DESIGN AND ANALYSIS OF A DATA STRUCTURE FOR REPRESENTING SORTED LISTS

by

Mark R. Brown and Robert E. Tarjan

STAN-CS-78-7Ø9
December 1978

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Design and Analysis of a Data Structure for Representing Sorted Lists

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

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

December, 1978

Abstract.

In this paper we explore the use of 2-3 trees to represent sorted lists. We analyze the worst-case cost of sequences of insertions and deletions in 2-3 trees under each of the following three assumptions: (i) only insertions are performed; (ii) only deletions are performed; (iii) deletions occur only at the small end of the list and insertions occur only away from the small end. Cur analysis leads to a data structure for representing sorted lists when the access pattern exhibits a (perhaps time-varying) locality of reference. This structure has many of the properties of the representation proposed by Guibas, McCreight, Plass, and Roberts [4], but it is substantially simpler and may be practical for lists of moderate size.

Keywords: analysis of algorithms, deletion, finger, insertion, sorted list, 2-3 tree.

[*] Research partially supported by the National Science Foundation, Grant MCS-75-22870 A02, the Office of Naval Research, Contract N00014-76-C-0688, and a Guggenheim Fellowship. Reproduction in whole or in part is permitted for any purpose of the United States government.

0. Introduction.

The 2-3 tree [1] is a data structure which allows both fast accessing and fast updating of stored information. For example, 2-3 trees may be used to represent a sorted list of length n so that a search for any item in the list takes $O(\log n)$ steps. Once the position to insert a new item or delete an old one has been found (via a search), the insertion or deletion can be performed in $O(\log n)$ additional steps.

If each insertion or deletion in a 2-3 tree is preceded by a search requiring $\Omega(\log n)$ time,[*] then there is little motivation for improving the above bounds on the worst-case time for insertions and deletions. But there are several applications of 2-3 trees in which the regularity of successive insertions or deletions allows searches to proceed faster than $\Omega(\log n)$. One example is the use of a sorted list represented as a 2-3 tree to implement a priority queue [6, p. 152]. In a priority queue, insertions are allowed anywhere, but only the smallest item in the list at any moment can be deleted. Since no searching is ever required to find the next item to delete, an improved bound on the cost of consecutive deletions might lead to a better bound on the cost of the method as a whole.

In this paper, we prove several results about the cost of sequences of operations on 2-3 trees. In Section 1 we derive a bound on the total

[*] A function $g(n)$ is $\Omega(f(n))$ if there exist positive constants c and n_0 with $g(n) \geq cf(n)$ for all $n \geq n_0$; it is $\Theta(f(n))$ if there exist positive constants c_1, c_2, and n_0 with $c_1f(n) < g(n) \leq c_2f(n)$ for all $n \geq n_0$. Hence the 'Θ' can be read 'order exactly' and the 'Ω' as 'order at least'; Knuth [7] gives further discussion of the Θ and Ω notations.

cost of a sequence of insertions (as a function of the positions of the insertions in the tree) which is tight to within a constant factor. In Section 2 we derive a similar bound for a sequence of deletions. If the sequence of operations is allowed to include intermixed insertions and deletions, there are cases in which the naive bound cannot be improved: $\Theta(\log n)$ steps per operation may be required. However, we show in Section 3 that for the priority queue application mentioned above, a mild assumption about the distribution of insertions implies that such bad cases cannot occur.

In Section 4 we explore some consequences of these results. We propose a modification of the basic 2-3 tree structure which allows us to save a finger to an arbitrary position in the tree, with the property that searching d positions away from the finger costs $O(\log d)$ steps (independent of the tree size). Fingers are inexpensive to move, create, or abandon, and several fingers into the same structure can be maintained simultaneously. We use the bound on sequences of insertions to show that even when fingers are used to speed up the searches, the cost of a sequence of insertions is dominated by the cost of the searches leading to the insertions. The same result holds for a sequence of deletions and for a sequence of intermixed insertions and deletions satisfying the assumptions of Section 3. Our structure is similar to one proposed earlier by Guibas, McCreight, Plass, and Roberts [4], but it is much simpler to implement and may be practical for representing moderate-sized lists. Their structure has the interesting property that individual insertions and deletions are guaranteed to be efficient, while operations on our structure are efficient only when averaged over a sequence. Our structure has the compensating advantage that fingers are much easier to move. An obvious generalization of our structure to B-trees [2] makes it suitable for larger lists kept in secondary storage.

In the final section we discuss some practical issues arising in an implementation of the structure, describe some of its applications, and indicate directions for future work.

1. Insertions into 2-3 Trees.

A 2-3 tree [1,6] is a tree such that 2- or 3-way branching takes place at every internal node, and all external nodes occur on the same level. An internal node with 2-way branching is called a <u>2-node</u>, and one with 3-way branching a <u>3-node</u>. It is easy to see that the height of a, 2-3 tree with n external nodes lies between $\lceil \log_3 n \rceil$ and $\lfloor \lg n \rfloor$.[*] An example of a 2-3 tree is given in Figure 1.

[Figure 1]

There are several schemes for associating data with the nodes of a 2-3 tree; the usefulness of a particular organization depends upon the operations to be performed on the data. All of these schemes use essentially the same method for updating the tree structure to accomodate insertions, where <u>insertion</u> means the addition of a new external node at a given position in the tree. (Sometimes the operation of insertion is considered to include searching for the position to add the new node, but we shall consistently treat searches separately in what follows.)

Insertion is accomplished by a sequence of node expansions and splittings, as shown by example in Figure 2. When a new external node is attached to a terminal node p (an internal node having only external nodes as offspring), this node <u>expands</u> to accomodate the extra edge. If p was a 2-node prior to the expansion, it is now a 3-node, and the insertion is complete, If p was a 3-node prior to expansion, it is now a "4-node", which is not allowed in a 2-3 tree; therefore, p is <u>split</u> into a pair of 2-nodes. This split causes an expansion of p's parent, and the process repeats until either a 2-node expands into a 3-node or the root is split. If

[*] We use lg n to denote $\log_2 n$.

the root splits, a new 2-node is created which has the two parts of the old root as its children, and this new node becomes the root. An insertion in a 2-3 tree can be accomplished in $\Theta(1+s)$ steps, where s is the number of node splittings which take place during the insertion.

[Figure 2]

One way to represent a sorted list using a 2-3 tree is shown in Figure 3. The elements of the list are assigned to the external nodes of the tree, with key values of the list elements increasing from left to right. Keys from the list elements are also assigned to internal nodes of the tree in a "symmetric" order analogous to that of binary search trees. More precisely, each internal node is assigned one key for each of its sub-trees other than the rightmost, this key being the largest which appears in an external node of the subtree. Therefore each key except the largest appears in an internal node, and by starting from the root of the tree we can locate any element of the list in $O(\log n)$ steps, using a generalization of binary tree search. (Several 2-3 search tree organizations have been proposed which are similar but not identical to this one [1, p. 147; 6, p. 468].)

[Figure 3]

Any individual insertion into a 2-3 tree of size n can cause up to about $\lg n$ splittings of internal nodes to take place. On the other hand, if n consecutive insertions are made into a tree initially of size n, the total number of splits is bounded by about $\frac{3}{2}n$ instead of $n \lg n$, because each split generates a new internal node and the number of internal nodes is initially at least $(n-1)/2$ and finally at most $2n-1$. The following theorem gives a general bound on the worst-case splitting which can occur due to consecutive insertions into a 2-3 tree.

Theorem 1. Let T be a 2-3 tree of size n , and suppose that k insertions are made into T . If the positions of the newly-inserted nodes in the resulting tree are p_1 < P2 <.** < p_k , then the number of node splittings which take place during the insertions is bounded by

$$2\left(\lceil \lg(n+k)\rceil + \sum_{1<i\leq k} \lceil \lg(p_i-p_{i-1}+1)\rceil\right) .$$

The proof divides into two parts. In the first part, we define a rule for (conceptually) marking nodes during a 2-3 tree insertion. This marking rule has two important properties when a sequence of insertions is made: the number of marked nodes bounds the number of splits, and the marked nodes are arranged to form paths from the inserted external nodes toward the root of the tree.

The effect of marking the tree in this way is to shift our attention from dealing with a dynamic situation (the 2-3 tree as it changes due to insertions) to focus on a static object (the 2-3 tree which results from the sequence of insertions). The second part of the proof then consists of showing that in any 2-3 tree, the number of nodes lying on the paths from the external nodes in positions $p_1 < p_2 < \ldots < p_k$ to the root is bounded by the expression given in the statement of the theorem.

We now define the marking rule described above. On each insertion into a 2-3 tree, one or more nodes are marked as follows:

(1) The inserted (external) node is marked.

(2) When a marked node splits, both resulting nodes are marked. When an unmarked node splits, a choice is made and one of the resulting nodes is marked; if possible, a node is marked which has a marked child.

We establish the required properties of these rules by a series of lemmas.

Lemma 1. After a sequence of insertions, the number of marked internal nodes equals the number of splits.

Proof. No nodes are marked initially, and each split causes the number of marked internal nodes to increase by one. □

Lemma 2. If a 2-node is marked, then at least one of its children is marked; if a 3-node is marked, then at least two of its children are marked.

Proof. We use induction on the number of marked internal nodes. Since both assertions hold vacuously when there are no marked internal nodes, it is sufficient to show that a single application of the marking rules preserves the assertions. There are two cases to consider when a 3-node X splits:

Case 1. X is marked. Then before the insertion which causes X to split, X has at least two marked children. When the insertion expands X to overflow, this adds a thirdmarked child (by rule 1 or rule 2). Thus the two marked 2-nodes which result from the split of X each have at least one marked child.

