A FAST MERGING ALGORITHM

by

Mark R. Brown and Robert E. Tarjan

STAN-CS-77-625
AUGUST 1977

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

A Fast Merging Algorithm

Mark R. Brown
Department of Computer Science
Yale University
New Haven, Connecticut 06520

Robert E. Tarjan */
Computer Science Department
Stanford University
Stanford, California 94305

Abstract.

We give an algorithm which merges sorted lists represented as balanced binary trees. If the lists have lengths m and n (m < n) then the merging procedure runs in $0\left(m \log \frac{n}{m}\right)$ steps, which is the same order as the lower bound on all comparison-based algorithms for this problem.

Keywords and phrases: AVL tree, balanced tree, 2-3 tree, linear list, merging.

*/ This research was supported in part by National Science Foundation grant MCS-75-22870 and by the Office of Naval Research contract N00014-76-C-0688.

0. 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 $\lceil \lg \binom{m+n}{n} \rceil$ comparisons are necessary to distinguish these possible orderings. If we take $m < n$ then $\lceil \lg \binom{m+n}{n} \rceil = \Theta\left(m \log \frac{n}{m} \right)$.* The best merging procedure presently known within this framework is the "binary merging" algorithm of Hwang and Lin [4,5], 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 computers or on more abstract models such as pointer machines [9]. If we assume that comparisons are the only way of gaining information about key values then the $\Omega\left(m \log \frac{n}{m} \right)$

---

* That is, thelefthand side has order exactly $m \log \frac{n}{m}$; see [7] for a precise definition of the Θ and Ω notations.

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 $O(m \log \frac{n}{m})$ time on a real computer or a pointer machine. The algorithm uses balanced binary (AVL) trees [5] 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 balanced trees in Section 2, and in Section 3 we prove that the procedure runs in $O\left(m \log \frac{n}{m}\right)$ time. Section 4 gives the results of experiments comparing our algorithm with three straightforward merging methods. A high-level language implementation of the fast merging algorithm is contained in the Appendix.

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 $m < n$, such that

$$a_1 < a_2 < \ldots < a_m \quad \text{and}$$

$$b_1 < b_2 < \ldots < b_n$$

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.

[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'.

[Figure 2]

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.

This algorithm uses comparisons very efficiently, as evidenced by the small gap between the upper bound of $\lceil \lg \binom{m+n}{n} \rceil + \min(m,n)$ comparisons required for binary merging [4] and the lower bound of $\lceil \lg \binom{m+n}{n} \rceil$ comparisons for any merging method based on comparisons, But representing the lists A and B as arrays, which is the obvious way of making the individual comparisons take constant time, forces insertions to be expensive: they involve moving items over to make room for the inserted items. Hwang and Lin [4] 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. We would like to be able to merge list structures A and B and produce a result list of the same type in $O\left(m \log \frac{m}{n}\right)$ total operations.

2. Balanced Tree Merging.

Balanced binary trees [2,5] are good data structures for representing linear lists when both searches and insertions must be performed. A binary tree is called balanced if the height of the left subtree of every node never differs by no 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 balanced binary tree, the i-th element of a list becomes the i-th node visited during a symmetric order traversal of the balanced tree; if the list is sorted, as in Figure 3, then its keys appear in increasing order during such a traversal. When 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, 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 reader familiarity with Algorithm 6.2.3A, the balanced tree search and insertion algorithm of [5].

[Figure 3]

An obvious method of merging two sorted lists represented as 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 $O\left(m \log \frac{n}{m}\right)$ time.

[Figure 4]

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 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 lg 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 $O(m \log n - m \log m + m) = O(m \log \frac{n}{m})$ 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 $y \geq y$ but $y < \beta$ then y should be inserted into the right subtree of node γ ; if $y < y$ then y becomes the right offspring of x .

[Figure 5]

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 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 $a_1, a_2,$ [illegible] 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.

[Figure 6]

The general step has two parts. First the finger is retracted toward the root, just far enough so that the position of element a_{k+1} lies within the sub-tree 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 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 a 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.

[Figure 7]

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 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 those nodes must be examined while climbing upward in the tree in the first part of the general step. Bad cases may occur if we don't record these nodes and must 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 $y > \alpha$. If it turns out that $y < \alpha$, then y becomes the right offspring of x and the same situation can be repeated.

[Figure 8]

Using these ideas we can express the balanced tree merging algorithm in an Algol-like notation. (The control constructs used in this notation are adapted from Knuth [6].) 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[5]. 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 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

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] = 0

     (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
```

```
(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

(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[5]}
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 balanced tree merge}
```

3. Running Time.

In order to analyze the running time of the balanced tree merging algorithm, it is necessary to look at the details of the rebalancing procedure (steps 6 -10 of Algorithm 6.2.3A[5]). 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 balanced tree, this may cause ancestors of x in the tree to increase in height. To rebalance the tree we examine successiveancestors 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 node x was inserted. The two local transformations shown in Figure 9 will rebalance the subtree in all cases. Since the subtree rooted at 'z does not change in height, no nodes above z need be examined during the insertion.

[Figure 9]

Call a node "handled" if it is manipulated by the balanced tree merging algorithm. 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 binary 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. ▪I

Lemma 2. The total number of additions to a stack is bounded by the total number of 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, and while climbing up the path during insertions. 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. 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) deleted from the successor stack during the same insertion or (ii) has the property that y occurs in a subtree with root z such that Key(y) < Key(z) < Key(x) , where x is the node currently being inserted. (Here z is the node on top of successor when y is removed from path.)

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. Consider a node y satisfying (ii) after it has been deleted

from the path stack. Rebalancing may now 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 . 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 balanced tree merging algorithm is bounded by a constant times the number of handled nodes.

Proof. Immediate from Lemmas l-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 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 a subtree of a 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 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-l . Let T_{k-1} be the subset of handled nodes which forms a subtree with k-l or fewer terminal nodes containing all ancestors of the node inserted at the k-l -st step. Let H_{k-1} be the remaining k-l or fewer handled nodes. Each node handled during the k-th insertion is an ancestor of either the node x_{k-1} inserted at the k-l -st step or of the node x_k inserted . at the k-th step. Let T'_k be formed from T_{k-1} by adding all ancestors of x_k . Then T'_k forms a sub-tree with k or fewer terminal nodes.

The k-th 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 sub-trees 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 balanced tree of k nodes. Let T' be any subtree of T with at most ℓ terminal nodes. Then T' contains $O(\ell \log(k/\ell))$ nodes.

Proof. By Theorem 6.2.3A of [5], a balanced tree of height $h = 1.4404 \lg(k/\ell + 2) - 0.328$ must contain at least k/ℓ nodes. 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 $\{S_i\}$.

[Figure 10]

A 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 $\ell - 1 + a(1.4404 \lg(k/\ell + 2) + .672) = O(\ell \log(k/\ell))$. □

Theorem 2. The total number of nodes handled by the 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 balanced tree merging algorithm requires $O(m \log(n/m))$ time to merge lists of sizes m and n with $m < n$.

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 was 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.

4. Implementation.

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 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 balanced tree merging algorithm as Algorithm F.

One straightforward merging procedure for linear lists represented as 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 most useful when m is very small compared to n.

Another simple merging procedure for 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 stage. The final routine accepts nodes in increasing order and creates a 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 sub-trees of the root, building these subtrees recursively by the same method if they are nonempty, A more elaborate

construction which works even if the number of nodes is not known in advance is given in [5, Exercise 6.2.3-21]. Algorithm T requires $O(m+n)$ 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 balanced trees.

For the purposes of comparison, each of these algorithms was implemented in the assembly language of a hypothetical multiregister computer [6]. 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 [8, Appendix F].

The results of this evaluation are summarized in Figure 11, which gives formulas for the average running time of each of the four algorithms. Figure 12 compares the three 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 $m > 4.04n^{.253}$, and Algorithm F is faster than Algorithm T when $m \leq .355n$. Furthermore, Algorithm F is never more about 33% slower than Algorithm I, or 54% slower than Algorithm T. Thus Algorithm F seems to be a practical merging procedure for balanced trees.

[Figures 11 and 12]

In some situations the flexibility of 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, 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 rebalancing translates into an argument about the number of <u>full</u> 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 balanced trees, but as soon as a representation for 2-3 trees is specified the algorithm becomes just as complex. One possible advantage if 2-3 trees is that when they are represented as binary search trees [5, p. 469] they use only one bit per node as a balance factor.

The merging algorithm could be implemented to operate on triply-linked balanced trees [2], 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 a conventional 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 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 balanced trees are so well balanced, but it makes the worst case greater than $O(m \log(n/m))$.

Another possible scheme for fast merging is to use the linear list representation developed in [3]. This structure allows a finger into the list to be maintained such that all accesses in the neighborhood of the finger are guaranteed to be efficient. The algorithms of [3] can be extended to show that for the purposes of merging, the finger can also be moved efficiently with each access, giving an $O(m \log(n/m))$ merging algorithm. This list representation is very complicated, however, so the associated merging procedure is not "fast" in a practical sense.

Appendix. A Sail Implementation.

The following is a Sail implementation of the fast balanced tree merging algorithm . A complete description of the Sail programming language is given in [8], but the reader who is familiar with Algol W or Pascal should have little difficulty understanding the Sail constructs used below. The following points are worth noting:

1) A string constant preceding a statement is treated as a comment.
2) 'The statement 'DONE "blockName"' causes an exit from the loop on the block named "blockName".
3) RECORD POINTER parameters are passed by value.
4) The logical operations ∧ (and) and ∨ (or) are executed conditionally when evaluating an IF predicate, as in LISP but unlike Algol60. For example, the construct 'α ∧ β' means 'IF α THEN β ELSE FALSE'.

