PLANAR POINT LOCATION USING PERSISTENT SEARCH TREES

Neil Sarnak
Robert E. Tarjan

July, 1985

CS-TR-005-85

PLANAR POINT LOCATION USING PERSISTENT SEARCH TREES

Neil Sarnak

Courant Institute of Mathematical Sciences
New York University
New York, New York 11215

Robert E. Tarjan

Computer Science Department, Princeton University
Princeton, New Jersey 08544
and
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

July, 1985

ABSTRACT

The *planar point location problem* is that of preprocessing a polygonal subdivision of the plane so that, given a sequence of points, the polygon containing each point can be determined quickly. Several ways of solving this problem in $O(log\ n)$ query-time and $O(n)$ space are known, but they are all rather complicated. We propose a simple $O(log\ n)$-query time, $O(n)$-space solution, using *persistent search trees*. A persistent search tree differs from an ordinary search tree in that after an insertion or deletion, the old version of the tree can still be searched. We develop a persistent form of binary search tree that supports insertions and deletions in the present version and queries in any version, past or present.. The time per query or update is $O(log\ m)$, were m is the total number of updates, and the space needed is $O(1)$ per update. Our planar point location algorithm is an immediate application of this data structure.

PLANAR POINT LOCATION USING PERSISTENT SEARCH TREES

Neil Sarnak

Courant Institute of Mathematical Sciences
New York University
New York, New York 11215

Robert E. Tarjan

Computer Science Department, Princeton University
Princeton, New Jersey 08544
and
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

July, 1985

1. PLANAR POINT LOCATION

Let us consider a classical geometric retrieval problem. Suppose the Euclidian plane is subdivided into polygons by n line segments* that intersect only at their endpoints. (See Figure 1.) Given such a polygonal subdivision and a sequence of query points in the plane, the *planar point location problem* is the problem of determining, for each query point, the polygon containing it. (For simplicity we shall assume that no query point lies on a line segment of the subdivision.) We require that the answers to the queries be produced on-line; that is, each point must be located before the next point is known.

[Figure 1]

A solution to the point location problem consists of an algorithm that preprocesses the polygonal subdivision, building a data structure that facilitates location of individual

* We regard a line or half-line as being a line segment, and an infinite region whose boundary consists of a finite number of line segments as being a polygon.

query points. We measure the efficiency of such a solution by three parameters: the preprocessing time, the space required to store the data structure, and the time per query. Of these, the preprocessing time is generally the least important.

Many solutions to the point location problem have been proposed [9,10,12,17,21,22,30]. If binary decisions are used to locate the query points, $\Omega(\log n)$ time per query is necessary. Dobkin and Lipton [10] showed that this lower bound is tight, exhibiting a method with $O(\log n)$ query time needing $O(n^2)$ space and preprocessing time. The Dobkin-Lipton result raised the question of whether an $O(\log n)$ bound on query time can be achieved using only $O(n)$ space, which is optimal if the planar subdivision must be stored. Lipton and Tarjan [22] answered this question affirmatively by devising a complicated method based on the planar separator theorem [23].

More recent research has focused on providing a simpler algorithm with resource bounds the same as or close to those of the Lipton-Tarjan method. Algorithms with $O(\log n)$ query time using $O(n)$ space have been developed by Kirkpatrick [17], who used the fact that every planar graph has an independent set containing a fixed fraction of the vertices; by Edelsbrunner, Guibas, and Stolfi [12], who improved a method of Lee and Preparata [21] that uses the notion of separating chains; and by Cole [9], who noted tht the Dobkin-Lipton approach reduces planar point location to a problem of storing and accessing a set of similar lists.

Cole's observation is the starting point for our work. Let us review the Dobkin-Lipton construction. Draw a vertical line through each vertex (intersection of line segments) in the planar subdivison. (See Figure 2.) This splits the plane into vertical slabs. The line segments of the subdivision intersecting a slab are totally ordered, from the bottom to the top of the slab. Associate with each line segment the polygon just above it. Now it is possible to locate a query point with two binary searches: the first, on the x-coordinate, locates the slab containing the point; the second, on the line segments intersecting the slab, locates the nearest line segment below the point, and hence determines the polygon containing the point. (By introducing a dummy line segment running from $(-\infty, -\infty)$ to $(\infty, -\infty)$, we can guarantee that below every point there is a line segment.) Since testing whether a point is above or below a line segment takes $O(1)$ time, a point query takes $O(\log n)$ time. Unfortunately, if we build a separate search structure (such as a binary search tree) for each slab, the worst-case space requirement is $\Theta(n^2)$, since $\Theta(n)$ line segments can intersect $\Theta(n)$ slabs.

[Figure 2]

We can reduce the space bound by noticing as Cole did that the sets of line segments intersecting contiguous slabs are similar. Think of the x-coordinate as time. Consider how the set of line segments intersecting the current slab changes as the time increases from $-\infty$ to $+\infty$. As the boundary from one slab to the next is crossed, certain segments are deleted from the set and other segments are inserted. Over the entire time range, there are $2n$ insertions and deletions, one insertion and one deletion per segment. (Think of line segments going to $-\infty$ in the x-coordinate as being inserted at time $-\infty$, and line segments going to $+\infty$ in the x-coordinate as being deleted at time $+\infty$.)

We have thus reduced the point location problem to the problem of storing a sorted set subject to insertions and deletions so that all past versions of the set, as well as the current version, can be accessed efficiently. In general we shall call a data structure *persistent* if the current version of the structure can be modified and all versions of the structure, past and present, can be accessed. Ordinary data structures, which do not support access in the past, we call *ephemeral*.

Cole solved the point location problem by devising a persistent representation of sorted sets that occupies $O(m)$ space and has $O(\log m)$ access time, where m is the total number of updates (insertions and deletions) starting from an empty set. However, his data structure has two drawbacks. First, his method is indirect, preceeding by way of an intermediate problem in which item substitutions but not insertions or deletions are allowed. Second, the entire sequence of updates must be known in advance, making the data structure unusable in situations where the updates take place on-line. We shall propose a simpler data structure that overcomes these drawbacks.

Our main result, presented in Section 3, is a persistent form of binary search tree with an $O(\log m)$ worst-case access/insert/delete time and an amortized* space requirement of $O(1)$ per update. Our structure has neither of the drawbacks of Cole's. It provides a simple $O(n)$-space, $O(\log n)$-query-time point location algorithm. It can also replace Chazelle's "hive graph" [6], a rather complicated data structure with a variety of uses in geometric searching. Section 4 contains a brief discussion of these applications along with extensions and related results, details of which will appear in Sarnak's Ph.D. thesis [32] and in a forthcoming paper [33].

* By *amortized complexity* we mean the complexity of an operation averaged over a worst-case sequence of operations. For a full discussion of this concept, see Tarjan's survey paper [36].

2. PERSISTENT SORTED SETS AND SEARCH TREES

We are now faced with a problem that is purely in the realm of data structures, the *persistent sorted set problem*. We wish to maintain a set of items that changes over time. The items have distinct *keys*, with the property that any collection of keys of items that are in the set simultaneously can be totally ordered. (The keys of two items that are not in the set at the same time need not be comparable.) Three operations on the set are allowed:

access(x,t): Find and return the item in the set at time t with greatest key less than or equal to x. If there is no such item, return a special *null* item.

insert(e,t): At time t, insert item e (with predefined key) into the set, assuming it is not already there. Item e remains in the set until it is explicitly deleted.

delete(e,t): At time t, delete item e from the set, assuming it is there.

