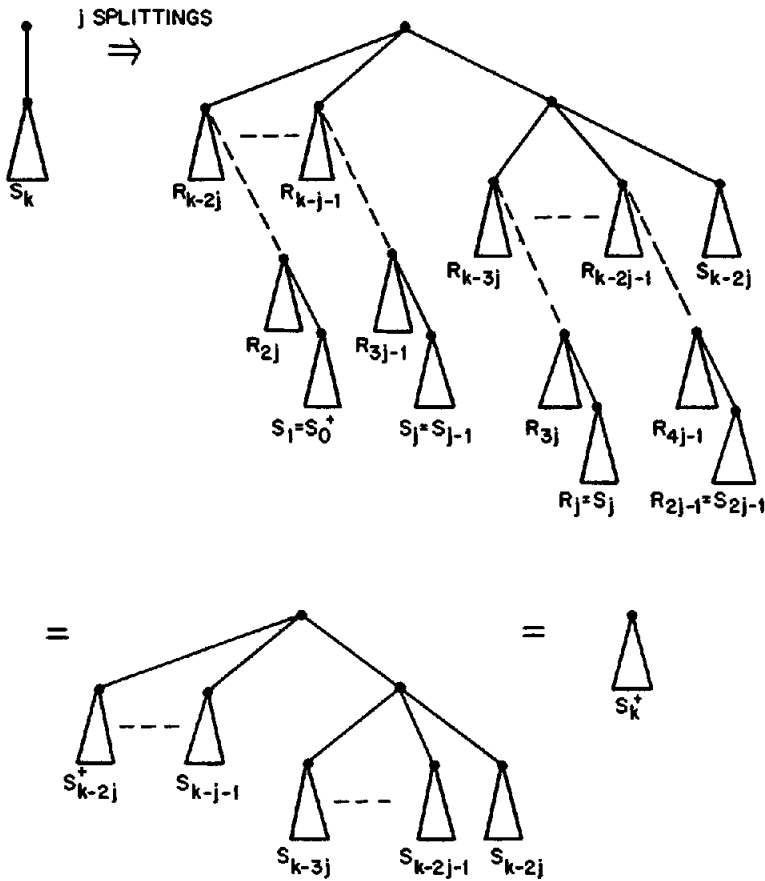FIG. 16.   Vertical unrolling of S_k for $k = ij$.

FIG. 17.   Self-reproduction of S_k for $k = ij$, i even, S_k^+ denotes S_k with an extra node.
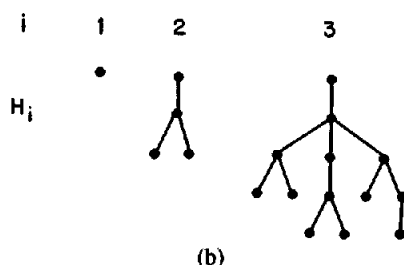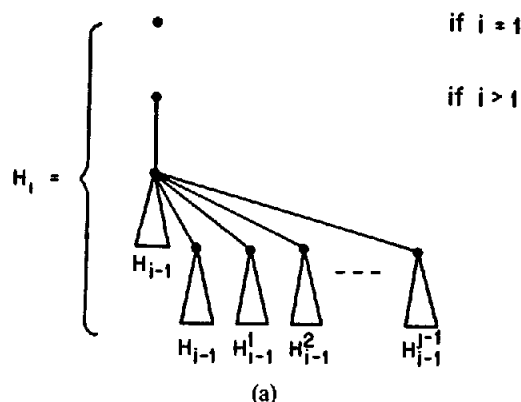
FIG. 18. H_i trees. (a) Recursive definition of H_i. (b) Examples of H_i for $J = 2$.

linking using T_k trees. Thus we obtain an $\Omega(m \log_{(1+m/n)}n)$ lower bound for the total number of nodes on find paths. The following theorem summarizes the situation for splitting with naive union:

THEOREM 5.   *The set union algorithm with splitting and naive union runs in* $\Theta((n + m)\log_{(2+m/n)}(min\{m, n\}))$ *time.*

We conclude this section by defining a class of trees self-reproducing under halving. Let $j \geq 1$. For $i \geq 1$, we shall define a tree H_i with the following property: If we perform j halvings on H_i, each on a path of $2i - 1$ nodes, and then link a single-node tree with the halved tree, the result is a new H_i tree with an extra node. The extra node is the starting node of the first path that was halved; this node is a leaf in the original H_i tree. We use H_i^h to denote H_i after the first h of the j self-reproducing halvings has been carried out.

We define H_i inductively, simultaneously proving the self-reproducing property. H_1 is a single-node tree; its self-reproduction is obvious. For $i > 1$, suppose H_{i-1} is defined and known to be self-reproducing. We define H_i to consist of a root and a single subtree formed by linking H_{i-1} with $H_{i-1}^1, H_{i-1}^2, \ldots, H_{i-1}^{j-1}$. (See Figure 18.) If we perform j halvings on H_i, one starting in each of the subtrees $H_{i-1}, H_{i-1}^1, H_{i-1}^2, \ldots, H_{i-1}^{j-1}$ linked to H_{i-1}, we produce a tree consisting of a root and subtrees $H_{i-1}, H_{i-1}^1, H_{i-1}^2, \ldots, H_{i-1}^{j}$. But the tree consisting of a root and a single subtree H_{i-1}^j is H_{i-1} with an extra node by the induction hypothesis. Thus, if we unite a single node tree with the halved tree, we obtain H_i with an extra node, and H_i is self-reproducing. (See Figure 19.) The size of H_i is at most $(j + 1)^i$. We can obtain bad examples for halving and naive linking using H_i as we did for compression and naive linking using T_k, except that in each cycle of j finds and a
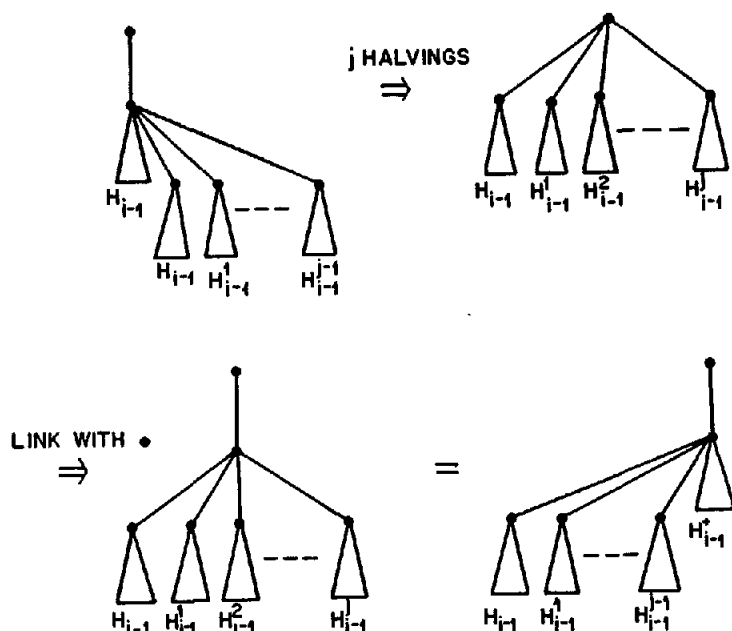
FIG. 19.   Self-reproduction of H_i. H_{i-1}^+ denotes H_{i-1} with an extra node.

