Must read research papers on Data Structures

Data Structures are not seen to be as important as Algorithms but in reality, it is equally important to solve computational problems efficiently. Here are the list of must read research papers on Data Structures.

Contents

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.

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

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-

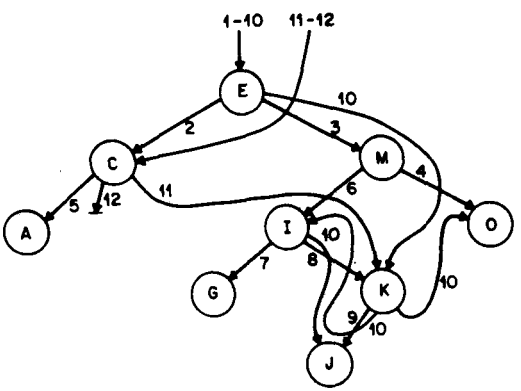


FIG. 1. A partially persistent search tree built using the fat node method, for the sequence of update operations consisting of insertions of E, C, M, O, A, I, G, K, J, followed by deletion of M, E, and A. The "extra" pointers are labeled with their version stamps. Left pointers leave the left sides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

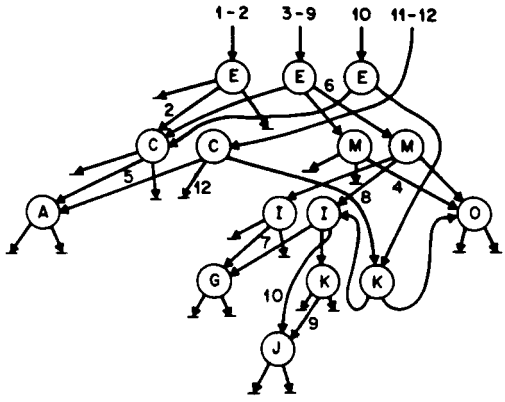


FIG. 2. A partially persistent search tree built using the node-copying method, for the same sequence of update operations as in Fig. 1. The version stamps of nodes and the copy pointers are omitted, since they are unnecessary.

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

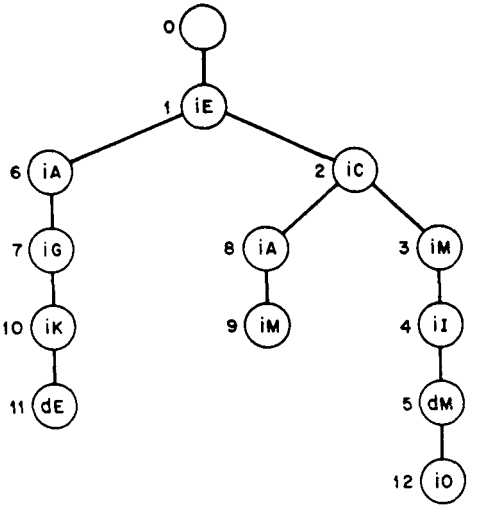


FIG. 3. A version tree. Each node represents an update operation; an "i" or "d" indicates an insertion or deletion of the specified item. The nodes are labeled with the indices of the corresponding operations. The version list is 1, 6, 7, 10, 11, 2, 8, 9, 3, 4, 5, 12.

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.

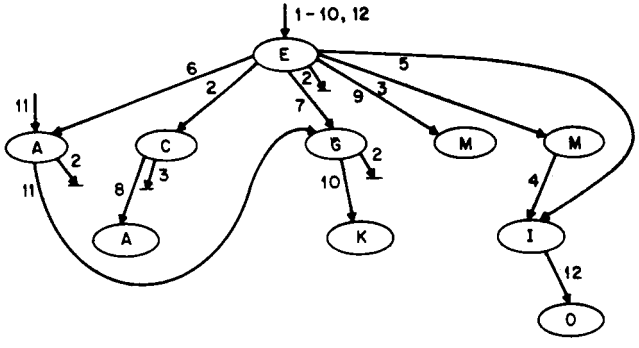


FIG. 4. A fully persistent search tree built using the fat node method, for the update sequence in Fig. 3. The version stamps and original null pointers of nodes are omitted as unnecessary.

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

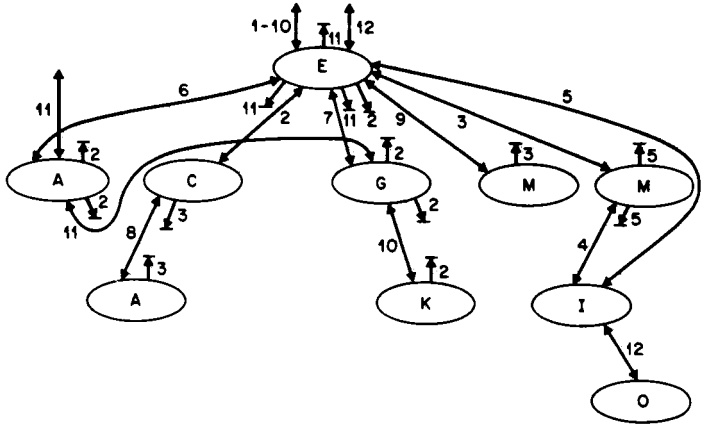


FIG. 5. A fully persistent search tree with inverse (parent) pointers built using the fat node method, for the update sequence in Fig. 3. Parent pointers leave the tops of nodes. Each double-headed arrow denotes two pointers, one the inverse of the other, with the same version stamp. The version stamps 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 $e \geq k$. 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

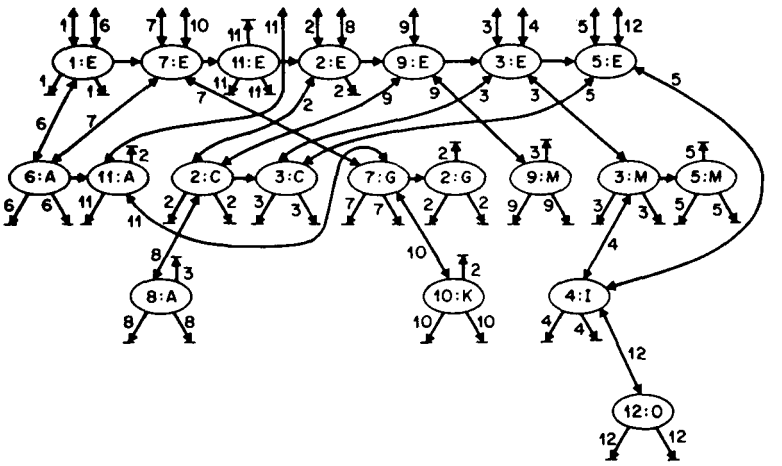


FIG. 6. A fully persistent search tree with inverse (parent) pointers built using the node-splitting method, for the update sequence in Fig. 3. Each pointer is numbered with its version stamp; the nodes are also numbered with their version stamps. The copy pointers leave the right sides of nodes and 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 $e \geq k$.) 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

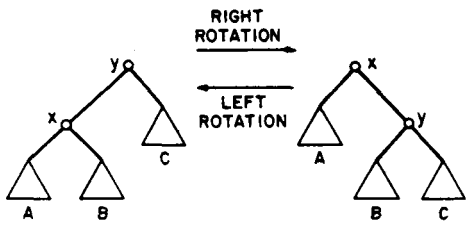


FIG. 7. A rotation in a binary tree. The tree shown can be a subtree of a larger tree.

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

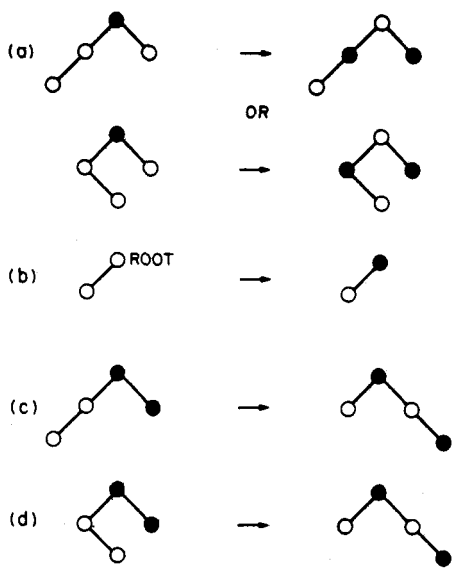


FIG. 8. The rebalancing transformations in red–black tree insertion. Symmetric cases are omitted. Solid nodes are black; hollow nodes are red. All unshown children of red nodes are black. In cases (c) and (d) the bottommost black node shown can be missing.

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

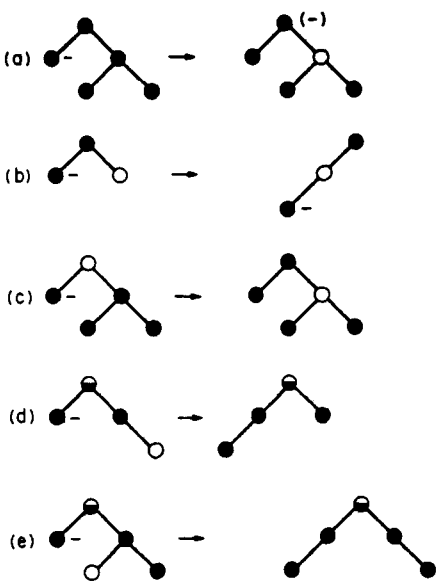


FIG. 9. The rebalancing transformations in red–black tree deletion. The two ambiguous (half-solid) nodes in (d) have the same color, as do the two in (e). Minus signs denote short nodes. In (a), the top node after the transformation is short unless it is the root.

