Reprinted from JOURNAL OF COMPUTER AND SYSTEM SCIENCES
All Rights Reserved by Academic Press, New York and London

Vol. 38, No. 1, February 1989
Printed in Belgium

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

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