JOURNAL OF COMPUTER AND SYSTEM SCIENCES 38, 86–124 (1989)

Making Data Structures Persistent*

JAMES R. DRISCOLL†

Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15218

NEIL SARNAK

IBM T.J. Watson Research Center, Yorktown Heights, New York 10598

DANIEL D. SLEATOR

Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15218

AND

ROBERT E. TARJAN‡

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

Received August 5, 1986

This paper is a study of *persistence* in data structures. Ordinary data structures are *ephemeral* in the sense that a change to the structure destroys the old version, leaving only the new version available for use. In contrast, a persistent structure allows access to any version, old or new, at any time. We develop simple, systematic, and efficient techniques for making linked data structures persistent. We use our techniques to devise persistent forms of binary search trees with logarithmic access, insertion, and deletion times and $O(1)$ space bounds for insertion and deletion. © 1989 Academic Press, Inc.

1. INTRODUCTION

Ordinary data structures are *ephemeral* in the sense that making a change to the structure destroys the old version, leaving only the new one. However, in a variety of areas, such as computational geometry [6, 9, 12, 25, 26, 29, 30], text and file editing [27], and implementation of very high level programming languages [19],

* A condensed, preliminary version of this paper was presented at the Eighteenth Annual ACM Symposium on Theory of Computing, Berkeley, California, May 28–30, 1986.

† Current address: Computer Science Department, University of Colorado, Boulder, Colorado 80309.

‡ Research partially supported by National Science Foundation Grant DCR 8605962.

0022-0000/89 $3.00

Copyright © 1989 by Academic Press, Inc.
All rights of reproduction in any form reserved.

multiple versions of a data structure must be maintained. We shall call a data structure *persistent* if it supports access to multiple versions. The structure is *partially persistent* if all versions can be accessed but only the newest version can be modified, and *fully persistent* if every version can be both accessed and modified.

A number of researchers have devised partially or fully persistent forms of various data structures, including stacks [22], queues [14], search trees [19, 21, 23, 25, 27, 30], and related structures [6, 9, 12]. Most of these results use ad hoc constructions; with the exception of one paper by Overmars [26], discussed in Section 2, there has been no systematic study of persistence. Providing such a study is our purpose in this paper, which is an outgrowth of the second author's Ph. D. thesis [28].

We shall discuss generic techniques for making linked data structures persistent at small cost in time and space efficiency. Since we want to take a general approach, we need to specify exactly what a linked structure is and what kinds of operations are allowed on the structure. Formally, we define a *linked data structure* to be a finite collection of *nodes*, each containing a fixed number of named *fields*. Each field is either an *information field*, able to hold a single piece of information of a specified type, such as a bit, an integer, or a real number, or a *pointer field*, able to hold a pointer to a node or the special value *null* indicating no node. We shall assume that all nodes in a structure are of exactly the same type, i.e., have exactly the same fields; our results easily generalize to allow a fixed number of different node types. Access to a linked structure is provided by a fixed number of named *access pointers* indicating nodes of the structure, called *entry nodes*. We can think of a linked structure as a labeled directed graph whose vertices have constant out-degree. If a node x contains a pointer to a node y, we call y a *successor* of x and x a *predecessor* of y.

As a running example throughout this paper, we shall use the binary search tree. A *binary search tree* is a binary tree [1] containing in its nodes distinct items selected from a totally ordered set, one item per node, with the items arranged in symmetric order: if x is any node, the item in x is greater than all items in the left subtree of x and less than all items in the right subtree of x. A binary search tree can be represented by a linked structure in which each node contains three fields: an *item* (information) field and *left* and *right* (pointer) fields containing pointers to the left and right children of the node. The tree root is the only entry node.

On a general linked data structure we allow two kinds of operations: *access operations* and *update operations*. An access operation computes an *accessed set* consisting of *accessed nodes*. At the beginning of the operation the accessed set is empty. The operation consists of a sequence of *access steps*, each of which adds one node to the accessed set. This node must either be an entry node or be indicated by a pointer in a previously accessed node. The time taken by an access operation is defined to be the number of access steps performed. In an actual data structure the successively accessed nodes would be determined by examining the fields of previously accessed nodes, and the access operation would produce as output some

[1] Our tree terminology is that of the monograph of Tarjan [31].

of the information contained in the accessed nodes, but we shall not be concerned with the details of this process. An example of an access operation is a search for an item in a binary search tree. The accessed set forms a path in the tree that starts at the root and is extended one node at a time until the desired item is found or a null pointer, indicating a missing node, is reached. In the latter case the desired item is not in the tree.

An *update operation* on a general linked structure consists of an intermixed sequence of access and *update steps*. The access steps compute a set of accessed nodes exactly as in an access operation. The update steps change the data structure. An update step consists either of creating a new node, which is added to the accessed set, changing a single field in an accessed node, or changing an access pointer. If a pointer field or an access pointer is changed, the new pointer must indicate a node in the accessed set or be null. A newly created node must have its information fields explicitly initialized; its pointer fields are initially null. As in the case of an access operation, we shall not be concerned with the details of how the steps to be performed are determined. The *total time* taken by an update operation is defined to be the total number of access and update steps; the *update time* is the number of update steps. If a node is not indicated by any pointer in the structure, it disappears from the structure; that is, we do not require explicit deallocation of nodes.

An example of an update operation is an insertion of a new item in a binary search tree. The insertion consists of a search for the item to be inserted, followed by a replacement of the missing node reached by the search with a new node containing the new item. A more complicated update operation is a deletion of an item. The deletion process consists of three parts. First, a search for the item is performed. Second, if the node, say x, containing the item has two children, x is swapped with the node preceding it in symmetric order, found by starting at the left child and following successive right pointers until reaching a node with no right child. Now x is guaranteed to have only one child. The third part of the deletion consists of removing x from the tree and replacing it by its only child, if any. The subtree rooted at this child is unaffected by the deletion. The total time of either an insertion or a deletion is the depth of some tree node plus a constant; the update time is only a constant.

Returning to the case of a general linked structure, let us consider a sequence of intermixed access and update operations on an initially empty structure (one in which all access pointers are null). We shall denote by m the total number of update operations. We index the update operations and the versions of the structure they produce by integers: update i is the ith update in the sequence; version 0 is the initial (empty) version, and version i is the version produced by update i. We are generally interested in performing the operations on-line; that is, each successive operation must be completed before the next one is known.

We can characterize ephemeral and persistent data structures based on the allowed kinds of operation sequences. An ephemeral structure supports only sequences in which each successive operation applies to the most recent version. A

partially persistent structure supports only sequences in which each update applies to the most recent version (update i applies to version $i-1$), but accesses can apply to any previously existing version (whose index must be specified). A fully persistent structure supports any sequence in which each operation applies to any previously existing version. The result of the update is an entirely new version, distinct from all others. For any of these kinds of structure, we shall call the operation being performed the *current operation* and the version to which it applies the *current version.* The current version is *not* necessarily the same as the newest version (except in the case of an ephemeral structure) and thus in general must be specified as a parameter of the current operation. In a fully persistent structure, if update operation i applies to version $j<i$, the result of the update is version i; version j is not changed by the update. We denote by n the number of nodes in the current version.

The problem we wish to address is as follows. Suppose we are given an ephemeral structure; that is, we are given implementations of the various kinds of operations allowed on the structure. We want to make the structure persistent; that is, to allow the operations to occur in the more general kinds of sequences described above. In an ephemeral structure only one version exists at any time. Making the structure persistent requires building a data structure representing all versions simultaneously, thereby permitting access and possibly update operations to occur in any version at any time. This data structure will consist of a linked structure (or possibly something more general) with each version of the ephemeral structure embedded in it, so that each access or update step in a version of the ephemeral structure can be simulated (ideally in $O(1)$ time) in the corresponding part of the persistent structure.

The main results of this paper are in Sections 2–5. In Section 2 we discuss how to build partially persistent structures, which support access but not update operations in old versions. We show that if an ephemeral structure has nodes of constant bounded in-degree, then the structure can be made partially persistent at an amortized [2] space cost of $O(1)$ per update step and a constant factor in the amortized time per operation. The construction is quite simple. Using more powerful techniques we show in Section 3 how to make an ephemeral structure with nodes of constant bounded in-degree fully persistent. As in Section 2, the amortized space cost is $O(1)$ per update step and the amortized time cost is a constant factor.

In Sections 4 and 5 we focus on the problem of making balanced search trees fully persistent. In Section 4 we show how the result of Section 2 provides a simple way to build a partially persistent balanced search tree with a worst-case time per operation of $O(\log n)$ and an amortized space cost of $O(1)$ per insertion or deletion. We also combine the result of Section 3 with a delayed updating technique of Tsakalidis [35, 36] to obtain a fully persistent form of balanced search tree with the same time and space bounds as in the partially persistent case, although the

[2] By *amortized cost* we mean the cost of an operation averaged over a worst-case sequence of operations. See the survey paper of Tarjan [33].

insertion and deletion time is $O(\log n)$ in the amortized case rather than in the worst case. In Section 5 we use another technique to make the time and space bounds for insertion and deletion worst-case instead of amortized.

Section 6 is concerned with applications, extensions, and open problems. The partially persistent balanced search trees developed in Section 4 have a variety of uses in geometric retrival problems, a subject more fully discussed in a companion paper [28]. The fully persistent balanced search trees developed in Sections 4 and 5 can be used in the implementation of very high level programming languages, as can the fully persistent deques (double-ended queues) obtainable from the results of Section 3. The construction of Section 2 can be modified so that it is write-once. This implies that any data structure built using augmented LISP (in which *replaca* and *replacd* are allowed) can be simulated in linear time in pure LISP (in which only *cons*, *car*, and *cdr* are allowed), provided that each node in the structure has constant bounded in-degree.

2. Partial Persistence

Our goal in this section is to devise a general technique to make an ephemeral linked data structure partially persistent. Recall that partial persistence means that each update operation applies to the newest version. We shall propose two methods. The first and simpler is the *fat node method*, which applies to any ephemeral linked structure and makes it persistent at a worst-case space cost of $O(1)$ per update step and a worst-case time cost of $O(\log m)$ per access or update step. More complicated is the *node-copying method*, which applies to an ephemeral linked structure of constant bounded in-degree and has an amortized time and space cost of $O(1)$ per update step and a worst-case time cost of $O(1)$ per access step.

2.1. *Known Methods*

We begin by reviewing the results of Overmars [26], who studied three simple but general ways to obtain partial persistence. One method is to store every version explicitly, copying the entire ephemeral structure after each update operation. This costs $\Omega(n)$ time and space per update. An alternative method is to store no versions but instead to store the entire sequence of update operations, rebuilding the current version from scratch each time an access is performed. If storing an update operation takes $O(1)$ space, this method uses only $O(m)$ space, but accessing version i takes $\Omega(i)$ time even if a single update operation takes $O(1)$ time. A hybrid method is to store the entire sequence of update operations and in addition every kth version, for some suitable chosen value of k. Accessing version i requires rebuilding it from version $k\lfloor i/k \rfloor$ by performing the appropriate sequence of update operations. This method has a time–space trade-off that depends on k and on the running times of the ephemeral access and update operations. Unfortunately any

choice of k causes a blowup in either the storage space or the access time by a factor of $\sqrt{m}$, if one makes reasonable assumptions about the efficiency of the ephemeral operations.

The third approach of Overmars is to use the dynamization techniques of Bentley and Saxe [2], which apply to so-called "decomposable" searching problems. Given an ephemeral data structure representing a set of items, on which the only update operation is insertion, the conversion to a persistent structure causes both the access time and the space usage to blow up by a logarithmic factor, again if one makes reasonable assumptions about the efficiency of the ephemeral operations. If deletions are allowed the blowup is much greater.

We seek more efficient techniques. Ideally we would like the storage space used by the persistent structure to be $O(1)$ per update step and the time per operation to increase by only a constant factor over the time in the ephemeral structure. One reason the results of Overmars are so poor is that he assumes very little about the underlying ephemeral structure. By restricting our attention to linked structures and focusing on the details of the update steps, we are able to obtain much better results.

2.2. *The Fat Node Method*

Our first idea is to record all changes made to node fields in the nodes themselves, without erasing old values of the fields. This requires that we allow nodes to become arbitrarily "fat," i.e., to hold an arbitrary number of values of each field. To be more precise, each fat node will contain the same information and pointer fields as an ephemeral node (holding original field values), along with space for an arbitrary number of extra field values. Each extra field value has an associated field name and a version stamp. The version stamp indicates the version in which the named field was changed to have the specified value. In addition, each fat node has its own version stamp, indicating the version in which the node was created.

