A Fast Merging Algorithm

MARK R. BROWN

Yale University, New Haven, Connecticut

AND

ROBERT E. TARJAN

Stanford University, Stanford, California

ABSTRACT An algorithm that merges sorted lists represented as height-balanced binary trees is given If the lists have lengths m and n ($m \leq n$) then the merging procedure runs in $O(m \log(n/m))$ steps, which is the same order as the lower bound on all comparison-based algorithms for this problem

KEY WORDS AND PHRASES AVL tree, height-balanced tree, linear list, merging, 2-3 tree

CR CATEGORIES 4 33, 4 34, 5 25, 5 31

Introduction

Suppose we are given two linear lists A and B, each of whose elements contains a key from a linearly ordered set, such that each list is arranged in ascending order according to key value. The problem is to *merge* A and B, i.e. to combine the two lists into a single linear list whose elements are in sorted order.

This problem can be studied on different levels. One approach is to ask how many comparisons between keys in the two lists are sufficient to determine the ordering in the combined list. This is an attractive problem because it is relatively easy to prove lower bounds on the number of comparisons as a function of the list sizes, using an "information-theoretic" argument. If the lists A and B have m and n elements, respectively, then there are $\binom{m+n}{n}$ possible placements of the elements of B in the combined list; it follows that $\lceil \lg \binom{m+n}{n} \rceil$ (see Footnote 1) comparisons are necessary to distinguish these possible orderings. If we take $m \leq n$ then $\lceil \lg \binom{m+n}{n} \rceil = \theta(m \log(n/m))$ (see Footnote 2). The best merging procedure presently known within this framework is the "binary merging" algorithm of Hwang and Lin [5, 6], which requires fewer than $\lceil \lg \binom{m+n}{n} \rceil + \min(m, n)$ comparisons to combine sets of size m and n.

A different approach is to study the actual running time of merging algorithms on real

The research of M R Brown was supported by a National Science Foundation graduate fellowship at Stanford University, the research of R E Tarjan was supported in part by the National Science Foundation under Grant MCS-75-22870 and by the Office of Naval Research under Contract N00014-76-C-0688.

Authors' addresses M R Brown, Computer Science Department, Yale University, New Haven, CT 06520, R E Tarjan, Computer Science Department, Stanford University, Stanford, CA 94305

[1] We use lg n to denote $\log_2 n$

[2] That is, the left-hand side has order exactly $m \log(n/m)$, Knuth [8] gives a precise definition of the θ and Ω notations.

computers or on more abstract models such as pointer machines [10]. If we assume that comparisons are the only way of gaining information about key values then the $\Omega(m \log(n/m))$ lower bound still applies to the running time of merging algorithms, but it is not clear how to achieve this bound using the Hwang-Lin procedure. The problem lies in implementing this algorithm to run in time proportional to the number of comparisons it uses.

In this paper we give a merging procedure which runs in $O(m \log(n/m))$ time on a real computer or a pointer machine. The algorithm uses height-balanced binary (AVL) trees [6] to represent the linear lists; 2-3 trees [1] could also be used.

In Section 1 we present the binary merging procedure of Hwang and Lin, and note why it seems difficult to give an efficient implementation of this algorithm. We develop a merging procedure for height-balanced trees in Section 2, and in Section 3 we prove that the procedure runs in $O(m \log(n/m))$ time. Section 4 gives the results of experiments comparing our algorithm with three straightforward merging methods and describes an application of the algorithm. A high level language implementation of the fast merging algorithm is contained in an earlier version of this paper [2].

1. *Binary Merging*

We begin with an informal description of the Hwang-Lin binary merging algorithm. Let A and B be lists containing distinct elements, of respective lengths m and n with $m \leq n$, such that

$$a_1 < a_2 < \cdots < a_m \quad \text{and} \quad b_1 < b_2 < \cdots < b_n.$$

The merging method is most easily described recursively. When $m = 0$ (i.e. the shorter list is empty) there is no merging to be done and the procedure terminates. Otherwise we attempt to insert a_1, the smallest element in the shorter list A, into its proper position in the longer list B. To do this, let $t = \lfloor \lg(n/m) \rfloor$ and compare $a_1 : b_{2^t}$ (2^t is the largest power of two not exceeding n/m). See Figure 1.

If $a_1 < b_{2^t}$, then a_1 belongs somewhere to the left of b_{2^t} in Figure 1 By using binary search the proper location of a_1 among $b_1, b_2, \ldots, b_{2^t-1}$ can be found with exactly t more comparisons. The result of this search is a position k such that $b_{k-1} < t < b_k$; this information allows us to reduce the problem to the situation illustrated in Figure 2(a). To complete the merge it is sufficient to perform binary merging on the lists A' and B'.

If, on the other hand, $a_1 > b_{2^t}$, then a_1 belongs somewhere to the right of b_{2^t} in Figure 1, and the problem immediately reduces to the situation illustrated in Figure 2(b). We can finish the merge by applying the binary merging procedure to the lists A' and B'. Note that A' may be longer than B', so that in the recursive calls to the binary merging procedure the roles of A and B may become reversed; this may also happen in the first reduction above.

Hwang and Lin [5] derive a $\lceil \lg(\binom{m+n}{n}) \rceil + \min(m, n)$ upper bound on the number of comparisons used by this algorithm, using a somewhat complicated analysis. The algorithm uses comparisons very efficiently, as evidenced by the small gap between this upper bound and the lower bound of $\lceil \lg(\binom{m+n}{n}) \rceil$ comparisons for any merging method based on comparisons. However, overhead is a problem with this algorithm. Representing the lists A and B as arrays, which is the obvious way of making the individual comparisons take constant