```
RECORD-CLASS Node (RECORD-POINTER(Node) lLink!, rLink!; INTEGER B!; INTEGER Key!);
COMMENT
Format of tree nodes: pointers lLink and rLink to the left and right subtrees,
an integer key, and a balance factor which is the height of the right subtree
minus the height of the left subtree, i.e.,
  B[p] = -1 ≡ node p is unbalanced to the left (left subtree is taller),
  B[p] =  0 ≡ node p is balanced,
  B[p] = +1 ≡ node p is unbalanced to the right.
We use the names leftTaller, balanced, and rightTaller respectively for these
values. The only relation between them which is significant to the program is
that leftTaller = -rightTaller.;

RECORD-CLASS ListHeader (RECORD-POINTER(Node) Root!; INTEGER Height!, Size!);
COMMENT
Format of list header: pointer to the root of the balanced tree, plus an
integer giving the height of tree, and an integer giving the number of nodes in
the tree.;

COMMENT Abbreviations for Node and ListHeader fields;
DEFINE lLink = {Node:lLink!};
DEFINE rLink = {Node:rLink!);
DEFINE B =     {Node:B!};
DEFINE Key =   {Node:Key!};
DEFINE Root =  {ListHeader:Root!};
DEFINE Height = {ListHeader:Height!};
DEFINE Size =  {ListHeader:Size!};

COMMENT Manifest constants;
DEFINE balanced = {0};
DEFINE leftTaller = {-1};
DEFINE rightTaller = {+1};
DEFINE maxDepth = (24);
```

```
PROCEDURE FastMerge(RECORD_POINTER(ListHeader) src, dst);
BEGIN "FastMerge"
COMMENT
The FastMerge procedure performs merging of sorted lists represented as balanced
binary trees. The two lists are passed to FastMerge by passing the two pointers
src and dst to their respective list header nodes: the src list is empty on
return from FastMerge, and the dst list contains the result of merging the two
lists.

The merging algorithm is best viewed as containing two relatively independent
processes, the dismantling process and the insertion process. (In fact, the
most natural program structure for the FastMerge procedure would use coroutines
for these processes.) The dismantling process operates on the smaller of the
two lists, which contains m nodes. It performs a symmetric-order traversal of
the binary tree representing this list, lopping off the nodes in order of
increasing key size, and supplies these nodes to the insertion process upon
demand. The dismantling process runs in O(m) steps.

The insertion process inserts these nodes successively into their proper
position in the larger list, which contains n nodes. The details of the
insertion algorithm are complicated, but the idea is simple. The first
insertion is performed using the normal tree search and insertion algorithm.
The subsequent insertions are not independent from one another, since the
insertions are done in increasing order. So the algorithm performs these
insertions by first searching upward from the site of the previous insertion for
the root of a subtree which can be guaranteed to contain the node being
inserted. Then the insertion is completed by the usual procedure. The
insertion process runs in O(m log(n/m)) steps, so the running time of the entire
merging algorithm is also O(m log(n/m)).;

  RECORD_POINTER(Node) ARRAY dStk[1:maxDepth]; INTEGER dPtr;
  COMMENT
  The dStk array is used as a stack, containing nodes not yet output during the
  dismantling process, and the integer dPtr is its stack pointer. This
  structure is used by the procedures InitDismantle and GetNext below.;

  PROCEDURE InitDismantle(RECORD_POINTER(ListHeader) Head);
  BEGIN "InitDismantle"
  COMMENT
  This procedure initializes a 'stream' which produces the nodes of the list
  headed by Head. The nodes come from the stream in increasing order of Key
  value, one node per call to GetNext. The list is destroyed in this process,
  so InitDismantle sets all fields of Head to a null state.;
    IF Size[Head] = 8 THEN dPtr ← 8
                      ELSE dStk[dPtr ← 1] ← Root[Head];
    Root[Head] ← NULL-RECORD; Size[Head] ← Height[Head] ← 0
  END "InitDismantle";

  RECORD_POINTER(Node) PROCEDURE GetNext;
  BEGIN "GetNext"
  COMMENT
  A call to this procedure returns the next node in the list given to
  InitDismantle, with lLink = rLink = NULL-RECORD and B = 0. If no nodes remain
  in the list, the value NULL-RECORD is returned.;
    RECORD-POINTER(Node) Nxt, Nxt1;
    IF dPtr = 8 THEN RETURN(NULL_RECORD);
    Nxt ← dStk[dPtr]; dPtr ← dPtr-1;
    WHILE lLink[Nxt] ≠ NULL_RECORD DO BEGIN
      Nxt1 ← lLink[Nxt]; lLink[Nxt] ← NULL-RECORD;
      dPtr ← dPtr+1; dStk[dPtr] ← Nxt;
      Nxt ← Nxt1
    END;
```

```
  IF rLink[Nxt] ≠ NULL-RECORD THEN BEGIN
    dPtr ← dPtr+1;  dStk[dPtr] ← rLink[Nxt];
    rLink[Nxt] ← NULL-RECORD
  END;
  B[Nxt] ← balanced;
  RETURN(Nxt)
END "GetNext";

RECORD-POINTER(Node) ARRAY pathStk [1:maxDepth]; INTEGER pathPtr;
INTEGER ARRAY succStk [1:maxDepth];  INTEGER succPtr;

"Invariant 'PathProp' :

  The pathStk contains an initial segment of the path from the root of lgLst
  (as modified by the insertions so far from smLst) to the position which some
  node  z (specified when this invariant is applied below) from smLst has, or
  will have after an insertion, in lgLst.
  The succStk contains the indices of all pathStk entries whose lLinks are
  also in pathStk, i.e. all nodes on the path (excluding the last) which are
  greater than the last node inserted."