We simulate ephemeral update steps on the fat node structure as follows. Consider update operation i. When an ephemeral update step creates a new node, we create a corresponding new fat node, with version stamp i, containing the appropriate original values of the information and pointer fields. When an ephemeral update step changes a field value in a node, we add the corresponding new value to the corresponding fat node, along with the name of the field being changed and a version stamp of i. For each field in a node, we store only one value per version; when storing a field value, if there is already a value of the same field with the same version stamp we overwrite the old value. Original field values are regarded as having the version stamp of the node containing them. (Our assumptions about the workings of linked data structures do not preclude the possibility of a single update operation on an ephemeral structure changing the same field in a node more than once. If this is not allowed, there is no need to test in the persistent structure for two field values with the same version stamp.)

The resulting persistent structure has all versions of the ephemeral structure

embedded in it. We navigate through the persistent structure as follows. When an ephemeral access step applied to version i accesses field f of a node, we access the value in the corresponding fat node whose field name is f, choosing among several such values the one with maximum version stamp no greater than i.

We also need an auxiliary data structure to store the access pointers of the various versions. This structure consists of an array of pointers for each access pointer name. After update operation i, we store the current values of the access pointers in the ith positions of these access arrays. With this structure, initiating access into any version takes $O(1)$ time.

Remark. The only purpose of nodes having version stamps is to make sure that each node only contains one value per field name per version. These stamps are not needed if there is some other mechanism for keeping track during update operation i of the newest field values of nodes created during update i. In order to navigate through the structure it suffices to regard each original field value in a node as having a version stamp of zero.

Figure 1 shows a persistent binary search tree constructed using the fat node method. The update operations are insertions and deletions, performed as described in Section 1. Version stamps on nodes are easy to dispense with in this application. Since the original pointers in each node are known to be null, they too can be omitted. (The version f pointer field of a node is taken to be null if no field f pointer stored in the node has a version stamp less than or equal to i.)

The fat node method applies to any linked date structure and uses only $O(1)$ space per ephemeral update step in the worst case, but it has two drawbacks. First, the fat nodes must be represented by linked collections of fixed-size nodes. This poses no fundamental difficulty but complicates the implementation. Second, choos-

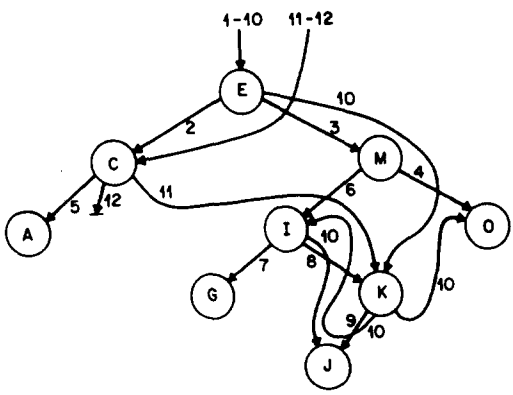

FIG. 1. A partially persistent search tree built using the fat node method, for the sequence of update operations consisting of insertions of E, C, M, O, A, I, G, K, J, followed by deletion of M, E, and A. The "extra" pointers are labeled with their version stamps. Left pointers leave the left sides of nodes and right pointers leave the right sides. The version stamps and original null pointers of nodes are omitted, since they are unnecessary.

ing which pointer in a fat node to follow when simulating an access step takes more than constant time. If the values of a field within a fat node are ordered by version stamp and stored in a binary search tree, simulating an ephemeral access or update step takes $O(\log m)$ time. This means that there is a logarithmic factor blow-up in the times of access and update operations over their times in the ephemeral structure.

Remark. Although fat nodes must in general be able to hold arbitrarily many pointers, this is not true in the binary tree application discussed above if insertion is the only update operation. In this case each "fat" node only needs to hold one item and two pointers, one left pointer and one right pointer, each with a version stamp.

2.3. *The Node-Copying Method*

We eliminate the drawbacks of fat nodes with our second idea, *node copying*. We allow nodes in the persistent structure to hold only a fixed number of field values. When we run out of space in a node, we create a new copy of the node, containing only the newest value of each field. We must also store pointers to the new copy in all predecessors of the copied node in the newest version. If there is no space in a predecessor for such a pointer, the predecessor, too, must be copied. Nevertheless, if we assume that the underlying ephemeral structure has nodes of constant bounded in-degree and we allow sufficient extra space in each node of the persistent structure, then we can derive an $O(1)$ amortized bound on the number of nodes copied and the time required per update step.

2.3.1. *The Data Structure*

In developing the details of this idea, we shall use the following terminology. We call a node of the underlying ephemeral structure an *ephemeral node* and a node of the persistent structure a *persistent node*. If x is an ephemeral node existing in version i of the ephemeral structure, then *version* i of x is x together with all its field values in version i of the structure. That is, we think of an ephemeral node as going through several versions as its fields are changed. We shall denote by $\bar{x}$ a persistent node corresponding to an ephemeral node x.

In the node-copying method as we shall describe it, each persistent node contains only one version of each information field but may contain multiple versions of pointer fields. (Other variants of the method, allowing multiple versions of information fields, are easily formulated.) Let d be the number of pointer fields in an ephemeral node and let p be the maximum number of predecessors of an ephemeral node in any one version. We assume that p is a constant. Each persistent node will contain $d+p+e+1$ pointer fields, where e is a sufficiently large constant, to be chosen later. Of these fields, d are the same as the pointer fields of an ephemeral node and contain *original pointers*, p are for *predecessor pointers*, e for *extra pointers*, and one is for a *copy pointer*. Each persistent node has the same information fields as an ephemeral node, and it also has a version stamp for the node itself and

a field name and a version stamp for each extra pointer. Original pointers in a node are regarded as having version stamps equal to that of the node.

The correspondence between the ephemeral structure and the persistent structure is as follows. Each ephemeral node corresponds to a set of persistent nodes, called a *family*. The members of the family form a singly-linked list, linked by the copy pointers, in increasing order by version stamp, i.e., the newest member is last on the list. We call this last member *live* and the other members of the family *dead*. Each version of the ephemeral node corresponds to one member of the family, although several versions of the ephemeral node may correspond to the same member of the family. The live nodes and their newest field values represent the newest version of the ephemeral structure. We call a pointer in the persistent structure representing a pointer in the newest version of the ephemeral structure a *live pointer*. To facilitate node copying, each live pointer in the persistent structure has a corresponding *inverse pointer*; i.e., if live node $\bar{x}$ contains a live pointer to live node $\bar{y}$, then $\bar{y}$ contains a pointer to $\bar{x}$ stored in one of its p predecessor fields.

As in the fat node method, we use access arrays, one per access pointer name, to store the various versions of the access pointers. Thus we can access any entry node in any version in $O(1)$ time. Navigation through the persistent structure is exactly the same as in the fat node method: to simulate an ephemeral access step that applies to version i and follows the pointer in field f of an ephemeral node x, we follow the pointer with field name f in the corresponding persistent node, selecting among several such pointers the one with maximum version stamp no greater than i. Simulating an ephemeral access step in the persistent structure takes $O(1)$ time.

2.3.2. *Update Operations*

When simulating an ephemeral update operation, we maintain a set S of nodes that have been copied. Consider update operation i. We begin the simulation of this operation by initializing S to be empty. We simulate ephemeral access steps as described above. When an ephemeral update step creates a new node, we create a corresponding new persistent node with a version stamp of i and all original pointers null. When an ephemeral update step changes an information field in an ephemeral node x, we inspect the corresponding persistent node $\bar{x}$. If $\bar{x}$ has version stamp i, we merely change the appropriate field in $\bar{x}$. If $\bar{x}$ has version stamp less than i, but has a copy $c(\bar{x})$ (which must have version stamp i), we change the appropriate field in $c(\bar{x})$. If $\bar{x}$ has version stamp less than i but has no copy, we create a copy $c(\bar{x})$ of $\bar{x}$ with version stamp i, make the copy pointer of $\bar{x}$ point to it, and fill it with the most recent values of the information fields of x, which, excluding the new value of the changed field, can be obtained from $\bar{x}$. We also add to $c(\bar{x})$ pointers corresponding to the most recent values of the pointer fields of x. This requires updating inverse pointers and is done as follows. Suppose that $\bar{x}$ contains a pointer to a node $\bar{y}$ as the most recent version of field f. We store in original pointer field f of node $c(\bar{x})$ a pointer to $\bar{y}$, or to the copy $c(\bar{y})$ of $\bar{y}$ if $\bar{y}$ has been copied. We erase the pointer to $\bar{x}$ in one of the predecessor fields of $\bar{y}$, and we store

a pointer to $c(\bar{x})$ in the predecessor field of $\bar{y}$ or $c(\bar{y})$ as appropriate. Once $c(\bar{x})$ has all its original pointer fields filled, we add $\bar{x}$ to the set S of copied nodes.

Simulating an ephemeral update step that changes a pointer field is much like simulating a step that changes an information field. If x is the ephemeral node in which the change takes place, we inspect the corresponding persistent node $\bar{x}$. If $\bar{x}$ has version stamp i, we merely change the appropriate original pointer field in $\bar{x}$. If $\bar{x}$ has version stamp less than i but has a copy $c(\bar{x})$, we change the appropriate original pointer field in $c(\bar{x})$. If $\bar{x}$ has a version stamp less than i but has no copy, we check whether $\bar{x}$ has space for an extra pointer. If so, we store the appropriate new pointer in $\bar{x}$, along with the appropriate field name and a version stamp of i. If not, we create a new copy $c(\bar{x})$ of $\bar{x}$, fill it as described above, and add $\bar{x}$ to S. During the simulation, whenever we install a pointer in a persistent node $\bar{x}$, we make sure it points to a live node. More precisely, if the pointer indicates $\bar{y}$ but $\bar{y}$ has a copy $c(\bar{y})$, we place in $\bar{x}$ a pointer to $c(\bar{y})$ instead of $\bar{y}$. Also, whenever installing a new pointer, we update inverse pointers appropriately.

After simulating all the steps of the update operation, we postprocess the set S to make live pointers point to live nodes. This postprocessing consists of repeating the following step until S is empty:

Update pointers. Remove any node $\bar{y}$ from S. For each node $\bar{x}$ indicated by a predecessor pointer in $\bar{y}$, find in $\bar{x}$ the live pointer to $\bar{y}$. If this pointer has version stamp equal to i, replace it by a pointer to $c(\bar{y})$. If the pointer has version stamp less than i, add a version i pointer from $\bar{x}$ to $c(\bar{y})$, copying $\bar{x}$ as described above if there is no space for this pointer in $\bar{x}$. If $\bar{x}$ is copied, add $\bar{x}$ to S. When installing pointers, update inverse pointers appropriately.

To complete the update operation, we store the newest values of the access pointers in the ith positions of the access arrays. We are then ready to begin the next operation.

2.3.3. *An Example*

As an example of the use of the node-copying method, let us apply it to binary search trees. In this application a major simplification is possible: we can dispense with inverse pointers in the persistent structure, because in the ephemeral structure there is only one access path to any node, and accessing a node requires accessing its predecessor. In general we can avoid the use of inverse pointers if in the ephemeral structure a node is only accessed after *all* its predecessors are accessed. In the binary tree case we can also dispense with the copy pointers and the version stamps of nodes, since node copying proceeds in a very controlled way, back through ancestors of the node where a change occurs. Figure 2 illustrates a persistent binary search tree built using the node-copying method, without inverse and copy pointers and version stamps of nodes. The update operations are insertions and deletions. Each persistent node has one extra pointer field, which, as we shall see below, suffices to guarantee an $O(m)$ space bound on the size of the persistent structure, since in this case $p = 1$.

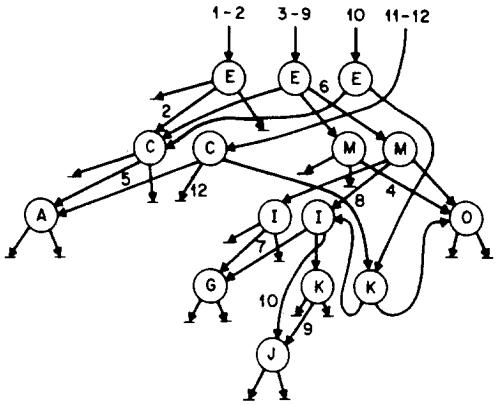

FIG. 2. A partially persistent search tree built using the node-copying method, for the same sequence of update operations as in Fig. 1. The version stamps of nodes and the copy pointers are omitted, since they are unnecessary.

