A REPRESENTATION FOR LINEAR LISTS WITH MOVABLE FINGERS

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

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

Abstract.

This paper describes a data structure which is useful for representing linear lists when the pattern of accesses to a list exhibits a (perhaps time-varying) locality of reference. The structure has many of the properties of the representation proposed by Guibas, McCreight, Plass, and Roberts [4], but is substantially simpler and may be practical for lists of moderate size. The analysis of our structure includes a general treatment of the worst-case node splitting caused by consecutive insertions into a 2-3 tree.

0. Introduction.

In [4], a representation for linear lists was introduced which allows very efficient access in the neighborhood of selected positions of a list. More precisely, if a finger is defined to be a fixed value in the space of keys stored in the list, then the structure allows one to access, insert, or delete a list item located within d positions of the finger in $O(\log d)$ time. (Throughout the paper, we understand $O(\log d)$ to mean $O(\log(\max(d,2)))$ when appropriate.) The structure also supports multiple fingers; when f fingers are present the time bound grows to $O(f+\log d)$ per access.

This data structure has certain limitations, however. First, the structure and the algorithms which maintain it are very complicated, making implementation difficult. The representation is based on B-trees [2; 6, p. 473] having a minimum order (degree of branching) of 25 ; hence it seems unsuitable for small or moderate-sized lists which can be kept in core. The large overhead associated with the structure makes it applicable only to extremely large lists kept in external storage.

Another limitation is that the structure does not support the operation of efficiently moving a finger. To move a finger to a new position in the key space requires that the finger be abandoned and a new finger created from scratch at a cost of $\Theta(\log n)$,**/ even if the finger moves only a few positions in the list. It would be useful to be able to move a finger d positions in $O(\log d)$ steps.

In this paper we give a list structure which partially solves the problems described above. With this structure the cost of moving a finger d positions is $O(\log d)$, the same as the cost of searching d positions from a finger. The cost of inserting a new item at a distance d from a finger can be $\Theta(\log n)$, where n is the list size; but when consecutive insertions are made at distances $d_1,d_2,\ldots,d_k$ from a finger, the worst-case cost is $\Theta\left(\log n + \sum_{1\le i\le k} \log d_i\right)$. Our structure is based on 2-3 trees [1] and is practical for lists of moderate size, while an obvious generalization of the structure to B-trees makes it suitable for very large lists. The major limitation of our structure is that when deletions are mixed with insertions, a worst case of $\Theta(\log n)$ steps per insertion or deletion can occur.

In the next section we describe the list structure and give the search and insertion algorithms. In Section 2 we prove the time bound for consecutive insertions; this requires an analysis of node splitting in 2-3 tree insertion which applies to other algorithms using 2-3 trees. In the final section we discuss some practical issues arising in an implementation of the structure, describe some applications, and indicate directions for future work.

*/ This research was supported in part by National Science Foundation grant MCS75-22870 and by Office of Naval Research contract N00014-76-C-0688.

**/ A function $g(n)$ is $\Theta(f(n))$ if there exist positive constants c , c' , and n_0 with $cf(n) \le g(n) \le c'(f(n))$ for all $n \ge n_0$. Hence the 'Θ' can be read 'order exactly'; see [7] for further discussion of the Θ notation.

1. The Structure.

We use a data structure based on 2-3 trees [1;6]. A 2-3 tree 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 2-node, and one with 3-way branching a 3-node. 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.

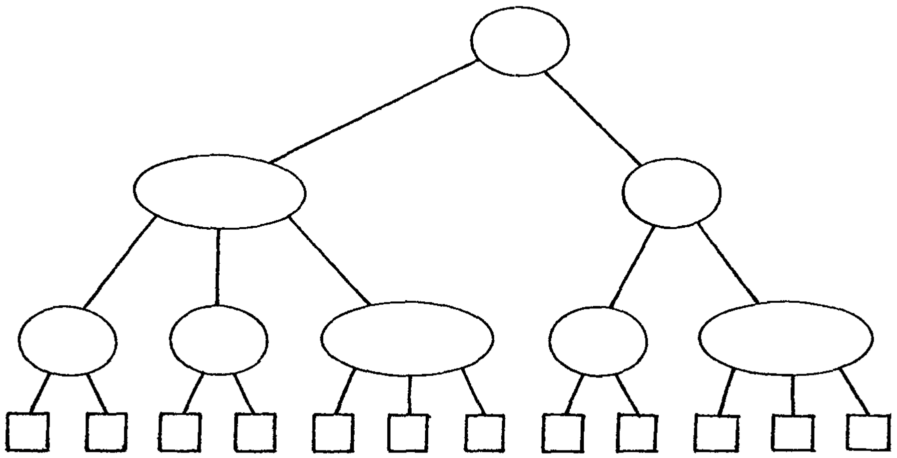


Figure 1. A 2-3 tree.