Starting with an empty set, we wish to perform on-line a sequence of operations, including m updates (insertions and deletions), with the following property:

(*) Any update occurs at a time no earlier than any previous operation in the sequence. That is, updates are allowed only in the present.

The explicit time parameter t in the operations formalizes the notion of persistence. Property (*) allows accesses to take place either in the present or in the past. In the usual ephemeral version of the sorted set problem, the time of an operation is implicit, coresponding to its position in the sequence of operations. An equivalent definition of the ephemeral problem is obtained by requiring the sequence of operations to have the following stronger property in place of (*): the operations in the sequence occur in non-decreasing order by time.

This problem and variants of it have been studied by many authors [7,9,11,20,25,26,29,31]. Dobkin and Munro [11] considered the problem of maintaining a persistent list subject to access, insertion and deletion by list position. (The items in the list have positions 1 through n counting from the front to the back of the list.) The persistent list problem seems to be harder than the persistent sorted set problem. Dobkin and Munro proposed an off-line method (all updates occur in the sequence before all accesses) with $O((\log m)^2)$ access time using $O(m \log m)$ space. Overmars [29] proposed an on-line method for the persistent list problem with $O(\log m)$ access time using $O(m \log m)$

space. Overmars also studied the much easier version of the persistent sorted set problem in which an operation *access(x,t)* need only return an item if the set contains an item with key exactly equal to x. For this version, he developed an $O(m)$-space, $O(\log m)$-access-time on-line algorithm. Chazelle [7] devised an $O(m)$-space, $O((\log m)^2)$-access-time method for the off-line version of the original persistent sorted set problem. As discussed in Section 1, Cole [9] discovered an $O(m)$-space, $O(\log m)$-access-time off-line algorithm.

All these methods use data structures that are somewhat ad hoc and baroque. A more direct approach is to start with an ephemeral data structure for sorted sets or lists and make it persistent. This idea was pursued independently by Myers [25,26], Krijnen and Meertens [20], and Reps, Teitelbaum, and Demers [31], who independently proposed essentially the same idea, which we shall call *path copying*. The resulting data structure can be used to represent both persistent sorted sets and bound persistent lists with an $O(\log m)$ time bound per operation and an $O(\log m)$ space bound per update.

In the remainder of this section we shall review binary search trees and how they can be made persistent using path copying. In Section 3 we propose a new method that uses space even more efficiently than path copying. It leads to a data structure for persistent sorted sets (but not persistent lists) that has bounds of $O(\log m)$ worst-case time per operation and $O(1)$ amortized space per update.

A standard data structure for representing ephemeral sorted sets is the *binary search tree*. This is a binary tree* containing the items of the set in its nodes, one item per node, with the items arranged in *symmetric order*: if x is any node, the key of the item in x is greater than the keys of all items in its left subtree and less than the keys of all items in its right subtree. The symmetric-order item arrangement allows us to perform an access operation by starting at the tree root and searching down through the tree, along a path determined by comparisons of the query key with the keys of items in the tree: if the query key is equal to the key of the item in the current node, we terminate the access by returning the item in the current node; if it is less, we proceed to the left child of the current node; if it is greater, we proceed to the right child. Either the search terminates having found an item with key equal to the query key, or it runs off the bottom of the tree. In the latter case, we return the item in the node from which the search last went right; if there is no such node, we return null.

The time for an access operation in the worst case is proportional to the depth of the tree. If the tree is binary, its depth is at least $\lfloor \log n \rfloor + 1$, where n is the number of tree nodes. This bound is tight for *balanced binary trees* which have depth $O(\log n)$ and

* See the books of Knuth [18] and Tarjan [34] for our tree terminology.

insertion and deletion time bounds of $O(log\ n)$ as well. There are many types of balanced trees, including *AVL* or *height-balanced trees* [1], *trees of bounded balance* or *weight-balanced trees* [27], and *red-black trees* [13]. In such trees balance is maintained by storing certain *balance information* in each node (of a kind that depends upon the type of tree) and rebalancing after an insertion or deletion by performing a series of *rotations* along the access path (the path from the root to the inserted or deleted item). A rotation (see Figure 3) is a local transformation that changes the depths of certain nodes, preserves symmetric order, and takes $O(1)$ time, assuming that a standard binary tree representation is used (such as storing two pointers in each node, to its left and right children).

[Figure 3]

To make our discussion concrete, we shall restrict our attention to red-black trees. (As noted below, our ideas also apply to certain other kinds of balanced trees.) In a red-black tree each node has a color, either *red* or *black*, subject to the following constraints:

(i) All missing (external) nodes are regarded as black;

(ii) All paths from the root to a missing node contain the same number of black nodes;

(iii) Any red node, if it has a parent, has a black parent.

This definition is due to Guibas and Sedgewick [13]. Bayer [2] introduced these trees, calling them *symmetric binary B-trees*. Olivié [28] gave an equivalent definition (see [35]) and used the term *half-balanced trees*.

Updating red-black trees is especially efficient as compared to updating other kinds of balanced trees. Rebalancing after an insertion or deletion can be done in $O(1)$ rotations and $O(log\ n)$ color changes [35]. The insertion and deletion algorithms are as follows. To perform an insertion, we proceed as in an access operation. At the place where the search runs off the bottom of the tree, we attach a new node containing the new item. We color this node red. This preserves the black constraint (ii) but may violate the red constraint (iii). If there are now two red nodes in a row the topmost of which has a red sibling, we color the topmost red node and its red sibling black and their common parent (which must be black) red. (See Figure 4(a).) This may produce a new violation of the red constraint. We repeat the transformation of Figure 4(a), moving the violation up the tree, until this transformation no longer applies. If there is still a violation we apply the appropriate one of the transformations in Figures 4(b), (c), and (d) to eliminate the violation. This terminates the insertion. The only rotations are in the terminal cases: 4(c) takes one rotation and 4(d) takes two.

[Figure 4]

A deletion is similar. We first search for the item to be deleted. If it is in a node with a left child, we swap the item with its predecessor (in symmetric order), which we find by taking a left branch and then right branches until reaching a node with no right child. Now the item to be deleted is in a node with no right child. We delete this node and replace it by its left child (if any). This does not affect the red constraint but will violate the black constraint if the deleted node was black. If there is a violation the replacing node (which may be missing) is *short*: paths down from it contain one fewer black node than paths down from its sibling. We bubble the shortness up the tree by repeating the recoloring transformation of Figure 5(a) until it no longer applies. Then we perform the transformation of Figure 5(b) if it applies, followed if necessary by one application of 5(a), (c), (d) or (e). The maximum number of rotations needed is three.

[Figure 5]

Let us now consider how to make red-black trees persistent. We need a way to retain the old version of the tree when a new version is created by an update. We can of course copy the entire tree each time an update occurs, but this takes $O(n)$ time and space per update. The idea of Myers [25,26], Krijnen and Meertens [20], and Reps, Teitelbaum, and Demers [31] is to copy only the nodes in which changes are made. Any node that contains a pointer to a node that is copied must itself be copied. Assuming that every node contains pointers only to its children, this means that copying one node causes a ripple of copying, back through ancestors (along the access path) all the way to the root of the tree. Thus we shall call this method *path copying*. The effect of this method is to create a set of search trees, one for each update, that have different roots but share common subtrees. Since node colors are needed only for update operations, all of which take place in the most recent version of the tree, we do not need to copy a node when its color changes; we merely overwrite the old color. This saves a constant factor in space. (See Figure 6.)