2.3.4. *Space and Time Analysis*

It remains for us to analyze the space needed for the persistent structure and the time needed for update steps. We shall derive an amortized $O(1)$ bound on the space and time per update step using the "potential" technique [33]. To obtain this bound, it suffices to choose e to be any integer constant such that $e \geqslant p$.

Recall that a persistent node is live if it has not been copied and dead otherwise. All updates to fields are in live nodes, except for erasures of inverse pointers, which can occur in dead nodes. We define the *potential* of the persistent structure to be $e/(e-p+1)$ times the number of live nodes it contains minus $1/(e-p+1)$ times the number of unused extra pointer fields in live nodes. Observe that the potential of the initial (empty) structure is zero and that the potential is always nonnegative, since any node has at most e unused extra pointer fields. We define the *amortized space cost* of an update operation to be the number of nodes it creates plus the net increase in potential it causes. With this definition, the total number of nodes created by a sequence of update operations equals the total amortized space cost minus the net increase in potential over the sequence. Since the initial potential is zero and the final potential is nonnegative, the net potential increase over any sequence is nonnegative, which implies that the total amortized space cost is an upper bound on the total number of nodes created.

The key part of the analysis is to show that the amortized space cost of an update operation is linear in the number of update steps. Consider an update operation that performs u update steps. Each update step has an amortized space cost of $O(1)$ and adds at most one node to S. Consider the effect of the postprocessing that empties S. Let $t \leqslant u$ be the number of nodes in S at the beginning of the postprocessing and let k be the number of nodes copied during the postprocessing. Each time a node is copied during postprocessing, a live node with potential

$e/(e-p+1)$ becomes dead and a new live node with potential zero is created, for a net potential drop of $e/(e-p+1)$. In addition, a node is added to S. The total number of nodes added to S before or during the postprocessing is thus $t+k$. When a node is removed from S and pointers to its copy are installed, there is a potential increase of $1/(e-p+1)$ for each pointer stored in an extra pointer field. There are at most p such possible storages per node removed from S, for a total of $p(t+k)$, but at least k of these do not in fact occur, since the pointers are stored in original fields of newly copied nodes instead. Thus the net potential increase during the postprocessing is at most $(p(t+k)-k)/(e-p+1)$ caused by storage of pointers in extra field minus $ke/(e-p+1)$ caused by live nodes becoming dead. The amortized space cost of the postprocessing is thus at most

$$\begin{aligned} &k+(p(t+k)-k)/(e-p+1)-ke/(e-p+1) \\ &\quad = k+pt/(e-p+1)+k(p-1-e)/(e-p+1) \\ &\quad = pt/(e-p+1) = O(t). \end{aligned}$$

Hence the amortized space cost of the entire update operation is $O(t)=O(u)$, i.e., $O(1)$ per update step. The same analysis shows that the amortized time per update step is also $O(1)$.

3. Full Persistence

In this section we address the harder problem of making an ephemeral structure fully persistent. We shall obtain results analogous to those of Section 2. Namely, with the fat node approach we can make an ephemeral linked structure fully persistent at a worst-case space cost of $O(1)$ per update step and an $O(\log m)$ worst-case time cost per access or update step. With a variant of the node-copying method called *node splitting*, we can make an ephemeral linked structure of constant bounded in-degree fully persistent at an $O(1)$ amortized time and space cost per update step and an $O(1)$ worst-case time cost per access step.

3.1. *The Version Tree and the Version List*

The first problem we encounter with full persistence is that whereas the various versions of a partially persistent structure have a natural linear ordering, the versions of a fully persistent structure are only partially ordered. This partial ordering is defined by a rooted *version tree*, whose nodes are the versions (0 through m), with version i the parent of version j if version j is obtained by updating version i. Version 0 is the root of the version tree. The sequence of updates giving rise to version i corresponds to the path in the version tree from the root to i.

Unfortunately, the lack of a linear ordering on versions makes navigation through a representation of a fully persistent structure problematic. To eliminate

this difficulty, we impose a total ordering on the versions consistent with the partial ordering defined by the version tree. We represent this total ordering by a list of the versions in the appropriate order. We call this the *version* list. When a new version, say i, is created, we insert i in the version list immediately after its parent (in the version tree). The resulting list defines a preorder on the version tree, as can easily be proved by induction. This implies that the version list has the following crucial property: for any version i, the descendants of i in the version tree occur consecutively in the version list, starting with i (see Fig. 3). We shall refer to the direction toward the front of the version list (from a given item) as *leftward* and the direction toward the back of the list as *rightward.*

In addition to performing insertions in the version list, we need to be able to determine, given two versions i and j, whether i precedes or follows j in the version list. This *list order problem* has been addressed by Dietz [10], by Tsakalidis [3, 4], and most recently by Dietz and Sleator [11]. Dietz and Sleator [11] proposed a list representation supporting order queries in $O(1)$ worst-case time, with an $O(1)$ amortized time bound for insertion. They also proposed a more complicated representation in which both the query time and the insertion time are $O(1)$ in the worst case.

3.2. *The Fat Node Method*

Having dealt in a preliminary way with the navigation issue, let us investigate how to use fat nodes to make a linked structure fully persistent. We use the same node organization as in Section 2; namely, each fat node contains the same fields as

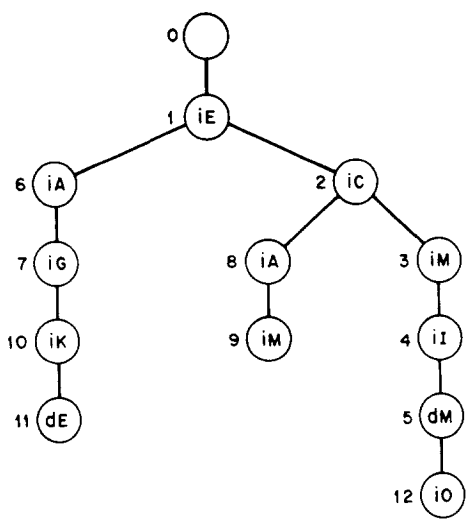

FIG. 3. A version tree. Each node represents an update operation; an "i" or "d" indicates an insertion or deletion of the specified item. The nodes are labeled with the indices of the corresponding operations. The version list is 1, 6, 7, 10, 11, 2, 8, 9, 3, 4, 5, 12.

an ephemeral node (for storing original field values), as well as space for an arbitrary number of extra field values, each with a field name and a version stamp, and space for a version stamp for the node itself. Each original field in a fat node is regarded as having the same version stamp as that of the node itself.

Navigation through the persistent structure is the same as in the partially persistent case, except that versions are compared with respect to their positions in the version list, rather than with respect to their numeric values. That is, to find the value corresponding to that of field f in version i of ephemeral node x, we find in the fat node $\bar{x}$ corresponding to x the value of field f whose version stamp is rightmost in the version list but not to the right of i.

Updating differs slightly from what happens in the partially persistent case, because the insertion of new versions in the middle of the version list makes it, in general, necessary to store two updated field values per update step, rather than one. We begin update operation i by adding i to the version list as described above. When an ephemeral update step creates a new ephemeral node, we create a corresponding new fat node with version stamp i, filling in its original fields appropriately. Suppose an ephemeral update step changes field f of ephemeral node x. Let $i+$ denote the version after i in the version list, if such a version exists. To simulate the update step, we locate in the fat node $\bar{x}$ corresponding to x values v_1 and v_2 of field f such that v_1 has rightmost version stamp not right of i and v_2 has leftmost version stamp right of i (in the version list). Let i_1 and i_2 be the version stamps of v_1 and v_2, respectively. There are two cases:

(i) If $i_1 = i$, we replace v_1 by the appropriate new value of field f. If in addition i is the version stamp of node $\bar{x}$, v_1 is a null pointer, and $i+$ exists, we store in $\bar{x}$ a null pointer with field name f and version stamp $i+$, unless $\bar{x}$ already contains such a null pointer.

(ii) If $i_1 < i$, we add the appropriate new value of field f to node $\bar{x}$, with a field name of f and a version stamp of i. If in addition $i_1 < i$ and $i+ < i_2$ (or $i+$ exists but i_2 does not exist), we add to $\bar{x}$ a new copy of v_1 with a field name of f and a version stamp of $i+$. This guarantees that the new value of field f will be used only in version i, and value v_1 will still be used in versions from $i+$ up to but not including i_2 in the version list.

At the end of the update operation we store the current values of the access pointers in the ith positions of the access arrays. (The current values of the access pointers are those of the parent of i in the version tree, as modified during the update operation.)

Let v be a value of a field f having version stamp i in fat node $\bar{x}$. We define the *valid interval* of v to be the interval of versions in the version list from i up to but not including the next version stamp of a value of field f in $\bar{x}$, or up to and including the last version in the version list if there is no such next version stamp. The valid intervals of the values of a field f in a fat node $\bar{x}$ partition the interval of versions from the version stamp of $\bar{x}$ to the last version. The correctness of the fat

node method can be easily established using a proof by induction on the number of update steps to show that the proper correspondence between the ephemeral structure and the persistent structure is maintained.

Figure 4 shows a fully persistent binary search tree built using the fat node method. In this application, we can as in the partially persistent case omit the version stamps of nodes and the original null pointers of nodes.

The fat node method provides full persistence with the same asymptotic efficiency as it provides partial persistence; namely, the worst-case space cost per update step is $O(1)$ and the worst-case time cost per access and update step is $O(\log m)$, provided that each set of field values in a fat node is stored in a search tree, ordered by version stamp. A more accurate bound is O $(\log h)$ time per access or update step, where h is the maximum number of changes made to an ephemeral node. As in the case of partial persistence, the fat node method applies even if the in-degree of ephemeral nodes is not bounded by a constant.

3.3. *The Node-Splitting Method*

We improve over the fat node method by using a variant of node copying. We shall call the variant *node splitting*, since it resembles the node splitting used to perform insertions in B-trees [1]. The major difference between node splitting and node copying is that in the former, when a node overflows, a new copy is created and roughly half the extra pointers are moved from the old copy to the new one, thereby leaving space in both the old and new copies for later updates. The efficient implementation of node splitting is more complicated than that of node copying, primarily because the correct maintenance of inverse pointers requires care. Also, access pointers for old versions must sometimes be changed. This requires the maintenance of inverse pointers for access pointers as well as for node-to-node pointers.

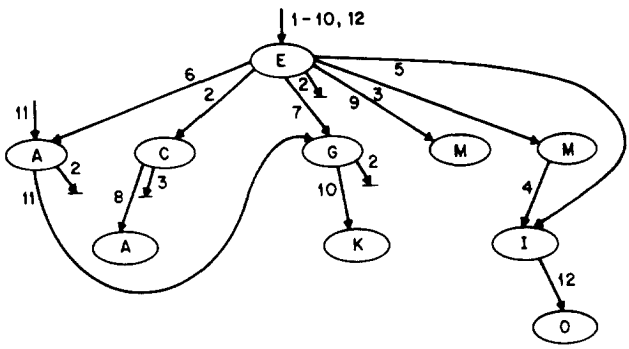

FIG. 4. A fully persistent search tree built using the fat node method, for the update sequence in Fig. 3. The version stamps and original null pointers of nodes are omitted as unnecessary.

3.3.1. *The Augmented Ephemeral Structure*

We begin our development of node splitting by discussing the addition of inverse pointers to an ephemeral structure. This addition will produce what we shall call the *augmented ephemeral structure*. Let p be a constant upper bound on the number of pointers to an ephemeral node in any one version, including access pointers. (In Section 2, access pointers were not counted in p.) To each ephemeral node we add p *inverse fields*, with new, distinct field names, to hold inverse pointers. Each inverse field is capable of holding either a pointer to a node or an access pointer name. (This allows representation of inverses for access pointers.) During each ephemeral update operation, we update the inverse pointers as follows. Consider an update step that changes pointer field f in node x to point to node y. If this field was not null before the change but contained a pointer to a node, say z, we find in z an inverse field containing a pointer to x and make it null. Whether or not field f in node x was previously null, we find in y a null inverse pointer field and store in it a pointer to x. Then we change field f in x to point to y. Thus each step changing a pointer in the original ephemeral structure becomes three (or possibly fewer) update steps in the augmented structure.

We must also update inverse pointers when an access pointer changes. Suppose an access pointer named a is changed to point to a node x. If this pointer was previously not null but contained a pointer to a node, say z, we find in z an inverse field containing name a and make it null. We find in node x a null inverse field and store in it the name a. Then we change access pointer a to point to x.