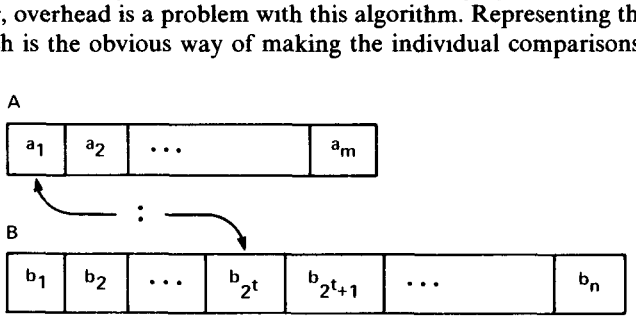

FIG 1 First comparison during a binary merge

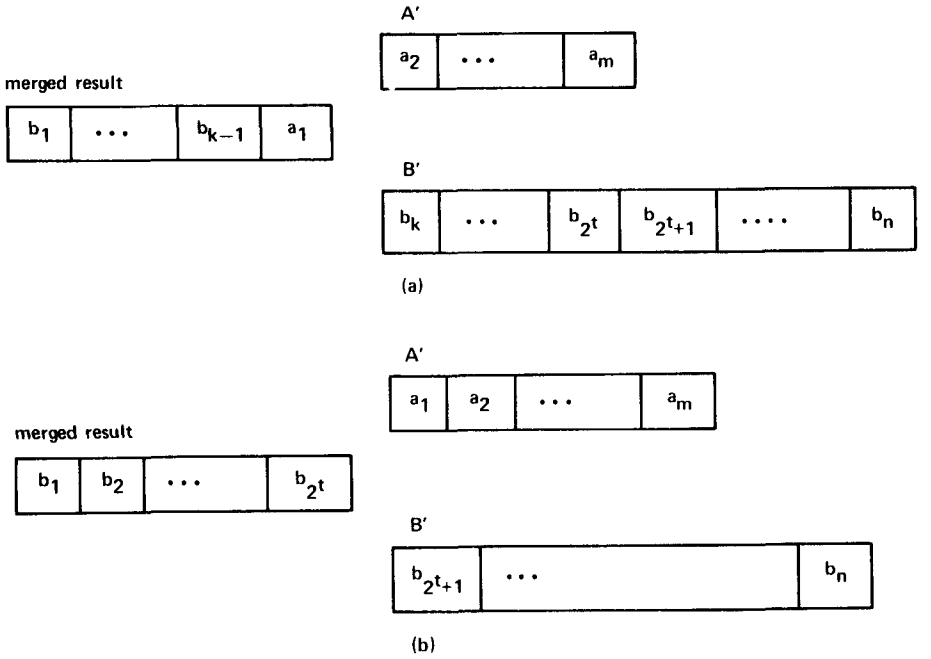

FIG 2 Outcomes after the first part of a binary merge (a) $a_1 < b_{2^t}$, (b) $a_1 > b_{2^t}$

time, forces insertions to be expensive: They involve moving items over to make room for the inserted items. Hwang and Lin [5] were concerned with an application in which A and B are read from tapes and the merged result is written onto a tape; in this situation the merge requires linear time (since the entire result must be written), and binary merging can only improve on the traditional tape merging algorithm by a constant factor. Our goal is to be able to merge list structures A and B and produce a result list of the same type in $O(m \log(n/m))$ total operations.

2. *Height-Balanced Tree Merging*

Height-balanced binary trees [6] are good data structures for representing linear lists when both searches and insertions must be performed. A binary tree is called *height-balanced* if the height of the left subtree of every node never differs by more than ± 1 from the height of its right subtree. (The *height* of a tree is the length of the longest path from the root to an external node.) When representing a list by a height-balanced binary tree, the ith element of a list becomes the ith node visited during a symmetric order traversal of the binary tree; if the list is sorted, as in Figure 3, then its keys appear in increasing order during such a traversal. If a sorted list of length n is maintained in this way, we can locate the proper position in the list for a new element in $O(\log n)$ steps by using ordinary binary tree search. To insert this element into the list may require $O(\log n)$ additional steps for rebalancing the tree. We shall assume that the reader is familiar with Knuth's presentation of height-balanced tree search and insertion (Algorithm 6.2.3A [6]).

An obvious method of merging two sorted lists represented as height-balanced trees is to insert the elements of the smaller list into the larger list one by one. The result of merging the two lists of Figure 3 using this scheme is shown in Figure 4. If the smaller list contains m elements and the larger has n elements then this algorithm performs m insertions of $O(\log n)$ steps each, for a total cost of $O(m \log n)$. But we are seeking a method which runs in $O(m \log (n/m))$ time.

To see why there is some hope of improving this simple merging procedure we refer again to Figure 4, which shows the search paths traced out during the insertions An

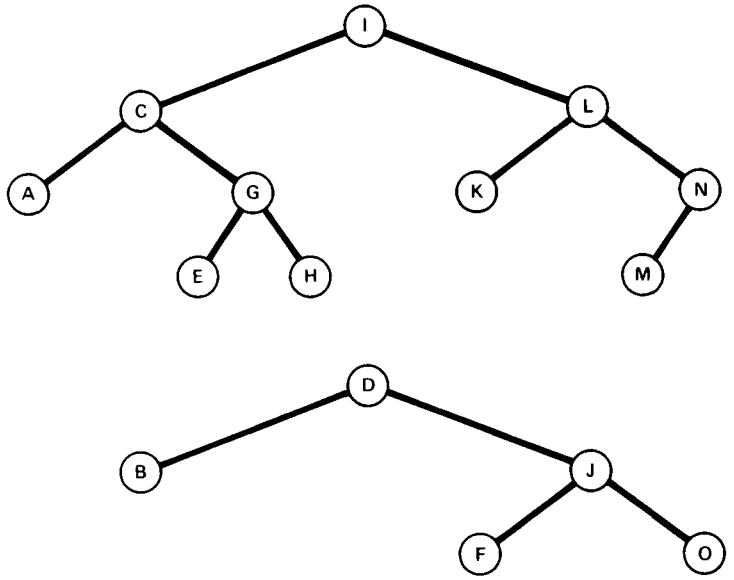

FIG 3 Sorted lists represented as height-balanced binary trees

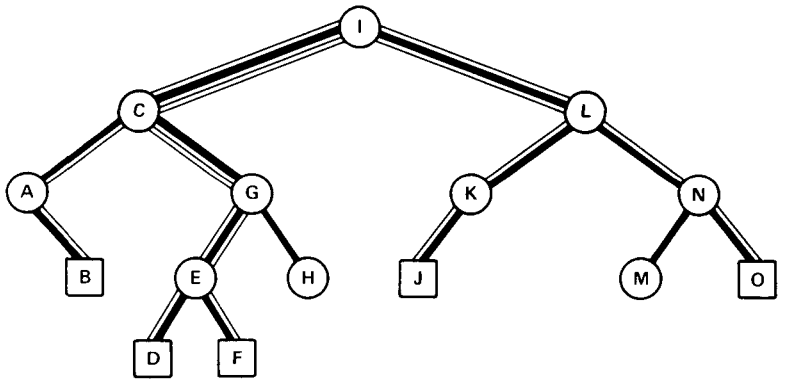

FIG 4 An example of merging by independent insertions (square nodes have been inserted)

interesting property of these paths is that they share many nodes near the top of the tree. The root is visited on all of the searches, and its two offspring are each visited on roughly half of the searches; we must descend at least lg m levels into the tree before all of the search paths become disjoint. It appears that our simple merging strategy spends lg m steps on each insertion, or $O(m \log m)$ steps total, examining nodes in the top lg m levels of the tree. Since there are only $O(m)$ nodes contained in these levels, eliminating duplicate visits should make our algorithm run in $O(m \log n - m \log m + m) = O(m \log(n/m))$ time.