link, the link follows the finds. Thus we obtain a lower bound of $\Omega(m \log_{(1+m/n)}n)$ for the total number of nodes on find paths. The following theorem summarizes what we know about halving with naive linking:

THEOREM 6.   *The set union algorithm with halving and naive union runs in* $O((n + m)\log_{(2+m/n)} (\min\{m, n\}))$ *time and in* $\Omega(n + m \log_{(2+m/n)}n)$ *time. (These bounds match for* $m \geq n$.)

4. Reversal

Van Leeuwen and van der Weide [13] proposed another one-pass find method, called *reversal*. Although reversal is superficially appealing, we shall see that it is not as efficient as the methods studied in Section 3. Reversal is really a class of methods rather than a single method. A *reversal of type zero* is performed on a find path by making every node on the path point to the *first* node on the path. (See Figure 20a.) Note that this changes the canonical element of the set. For any integer $k \geq 1$, a *reversal of type* k is performed by making the first node and the last k nodes on the path point to the last node, and making the remaining nodes point to the first node. Figures 20b and c illustrate type one reversal and type two reversal, respectively. Van der Weide [14] called type three reversal *node transportation*. We can use any type of reversal in combination with any linking rule.

As k increases, type k reversal approximates compression more and more closely, but for no fixed k is type k reversal as efficient as compression. We shall analyze the efficiency of types zero, one, and two, and leave the analysis of type k for $k \geq 3$ as an open problem. We begin our analysis of reversal by showing that both type zero and type one use $O(n + m \log n)$ time for any linking method. Van Leeuwen and van der Weide [13] observed that if we reverse a path of $i + 1$ nodes in a B_i tree, the result is a new B_i tree. (See Figure 21.) This is true for both type zero and type one, and implies a lower bound of $\Omega(n + m \log n)$ for either of these algorithms with any linking method, since we can build a B_i tree with any linking method.
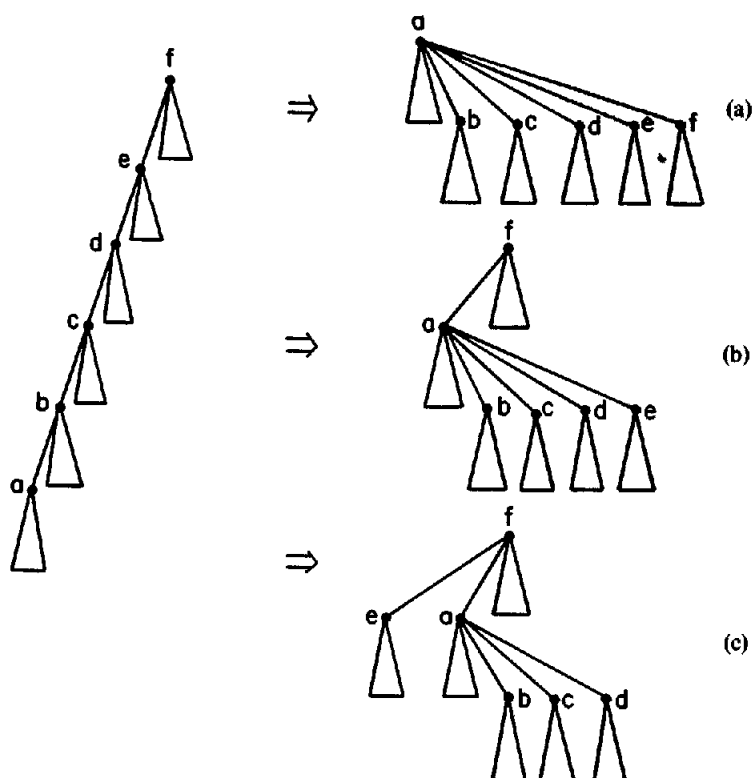
FIG. 20. Reversal. (a) Type zero. (b) Type one. (c) Type two.

To obtain a corresponding upper bound, we use a version of the multiple-partition method much like that used to analyze compression with naive linking. Our bound of $O(n + m \log n)$ improves van Leeuwen and van der Weide's bound of $O((n + m)\log(n + m))$.

LEMMA 10. *In any sequence of set operations implemented using type zero or type one reversal and any linking method, the total number of nodes on find paths is $O(n + m \log n)$.*

PROOF. We shall prove that if we start with an arbitrary n-node forest and perform an arbitrary sequence of at most $n - 1$ links and $m \le n$ intermixed finds, then the total number of nodes on find paths is at most $4m\lceil\log n\rceil + 7m + n$. This implies the lemma. (Apply the bound repeatedly to groups of n consecutive operations.)

We first consider type zero reversal. For each node in the forest, we define a *rank*, which may change as the links and finds are carried out. (This new definition of rank is for analytical purposes only and does not affect the implementation of the linking by rank rule.) Initially, the rank of a node is its height in the original forest. When a link operation causes a tree root r to become the parent of another tree root s, we redefine the rank of r to be the maximum of its old value and one more than the rank of s. When a find operation begins at a node e and returns a node r, we redefine the rank of e to be one more than the rank of r.