Case 2. X is unmarked. Then before the insertion which causes X to split, X may have no marked children. When the insertion expands X to overflow, a new marked child is created. Thus the single marked 2-node which results from the split of X can be chosen to have a marked child.

A marked 3-node is created when a marked 2-node expands. This expansion always increases the number of marked children by one. Since a marked 2-node has at least one marked child, it follows that a marked 3-node has at least two marked children. □

Lemma 3. After a sequence of insertions, there is a path of marked nodes from any marked node to a marked external node.

Proof. Obvious from Lemma 2. □

Lemma 4. The number of splits in a sequence of insertions is no greater than the number of internal nodes in the resulting tree which lie on paths from the inserted external nodes to the root.

Proof. Immediate from Lemmas 1 and 3. □

This completes the first part of the proof as outlined earlier; to finish the proof we must bound the quantity in Lemma 4. We shall require the following two facts about binary arithmetic. For any non-negative integer k, let $\nu(k)$ be the number of one bits in the binary representation of k.

Lemma 5 [5, p. 483 (answer to ex.1.2.6-11)]. Let a and b be non-negative integers, and let c be the number of carries when the binary representations of a and b are added. Then $v(a) + \nu(b) = \nu(a+b) + c$.

Lemma 6. Let a and b be non-negative integers such that $a < b$ and let i be the number of bits to the right of and including the leftmost bit in which the binary representations of a and b differ. Then $i \leq \nu(a) - \nu(b) + 2\lceil \lg(b-a+1) \rceil$.

Proof. If k is any positive integer, the length of the binary representation of k is $\lceil \lg(k+1) \rceil$. Let c be the number of carries when a and b-a are added. By Lemma 5, $\nu(a) + \nu(b-a) = \nu(b) + c$. When a and b-a are added, at least $i - \lceil \lg(b-a+1) \rceil$ carries are required to produce a number which differs from a in the i-th bit. Thus $i - \lceil \lg(b-a+1) \rceil \leq c$. Combining inequalities, we find that

$$i \leq c + \lceil \lg(b-a+1) \rceil \leq v(a) - \nu(b) + \nu(b-a) + \lceil \lg(b-a+1) \rceil$$

$$\leq \nu(a) - \nu(b) + 2\lceil \lg(b-a+1) \rceil \; . \qquad .$$

Lemma 7. Let T be a 2-3 tree with n external nodes numbered $0, 1, \ldots, n-1$ from left to right. The number M of nodes (internal and external) which lie on the paths from external nodes $p_1 < p_2 < \ldots < p_k$ to the root of T satisfies

$$M \leq 2\left(\lceil \lg n \rceil + \sum_{1 < i < k} \lceil \lg(p_i - p_{i-1} + 1) \rceil \right) \; .$$

Proof. For any two external nodes p and q , let $M(p,q)$ be the number of nodes which are on the path from q to the root but not on the path from p to the root. Since the path from p_1 to the root contains at most $\lceil \lg n \rceil + 1$ nodes, we have

$$M \leq \lceil \lg n \rceil + 1 + \sum_{1<i<k} M(p_{i-1},p_i) \ .$$

We define a label ℓ for each external node as follows. If t is an internal node of T which is a 2-node, we label the left edge out of t with a 0 and the right edge out of t with a 1 . If t is a 3-node, we label the left edge out of t with a 0 and the middle and right edges out of t with a 1 . Then the label L(p) of an external node p is the integer whose binary representation is the sequence of 0's and 1's on the path from the root to p .

Note that if p and q are external nodes such that q is the right neighbor of p , then $\ell(q) \leq \ell(p)+1$. It follows by induction that $\ell(p_i) - \ell(p_{i\ 1}) \leq p_i - p_{i\ 1}$ for $1 < i < k$.

Consider any two nodes p_{i-1} , p_i . Let t be the internal node which is farthest from the root and which is on the path from the root to p_{i-1} and on the path from the root to p_i . We must consider two cases.

Case 1. The edge out of t leading toward p_{i-1} is labelled 0 and the edge out of t leading toward p_i is labelled 1 . Then $\ell(p_i) > \ell(p_{i-1})$. Furthermore $M(p_{i-1},pi)$, which is the number of nodes on the path from t to p_i (not including t), is equal to the number of bits to the right of and including the leftmost bit in which the binary representations of $\ell(p_{i-1})$ and $\ell(p_i)$ differ. By Lemma 6,

$$\begin{aligned} M(p_{i-1},p_i) &\leq \nu(\ell(p_{i-1})) - \nu(\ell(p_i)) + 2\lceil \lg(\ell(p_i) - \ell(p_{i-1})+1)\rceil \\ &\leq \nu(\ell(p_{i-1})) - \nu(\ell(p_i)) + 2\lceil \lg(p_i-p_{i-1}+1)\rceil \ . \end{aligned}$$

<u>Case 2</u>. The edge out of t leading toward p_{i-1} is labelled 1 and the edge out of t leading toward p_i is also labelled 1 . Let $\ell'(p_{i-1})$ be the label of $p_{i\ 1}$ if the edge out of t leading toward p_{i-1} is relabelled 0 . Then $\ell(p_i)$ - "(pi-l) $\leq p_i-p_{i-1}$ and $\ell(p_i) > \ell'(p_{i-1})$. Furthermore $M(p_{i-1},p_i)$ is equal to the number of bits to the right of and including the leftmost bit in which the binary representations of $\ell'(p_{i-1})$ and $\ell(p_i)$ differ. By Lemma 6,

$$\begin{aligned} M(P_{i-1},p_i) &\leq \nu(\ell'(p_{i-1})) - \nu(\ell(p_i)) + 2\lceil \lg(\ell(p_i) - \ell'(p_{i-1})+1)\rceil \\ &\leq \nu(\ell'(p_{i-1})) - \nu(\ell(p_i)) + 2\lceil \lg(p_i-p_{i-1}+1)\rceil \\ &\leq \nu(\ell(p_{i-1})) - \nu(\ell(p_i)) + 2\lceil \lg(p_i-p_{i-1}+1)\rceil \end{aligned}$$

since $\nu(\ell(p_{i-1})) = \nu(\ell'(p_{i-1}))+1$.

Substituting into the bound on M given above yields

$$M \leq \lceil \lg n\rceil + 1 + \sum_{1<i\leq k} (\nu(\ell(p_{i-1})) - \nu(\ell(p_i)) + 2\lceil \lg(p_i-p_{i-1}+1)\rceil) \ .$$

But much of this sum telescopes, giving

$$\begin{aligned} M &\leq \lceil \lg n\rceil + 1 + \nu(\ell(p_1)) - \nu(\ell(p_k)) + 2 \sum_{1<i\leq k} \lceil \lg(p_i-p_{i-1}+1)\rceil \\ &\leq 2\left(\lceil \lg n\rceil + \sum_{1<i\leq k} \lceil \lg(p_i-p_{i-1}+1)\rceil\right) \end{aligned}$$

(since $\nu(\ell(p_k)) \geq 1$ and $\nu(\ell(p_1)) \leq \lceil \lg n\rceil$ unless $k = 1$). This completes the proof of Lemma 7 and Theorem 1. □

The bound given in Theorem 1 is tight to within a constant factor; that is, for any n and k there is a 2-3 tree with n external nodes and some sequence of k insertions which causes within a constant factor of the given number of splits. We omit a proof of this fact.

2. Deletions from 2-3 Trees.

The operation of deletion from a 2-3 tree means the elimination of a specified external node from the tree. As with insertion, the algorithm for deletion is essentially independent of the particular scheme used for associating data with the tree's nodes.

The first step of a deletion is to remove the external node being deleted. If the parent of this node was a 3-node before the deletion, it becomes a 2-node and the operation is complete. If the parent was a 2-node, it is now a "l-node", which is not allowed in a 2-3 tree; hence some additional changes are required to restore the tree. The local transformations shown in Figure 4 are sufficient, as we shall now explain. If the l-node is the root of the tree, it can simply be deleted, and its child is the final result (Figure 4(c)). If the l-node has a 3-node as a parent or as a sibling, then a local rearrangement will eliminate the l-node and complete the deletion (Figures 4(d), 4(e)). Otherwise we fuse the l-node with its sibling 2-node (Figure 4(f)); this creates a 3-node with a l-node as parent. We then must repeat the transformations until the l-node is eliminated. Figure 5 shows an example of a complete deletion.

[Figure 4]

[Figure 5]

A deletion in a 2-3 tree requires $O(1+f)$ steps, where f is the number of node fusings required for the deletion. Since the propogation of fusings up the path during a deletion is similar to the propogation of splittings during an insertion, it is not surprising that a result analogous to Theorem 1 holds for deletions.

Theorem 2. Let T be a 2-3 tree of size n, and suppose that $k < n$ deletions are made from T. If the positions of the deleted external nodes in the original tree were $p_1 < p_2 < \ldots < p_k$, then the number of node fusings which took place during the deletions is bounded by

$$2\left(\lceil \lg n \rceil + \sum_{1<i<k} \lceil \lg(p_i - p_{i-1} + 1)\rceil\right) .$$

Proof. We shall initially mark all nodes in T which lie on a path from the root of T to one of the deleted nodes. By Lemma 7, the number of marked nodes is bounded by the given expression; hence the proof is complete if we show that during the sequence of deletions it is possible to remove one mark from the tree for each fusing.