[Figure 6]

Since an insertion or deletion in a red-black tree changes only nodes along a single access path, the time and space needed per update is $O(\log n)$. If we use the path copying method to represent a persistent sorted set for which the update times are arbitrary real numbers, we must build an auxiliary structure to facilitate access to the appropriate root when searching in the past. An array of pointers to the roots, ordered by time of creation, suffices. We can use binary search in this array to access the appropriate root. This increases the time necessary for an access in the past to $O(\log m)$. If the update times are consecutive integers, we can use direct access in tha array to provide $O(1)$-time access to the roots, and the time for an access operation is only $O(\log n)$.

Path copying works on any kind of balanced tree, not just on red-black trees. Myers used AVL trees, Krijnen and Meertens used B-trees, and Reps, Teitelbaum and Demers used 2,3 trees. Path copying is also quite versatile in the applications it supports. By storing in each node the size of the subtree rooted there, we can obtain an implementation of persistent lists (in which access is by position rather than by key). We also have the ability to update *any* version, rather than just the current one, provided that an update is assumed to create an entirely new version, independent of all other versions. In order to have this more general kind of updatability, we must copy a node when its balance information changes as well as when one of its pointers changes, but this increases the time and space needed for updates by only a constant factor.

3. SPACE-EFFICIENT PERSISTENT SEARCH TREES

A major drawback of the path copying method is its non-linear space usage. In this section we shall propose a remedy to this problem. We shall restrict our attention to the original version of the sorted set problem, and we shall use the fact that past balance information need not be saved (although this is not essential, as we shall discuss at the end of the section).

Path copying uses non-linear space because a single update can cause $\Omega(\log n)$ nodes to be copied: the entire access path must be copied to accomodate a single pointer change. However ordinary ephemeral red-black trees need only $O(1)$ pointer changes per update. Thus, there is hope that by avoiding the copying of entire access paths, we may be able to reduce the space per update in persistent red-black trees to $O(1)$.

Our first idea is to avoid node copying entirely. Instead, we allow individual nodes to become arbitrarily "fat"; that is, to hold an arbitrary number of pointers. We simulate the ephemeral insertion and deletion algorithms as follows. When an ephemeral algorithm calls for a pointer to be changed, we add the new pointer to the node (without erasing the old pointer), inserting as well a time stamp that indicates when the new pointer was added and a bit that indicates whether the new pointer is a left or right pointer. (This bit is actually redundant, since we can determine whether a pointer is left or right by comparing the key of the item in the node containing the pointer to that of the item in the node indicated by the pointer.) When a node color is changed we overwrite the old color. (See Figure 7.)

[Figure 7]

With this approach an insertion or deletion in a persistent red-black tree takes only $O(1)$ space, since an insertion creates only one new node and either kind of update causes only $O(1)$ pointer changes. The drawback of the method is its time penalty: since a node can contain an arbitrary number of left or right pointers, deciding which one to follow during a search is not a constant-time operation. If we use binary search by time stamp to decide which pointer follow, then choosing the correct pointer takes $O(\log m)$ time, and the time for an access, insertion, or deletion is $O((\log n)(\log m))$.

We can eliminate this time penalty by introducing limited node copying. We allow each node to hold k pointers in addition to its original two. We choose k to be a small constant; $k=1$ will do. When adding a pointer to a node, if there are no empty slots for pointers, we copy the node, setting the initial left and right pointers of the copy to their latest values. (Thus the new node has k empty slots.) A new pointer must also be stored in the latest parent of the copied node. This will cause the parent to be copied if it has no free slot. Node copying ripples back through ancestors until the root is copied or a node with a free slot is reached. (See Figure 8.)

[Figure 8]

Searching the resulting data structure is quite easy: when arriving at a node, we determine what pointer to follow by examining the key to decide whether to branch left or right and examining the time stamps of the extra pointers to select among multiple left or multiple right pointers. (We follow the pointer with the latest time stamp no greater than the search time if there is one, or else the initial pointer.) As noted in Section 2, if the update times are arbitrary real numbers we must build an auxiliary array to guide access operations to the proper roots. This makes the time for an access operation $O(\log m)$, whereas the time for an update operation is $O(\log n)$. However, in practice the number of roots is likely to be much smaller than m, since a root will be duplicated relatively infrequently. If the update times are consecutive integers, the auxiliary array provides

$O(1)$-time access to the roots.

It remains for us to analyze the space used by the data structure. As with path copying, a single update operation can cause $O(log\ n)$ node copyings. However, amortized over a sequence of updates, the number of node copyings is only $O(1)$, as we shall now show.

At any given time, we partition the nodes in the data structure into two classes, *live* and *dead*. The live nodes are those reachable from the latest root by following pointers valid at the current time; they comprise the latest version of the search tree. All other nodes are dead; they are unaffected by any subsequent update.

To carry out the analysis we use the potential paradigm [36]. We define the *potential* of a configuration of the data structure to be the number of active nodes minus $1/k$ times the number of free slots in active nodes. We define the *amortized space cost* of an update opertion to be the actual number of nodes it creates plus the net incease in potential it causes. With these definitions, the actual number of nodes created by a sequence of updates is bounded by the sum over all updates of the amortized space cost minus the net increase in potential over the sequence. If we start with an empty data structure, the initial potential is zero, and since the potential is always non-negative the total amortized space cost is an upper bound on the actual number of nodes created.

The definition of potential is such that a node copying has an amortized space cost of zero. Storing a new pointer in a node has an amortized space cost of $1/k$. The addition of a new node during an insertion has an amortized space cost of one. Since an insertion or deletion requires storing $O(1)$ new pointers not counting node copying, the amortized space cost of an update is $O(1)$. A more careful count shows that an insertion has an amortized space cost of at most $1 + 6/k$; a deletion, at most $7/k$. In the special case of $k=1$, the amortized space cost per update is slightly less than indicated by these bounds: at most 6 for an insertion or deletion.

The choice $k=1$ is probably the most convenient in practice and is certainly the easiest to implement. However, choosing a larger value of k may reduce the space needed by the data structure, since although the space per node increases, the number of node copyings decreases. The best choice of k depends on the exact way nodes are stored in memory and on the average (as opposed to worst-case) number of new pointers created by updates. Nevertheless, we shall give a simplified analysis based on the amortized bounds derived above. Suppose that memory is divided into words, each of which is large enough to hold an item, a time stamp, or a pointer. We shall ignore the space needed to store node colors and the types of extra pointers (left or right); as noted above the latter information is redundant and the color of a node can if necessary be encoded by swapping or not swapping the original left and right pointers in a node. Under these assumptions a node requires $2k+3$ words of memory, and the amortized space cost in words per update is at most $(2k+3)(1+6/k) = 2k + 18/k + 15$. This is minimized at 27 words per update for $k=3$. This choice is only marginally better than the 30 words per update (six nodes of five words each) needed for $k=1$. Both these estimates are probably much larger than the expected values.

We close this section with a few remarks about the generality of our technique. The fact about red-black trees that we have used to make our construction work is that only $O(1)$ pointer changes are necessary to rebalance after an insertion or deletion. Although this bound is worst-case, an amortized $O(1)$ bound will do as well, since the resulting space bound is amortized in either case. This means that top-down updating of red-black trees [37] can be used, or red-black trees can be replaced by certain other kinds of balanced trees, such as weight-balanced trees [4] or "weak" or "hysterical" B-trees [14,15,24].