With this definition, ranks always strictly increase along any path in the forest, and every rank is in the range $[0, 3n - 2]$. (The original ranks are in the range $[0, n - 1]$, and a set operation can increase the maximum rank by at most one.) We
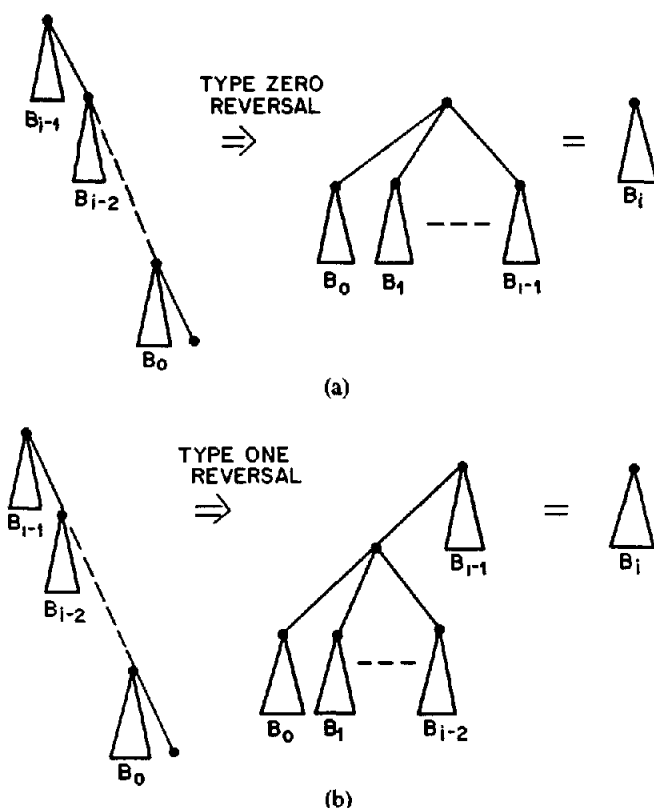
Fig. 21. Self-reproduction of B_i by reversal. (a) Type zero reversal. (b) Type one reversal.

define the *level* of a node x as in the proof of Lemma 9 to be zero if $p(x) = x$ and the maximum value of i such that $rank(p(x)) - rank(x) \geq 2^{i-1}$ if $p(x) \neq x$. If $level(x) \neq 0$, then the rank of x cannot change, and the level of x cannot decrease, until a find of x occurs.

We define a node to be *active* it it starts or ends a find path and *passive* otherwise; there are at most $2m$ active nodes. Once a node x has an active parent, it retains an active parent. We charge for a find almost exactly as in Lemma 9:

Find Charging. Charge $\lceil \log n \rceil + 4$ to the find.

Active Node Charging. If x is an active node of level i that is not the first node on the find path and is followed somewhere on the path by another node of level i, charge one to x.

Passive Node Charging. If x is a passive node, charge one to x.

Since all levels are in the range $[0, \lceil \log n \rceil + 2]$, the total charge for a find is at least the number of nodes on the find path. Charging a passive node causes one of its children to acquire an active parent. This can happen at most n times. Charging an active node causes its level to increase by at least one. This can happen at most $\lceil \log n \rceil + 1$ times before a find on the node occurs. Thus the total charge to active nodes is $3m(\lceil \log n \rceil + 1)$. The total charge to finds is $m(\lceil \log n \rceil + 4)$, giving a grand total charge of $4m\lceil \log n \rceil + 7m + n$.
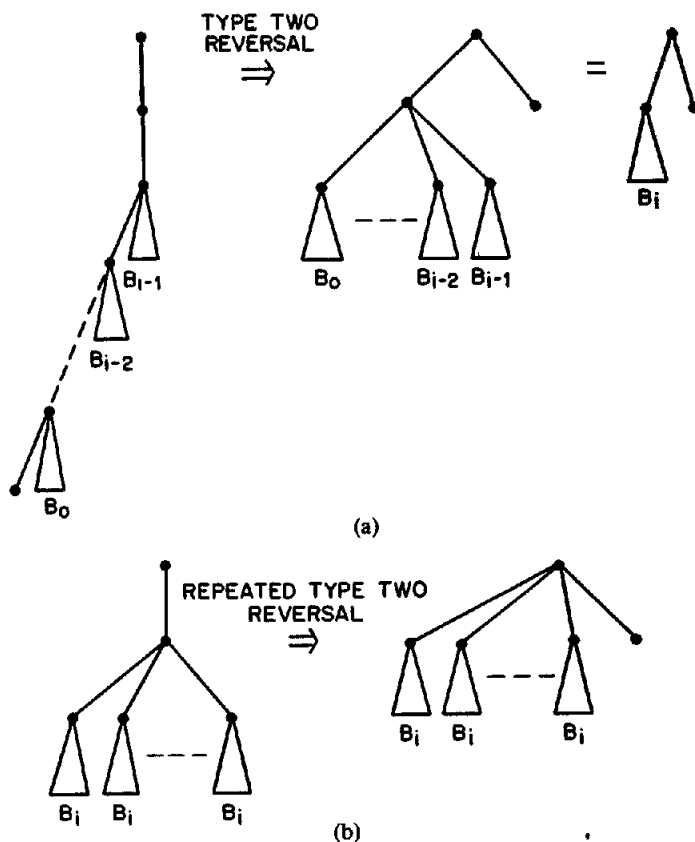
TYPE TWO
REVERSAL
$\Rightarrow$

(a)

REPEATED TYPE TWO
REVERSAL
$\Rightarrow$

(b)

FIG. 22. Effect of type two reversal on a collection of linked B_i trees. (a) Reversal on a single B_i tree. (b) Reversal on several B_i trees.

Exactly the same proof works for type one reversal, if we change the definition of rank as follows: When a find of a node e returning a node r is performed, we redefine the rank of e to be the old rank of r and redefine the rank of r to be one more than its previous value. We obtain the same upper bound of $4m\lceil\log n\rceil + 7m + n$ on the total number of nodes on find paths, assuming $m \leq n$. □

THEOREM 7. *The set union algorithm with reversal of type zero or one and naive linking, linking by rank, or linking by size runs in $O(n + m \log n)$ time.*

Type two reversal has more interesting behavior than either type zero or type one. With naive linking, the method runs in $O(n + m \log(2 + n^2/(n + m)))$ time; with either linking by rank or linking by size, the method runs in $O(n + m \log(2 + n \log n/(n + m)))$ time. As with the other types of reversal, we begin with the lower bounds. Figure 22a illustrates that if we perform a type two reversal on a tree consisting of a root, a single child, and a single B_i tree at depth 2, we obtain a tree in which the B_i tree has moved to depth one. More generally, if we form a tree consisting of a root and j subtrees, each a copy of B_i, we can reproduce this tree by linking it with a single-node tree and performing j type two reversals, one on each B_i tree. Each path reversed contains $i + 3$ nodes. (See Figure 22b.)