During the sequence of deletions, we shall maintain the invariant property that every 2-node on the path from a marked external node to the root is marked. This is clearly true initially. During a deletion, the marks are handled as indicated in Figure 6. An 'x' on the left side of a transformation indicates a node which the invariant (or a previous application of transformation (b) or (f)) guarantees will be marked; an 'x' on the right side indicates a node to be marked after the transformation. These rules make only local rearrangements and create only *marked* 2-nodes, and hence they maintain the invariant. The fusing transformation (f) removes at least one mark from the tree. One of the terminating transformations (e) may create a new mark, but this is compensated by the starting transformation (b) which always destroys a mark. Hence a deletion always removes at least one mark from the tree per fusing, which proves the result. □

[Figure 6]

The bound of Theorem 2 is tight to within a constant factor; that is, for any n and $k<n$ there is a 2-3 tree with n external nodes and a sequence of k deletions which causes within a constant factor of the given number of fusings. We omit a proof.

3. Mixed Sequences of Operations.

When both insertions and deletions are present in a sequence of operations on a 2-3 tree, there are cases in which $\Omega(\log n)$ steps are required for each operation in the sequence. A simple example of this behavior is shown in Figure 7, where an insertion causes splitting to go to the root of the tree, and deletion of the inserted element causes the same number of fusings. We expect that when insertions and deletions take place in separate parts of the tree, it is impossible for them to interact in this way. The following results shows that this intuition is justified, at least for a particular access pattern arising from priority queues.

[Figure 7]

Theorem 3. Let T be a 2-3 tree of size n, and suppose that a sequence of k insertions and ℓ deletions is performed on T. If all deletions are made on the leftmost external node of T, and no insertion is made closer than $(\lg m)^{1.6}$ positions from the point of the deletions (where m is the tree size when the insertion takes place), then the total cost of the operations is

$$O\left(\log n + k + \ell + \sum_{1<i\leq k'} \log(p_i - p_{i-1})\right) ,$$

where $k' < k$ is the number of inserted nodes that have not been deleted and $p_1 < p_2 < \ldots < p_{k'}$ are the positions of these nodes in the final tree.

Proof. We shall first sketch the argument and then give it in more detail. Insertions are accounted for by marking the tree in a manner almost identical to that used in proving Theorem 1. Deletions may destroy some of these marks, so we charge a deletion for the marks it removes; the remaining marks are then counted using Lemma 7. Because we assume that insertions are bounded $(\lg m)^{1.6}$ positions away from the point of deletions, the left path is unaffected by insertions up to a height of at least $\lg \lg m$. Therefore roughly $\lg m$ deletions occur between successive deletions that reference an "unprotected" section of the left path. These $\lg m$ deletions cost $O(\log m)$ altogether, as does a single deletion that goes above the protected area, so ℓ deletions cost roughly $O(\ell)$ steps to execute. Adding this to the cost of the insertions gives the bound.

We shall present the full argument as a sequence of lemmas. First we need some terminology. The left path is the path from the root to the leftmost external node. Note that deletions will involve only left-path nodes and the children of such nodes. We say that an insertion changes the left path if it splits a 3-node or expands a 2-node on the left path.

Lemma 8. Under the assumptions of Theorem 3, the cost of the sequence of insertions is

$$O\left(\log n + k + \sum_{1 \le i \le k'} \log(p_i - p_{i-1}\right) + O(\text{cost of deletions}) .$$

Proof. On each insertion, we mark the nodes of T according to rules (1) and (2) in the proof of Theorem 1, while observing the following additional rule:

(3) When a marked 2-node on the left path expands, an *unmarked* 3-node is created.

As in the proof of Theorem 1, the cost of all insertions is bounded by the number of marks created using rules (1) and (2). Rule(3), which destroys a mark, can be applied at most once per insertion, and hence the number of marks removed by this rule is O(k) .

This marking scheme preserves the property that on the left path, no 3-node ever becomes marked. It does not preserve any stronger properties on the left path; for example, a marked 2-node with no marked offspring may occur. But it is easy to prove by induction on the number of insertion steps that the stronger properties used in the proof of Theorem 1 (a marked 2-node has at least one marked offspring, a marked 3-node has at least two marked offspring) do hold on the rest of the tree. The intuitive reason why the corruption on the left path cannot spread is that it could do so only through the splitting of 3-nodes on the path; since these nodes aren't marked, they never create "unsupported" 2-nodes off the left path.

The motivation for these marking rules is that deletions will necessarily corrupt the left path. During deletions, we treat marks according to the following rule:

(4) Any node involved in a deletion transformation (i.e., any node shown explicitly in Figure 4) is unmarked during the transformation.

This rule removes a bounded number of marks per step, and hence over ℓ deletions the number of marks removed is O(cost of deletions) . Since this rule never creates a marked node, it preserves the property of no marked 3-nodes on the left path. It also preserves the stronger in-variants on the rest of the tree, since it will only unmark a node whose parent is on the left path.

It follows that after the sequence of insertions and deletions, all marked nodes lie on paths from the inserted external nodes to the root, except possibly some marked 2-nodes on the left path. The number of nodes on the left path is $O(\log(n+k-\ell))$, and by Lemma 7 the number of marked nodes in the rest of the tree is

$$O\left(\log(n+k-\ell) + \sum_{1<i\leq k'} \log(p_i - p_{i-1} + 1)\right) .$$

Adding these bounds to our previous estimates for the number of marks removed by rules (3) and (4), and noting that $\lg(x+y) \leq \lg x + y$ for $x,y \geq 1$, gives the result. □

Lemma 9. Suppose that a sequence of j deletions is made on the leftmost external node of a 2-3 tree, such that the deletions do not reference any left-path nodes changed by an insertion made during the sequence. Then the cost of the sequence is $O(j)$ + O(height of the tree before the deletions) .

Proof. The cost of a deletion is $O(1+f)$ where f is the number of fusings required. Each fusing destroys a 2-node on the left path, so the total cost of the j deletions is $O(j)$ + O(number of left-path 2-nodes destroyed) . But each deletion creates at most one left-path 2-node, and insertions do not create any 2-nodes that are referenced by the deletions, so the cost is in fact $O(j)$ + O(number of originally present left-path 2-nodes destroyed) . This is bounded by the quantity given above. □

Lemma 10. Under the assumptions of Theorem 3, if the tree T has size m then an insertion cannot change any left-path node of height less than $\lg \lg m$.

Proof. A 2-3 tree of height h contains at most 3^h external nodes. Hence a sub-tree of height $\lg \lg m$ contains $\leq 3^{\lg \lg m} = (\lg m)^{\lg 3}$ external nodes, which is strictly less than the $(\lg m)^{1.6}$ positions that are protected from insertions under the conditions of Theorem 3. □

Lemma 11. Suppose that the bottommost k nodes on the left path are all 3-nodes, and deletions are performed on the leftmost external node. If insertions do not change any nodes of height $\leq k$ on the left path, then at least 2^k deletions are required to make a deletion reference above height k on the left path.

Proof. Let us view the left path as a binary integer, where a 2-node is represented by a zero and a 3-node by a one, and the root corresponds to the most significant bit. Then deletion of the leftmost external node corresponds roughly to subtracting one from this binary number. Consideration of the deletion algorithm shows that the precise effect is as follows: if the left path is $xx\ldots x1$ then a deletion causes it to become $xx\ldots x0$ (subtraction of 1), and if the path is $xx\ldots x1\overbrace{00\ldots0}^{i}$ then it becomes either $xx\ldots x0\overbrace{11\ldots1}^{i}$ (subtraction of 1) or $xx\ldots x10\overbrace{1\ldots1}^{i-1}$ (addition of $2^{i-1}-1$). Only this final possibility (corresponding to using the transformation in Figure 4(e)) differs from subtraction by one. Note that under these rules everything to the left of the rightmost one bit is unreferenced by a deletion.

Before a deletion reference above height k can take place, the number represented by the rightmost k bits must be transformed from 2^k-1 into 0 by operations which either subtract one or add a positive number. Thus 2^k-1 subtractions are required, corresponding to 2^k-1 deletions. □

Lemma 12. Under the assumptions of Theorem 3, the cost of the sequence of deletions is $O(\log n+k+\ell)$.

Proof. For accounting purposes we shall divide the sequence of ℓ deletions into disjoint epochs, with the first epoch starting immediately before the first deletion. Intuitively, epochs represent intervals during which insertions do not interact directly with deletions. We define the current epoch to end immediately before a deletion that references any node on the left path that has been changed by an insertion since the first deletion of the epoch. This deletion is then the first in the new epoch; the final epoch ends with the last deletion of the sequence. According to this definition, each epoch contains at least one deletion.

Let ℓ_i denote the number of deletions during the i-th epoch, k_i the number of insertions during this epoch, and m_i the tree size at the start of the epoch. The first deletion of epoch i costs $O(\log m_i)$. By Lemma 9, the final ℓ_i-1 deletions cost $O(\ell_i + \log m_i)$ since they operate on a section of the left path that is unaffected by insertions. Hence the total cost of the deletions in epoch i is $O(\ell_i + \log m_i)$. We shall prove that except for the first and last epochs, this cost is $O(\ell_i + k_{i-1})$, so that the total cost of these epochs is $O(\ell + k)$. Since $m_i < n+k$, each of the first and last epochs costs $O(\ell_i + \log(n+k))$. Combining gives the bound in the lemma.