4. APPLICATIONS AND EXTENSIONS

We have proposed a data structure for representing persistent sorted sets. Our structure has $O(\log m)$ access time, $O(\log n)$ update time, and needs $O(1)$ amortized space per update starting from an empty set. Here n is the current set size and m is the total number of updates. Our resource bounds match those of Cole [9], but our data structure is on-line and is simple enough to have potential practical applications.

Our structure supports various additional operations on sorted sets. In particular, it supports range queries of the following kind:

access range(x,y,t): Find and return all items in the set at time t with keys between x and y (inclusive).

To carry out an access range operation, we proceed exactly as in an ephemeral search tree: we search for x, search for y, and return all items in nodes between the two access paths, as well as the appropriate ones on the access paths. The time needed for an access range operation is $O(\log m + k)$, where k is the number of items returned.

We can also represent several sets simultaneously, and perform joining (concatenation) and splitting of sets. The time per join or split is $O(\log n)$; the amortized space cost is $O(1)$ for a join, $O(\log n)$ for a split [33].

As discussed in Section 1, our structure provides an efficient solution to the planar point location problem. For a planar subdivision of n line segments, the preprocessing time necessary to build the data structure is $O(n \log n)$, the space needed is $O(n)$, and the query time is $O(\log n)$. Although these bounds have been obtained by others [9,13,17,22], our method is simple enough to be useful in practice as well as efficient in theory.

Our structure also supports a generalization of the planar point location problem in which the queries of the following form: given a vertical line segment, report all polygons the segment intersects. Such a query is equivalent to an access range operation on the corresponding persistent sorted set and thus takes $O(\log n + k)$ time where k is the number of reported polygons. This bound has also been obtained by Chazelle [6], but only by using a complicated data structure, the *hive graph*, which is built as an extension to a data structure for the planar point location problem. Our structure solves both problems at once.

Chazelle gives a number of applications of hive graphs to geometric retrieval problems; for each of these, our structure provides a simpler solution. As an example, given a collection of line segments in the plane with i cossings, we can in $O((n+i)\log n)$ time construct a data structure of size $O(n+i)$ that, given a vertical query segment, will allow us to report all data line segments the query segment crosses in $O(\log n + k)$ time, where k is the number of reported segments. Cole [9] gives several other applications to which our structure applies.

We have obtained several extensions to the result presented here. The limited node copying technique generalizes to show that any ephemeral linked data structure, provided its nodes have constant in-degree as well as constant out-degree, can be made persistent at an amortized space cost of $O(1)$ per structural change and an additive $O(\log m)$ time penalty per access [33]. Whereas limited node copying as discussed in the present paper resembles node-spliting in B-trees, the generalized technique resembles the "fractional cascading" idea of Chazelle and Guibas [9]. Among other applications, the generalized technique allows the addition of extra pointers, such as parent pointers and level links [5], to persistent red-black trees.

Our implementation of persistent search trees, although more space-efficient than the path copying method, is not as versatile. For example, path copying provides to a representation for persistent lists as well as persistent sorted sets. For the list application limited node copying is equivalent to path copying, because the size information necessary for access by position must be updated all the way along an access path after any insertion or deletion, causing $\Theta(\log n)$ space usage per update. As noted in Section 2, path copying also provides the ability to update *any* version, rather than just the current one. Adding additional pointers, such as parent pointers, to the resulting data structure seems difficult. Nevertheless, path copying can be extended to finger search trees,* reducing the space usage for updates in the vicinity of fingers [33].

There are many open problems concerning geometric retrieval problems and persistent data structures. Perhaps one of the most interesting is how to make our planar point location algorithm, or any such algorithm, dynamic, so that line segments can be inserted and deleted on-line. The dynamization techniques of Bentley and Saxe [3] provide a way to handle insertions while presenting the $O(1)$ space bound. However, the access and insertion time becomes $O((\log n)^2)$. Deletion seems to be harder to handle. An even more challenging problem is to find a persistent representation for a dynamically changing planar subdivision. A good data structure for this purpose would have many applications in computational geometry [9].

* A *finger search tree* is a search tree augmented with a few pointers to favored nodes, called *fingers*. Access and update operations in the vicinity of fingers are especially efficient [5,15,16,19,39].

REFERENCES

[1] G. M. Adelšon-Velškii and E. M. Landis, "An algorithm for the organization of information," *Soviet Math. Dokl.* 3(1962), 1259-1262.

[2] R. Bayer, "Symmetric binary *B*-trees: data structure and maintenance algorithms," *Acta Informatica* 1(1972) 290-306.

[3] J. L. Bentley and J. B. Saxe, "Decomposable searching problems I: static-to-dynamic transformation," *J. Algorithms* 1(1980), 301-358.

[4] N. Blum and K. Mehlhorn, "On the average number of rebalancing operations in weight-balanced trees," A-78/06, Fachbereich Angewandte Mathematik und Informatik, Universität des Saarlandes, Saarbrücken, West Germany, 1978.

[5] M. R. Brown and R. E. Tarjan, "Design and analysis of data structures for representing sorted lists," *SIAM J. Comput.* 9(1980) 594-614.

[6] B. Chazelle, "Filtering search: a new approach to query-answering," *Proc.* 24^{th} *Annual IEEE Symp. on Foundations of Computer Science* (1983), 122-132.

[7] B. Chazelle, "How to search in history," *Information and Control*, to appear.

[8] B. Chazelle and L. J. Guibas, "Fractional cascading: a data structuring technique with geometric applications," to appear.

[9] R. Cole, "Searching and storing similar lists," *J. Algorithms*, to appear.

[10] D. Dobkin and R. J. Lipton, "Multidimensional searching problems, *SIAM J. Comput.* 5(1976), 181-186.

[11] D. P. Dobkin and J. I. Munro, "Efficient uses of the past," *Proc.* 21^{st} *Annual IEEE Symp. on Foundations of Computer Science* (1980), 200-206.

[12] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, "Optimal point location in a monotone subdivision," Digital Systems Research Center, 1984.

[13] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," *Proc.* 19^{th} *Annual IEEE Symp. on Foundations of Computer Science* (1978), 8-21.

[14] S. Huddleston and K. Mehlhorn, "Robust balancing in *B*-trees," *Lecture Notes in Computer Science* 104, Springer-Verlag, Berlin, West Germany (1981), 234-244.

[15] S. Huddleston and K. Mehlhorn, "A new data structure for representing sorted lists," *Acta Informatica* 17(1982), 157-184.

[16] S. Huddleston, "An efficient scheme for fast local updates in linear lists," Dept. of

Information and Compuer Science, University of California, Irvine, CA, 1981.

[17] D. Kirkpatrick, "Optimal search in planar subdivisions," *SIAM J. Ccmput.* 12(1983), 28-35.

[18] D. E. Knuth, *The Art of Computer Progamming, Vol. 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1973.

[19] S. R. Kosaraju, "Localized search in sorted lists," *Proc. 14^{th} Annual ACM Symp. on Theory of Computing* (1981), 62-69.

[20] T. Krijnen and L. G. L. T. Meertens, "Making *B*-trees work for *B*," IW 219/83, The Mathematical Centre, Amsterdam, The Netherlands, 1983.

[21] D. T. Lee and F. P. Preparata, "Location of a point in a planar subdivision and its applications," *SIAM J. Comput.* 6(1977), 594-606.

[22] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," *Proc.* 18^{th} *Annual IEEE Symp. on Foundations of Computer Science* (1977), 162-170.

[23] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM J. Appl. Math.* 36(1979), 177-189.