We obtain bad examples for type two reversal with naive linking as follows: For any $m \geq n \geq 3$, let $j = \lfloor m/n \rfloor$. Suppose $n/2j \geq 1$ (otherwise $n^2/(n + m) = O(1)$) and
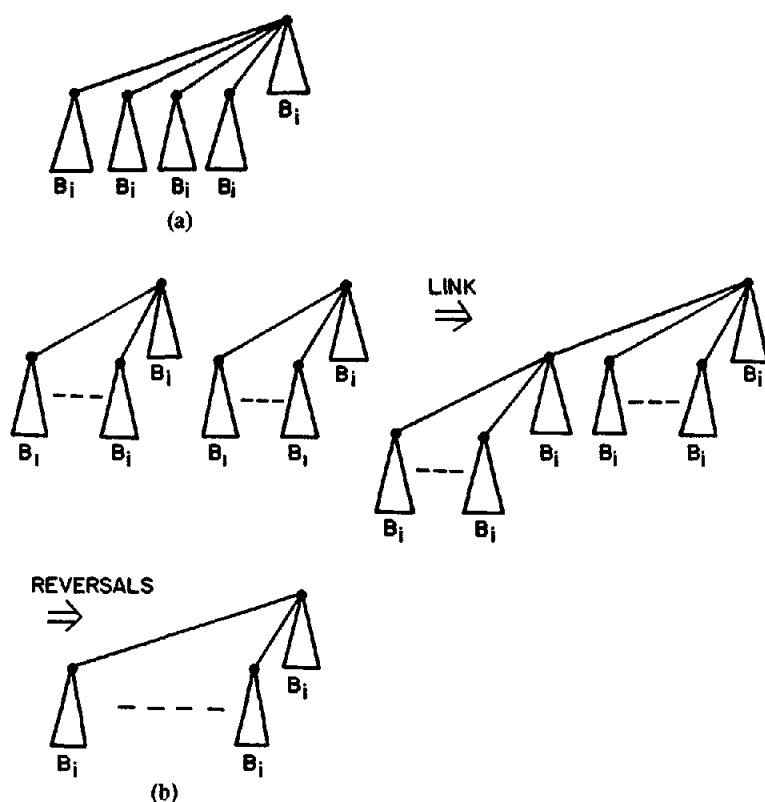
FIG. 23. Bad examples for type two reversal with linking by rank or size.
(a) Initial trees for $j = 5$. (b) Typical link followed by reversals.

let $i = \lfloor\log(n/2j)\rfloor$. Build a tree consisting of a root and j subtrees, each a B_i tree. This tree contains at most $\lfloor n/2\rfloor + 1$ nodes. Repeat the following operations $\lceil n/2\rceil - 1$ times. Link a single-node tree with the existing tree (which is isomorphic to the original tree with some extra nodes) and perform j type two reversals, each on a path of $i + 3$ nodes, to reproduce the original tree with some extra nodes. The total number of nodes on find paths is at least

$$j(\lceil n/2\rceil - 1)(i + 3) = \Omega\!\left(m \log\left(\frac{n^2}{n + m}\right)\right).$$

For $n \geq 3$ but $m < n$, the same example with $j = 1$ and at most m finds gives a lower bound of $\Omega(m \log n)$; a lower bound of $\Omega(n)$ is obvious.

The way we obtain bad examples for type two reversal with linking by rank or size is similar but a little more complicated. For any $m \geq n \geq 64$, let $j = \lfloor m/n\rfloor$. Suppose $\log n/j \geq 1$ (otherwise $n \log n/m = O(1)$) and let $i = \lfloor\log(\log n/j)\rfloor$, $k = \lfloor\log(n/\log n)\rfloor$. Since $n \geq 64$, $k \geq 3$. By means of links by rank or size, build $j2^k$ copies of B_i. These trees contain no more than $j2^{i+k} \leq n$ nodes. Link the B_i trees in groups of j using linking by rank or size, giving 2^k trees, each consisting of B_i linked with $j - 1$ copies of B_i. (See Figure 23a.) Repeat the following operations until only one tree is left: Link the existing trees in pairs; then perform a type two reversal on each of the B_i trees of depth two. (See Figure 23b.) Each tree existing after a sequence of reversals consists of a B_i tree linked with a number of copies of

B_i. Each reversal is on a path of $i + 3$ nodes. The total number of reversals is

$$\sum_{h=0}^{k-1} 2^{k-h-1}(2^h j - 1) = k2^{k-1}j - 2^k + 1 \leq m.$$

The total number of nodes on find paths is

$$(i + 3)(k2^{k-1}j - 2^k + 1) \geq (i + 3)(kj - 2)2^{k-1} = \Omega\left(m \log\left(\frac{n \log n}{m}\right)\right).$$

For $n \geq 64$ but $m < n$, the same example with $j = 1$ and at most m finds gives a lower bound of $\Omega(m \log \log n)$; a lower bound of $\Omega(n)$ is obvious.

To derive upper bounds for type two reversal, we use almost the same proof as in Lemma 10.

LEMMA 11. *In any sequence of set operations implemented using naive linking and type two reversal, the total number of nodes on find paths is $O(n + m \log(2 + n^2/(n + m)))$. In any sequence of set operations implemented using linking by rank or size and type two reversal, the total number of nodes on find paths is $O(n + m \log(2 + n \log n/(n + m)))$.*

PROOF. We define the rank of a node as in the proof of Lemma 10, with the following difference: When a find operation begins at a node e and returns a node r, we redefine the rank of e to be one less than the rank of r. As in the proof of Lemma 10, this new definition of rank does not affect the implementation of the linking by rank rule. With this definition, ranks always strictly increase along any path in the forest. With naive linking, all ranks remain in the range $[0, n - 1]$; with linking by size or rank, all ranks remain in the range $[0, \lfloor \log n \rfloor]$. We define the level of a node as in the proofs of Lemmas 9 and 10 and active and passive nodes as in the proof of Lemma 10; thus there are at most $2m$ active nodes. We charge for a find using the following rules, where k is a parameter to be chosen later:

Find Charging. Charge $k + 3$ to the find.

Active Node Charging. If x is an active node of level i that is not the first node on the find path and is followed by another node of level at least $\min\{i, k + 1\}$, charge one to x.

Passive Node Charging. If x is a passive node, charge one to x.

Consider any find. A node on the find path that is not charged for the find must be either the first, the last of a level i in the range $[0, k]$, or the last with level at least $k + 1$. Thus, there are at most $k + 3$ nodes not charged, and the total charge for the find is at least the number of nodes on the find path.

The charge to passive nodes is at most one per passive node, for a total of at most n. An active node can be charged at most once per level for each level in the range $[1, k]$ before a find on the node occurs; this gives a total charge of $3mk$ to active nodes on levels in the range $[1, k]$. Each time an active node is charged on level $k + 1$ or higher, the rank of its parent increases by at least 2^k. This can happen at most $r/2^k$ times, where r is the maximum rank of any node. Thus the total charge is at most $3mk + n(r/2^k) + n$, since the number of active nodes is at most n.