PROCEDURE InitInsertion(RECORD_POINTER(ListHeader) Head);
BEGIN "InitInsertion"
COMMENT
This procedure initializes the insertion process on the list headed by Head,
and sets all of the fields of Head to a null state.;
  pathPtr ← 1;  pathStk[pathPtr ] ← Root[Head];
  succPtr ← 8;
  Root[Head] ← NULL-RECORD; Size[Head] . Height[Head] ← 8
END "InitInsertion";

RECORD-POINTER(ListHeader)  smLst, lgLst;
RECORD-POINTER(Node) p, q, r, s, t, x;
INTEGER m, n, ht, insCount, sPtr, a, k;
```

```
"Initializations."

IF Size[dst] ≥ Size[src] THEN BEGIN lgLst ← dst; smLst ← src END
                         ELSE BEGIN smLst ← dst; lgLst ← src END;
m ← Size[smLst]; n ← Size[lgLst]; ht ← Height[lgLst];
InitDismantle(smLst); InitInsertion(lgLst);


"The insertion process."

FOR insCount ← 1 STEP 1 UNTIL m DO BEGIN "InsertLoop"
  x ← GetNext;
  k ← Key[x];

  "Now x is the next node from smLst to be inserted into lgLst, with lLink[x]=
  rLink[x]=NULL_RECORD, B[x]=balanced, and k=Key[x ]. PathProp holds with z =
  the previous node inserted into lgLst; on the first insertion PathProp does
  not hold, but succStk is empty so UpLoop below is never executed. The
  purpose of UpLoop is to make PathProp hold with z = x, by retracting the
  path as little as possible toward the root."

  WHILE succPtr ≠ 8 DO BEGIN "UpLoop"
    IF k < Key[pathStk[succStk[succPtr]]] THEN DONE "UpLoop";
    pathPtr ← succStk[succPtr]; succPtr ← succPtr-1
  END "UpLoop";
  p ← pathStk[pathPtr];

  "Now x and k are as before, and p is on top of pathStk. Also, PathProp
  holds with z = x. The purpose of SearchLoop is to maintain this property
  while extending the path to a leaf of lgLst, and then to add x to lgLst and
  to the path."

  WHILE TRUE DO BEGIN "SearchLoop"
    IF k < Key[p] THEN BEGIN "Move left"
      succPtr ← succPtr+1; succStk[succPtr] ← pathPtr;
      q ← lLink[p];
      IF q = NULL-RECORD THEN BEGIN lLink[p] ← x; DONE "SearchLoop" END
    END
    ELSE BEGIN "Move right"
      q ← rLink[p];
      IF q = NULL-RECORD THEN BEGIN rLink[p] ← x; DONE "SearchLoop" END
    END;
    p ← q;
    pathPtr ← pathPtr+1; pathStk[pathPtr] ← p
  END "SearchLoop";
  pathPtr ← pathPtr+1; pathStk[pathPtr] ← x;

  "Now PathProp holds with z = x, and in fact x is on top of pathStk. The
  purpose of AdjustLoop is to adjust all of the balance factors on the path
  between x and s, which is defined to be the first unbalanced node on the
  path above x (the root if there are no unbalanced nodes on the path..)
  AdjustLoop does not alter the path."

  sPtr ← pathPtr-1;
  WHILE TRUE DO BEGIN "AdjustLoop"
    s ← pathStk[sPtr];
    IF B[s] ≠ balanced v sPtr=1 THEN DONE "AdjustLoop";
    B[s] ← (IF k < Key[s] THEN leftTaller ELSE rightTaller);
    sPtr ← sPtr-1
  END "AdjustLoop";
  a ← (IF k < Key[s] THEN leftTaller ELSE rightTaller);
```

```
      "The purpose of the following is to maintain balance in the subtree rooted
      at s.  In two cases this is trivial,  and the path is not affected.  In the
      third case rebalancing must take place,  which invalidates a portion of the
      path; this portion is discarded,  and the root of the rebalanced subtree
      becomes the final node on the path.  In any case, PathProp will still hold
      with z = x."

      IF B[s] = balanced THEN BEGIN
        B[s] ← a;  ht ← ht+1
        END
      ELSE IF B[s] = -a THEN BEGIN
        B[s] ← balanced
      END
      ELSE BEGIN "Rebalance"
        r ← pathStk[sPtr+1];
        IF B[r] = a THEN BEGIN "SingleRotation"
          p ← r;
          IF a = rightTaller THEN BEGIN rLink[s] ← lLink[r]; lLink[r] ← s END
                             ELSE  BEGIN lLink[s] ← rLink[r]; rLink[r] ← s END;
          B[s] ← B[r] ← balanced;
        END "SingleRotation"
        ELSE BEGIN "DoubleRotation"
          IF a = rightTaller THEN BEGIN
            p ← lLink[r];  lLink[r] ← rLink[p]; rLink[p] ← r;
            rLink[s] ← lLink[p]; lLink[p] ← s
          END
          ELSE BEGIN,
            p ← rLink[r];  rLink[r] ← lLink[p]; lLink[p] ← r;
            lLink[s] ← rLink[p]; rLink[p] ← s
          END;
          B[s] ← (IF B[p] = +a THEN -a ELSE balanced);
          B[r] ← (IF B[p] = -a THEN +a ELSE balanced);
          B[p] ← balanced;
        END "DoubleRotation";
        IF sPtr > 1 THEN BEGIN
          t ← pathStk[sPtr-1];
          IF s = rLink[t] THEN rLink[t] ← p
                          ELSE lLink[t] ← p
        END;
        "The tree is rebalanced;  delete the invalidated section from pathStk and
        succStk."
        pathPtr ← sPtr;  pathStk[pathPtr] ← p;
        WHILE succPtr>0 ∧ succStk[succPtr]≥pathPtr DO
          succPtr ← succPtr-1;
      END  "Rebalance";

      "Now PathProp holds with z = x."