[24] D. Maier and S. C. Salveter, "Hysterical *B*-trees," *Inform. Process. Lett.* 12(1981), 199-202.

[25] E. W. Myers, "AVL dags," TR 82-9, Dept. of Computer Science, The University of Arizona, Tucson, AZ, 1982.

[26] E. W. Myers, "Efficient applicative data types," *Conf. Record Eleventh Annual ACM Symp. on Piniciples of Programming Languages* (1984), 66-75.

[27] J. Nievergelt and E. M. Reingold, "Binary search trees of bounded balance," *SIAM J. Comput.* 2(1973), 33-43.

[28] H. Olivié, "A new class of balanced search trees: half-balanced binary search trees," *RAIRO Informatique Théoretique* 16(1982), 51-71.

[29] M. H. Overmars, "Searching in the past I," *Information and Control*, to appear.

[30] F. P. Preparata, "A new approach to planar point location," *SIAM J. Comput.* 10(1981), 473-482.

[31] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language-based editors," *ACM Trans. on Prog. Sys. and Lang.* 5(1983), 449-477.

[32] N. Sarnak, "Persistent data structures," Ph.D. Thesis, Dept. of Computer Science, New York University, New York, NY, to appear.

[33] N. Sarnak and R. E. Tarjan, "Persistent search trees and geometric retrieval," to

appear.

[34] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[35] R. E. Tarjan, "Updating a balanced search tree in $O(1)$ rotations," *Inform. Process. Lett.* 16(1983), 253-257.

[36] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Alg. Disc. Meth.* 6(1985), 306-318.

[37] R. E. Tarjan, "Efficient top-down updating of red-black trees," to appear.

[38] A. K. Tsakalidis, "AVL-trees for localized search," *Lecture Notes in Computer Science* 172, Springer-Verlag, Berlin, West Gemany (1984), 473-485.

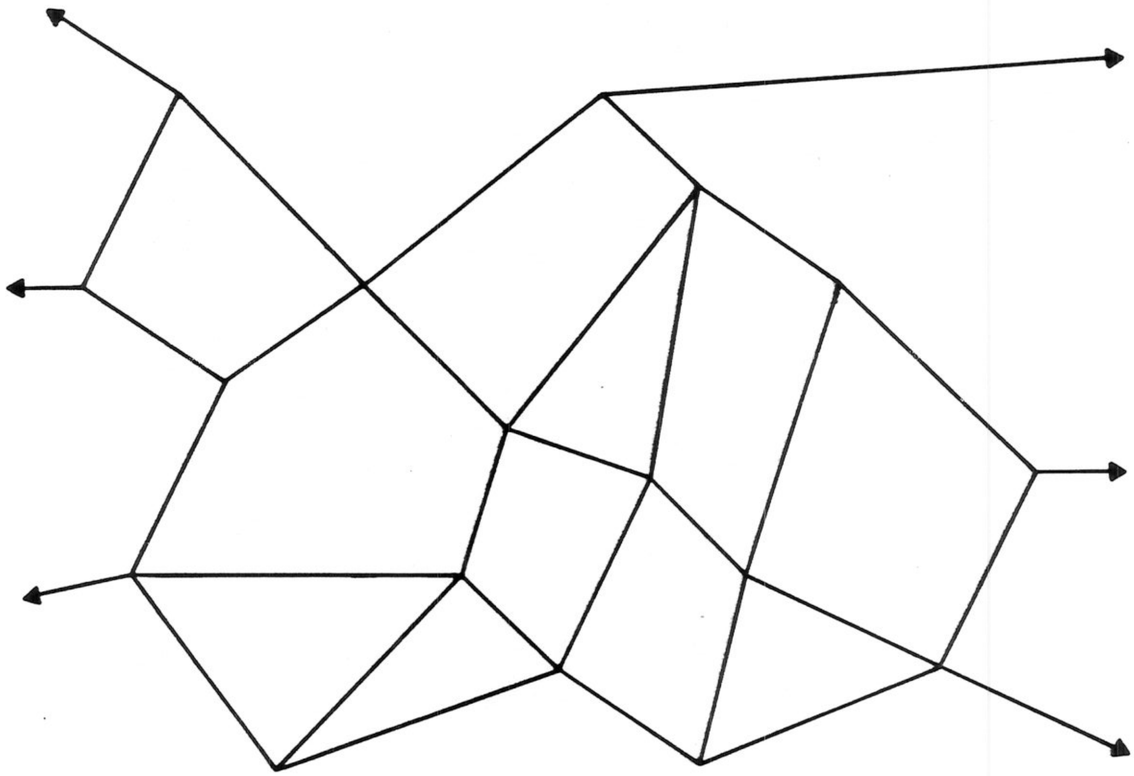

Figure 1. A polygonal subdivision. Arrows denote line segments going to infinity.

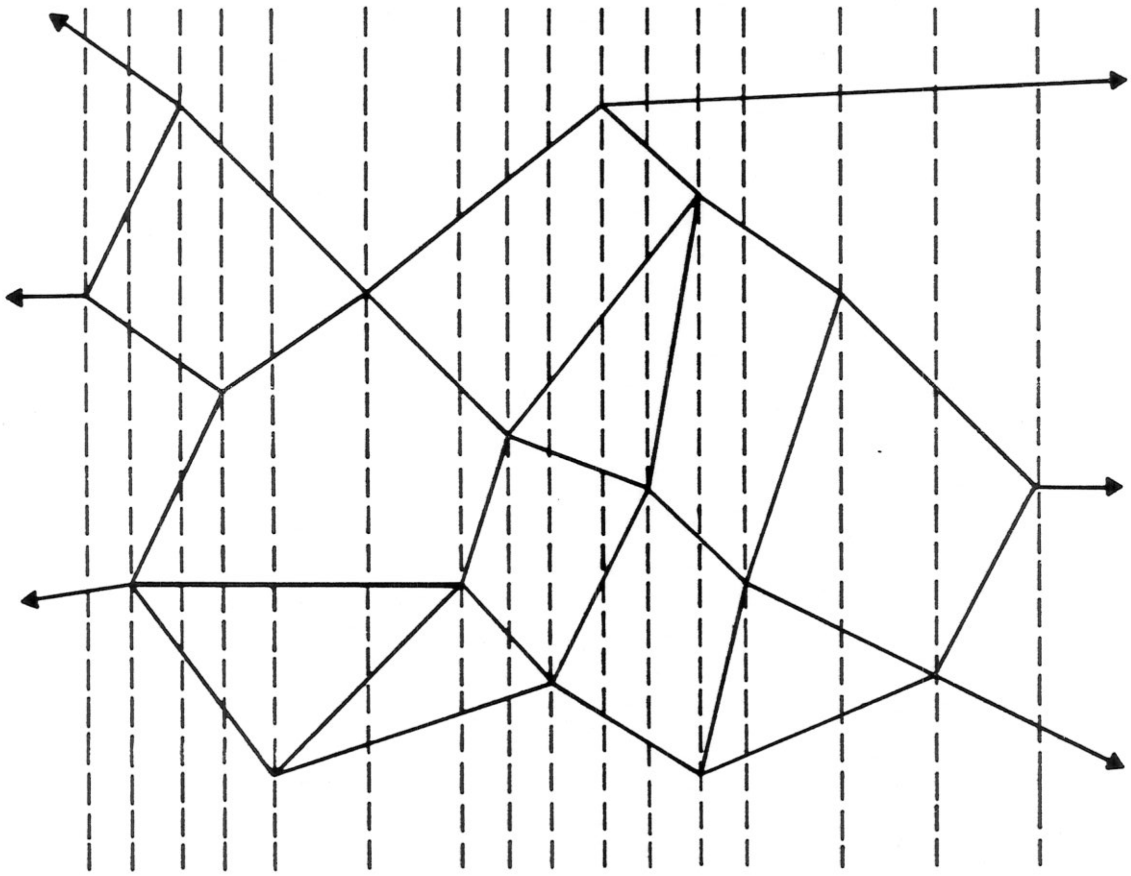

Figure 2. The polygonal subdivision of Figure 1 divided into slabs. The dashed lines are slab boundaries.