With naive linking, $r \leq n - 1$. If we choose $k = \log(2 + n^2/(n + m))$, the total charge is at most

$$3m \log\left(2 + \frac{n^2}{n + m}\right) + \frac{n + m}{4} + n = O\left(n + m \log\left(2 + \frac{n^2}{n + m}\right)\right).$$

With linking by rank or size, $r \leq \log n$, and if we choose $k = \log(2 + n \log n/(n + m))$, the total charge is at most

$$3m \log\left(2 + \frac{n\log n}{n + m}\right) + \frac{(n + m)}{4} + n = O\left(n + m \log\left(2 + \frac{n\log n}{n + m}\right)\right). \quad \square$$

THEOREM 8. *The set union algorithm with naive linking and type two reversal runs in $\Theta(n + m \log(2 + n^2/(m + n)))$ time. The algorithm with linking by rank or size and type two reversal runs in $\Theta(n + m \log(2 + n \log n/(n + m)))$ time.*

5. *Collapsing and Splicing*

Yet another way to speed up the set union algorithm is to spend more than $O(1)$ time on each *link* operation, in the hope of saving time on finds. The most extreme way of doing this is *collapsing*: To link two trees, we make every node in one tree point to the root of the other. (See Figure 24.) Collapsing is well known; a discussion of it appears in [2]. We can use collapsing with naive linking, linking by rank, or linking by size. (Note that with linking by rank, the rank of the root of a tree bears no relation to its depth, which is always zero or one.) Collapsing causes each find to take $O(1)$ time (each find path contains only one or two nodes), but a link takes time proportional to the size of one of the sets being linked. The following theorem is easy to prove:

THEOREM 9 [2]. *The set union algorithm with collapsing and naive linking runs in $\Theta(n^2 + m)$ time. The algorithm with collapsing and linking by rank or size runs in $\Theta(n \log n + m)$ time.*

Although superficially appealing, collapsing has two serious drawbacks that make it inferior to any form of compaction with respect to both space and time. First, it requires two pointers per node rather than one, since each set must be represented as a linked list. (Circular linking is best; see Figure 25.) Second, the algorithm with collapsing always performs at least as many parent pointer assignments as any form of compaction, as the following theorem shows:

THEOREM 10. *Let algorithm A_1 be the set union algorithm with collapsing and any linking method, and let algorithm A_2 be the set union algorithm with any form of compaction and the same linking method as A_1. Then on any sequence of set operations, algorithm A_1 performs at least as many assignments to parent pointers as algorithm A_2.*

PROOF. For any sequence of set operations, define the *reference forest* to be the forest whose nodes are the elements, such that element e is the parent of element f if and only if a link operation makes e the parent of f. Algorithm A_1 performs an assignment $p(x) = y$ for every pair of nodes x and y such that y is a proper ancestor of x in the reference forest; algorithm A_2 performs an assignment $p(x) = y$ only if y is a proper ancestor of x in the reference forest. $\square$

The final set union algorithms we shall study are based on an idea of Rem [4]. Rem's idea was to combine the two finds and the link used to carry out a *unite*
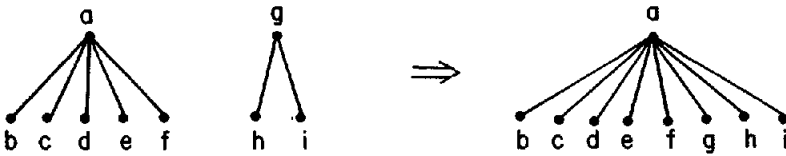
FIG. 24.   Collapsing during *link* (*a*, *g*).
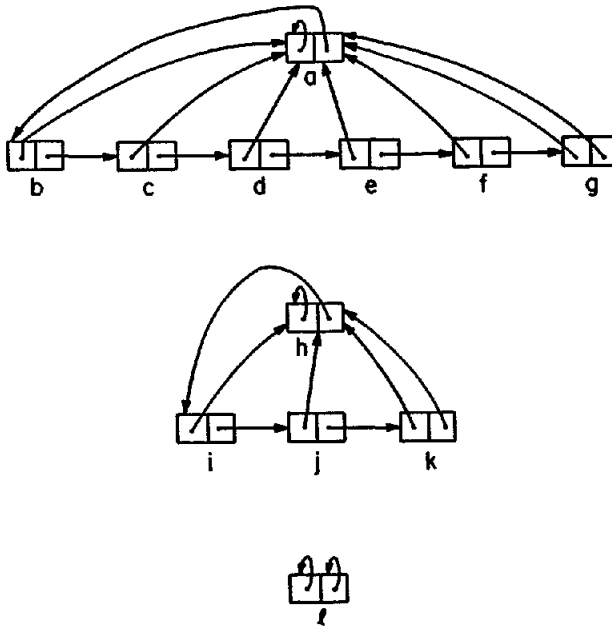
FIG. 25.   Representation of sets {*a*, *b*, *c*, *d*, *e*, *f*, *g*}, {*h*, *i*, *j*, *k*}, {*l*} using collapsing data structure. Each node has a *parent* and a *next* field; *rank* and *label* fields are omitted in the figure.

into a single operation that scans the two find paths simultaneously, restructuring the forest in the process. The most natural way to present Rem's method is as a solution to a variant of the set union problem that we call the *contingent union problem*: Carry out an intermixed sequence of two kinds of operations on unlabeled sets:

make set(*e*): Create a new set containing the single element *e*. This operation requires that *e* initially be in no set.

contingently unite(*e*, *f*): If elements *e* and *f* are in the same set, return **false**. Otherwise combine the sets containing *e* and *f* into a single set and return **true**.

The following procedure implements contingent union using two finds and a link (we must modify *link* to avoid updating set labels):

```
predicate contingently unite(e, f);
    local r, s:
    r := find(e);
    s := find(f);
    if r = s → return false
    [] r ≠ s → link(r, s);return true
    fi
end contingently unite;
```
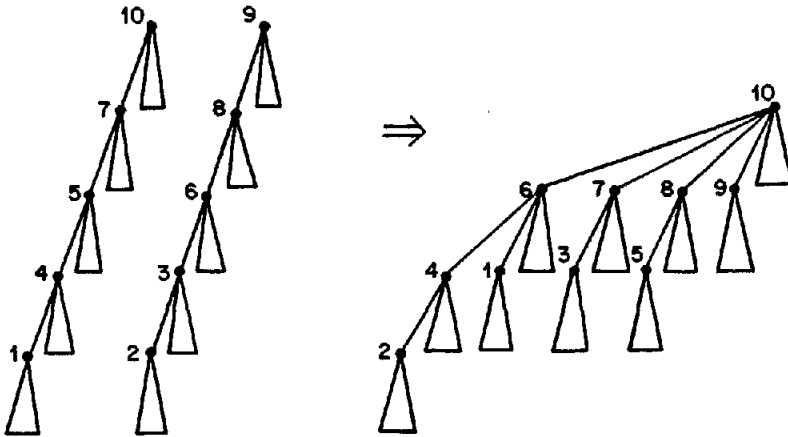
FIG. 26.   Contingent union of nodes 1 and 2 by splicing.