There is one more detail of the method. At the end of each update operation, we examine each access pointer. If an access pointer named a contains a pointer to a node named x, we find in x the inverse field containing name a and write the name a on top of itself. The purpose of doing this is to ensure that the inverse of every access pointer is explicitly set in every version, which becomes important when we make the structure persistent.

The augmented ephemeral structure has the following important *symmetry properties*:

(1) If a node x contains a pointer to a node y then y contains a pointer to x.

(2) An access pointer named a points to a node x if and only if node x contains name a (in an inverse field).

3.3.2. *The Fat Node Method Revisited*

Suppose now that we apply the fat node method to make this augmented ephemeral structure fully persistent. In the fat node structure, the symmetry properties become the following:

(3) If a node $\bar{x}$ contains a pointer to a node $\bar{y}$ such that the valid interval of the pointer includes a version i, then $\bar{y}$ contains a pointer to $\bar{x}$ such that the valid interval of this pointer includes i also.

(4) The access array representing an access pointer named a contains a

pointer to a node $\bar{x}$ in position i if and only if node $\bar{x}$ contains name a with version stamp i. (This property follows from the explicit setting of access pointer inverses after every update operation in the augmented ephemeral structure.)

Figure 5 illustrates a fully persistent search tree with inverse pointers built using the fat node method. In this application, it is useful to make all pointers in a deleted ephemeral node null when the deletion takes place. This guarantess that every node in the nonaugmented ephemeral structure has at most one incoming pointer. (Otherwise it could have additional incoming pointers from deleted nodes.)

One more observation about the fat node structure is useful. Suppose that a fat node $\bar{x}$ contains a value v having field name f and valid interval I. Let i be any version in I other than the first (the version stamp of v). If we add to $\bar{x}$ a (redundant) copy of v having field name f and version stamp i, we affect neither the navigational correspondence between the ephemeral structure and the persistent structure nor the symmetry properties; the two copies of v have valid intervals that partition I.

3.3.3. *The Split Node Data Structure*

Having discussed inverse pointers, we are ready to focus on the node-splitting method itself. The effect of node splitting is to divide each fat node into one or more constant-size nodes, possibly introducing redundant field values of the kind described above. Let d be the number of pointer fields in an unaugmented ephemeral node, and let $k = d + p$ be the number of pointer fields in an augmented ephemeral node. Each persistent node will contain a version stamp for itself, the

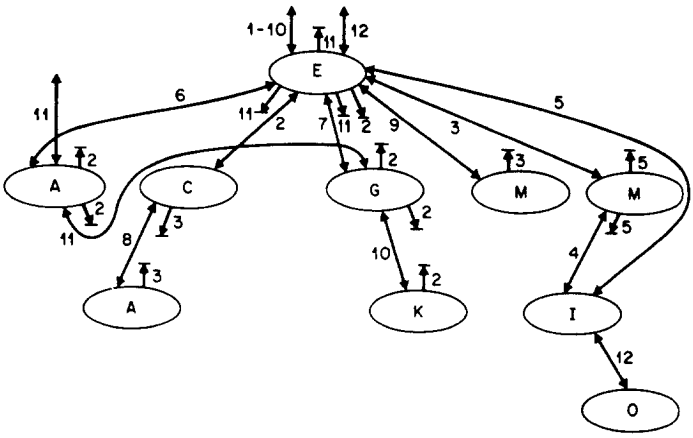

FIG. 5. A fully persistent search tree with inverse (parent) pointers built using the fat node method, for the update sequence in Fig. 3. Parent pointers leave the tops of nodes. Each double-headed arrow denotes two pointers, one the inverse of the other, with the same version stamp. The version stamps and original null pointers of nodes are omitted as unnecessary.

same information fields as an ephemeral node, and $k + 2e + 1$ pointer fields, where e is an integer satisfying $e \geqslant k$. Of these pointer fields, k are the same as the pointer fields in an augmented ephemeral node and contain *original pointers*, $2e$ are for *extra pointers*, and one is for a *copy pointer*. The original pointers are regarded as having the same version stamp as the persistent node itself. Each extra pointer has an associated field name and version stamp. Every pointer field except the copy field is able to hold an access pointer name instead of a pointer to a node and is thus capable of representing the inverse of an access pointer.

The persistent nodes are grouped into *families*, each family consisting of a singly linked list of nodes linked by copy pointers. We denote by $c(\bar{x})$ the node indicated by the copy pointer in $\bar{x}$ (the next node after $\bar{x}$ in the family containing it). Each family corresponds to a single fat node in the fat node method, or equivalently to all versions of an augmented ephemeral node. The members of a family occur in the family list in increasing order by version stamp. The *valid interval of a persistent node* $\bar{x}$ is the interval of versions in the version list from the version stamp of $\bar{x}$ up to but not including the version stamp of $c(\bar{x})$, or up to and including the last version in the version list if $c(\bar{x})$ does not exist. The *valid interval of a field value* v is the interval of versions from the version stamp of v up to but not including the next version stamp of a value of the same field in the same family, or up to and including the last version if there is no such next version stamp. The valid intervals of the persistent nodes in a family partition the valid interval of the corresponding fat node; the valid intervals of the values of a field in a persistent node partition the valid interval of the node.

In the split node date structure, the symmetry properties become the following:

(5) If a node $\bar{x}$ contains a pointer to a node $\bar{y}$ such that the valid interval of the pointer includes a version i, then some node in the family containing $\bar{y}$ contains a pointer to some node in the family containing $\bar{x}$ such that this pointer has a valid interval including i also.

(6) The access array representing an access pointer named a contains a pointer to a node $\bar{x}$ in position i if and only if some node in the family containing $\bar{x}$ contains name a with version stamp i.

We need two more concepts before we can begin to discuss update operations. A pointer with version stamp i is *proper* if the pointer indicates a node $\bar{x}$ whose valid interval contains i. The pointer is *overlapping* if its valid interval is not contained within the valid interval of $\bar{x}$. We extend these definitions to access pointers as follows: the pointer stored in position i of an access array is *proper* if it indicates a node $\bar{x}$ whose valid interval contains i and *overlapping* otherwise. Except during update operations, all pointers in the split node structure will be proper and non-overlapping. When this is true, the symmetry properties (5) and (6) become the stronger properties (3) and (4). Furthermore, we can navigate through the structure exactly as in the fat node method, spending $O(1)$ time in the worst case per access step.

3.3.4. *Update Operations*

Node splitting affects the invariant that all pointers are proper and nonoverlapping, and the hard part of performing update operations is the restoration of this *pointer invariant*. Each update operation consists of two phases. The first phase simulates the steps of the update operation on the augmented ephemeral structure, adding new field values to new copies of nodes. This may invalidate the pointer invariant, i.e., it may cause some pointers to indicate the wrong nodes over parts of their valid intervals. The second phase repairs this damage by judiciously copying pointers. Since these new pointers may require node splitting to make room for them, which may in turn invalidate other pointers, the second phase is a cascading process of alternating pointer copying and node splitting, which ends when the pointer invariant is entirely restored.

Consider update operation i. The first phase proceeds exactly as in the fat node method except that new field values are not stored in previously existing persistent nodes but rather in new copies of these nodes. When an ephemeral update step creates a new node x, we create a new persistent node $\bar{x}$, with a version stamp of i and appropriate values of its information and original pointer fields. (The latter are null.) Subsequent changes to this node during update operation i are made by overwriting the appropriate fields, except that in addition, whenever a pointer field f is first set to a value other than null, if $i+$ exists we store in $\bar{x}$ a new null pointer with field name f and version stamp $i+$. (We do this to preserve the symmetry properties.)

Suppose an ephemeral update step changes field f of a node not created during update operation i. To simulate this step, we locate the persistent node $\bar{x}$ corresponding to x. If $\bar{x}$ has a version stamp of i, we merely modify the appropriate original field in $\bar{x}$. Otherwise, we proceed as follows. If i is followed by another version $i+$ in the version list and $c(\bar{x})$ either does not exist or has a version stamp greater than $i+$, we create two new nodes, $\bar{x}'$ and $\bar{x}''$. We make the copy pointer of $\bar{x}''$ point to $c(\bar{x})$, that of $\bar{x}'$ point to $\bar{x}''$, and that of $\bar{x}$ point to $\bar{x}'$. (That is, now $c(\bar{x})=\bar{x}'$, $c(\bar{x}')=\bar{x}''$, and $c(\bar{x}'')$ is the old value of $c(\bar{x})$.) We give node $\bar{x}'$ a version stamp of i and node $\bar{x}''$ a version stamp of $i+$. We fill in the original pointers and information fields of $\bar{x}'$ and $\bar{x}''$ by consulting those of $\bar{x}$, as follows. Each information field of $\bar{x}'$ and $\bar{x}''$ is exactly as in $\bar{x}$. Each original pointer field of $\bar{x}'$ ($\bar{x}''$) is filled in with the value of this field in $\bar{x}$ having rightmost version stamp not right of i (not right of $i+$) in the version list. We delete from $\bar{x}$ all extra pointers with version stamps of $i+$ (they have been copied into original fields of $\bar{x}''$) and move those with version stamps right of $i+$ to extra fields of $\bar{x}''$. Changes in the fields of x during update operation i are recorded in the original fields of $\bar{x}'=c(\bar{x})$, which corresponds to version i of x.

The construction is similar but simpler if i is the last version on the version list or $c(\bar{x})$ has a version stamp of $i+$. In this case we create a single new node $\bar{x}'$ with version stamp i, make the copy pointer of $\bar{x}'$ point to $c(\bar{x})$ and that of $\bar{x}$ point to $\bar{x}'$, and initialize the information and original pointer fields of $\bar{x}'$ as described above.

Changes in the fields of x during update operation i are recorded in the original fields of $\bar{x}' = c(\bar{x})$, which corresponds to version i of x.

During phase one, we also make a list of every node created during the phase and every node whose successor in its family list has changed. At the end of phase one, we process each node $\bar{x}$ on the list as follows. We inspect all pointers to $\bar{x}$, changing them if necessary to make them proper. We find these pointers by following pointers from $\bar{x}$ and looking in the nodes or access array positions they indicate. Each pointer to $\bar{x}$, if improper, can be made proper by changing it to indicate $c(\bar{x})$ or $c(c(\bar{x}))$. We now have a data structure in which all pointers are proper but not necessarily nonoverlapping. We construct a set S containing every node having at least one overlapping pointer. The potentially overlapping pointers are exactly the ones inspected and possibly changed to make them proper.

Once the set S is constructed, we begin the second phase. This phase consists of processing the set S by removing an arbitrary node $\bar{x}$, performing the following three steps, and repeating until S is empty, thereby making all pointers proper and nonoverlapping.

Step 1 (add new pointers). Construct a list L of the original and extra pointers in $\bar{x}$ and process these pointers in left-to-right order by version stamp. To process a pointer, determine whether the pointer is overlapping. (A pointer to $\bar{x}$ is regarded as nonoverlapping.) If not, continue with the next pointer. If so, let $\bar{y}$ be the node indicated by the pointer and let i be the version of $c(\bar{y})$. Add a pointer to $c(\bar{y})$, with version stamp i and the same field name as the pointer to $\bar{y}$, to the list L. The old pointer to $\bar{y}$ (but not necessarily the new pointer to $c(\bar{y})$), is now nonoverlapping. Continue with the pointer after the processed one.

After Step 1, all pointers in L are proper and nonoverlapping.

Step 2 (split $\bar{x}$). Let i be the version stamp of $\bar{x}$. If all the pointers in list L will fit into $\bar{x}$ (there are at most $2e$ such pointers with version stamp right of i in the version list) fit them all into $\bar{x}$ and skip Step 3; no node splitting is necessary. Otherwise, work from the back of L, partitioning it into groups each of which will fit into a single node with e extra pointer fields left vacant, except for the group corresponding to the front of L, which should fit into a single node with possibly no extra pointer fields left vacant. Store the first group of pointers in $\bar{x}$ and each successive group in a newly created node, to which the copy pointer of the previous node points. The last new node points to the old $c(\bar{x})$. The version stamp of a new node is the smallest version stamp of a pointer stored in it; all pointers with this version stamp are stored in original pointer fields, and all pointers with rightward version stamps in extra fields. Additional values needed for original pointer fields and for information fields can be obtained from the preceding node in the family list.

After Step 2, some of the pointers to $\bar{x}$ may be improper.