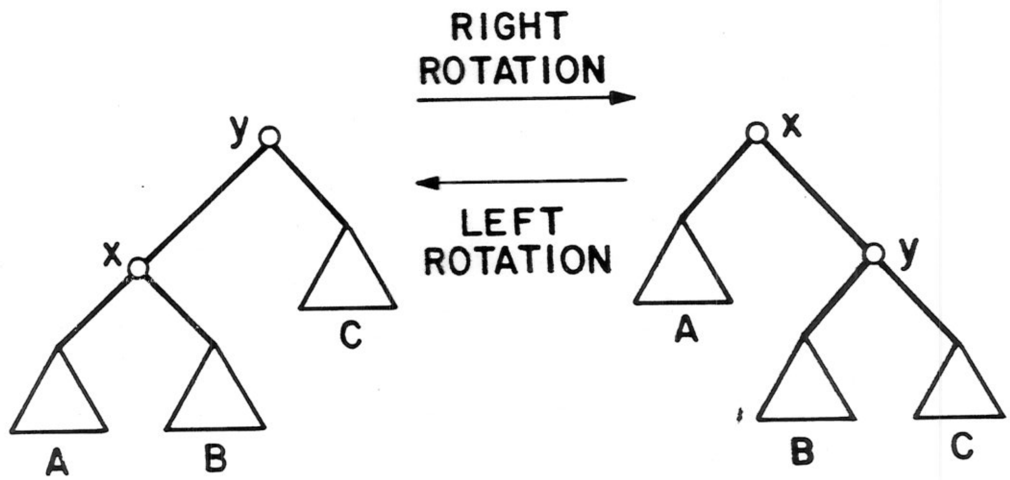

Figure 3. A rotation in a binary tree. The tree shown can be a subtree of a larger tree.

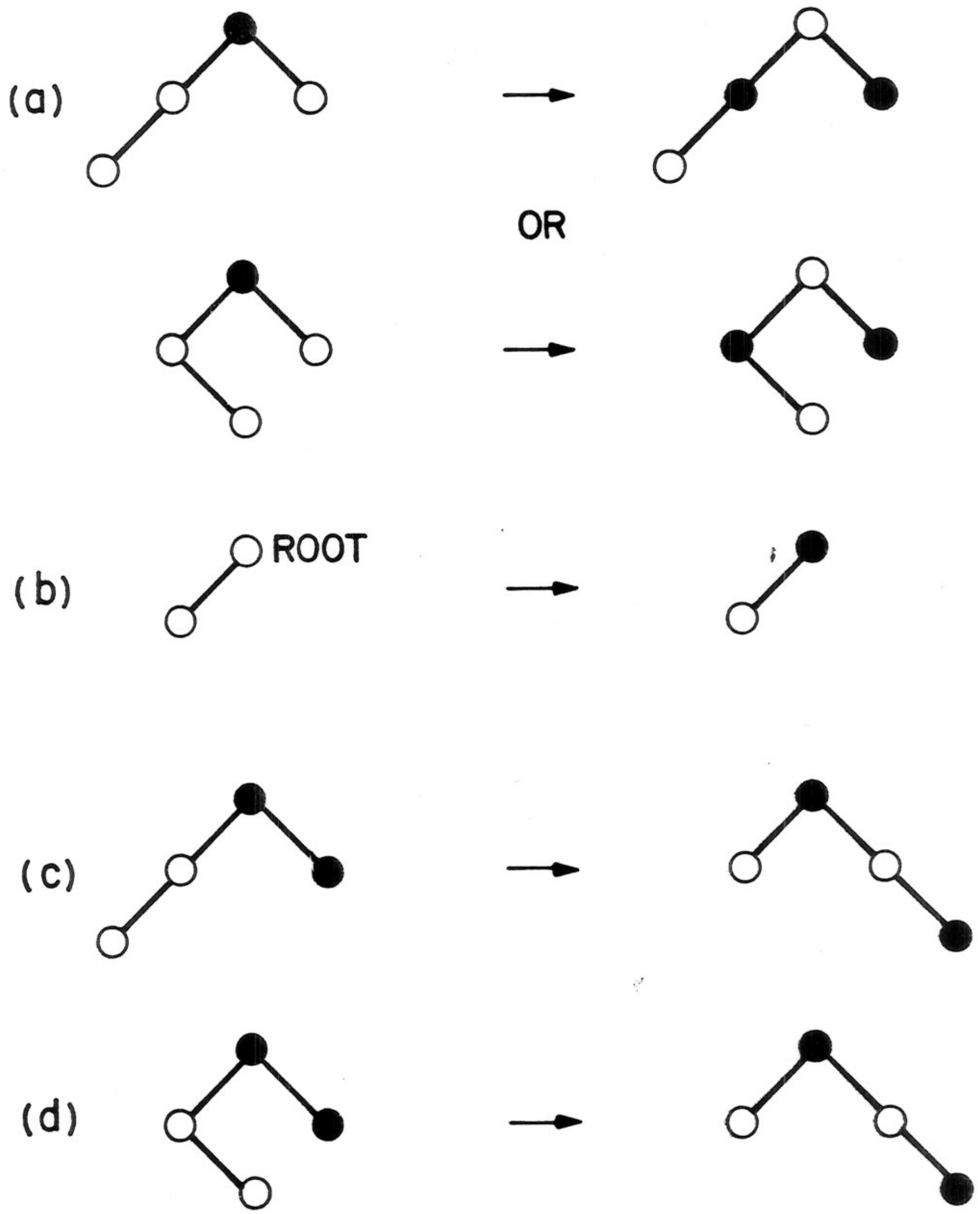

Figure 4. The rebalancing transformations in red-black tree insertion. Symmetric cases are omitted. Solid nodes are black; hollow nodes are red. All unshown children of red nodes are black. In cases (c) and (d) the bottommost black node shown can be missing.