We shall call Rem's solution to the contingent union problem *naive splicing*. We assume that the elements are totally ordered in an arbitrary way. (We can impose such an ordering by numbering the elements from 1 to n.) To carry out *contingently unite*(e, f), we scan the two find paths concurrently, taking a step in one path at a time. If x and y are the current nodes on the two paths, we take a step by comparing $p(x)$ and $p(y)$. If $p(x) = p(y)$, we stop and return **false**. Otherwise, we make the node with the smaller parent, say z, point to the larger parent and replace it by its old parent. If z did not change (we are at a root), we stop and return **true**. (See Figure 26.) Note that splicing maintains the property that $p(x) \geq x$ for all nodes x. The following procedure implements contingent union with naive splicing:

predicate *contingently unite*(e, f);
  local x, y, z;
  $x, y := e, f$;
  do $p(x) = p(y) \rightarrow$ **return false**
  ☐ $p(x) < p(y) \rightarrow$
      $x, p(x), z := p(x), p(y), x$;
      if $x = z \rightarrow$ **return true fi**
  ☐ $p(x) > p(y) \rightarrow$
      $y, p(y), z := p(y), p(x), y$;
      if $y = z \rightarrow$ **return true fi**
  od
end *contingently unite*;

In our analysis of naive splicing, we assume that n is the number of elements that are ever parameters to contingent union operations and that m is the number of contingent unions; thus $m \geq n/2$. The running time of naive splicing is $\Theta(m \log_{(2+m/n)} n)$, making it asymptotically as fast as compaction with naive linking. We obtain the lower bound by noting that if e and f are different tree roots, splicing does the same thing as naive linking; if f is the root of the tree containing e, then splicing compresses the path from e to f. Thus the bad examples constructed in Section 3 for compression with naive linking work for splicing, and we obtain an $\Omega(m \log_{(2+m/n)} n)$ lower bound. (This bound generalizes the $\Omega(n \log n)$ lower bound of van Leeuwen and van der Weide [11].) To obtain the upper bound, we use a complicated version of the multiple-partition method.

LEMMA 12. *In any sequence of contingent unions implemented with naive splicing, the total number of node visits is $O(m \log_{(2+m/n)} n)$, where we call a node visited if it has its parent changed by the splice.*

PROOF. To each node x we assign a (permanent) rank in the range $[0, n - 1]$ corresponding to the position of x in the total order of nodes. For each level $i \in [O, \lceil \log_{2+m/n} n \rceil]$ we define a partition on the ranks whose blocks are

$$block(i, j) = [j\lfloor 2 + m/n \rfloor^i, (j + 1)\lfloor 2 + m/n \rfloor^i - 1]$$

for $j \geq 0$. If x and y are a pair of nodes such that $rank(x) < rank(y)$, we define $level(x, y)$ to be the minimum value of i such that $rank(x)$ and $rank(y)$ are in the same block of the level i partition. Then $level(x, y) \in [1, \lceil \log_{2+m/n} n \rceil]$. (This partition is the same as the partition used in Lemma 7 to analyze compaction with naive linking.)

Consider a splice of nodes e and f. Let $x_0, x_1, \ldots, x_k$ be the nodes visited during the splice, in increasing order by rank. These nodes comprise part or all of the paths from e and f to the roots of their respective trees. In order to charge for the splice, we define a *layer* for each node x_i by $layer(x_k) = 0$, $layer(x_i) = level(x_i, x_{i+1})$ for $i \in [0, k - 1]$. We charge according to the following rules:

Node Charging. For each node x_i such that $i \geq 3$ and

$$layer(x_i) \geq \max\{layer(x_{i-3}), layer(x_{i-2}), layer(x_{i-1})\},$$

charge $layer(x_i) - \max\{layer(x_{i-3}), layer(x_{i-2}), layer(x_{i-1})\} + 1$ to either x_{i-3}, x_{i-2}, or x_{i-1} as follows, where p is as defined before the splice:

(a) If $p(x_{i-1}) = x_i$ (x_{i-1} and x_i are on the same path), then charge x_{i-1}.
(b) Otherwise, if $p(x_{i-2}) = x_i$ (x_{i-2} and x_i are on one path and x_{i-1} is on the other), then charge x_{i-2}.
(c) Otherwise (x_{i-2} and x_{i-1} are on one path and x_i is on the other), charge x_{i-3}.

Splice Charging. Charge $1 + \lceil \log_{2+m/n} n \rfloor$ to the splice.

Remark. Case (c) of the node charging rule is really two cases, which we shall have to distinguish in the analysis:

(ci) $p(x_{i-3}) = x_i$ (x_{i-3} and x_i are on one path and x_{i-2} and x_{i-1} are on the other).
(cii) $p(x_{i-3}) \neq x_i$ (x_{i-3}, x_{i-2}, and x_{i-1} are on one path and x_i is on the other).

In order to relate the charge for a splice to the number of node visits, let $y_0, y_1, \ldots, y_h$ be the subsequence of $x_0, x_1, \ldots, x_k$ containing the node of maximum layer among each consecutive triple x_i, x_{i+1}, x_{i+2}, for $i \in [0, k - 2]$, breaking ties by choosing the last node of maximum layer. Note that a node y_j can be maximum in up to three triples x_i, x_{i+1}, x_{i+2}.

If $k \geq 3$, the y-subsequence contains at least one-third of the nodes in the x-sequence. If $P = \{j \mid j \in [0, h - 1]$ and $layer(y_j) \leq layer(y_{j+1})\}$, the first part of the proof of Lemma 3 shows that

$$\sum_{j \in P} (layer(y_{j+1}) - layer(y_j) + 1) + \lceil \log_{2+m/n} n + 1 \rceil \geq h + 1 \geq (k + 1)/3.$$

Thus, if we can show that every node y_{j+1} with $layer(y_{j+1}) \geq layer(y_j)$ generates a charge of at least $layer(y_{j+1}) - layer(y_j) + 1$, we can conclude that the total charge for a splice is at least one-third the number of node visits.