Consider an epoch i that is not the first or the last.' The first deletion of an epoch transforms all nodes below height h on the left path into 3-nodes, where h is the height of some left-path node that has been changed by an insertion since the start of epoch i-l . Let $h_i = \lfloor \lg \lg m_i \rfloor - 1$. By Lemma 10, the allowable insertions at this point cannot change the left path below height h_i . This remains true even if the tree size grows to m_i^2 or shrinks to $\sqrt{m_i}$, since this changes the value of lg lg m by only l . Hence if $h \geq h_i$ (i.e., all left-path nodes below height h are 3-nodes), Lemma 11 shows that $2^{h_i} = \Omega(\log m_i)$ deletions are necessary to reference a node above height h_i . Thus $\ell_i = \Omega(\log m_i)$, which means that $O(\ell_i + \log m_i)$, the cost of the epoch, is $O(\ell_i)$. If on the other hand $h < h_i$, this implies that at some point during epoch i-l the tree size m was much smaller than m_i , in particular $m < \sqrt{m_1}$. But this shows that $k_{i\ 1} = \Omega(m_i)$, so $O(\ell_i + \log m_i) = O(\ell_i + k_{i\ 1})$. In summary, we have shown that the cost of epoch i is $O(\ell_i + k_{i-1})$ regardless of the value of h , □

Combining the results of Lemmas 8 and 12 proves Theorem 3. □

Theorem 3 is certainly not the ultimate result of its kind. For example, it is possible to allow some number of insertions to fall close to the point of deletion and still preserve the time bound. (N0te that Lemma 8 does not depend on any assumption about the distribution of insertions, so only the proof of the bound on deletions needs to be

4. Level-Linked Trees.

The results in Sections l-3 show that in several interesting cases the $O(\log n)$ bound on individual insertions and deletions in a 2-3 tree is overly pessimistic. In order to use this information we must examine the cost of searching for the positions where the insertions and deletions are to take place. If the pattern of accesses is random, there is little hope of reducing the average search time below $O(\log n)$; it is impossible for any algorithm based solely on comparisons to beat $\Omega(\log n)$. But in many circumstances there is a known regularity in the reference pattern that we can exploit.

One possible method of using the correlation between accesses is to keep a finger -- a pointer to an item in the list. For a suitable list representation it should be much more efficient to search for an item near the finger than one far away. Since the locale of interest may change with time, the list representation should make it easy to move a finger while still enjoying fast access near it. There may be more than one busy area in the list, so it should be possible to efficiently maintain multiple fingers.

The basic 2-3 tree structure for sorted lists shown in Figure 3 is not suitable for finger searching, since there are items adjacent in the list whose only connection through the tree structure is a path of length $\Theta(\log n)$. Figure 8 shows an extension of this structure that does support efficient access in the neighborhood of a finger. The arrangement of list elements and-keys is unchanged, but the edges between internal nodes are made traversible upwards as well as downwards, and horizontal links are added between external nodes that are neighbors (adjacent on the same level). We shall call this list representation a level-linked 2-3 tree.

[Figure 8]

A finger into this structure consists of a pointer to a terminal node of the tree. It would seem more natural for the finger to point directly to an external node, but no upward links leading away from the external nodes are provided in a level-linked tree; the reasons for this decision will become evident when implementation considerations are discussed in Section 5. Note that the presence of a finger requires no change to the structure.

Roughly speaking, the search for a key k using a finger f proceeds by climbing the path from f toward the root of the tree. We stop ascending when we discover a node (or a pair of neighboring nodes) which subtends a range of the key space in which k lies. We then search downward for k using the standard search technique.

A more precise description of the entire search procedure is given below in an Algol-like notation. If t is an internal node, then we define Large&key(t) and Smallestkey(t) to be the largest and smallest keys contained in t , and let Leftmostlink(t) and Rightmostlink(t) denote respectively the leftmost and rightmost downward edges leaving t . The fields ℓNbr(t) and rNbr(t) give the left and right neighbors of t, and are Nil if no such nodes exist; Parent(t) is the parent of t, and is Nil if t is the root.

```
procedure FingerSearch(f,k)
    comment Here f is a finger (a pointer to a terminal node) and k
    is a key. If there is an external node with key k in the structure
    fingered by f , then FingerSearch returns a pointer to the parent
    of the rightmost such node. Otherwise the procedure returns a pointer
    to a terminal node beneath which an external node with key k may be
    inserted. Hence in either case the result may be used as a (new) finger.
    if     k ≥ Large&Key(f) then return SearchUpRight(f,k)
    elseif k < SmallestKey then return SearchUpLeft(f,k)
    else return f
    endif
end FingerSearch


procedure SearchUpRight(p,k)
    loop
        comment At this point either f = p , or f lies to the left
        of p's right subtree. The key k is larger than the leftmost
        (smallest) descendant of p .
        if   k < LargestKey or rNbr(p) = Nil then return SearchDown(p,k)
        else q ← rNbr(p)
            if     k < SmallestKey then return SearchDownBetween(p,q,k)
            elseif k < LargestKey then return SearchDown(q,k)
            else   p ← Parent(q)
            endif
        endif
    repeat
end SearchUpRight


procedure SearchUpLeft(p,k)
    (similar to the above]
procedure SearchDownBetween(p,q,k)
     loop until p and q are terminal:
          comment Here p is the left neighbor of q , and k is contained
          in the range of key values spanned by the children of p and q.
          if     k < LargestKey then return SearchDown(p,k)
          elseif k ≥ SmallestKey(q) then return SearchDown(q,k
          else   p ← RightmostLink(p)
                 q ← LeftmostLink
          endif
     repeat
     if k < Key[RightmostLink(p)] then return p
                                   else return q
     endif
end SearchDownBetween

procedure SearchDown(p,k)
     {the standard 2-3 tree search procedure)
```

This algorithm allows very fast searching in the vicinity of fingers, In spite of this, we shall show that if a sequence of intermixed searches, insertions, and deletions is performed on a level-linked 2-3 tree, the cost of the insertions and deletions is dominated by the search cost, at least in the cases studied in Sections 1-3. In order to carry out this analysis we must first examine the cost of individual operations on a level-linked tree.

Lemma 13. If the key k is d keys away from a finger f , then FingerSearch(f,k) runs in $\Theta(\log d)$ steps.

Proof. The running time of FingerSearch is bounded by a constant times the height of the highest node examined, since the search procedure examines at most four of the nodes at each level. It is not hard to see from the invariants in SearchUpRight (and SearchUpLeft) that in order for the search to ascend ℓ levels in the tree, there must exist a sub-tree of size ℓ-2 all of whose keys lie between k and the keys of the finger node. The lemma follows. □

Lemma 14. A new external node can be inserted in a given position in a level-linked 2-3 tree in $\Theta(1+s)$ steps, where s is the number of node splittings caused by the insertion.

Proof. We sketch an insertion method which can be implemented to run in the claimed time bound. Suppose we wish to insert a new external node with key k . During the insertion process we must update the links and the keys in the internal nodes. Let node p be the prospective parent of node e . If e would not be the rightmost child of p , we make e a child of p , insert the key k in node p and proceed with node-splitting as necessary. If e would be the rightmost child of p but e has a right neighbor, we make e a child of the right neighbor. Otherwise k is larger than all keys in the tree. In this case we make e a child of p and place the previously largest key in node p . (The key k is not used in an internal node until it is no longer the largest.)

When a 4-node q splits during insertion, it is easy to update the links in constant time. To maintain the internal key organization, we place the left and right keys of q in the new 2-nodes produced by the split, and the middle key in the parent of q. □

Lemma 15. An external node can be deleted from a level-linked 2-3 tree in $\Theta(1+f)$ steps, where f is the number of node fusings.

Proof. Similar to the proof of Lemma 14. □

Lemma 16. Creation or removal of a finger in a level-linked 2-3 tree requires Θ 1) time.

Proof. Obvious. □

Now we apply the results of Sections 1 - 3 to show that even though the search time in level-linked 2-3 trees can be greatly reduced by maintaining fingers, it still dominates the time for insertions and deletions in several interesting cases.

Theorem 4. Let L be a sorted list of size n represented as a level-linked 2-3 tree with one finger established. Then in any sequence of searches, finger creations, and k insertions, the total cost of the k insertions is $O(\log n +$ total cost of searches$)$.

Proof. Let S be any sequence of searches, finger creations, and insertions which includes exactly k insertions. Let the external nodes of L after the insertions have been performed be named $0,1,\ldots,n+k-1$ from left to right. Assign to each external node p a label $\ell(p)$, whose value is the number of external nodes lying strictly to the left of p which were present before the insertions took place; these labels lie in the range $0,1,\ldots,n$.

Consider the searches in S which lead either to the creation of a new finger (or the movement of an old one) or to the insertion of a new item. Call an item of L accessed if it is either the source or the destination of such a search. (We regard an inserted item as the destination of the search which discovers where to insert it.) Let $p_1 < p_2 < \ldots < p_\ell$ be the accessed items.

We shall consider graphs whose vertex set is a subset of $\{p_i \mid 1 \le i \le \ell\}$, We denote an edge joining $p_i < p_j$ in such a graph by $p_i\text{-}p_j$ and we define the cost of this edge to be $\max(\lceil \lg(\ell(p_j) - \ell(p_i)+1)\rceil , 1)$. For each item p_i (except the initially fingered item) let q_i be the fingered item from which the search to p_i was made. Each q_i is also in $\{p_i \mid 1 \le i \le \ell\}$ since each finger except the first must be established by a search. Consider the graph G with vertex set $\{p_i \mid 1 \le i \le \ell\}$ and edge set $\{(q_i,p_i) \mid 1 \le i \le \ell$ and p_i is not the originally fingered item] .