We can eliminate extra visits since the items being inserted are themselves already sorted; by simply inserting these items in order we can ensure that once an item has been inserted, no smaller item will be inserted later. Figure 5, which shows the situation after a node x has been inserted, indicates how this can help. If node $y \geq x$ is now inserted, then y must lie somewhere to the right of x in the tree. To determine where y belongs it is sufficient to climb back up the search path, comparing y to nodes on the path which are greater than x until a node is found which is greater than y; then y can be inserted into the right subtree of the previous node examined during the climb. (For this purpose it is convenient to think of the root as having a parent with key $+\infty$.) In Figure 5, if $y \geq \gamma$ but $y < \beta$ then y should be inserted into the right subtree of node γ; if $y < \gamma$ then y becomes the right offspring of x.

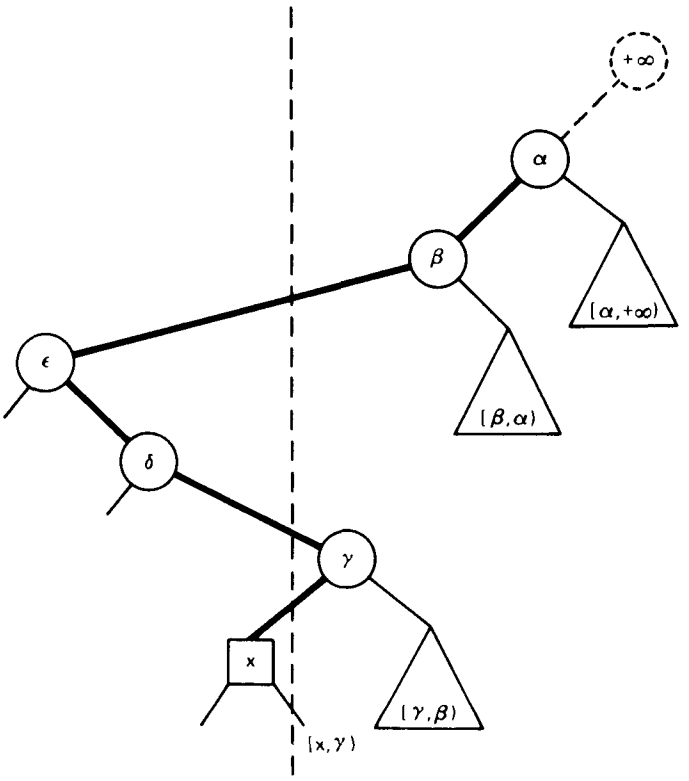

FIG 5 Inserting an item larger than x (subtrees are labeled with the range of key values which may be inserted)

An algorithm based on this idea is easy to state informally. As in our description of binary merging, let A and B be sorted lists of length m and n, with $m \leq n$, and assume that these lists are represented as height-balanced trees. In the first step of our algorithm we insert a_1, the smallest element of A, into the tree B. At the start of a general step, elements $a_1, a_2, \ldots, a_k$ have been inserted into B, and we have a record of the search path to a_k (Figure 6(a)). This path acts as a "finger" into the tree B during the algorithm, moving from left to right through B as elements from A are inserted; the finger is useful because only nodes to the right of it can be visited during later insertions.

The general step has two parts. First the finger is retracted toward the root, just far enough so that the position of the element a_{k+1} lies within the subtree rooted at the end of the finger (Figure 6(b)). Then a_{k+1} is inserted into this subtree, and the finger is extended to follow the path of this insertion (Figure 6(c)). After $m - 1$ executions of this general step the merge is complete.

This scheme is complicated by the fact that rebalancing may be necessary during insertions into a height-balanced tree. When rebalancing takes place, it may remove a node from the finger path traced out by the search. It is possible to update the recorded path to be consistent with this rearrangement, but it seems easier just to "forget" about the part of the path which is corrupted, i.e. to retract the finger path back to the point of rebalancing. The algorithm then takes the form shown in Figure 7. At the start of the general step we now have recorded only *part* of the search path to the last element inserted. The general step proceeds as before, but after the insertion a part of the search path may be discarded. There is no need to treat the first insertion specially in this algorithm; we simply initialize the finger path to be the root of B (which is certainly on the path to the first insertion) and execute the general step m times.

In an implementation of this scheme it is useful to maintain a record of those nodes on the finger path at which the path turns left (i.e. nodes on the path whose left offspring is