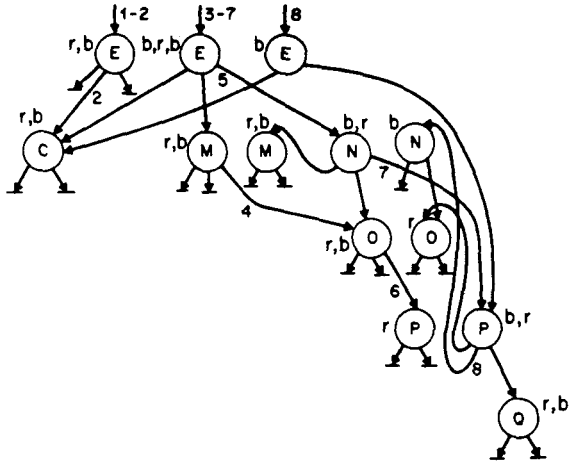


FIG. 10. A partially persistent red–black tree built using the node copying method of Section 2. The sequence of operations consists of insertions of E, C, M, O, N, P, Q, followed by deletion of M. Extra pointers are labeled with their version stamps. Each node is labeled with its successive colors.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

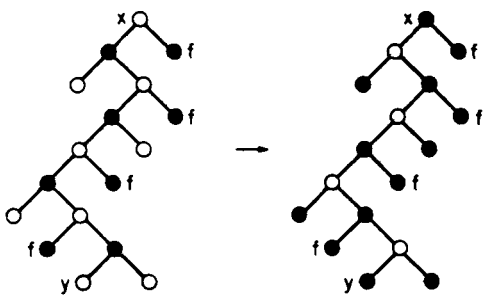


FIG. 11. An insertion interval with top node x and bottom node y, showing the desired recoloring. The nodes marked "f" are the fringe nodes of the interval. The interval is represented by storing in node x the item in node y (which is also still stored in y) along with a flag indicating that x is the 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 $O(1)$.

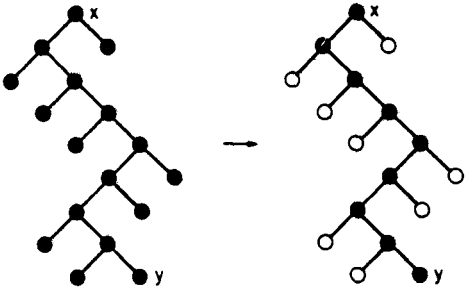


FIG. 12. A deletion interval with top node x and bottom node y, showing the desired recoloring. The interval is represented by storing in node x the item in node y along with a flag indicating that x is the top of a deletion interval.

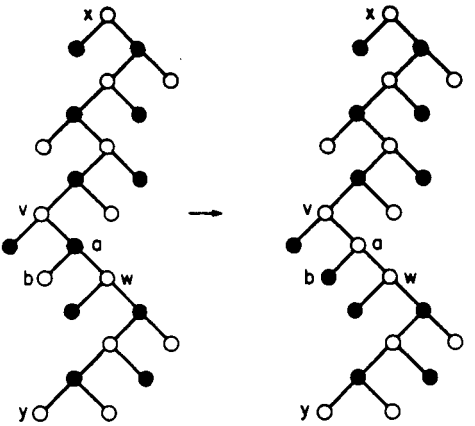


FIG. 13. Splitting an insertion interval. The nodes in the interval have their uncorrected colors. Recoloring the nodes a and b splits the insertion interval from x to y into two insertion intervals, one from x to v and one from w to y.

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

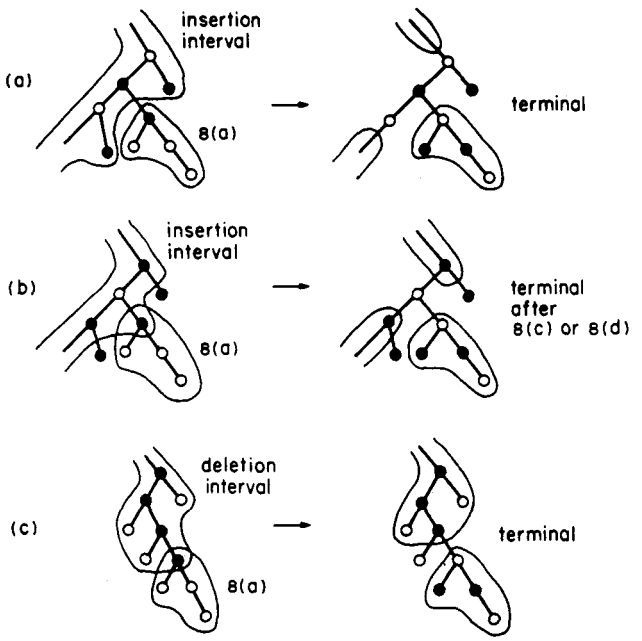


FIG. 14. The ways a propagating insertion can run into a recoloring interval. Recoloring intervals are shown correctly colored. In each case the transformation of Fig. 8a is to be applied. (a) Recoloring a fringe node. This terminates the insertion. The insertion interval splits. (b) Recoloring a node 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).

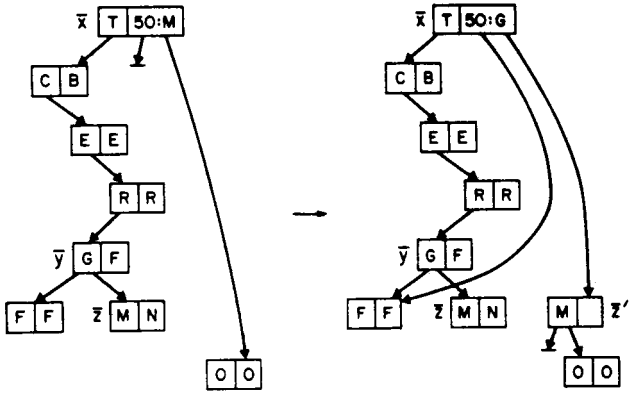


FIG. 15. Shrinking a displacement path. Irrelevant fields of nodes are omitted; all unshown version stamps of change records are assumed to be unrelated to 50 in the version tree. The displacement path with head $\bar{x}$ and tail $\bar{z}$ is shrunk by making a new copy $\bar{z}'$ of $\bar{z}$. The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

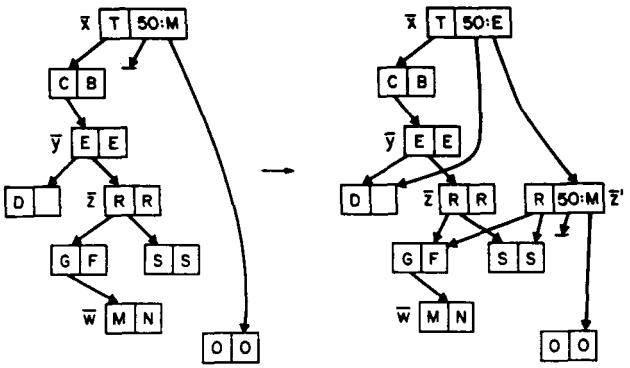


FIG. 16. Splitting a displacement path. Irrelevant fields of nodes are omitted; all unshown version stamps of change records are assumed to be unrelated to 50 in the version tree. The displacement path with head $\bar{x}$ and tail $\bar{w}$ is split at $\bar{y}$ by making a new copy $\bar{z}'$ of $\bar{z}$.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

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium

12

Fractional Cascading

Bernard Chazelle and Leonidas J. Guibas

June 23, 1986

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center and two other corporate research laboratories are committed to filling that need.

SRC opened its doors in 1984. We are still making plans and building foundations for our long-term mission, which is to design, build, and use new digital systems five to ten years before they become commonplace. We aim to advance both the state of knowledge and the state of the art.

SRC will create and use real systems in order to investigate their properties. Interesting systems are too complex to be evaluated purely in the abstract. Our strategy is to build prototypes, use them as daily tools, and feed the experience back into the design of better tools and the development of more relevant theories. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

During the next several years SRC will explore applications of high-performance personal computing, distributed computing, communications, databases, programming environments, system-building tools, design automation, specification technology, and tightly coupled multiprocessors.

SRC will also do work of a more formal and mathematical flavor; some of us will be constructing theories, developing algorithms, and proving theorems as well as designing systems and writing programs. Some of our work will be in established fields of theoretical computer science, such as the analysis of algorithms. computational geometry, and logics of programming. We also expect to explore new ground motivated by problems that arise in our systems research.

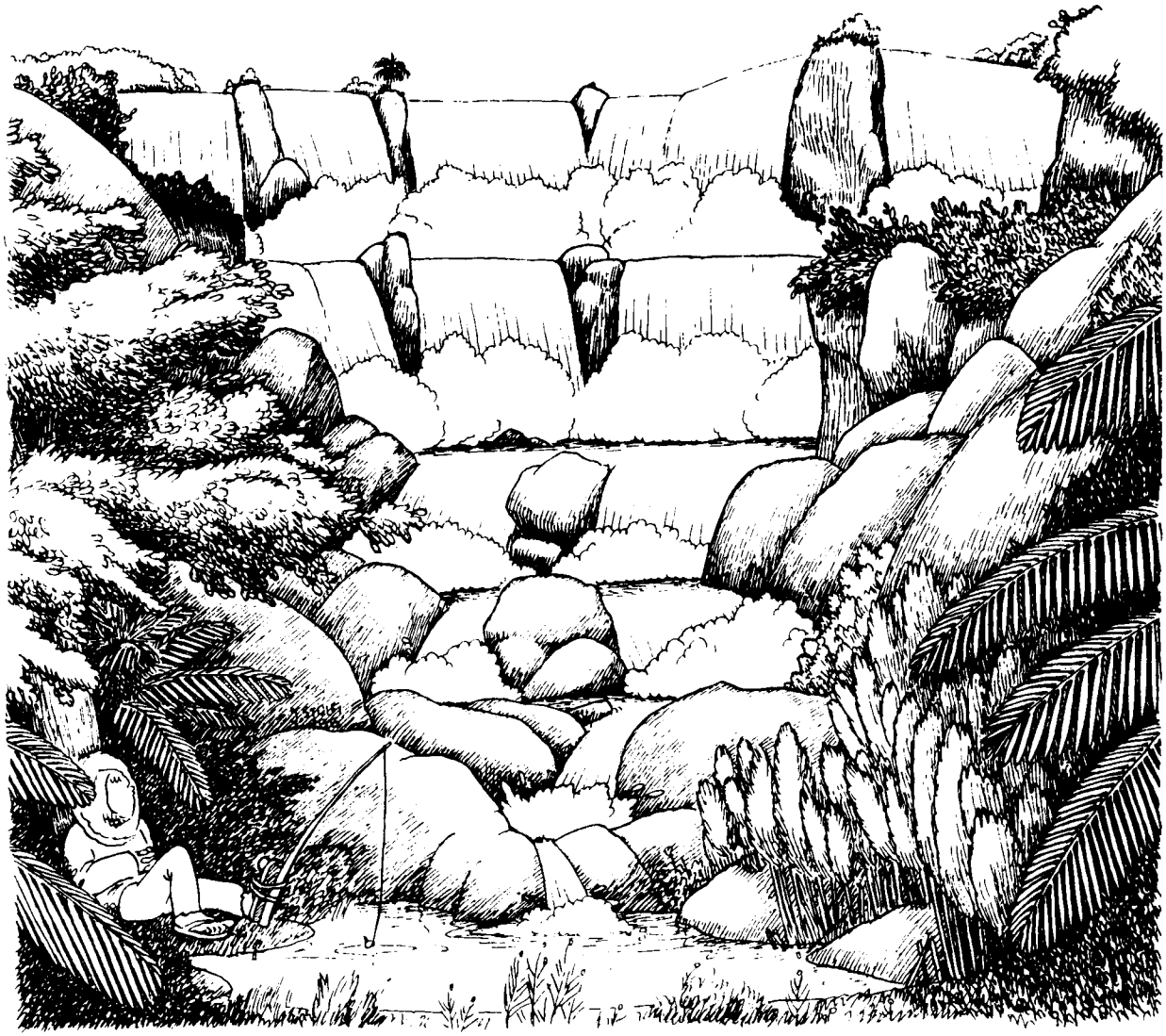
DEC is committed to open research. The improved understanding that comes with widespread exposure is more valuable than any transient competitive advantage. SRC will freely report results in conferences and professional journals. We will actively seek users for our prototype systems among those with whom we have common research interests. We will encourage visits by university researchers and conduct collaborative research.