There are many schemes for associating data with the nodes of a 2-3 tree; the usefulness of a particular organization depends on 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 insertion 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 expands 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 split 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 to a 3-node, or the root is split. If the root splits, a new 2-node is created which has the two pieces of the root as its children, and this new node becomes the root.

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 subtrees 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. (This 2-3 search tree organization is similar but not identical to those given in [1, p. 147; 6, p. 468].)

This structure is elaborated in Figure 4 to support efficient access in the neighborhood of a finger. The arrangement of list elements and keys is not changed, but the edges between internal nodes are made traversible upwards as well as downwards, and horizontal links are added between internal nodes which are neighbors (adjacent on the same level).

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 our structure; the reasons for this decision will become evident when implementation considerations are discussed in Section 3. Note that the presence of a finger requires no change to the structure.

*/ We use $\lg n$ to denote $\log_2 n$.

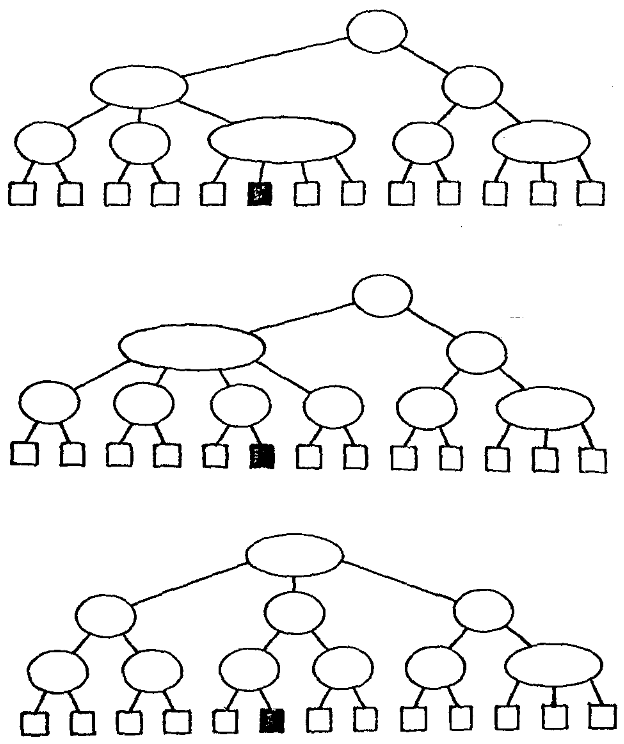


Figure 2. A 2-3 tree insertion.

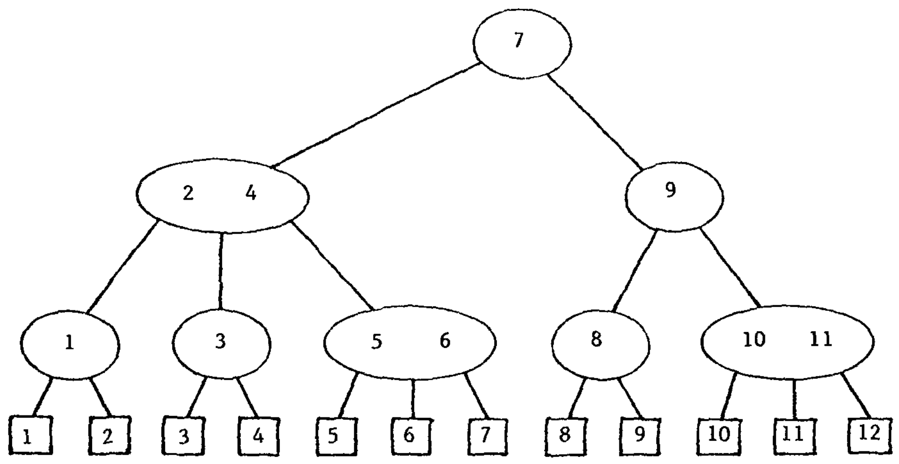


Figure 3. A 2-3 tree structure for sorted lists.