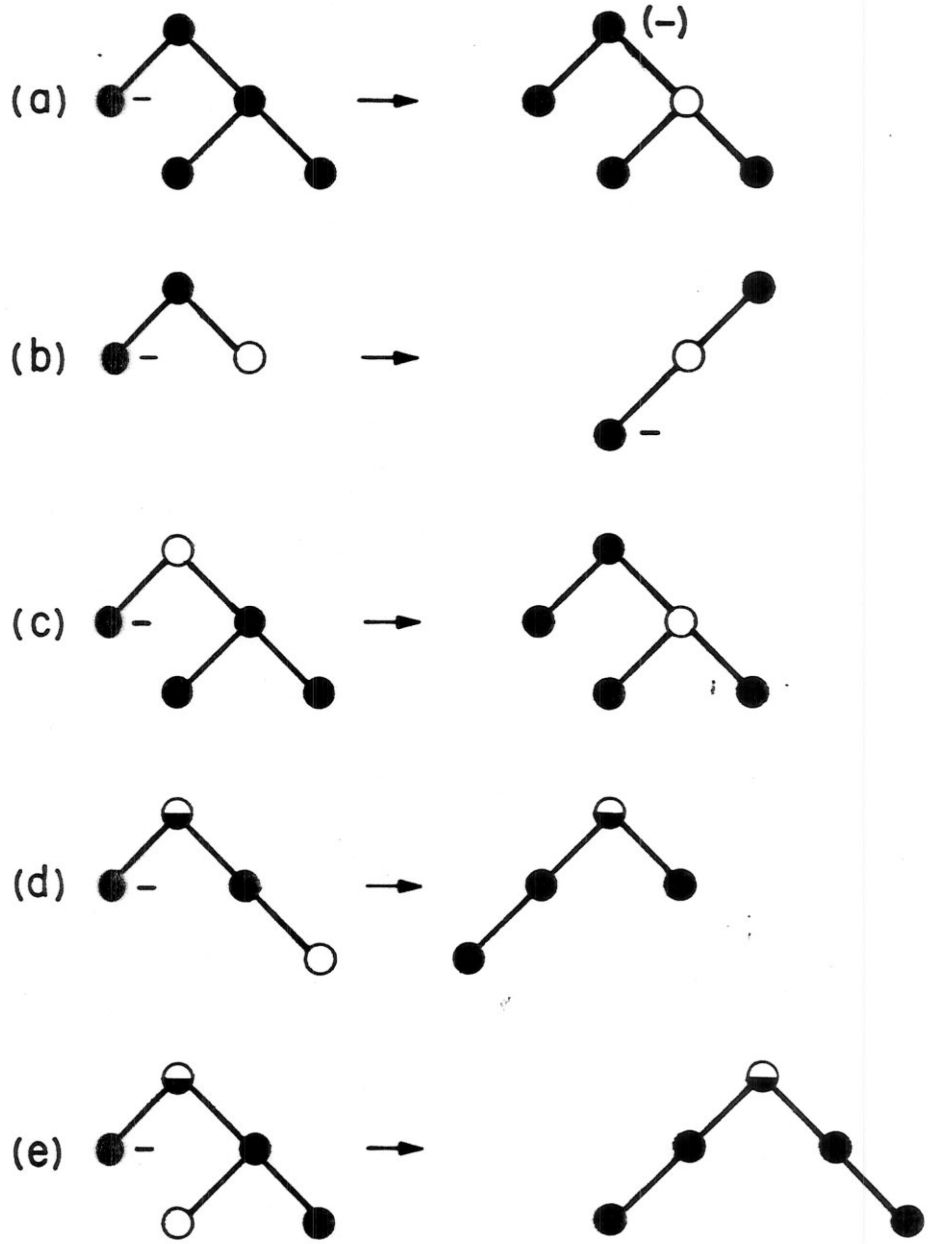

Figure 5. The rebalancing transformations in red-black tree deletion. The two ambiguous (half-solid) nodes in (d) have the same color, as do the two in (e). Minus signs denote short nodes. In (a), the top node after the transformation is short unless it is the root.

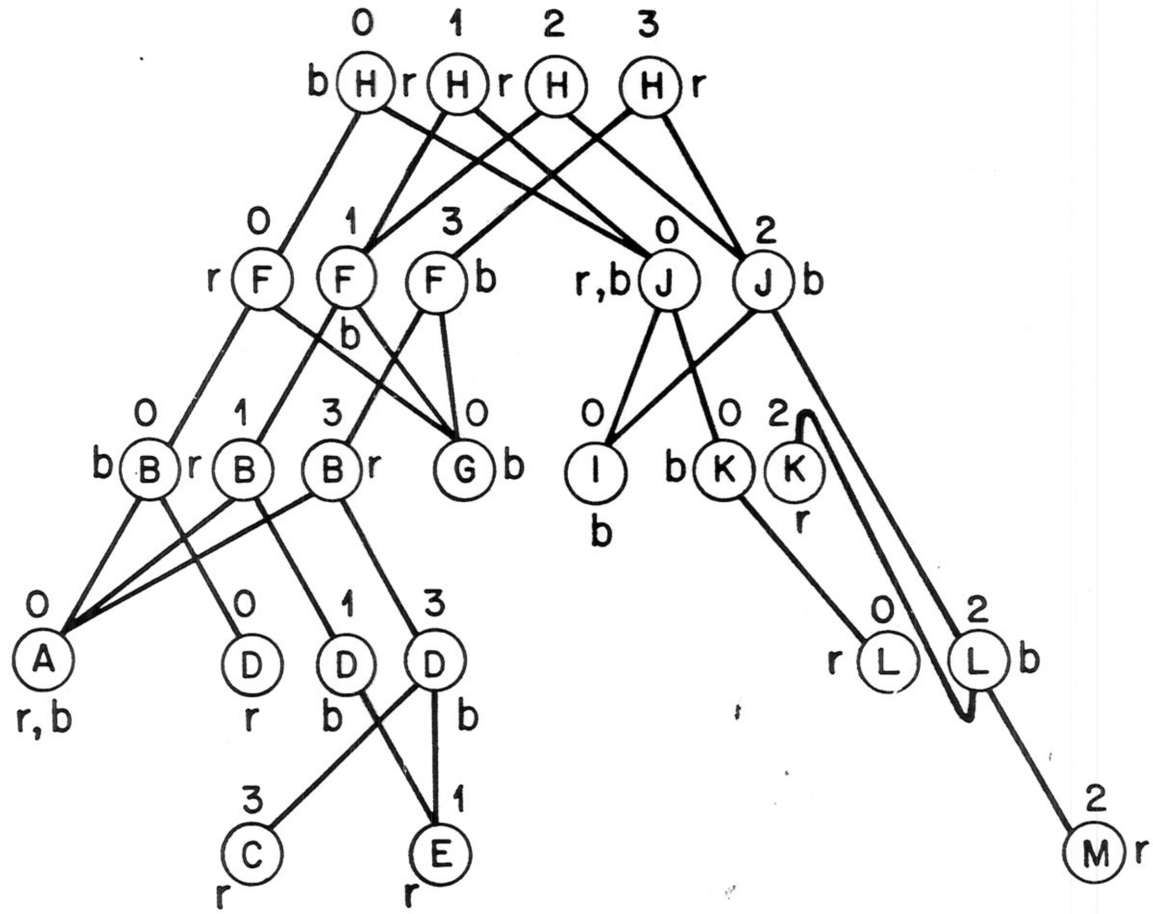

Figure 6. A persistent red-black tree with path copying. The initial tree, existing at time 0, contains A,B,D,F,G,H,I,J,K. Item E is inserted at time 1, item M at time 2, and item C at time 3. The nodes are labeled by their colors, r for red, b for black. The nodes are also labeled by their time of creation. All edges exit the bottoms of nodes and enter the tops.