Some constant times the sum of edge costs in G is a lower bound on the total search cost, since $|\ell(p_i) - \ell(q_i)|+1$ can only underestimate the actual distance between q_i and p_i when p_i is accessed. We shall describe a way to modify t , while never increasing its cost, until it becomes

$$r_1 - r_2 - \cdots - r_k$$

where $r_1 < r_2 < \ldots < r_k$ are the k inserted items. Since the cost of this graph is $\sum_{1<i\le k} \lceil \lg(r_i-r_{i-1}+1) \rceil$, the theorem then follows from Theorem 1.

The initial graph G is connected, since every accessed item must be reached from the initially fingered item. We first delete all but $\ell-1$ edges from G so as to leave a spanning tree; this only decreases the cost of G .

Next, we repeat the following step until it is no longer applicable: let p_i-p_j be an edge of G such that there is an accessed item p_k satisfying $p_i < p_k < p_j$. Removing edge p_i-p_j now divides G into exactly two connected components. If p_k is in the same connected component as p_i , we replace p_i-p_j by p_k-p_j ; otherwise, we replace p_i-p_j by p_i-p_k . The new graph is still a tree spanning $\{p_i \mid 1 \le i \le \ell\}$ and the cost has not increased.

Finally, we eliminate each item p_j which is not an inserted item by transforming $p_i-p_j-p_k$ to p_i-p_k , and by removing edges p_j-p_k where there is no other edge incident to p_j . This does not increase cost, and it results in the tree of inserted items

$$r_1 - r_2 - \cdots - r_k$$

as desired. □

Theorem 5. Let L be a sorted list of size n represented as a level-linked 2-3 tree with one finger established. Then in any sequence of searches, finger creations, and k deletions, the cost of the deletions is $O(\log n + \text{total cost of searches})$.

Proof. Similar to the proof of Theorem 4, using Theorem 2. □

Theorem 6. Let L be a sorted list of size n represented as a level-linked 2-3 tree with one finger established. For any sequence of searches, finger creations, k insertions, and ℓ deletions, the total cost of the insertions and deletions is $O(\log n + \text{total cost of searches})$ if the insertions and deletions satisfy the assumptions of Theorem 3.

Proof. Similar to the proof of Theorem 4, using Theorem 3. □

5. Implementation and Applications.

In Section 4 we described a level-linked 2-3 tree in terms of internal and external nodes. The external nodes contain the items stored in the list, while the internal nodes are a form of "glue" which binds the items together. The problem remains of how to represent these objects in storage.

External nodes present no difficulty; they can be represented by the items themselves, since we only maintain links going to these nodes (and none coming from them). Internal nodes may be represented in an obvious way by a suitable record structure containing space for up to two keys and three downward links, a tag to distinguish between 2- and T-nodes, and other fields. One drawback of this approach is that because the number of internal nodes is unpredictable, the insertion and deletion routines must allocate and deallocate nodes. In random 2-3 trees [9] the ratio of 2-nodes to 3-nodes is about 2 to 1, so we waste storage by leaving room for two keys in each node. Having different record structures for the two node types might save storage at the expense of making storage management much more complicated.

Figure 9 shows a representation which avoids these problems. A 3-node is represented in a linked fashion, analogous to the binary tree structure for 2-3 trees [6, p.469]. The internal node component containing a key k is combined as a single record with the representation of the item (external node) with key k . Hence storage is allocated and deallocated only when items are created and destroyed, and storage is saved because the keys in the internal nodes are not represented explicitly. (The idea of combining the representations of internal and external nodes is also found in the "loser-oriented" tree for replacement selection [6, p.256].)

[Figure 9]

An example which illustrates this representation is shown in Figure 10. Each external node except the largest participates in representing an internal node, so it is convenient to assume the presence of an external node with key $+\infty$ in the list. This node need not be represented explicitly, but can be given by a null pointer as in the figure. Null rLinks are also used to distinguish a 3-node from a pair of neighboring 2-nodes. There are several ways to identify the ℓLinks and rLinks that point to external nodes: one is to keep track of height in the tree during FingerSearch , since all external nodes lie on the same level. Another method is to note that a node p is terminal if and only if $\ell\mathrm{Link}(p) = p$.

[Figure 10]

We now consider the -potential applications of this list representation. One application is in sorting files which have a bounded number of inversions. The result proved by Guibas et. al. [4], that insertion sort using a list representation with one finger gives asymptotically optimal results, applies equally to our structure since insertion sort does not require deletions.

A second application is in merging: given sorted lists of lengths m and n , with $m \leq n$, we wish to merge them into a single sorted list. Any comparison-based algorithm for this problem must use at least $\left\lceil \lg \binom{m+n}{m} \right\rceil = \Theta\left(m \log \frac{n}{m}\right)$ comparisons; we would like an algorithm whose running time has this magnitude. We solve this problem using our list structure 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 $O\left(\log n + \sum_{1<i\le m} \log d_i\right)$ steps for the insertions, where d_i is the distance from the finger to the i-th 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 $\sum_{1<i<m} \log d_i$ subject to this constraint we choose the d_i to be equal, and this gives the desired bound of $O(m \log(n/m))$ steps for the algorithm, (The usual height-balanced or 2-3 trees can be used to perform fast merging [3], but the algorithm is not obvious and the time bound requires an involved proof.)

When an ordered set is represented as a sorted list, the merging algorithm just described can be modified to perform the set union operation: we simply check for, and discard, duplicates when inserting items from the smaller list into the larger list. This obviously gives an $O(m \log(n/m))$ algorithm for set intersection as well, if we retain the duplicates rather than discarding them. Trabb Pardo[8] has developed algorithms based on trie structures which also solve the set intersection -problem (and the union or merging problems) in $O(m \log(n/m))$ time, but only on the average.

Another application for the level-linked 2-3 tree is in implementing a priority queue used as a simulation event list. In this situation the items being stored in the list are -procedures to be executed at a known instant of simulated "time"; to perform one simulation step we delete the item from the list having the smallest time and then execute it, which may cause new events to be inserted into the list. Theorem 3 shows that unless these new events are often **very** soon to be deleted, a 2-3 tree can process a long sequence of such simulation steps with only a constant cost per operation (independent of the queue size). Furthermore, searches using fingers will usually be very efficient since the simulation program produces events according to known patterns. (Some simulation languages already give programmers access to crude "fingers", by allowing the search to begin from a specified end of the event list.)

An obvious question relating to our structure is whether it can be generalized so that arbitrary deletions will not change the worst-case time bound for a sequence of accesses. This seems to be difficult, since the requirement for a movable finger conflicts with the need to maintain path regularity constraints [4]. Thus a compromise between the unconstrained structure given here and the highly constrained structure of Guibas et. al. [4] should be explored.

Even if such a more general structure could be found, it might be less practical than ours. To put. the problem of deletions in perspective, it would be interesting to derive bounds on the average case performance of our structure under insertions and deletions, using a suitable model of random insertions and deletions. It may be possible, even without detailed knowledge of random 2-3 trees, to show that operations which require $\Theta(\log n)$ time are very unlikely.

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] Rudolf Bayer and Edward M. McCreight. "Organization and maintenance of large ordered indexes," Acta Informatica 1 (1972), 173-189.

[3] Mark R. Brown and Robert E. Tarjan. "A fast merging algorithm," Stanford Computer Science Department Report STAN-C S-77-625, August 1977; Journal ACM (to appear).

[4] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. "A new representation for linear lists," Proc. Ninth Annual ACM Symp. on Theory of Computing (1977), 49-60.

[5] Donald E. Knuth. The Art of Computer Programming, Volume 1, Fundamental Algorithms. Addison-Wesley, Reading, Mass., (1975 - Second Edition).

[6] Donald E. Knuth. The Art of Computer Programming, Volume 3, Sorting and Searching. Addison-Wesley, Reading, Mass., (1973).

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

[8] Luis Trabb Pardo, "Set Representation and Set Intersection," Stanford Computer Science Department, Report STAN-CS-78-681, December 1978.

[9] Andrew C.-C. Yao, "On random 2-3 trees," Acta Informatica 9 (1978), 159-170.