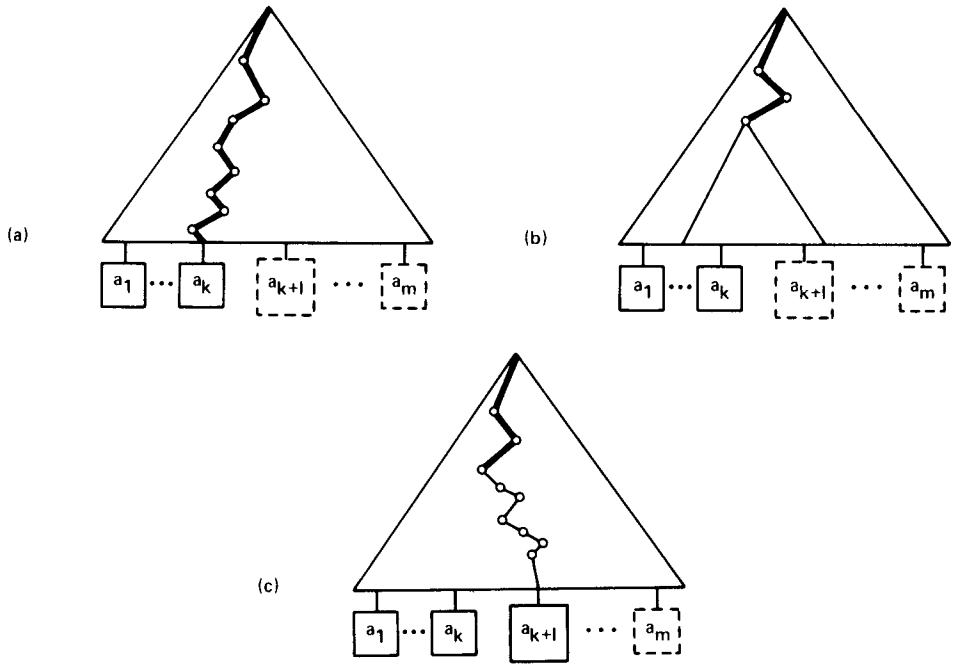

FIG 6 Insertion using the finger path

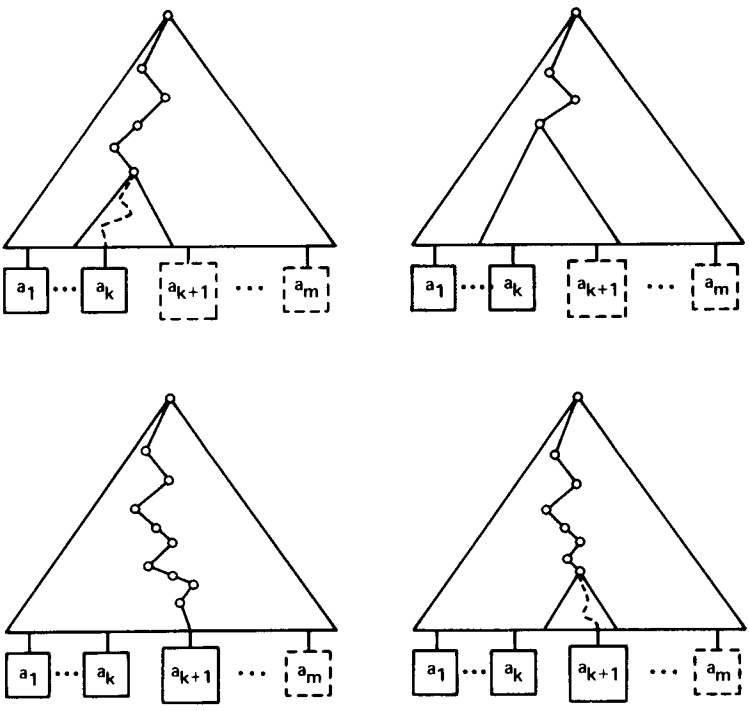

FIG. 7 Retracting the path for rebalancing

also on the path). It is easy to see that these are precisely the nodes on the path (excluding the last node) which are larger than the most recently inserted item; according to Figure 5, only these nodes must be examined while climbing upward in the tree in the first part of the general step. Bad cases may occur if we do not record these nodes and we must

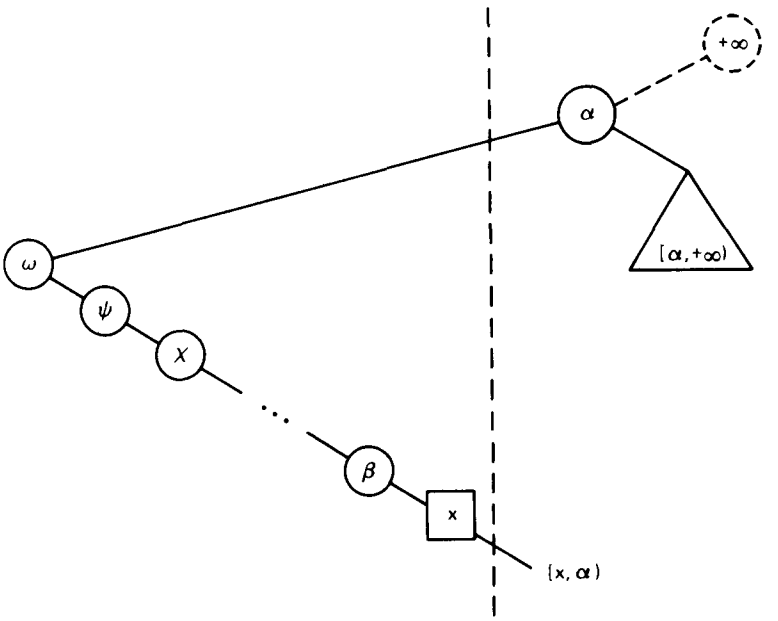

FIG 8 A bad insertion

examine small nodes on the finger path, as illustrated in Figure 8. If a node $y \geq x$ is inserted in the situation shown, the entire path up to the root must be climbed to see if $y \geq \alpha$. If it turns out that $y < \alpha$, then y becomes the right offspring of x and the same situation can be repeated.

Using these ideas we can express the height-balanced tree merging algorithm in an Algol-like notation. (The control constructs used in this notation are adapted from Knuth [7].) We keep pointers to nodes on the finger path in a stack *path*, and pointers to the "large" path nodes (in the sense of the previous paragraph) in a *successor* stack. The nodes of the balanced tree are taken to have fields *Key*, *lLink*, *rLink*, and *B* (balance factor), as in Algorithm 6.2.3A [6]. The balance factor may take on the values *leftTaller*, *balanced*, and *rightTaller*, which have obvious interpretations; the rebalancing step depends on the relation *leftTaller* = −*rightTaller*, which is assumed to hold.

```
begin {Fast height-balanced tree merge}
initialize path to contain the root of the larger tree, and height to be the height of the larger tree
initialize successor to be empty
{We say that PathPredicate holds for node z if
 (1) path contains an initial segment of the path from the root of the larger tree (as modified by the insertions so
     far) to the position which z has, or should have after an insertion, in the larger tree, and
 (2) successor contains all path entries whose lLinks are also in path, i e all nodes on the path (excluding the last)
     which are greater than the last node inserted.}
loop for each node in the smaller tree·
   x ← next node from the smaller tree, in symmetric order, initialized so that lLink[x] = rLink[x] = Nil and
        B[x] = balanced
{Here PathPredicate holds for the node just inserted (the first time through the loop, PathPredicate holds for any
 node) The following loop removes nodes from path (and successor) until PathPredicate holds for x }
  {climb up}
  loop until successor is empty or Key[x] < Key[top of successor]
    loop until top of path = top of successor.
      remove top from path
    repeat
    remove top from successor
  repeat
  p ← top of path
```

```
{Here PathPredicate holds for x; the following loop adds nodes to path (and successor), preserving this property,
 until an external node is found and replaced by x }
  {search down and insert}
  loop
    if Key[x] < Key[p] then
      if lLink[p] = Nil then goto leftNil
                        else push p onto successor
                             p ← lLink[p] endif
    else
      if rLink[p] = Nil then goto rightNil
                        else p ← rLink[p] endif
    endif
    push p onto path
  repeat
  then leftNil ⇒ lLink[p] ← x
    rightNil ⇒ rLink[p] ← x
  endloop

{Here PathPredicate holds for x, and x is a child of the top node on path The tree, with x inserted, may be out
 of balance The following loop removes balanced nodes from path (and successor) until an unbalanced node s
 (or the root) is found }

  {adjust balance factors}
  loop
    pop path into s
  until B{s} ≠ balanced or path is empty.
    B[s] ← (if Key[x] < Key[s] then leftTaller else rightTaller)
    if successor is not empty and top of path = top of successor
      then remove top from successor endif
  repeat
  a ← (if Key[x] < Key[s] then leftTaller else rightTaller)

  {rebalance the subtree rooted at s, this part of the program is essentially a translation of steps 7-10 of
   Algorithm 6 2.3A [6]}

  if B[s] = balanced then {entire tree has increased in height}
    B[s] ← a, height ← height + 1
  else if B[s] = −a then {subtree has become more balanced}
    B[s] ← balanced
  else {rotation is necessary to restore balance}
    r ← (if Key[x] < Key[s] then lLink[s] else rLink[s])
    if B[r] = a then {single rotation}
      if a = rightTaller then rLink[s] ← lLink[r], lLink[r] ← s
                         else lLink[s] ← rLink[r], rLink[r] ← s endif
      B[s] ← B[r] ← balanced
      s ← r
    else {double rotation}
      if a = rightTaller then
        p ← lLink[r]; lLink[r] ← rLink[p], rLink[p] ← r
        rLink[s] ← lLink[p], lLink[p] ← s
      else
        p ← rLink[r]; rLink[r] ← lLink[p], lLink[p] ← r
        lLink[s] ← rLink[p], rLink[p] ← s
      endif
      B[s] ← (if B[p] = +a then −a else balanced)
      B[r] ← (if B[p] = −a then +a else balanced)
      B[p] ← balanced
      s ← p
    endif
  endif
  push s onto path

repeat
{The root of the result tree is on the bottom of path, and its height is height}
end {Fast height-balanced tree merge}
```