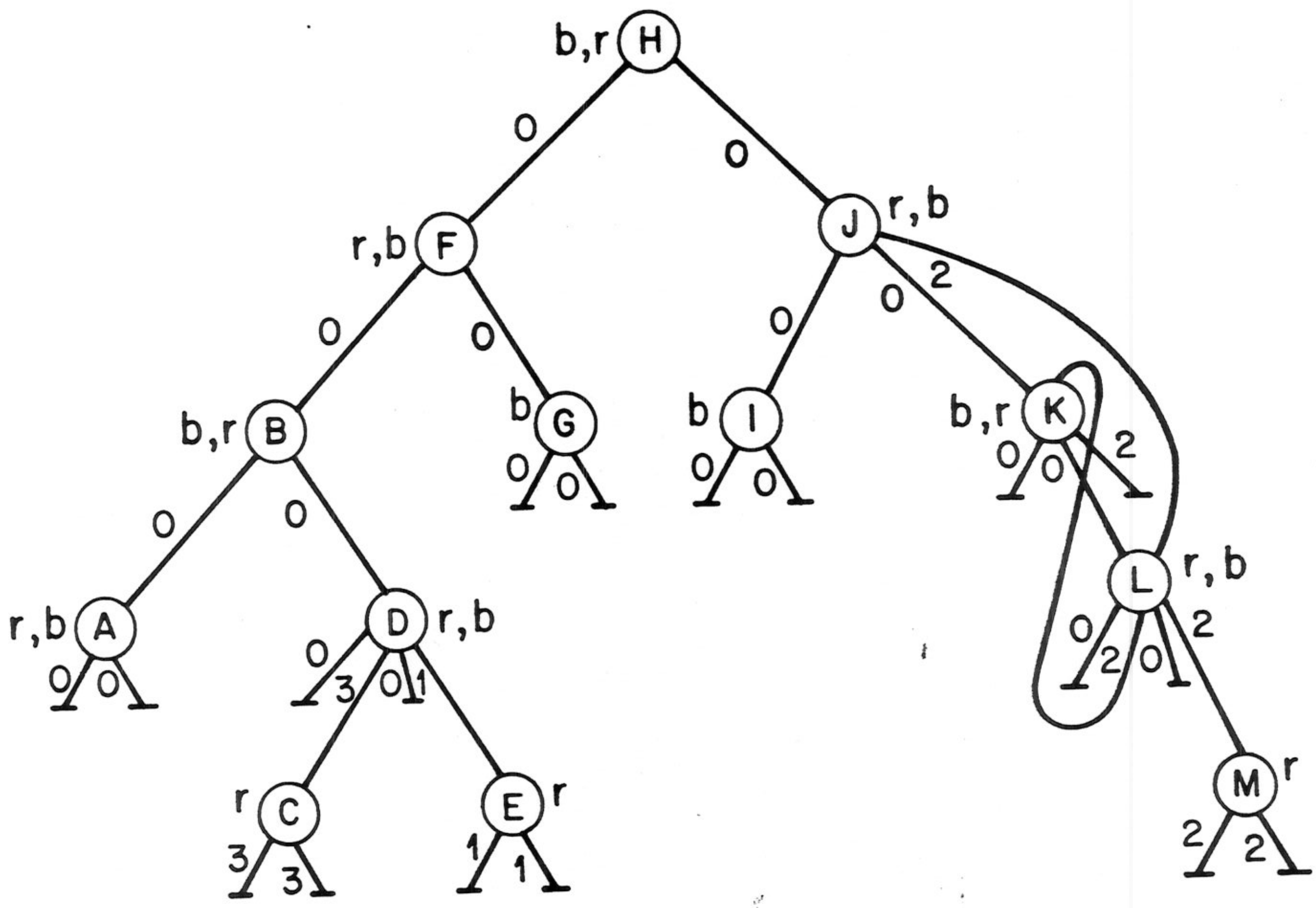

Figure 7. A persistent red-black tree with no node copying. The initial tree and insertions are as in Figure 6. The edges are labeled with their time of creation; the nodes are labeled with their colors. Connections to horizontal lines denote null pointers.

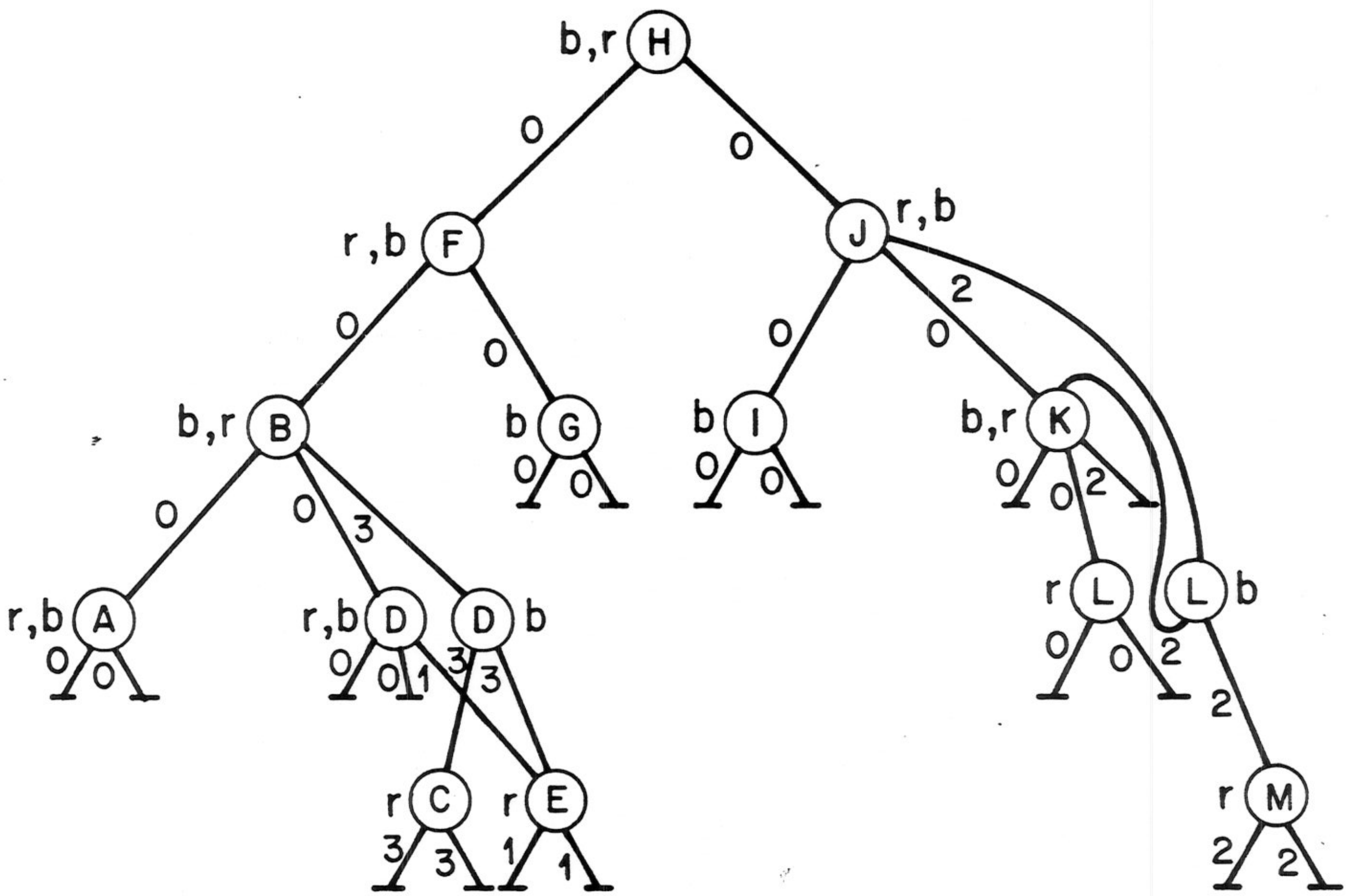

Figure 8. A persistent red-black tree with limited node copying assuming each node can hold one extra pointer. The initial tree and insertions are as in Figure 6. The labeling is as in Figure 7.