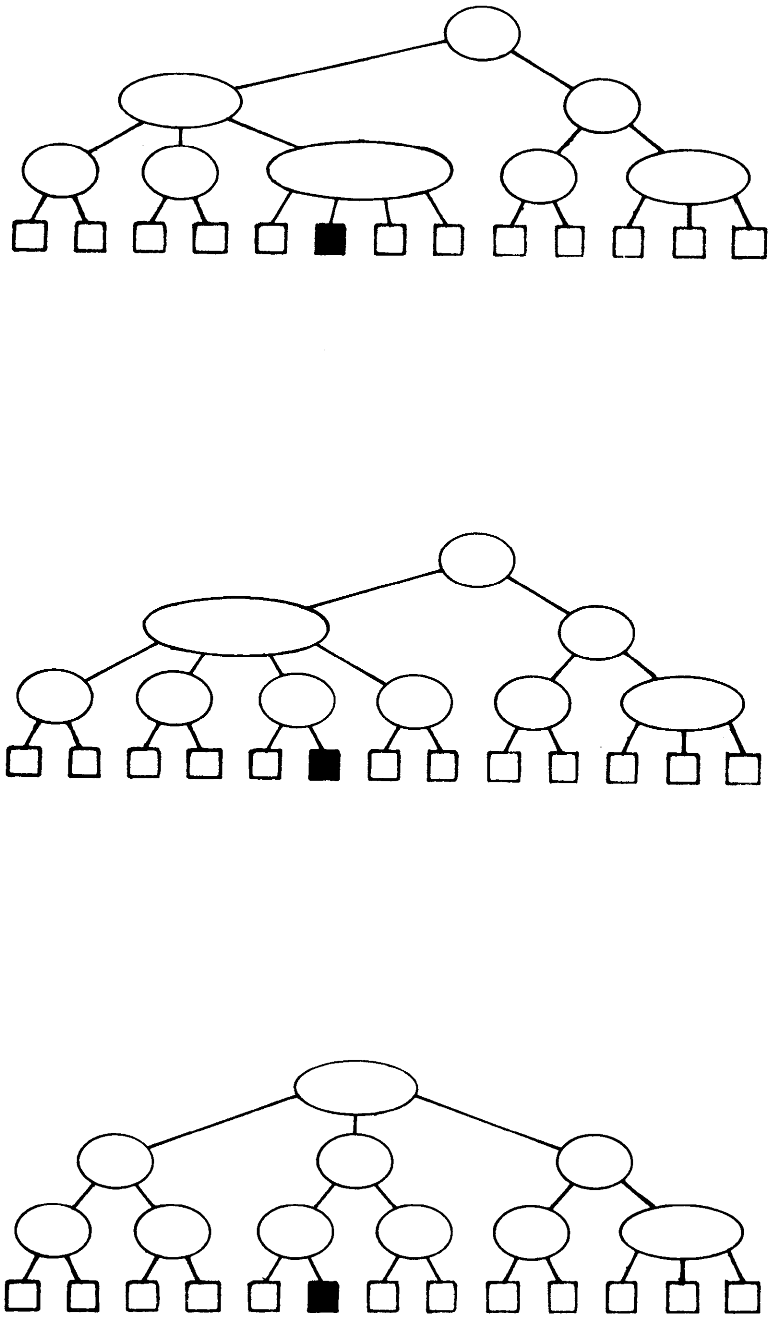

Figure 2. A 2-3 tree insertion.

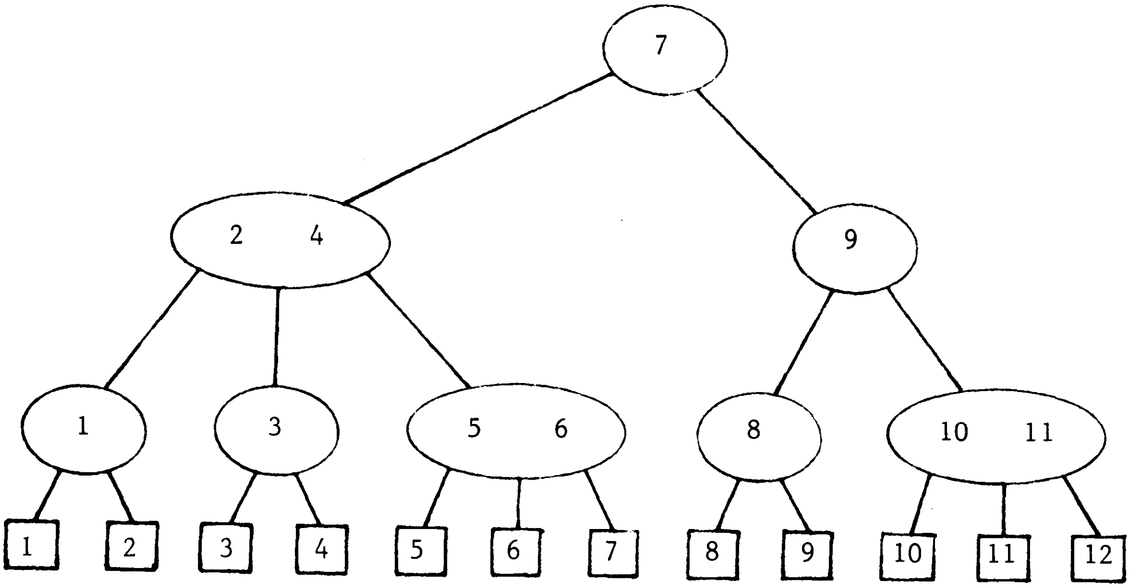

Figure 3. A 2-3 tree structure for sorted lists.

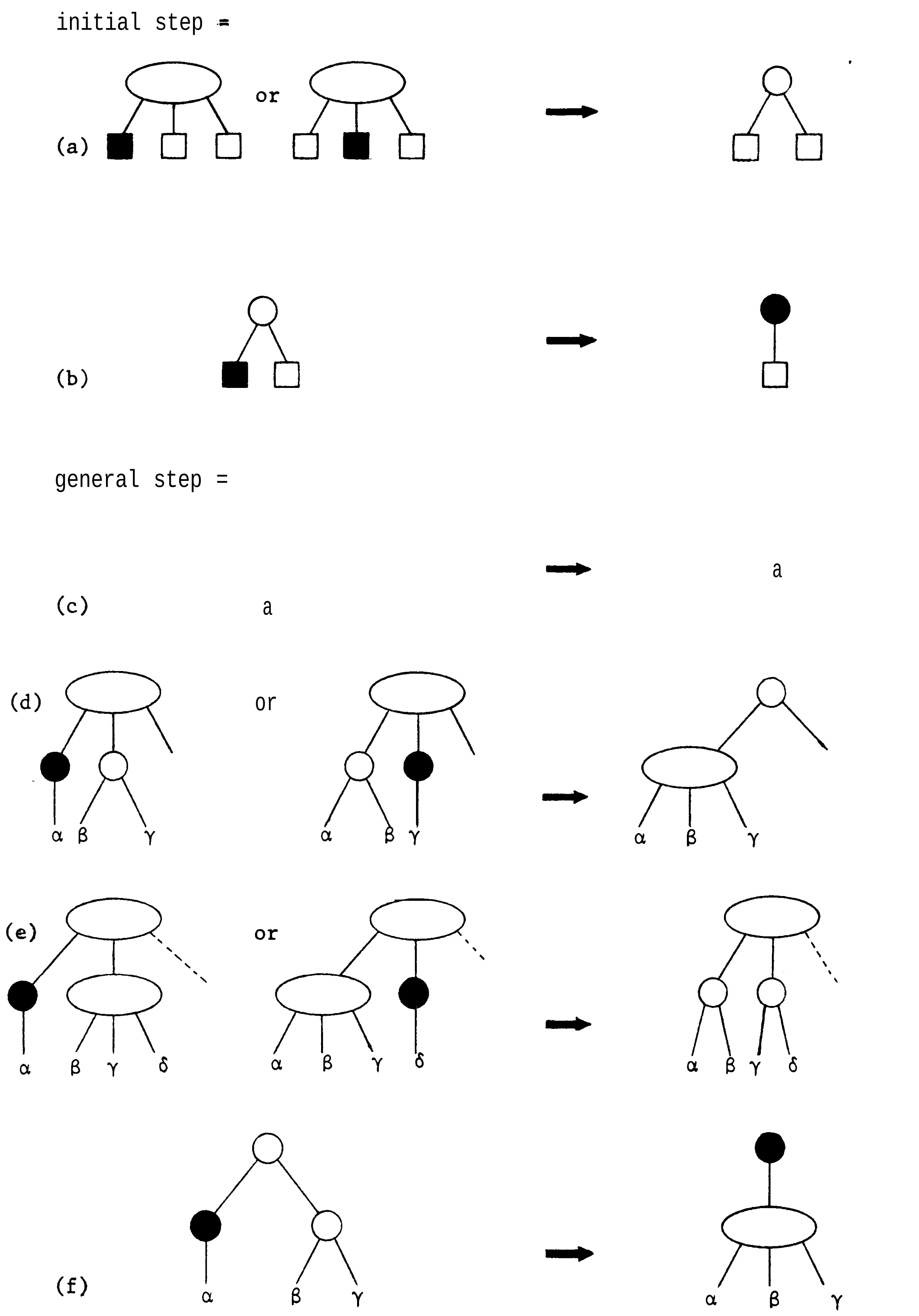

Figure 4: Transformations for 2-3 tree deletion. (Mirror-images of all transformations are possible.)

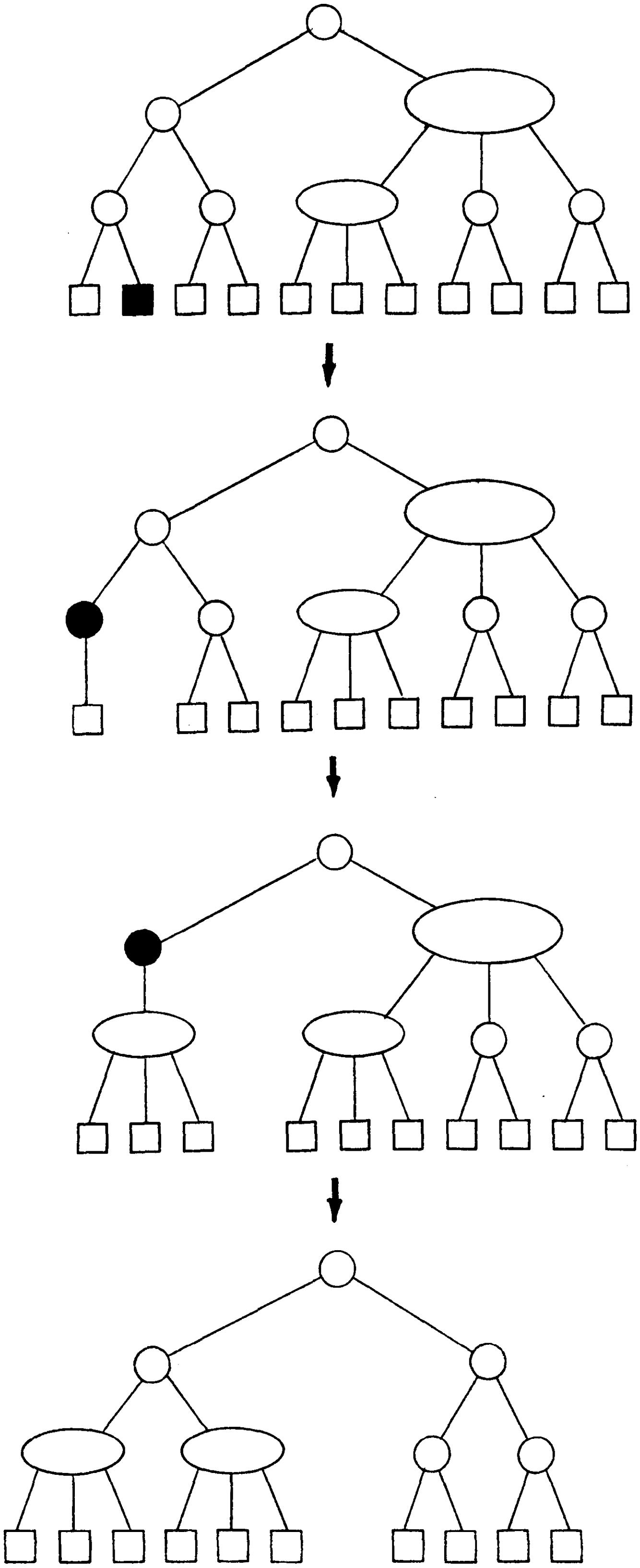

Figure 5: A 2-3 tree deletion.