    END "InsertLoop";

    Root[dst] ← pathStk[1];  Height[dst] ← ht; Size[dst] ← m + n

  END "FastMerge";
```

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[2] Clark A. Crane, "Linear lists and priority queues as balanced binary trees," Ph.D. thesis, Computer Science Dept., Stanford University, STAN-CS-72-259, (February 1972), 131 pp.

[3] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts, 'A new representation for linear lists,' Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, Boulder, Colorado, (1977), 49-60.

[4] Frank K. Hwang and Shen Lin, "A simple algorithm for merging two disjoint linearly ordered sets," SIAM J. Comput. 1, 1 (March 1972), 31-39.

[5] Donald E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[6] Donald E. Knuth, "Structured programming with goto statements," Computing Surveys 6, 4 (December 1974), 261-301.

[7] Donald E. Knuth, "Big omicron and big omega and big theta," SIGACT News 8, 2 (April 1976), 18-24.

[8] John F. Reiser, ed., "SAIL," Stanford Computer Science Department Report STAN-CS-76-575, (August 1976), 173 pp.

[9] Robert E. Tarjan, "Reference machines require non-linear time to maintain disjoint sets," Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, Boulder, Colorado, (1977), 19-29.

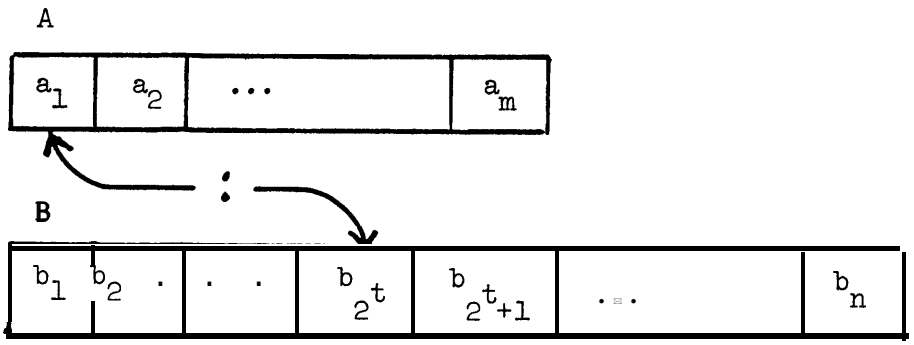


Figure 1. First comparison during a binary merge.

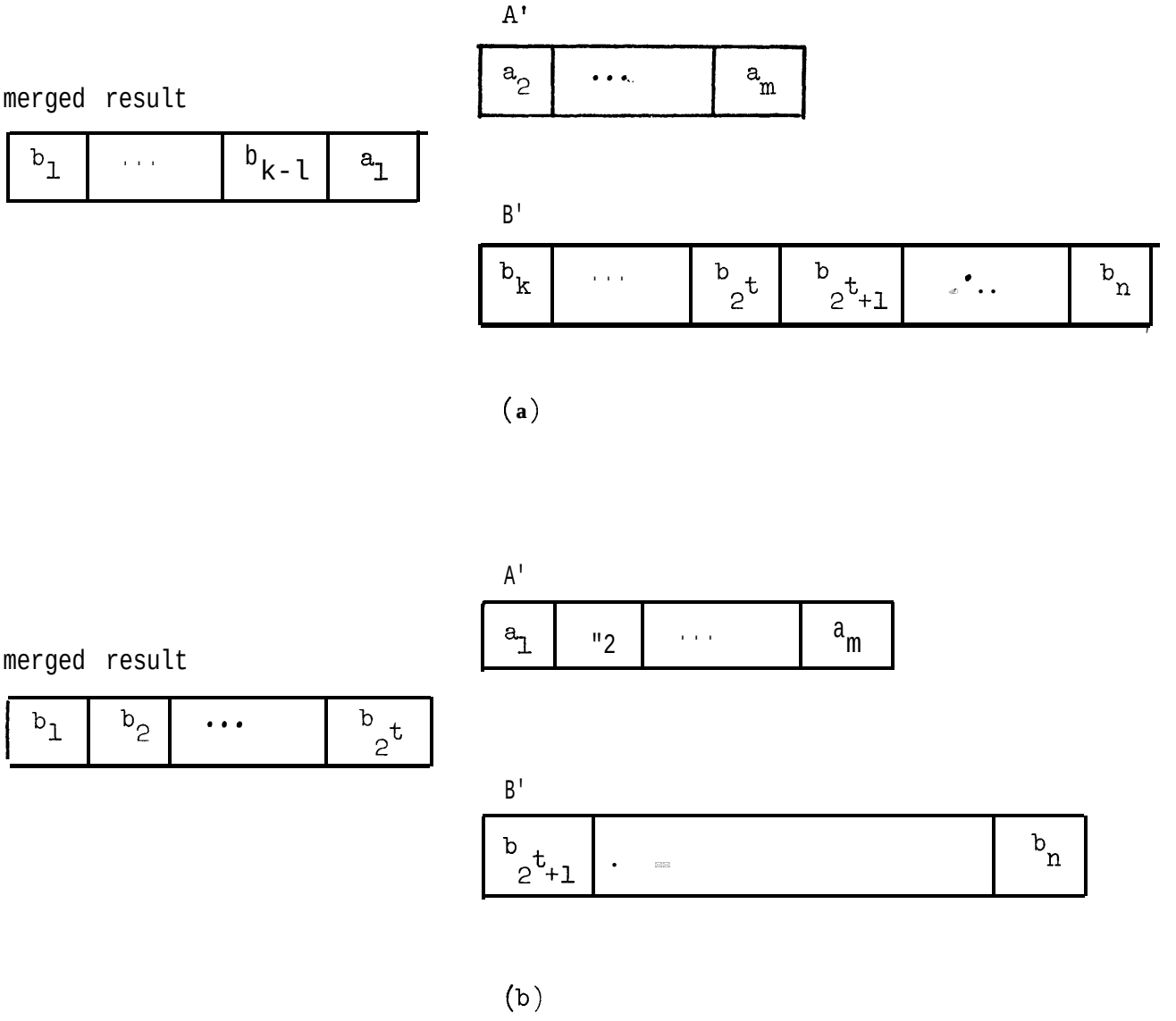


Figure 2. Outcomes after first part of a binary merge.

(a) $a_1 < b_{2^t}$.

(b) $a_1 > b_{2^t}$.

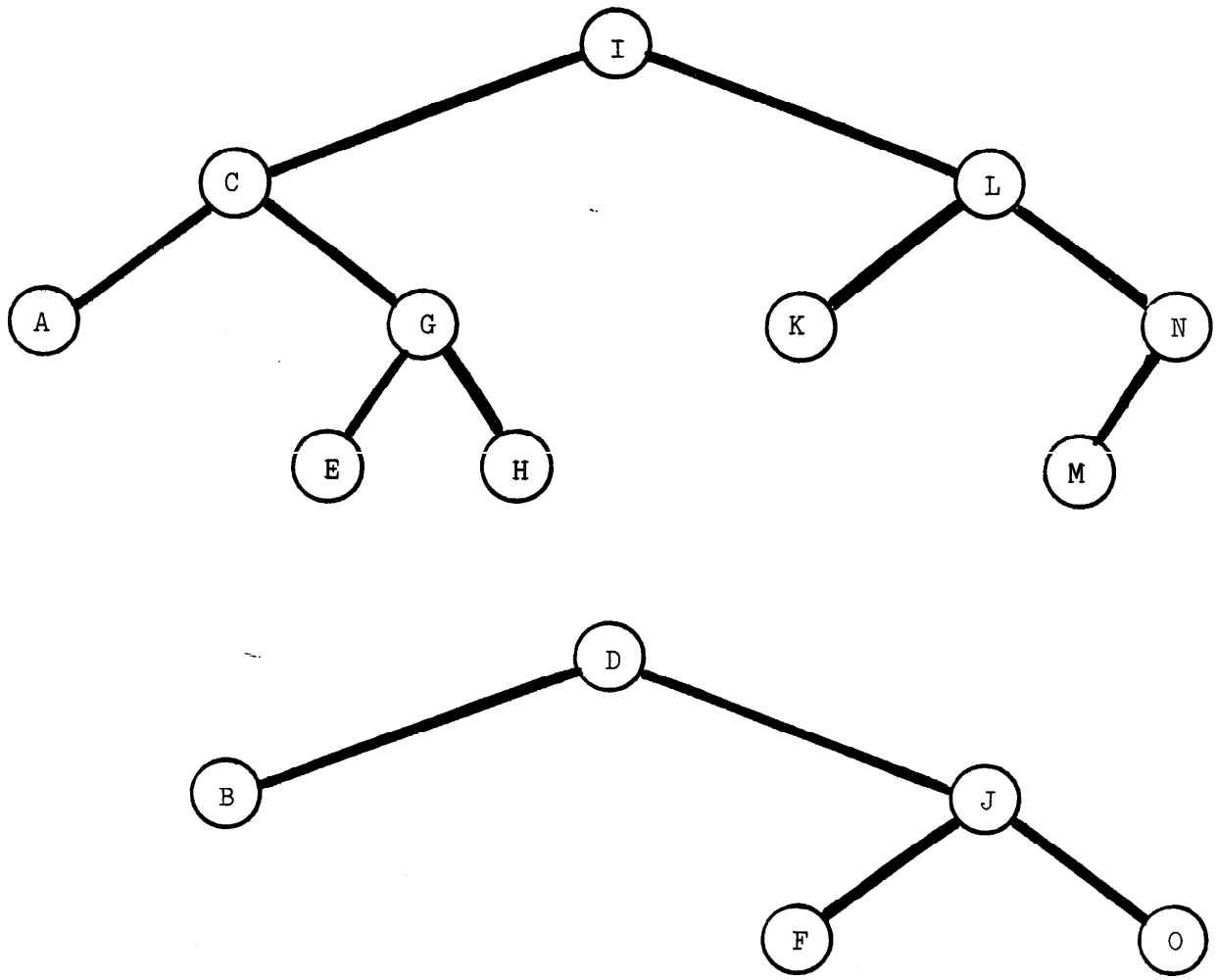


Figure 3. Sorted lists represented as balanced binary trees.

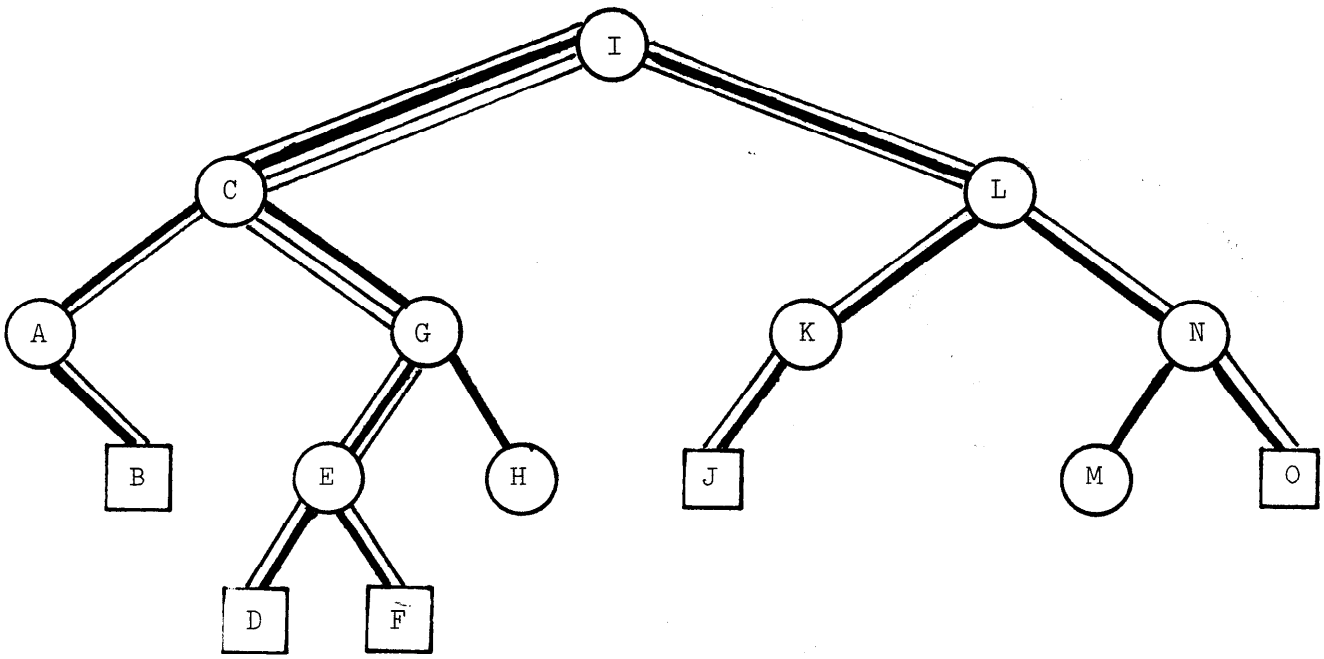


Figure 4. An example of merging by independent insertions (square nodes have been inserted).