3. *Running Time*

In order to analyze the running time of the height-balanced tree merging algorithm, it is necessary to look at the details of the rebalancing procedure (steps 6–10 of Algorithm 6.2.3A [6]). For the purpose of this discussion we shall adopt a concise notation for balance factors: The balance factor of any node is either 0 (left and right subtrees of equal height), + (right subtree of height one greater than left subtree), or − (right subtree of height one less than left subtree). A node with balance factor 0 is called *balanced*, and the other nodes are *unbalanced*.

When a node x is inserted in place of an external node in a height-balanced tree, this may cause ancestors of x in the tree to increase in height. To rebalance the tree we examine successive ancestors of x, moving up toward the root. During this climb we change the balance factor of each balanced node to + or − as appropriate until an unbalanced node, say z, is found. (If we reach the root without finding an unbalanced node then the entire tree has increased in height and the insertion is complete.) Insertion of x causes node z to become either balanced or doubly heavy on one side. If z becomes balanced we simply change its balance factor to 0; otherwise we locally modify the subtree rooted at z to restore balance while leaving its height the same as it was before x was inserted. The two local transformations shown in Figure 9 will rebalance the subtree in all cases. Since the subtree

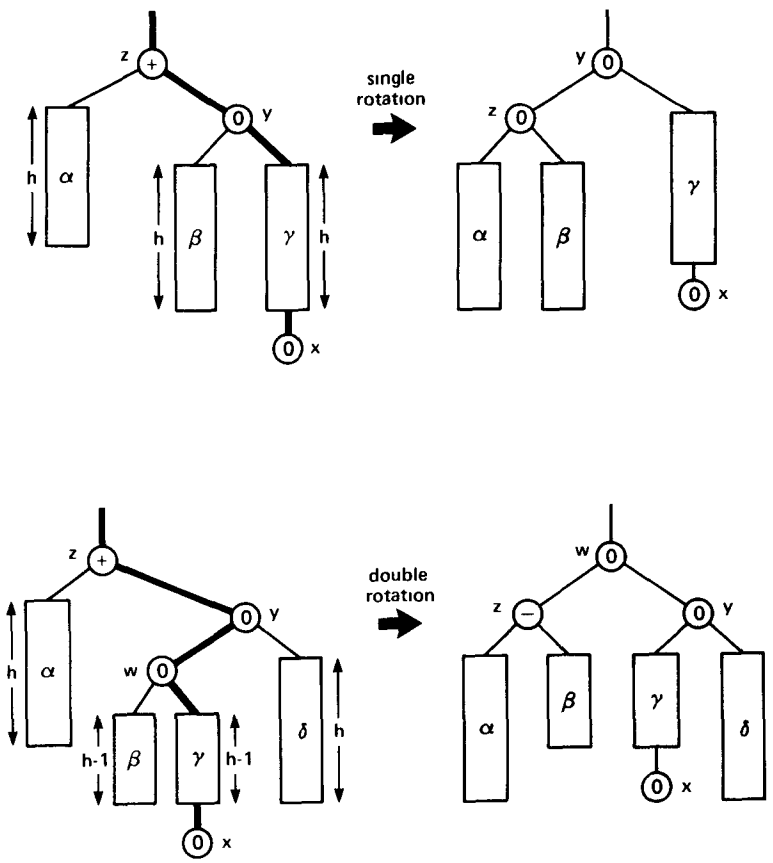

FIG 9 Rebalancing after inserting node x (In both single and double rotations the mirror-image transformation is possible There are actually three subcases of a double rotation, one of which is shown When $h = 0$, node x is node w and the subtrees α, β, γ, and δ are empty When $h > 0$, node x may be inserted into either the left or right subtree of w)

rooted at z does not change in height, no nodes above z need be examined during the insertion.

Call a node "handled" if it is manipulated by the height-balanced tree merging algorithm. The handled nodes consist of all nodes in the smaller tree (the nodes which are inserted into the larger tree) and all nodes in the larger tree which are examined while searching for the location to make an insertion. (Each node examined while rebalancing must have been previously examined while searching.) We shall obtain an $O(m \log(n/m))$ bound on the running time of the algorithm by showing that

(i) the time required by the algorithm is proportional to the number of handled nodes (where a node is counted only once even if it is handled many times), and
(ii) the total number of handled nodes is $O(m \log(n/m))$.

We proceed by means of a sequence of lemmas.

LEMMA 1. *The time required by the height-balanced tree merging algorithm is bounded by a constant times m plus a constant times the number of additions to and deletions from the path and successor stacks.*

PROOF. An inspection of the program shows that the algorithm requires a bounded amount of time per insertion plus a bounded amount of time per addition to or deletion from a stack. □

LEMMA 2. *The total number of additions to a stack is bounded by the total number of stack deletions plus the number of handled nodes.*

PROOF. The number of stack additions exceeds the number of deletions by the number of elements in the stack when the algorithm terminates. But each node in the stack has been handled, so the result follows. (The maximum stack depth is actually $O(\log(n + m))$, which is generally much smaller than the number of handled nodes.) □

Nodes are deleted from stacks at two points in the program: while adjusting balance factors during rebalancing (in the loop labeled "adjust balance factors"), and while climbing up the path during insertions (in the loop labeled "climb up"). We now analyze each case in turn.