Robert W. Taylor, Director

Fractional Cascading

Bernard Chazelle and Leonidas J. Guibas

June 23, 1986



Publication history

An earlier version of this report appeared in the Proceedings of 12th ICALP Colloquium, 1985, 90–100, and was published as Lecture Notes in Computer Science, 194, by Springer-Verlag, 1985. This material will also appear in Algorithmica.

Acknowledgements

Bernard Chazelle is currently on leave of absence from Brown University at Ecole Normale Supérieure. He was supported in part by NSF grants MCS 83-03925 and the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786. Part of this work was done while the second author was employed by the Xerox Palo Alto Research Center. Contact author's address: Leonidas J. Guibas, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, Ca. 94301.

Copyright and reprint permissions

Authors' abstract

In computational geometry many search problems and range queries can be solved by performing an iterative search for the same key in separate ordered lists. In Part I of this report we show that, if these ordered lists can be put in a one-to-one correspondence with the nodes of a graph of degree d so that the iterative search always proceeds along edges of that graph, then we can do much better than the obvious sequence of binary searches. Without expanding the storage by more than a constant factor, we can build a data-structure, called a *fractional cascading structure*, in which all original searches after the first can be carried out at only $\log d$ extra cost per search. Several results related to the dynamization of this structure are also presented. Part II gives numerous applications of this technique to geometric problems. Examples include intersecting a polygonal path with a line, slanted range search, orthogonal range search, computing locus functions, and others. Some results on the optimality of fractional cascading, and certain extensions of the technique for retrieving additional information are also included.

Keywords: binary search, B-tree, iterative search, multiple look-up, range query, dynamization of data structures, fractional cascading.

Bernard Chazelle and Leonidas J. Guibas

Capsule review

Suppose we have to search the same key in several sorted lists, each of size n. The obvious approach — perform a binary search in each list — requires $O(\log n)$ operations for each list. Fractional cascading is a method of cross-linking those lists in such a way that the $O(\log n)$ cost of binary search has to be paid only once: to locate the key in one of the lists. The cross-links then allow the key to be located in each additional list with only a constant number of operations.

The first part of the paper describes algorithms for the construction, use, and updating of the fractional cascading structure. The total number of cross-links (and the time required to build them) is proved to be linear in the total size of the lists. The second part shows how fractional cascading can be used to reduce the theoretical complexity of several geometric search problems.

The bias towards geometrically flavored examples here reflects the authors' background and interests, and not any intrinsic limitation of the technique. Fractional cascading is a purely combinatorial data structuring method, and it will certainly be of great value in many other areas.

Jorge Stolfi

Contents

Fractional Cascading: I
A Data Structuring Technique

1. Introduction

This paper introduces a new data structuring technique for improving existing solutions to retrieval problems. For illustrative purposes, let us consider the following three classical problems in computational geometry:

(a) Given a collection of intervals on the line, how many of them intersect an arbitrary query interval?

(b) Given a polygon P, which sides of P intersect an arbitrary query line?

(c) Given a collection of rectangles, which of them contain an arbitrary query point?

What do these problems have in common? Except that they each fall into the broader class of *geometric retrieval problems*, little seems to relate them together in one way or the other. Yet, we can speed up the best algorithms known for solving these problems using a single common technique, which we call *fractional cascading*. This novel technique is general enough to speed up the solutions not only of these three problems but of a host of others; we will give numerous examples in part II of this paper.

In a nutshell, fractional cascading is an efficient strategy for dealing with the following problem, termed *iterative search*: let G be a graph whose vertices are in one-to-one correspondence with a set of sorted lists; given a query consisting of a key q and a subgraph π of G, search for q in each of the lists associated with the vertices of π. This problem has a trivial solution involving repeated binary searches. Fractional cascading establishes that it is possible to do much better: under some weak assumptions, we show that with only linear space it is possible to organize the set of lists so that all the searches can be accomplished in optimal time, at roughly constant cost per search.

As the second part of this paper amply demonstrates, iterative search is a fundamental component of many query-answering algorithms. Let us take Problem (c), for instance: *given a collection of rectangles, which of them contain an arbitrary query point*? The data structure for this problem with the most efficient asymptotic performance [Ch1] is a complete binary tree whose nodes point to auxiliary lists. Answering a query involves tracing a path in the tree, while searching for a given value (one of the coordinates of the query point) in *each* auxiliary list associated with the nodes visited on the path. Here, as well as in many other algorithms for retrieval problems, iterative search is the main computational bottleneck. For this reason, it is desirable to treat the problem in an abstract setting, so the results obtained can be directly applied to as many problems as possible.

Following this approach, we present an optimal solution to iterative search, which we then apply to a number of retrieval problems. By doing so, we are able to improve upon a host of previous complexity results. It is worth noting, and this will become even more apparent when we go into applications of fractional cascading, that this technique can be usefully thought of as a postprocessing step that can be applied to speed up already existing solutions of various problems.

Part I of this paper describes and analyzes fractional cascading in a general setting. We present and discuss the construction of the data structure, its use for query-answering, and the issues involved in making our solution dynamic. In part II we present a number of specific applications of the technique, and examine the complexity of iterative search in the light of fractional cascading. The two parts can be read almost independently of each other. Only Section 2.1 of this part, which introduces the basic concepts and presents the main results, is necessary for reading the second part.

2. The Fractional Cascading Technique

In this section we present a static description of what the fractional cascading structure is and how it can be used to solve the iterated search problem.

2.1. Preliminaries: Setting the Stage; Summary of the Main Result

We consider a fixed graph $G = (V, E)$ of $|V| = n$ vertices and $|E| = m$ edges. The graph G is undirected and connected, and contains no loops or multiple edges. In addition to this classical graph structure, we have associated with each vertex v of G a catalog C_v, and associated with each edge e a range R_e.

A *catalog* is an ordered collection of records, where each record has an associated value in the set $\Re \cup \{-\infty, +\infty\}$. The records are stored in the catalog in non-decreasing order of their value; note that different records may contain the same value. A catalog is never empty: it always contains one record with value $-\infty$ and one record with value $+\infty$. These special records play the role of sentinels so as to simplify the algorithms.[1] A *range* is simply an interval of the form $[x, y]$, $[-\infty, y]$, $[x, +\infty]$, or $[-\infty, +\infty]$. In all cases, it is specified by two endpoints chosen from the linear order. We will refer to our graph G, together with the associated catalogs and ranges, as a *catalog graph*. This is the combinatorial structure to which fractional cascading can be applied.

For notational convenience we make the following assumption: if value K is an endpoint of the range $R_{(u,v)}$ associated with edge (u, v), then K appears as the value of some record in both catalogs C_u and C_v. In fact, if two ranges $R_{(u,v)}$ and $R_{(v,w)}$ have an endpoint in common, its value will appear *twice* in the catalog C_v of their shared vertex v. This requirement does not in any way restrict the generality of our discussion and, since G is connected, it provides a notational advantage. Indeed the space required to store a catalog graph is proportional to the total size of its catalogs. If $s = \sum_{v \in V} |C_v|$, then the $O(m + n)$ storage required to represent the graph structure itself, plus the storage for all the sorted multisets which are the catalogs, plus that for the intervals which are the ranges, adds in total to $O(s)$.

Next, we give three definitions to introduce some basic concepts. We start with a notion related to the degree of the vertices because, as we will see, the performance of our data structure will be very sensitive to high degrees, and more accurately, to high *local degrees*.

Definition 1. A catalog graph is said to have *locally bounded degree d* if for each vertex v and each value $x \in \Re$ the number of edges incident on v whose range includes x is bounded by d.

Note that if G has bounded degree it also has locally bounded degree, but the converse is not true in general. From now on, unless specified otherwise, we will assume that G has locally bounded degree d. The next definition formalizes the intuitive notion of enumerating the vertices of a subgraph in a "connected" way. The one after that makes precise the type of query underlying the notion of iterative search.

Definition 2. A *generalized path* π in G is a sequence of vertices $v_1, v_2, \ldots, v_p$ and corresponding edges $e_2, \ldots, e_p$ such that for each vertex v_i, $i > 1$, the edge e_i connects v_i to a vertex v_j of the path, with $j < i$.

[1] Our assumption that the values are real numbers is only for notational convenience; any linearly ordered set will do.

Since our graph G is connected, it is obvious that there exist permutations of V that are generalized paths of G. In general, any connected subgraph of G gives rise to a generalized path.

Definition 3. A *multiple look-up query* is a pair (x, π), where x is a key value in $\Re$ and π is a generalized path of G. The value x *must* fall within the range of every edge of π. The path π may be specified *on-line*, in other words, one edge at a time.

For a catalog C we will denote by $\sigma(x, C)$ the first record in C whose value is greater than or equal to x; we will call the value of this record the *successor* of x in C. Computing this value is equivalent to locating x in the sorted multiset of values represented by C. The main subject of this work, the *iterative search problem*, can now be formally stated as follows:

> Given a multiple look-up query (x, π), look up x successively in the catalogs C_v associated with each vertex v of π, and in each case report $\sigma(x, C_v)$. If π is given on-line, then the reporting is to be done on-line as well.