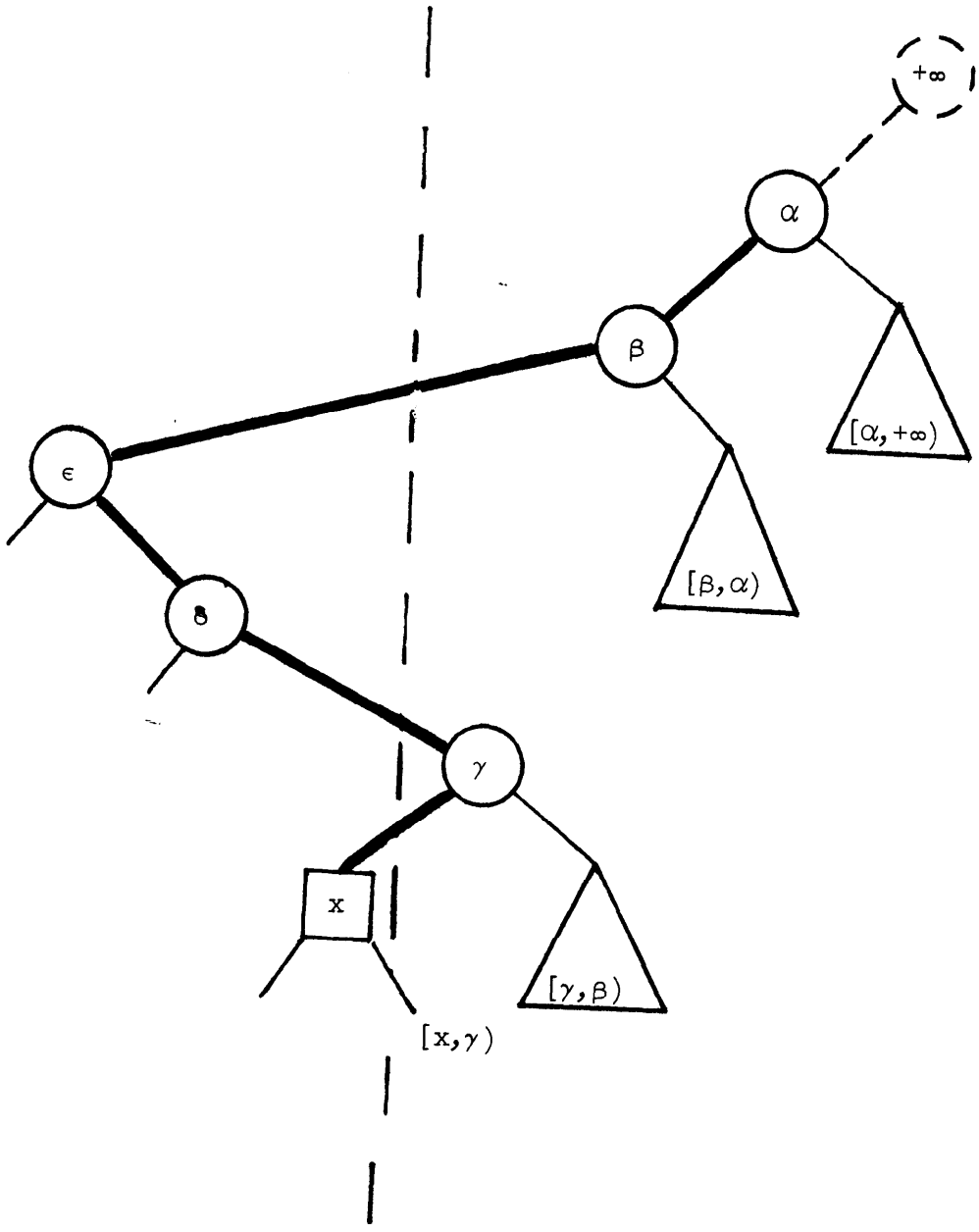
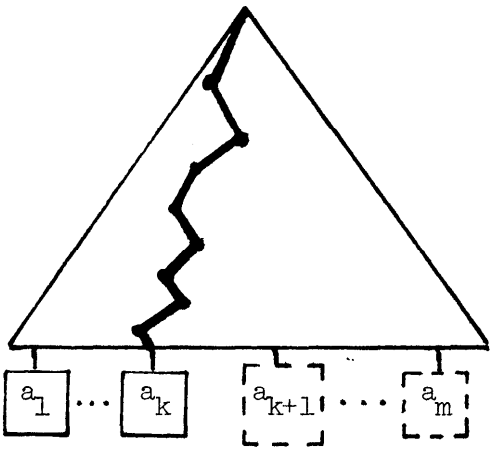
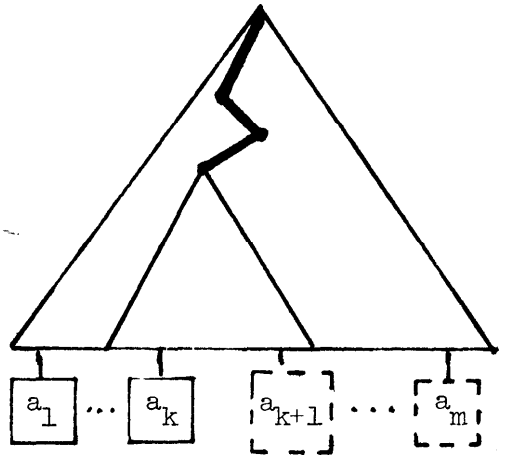


Figure 5. Inserting an item larger than x . (Subtrees are labeled with the range of key values which may be inserted.)

(a)



(b)



( )

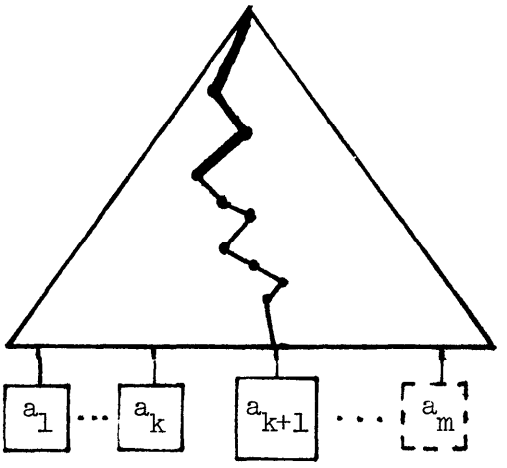


Figure 6. Insertion using the finger path.

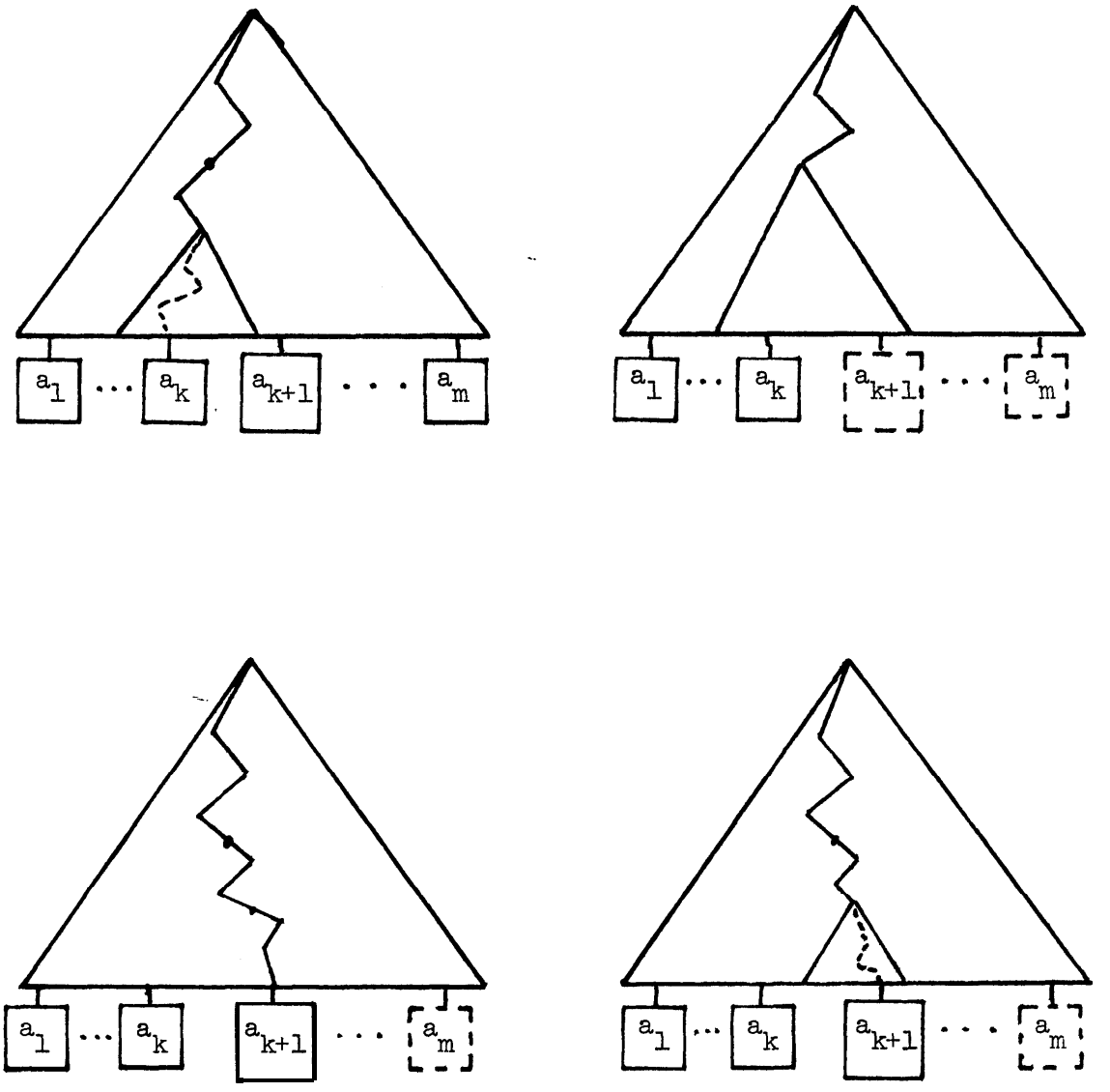


Figure 7. Retracting the path for rebalancing.

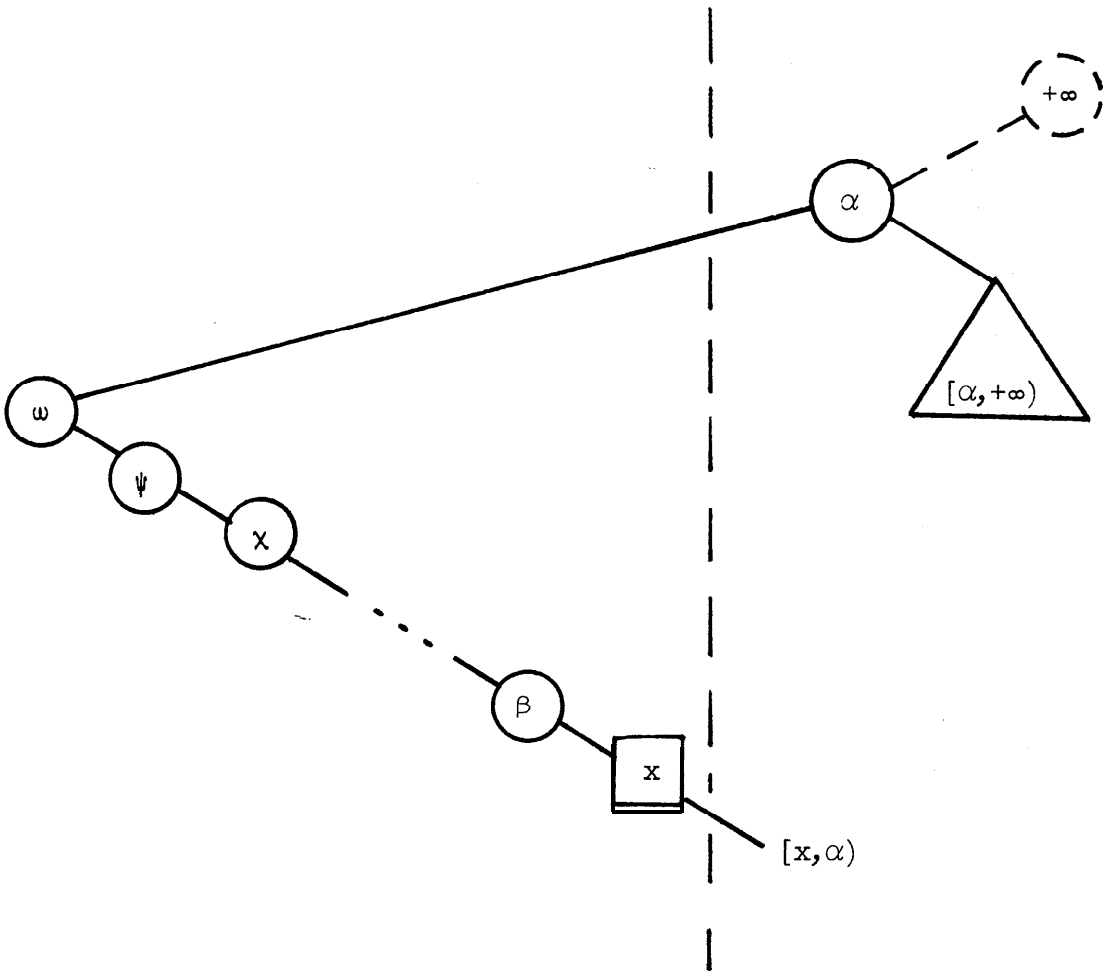
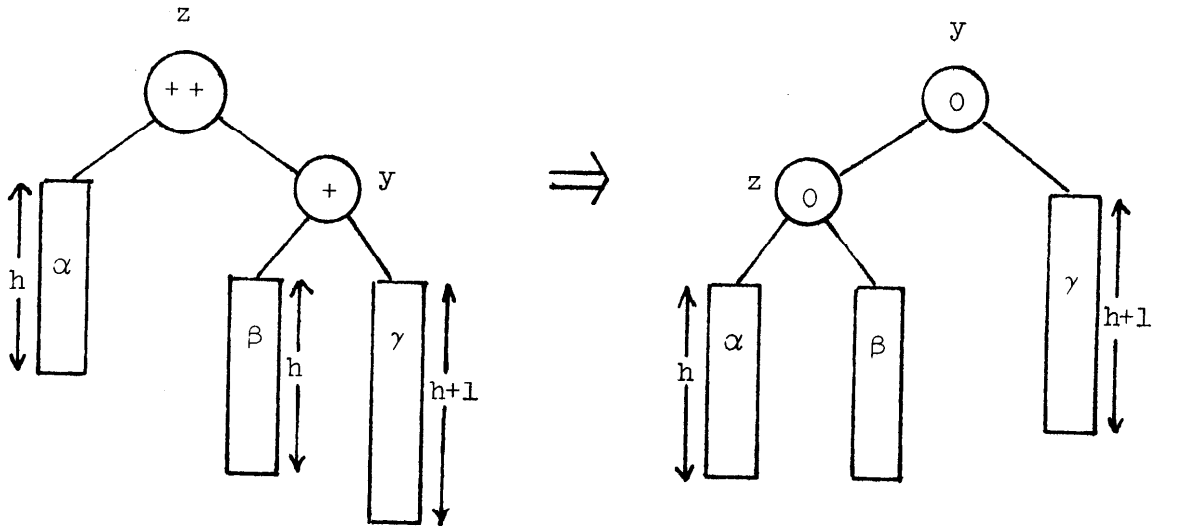
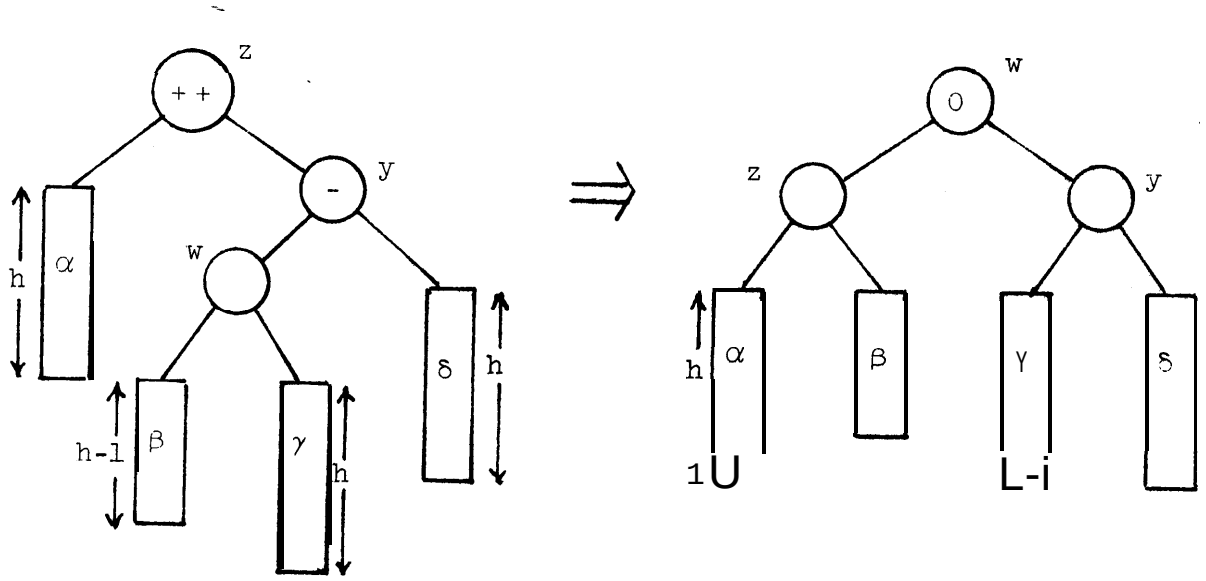