LEMMA 3. *The number of deletions from stacks during rebalancing cannot exceed a constant times the number of handled nodes.*

PROOF. All nodes handled during a rebalancing step except at most three have their balance factor changed from 0 to either + or −. Thus each *path* stack deletion except three per rebalancing "uses up" a balanced node, and each rebalancing creates at most three balanced nodes. Since the initial pool of balanced nodes which are handled cannot exceed the total number of handled nodes, the total number of *path* stack deletions during rebalancing is no more than the number of handled nodes plus six times the number of rebalancings. Since the number of rebalancings cannot exceed the number of nodes inserted, and each such node is handled, the number of *path* stack deletions is bounded by a constant times the number of handled nodes. The number of *successor* stack deletions during rebalancing cannot exceed the number of *path* stack deletions during rebalancing. The lemma follows. □

LEMMA 4. *The number of deletions from stacks during insertions cannot exceed a constant times the number of handled nodes.*

PROOF. Each node y deleted from the *successor* stack during insertion has a key smaller than the key of the node x currently being inserted; thus y can never again be added to (or deleted from) the *successor* stack. Hence each handled node can be deleted from the *successor* stack during insertion at most once.

Each node y deleted from the *path* stack during insertion is either (i) immediately deleted from the *successor* stack or (ii) has the property that there is a node z such that z is a proper ancestor of y in the currently existing tree and $Key(y) < Key(z) < Key(x)$, where x is the node currently being inserted. To verify this claim, let z be the node on top of the *successor* stack when y is deleted from the *path* stack. An inspection of the program reveals

that if $y = z$, z is immediately deleted from the *successor* stack, and if $y \neq z$, then (ii) holds.

The number of nodes y satisfying (i) cannot exceed the number of nodes deleted from the *successor* stack, and hence the number of handled nodes. To verify the same bound for nodes satisfying (ii), we must show that if (ii) holds for a node y immediately after it is deleted from the *path* stack, then it holds for y at all times thereafter. The keys being inserted are increasing, so the rightmost inequality in (ii) is preserved when beginning a new insertion (although x changes). Insertions may cause rebalancing to take place above z, at z, or in the right subtree of z, but inspection of Figure 9 shows that in any case property (ii) is preserved for y after rebalancing. (A rotation may change the identity of z.) A node y which satisfies (ii) and is not on the *path* stack can never again be added to (or deleted from) the *path* stack, since the key of the left child of z will never be compared against the key of a node being inserted. Thus the number of *path* stack deletions satisfying (ii) is at most one per handled node.

In summary, at most three stack deletions per handled node can occur during insertions. □

THEOREM 1. *The total time required by the height-balanced tree merging algorithm is bounded by a constant times the number of handled nodes.*

PROOF. Immediate from Lemmas 1–4. □

The bound on the number of handled nodes is proved in two steps. First we show that when the algorithm terminates, most of the handled nodes constitute a subtree of the height-balanced tree resulting from the merge, and this subtree has at most m terminal nodes (nodes having no internal node of the subtree as an offspring). Then we bound the number of nodes that any such subtree of a height-balanced tree may have.

LEMMA 5. *After k insertion-rebalancing steps, the set of handled nodes consists of no more than k nodes plus a subtree of the entire height-balanced tree containing no more than k terminal nodes and containing all ancestors of the most recently inserted vertex.*

PROOF. We prove the lemma by induction on k. The lemma is certainly true for $k = 0$. Suppose the lemma is true for $k - 1$. Let T_{k-1} be the subset of handled nodes which forms a subtree with $k - 1$ or fewer terminal nodes containing all ancestors of the node inserted at the $(k - 1)$-st step. Let H_{k-1} be the remaining $k - 1$ or fewer handled nodes. Each node handled during the kth insertion is an ancestor of either the node x_{k-1} inserted at the $(k - 1)$-st step or of the node x_k inserted at the kth step. Let T'_k be formed from T_{k-1} by adding all ancestors of x_k. Then T'_k forms a subtree with k or fewer terminal nodes.

The kth rebalancing step does not handle any new nodes but may alter the shape of the overall tree and thus may rearrange the vertices in T'_k. However, an inspection of Figure 9 reveals that, after rebalancing, the nodes in T'_k still form a subtree having the same number of terminal nodes as before, except for possibly one additional "special" terminal node. This special node is a child of a node with two offspring, so removing it from T'_k does not create a new terminal node. In Figure 9, node z becomes special if T'_k does not enter either of the subtrees α or β. If z becomes special after rebalancing, let $T_k = T'_k - \{z\}$ and $H_k = H_{k-1} \cup \{z\}$; otherwise let $T_k = T'_k$ and $H_k = H_{k-1}$. Then T_k and H_k satisfy the lemma for k. □

LEMMA 6. *Let T be any height-balanced tree of k nodes. Let T' be any subtree of T with at most l terminal nodes. Then T' contains $O(l \log(k/l))$ nodes.*

PROOF. A height-balanced tree of height $h = 1.4404 \lg(k/l + 2) - 0.328$ must contain at least k/l nodes [6, Theorem 6.2.3A]. If T has height less than $h + 2$, then T' can be partitioned into l paths, each of length less than $h + 2$, and the lemma is true.

On the other hand, suppose the height of T is no less than $h + 2$. We shall conceptually subdivide T into smaller trees as follows: Let the set R consist of the root of T, plus all other nodes in T which have height $h + 2$ or greater. It is not hard to see that R forms a subtree of T, as shown in Figure 10, and that the remaining nodes of T are partitioned into a set of disjoint subtrees $\{S_i\}$.

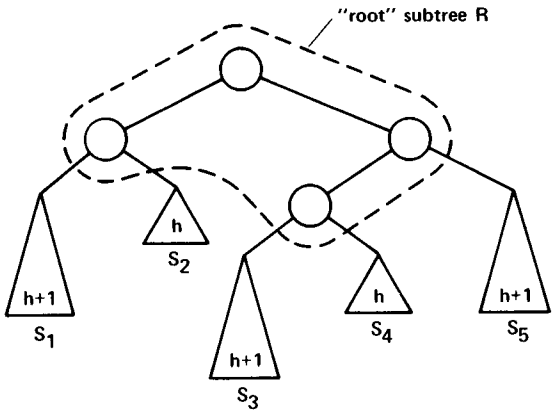

FIG 10 Subdivision of a height-balanced tree

A height-balanced binary tree has the property that if v is any node, the heights of the two children of v differ by at most one. Thus the difference in height between v and either of its two children is at most two. It follows that each subtree S_i has height h or $h + 1$, since if the height was less than h then the parent (which lies in R) would have height less than $h + 2$. By the choice of h this guarantees that each S_i contains at least k/l nodes, so there are at most l subtrees S_i. Each of these subtrees is attached to an external node of the "root" subtree R, so there are at most $l - 1$ nodes in R.