The problem which we are confronting is to preprocess a catalog graph G, along with its associated catalogs and ranges, so as to answer any multiple look-up query efficiently. If we do no preprocessing whatsoever, the catalog graph takes up $O(s)$ space, as previously observed. In order to answer a particular query, we look up x in each catalog along π. If this is done by using binary search in each catalog, the total reporting cost will be $O\left(\sum_{i=1}^{p} \log(|C_{v_i}|)\right)$, where the sum is over all vertices of π.

The strategy adopted by fractional cascading is to do only one binary search at the beginning, and then, as each vertex v of π is specified, locate x in C_v with an additional effort that only depends on d (the locally bounded degree). If for simplicity we assume that each catalog has the same size c, and that d is a constant, then fractional cascading reduces the query time from $O(p \log c)$ in the naive method to $O(p + \log c)$. Of course if the catalogs to be queried are unrelated, then knowing the position of x in one catalog might not help to locate it in its neighboring catalogs. So fractional cascading has to build auxiliary structures that correlate these catalogs.

One way to attain query time additive in $\log c$ and p is to merge all the catalogs into a master catalog M, and then for each catalog C to keep a correspondence dictionary between positions in C and positions in M. If we do this, we can look up x in M once and for all when a query is specified, and subsequently, for each vertex of π, simply follow the appropriate correspondence dictionary to locate x in the catalog of that vertex in constant time. Unfortunately the correspondence dictionaries altogether take up space $\Omega(n \sum_{v \in V} |C_v|)$, which is not $O(s)$. For example, in the special case considered above, the storage required grows from optimal $\Theta(nc)$ with the naive method, to $\Theta(n^2 c)$ when the master catalog is used. An important accomplishment of fractional cascading is that it attains the query time claimed while still keeping the overall storage linear.

A side remark is appropriate here: the reason the edges of G have been assigned ranges is to make fractional cascading more general and unifying. If G has bounded degree, however, the notion of ranges becomes irrelevant and the requirement "x must fall within the range of every edge of π" (Definition 3) can be dropped altogether, as each range can be taken to be $[-\infty, +\infty]$. The range enhancement is not gratuitous; it will come in very handy in some of the applications treated later on. Now, before embarking on a fairly long technical development, let us summarize our main result concerning fractional cascading, as will be proven in Sections 3 through 5.

Theorem S. *Let G be a catalog graph of size s and locally bounded degree d. In $O(s)$ space and time, it is possible to construct a data structure for solving the iterative search problem. The structure allows multiple look-ups along a generalized path of length p to be executed in time $O(p \log d + \log s)$. If d is a constant, this is optimal.*

So far we have dealt only with static catalogs. In many applications, however, allowing insertions and deletions of records into or from these catalogs is necessary. Thus Section 7 investigates how fractional cascading can be made dynamic. The results we have obtained there are less conclusive:

Theorem D. *The fractional cascading data structure can be made dynamic with the following bounds: If only insertions and look-ups are performed, the amortized time for each insertion can be $O(\log s)$, while the look-up cost remains the same as before. Here we are amortizing over a sequence of $O(s)$ insertions. The same bounds hold for deletions and look-ups only. If intermixed insertions and deletions are desired, then each of them can still be done in $O(\log s)$ amortized time, but the time required for a query increases to $O(p \log d \log \log s + \log s)$.*

For a discussion of amortized computational complexity the reader is referred to a paper by Tarjan [Ta].

2.2. The Fractional Cascading Data Structure

There are two key goals that the fractional cascading structure must accomplish: (1) somehow correlate each pair of neighboring catalogs in the catalog graph so a look-up in one of them aids the look-up in the other, and (2) keep the overall storage linear. The former goal suggests augmenting each catalog by introducing additional records borrowed from neighboring catalogs.

2.2.1. Bridges and Gaps

Each *original* catalog C_v will be enlarged with additional records to produce an *augmented* catalog A_v, which too will be a linear list of records whose values form a sorted multiset. Exactly how this is to be done is explained in Section 3. Here we will be content simply to describe the desired state of affairs after this augmentation has occurred. A related idea has been described in [VW]. Augmented catalogs for neighboring nodes in G will contain a number of records with common values. The corresponding pairs of records will be linked together to correlate locations in the two catalogs. More formally, for each node u and edge e connecting u with v in G we will maintain a list of *bridges* from u to v, D_{uv}, which will be an ordered subset of the records in A_v having values common to both A_u and A_v and lying in the range R_e; in particular, the endpoints of R_e are the first and last records in D_{uv}. We will have a symmetric situation with node v, where we maintain, for each bridge in D_{uv}, a *companion* bridge in D_{vu}. We call D_{uv} the *correspondence dictionary* from A_u to A_v. Remember that, in order to allow for the occasional presence of duplicates, we distinguish between a record of a catalog and its value. For example, D_{uv} and D_{vu} have no record in common, although they have the same set of values. A bridge is most usefully considered as a variant record in an augmented catalog pointing to a record with the same value in a neighboring augmented catalog. Bridges respect the ordering of equal-valued records, so they never "cross".

In order to disambiguate communication between catalogs of adjacent vertices, we add the requirement that each bridge should be associated with a *unique* edge of G. This means that

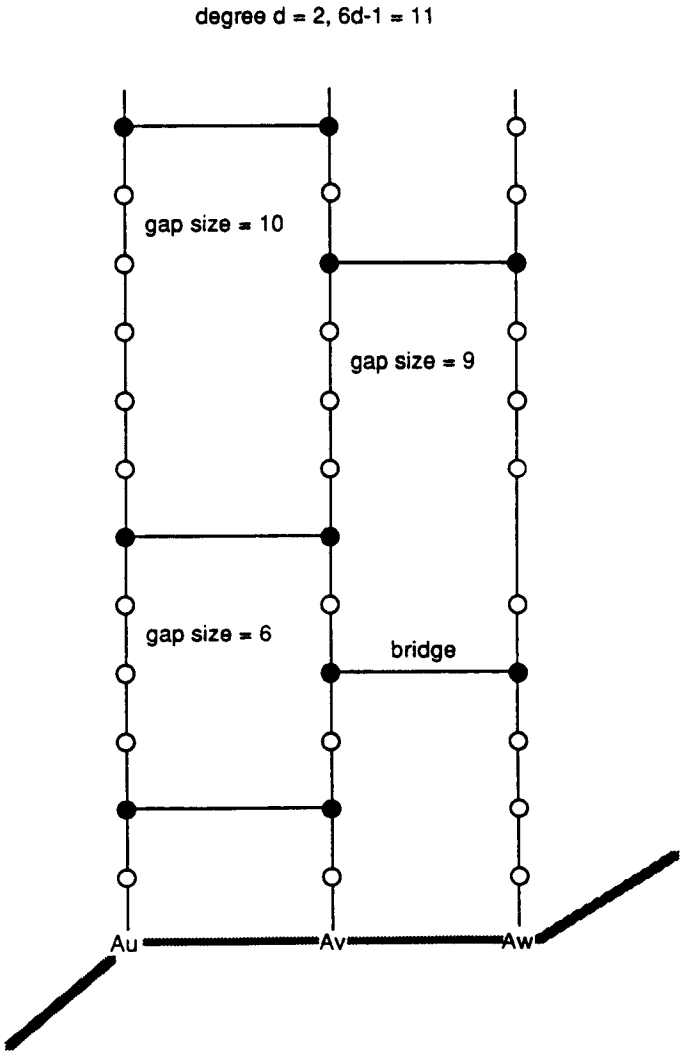


Figure 1. Bridges and gaps

if a given value in A_u is to be used to form a bridge in both D_{uv} and D_{uw} then it must be duplicated and stored in two separate records of A_u.

A pair of consecutive bridges associated with the same edge $e = (u, v)$ defines a *gap*. Let a_u and b_u be two consecutive bridges in D_{uv} and let a_v (resp. b_v) be the companion bridge of a_u (resp. b_u). Assume that b_u occurs after a_u in A_u. We form the *gap* of b_u by including into it each element of A_u positioned strictly between a_u and b_u and each element of A_v positioned strictly between a_v and b_v (a gap does *not* contain the bridges which define it). Note that the gap of b_u is the same as the gap of b_v. The element b_u (or b_v) is called the *upper bridge* of the gap. Except for the bridges formed by the endpoints of the range R_e, all other bridges associated with the edge e are both the upper bridge of some gap and the lower bridge of another. See figure 1. A key property of the structure built by fractional cascading is that gap size is kept small. This guarantees that the bridges correlating two adjacent catalogs are never too far apart. The particular constraint we maintain is:

The gap invariant: No gap can exceed $6d - 1$ in size.

We will see in Section 4 why the magic bound of $6d - 1$ has been chosen. Figure 1 illustrates the gaps and bridges of the augmented catalogs associated with three vertices on a single path. We end this subsection with some general remarks, before proceeding to the detailed description of the data structures needed for fractional cascading

The key to the design of the fractional cascading data structures is maintaining the correspondences between adjacent augmented catalogs, and between augmented catalogs and the associated original catalogs. The former facilitate the iterative search; the latter allow positions in the augmented catalogs to be translated into positions in the original catalogs. About the former correspondence and its implementation via bridges we will have a lot to say shortly in section 3.

Surprisingly, it is the latter correspondence, that between augmented and original catalogs, which becomes the bottleneck in the complexity when we need to deal with dynamic catalogs, where insertions and deletions are allowed. This is so because the records of C_v define an ordered partition of A_v into disjoint sets, each corresponding to a range of values between two successive records of C_v; by convention each such range contains its upper endpoint only. In the dynamization of the fractional cascading structures, we will need to implement insertions and deletions into both augmented and ordinary catalogs. While augmented catalog modifications clearly correspond to insertions/deletions of elements into one of the sets of the ordered partition, original catalog modifications give rise to splits and joins of adjacent sets in the partition. Thus we will need a data structure for handling the operations of *find* (what set contains a given element), *insert*, *delete*, *split*, and *join* in an ordered set partition. Maintaining the ordered set partition is an interesting data structure problem in its own right, which we will examine in Section 7.

For now we are confining our attention to building a static fractional cascading structure, so the correspondence between augmented and original catalogs can be finessed by just keeping, for each augmented catalog element, a separate pointer to indicate its successor in the associated original catalog. Formally, for a record r of an augmented catalog A_v with value x we define its *original successor* $\nu(r)$ to be $\sigma(x, C_v)$.