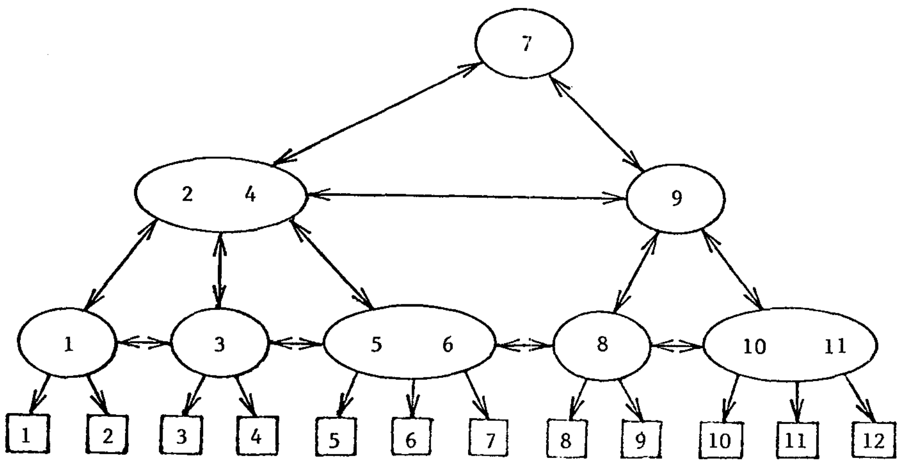


Figure 4. A structure to support finger searching.

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 Largestkey(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 ≥ LargestKey(f)   then return  SearchUpRight(f,k)
    elseif  k < SmallestKey(f)  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(p) or rNbr(p) = Nil then return SearchDown(p,k)
        else  q ← rNbr(p)
            if      k < SmallestKey(q)  then return  SearchDown2(p,q,k)
            elseif  k < LargestKey(q)   then return  SearchDown(q,k)
            else    p ← Parent(q)
            endif
        endif
    repeat
end SearchUpRight
{procedure SearchUpLeft is similar}
```

```
procedure SearchDown2(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(p)    then return  SearchDown(p,k)
        elseif $k \geq$ SmallestKey(q) then return  SearchDown(q,k)
        else   $p \leftarrow$ RightmostLink(p)
               $q \leftarrow$ LeftmostLink(q)
        endif
    repeat
    if $k <$ Key[RightmostLink(p)]  then return  p
                                    else return  q
    endif
end SearchDown2

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

The running time of FingerSearch is bounded by a constant times the height of the highest node examined, since it examines at most four of the nodes at each level. But 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 an entire subtree of height ℓ-2 all of whose keys lie between k and the keys of the finger node. Hence we have established the following property of FingerSearch :

Property 1. If the key k is d keys away from a finger f , then FingerSearch(f,k) runs in $\Theta(\log d)$ steps.

Now consider the operation of inserting a new external node with key k into the structure. It is clearly possible to update all of the links which are changed by a node splitting in constant time. The insertion procedure must also maintain the organization of keys in the internal nodes. If k is not the largest (rightmost) offspring of its parent p , then by the rule for key assignments k will be placed in node p . (If k is the largest, then k may be inserted into p's right neighbor unless k is larger than all keys in the tree; in that case, the key which was previously largest is placed in p , and k is unused.) If a "4-node" q splits during insertion, it will contain 3 keys; to maintain consistency the left and right keys are placed in the new 2-nodes, and the middle key is placed in the parent of q . This informal description forms the basis of a proof of the following property:

Property 2. A new external node can be inserted at a given position in the structure used by FingerSearch in $\Theta(1+s)$ steps, where s is the number of node splittings caused by the insertion.

Finally, consider the process of creating a new finger. This operation consists of copying a pointer to the desired terminal node, and hence requires $\Theta(1)$ time once we have reached this node by means of a search from another finger. Similarly, since a finger requires no change to the structure, any finger may be abandoned in constant time.