Figure 8. A bad insertion.



(a)



(b)

Figure 9. Rebalancing after an insertion.
(a) Case 1. Sub-tree γ contains inserted node x .
(b) Case 2. Either x = w and β and y are empty, or subtree y contains x .

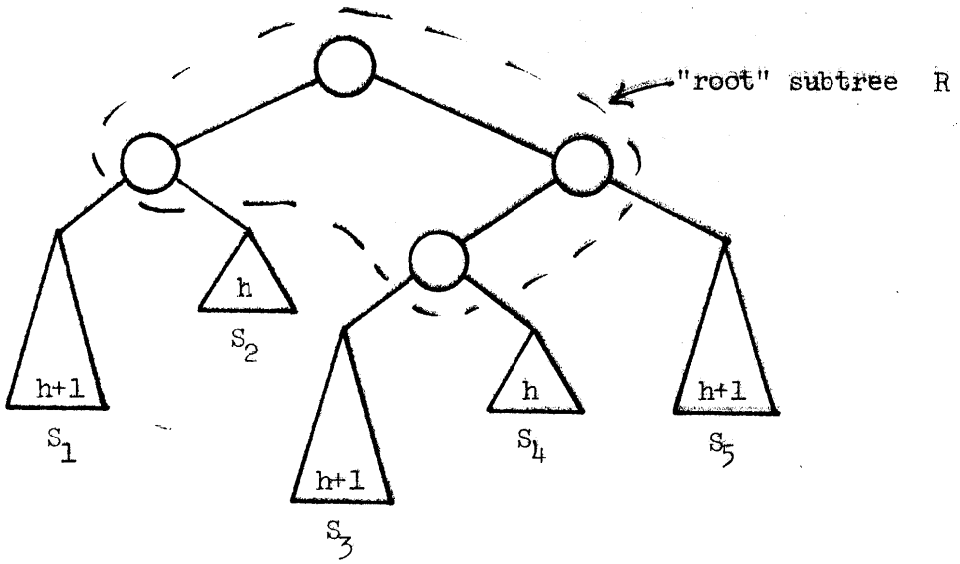


Figure 10. Subdivision of a balanced tree.

	average running time to merge lists of sizes m and n, with $m<n$
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	$10(m+n) + 4m + 32$

Figure 11. Comparison of methods.

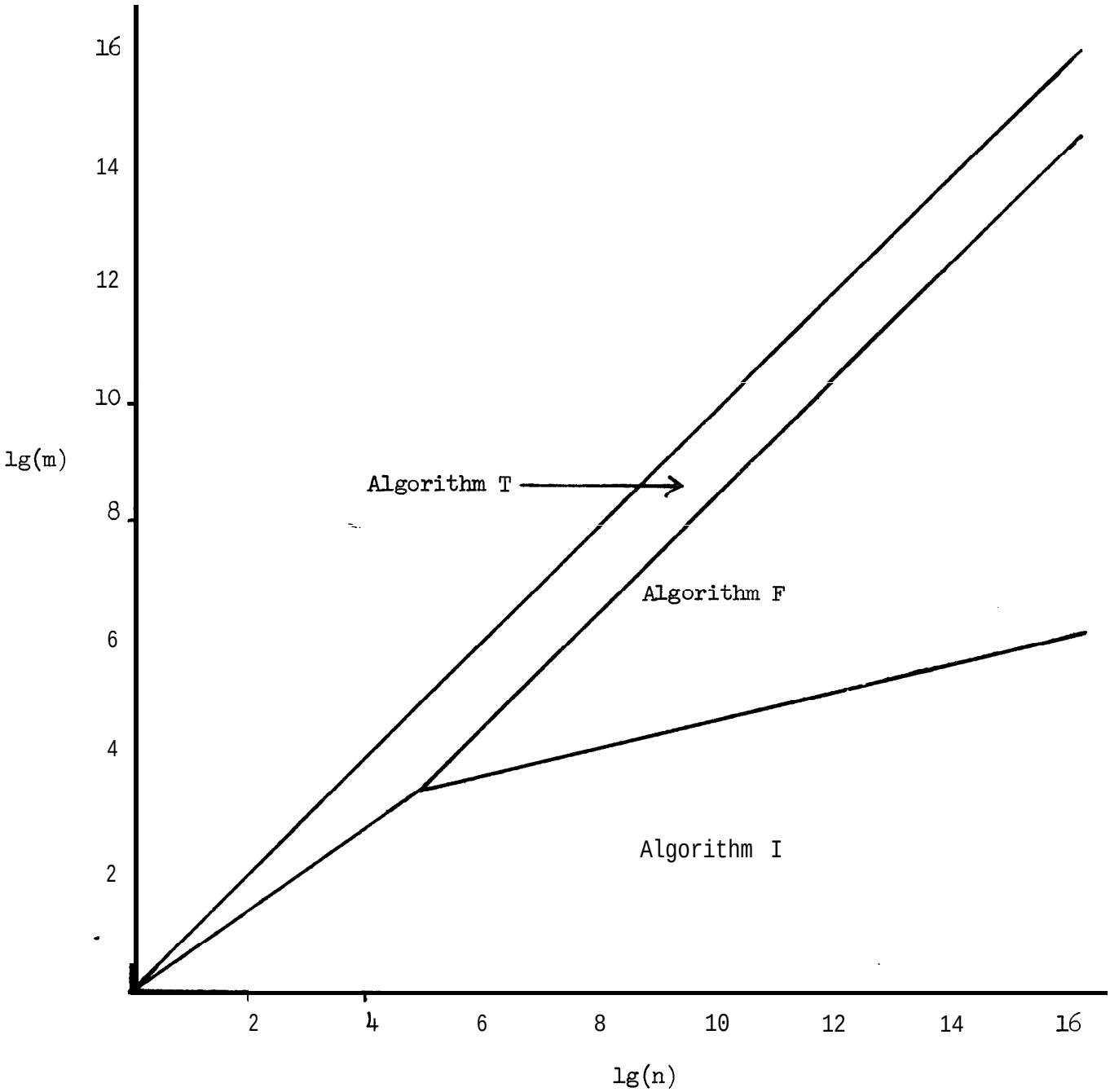


Figure 12. Zones of best performance for balanced tree merging algorithms.