2.2.2. A Close-Up of the Data Structure

Original and augmented catalogs will be represented by linked-list structures. Each record in C_v consists of two one-word (used here in the generic sense of a unit of storage) fields (*key*,*up-pointer*). The *key* field contains the value of the record, while the *up-pointer* field refers to the record in C_v immediately following the current one in increasing order. The last record in this chain has a key of $+\infty$ and its pointer refers to NIL. The structure for A_v is more complex. It can be described as a doubly-linked list of records containing cross-references to the records in C_v and with additional information stored in nodes that are bridges.

A record in A_v consists of five fields; four of these are each one word long. We assume that a word is large enough to contain a key value, or a pointer to another record, or an integer count. The fifth field is a single bit used internally by the algorithms. More specifically, the fields for a record r are:

(1) *key*: stores the value K of r.

(2) *C-pointer*: holds a pointer to $\nu(r)$, the successor of r in C_v (thus giving us a constant-time implementation of the *find* operation above).

(3) *up-pointer*: points to the next element in A_v (or NIL if last).

(4) *down-pointer*: points to the previous element in A_v (or NIL if first).

(5) *flag-bit*: a bit used during construction or update of the structure.

Bridge records need to store more specialized information, so they have the following additional five fields. These are all one word long.

(6) *prev-bridge-pointer*: if r is a bridge in D_{vw}, then this field points to the previous (lesser value) bridge in D_{vw}. A NIL pointer is used to indicate that this record is the lower endpoint of a range.

(7) *companion-pointer*: points to the companion bridge.

(8) *edge*: if r is a bridge in D_{vw}, then this field stores the label of edge vw.

(9) *count*: This field stores the number of records in A_v that belong to the gap of which r is the upper bridge. Set to 0 for the lowest bridge in a correspondence dictionary. Its sum with the corresponding count field in the companion bridge gives the gap size of this bridge.

(10) *rank*: used internally in the construction phase and during updates.

Figure 2 illustrates the data structure on a small example. Note that, aside from catalog-related information, the structure also contains a full description of the graph G because the range endpoints become bridges providing the node adjacency information. In the next section we describe how to answer an incoming query; we postpone discussion of the construction of the data structure until Section 3.[1]

2.2.3. Answering a Multiple Look-Up Query

How do we proceed to answer a multiple look-up query (x, π)? The idea is to follow the generalized path π via the bridges provided in the data structure. Each time a search is performed in an augmented catalog A_v, the result of the look-up must be carried over to the associated original catalog C_v as well. The following lemmas provide the two basic primitives needed.

Lemma 1. *If we know the position of a value x in the augmented catalog A_v, in other words a record r with the smallest value greater than or equal to x, then we can compute the position (in the same sense) of x in C_v in exactly one step.*

Proof: Use the C-field of the record to retrieve $\nu(r)$. ∎

Lemma 2. *If we know the position of a value x in the augmented catalog A_v, and $e = (v, w)$ is an edge of G such that x is in the range R_e, then we can compute the position of x in A_w in $O(d)$ time.*

Proof: From the position of $r = \sigma(x, A_v)$ in A_v follow up-pointers until a bridge is found that connects to A_w. To do so, simply check the edge-field of every bridge visited. Because of the gap invariant, such a bridge will be found within $6d$ steps. At this point, follow the bridge-pointer and traverse A_w following down-pointers until x has been located. Again because of the gap invariant, both these traversals can be accomplished in at most $6d + 2$ comparisons. ∎

[1] A structure such as the above can easily be built by following the naive approach, mentioned earlier. Construct a master catalog M by merging all catalogs together and repeating each record as many times as the degree of the vertex it came from. Then make M the augmented catalog of each vertex. We can easily choose the bridges so that each gap has size at most $2d$. The interesting task ahead will be how to avoid the blow-up in storage which this simple-minded approach implies.

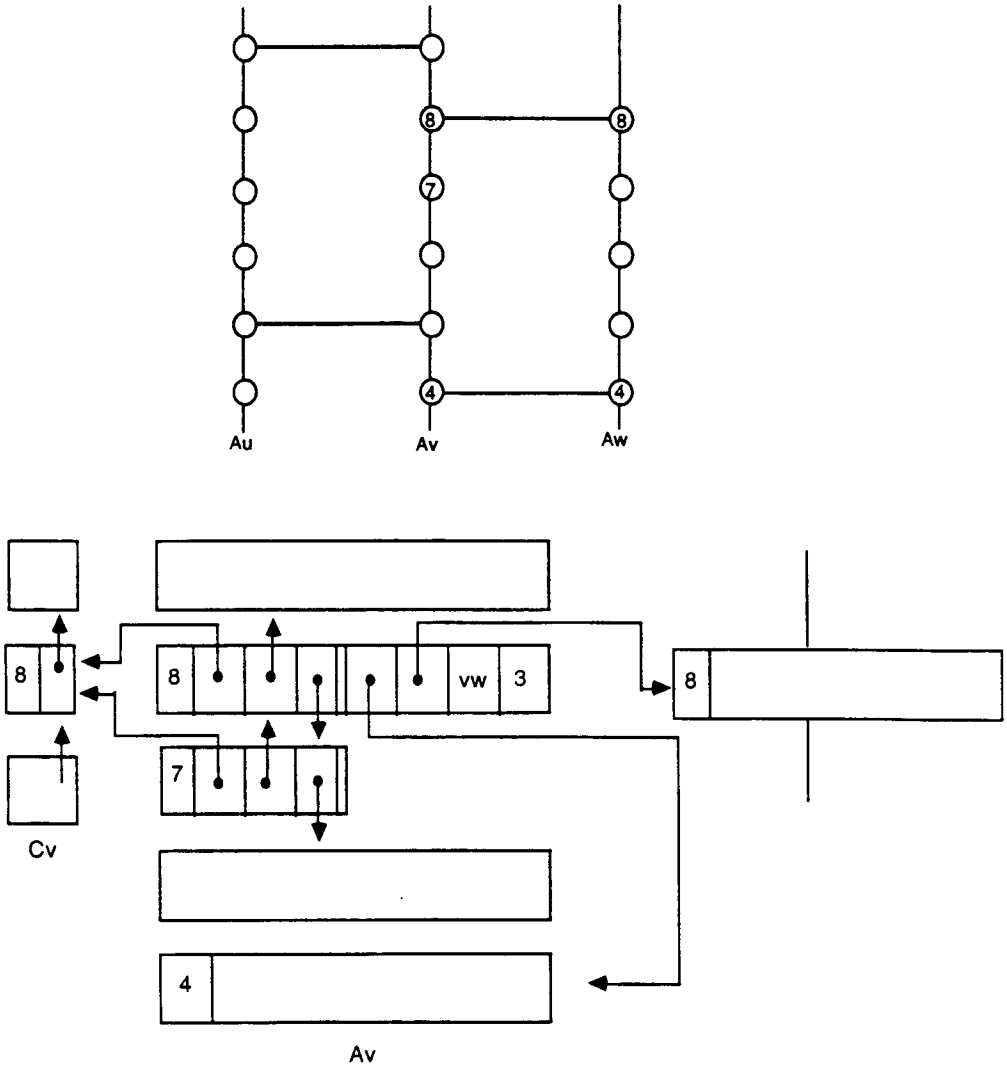


Figure 2. A close-up of the augmented catalogs

Lemmas 1 and 2 show that a multiple look-up query (x, π) can be answered very efficiently, provided that the position of x in A_f is known, where f is the first vertex in π. It is too early now to describe in detail how to compute the position of x in the initial catalog A_f. If we were to store A_v as a one-dimensional array as well, then we can certainly locate x in it in $O(\log s)$ time. However, this solution is rather inconsistent with our previous list-based structures. We will show in Section 6 that this initial search for x can be accomplished in $O(\log s)$ time using a technique which preserves the unity and simplicity of the data structure.

To summarize the situation at this point, we can handle any multiple look-up query satisfactorily, provided that the fractional cascading structures have already been built, and that efficient search is possible for the first augmented catalog to be considered. In the following section we show that the fractional cascading structure can be constructed in time $O(ds)$ and space $O(s)$. One remarkable feature of this data structure is that its size is independent of d. In the ensuing developments, d is considered a parameter and not a constant. It will therefore not disappear in the O-notation.

3. The Construction of the Fractional Cascading Structures

In order to add motivation to our discussion, we will start by describing an approach which, although flawed and ultimately inadequate, introduces the basic idea of fractional cascading in very simple terms. The reader who does not care for motivation at this point may skip the next paragraph.

Since this discussion is only for motivation, let us be concrete and assume that G is regular of degree d and each catalog C_v has size exactly c. Define a k-sample of a catalog C to be a maximal subcatalog of C obtained by taking values k apart; we call k the *sampling order*. Then A_v will be simply C_v, together with a $(2d)$-sample of each neighbor of v one away, a $(2d)^2$-sample of each neighbor two away, and so on. Here we are counting distances according to the underlying graph G. The size of A_v will be bounded by

$$c + d\frac{1}{2d}c + d^2\left(\frac{1}{2d}\right)^2 c + \cdots = 2c,$$

and thus the size of all the augmented catalogs is bounded by twice the size of the original catalogs. Any two adjacent nodes in G differ in their distances to a third node by at most ± 1. Therefore any two samples merged into the adjacent catalogs A_v and A_w may differ by a factor of at most $2d$ in the sampling order. This might make us hope that the gap invariant would also be satisfied. Unfortunately this is not necessarily the case, as the merge of two k-samples can leave gaps of size $2k$. It is this problem that makes the argument above only a heuristic and not a rigorous construction. To overcome this difficulty we must do the sampling in parallel with the construction of the augmented catalogs, as described in the sequel. Specifically, our plan will be to insert one new record at a time, maintaining the gap invariant as we go along. To ensure this, splitting some gaps into smaller gaps will occasionally be necessary. Although the time taken by a specific insertion is fairly unpredictable, the total running time of the algorithm can be made optimal with a careful implementation. Incidentally, the key idea of propagating geometrically decreasing samples of each catalog to nodes further away is responsible for the term "fractional cascading".

We now explain rigorously how, for every vertex v of G, the augmented catalogs A_v can be built efficiently. We will present the construction of the fractional cascading structures in an incremental fashion. By incremental we mean that we will show how to update these structures when a new record is added to one of the original catalogs. Starting then from a graph G with all catalogs empty, we can arrive at the desired state with repeated insertions.