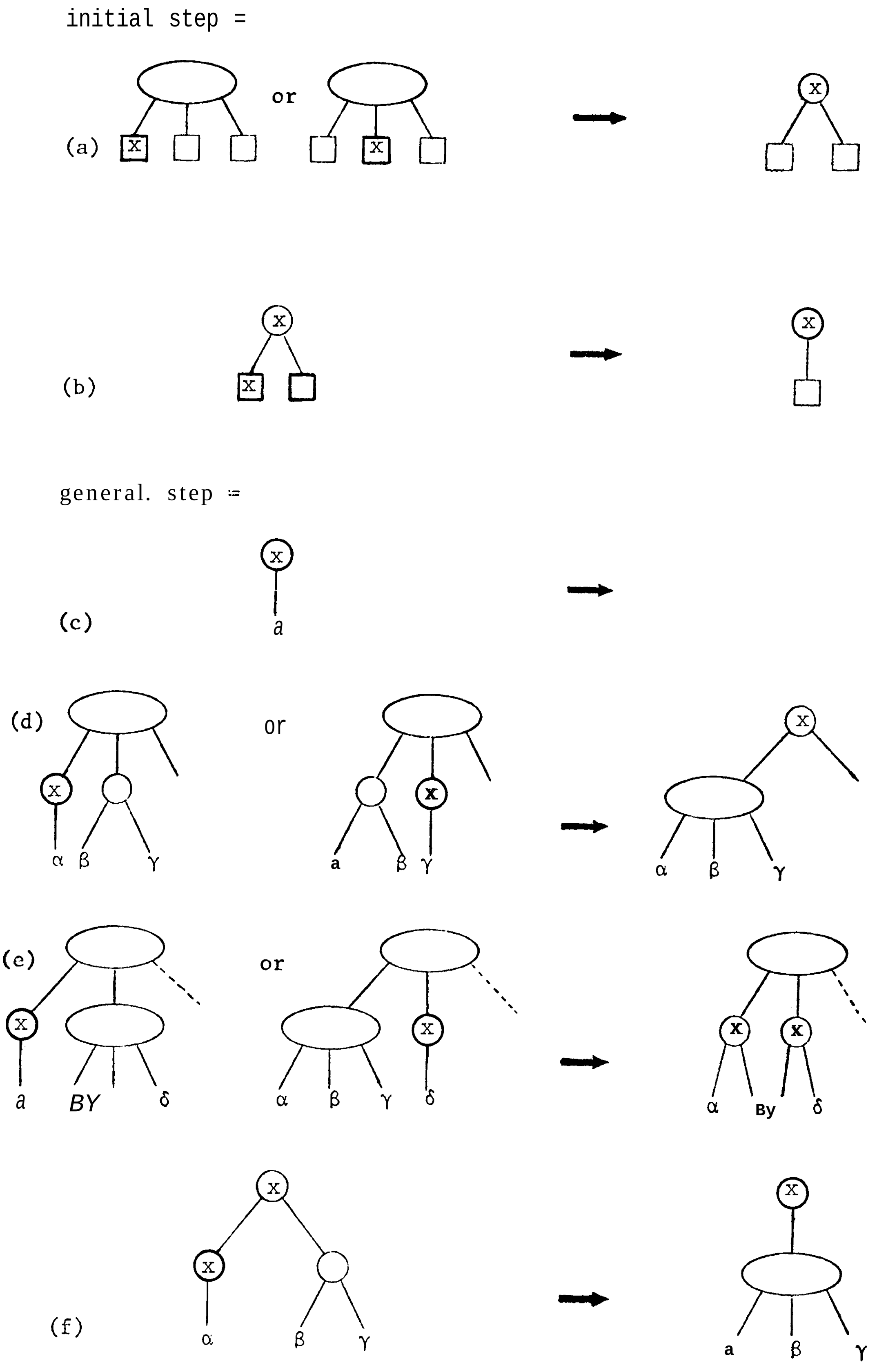

Figure 6. Deletion transformations for -proof of Theorem 2.

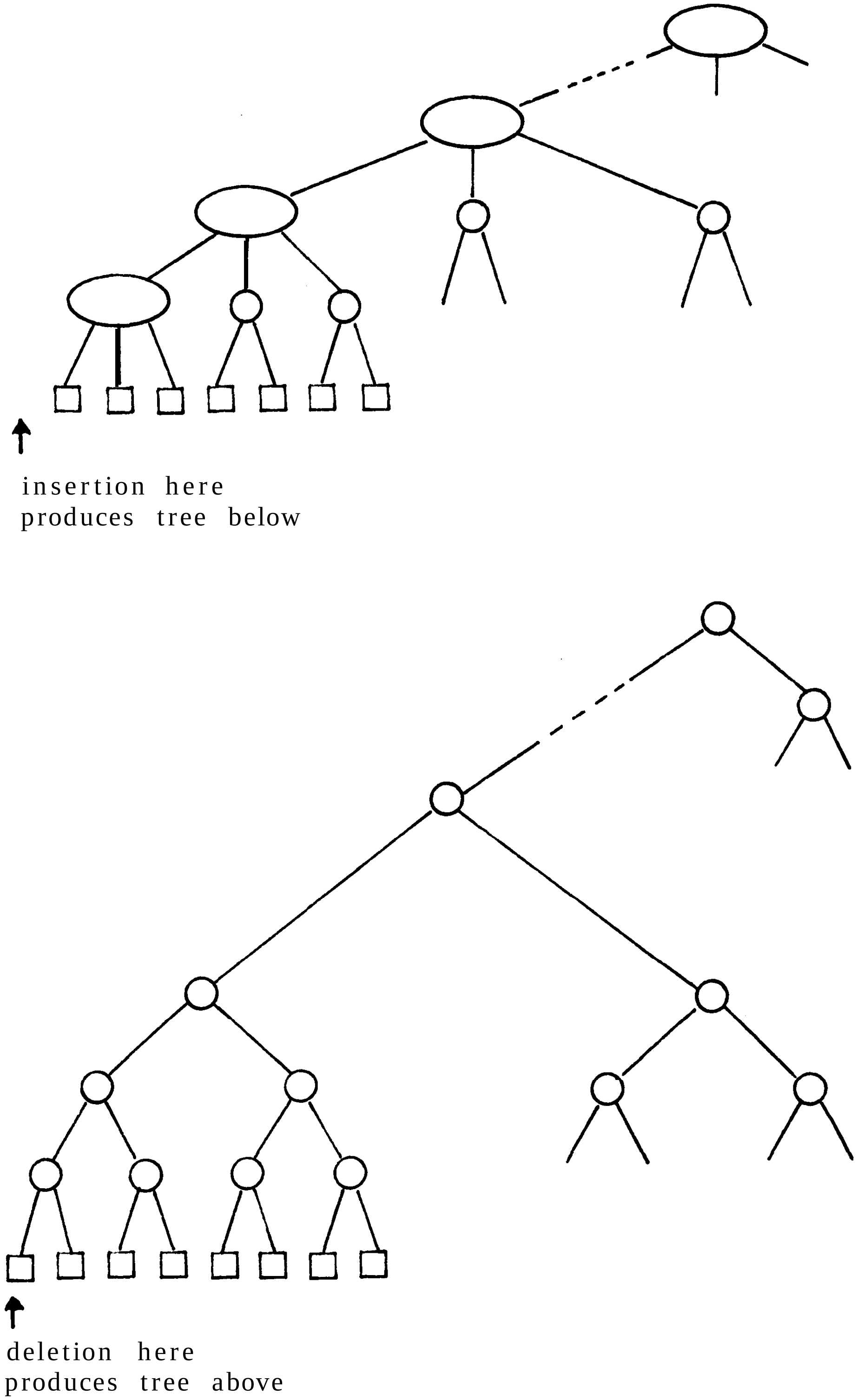

Figure 7: An expensive insert/delete pair.