Step 3 (make all pointers proper). By scanning through $\bar{x}$ and the newly

created nodes, locate all pointers in these nodes to $\bar{x}$. Make each such pointer proper by making it point to the node containing it. (Each such pointer is guaranteed to be nonoverlapping). By following pointers contained in $\bar{x}$ and in the newly created nodes, locate all other pointers to $\bar{x}$. Make each one proper (by making it point to the last node among $\bar{x}$ and the newly created nodes having version stamp not right of that of the pointer). If such a pointer is overlapping, add the node containing it to S if it is not already there.

After Step 3, all pointers are proper and all nodes containing overlapping pointers are in S. It follows by induction that at the end of the second phase, all pointers are proper and nonoverlapping. A proof by induction on the number of update steps shows that the correct correspondence between the ephemeral structure and the persistent structure is maintained. The correctness of the node-splitting method follows.

Remark. The node-splitting method as we have described it always creates one or two new copies of each node affected by an update operation. However, if the updates to a node during an update operation are only to pointer fields and all the new pointers fit in the extra fields of the existing node, then these new copies of the node need not be made. If the new pointers fit in the extra fields of the existing node and in one new node, then only one copy instead of two needs to be made. This optimization saves some space but complicates the implementation of the method, because it increases the number of cases that must be considered during phase one of an update operation.

3.3.5. *An Example*

Figure 6 shows a fully persistent search tree built using the node-splitting method, modified to avoid unnecessary node copying as proposed in the remark above. The value chosen for e is one (i.e., each node contains three original pointers and two extra pointers). This choice is not sufficient to guarantee a linear-space data structure, but it keeps the example manageable. In this example, even with $e = 1$ there is no node splitting during phase two.

3.3.6. *Space and Time Analysis*

It remains for us to analyze the time and space efficiency of the node-splitting method. Phase one of an update operation creates $O(1)$ new nodes per update step. Steps 1, 2, and 3 of phase two take $O(1)$ time plus $O(1)$ time per newly created node, because of the following observations:

(i) During Step 1, there are at most $k + 2e = O(1)$ unprocessed pointers on the list L;

(ii) the pointers inspected during Step 3 that are not in $\bar{x}$ and not in new nodes are in at most $k + 2e$ different nodes, which means that at most $(k + 2e)^2 = O(1)$ such pointers are inspected, and at most $k + 2e = O(1)$ nodes are

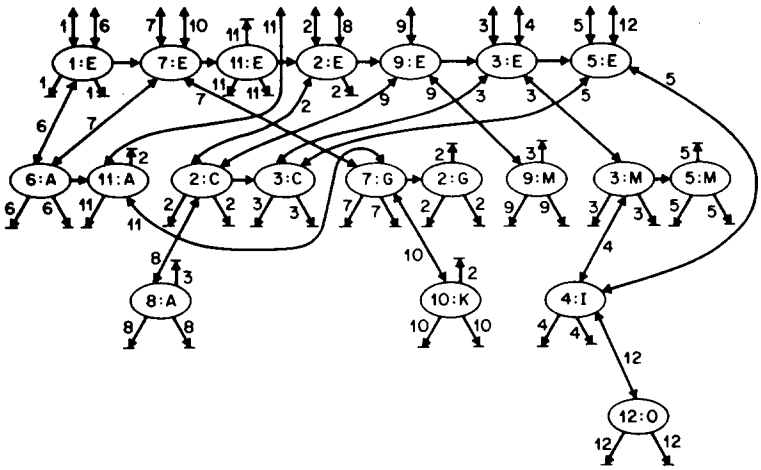

FIG. 6. A fully persistent search tree with inverse (parent) pointers built using the node-splitting method, for the update sequence in Fig. 3. Each pointer is numbered with its version stamp; the nodes are also numbered with their version stamps. The copy pointers leave the right sides of nodes and enter the left sides. As in Fig. 5, pointer fields of deleted nodes are made null during the deletion.

added to S. The time to make each inspected pointer proper is proportional to one plus the number of nodes created in Step 2.

It follows that the total time spent during an update operation is $O(1)$ per update step plus $O(1)$ per newly created node. We shall derive an $O(1)$ amortized bound on the number of nodes created per update step, thus obtaining an $O(1)$ amortized bound on the time and space per update step.

We define the potential of the split node structure to be $e/(e-k+1)$ times the number of persistent nodes minus $1/(e-k+1)$ times the number of vacant extra pointer fields, counting only at most e vacant extra pointer fields per node. (Recall that we require $e \geqslant k$.) We define the amortized space cost of an update step to be the number of new nodes the step creates plus the net increase in potential it causes. Because the initial potential is zero and the potential is always nonnegative, the total number of nodes in the split node structure is at most the total amortized space cost of the update operations.

We complete the analysis by showing that the amortized space cost of an update operation is linear in the number of update steps. Consider an update operation that performs u update steps. Phase one of the update operation has an amortized space cost of $O(u)$ and creates at most $2u$ new nodes. Let l be the number of new nodes created during phase two. Each new node created during phase one or two can ultimately result in the creation of up to k new pointers (one new incoming pointer corresponding to each of its k original outgoing pointers). Creation of a new node in phase two adds zero to the potential. The amortized space cost of phase two is thus at most l (for the newly created nodes) plus $(2u+l)k/(e-k+1)$

(for the vacant extra pointer fields filled by the new pointers) minus $(e+1)l/(e-k+1)$ (because each newly created node in phase two contains at least $e+1$ pointers that do not contribute to the potential, e in extra pointer fields and one in an original pointer field). This sum is

$$\begin{aligned} & l+(2u+l)k/(e-k+1)-(e+1)l/(e-k+1) \\ & \quad = l+2uk/(e-k+1)+l(k-e-1)/(e-k+1) \\ & \quad = 2uk/(e-k+1) = O(u). \end{aligned}$$

Thus the amortized space cost of phase two is $O(u)$. It follows that the amortized space cost of the entire update operation is $O(u)$, i.e., $O(1)$ per update step.

In summary, we have shown that the node-splitting method will make a linked data structure of constant bounded in-degree fully persistent at an amortized time and space cost of $O(1)$ per update step and a worst-case time of $O(1)$ per access step. In obtaining these $O(1)$ bounds it suffices to use the simple method of Dietz and Sleator [11] to maintain the version list. We note in conclusion that our node-splitting process is very similar to the *fractional cascading* method of Chazelle and Guibas [7, 8], which was devised for an entirely different purpose. The connections between these two ideas deserve further exploration.

4. Persistent Balanced Search Trees

In this section we shall focus on the question of making a specific data structure, namely a balanced search tree, persistent. We shall discuss a particular kind of search tree, the red–black tree, although our ideas apply as well to certain other kinds of search trees. A *red–black tree* [13, 32, 32] is a binary search tree, each node of which is colored either *red* or *black*, subject to the following three constraints:

(i) (Missing node convention) Every missing (external) node is regarded as being black.

(ii) (Red constraint) Every red node has a black parent or has no parent, i.e., is the root.

(iii) (Black constraint) From any node, all paths to a missing node contain the same number of black nodes.

The color constraints imply that the depth of an n-node red–black tree is at most $2\log_2 n$ and hence that the time to access any item is $O(\log n)$. To maintain the color constraints, a red–black tree is rebalanced after each insertion or deletion. Rebalancing requires recoloring certain nodes and performing local transformations called *rotations*, each of which preserves the symmetric order of the items in the nodes but changes the depths of some of them, while taking $O(1)$ time (see Fig. 7).

A bottom-up rebalancing strategy [31, 32] leads to especially efficient insertion

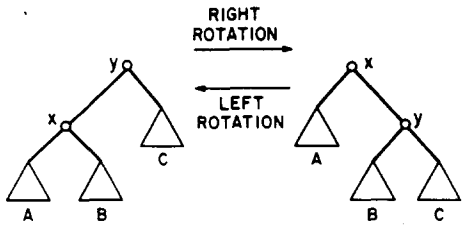

FIG. 7. A rotation in a binary tree. The tree shown can be a subtree of a larger tree.

and deletion algorithms. To perform an insertion, we follow the access path for the item to be inserted until we reach a missing node. At the location of the missing node we insert a new node containing the new item. We color the new node red. This may violate the red constraint, since the parent of the new node may be red. In this case we bubble the violation up the tree by repeatedly applying the recoloring transformation of Fig. 8a until it no longer applies. This either eliminates the violation or produces a situation in which one of the transformations in Figs. 8b, c, and d applies, each of which leaves no violation.

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 this node with the node preceding it in symmetric order, which we find by starting at the left child and following successive right pointers

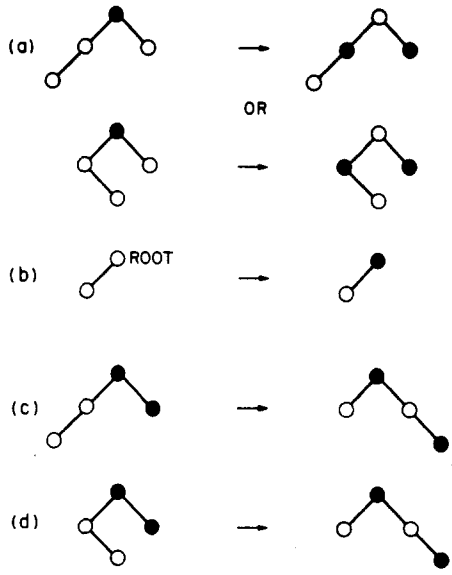

FIG. 8. The rebalancing transformations in red–black tree insertion. Symmetric cases are omitted. Solid nodes are black; hollow nodes are red. All unshown children of red nodes are black. In cases (c) and (d) the bottommost black node shown can be missing.

until reaching a node with no right child. Now the item to be deleted is in a node with at most one child. We replace this node by its only child (if any). This does not affect the red constraint but will violate the black constraint if the deleted node is black. If there is a violation, the replacing node (which may be missing) is *short*: paths from it to missing nodes contain one fewer black node than paths from its sibling. If the short node is red we merely color it black. Otherwise, we bubble the shortness up the tree by repeating the recoloring transformation of Fig. 9a until it no longer applies. Then we perform the transformation in Fig. 9b if it applies, followed if necessary by one application of 9c, d, or e.

An insertion requires $O(\log n)$ recolorings plus at most two rotations; a deletion, $O(\log n)$ recolorings plus at most three rotations. Furthermore the amortized number of recolorings per update operation is $O(1)$, i.e., m update operations produce $O(m)$ recolorings [15, 16, 20].

4.1. *Partial Persistence*

We can make a red–black tree partially persistent by using the node copying method of Section 2 with e (the number of extra pointers per node) equal to one. Because in the ephemeral structure there is only one access path to any node, we do not need inverse pointers. We obtain an additional simplification because colors are not used in access operations. Thus we do not need to save old node colors but can

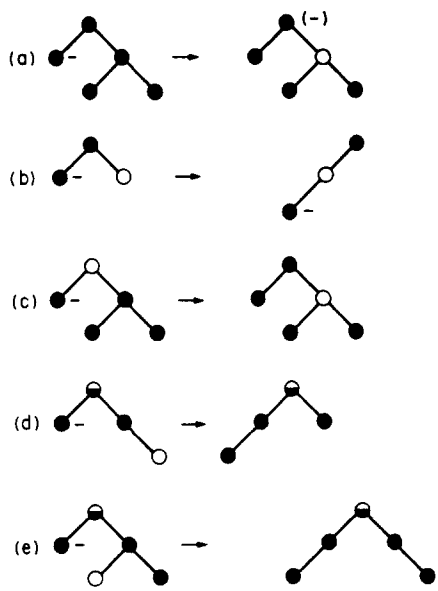

FIG. 9. The rebalancing transformations in red–black tree deletion. The two ambiguous (half-solid) nodes in (d) have the same color, as do the two in (e). Minus signs denote short nodes. In (a), the top node after the transformation is short unless it is the root.