The overall algorithm consists of two nested loops. For each vertex v of G in turn, we consider the elements of C_v in increasing order and insert them into A_v one at a time. Before inserting an element we make sure that *all* the gap invariants have been restored since the previous insertion. Note that even before any element of C_v has been inserted into A_v, this augmented catalog is already likely to contain elements originating from other catalogs. Therefore we must implement the insertion by merging C_v into A_v. Each insertion of a given element of C_v may cause serious changes in A_v, as well as in other augmented catalogs, necessitated by the restoration of gap invariants. The total cost of these operations, however, will be at most proportional to the final size of all the augmented catalogs.

3.1. Adding a New Record

We will partition the processing required when inserting a new record into three stages. In stage 1 we simply insert the new record r into the appropriate place in its augmented catalog A_v. After such an insertion we must update the count-fields of all gaps containing r and then split excessive gaps into smaller ones. These splits will cause additional insertions in neighboring catalogs, so count-fields must be checked again, and so forth. The counting of gap sizes and the splitting of excessive gaps constitute respectively stages 2 and 3. We may

need to loop around stages 2 and 3 several times, but eventually all gap invariants will be restored and this process will terminate. We now describe these operations in detail.

Stage 1: *Insert new record* — Let p be the next record from C_v to be inserted into A_v. Recall that p may possibly be the endpoint of a range. Let r' be the record from C_v previously inserted into A_v (or the first record of A_v, if none has been). Starting from r', follow the up-pointers of A_v until the correct position of p has been found. At this point, insert a copy r of p into A_v (breaking ties arbitrarily). The initialization of the first five fields is straightforward; the flag bit is set to 0. We also add a pointer to r into a set of newly inserted records, called the *count-queue*. When the previous insertion was fully processed, the count-queue became empty, so now its only element is r.

Stage 2 is invoked next to update the count-fields. In the general situation the count-queue will contain references to several new records created by the gap splitting process of stage 3.

Stage 2: *Update count fields* — Our task is to find all gaps containing each of the records referenced by the count-queue and update their count-fields. A simple solution consists of identifying these gaps, and then traversing each of them in order to evaluate their current size. The difficulty with this method is that gaps can grow to be very large and these repeated traversals can be costly. It is not so obvious how such a bad situation can arise, but appendix A shows that it really does. This forces us to use a cleverer method, which is described below.

We process the pointers in the count-queue twice. In the first traversal we identify the maximal groups of new records belonging to the same augmented catalog such that no two consecutive new records in the group are further than $6d$ apart. These groups are called *clusters*. Note that no gap can contain new elements in a given augmented catalog that come from more than one cluster. In the next traversal we visit each cluster and update the count-fields of the bridge records covered by the cluster. If some gap sizes have overflowed, then these gaps are added to the *wide-gap-queue*, which forms the input to stage 3. In more detail, the traversals work as follows:

First Traversal: For each reference to a new record in the count-queue go to that record and walk $6d-1$ steps down from it in its augmented catalog. In the process mark the $6d$ records thus visited by setting their flag bit to the value 1. In each augmented catalog the maximal runs of records with flag bits set to 1 define the clusters discussed above.

Second Traversal: Now visit every reference in the count-queue once more, this time removing each reference from the queue as it is processed. If a reference points to a record r, in (say) A_v, with its flag bit set to 0 then do nothing: the cluster of A_v in which that record belongs has already been taken care of. Otherwise we must process that cluster. As long as we see records with their flag bit set to 1, we walk down A_v from r to the last such record, or to the bottom of the catalog, whichever comes first. Let p denote the bottom record thus identified. We next walk up from p and in the process update the count fields of all bridges in A_v whose gaps contain new records in the cluster of r. We call this the *ranking process*.

The ranking process proceeds from p up to $6d$ steps past the last record encountered whose flag bit is set to 1. A running count of the records visited during the ascent is maintained, called the *rank*. We start out by giving p rank 1. Whenever we come to a bridge b we take a number of actions. First we store in the rank field of b the current value of this count. By following the prev-bridge-pointer of b to b' and looking at the rank field of that record, we can compute the number j of records from A_v currently belonging to the gap of b. Let i be the current value of the count-field of b, and k the count-field of the companion bridge of b. In general $j > i$ and the gap of b has increased in size from $i+k$ to at least $j+k$ ($j+k$ need not be the true size since the other side of the gap might not have been ranked yet). We now set the count-field of b to j and, if $j+k \geq 6d$

but $i + k < 6d$, then add the gap of b to the wide-gap queue for splitting during stage 3. The conditions above guarantee that a gap is added to the wide-gap queue only the first time a ranking process shows it has overflowed. Our last action in processing the bridge b is to set the rank of b' to 0. When the ranking process has reached its last record, the rank field of the last bridge encountered is also set to 0. In addition, the flag bit of each record visited in the process is set to 0—thus marking the cluster as processed.

At the end of stage 2 the count-queue is empty, all count fields of bridges are correct, and all gaps whose size exceeds $6d - 1$ have been placed in the wide-gap queue.

Stage 3: ***Restore gap invariants*** — If the wide-gap queue is not empty, remove its top element and split the gap of the upper bridge to which it points. To do so, merge all the elements of the gap into a temporary linked list. Let $K_1, \ldots, K_g$ be a labeling of this list in non-decreasing order, and let H_1 be the first group of $3d$ elements, H_2 the second group of $3d$ elements, etc. chosen from this list. Since the gap count g satisfies $g \geq 6d$, we have at least two groups, and more precisely $i = \lceil \frac{g}{3d} \rceil$ of them. If the last group H_i contains fewer than $3d$ elements, then we merge H_i and H_{i-1} together. Let j be the new number of groups ($j = i$ or $j = i - 1$). We separate H_1 from H_2 by making two copies of the largest element in H_1, each to become a bridge in the augmented catalogs associated with the gap. We then iterate on this process for the j groups, which leads to the introduction of $2(j - 1)$ bridges. All gaps produced have size exactly $3d$, except possibly for the last one, which has size $g - 3(j - 1)d \leq 6d - 1$.

Note that each partitioning element already occurs on one side of the gap. If it is not already a bridge to another neighbor on either side, then it need be duplicated only on the missing side. Otherwise it must be duplicated also on the side where it already occurs as a bridge, because of our convention that a record in an augmented catalog can only function as a bridge for a single edge. See figure 3 for an illustration of the splitting process. We omit the details of the initialization of the new records; we just mention that it is imperative to insert references to them into the count-queue.

At the end of stage 3 the wide-gap queue is empty and no gap has size exceeding $6d - 1$, according to the count fields present in the structure. All new records created from the splitting are referenced in the count-queue.

We now recapitulate the basic flow of operations. Stage 1 is called to insert a new key. At this point, stage 2 updates all count fields. Stage 3 is then called to restore the gap invariants. At termination, all gaps will have acceptable size, *if we discount the new elements that stage 3 has created.* To remedy this discrepancy, we call stage 2 again to obtain the list of flawed gaps. Stage 3 is then invoked to fix them, and the process iterates in this way until stage 2 fails to reveal any flawed gaps. It is important to ensure that stage 2 and stage 3 operate completely separately. All count fields must be correct before restoring any gap invariant and all gaps must be valid (up to the discrepancies caused by newcomers) before stage 2 is called again into action.

3.2. Proof of Correctness

Why is this process correct, and why should it always terminate? Let us leave termination aside for the time being; we first prove the two assertions made earlier: (1) after completion of stage 2, all count-fields are correct; (2) after completion of stage 3, no gap contains more than $6d - 1$ elements which were also in existence before.

We prove these assertions by induction. The second one follows directly from the description of the algorithm. Incidentally, note that after stage 3 has started, some splits may occur with a value of the count-field less than the correct one, because of earlier insertions caused by this stage. We now turn to stage 2. By the induction hypothesis (stating the correctness of the previous applications of stages 2 and 3), only the gaps containing the elements in

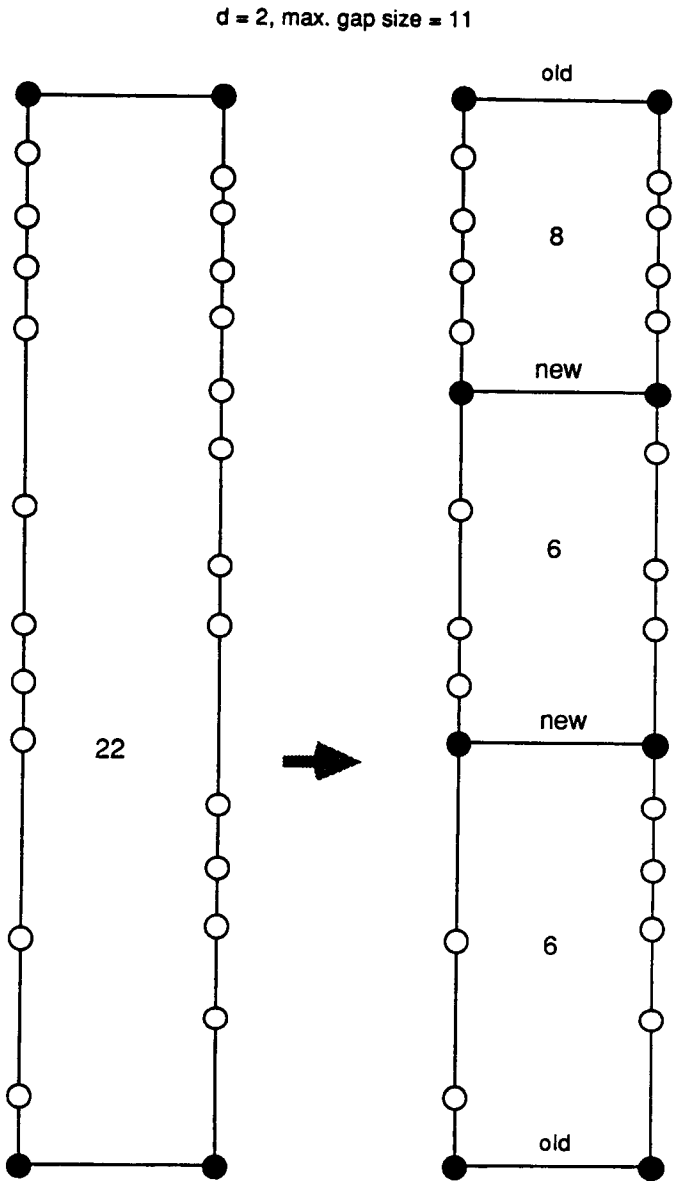


Figure 3. The gap splitting process

the count-queue need have their count-fields updated. We will first show that the updating performed in stage 2 correctly restores the counts of the gaps it touches, and then that all affected gaps are processed.