2. The Analysis of 2-3 Tree Splitting.

Any individual insertion into a 2-3 tree of size n may cause up to about lg n splittings of internal nodes to take place. On the other hand, if n consecutive insertions are made into such a tree, 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 less than the tree size. 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 $p_1 < p_2 < \cdots < p_k$, then the number of node splittings which took 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, 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 is greater than 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 the problem from being concerned 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 $p_1 < p_2 < \cdots < p_k$ 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 third marked 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 $\nu(a) + \nu(b) = \nu(a+b) + c$.

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

$$\begin{aligned} i &\leq c + \lceil \lg(b-a+1)\rceil \leq \nu(a) - \nu(b) + \nu(b-a) + \lceil \lg(b-a+1)\rceil \\ &\leq \nu(a) - \nu(b) + 2\lceil \lg(b-a+1)\rceil \quad . \end{aligned}$$ □

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 $p_1 < p_2 < \cdots < p_k$ to the root of T satisfies

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

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

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

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 $\ell(p)$ 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 $1 < i \leq k$.

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 $M(p_{i-1},p_i)$, 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,

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

Case 2. 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 if the edge out of t leading toward p_{i-1} is relabelled 0. Then $\ell(p_i) - \ell'(p_{i-1}) \leq p_i - p_{i-1}$ 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

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

(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; i.e., for any n and k there is a 2-3 tree with n external nodes which some sequence of k insertions causes within a constant factor of the given number of splits. We omit the proof of this fact.

Our goal in applying Theorem 1 to FingerSearch is to show that for any sequence of insertions, the cost (in steps) of the insertions is dominated by the cost of the searches. (This is clearly the case when searches are made from the root, but is less obvious when the tree is searched from the bottom up.)

Theorem 2. Let L be a linear list of size n, represented using the structure of Figure 4, with one finger established. Then in any sequence of searches, finger creations, and k insertions, the total cost of the k insertions is $O(\log(n+k) + \text{total cost of searches})$.

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 $p_1 < p_2 < \cdots < p_\ell$ 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-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 G, while never increasing its cost, until it becomes

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

where $r_1 < r_2 < \cdots < r_k$ 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 intially 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. □

3. Implementation and Applications.

In Section 1 we described our structure 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 3-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 the model of random 2-3 trees studied in [9], the ratio of 2-nodes to 3-nodes in a random tree 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 5 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 mentioned in [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].)