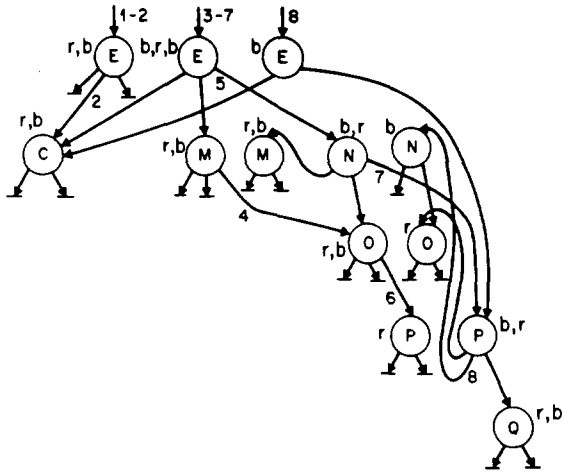

FIG. 10. A partially persistent red–black tree built using the node copying method of Section 2. The sequence of operations consists of insertions of E, C, M, O, N, P, Q, followed by deletion of M. Extra pointers are labeled with their version stamps. Each node is labeled with its successive colors. Insertion of item N triggers the transformation in Fig. 8d. Deletion of M triggers the transformation in Fig. 9d.

instead overwrite them. It is also easy to dispense with the copy pointers and the version stamps of the nodes. Thus each persistent node contains an item, three pointers, a color bit, and a version stamp and field name (left or right) for the third (extra) pointer. Figure 10 illustrates a partially persistent red–black tree.

A partially persistent red–black tree representing a history of m insertions and deletions occupies $O(m)$ space and allows $O(\log n)$-time access to any version, where n is the number of items in the accessed version. The worst-case time per insertion or deletion is also $O(\log n)$. These trees and their applications, of which there are a number in computational geometry, are discussed more fully in a companion paper [29].

The node-copying method applies more generally to give an $O(m)$-space partially persistent form of any balanced search tree with an $O(1)$ amortized update time for insertion and deletion. Such trees include weight-balanced trees [3, 24] and weak B-trees [15, 16, 20]. Increasing the number of extra pointers per node allows the addition of auxiliary pointers such as parent pointers and level links [4, 16]. Various kinds of *finger search trees* can be built using such extra pointers [4, 16, 17, 18, 35, 36]. (A finger search tree is a search tree in which access operations in the vicinity of certain preferred items, indicated by access pointers called *fingers*, are especially efficient.) By applying the node-copying method we can obtain partially persistent finger search trees occupying $O(m)$ space, with only a constant factor blowup in the amortized access and update times over their ephemeral counterparts.

4.2. *Full Persistence*

Let us turn to the problem of making a red–black tree fully persistent. We would like to obtain an $O(m)$-space fully persistent red–black tree by using the node splitting method of Section 3. There are two difficulties, however. First, we must save old color bits; we are not free to overwrite them. Second, if we include recoloring time the $O(1)$ update time bound for ephemeral insertion and deletion is only amortized, whereas we need a worst-case $O(1)$ bound. Thus our goal becomes the construction of a variant of ephemeral red–black trees in which the worst-case update time per insertion or deletion, including recoloring, is $O(1)$.

To obtain such a variant, we use an idea of Tsakalidis [35, 36], *lazy recoloring*. Observe that the recoloring during a insertion or deletion affects only nodes along a single path in the tree and children of nodes along the path. Furthermore, the recoloring is extremely uniform except at the ends of the path. We exploit this observation by postponing all the recoloring except that at the path ends. We encode the postponed recoloring by storing certain information in the tree, information sufficient to allow us to perform the delayed recoloring incrementally, as it becomes necessary.

Consider an insertion. If we ignore $O(1)$ update steps at the beginning of the insertion and $O(1)$ update steps at the end, the rest of the updating consists of recoloring a path and siblings of nodes along the path as illustrated in Fig. 11: each node on the path has its color flipped, and the red siblings of nodes on the path become black. We call the configuration consisting of the path and the siblings to be recolored an *insertion interval*. The top and bottom nodes of the path are the *top* and *bottom* of the insertion interval, respectively. The siblings of nodes on the path that are not to be recolored are the *fringe nodes* of the interval. (The sibling of the top node is neither in the interval nor a fringe node.) Rather than actually recoloring an insertion interval, we store in the top node of the interval two pieces of information: a flag indicating the beginning of an insertion interval and the item in the bottom node of the interal. (Thus the top node contains two items, including its own.)

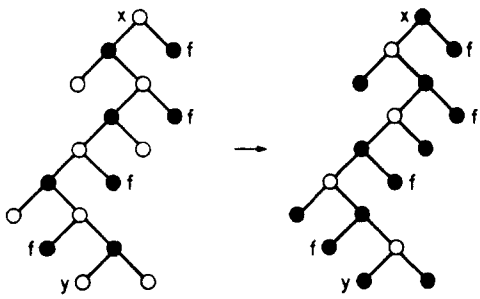

FIG. 11. An insertion interval with top node x and bottom node y, showing the desired recoloring. The nodes marked "f" are the fringe nodes of the interval. The interval is represented by storing in node x the item in node y (which is also still stored in y) along with a flag indicating that x is the top of an insertion interval.

In the case of a deletion all but $O(1)$ of the update time consists of recoloring siblings of nodes along a path as in Fig. 12: all the siblings, originally black, become red. We call the configuration consisting of the path and the siblings to be recolored a *deletion interval.* The top and bottom nodes of the path are the *top* and *bottom* of the interval, respectively. Rather than performing the recoloring in such an interval, we store in the top node of the interval two pieces of information: a flag indicating the beginning of a deletion interval and the item in the bottom node on the path.

We call insertion intervals and deletion intervals *recoloring intervals.* Our objective is to maintain the invariant that the recoloring intervals are vertex-disjoint. We shall show that this can be done using $O(1)$ node recolorings per insertion or deletion. Let us see what the disjointness of the recoloring intervals implies. Since during an access or update operation we always enter a recoloring interval via the top node, we always know whether we are in an interval and when we are entering or leaving one. (When entering an interval, we remember the extra item stored in the top node, which allows us to tell when we leave the interval.) Thus when we are in an interval we can keep track of the correct current node colors, which are not necessarily the same as their actual (unchanged) colors. This gives us the information we need to perform insertions and deletions.

To keep the recoloring intervals disjoint, we maintain insertion intervals so that their top and bottom nodes are actually red (but should be black), as in Fig. 11. This property can be imposed on an insertion interval not having it by coloring $O(1)$ nodes at the top and bottom of the interval, thereby shrinking it to an interval with the property. More generally, a recoloring interval can be shrunk by $O(1)$ nodes at either the top or the bottom by doing $O(1)$ recoloring. It can also be split in the middle into two recoloring intervals by doing $O(1)$ recoloring (see Fig. 13). Shrinking at the top of an interval requires moving the header information in the top node of the interval down to the new top node. Shrinking at the bottom requires changing the header information in the top node. Splitting an interval requires changing the header information in the top node of the original interval and adding header information to the new top node of the bottom half of the interval. In all cases the update time is $O(1)$.

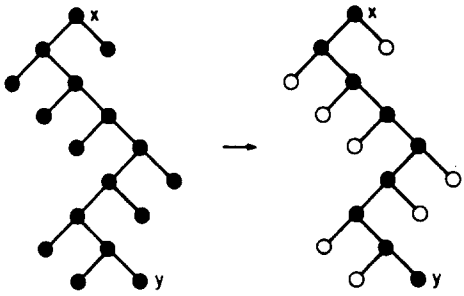

FIG. 12. A deletion interval with top node x and bottom node y, showing the desired recoloring. The interval is represented by storing in node x the item in node y along with a flag indicating that x is the top of a deletion interval.

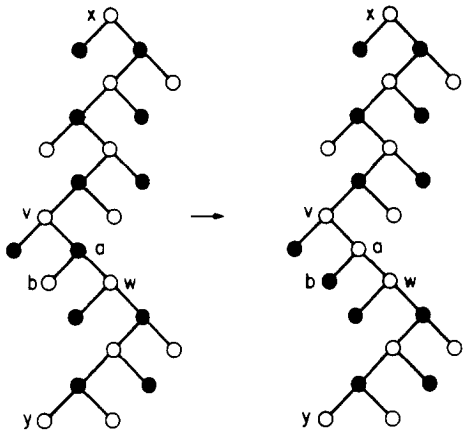

FIG. 13. Splitting an insertion interval. The nodes in the interval have their uncorrected colors. Recoloring the nodes a and b splits the insertion interval from x to y into two insertion intervals, one from x to v and one from w to y.

We maintain two invariants on recoloring intervals besides vertex disjointness: (i) every recoloring interval contains at least two nodes, and (ii) every fringe node of an insertion interval has a correct color of black. When a recoloring interval shrinks to one node, we correctly color it. When a fringe node of an insertion interval has its correct color changed from black to red, we split the insertion interval so that the node is no longer a fringe node. (A fringe node can be in another recoloring interval, but only as its top node.)

The crucial property of recoloring intervals is that they stop the propagation of color changes during an insertion or deletion. This is what allows us to keep the recoloring intervals disjoint while doing only $O(1)$ update steps per insertion or deletion.

In general, we represent a red–black tree as a collection of correctly colored nodes not in recoloring intervals and a vertex-disjoint set of recoloring intervals having the two properties listed above. To perform an insertion, we search from the tree root for the insertion location, keeping track of the bottommost recoloring interval on the way down the tree. We add the appropriate new red node to the tree. If the red constraint is violated, we walk up the tree implicitly applying the propagating insertion rule (Fig. 8a), without actually making the implied color changes, until either this rule is no longer applicable or applying the rule would recolor a node in a recoloring interval or a fringe node. There are only three ways in which the propagating insertion rule can recolor a node in a recoloring interval or a fringe node, each of which is terminal after at most one more transformation among those in Fig. 8 (see Fig. 14).

To complete the insertion, we shrink the recoloring intervals away from the vicinity of the terminating transformation sufficiently so that this transformation applies to nodes not in recoloring intervals and not fringe nodes. This may require

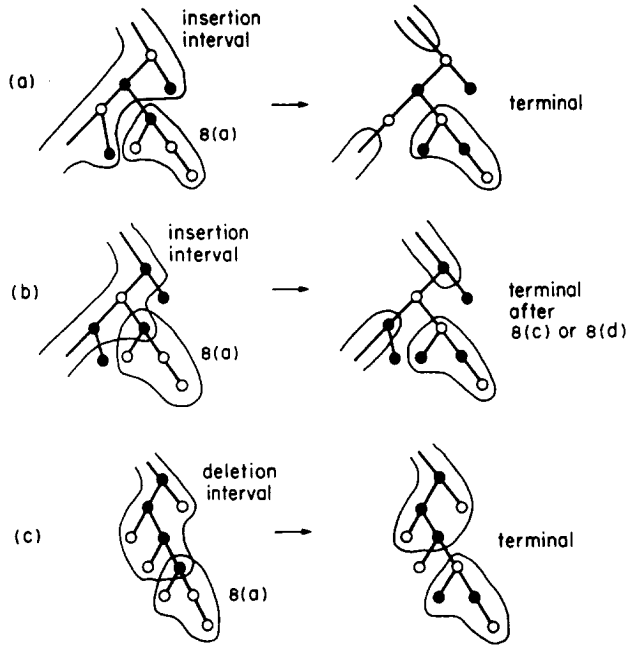

FIG. 14. The ways a propagating insertion can run into a recoloring interval. Recoloring intervals are shown correctly colored. In each case the transformation of Fig. 8a is to be applied. (a) Recoloring a fringe node. This terminates the insertion. The insertion interval splits. (b) Recoloring a node in an insertion interval. The next transformation is terminal. The interval splits. (c) Recoloring a node in a deletion interval. The recolored node must be the bottom of the interval. This terminates the insertion. The interval gets shorter.

splitting at most one recoloring interval, the bottommost along the original search parth. We encode the remaining unperformed color changes as a single new insertion interval, disjoint from all other recoloring intervals. (Some of its fringe nodes may be the tops of other intervals.)

A deletion is similar. We search from the tree root for the item to be deleted, keeping track of the bottommost recoloring interval along the search path. If the item to be deleted is in a node with two children, we swap this node with its predecessor in symmetric order. (If either of the swapped nodes is the top or bottom of a recoloring interval, the header information in the top node of the interval may need to be updated.) Once the item to be deleted is in a node with only one child, we delete this node, first shrinking away from it any recoloring interval in the vicinity so that it is not affected by the deletion. We replace the deleted node by its child if it has one. Then we walk back up along the search path from the deleted node, implicitly applying the propagating deletion rule (Fig. 9a), until it no longer applies. This rule cannot recolor any node in a recoloring interval or any fringe node; furthermore, as soon as a node in a recoloring interval or a fringe node

becomes short, the propagating rule no longer applies. We shrink away recoloring intervals in the vicinity of the top of the recoloring path, perform the up to two terminating transformations needed to complete the deletion, and encode the unperformed color changes in a new deletion interval, disjoint from all other intervals.