Let us concentrate our attention on the augmented catalog A_v. We call *new* any element on the count-queue just before stage 2; other elements will be called *old*. The key to the correctness of stage 2 is that no gap can contain more than $6d - 1$ *consecutive* old elements in A_v. Indeed, this would contradict the induction hypothesis that gaps were valid before the

introduction of new elements. Consequently, all new elements in A_v within a given gap must be linked together in stage 2 into one cluster, so the updating cannot miss any of them. The use of ranks is to identify exactly how many new elements lie in a given gap.

To see that the work in stage 2 is sufficient, we will prove that the algorithm does examine any gap which contains a new element. Let γ be a gap with upper bridge $K \in A_v$, and let K_u be the new element positioned highest in A_v such that K_u occurs within γ. K cannot lie more than $6d$ places above K_u in A_v (by the gap invariant), therefore K will be processed in that stage when the cluster of the new element K_u is handled. This completes the proof of correctness of our algorithm.

4. The Complexity of Fractional Cascading

4.1. Time Requirement

We now show that the insertion algorithm not only terminates, but that it does so with delay which is $O(d)$ when amortized over all insertions performed during the construction of the fractional cascading structures. In order to prove this bound we will use certain accounting techniques common in amortized complexity analysis [Ta]. The essence of those techniques is to associate "bank accounts" with parts of the data structure, into which deposits and withdrawals are made at appropriate instants during the execution of the algorithm. It is important to realize that these book-keeping operations are only an artifact of the analysis and not part of the algorithm proper.

Each gap has associated with it a ***piggy-bank*** holding some ***tokens***. A token can pay for a constant amount of computation (recall that $O(d)$ is not interpreted as "constant"). We choose this amount large enough so as to cover the actual cost in our implementation for any of the following: (1) creating a record for a new element or a new bridge and properly linking it into its augmented catalog (stages 1 and 3), (2) processing an element during the traversals designed to update the count fields in stage 2, and (3) processing an element in the merge preceding the gap splitting procedure of stage 3. Besides the piggy-banks, we have a ***cash-bank*** associated with each new element in the count-queue. We will make deposits or withdrawals from these banks in order to cover the ***restoration*** costs of an insertion: these are the costs associated with restoring the gap conditions. We will maintain the following invariant.

> Each gap of size k,$0 \leq k < 6d$, holds in its piggy-bank a number of tokens equal to at least $21 \max(0, k - 3d)$. Each new element contains $6d$ tokens in its cash-bank.

When an element K of C_v is to be inserted into A_v, it is given $27d + 1$ tokens. Twenty-one tokens go into the piggy-bank of each gap containing K. Since there are at most d such gaps, there are at least $6d + 1$ remaining tokens; K keeps $6d$ tokens for its own cash-bank, uses one token for the creation of its new record, and throws away the others. All bank conditions are then satisfied. Except for the double loop which performs the actual updating of the count fields, the time taken by stage 2 is clearly proportional to dN, where N is the number of new elements. As to the double loop, its time of execution is $O(dN + V)$, where V is the number of new elements visited during each count update. But because of the locally bounded degree condition, no new element can be examined more than d times. Therefore the total running time of stage 2 is still $O(dN)$. By our assumption about the token value, all this can be paid for with the $6d$ tokens from the cash-bank of each new element.

Processing each element G in the wide-gap queue during stage 3 takes time proportional to the size of the gap being split. Consider the new gaps produced during the splitting. We can distribute the tokens of the old piggy-bank of G into packets. The highest new gap H

is of size between $3d$ and $6d-1$; it receives a packet containing $21t$ tokens, where t is the excess of the size of H over $3d$. This packet supplies the (new) piggy-bank of H. Each of the other gaps can thus receive a packet containing $21 \times 3d = 63d$ tokens. Since, however, they all have size $3d$, their piggy-banks do not need any tokens at all. Now each new gap, except H, has to pay for the creation of one or two new bridges, as well as the necessary deposits to the piggy-banks of other gaps that the insertion of these bridges necessitates. Obviously at most $2(d-1)$ other gaps are affected. Thus we need the following: 2 tokens to create the two new bridge records; $2 \times 6d = 12d$ tokens to deposit into their cash-banks; and $21 \times 2(d-1) = 42d - 42$ tokens for deposits to other piggy-banks. Since we have a total of $63d$ tokens on hand, we can do all that and still have $9d + 40$ tokens left over.

We must still account for the work of splitting the gap G. If k denotes the size of G, then k tokens suffice to pay for splitting. Suppose that G is broken up into H, the highest gap of size between $3d$ and $6d-1$, and j other gaps, $j \geq 1$, of size exactly $3d$. By the analysis above each of the latter gaps has a surplus of $9d+40$ tokens, for a total of $(9d+40)j$. Since $(9d+40)j \geq 3jd + 6d - 1 \geq k$, we have enough to pay for the splitting out of the pooled surpluses.

In conclusion, the entire insertion process can be paid for with $(27d+1)s$ tokens (recall that s is the total catalog size) and therefore the preprocessing time of the algorithm is $O(ds)$. Next, we turn our attention to the storage utilized by the data structure.

4.2. Space Requirement

A space-token, or token for short, will buy 10 words of memory—that is, storage for one record in A_v. We take space tokens to be divisible units and divide each such token into d equal credits. We maintain the following invariant:

> At the completion of each stage, every gap of size g has at least $2\max(0, g-3d)$ credits in its (space) piggy-bank.

To handle an initial insertion (stage 1), we grant each new key three tokens. One of them covers the storage for the key itself. The other two tokens are exchanged for $2d$ credits: two of the credits are then deposited in the space piggy-bank of each containing gap; the remaining credits are thrown away. Note that this transaction preserves the piggy-bank invariant. To handle the gap splitting of stage 3, we use the packet argument of the previous section. This shows that each bridge of a newly created pair receives $6d/2 = 3d$ credits to use, after preserving all bank conditions. Two of them are deposited into the piggy-bank associated with each of the gaps containing the endpoints of the bridge. This still leaves at least $3d - 2(d-1) = d+2$ credits per bridge, which is more than one token, so the bridge can then pay for its own record. As a net result, only $3s$ tokens must be used to account for all the space used, so this space is $O(s)$. More precisely, only 30 words of memory are necessary per catalog element (on the average).

Theorem 1. *(Preliminary result) — Let G be a catalog graph of size s and locally bounded degree d. In $O(s)$ space and $O(ds)$ time, it is possible to construct a data structure for solving the iterative search problem. The structure allows multiple look-ups along a generalized path of length p to be executed in time $O(dp + \log s)$. If d is a constant, this is optimal.*

We conclude by remarking that in our $6d-1$ bound for the gap size invariant, the constant 6 can be reduced to $4+\varepsilon$, for any $\varepsilon > 0$. As it turns out, when ε goes to zero, the implied constants in our time and space analysis (in other words, the number of time or space tokens needed per insertion) go to infinity. Although the analysis breaks down for gap sizes less than

or equal to $4d$, the algorithms we have presented continue to work correctly. Figures 4 and 5 show two examples of fractional cascading structures on two simple graphs, where we in fact used $4d - 1$ as the maximum allowed gap size.

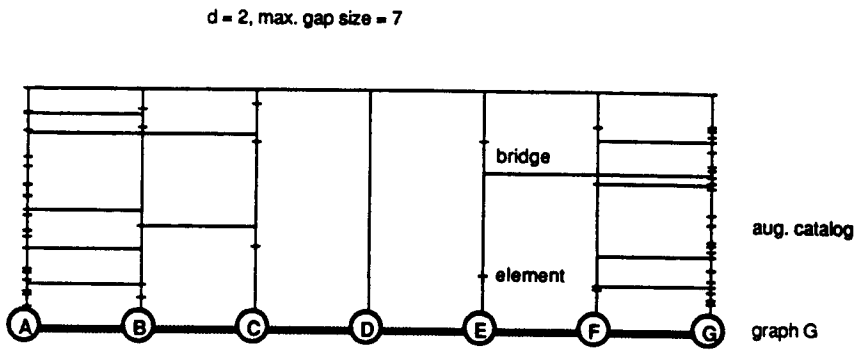


Figure 4. Example (1) of fractional cascading structure

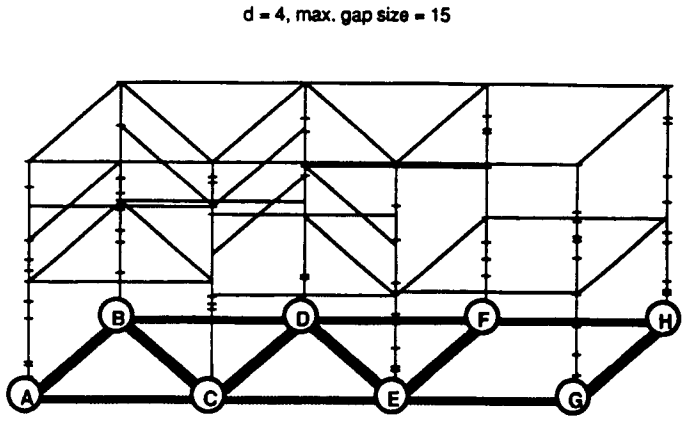


Figure 5. Example (2) of fractional cascading structure

Although our current result will be improved shortly, it is interesting in its own right because it does not attempt to modify the combinatorial nature of the graph. We will see in the next section that by rewriting the graph G in a canonical manner so that it has bounded degree, the preprocessing time can be reduced to $O(s)$, while the query time goes down to $O(p \log d + \log s)$ from $O(pd + \log s)$. In practice, on the other hand, d is most likely to be a small constant, so these asymptotic considerations are immaterial.

5. An Improved Implementation of Fractional Cascading

We have seen that the complexity of the query-answering process is proportional to the degree d. This is unavoidable given the approach taken here: the gap size must be proportional to the

degree if the overall storage is to remain linear. Through the medium of bridges, the query-answering process simulates a traversal of a graph of degree d represented by traditional adjacency lists. This means that in the worst case, to go from node v to its neighbor w, we may have to look at *all* d neighbors of v. To avoid this delay, we choose to resolve high degrees in the graph G by rewriting it in a canonical fashion. This will lead to a graph G^* of bounded degree which emulates G and allows us to go from a vertex v of G to a particular neighbor w in $O(\log d)$ time. Briefly, G^* is constructed by adding a small balanced tree at each node, called a *star-tree*. We solve the iterative search problem on G by applying fractional cascading to G^*, as described in the previous section.