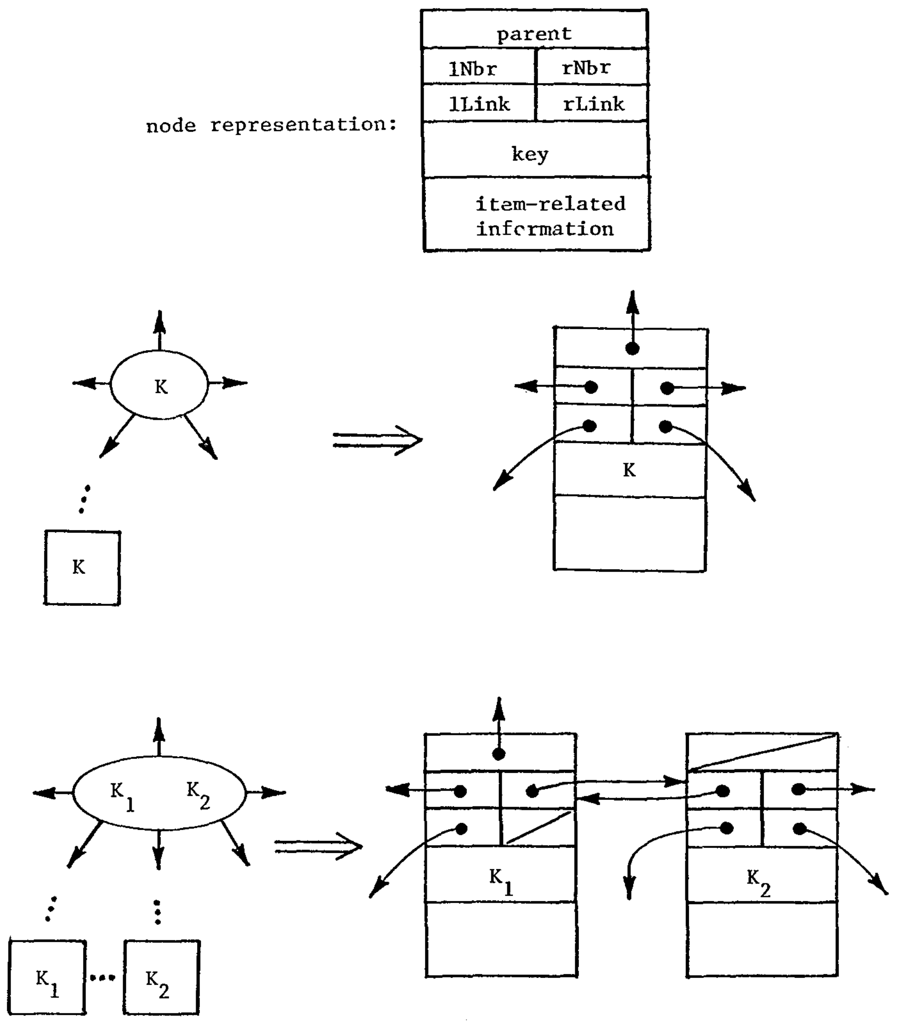
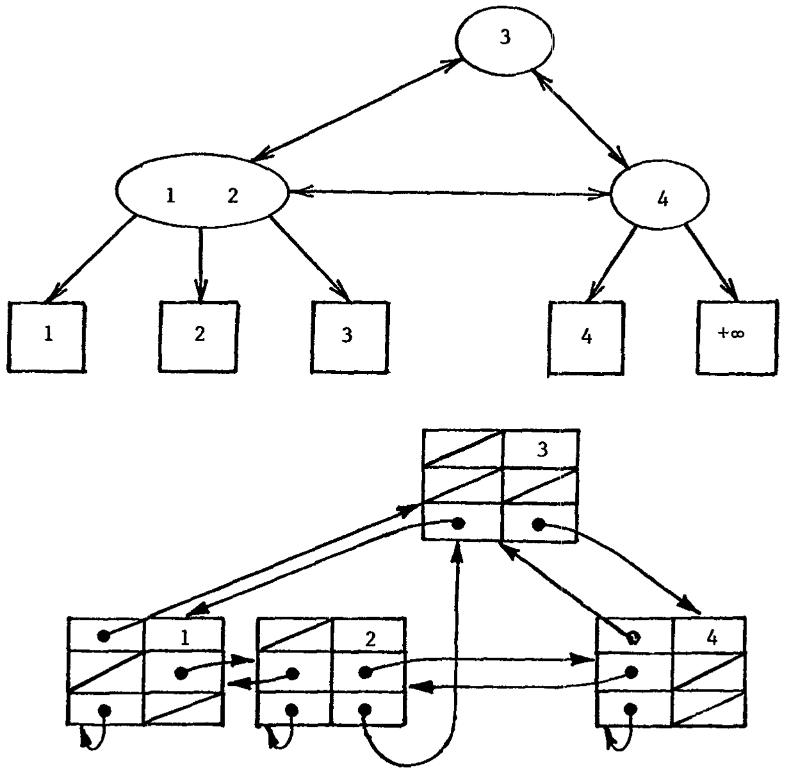


Figure 5. A storage representation for internal and external nodes.

An example which illustrates this representation is shown in Figure 6. 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 determine those ℓLinks and rLinks which 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 $\ell\text{Link}(p) = p$.

We now consider the potential applications of this list representation. One application, described in [4], is in sorting files which have a bounded number of inversions. The result proved in [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 $m \le n$, 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



node format:

parent	key
lNbr	rNbr
lLink	rLink

Figure 6. A structure and its storage representation.

thus $\sum_{1<i\leq m} d_i \leq n$. To maximize $\sum_{1<i\leq m} \log d_i$ 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.

An obvious question relating to our structure is whether it can be generalized so that 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 (see [4]). Thus a compromise between the unconstrained structure given here and the highly constrained structure of [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 Dept. Report STAN-CS-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 Dept. Ph. D. thesis, to appear.

[9] Andrew C.-C. Yao. "On random 2-3 trees," Acta Informatica (to appear).