With T subdivided in this way it is easy to bound the number of nodes in T'. The nodes of T' which do not lie in R can be partitioned into l paths, each lying completely within a subtree S_i. Since each such path has length not exceeding $h + 1$, the total number of nodes in T' cannot exceed $l - 1 + l(1.4404 \lg(k/l + 2) + 0.672) = O(l \log(k/l))$. □

THEOREM 2. *The total number of nodes handled by the height-balanced tree merging algorithm is* $O(m \log(n/m))$.

PROOF. The total number of nodes in the tree resulting from the merge is $m + n$. Thus by Lemmas 5 and 6 the total number of handled nodes is no more than $m + O(m \log((m + n)/m)) = O(m \log(n/m))$. □

THEOREM 3. *The height-balanced tree merging algorithm requires* $O(m \log(n/m))$ *time to merge lists of sizes* m *and* n *with* $m \leq n$.

PROOF. Immediate from Theorems 1 and 2. □

One may wonder why the proof of Theorem 3 is so complicated, while the informal motivation given for this bound in Section 2 is so simple. Perhaps the reason is that each insertion changes the structure of the tree; thus it seems necessary to analyze the stack operations directly in order to have confidence in the proof.

4. *Implementation and Application*

It is possible for an algorithm to be very fast asymptotically, but to be terribly slow when applied to problems of a practical size for present-day computers. Therefore it is worthwhile for us to compare our height-balanced tree merging algorithm with other merging procedures to determine when the new method is actually "fast." In the discussion below we shall refer to our height-balanced tree merging algorithm as Algorithm F.

One straightforward merging procedure for linear lists represented as height-balanced trees has already been described in Section 2: that of inserting the elements of the smaller tree one by one into the larger tree. We shall call this method Algorithm I. Since this procedure requires $\theta(m \log n)$ time, we expect it to be the most useful when m is very small compared to n.

Another simple merging procedure for height-balanced trees is to scan entirely through both trees in increasing order and perform a standard two-way merge of the lists. This method, which we call Algorithm T, divides nicely into three stages of coroutines. The first-stage routines dismantle the input trees and send their nodes in increasing order to the next stage. (Identical routines are also needed to dismantle the smaller tree in Algorithms F and I.) The second stage compares the smallest elements remaining in the two lists, and sends the smaller of the two elements to the third tage. The final routine accepts nodes in increasing order and creates a height-balanced tree from them. Given that the total number of nodes is known in advance, a simple way to construct this tree in linear time is to divide the nodes as evenly as possible between the left and right subtrees of the root, building these subtrees recursively by the same method if they are nonempty. (By the use of a more elaborate construction [6, Exercise 6.2.3-21], a height-balanced tree can be built in linear time even if the number of nodes is not known in advance.) Algorithm T requires $\theta(m + n)$ time, so it may be a good method when m is almost as large as n.

A final method which should be part of our comparison is Algorithm L, standard two-way merging of singly linked linear lists. The running time of this procedure is $\theta(m + n)$, like Algorithm T, but we expect Algorithm L to be more efficient because the first and third stages of Algorithm T become much simpler when singly linked lists are used instead of height-balanced trees.

For purposes of comparison, each of these algorithms was implemented in the assembly language of a hypothetical multiregister computer [7]. Each instruction executed is assumed to cost one unit of time, plus another unit if it references memory for data. By inspecting the programs, we can write expressions for their running time as a function of how often certain statements are executed. The average values of these execution frequencies are then determined either mathematically (in the case of Algorithms T and L) or experimentally (in the case of some factors in Algorithms F and I). The experimental averages are determined by executing high level language versions of the algorithms under a system which automatically records how often each statement is executed [9, Appendix F].

The results of this evaluation are summarized in Table I, which gives formulas for the average running time of each of the four algorithms. Figure 11 compares the three height-balanced tree merging algorithms by showing the values of the list sizes m and n for which each of the three algorithms is faster than the other two. It turns out that Algorithm F beats Algorithm I when $m \geq 4.04n^{0.253}$, and Algorithm F is faster than Algorithm T when $m \leq 0.355n$. Furthermore, Algorithm F is never more than about 33 percent slower than Algorithm I, or 54 percent slower than Algorithm T. Thus Algorithm F seems to be a practical merging procedure for height-balanced trees.

In some situations the flexibility of height-balanced trees may not be needed, and the simpler singly linked list representation might seem preferable. Our comparison shows that from the standpoint of merging, height-balanced trees are worthwhile whenever the lists being merged differ in size by a factor of 16.5 or more. So in order to derive a benefit from the simpler representation we must keep the merges fairly well balanced.

It now seems appropriate to make some general remarks about Algorithm F and its implementation. Our first observation is that the general scheme of the algorithm and its running time proof apply directly to 2-3 trees (or general B-trees). For example, the argument of Lemma 3 concerning the number of balanced nodes handled during rebal-

TABLE I COMPARISON OF METHODS

	Average running time to merge lists of sizes m and n, with $m \leq n$
Algorithm F	$15.0\, m \lg(n/m) + 118.5m + 43.5$
Algorithm I	$11.2m \lg n + 88.3m + 21$
Algorithm T	$35.9(m + n) + 4m + 59.2$
Algorithm L	$10(m + n) + 4m + 32$

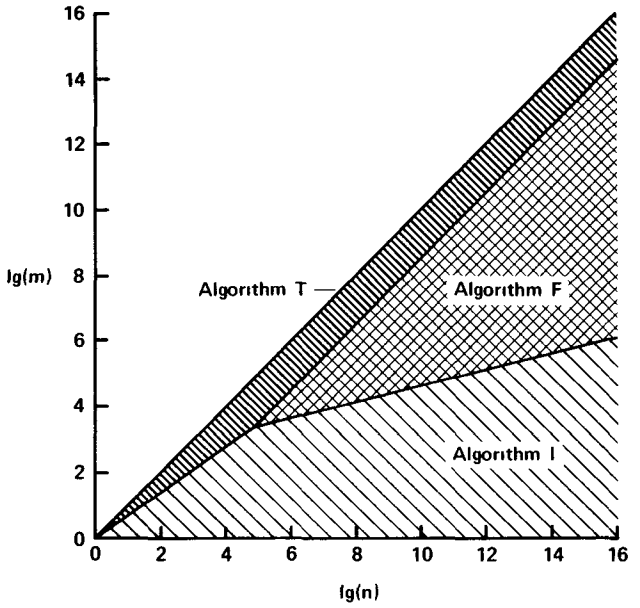

FIG 11. Zones of best performance for height-balanced tree merging algorithms

ancing translates into an argument about the number of *full* nodes (nodes containing two keys) handled during splitting in the 2-3 tree case. The algorithm might be easier to state in an abstract way in terms of 2-3 trees, rather than height-balanced trees, but as soon as a representation for 2-3 trees is specified the algorithm becomes just as complex.