Suppose $layer(y_{j+1}) \geq layer(y_j)$. The first node x_i following y_j in the x-sequence and satisfying $layer(x_i) \geq layer(y_j)$ must have maximum layer among x_{i-2}, x_{i-1}, x_i, since none has layer greater than $layer(x_i)$. Thus $x_i = y_{j+1}$. The set of three consecutive nodes among which y_j has maximum layer cannot include x_i by the tie-breaking rule. Thus,

$$layer(y_{j+1}) = layer(x_i) \geq \max\{layer(x_{i-3}), layer(x_{i-2}), layer(x_{i-1})\} = layer(y_j),$$

and y_{j+1} generates a node charge of $layer(y_{j+1}) - layer(y_j) + 1$ as desired.

It remains for us to bound the total charge. Consider the charge generated by a node x_i. Suppose that $layer(x_i) \geq \max\{layer(x_{i-3}), layer(x_{i-2}), layer(x_{i-1})\}$. In case (a), $(p(x_{i-1}) = x_i)$, $level(x_{i-1}, p(x_{i-1}))$ increases because of the splice by at least the amount charged minus one. (The new parent of x_{i-1} has rank greater than or equal to that of x_{i+1}.) In case (b), $(p(x_{i-2}) = x_i)$, $level(x_{i-2}, p(x_{i-2}))$ increases by at least the amount charged minus one, and in case (ci), $(p(x_{i-3}) = x_i)$, $level(x_{i-3}, p(x_{i-3}))$ increases by at least the amount charged minus one. The total charge for all such cases is at most twice the number of levels times the number of nodes, or $2n\lceil 1 + \log_{\lfloor 2+m/n\rfloor} n\rceil$. In case (cii), $(p(x_{i-3}) = x_{i-2}$ and $p(x_{i-2}) = x_{i-1})$, $level(x_{i-3}, p^2(x_{i-3}))$ increases by at least the amount charged minus one. The total charge for this case is also at most $2n\lceil 1 + \log_{\lfloor 2+m/n\rfloor} n\rceil$. (The new grandparent of x_{i-3} has rank greater than or equal to that of x_{i+1}.)

We must also account for the charge generated if $layer(x_i) = \max\{layer(x_{i-3}),$ $layer(x_{i-2}), layer(x_{i-1})\}$. Let $h = layer(x_i)$. In case (a), the splice causes $rank(p(x_{i-1}))$ to move from one level $h - 1$ block to another. In case (b), the splice causes $rank(p(x_{i-2}))$ to move from one level $h - 1$ block to another, and in case (ci), the splice causes $rank(p(x_{i-3}))$ to move from one level $h - 1$ block to another. If x is any node, $p(x)$ can be in at most $\lfloor 1 + m/n\rfloor$ level $h - 1$ blocks before $level(x, p(x))$ $> h$. Thus the total charge in all such cases is at most the number of levels times the number of nodes times $\lfloor 2 + m/n\rfloor$, or at most $(m + 2n)\lceil 1 + \log_{\lfloor 2+m/n\rfloor} n\rceil$. A similar argument using $level(x, p^2(x))$ shows that the total charge in case (cii) if $layer(x_i) = \max\{layer(x_{i-3}), layer(x_{i-2}), layer(x_{i-1})\}$ is also at most $(m + 2n)\lceil 1 + \log_{\lfloor 2+m/n\rfloor} n\rceil$.

The total charge to splices is $m\lceil 1 + \log_{\lfloor 2+m/n\rfloor} n\rceil$. Thus the grand total charge is at most $(3m + 8n)\lceil 1 + \log_{\lfloor 2+m/n\rfloor} n\rceil$. □

THEOREM 11. *The contingent union algorithm with naive splicing runs in* $\Theta(m$ $log_{(2+m/n)} n)$ *time.*

We conclude that Rem's algorithm solves the contingent union problem as efficiently as compression, splitting, or halving with naive union. The disadvantage of Rem's algorithm is that it does not use linking by rank or size and thus is not asymptotically optimal. However, we can obtain an asymptotically optimal algorithm by combining splicing with linking by rank, resulting in a method that we shall call *splicing by rank.* When making a set containing the single element e, we initialize the rank of e to be zero. To carry out *contingently unite*(e, f), we scan the two find paths concurrently, taking a step in one path at a time. If x and y are the current nodes in the two paths, we take a step by comparing $rank(p(x))$ and $rank(p(y))$. If $rank(p(x)) < rank(p(y))$, we simultaneously replace x by $p(x)$ and $p(x)$ by $p(y)$; if this does not change x, we stop and return **true**. The case $rank(p(x))$ $> rank(p(y))$ is symmetric. If $rank(p(x)) = rank(p(y))$, what we do depends upon $x, p(x), y,$ and $p(y)$. If $p(x) = p(y)$, we stop and return **false**. If $p(x) \neq p(y)$ and $x \neq p(x)$, we replace x by $p(x)$. If $p(x) \neq p(y)$ and $y \neq p(y)$, we replace y by $p(y)$.

Finally, if $p(x) \neq p(y)$, but $x = p(x)$ and $y = p(y)$, we replace $p(x)$ by $p(y)$, add one to the rank of y, and stop, returning **true**. The following procedure implements contingent union with splicing by rank.

```
predicate contingently unite(e, f);
    local x, y, z;
    x, y := e, f;
    do rank(p(x)) < rank(p(y)) →
        x, p(x), z := p(x), p(y), x;
        if x = z → return true fi
    □ rank(p(x)) > rank(p(y)) →
        y, p(y), z := p(y), p(x), y;
        if y = z → return true fi
    □ rank(p(x)) = rank(p(y)) →
        if p(x) = p(y) → return false fi;
        if x ≠ p(x) → x := p(x)
        □ y ≠ p(y) → y := p(y)
        □ x = p(x)    and    y = p(y) →
            p(x) := p(y); rank(y) := rank(y) + 1; return true
        fi
    od
end contingently unite;
```

Splicing by rank maintains the invariant that $rank(x) < rank(p(x))$ for each node x such that $x \neq p(x)$. We can define the reference forest for a sequence of *make set* and *contingently unite* operations as we did in Section 3: The parent of a node x in the reference forest is the first value other than x assigned to $p(x)$ by the algorithm. With this definition, Lemma 2 holds for the reference forest, if we take the rank of a node x to be the last value assigned to $rank(x)$ by the algorithm. Combining the analysis in Section 3 for compaction and linking by rank with the ideas in the proof of Lemma 12, we can obtain a bound of $O(m\alpha(m + n, n))$ for splicing by rank.