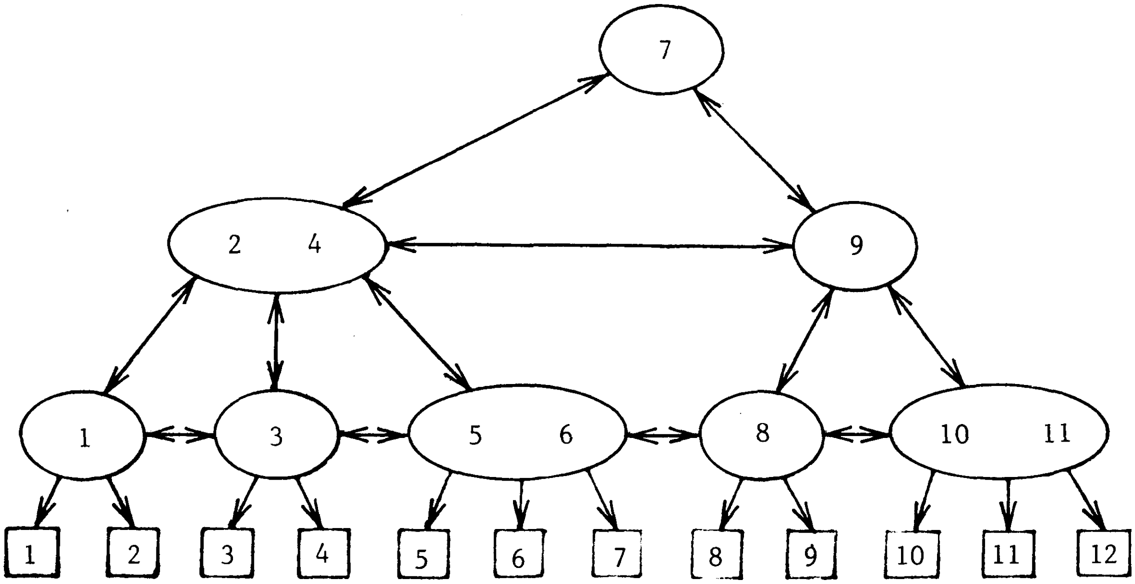

Figure 8. A level-linked 2-3 tree.

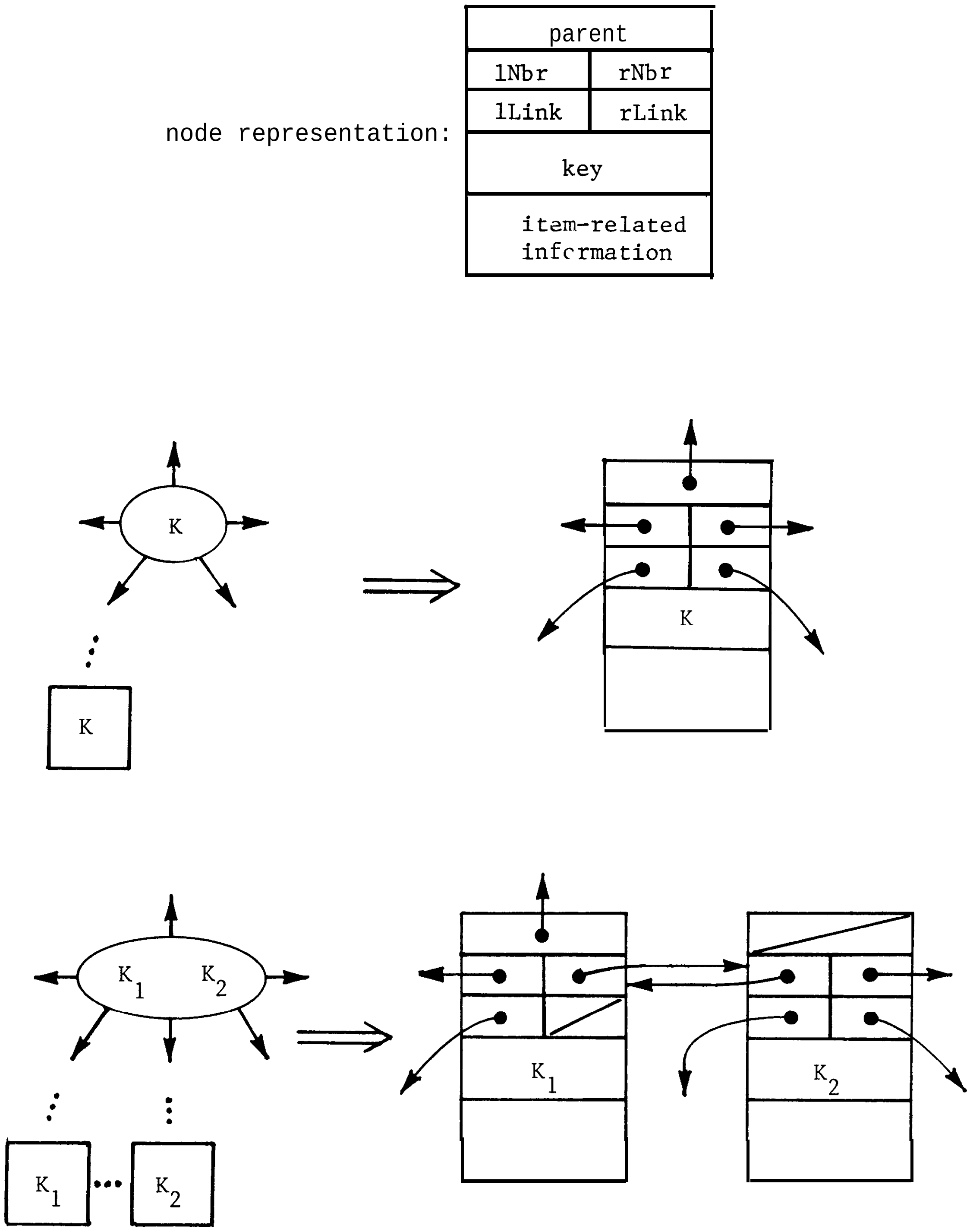

Figure 9. A storage representation for internal and external nodes.

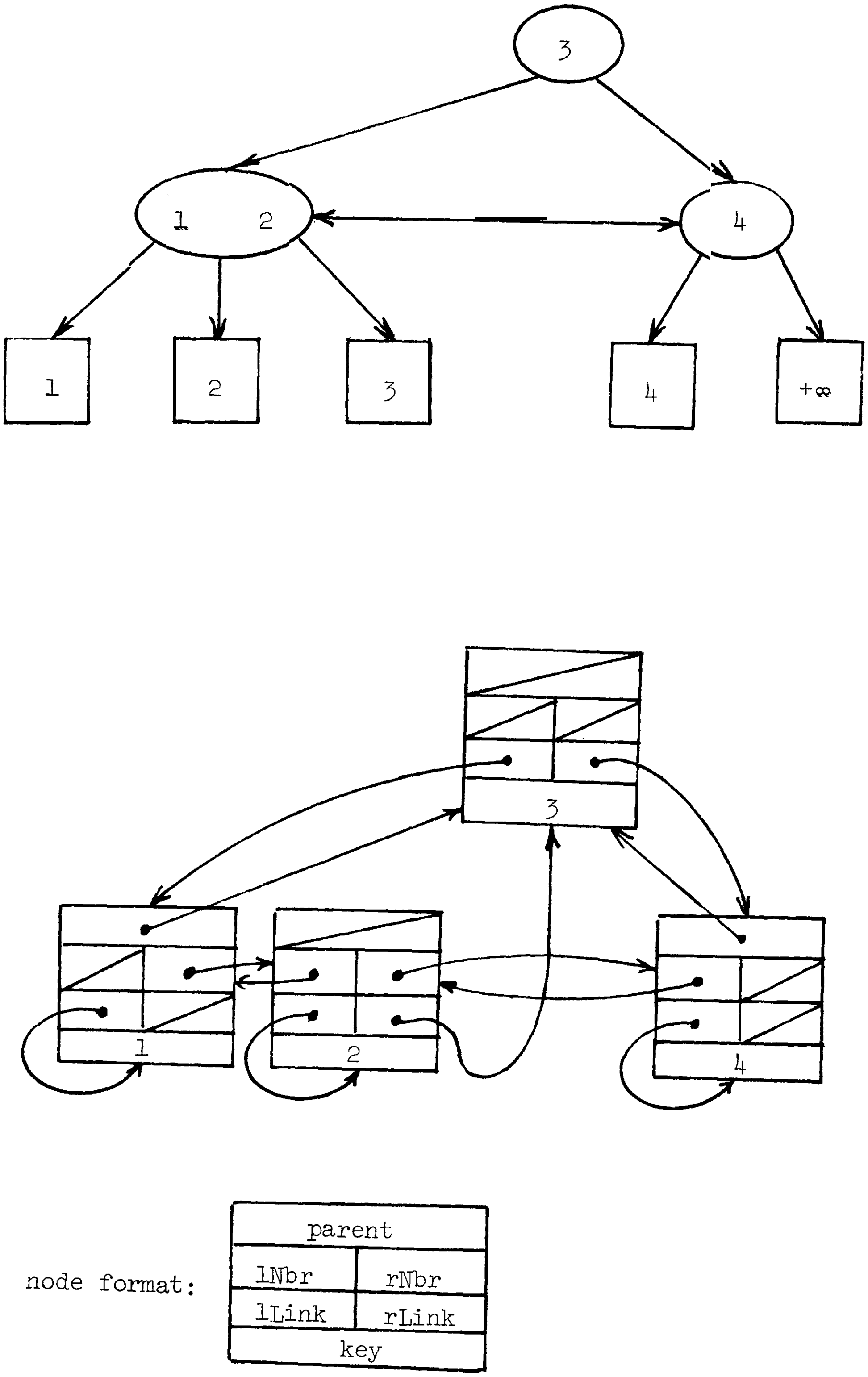

Figure 10. A structure and its storage representation