The total time for either an insertion or a deletion is $O(\log n)$, but the update time (in the technical sense defined in Section 1) is only $O(1)$. This improves Tsakalidis's original lazy recoloring scheme (invented to solve a different problem), which has an $O(\log n)$ worst-case update time bound.

Since red–black trees with lazy recoloring have an $O(1)$ worst-case update time bound per insertion or deletion, the node-splitting method of Section 3 makes such trees fully persistent an an amortized space cost of $O(1)$ per insertion or deletion. The worst-case time for an access operation is $O(\log n)$, as is the amortized time of an insertion or deletion. The only competitive method for obtaining full persistence of search trees is due independently to Myers [21, 23], Krijnen and Meertens [19], Reps, Teitelbaum, and Demers [27], and Swart [30]. Their method consists of copying the entire access path and all other changed nodes each time an insertion or deletion occurs. Although very simple, this method requires $O(\log n)$ space and time in the worst case per insertion or deletion. Thus our method saves a logarithmic factor in space.

5. Fully Persistent Balanced Search Trees Revisited

In this section we discuss an alternative way to make balanced search trees fully persistent. This method makes the $O(\log n)$ time bound and the $O(1)$ space bound per insertion or deletion worst-case instead of amortized. As in Section 4 we shall focus on red–black trees, although our method applies more generally. Our approach is to combine node copying as refined for the special case of search trees with a new idea, that of *displaced storage of changes.* Instead of indicating a change to an ephemeral node x by storing the change in the corresponding persistent node $\bar{x}$, we store information about the change in some possibly different node that lies on the access path to $\bar{x}$ in the new version. Thus the record of the change is in general displaced from the node to which the change applies. The path from the node containing the change information to that of the affected node is called the *displacement path.* By copying nodes judiciously, we are able to keep the displacement paths sufficiently disjoint to guarantee an $O(1)$ worst-case space bound per insertion or deletion and an $O(\log n)$ worst-case time bound per access, insertion, or deletion.

5.1. *Representation of the Version Tree*

To obtain full persistence with this method, we must change the scheme used to navigate through the persistent structure. Instead of linearizing the version tree as

in Section 3, we maintain a representation of the version tree that allows us to determine, given two versions i and j, whether or not i is an ancestor of j in the version tree. The appropriate representation, discovered by Dietz [10], is a list containing each version twice, once in its preorder position and once in its postorder position with respect to a depth-first traversal of the version tree. We call this list the *traversal list*. If i_{pre} and i_{post} denote the preorder and postorder occurrences of version i in the traversal list, then i is an ancestor of j in the version tree if and only if j_{pre} precedes i_{pre} but does not precede i_{post} in the traversal list. To update the traversal list when a new version i is created, we insert two occurrences of i in the traversal list immediately after the preorder occurrence of the parent of i; the first occurrence is i_{pre} and the second is i_{post}. If we maintain the traversal list using the more complicated representation of Dietz and Sleator [11], then the worst-case time per ancestor query or creation of a version is $O(1)$.

5.2. *The Displaced Change Data Structure*

The persistent structure itself consists of a collection of persistent nodes, each of which contains an ordered pair of *version records*. A version record has exactly the same structure as an ephemeral node; namely (in the case of a binary tree), it contains an item, a left pointer, a right pointer, and whatever additional fields are used in the particular kind of search tree. For a red–black tree with lazy recoloring, there are three additional fields, holding a color bit, an extra item specifying a recoloring interval, and a bit indicating whether the interval is an insertion or a deletion interval. The first version record in a persistent node is the *original record*; its item is regarded as being associated with the node itself. The second record in a node is the *change record*; it has a version stamp.

The item in the change record of a node need not be the same as the item associated with the node. If these items are different, the change record is said to be *displaced*. Let r be a change record in a node $\bar{x}$, let e be the item in r, and let i be the version stamp of r. The *displacement path* of r is the path taken by a search for e in version i that begins with $\bar{x}$ as the current node and proceeds as follows. If $\bar{y}$ is the current node, i is compared with the version stamp j of the change record in $\bar{y}$. If $\bar{y}$ has no change record, if the change record in $\bar{y}$ is displaced, or if i is not a descendant of j in the version tree, then the search follows the left or right pointer in the original version record of $\bar{y}$, depending on whether e is less than or greater than the item associated with $\bar{y}$. Otherwise ($\bar{y}$ has a nondisplaced change record with a version stamp that is an ancestor of i in the version tree), the search follows the left or right pointer in the change record, again depending on whether e is less than or greater than the item associated with $\bar{y}$. The search stops when it reaches a node $\bar{z}$ whose associated item is e. Node $\bar{z}$ is the last node on the displacement path; it is the one that should actually contain record r. Node $\bar{x}$ is called the *head* of the displacement path and node $\bar{z}$ is called the *tail*; the *body* of the path consists of all nodes on the path except the head, including the tail.

In addition to the traversal list and the linked collection of persistent nodes, we

maintain an access array containing a pointer for each version to the node representing the root of the corresponding ephemeral tree. The pointer for the appropriate version is installed after each update operation, as in Sections 2 and 3.

5.3. *Displacement Path Invariants*

The crux of the method is the maintainence of the displacement paths. We maintain the following three invariants on the data structure:

(i) Every change record r has a displacement path, i.e., the search described above eventually reaches a node whose associated item is the one in r. (Record r is displaced if and only if this path has a nonempty body.)

(ii) If r is a change record, then every node in the body of its displacement path contains a change record whose version stamp is not a descendant in the version tree of the version stamp of r.

(iii) Let r be a change record, and let s be another change record in a node on the displacement path of r, whose version stamp is an ancestor of the version stamp of r. Then the body (if any) of the displacement path of s is disjoint from the displacement path of r.

5.4. *Access Operations*

We navigate through the structure as follows. Suppose we wish to access item e in version i. We start a search for e at the root node of version i. During the search, we are at a current node $\bar{x}$, and we may have in hand a displaced record r. The general step of the search is as follows. If $\bar{x}$ has e as its associated item, the search terminates. Otherwise, we examine the original and change records in $\bar{x}$ and r, the record in hand. Among these records, we select the one whose item is the same as the item associated with $\bar{x}$ and whose version stamp is an ancestor of i in the version tree; if more than one record qualifies, we select the one whose version stamp is the nearest ancestor of i. We follow the left or right pointer of this version record, depending upon whether e is less than or greater than the item associated with $\bar{x}$. We compute the new displaced record in hand as follows. Let s be the change record in the old current node $\bar{x}$, and let $\bar{y}$ be the new current node. The new displaced record in hand is r (the old record in hand) if $\bar{y}$ is on the displacement path of r, and it is s if $\bar{y}$ is on the displacement path of s and the version stamp of s is an ancestor of i. In any other case there is no new displaced record in hand. Invariants (ii) and (iii) imply that r and s cannot both qualify to be the new record in hand. The worst-case time per access step is $O(1)$.

5.5. *Operations on Displacement Paths*

In order to simulate ephemeral update steps, we need three operations that shorten displacement paths. The first is *shrinking* a displacement path by removing its

tail. Let r be a displaced change record in a node $\bar{x}$ and let $\bar{z}$ be the tail of the displacement path of r. Let i be the version stamp of r, $\bar{y}$ the node preceding $\bar{z}$ in the displacement path of r, and s the record in $\bar{y}$ whose pointer to $\bar{z}$ is followed in traversing the displacement path of r. To shrink the displacement path of r, we create a new node $\bar{z}'$, regarded as a copy of $\bar{z}$, and add r to $\bar{z}'$ as its original record. We also create a copy s' of s, identical to s except that the pointer to $\bar{z}$ is replaced by a pointer to $\bar{z}'$, and insert s' in place of r as the change record in $\bar{x}$. Record s' gets version stamp i (see Fig. 15). After the shrinking, record r is no longer displaced; record s' is displaced unless $\bar{y} = \bar{x}$, and s' has a displacement path equal to the old path of r minus the tail. Node $\bar{z}'$ has no change record; thus the shrinking introduces a new place to put a change record.

The second, slightly more complicated operation on a displacement path is *splitting*. Let r be a displaced change record with version stamp i in a node $\bar{x}$, let $\bar{z}$ be a node in the body of the displacement path of r other than the tail, let $\bar{y}$ be the predecessor of $\bar{z}$ along the displacement path of r, let s be the record in $\bar{y}$ whose pointer to $\bar{z}$ is followed in traversing the displacement path, and let t be the record in $\bar{z}$ whose pointer is followed in continuing along the displacement path from $\bar{z}$. To split the path at $\bar{y}$, we create a new node $\bar{z}'$, regarded as a copy of $\bar{z}$, containing a copy of t as its original record and r with version stamp i as its change record. We also create a copy s' of s, replacing the pointer to $\bar{z}$ by a pointer to $\bar{z}'$, and insert s' in place of r as the change record of $\bar{x}$. Record s' gets version stamp i (see Fig. 16). This splits the old displacement path of r in two; the top half, from $\bar{x}$ to $\bar{y}$, is now the displacement path of s'; the bottom half, from $\bar{z}'$ (instead of $\bar{z}$) to the old tail is the new displacement path of s. Both shrinking and splitting preserve invariants (i)–(iii).

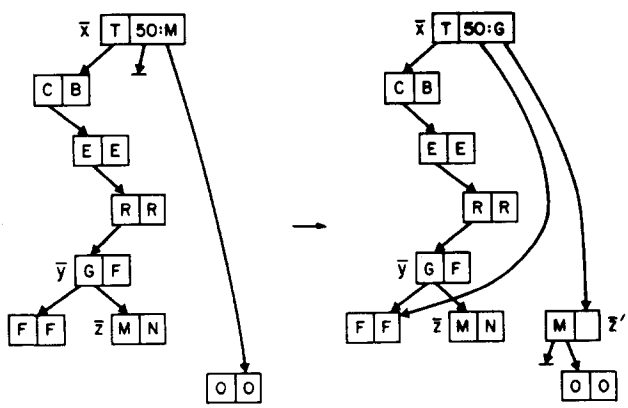

FIG. 15. Shrinking a displacement path. Irrelevant fields of nodes are omitted; all unshown version stamps of change records are assumed to be unrelated to 50 in the version tree. The displacement path with head $\bar{x}$ and tail $\bar{z}$ is shrunk by making a new copy $\bar{z}'$ of $\bar{z}$. The new displacement path is from $\bar{x}$ to $\bar{y}$.

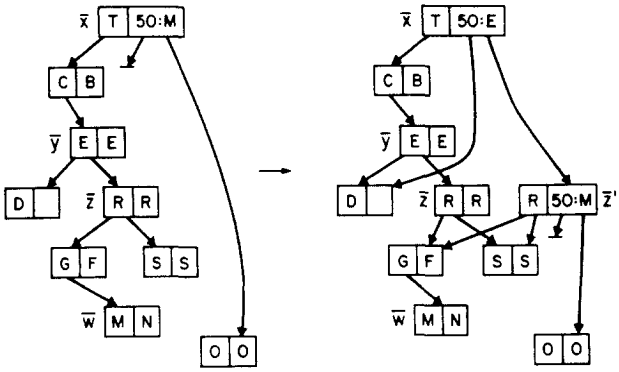

FIG. 16. Splitting a displacement path. Irrelevant fields of nodes are omitted; all unshown version stamps of change records are assumed to be unrelated to 50 in the version tree. The displacement path with head $\bar{x}$ and tail $\bar{w}$ is split at $\bar{y}$ by making a new copy $\bar{z}'$ of $\bar{z}$. The new displacement paths are from $\bar{x}$ to $\bar{y}$ and from $\bar{z}'$ to $\bar{w}$.

5.6. *Update Operations*

We are now ready to discuss the simulation of ephemeral update steps. For simplicity, we shall assume that each ephemeral update operation changes only one ephemeral node. The method is easily modified to handle any number of changes per update operation and , in particular, a constant number, as in red–black trees with lazy recoloring. Suppose we wish to simulate an ephemeral update that converts version i into the version j by changing node x. Let $\bar{x}$ be the persistent node corresponding to x. We construct a change record r with version stamp j containing the field values representing the new version of x. To find a place to store, r we back up along the access path to $\bar{x}$ in version i until we back up past the root of version i or we reach a node $\bar{y}$ such that either $\bar{y}$ has no change record or $\bar{y}$ is in the body of the displacement path of another change record s whose version stamp is an ancestor of i and such that s is in a node, say $\bar{z}$, on the access path to $\bar{x}$ in version i. There are four cases:

(1) We back up past the root. In this case we create a new root node for version j. The new node contains as its original record a copy of the record in the old root whose pointers are followed in version i. As its change record, the new node contains r.