The merging algorithm could be implemented to operate on triply linked height-balanced trees [4], which contain a pointer in each node to its parent. In this case the *path* stack would be unnecessary, since the upward links provide the information. If the tree were also threaded in an appropriate way then the *successor* stack could be eliminated.

The program given in Section 2 uses only conventional stack operations on the *path* and *successor* stacks; hence it is clear that this program can run on a pointer machine within our time bound. On an actual computer we would implement the stacks as arrays, with an integer stack pointer. Then rather than keeping pointers to nodes as entries in the *successor* stack, we can keep pointers to the *path* stack entries for these nodes. This allows us to delete all *path* entries up to the top node of *successor* by simply assigning the top element of *successor* to the *path* stack pointer, which makes the climbing-up phase of each insertion considerably faster and hence reduces the coefficient of $m \lg(n/m)$ in the running time. The implementation given in the Appendix of an earlier version of this paper [2] uses this stack technique, and also retracts the stacks during rebalancing only if rebalancing invalidates some of the path; the latter change in the algorithm has little effect on its running time since rebalancing seldom occurs high in the tree.

A further improvement in the algorithm comes from considering the relationship between our method and the Hwang-Lin binary merging procedure presented in Section 2. A principal distinction between the two is that binary merging always probes near to where the item being inserted is expected to fall; with balanced trees we climb up the search path during insertions and examine nodes which are very unlikely to be larger than the item being inserted. Using an array stack implementation we can avoid many useless comparisons by jumping directly to a node on the path where the next comparison will be less biased. The proof of Lemma 6 indicates that a possible strategy is to jump to a node of height h where h is chosen to guarantee that a subtree of this height contains at least n/m nodes. Since computing this height during the search is expensive, it seems preferable

to jump to a fixed *depth* in the tree, such as $\log_\phi m$, instead; this operation is extremely fast using an array stack implementation. Jumping back to a depth near lg m improves the average case, since random height-balanced trees are so well balanced, but it makes the worst case greater than $O(m \log(n/m))$.

Another scheme for fast merging uses a linear list representation [3] which is more powerful than height-balanced trees. This structure, based on 2-3 trees, allows a finger into the list to be maintained such that the cost of searching (or moving the finger) d positions is $O(\log d)$. When consecutive insertions into a list of size n are made at distances d_1, d_2, ..., d_k from a (possibly moving) finger, the total cost is $O(\log n + \sum_{1 \le i \le k} \log d_i)$.

Using this structure, we can solve the merging problem by inserting the items from the smaller list in increasing order into the larger list, keeping the finger positioned at the most recently inserted item. This process requires $O(m)$ steps to dismantle the smaller list, and $O(\log n + \sum_{1 < i \le m} \log d_i)$ steps for the insertions, where d_i is the distance from the finger to the ith insertion. Since the items are inserted in increasing order, the finger moves from left to right through the larger list, and thus $\sum_{1 < i \le m} d_i \le n$. To maximize $\sum_{1 < i \le m} \log d_i$ subject to this constraint, we choose the d_i to be equal, giving a bound of $O(m \log(n/m))$ on the running time. Because this merging algorithm uses standard primitive operations on the list structure, it is somewhat less efficient than Algorithm F; on the other hand, it will certainly adapt more easily to small changes in the problem.

An interesting application of such fast merging algorithms is to set manipulation. Suppose we wish to maintain sets consisting of elements from a linearly ordered set; then a sorted list is a natural set representation. It is easy to see how to modify a merging algorithm to perform set union for this representation: Elements which occur in both sets must be detected and discarded as the merge proceeds. This is easy to accomplish in either of the fast merging algorithms we have discussed, without changing the time bounds. Set intersection may be computed by simply retaining the duplicate elements and forming them into a new sorted list. Thus the fast merging algorithms lead to $O(m \log(n/m))$ algorithms for set union and intersection. Trabb Pardo [11] has shown that a trie structure can be used to compute set intersections in $O(m \log(n/m))$ time on the average, although not in the worst case, and his algorithms lead to an $O(m \log(n/m))$ average-time merging procedure.

Another application of a fast merging algorithm is to merge sorting. The standard merge sorting method [6] sorts in $O(n \log n)$ time by using an $O(m + n)$-time merge to combine singletons in pairs, then the pairs in pairs, and so on. With an $O(m \log(n/m))$-time merging algorithm, one can sort in $O(n \log n)$ time by starting with singleton sets, and merging ordered sets in an *arbitrary* order until one set remains.

REFERENCES

1. AHO, A V, HOPCROFT, J E, AND ULLMAN, J D *The Design and Analysis of Computer Algorithms* Addison-Wesley, Reading, Mass, 1974
2. BROWN, M.R., AND TARJAN, R.E A fast merging algorithm STAN-CS-77-625, Comptr Sci Dept, Stanford U, Stanford, Calif, Aug 1977
3. BROWN, M R., AND TARJAN, R E A representation for linear lists with movable fingers Proc Tenth Annual ACM Symp. on Theory of Computing, San Diego, Calif, 1978, pp 19–29
4. CRANE, C.A Linear lists and priority queues as balanced binary trees Ph D. Th, STAN-CS-72-259, Comptr Sci. Dept., Stanford U, Stanford, Calif, Feb 1972
5. HWANG, F K, AND LIN, S A simple algorithm for merging two disjoint linearly ordered sets. *SIAM J. Computg. 1*, 1 (March 1972), 31–39
6. KNUTH, D E *The Art of Computer Programming, Vol. 3 Sorting and Searching* Addison-Wesley, Reading, Mass, 1973.
7. KNUTH, D E Structured programming with **goto** statements *Comptng. Surveys 6*, 4 (Dec 1974), 261–301
8. KNUTH, D E Big omicron and big omega and big theta SIGACT News (ACM) *8*, 2 (April 1976), 18–24
9. REISER, J.F., Ed SAIL STAN-CS-76-575, Comptr Sci Dept, Stanford U, Stanford, Calif, Aug 1976

10 TARJAN, R.E. Reference machines require non-linear time to maintain disjoint sets Proc. Ninth Annual ACM Symp. on Theory of Computing, Boulder, Colo., 1977, pp 19-29
11 TRABB PARDO, L. Set representation and set intersection Ph D Th., Comptr Sci Dept., Stanford U., Stanford, Calif., 1978

RECEIVED AUGUST 1977, REVISED JUNE 1978