THEOREM 12. *The contingent union algorithm with splicing by rank runs in* $\Theta(m\alpha(m + n, n))$ *time.*

PROOF. Exercise. □

Splicing by rank has another property worth noting.

THEOREM 13. *If every contingent union operation returns* **true**, *then the contingent union algorithm with splicing by rank runs in* $O(n)$ *time.*

PROOF. There are at most $n - 1$ contingent unions. Consider the last one, combining trees with roots r and s. The time for this union is $O(\min\{rank(r), rank(s)\}) = O(\log(\min\{size(r), size(s)\}))$. If $t(n)$ is the total time as a function of n, we obtain the recurrence

$$t(1) = O(1);$$
$$t(n) = \max_{1 \leq i \leq n} \{t(i) + t(n - i) + O(\log(\min\{i, n - i\}))\} \quad \text{if } n > 1.$$

This recurrence has the solution $t(n) = O(n)$. □

6. Remarks

In this paper we have analyzed a total of twenty-six algorithms for the set union problem. Tables I and II list the asymptotic running times of the algorithms for

TABLE I.   WORST-CASE RUNNING TIMES OF SET UNION ALGORITHMS IF $m \geq n$

	Naive linking	Linking by rank or size
Naive find	$\Theta(mn)$	$\Theta(m \log n)$
Compression	$\Theta(m \log_{1 + m/n} n)$	$\Theta(m\alpha(m,n))$
Splitting	$\Theta(m \log_{1 + m/n} n)$	$\Theta(m\alpha(m,n))$
Halving	$\Theta(m \log_{1 + m/n} n)$	$\Theta(m\alpha(m,n))$
Type zero reversal	$\Theta(m \log n)$	$\Theta(m \log n)$
Type one reversal	$\Theta(m \log n)$	$\Theta(m \log n)$
Type two reversal	$\Theta(m \log(2 + n^2/m))$	$\Theta\left(m \log\left(2 + \dfrac{n \log n}{m}\right)\right)$
Collapsing	$\Theta(m + n^2)$	$\Theta(m + n \log n)$
Naive splicing	$\Theta(m \log_{1 + m/n} n)$	—
Splicing by rank	—	$\Theta(m\alpha(m,n))$

TABLE II.   WORST-CASE RUNNING TIMES OF SET UNION ALGORITHMS IF $m < n$

	Naive linking	Linking by rank or size
Naive find	$\Theta(mn)$	$\Theta(n + m \log n)$
Compression	$\Theta(n + m \log n)$	$\Theta(n + m\alpha(n,n))$
Splitting	$\Theta(n \log m)$	$\Theta(n + m\alpha(n,n))$
Halving	$\Omega(n + m \log n), O(n \log m)$	$\Theta(n + m\alpha(n,n))$
Type zero reversal	$\Theta(n + m \log n)$	$\Theta(n + m \log n)$
Type one reversal	$\Theta(n + m \log n)$	$\Theta(n + m \log n)$
Type two reversal	$\Theta(n + m \log n)$	$\Theta(n + m \log \log n)$
Collapsing	$\Theta(n^2)$	$\Theta(n \log n)$
Naive splicing	$\Theta(n + m \log m)$	—
Splicing by rank	—	$\Theta(n + m\alpha(m,m))$

$m \geq n$ and $m < n$, respectively. (In most applications $m \geq n$.) Seven of the algorithms are asymptotically optimal: compression, splitting, or halving combined with union by rank or size, and splicing by rank. The remaining methods are less efficient; most run in about $\Theta(\log n)$ time per find. Our analysis has displayed the power of the multiple-partition method for deriving upper bounds and of self-reproducing trees for giving worst-case examples. Two intriguing open problems remain: To analyze the running time of halving with naive linking if $m < n$, and to analyze the running time of type k reversal for $k \geq 3$.

Our analysis shows that some of the compression methods, while intuitively appealing, are not asymptotically optimal or are dominated by other methods. Specifically, the various kinds of reversal (see Section 4) are not asymptotically optimal; neither is Rem's splicing method for contingent union (see Section 5), unless it is modified to incorporate linking by rank. Furthermore, the collapsing method (see Section 5), sometimes called "fast find," is dominated by compression, splitting, or halving with the same linking rule. Lao [9] gives another rather complicated algorithm that is not asymptotically optimum. For practical applications we favor either halving or compression with linking by rank.

The question of whether there is a linear time set union algorithm remains open. The nonlinear lower bound for separable algorithms [11] suggests that such an algorithm, if there is one, will use the power of random-access memory. Recently, Gabow and Tarjan [6] have devised such a linear-time algorithm for a special case of set union that occurs in many applications, but the method does not seem to extend to the general case, as it requires advance knowledge and preprocessing of the link operations.

REFERENCES

1. ACKERMANN, W. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Ann. 99* (1928), 118–133.
2. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
3. BANACHOWSKI, L. A complement to Tarjan's result about the lower bound on the complexity of the set union problem. *Inf. Process. Lett. 11* (1980), 59–65.
4. DIJKSTRA, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N. J., 1976.
5. FISCHER, M.J. Efficiency of equivalence algorithms. In *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher, Eds. Plenum Press, New York, 1972, pp. 153–168.
6. GABOW, H.N., AND TARJAN, R.E. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the 15th ACM Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, 1983, pp. 246–251.
7. GALLER, B.A., AND FISHER, M.J. An improved equivalence algorithm. *Commun. ACM 7,* 5 (May 1964), 301–303.
8. HOPCROFT, J.E., AND ULLMAN, J.D. Set-merging algorithms. *SIAM J. Comput. 2* (1973), 294–303.
9. LAO, M.J. A new data structure for the union-find problem. *Inf. Process. Lett. 9* (1979), 39–45.
10. TARJAN, R.E. Efficiency of a good but not linear set union algorithm. *J. ACM 22,* 2 (Apr. 1975), 215–225.
11. TARJAN, R.E. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci. 18* (1979), 110–127.
12. TARJAN, R.E. Applications of path compression on balanced trees. *J. ACM 26,* 4 (Oct. 1979), 690–715.
13. VAN LEEUWEN, J., AND VAN DER WEIDE, T. Alternative path compression techniques, Tech. Rep. RUU-CS-77-3, Rijksuniversiteit Utrecht, Utrecht, The Netherlands, 1977.
14. VAN DER WEIDE, T. *Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms.* Mathematisch Centrum, Amsterdam, 1980.