(2) Node $\bar{y}$ contains no change record. We merely store r in $\bar{y}$ as its change record.

(3) Node $\bar{y}$ is the tail of the displacement path of s. We shrink the displacement path of s and store r in the newly created copy $\bar{y}'$ of $\bar{y}$ as its change record.

(4) Node $\bar{y}$ is in the body of the displacement path of s but is not the tail. We

split the displacement path of s at $\bar{y}$ and proceed as in case (3); after the displacement path is split, node $\bar{y}$ is the tail of the displacement path of the record replacing s in $\bar{z}$.

In all cases the number of nodes added to the persistent structure is $O(1)$ and invariants (i)–(iii) are preserved. If we apply this method to red–black trees with lazy recoloring, the worst-case time for an insertion or deletion is $O(\log n)$, including the time necessary to find the insertion or deletion position but not including the time necessary to update the list representing the version tree. The space used per insertion or deletion is $O(1)$. Thus we obtain a fully persistent form of red–black trees with the same resource bounds as in Section 4, but the $O(1)$ space bound and the $O(\log n)$ time bound per insertion or deletion are worst-case rather than amortized.

There are a number of ways in which one can vary the displaced-change method in an attempt to simplify it and improve its space usage by a constant factor. These include storing more than one change record in a persistent node, storing changes to individual fields (as in Sections 2 and 3) rather than sets of changes to all the fields of an ephemeral node, and storing in a change record a pointer to the node that should contain it rather than the item in that node. We leave as a topic for future research the discovery of the most elegant and practical variant of the method.

6. Applications, Extensions, and Open Problems

In this concluding section, we mention some applications and extensions of our work and some open problems. We first note some of the kinds of persistent data structures that can be obtained using our techniques.

(i) An ephemeral stack or queue can be implemented as a singly-linked list with $O(1)$-time insertion and deletion of items (at the appropriate ends) [31]. The node-copying method of Section 2 makes these data structures partially persistent at a space and time cost of $O(1)$ per operation; the node-splitting method of Section 3 gives the same bounds for fully persistent stacks and queues.

(ii) An ephemeral deque (double-ended queue) can be implemented as a doubly-linked list with $O(1)$-time insertion and deletion of items at either end [31]. We can make this structure partially or fully persistent using the method of Section 2 or 3, respectively, at a time and space cost of $O(1)$ per operation.

(iii) As discussed in Section 4, the node-copying method of Section 2 makes red–black trees partially persistent at an amortized space cost of $O(1)$ per insertion or deletion and a worst-case time cost of $O(\log n)$ per access, insertion, or deletion. The node-splitting method of Section 3 makes red–black trees with lazy recoloring fully persistent in the same resource bounds, except that the time bound for insertion and deletion becomes amortized instead of worst-case. As discussed in Sec-

tion 5, the displaced change method makes red–black trees fully persistent at a worst-case space cost of $O(1)$ and a worst-case time cost of $O(\log n)$ per insertion or deletion. These methods can be applied to other kinds of balanced search trees as well.

(iv) The techniques of Sections 4 and 5 can be extended to produce partially or fully persistent red–black trees with fingers, in which the time to access, insert, or delete an item d positions away from a finger is $O(\log d)$ and the time to move a finger d positions is $O(\log d)$. The space for an insertion or deletion is $O(1)$. For a finger movement, the space bound is $O(1)$ in the case of partial persistence and $O(\log d)$ in the case of full persistence. To obtain full persistence, an extension of the lazy recoloring method of Section 4 is needed.

These data structures have applications in computational geometry, text and file editing, and implementation of very high level languages. Partially persistent search trees have a variety of uses in geometric retrieval. In such geometric applications, the update operations are indexed by real numbers rather than by consecutive integers. To give access to the persistent structure, the access pointers to the roots of the various versions must be stored in a balanced search tree, ordered by index. This increases the time for an access or update operation to $O(\log m)$ from $O(\log n)$, since the time to find the access pointer for the desired version is $O(\log m)$ instead of $O(1)$. As discussed in the companion paper [29], partially persistent balanced search trees can be used to give a simple solution to the planar point location problem for a polygonal subdivision with a query time of $O(\log n)$, a space bound of $O(n)$, and a preprocessing time bound of $O(n \log n)$, where n is the number of line segments defining the subdivision. They can also be used as a substitute for Chazelle's "hive graph" structure, which has a number of uses in geometric retrieval [5]. Cole [9] lists a number of other geometric applications of partially persistent balanced search trees.

Fully persistent balanced search trees can be used to represent any sorted set or list as its evolves over time, allowing updates in any version. Such structures have been used in text editing by Myers [21, 23], in program editing by Reps, Teitelbaum, and Demers [27], and in implementation of high-level data structures in the programming language *B* by Krijnen and Meertens [19]. Our fully persistent red–black trees save a logarithmic space factor in these applications. Fully persistent deques and balanced search trees can also be used to implement lists and sets in SETL and related very high level programming languages.

The node-copying method of Section 2 can be modified so that it is write-once except for access pointers. The main modification is to handle inverse pointers as discussed in Section 3. The write-once property is important for certain kinds of memory technology. Also, it implies that any data structure of constant bounded in-degree that can be built using the augmented LISP primitives *cons*, *cdr*, *replaca*, and *replacd* can be simulated in linear time using only the pure LISP primitives *cons*, *car*, and *cdr*. Thus the result sheds light on the power of purely applicative programming languages.

Among open problems related in our work, the following four are especially significant:

(i) For the general methods of Sections 2 and 3, find a way to make the time and space per update step $O(1)$ in the worst case instead of in the amortized case. The issue is whether node copying or splitting can somehow be delayed or otherwise modified so that there is $O(1)$ node copying or splitting in the worst case per update step.

(ii) Find a way to allow update operations that combine two or more old versions of the structure. This would allow concatenation of lists, for example. The difficulty here is that the navigation problem becomes much more difficult; in the fully persistent case, the version tree of Section 3 becomes a directed acyclic graph. (The predecessors of each version are the versions from which it is formed.)

(iii) Find a way to allow update operations that change many versions simultaneously. For example, consider the case of multiple versions of a binary search tree, indexed by integers, in which each insertion or deletion affects all versions with indices in an interval whose endpoints are parameters of the update operation. The dynamization techniques of Bentley and Saxe [2] suggest an approach to this problem that works reasonably well in the case of insertions, but deletions seem to be much harder to handle.

(iv) Find a more efficient way than the fat node method to make linked structures of unbounded in-degree persistent. The node-copying and node-splitting methods completely break down in this case; the fat node method needs only $O(1)$ space per update step but increases access times by a logarithmic factor.

Acknowledgment

We thank Nick Pippenger for noting the connection between write-once data structures and the power of pure LISP.

References

1. R. Bayer and E. McCreight, Organization of large ordered indexes, *Acta Inform.* **1** (1972), 173–189.
2. J. L. Bentley and J. B. Saxe, Decomposable searching problems I: Static-to-dynamic transformations, *J. Algorithms* **1** (1980), 301–358.
3. N. Blum and K. Mehlhorn, On the average number of rebalancing operations in weight-balanced trees, *Theoret. Comput. Sci.* **11** (1980), 303–320.
4. M. R. Brown and R. E. Tarjan, Design and analysis of a data structure for representing sorted lists, *SIAM J. Comput.* **9** (1980), 594–614.
5. B. Chazelle, Filtering search: A new approach to query-answering, *SIAM J. Comput.* **15** (1986), 703–724.
6. B. Chazelle, How to search in history, *Inform. and Control* **77** (1985), 77–99.
7. B. Chazelle and L. J. Guibas, Fractional cascading: A data structuring technique with geometric applications (extended abstract), *in* "Automata, Languages, and Programming, 12th Colloquium" (W. Bauer, Ed.), Lecture Notes in Computer Science Vol. 194, pp. 90–100, Springer-Verlag, Berlin, 1985.

8. B. CHAZELLE AND L. J. GUIBAS, Fractional cascading I: A data structuring technique, *Algorithmica* **1** (1986), 138–162.
9. R. COLE, Searching and storing similar lists, *J. Algorithms* **7** (1986), 202–220.
10. P. DIETZ, Maintaining order in a linked list, *in* "Proceedings, 14th Annual ACM Symp. on Theory of Computing, 1982, pp. 62–69.
11. P. DIETZ AND D. D. SLEATOR, Two algorithms for maintaining order in a list, in "Proceedings, 19th Annual ACM Symp. on Theory of Computing, 1987," pp. 365–372.
12. D. P. DOBKIN AND J. I. MUNRO, Efficient uses of the past, *in* "Proceedings, 21st Annual IEEE Symp. on Foundations of Computer Science, 1980," pp. 200–206.
13. L. J. GUIBAS AND R. SEDGEWICK, A dichromatic framework for balanced trees, *in* "Proceedings, 19th Annual IEEE Symp. on Foundations of Computer Science, 1978," pp. 8–21.
14. R. HOOD AND R. MELVILLE, Real-time queue operations in pure LISP, *Inform. Process. Lett.* **13** (1981), 50–54.
15. S. HUDDLESTON AND K. MEHLHORN, Robust balancing in *B*-trees, *in* "Theoretical Computer Science VII" (P. Deussens, Ed.), Lecture Notes in Computer Science Vol. 104, pp. 234–244, Springer-Verlag, Berlin, 1981.
16. S. HUDDLESTON AND K. MEHLHORN, A new data structure for representing sorted lists, *Acta Inform.* **17** (1982), 157–184.
17. S. HUDDLESTON, "An Efficient Scheme for Fast Local Updates in Linear Lists," Dept. of Information and Computer Science, University of California, Irvine, CA, 1981.
18. S. R. KOSARAJU, Localized search in sorted lists, *in* "Proceedings, 13th Annual ACM Symp. on Theory of Computing, 1981," pp. 62–69.
19. T. KRIJNEN AND L. G. L. T. MEERTENS, "Making *B*-Trees Work for *B*," IW 219/83, The Mathematical Centre, Amsterdam, The Netherlands, 1983.
20. D. MAIER AND S. C. SALVETER, Hysterical *B*-trees, *Inform. Process. Lett.* **12** (1981), 199–202.
21. E. W. MYERS, "AVL Dags," TR 82-9, Dept. of Computer Science, The University of Arizona, Tucson, AZ, 1982.
22. E. W. MYERS, An applicative random-access stack, *Inform. Process. Lett.* **17** (1983), 241–248.
23. E. W. MYERS, Efficient applicative data types, *in* "Conf. Record Eleventh Annual ACM Symp. on Principles of Programming Languages, 1984," pp. 66–75.
24. J. NIEVERGELT AND E. M. REINGOLD, Binary search trees of bounded balance, *SIAM J. Comput.* **2** (1973), 33–43.
25. M. H. OVERMARS, Searching in the past, I, *Inform. and Computation*, in press.
26. M. H. OVERMARS, "Searching in the Past II: General Transforms," Technical Report RUU-CS-81-9, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1981.
27. T. REPS, T. TEITELBAUM, AND A. DEMERS, Incremental context-dependent analysis for language-based editors, *ACM Trans. Program. System. Lang.* **5** (1983), 449–477.
28. N. SARNAK, "Persistent Data Structures," Ph. D. thesis, Dept. of Computer Science, New York University, New York, 1986.
29. N. SARNAK AND R. E. TARJAN, Planar point location using persistent search trees, *Comm. ACM* **29** (1986), 669–679.
30. G. F. SWART, "Efficient Algorithms for Computing Geometric Intersections," Technical Report 85-01-02, Department of Computer Science, University of Washington, Seattle, WA, 1985.
31. R. E. TARJAN, "Data Structures and Network Algorithms," Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
32. R. E. TARJAN, Updating a balanced search tree in $O(1)$ rotations, *Inform. Process. Lett.* **16** (1983), 253–257.
33. R. E. TARJAN, Amortized computational complexity, *SIAM J. Algebraic Discrete Methods* **6** (1985), 306–318.
34. A. K. TSAKALIDIS, Maintaining order in a generalized linked list, *Acta Inform.* **21** (1984), 101–112.
35. A. K. TSAKALIDIS, AVL-trees for localized search, *Inform. and Control* **67** (1985), 173–194.
36. A. K. TSAKALIDIS, "An Optimal Implementation for Localized Search," A84/06, Fachbereich Angewandte Mathematik und Informatik, Universität des Saarlandes, Saarbrücken, West Germany, 1984.