Definition 3. A star-tree T_n is an oriented tree with n leaves (vertices of degree 1), endowed with a distinguished vertex called its *center*, and obtained inductively as follows.

(1) The tree T_1 is a single vertex which, of course, is also its center. The tree T_2 has two vertices connected by an edge; one of them is arbitrarily chosen to be the center.

(2) For $i > 2$, a T_i can be obtained from a T_{i-1} as follows: choose a leaf w of T_{i-1} which has minimum distance to the center of that tree. To form T_i attach two new edges to w and leave the center the same—see figure 6.

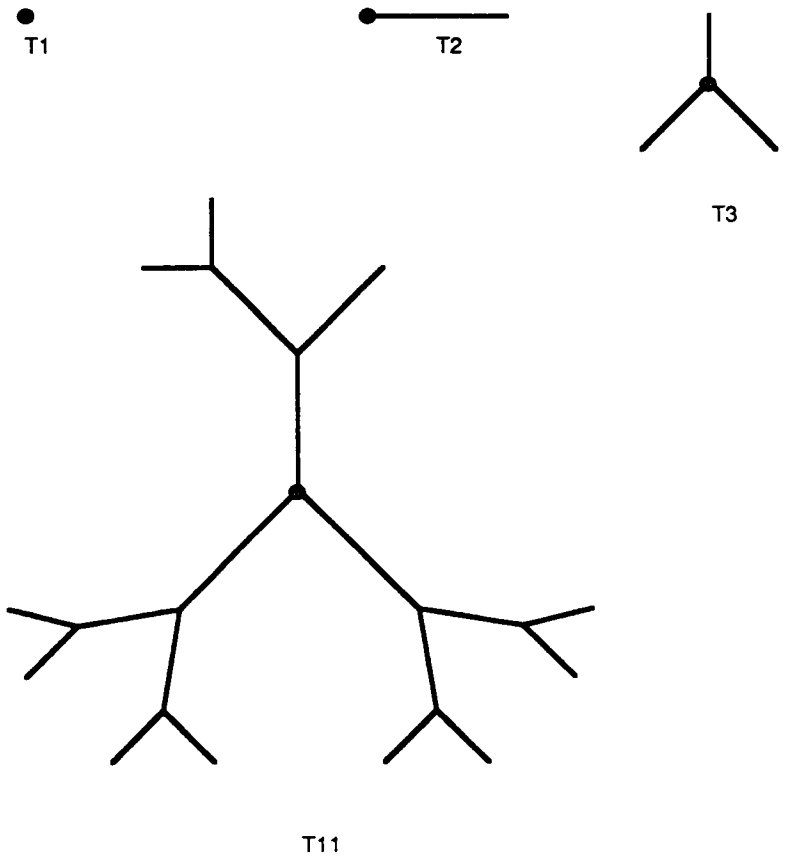


Figure 6. The star-trees used to resolve high degrees

Note that this definition is non-deterministic. In all cases, however, T_n has exactly n vertices of degree one, all interior vertices have degree three, and no vertex is at a distance greater

than $\lceil \lg n \rceil$ from the center.

Now let $e_1, \ldots, e_k$ be the edges of G adjacent to a vertex v of V. If $t_1, \ldots, t_d$ are the leaves of some T_d, we will attach each e_i to some t_j so that the leaves of T_d have a local degree of at most 2 in G^*. Computing the assignment is straightforward. At the outset, the index of each t_j is inserted into a *leaf-queue*. We also extract from C_v the $2k$ endpoints of $R_{e_1}, \ldots, R_{e_k}$ in sorted order (this does not require sorting, since these endpoints form a subset of C_v, which is itself assumed to be given in non-decreasing order). We perform the assignment by going through the endpoints in order, as follows. If the endpoint is a lower endpoint of R_{e_i}, remove any index j from the leaf-queue and assign edge e_i to leaf t_j. If the endpoint is an upper endpoint of R_{e_i}, re-insert back into the leaf queue the index l of the leaf t_l to which e_i had been previously assigned. Because of the locally bounded degree condition, the queue will always contain at least one label when one is needed. This whole process can be carried out in $O(k+d)$ time—see figure 7.

The graph G^* is obtained from G by replacing each node of G with a copy of T_d. (Actually, if a particular node of G has local degree $f < d$, a tree T_f could be used instead to save space). Each edge $e = (u, v)$ of G becomes an edge in G^* connecting the two leaves of the star-trees corresponding to u and v to which e has been assigned by the previous algorithm. In the star-tree T used to replace node u we assign empty catalogs to all nodes, except for the center to which we assign C_u. Also, each edge of T is given a range $[-\infty, +\infty]$. It is now easy to check that all the nodes of G^* have local degree bounded by 3. The graph G^* thus constructed has a number of edges proportional to m, the number of edges in G, and can clearly be built in time $O(s)$.

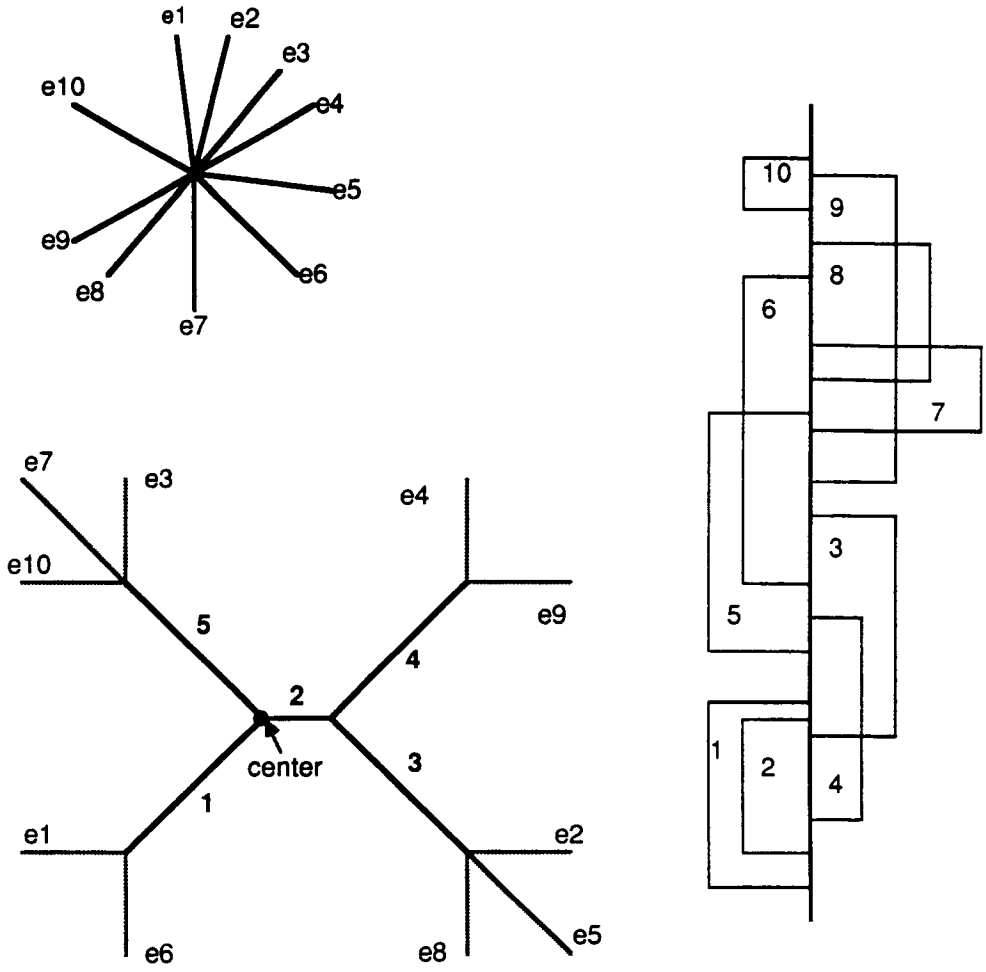
Searching for neighbors in G^* is trivial. Each tree T_d (or T_f) used in G^* has its edges labeled in a depth-first traversal. This allows us to go from one leaf to another in $O(\log d)$ time, provided that we know the labels of the starting and ending edges. All we have to do then is provide a correspondence table to translate the name of an edge in G into the local label of its new adjacent edges (see figure 7). Each edge of G will appear in at most two correspondence tables.

The emulation catalog graph is now ready for use. Note incidentally that the path of a star-tree between two of its leaves may avoid the center. Since the center *must* be visited in order to retrieve the desired information, we will fork the traversal into two paths: one going towards the center, the other pursuing its route towards the exit leaf. The emulation path is obviously still a generalized path. We conclude with an improved version of Theorem 1.

Theorem 2. *Let G be a catalog graph of size s and locally bounded degree d. In $O(s)$ space and time, it is possible to construct a data structure for solving the iterative search problem. The structure allows multiple look-ups along a generalized path of length p to be executed in time $O(p \log d + \log s)$. If d is a constant, this is optimal.*

6. The Notion of Gateways

We address here one of the points left open in previous sections: the location of a query value in the first catalog of the generalized path. The solution proposed earlier consisted of keeping a copy of each augmented catalog in a table, with the idea of performing a binary search in one of them in order to get a multiple look-up started. This is unsatisfying for at least two reasons. For one thing, the solution is inherently static and will support modifications only with great difficulty. Also, it breaks the unity of fractional cascading by stepping out of the list-based world in which we have (implicitly) pledged to remain.



Correspondence Table										
1	2	3	4	5	6	7	8	9	10	
1	3	5	4	3	1	5	3	4	5	

Figure 7. Using the star-tree T_{10}

The answer to these objections will be found in the notion of *gateways*. To each vertex v of G, attach an extra edge connecting v to a new vertex $g(v)$, called the *gateway* of v. The vertex $g(v)$ will have an augmented catalog attached to it but no catalog per se. The edge $(v, g(v))$ is called a *transit* edge; its range is $[-\infty, +\infty]$ — note that these definitions are made with respect to G and not its emulation graph G^*: the transition to G^* must come, as prescribed above, in a postprocessing phase and will ignore differences between edges and transit edges, etc. The augmented catalog of $g(v)$ is required to have exactly three elements. When the preprocessing takes place, if $A_{g(v)}$ should end up with less than three elements,

it is not created. On the other hand, if it ends up with more than three elements, another gateway is attached to it. This process might go on for a while, creating a chain of new vertices emanating from each vertex of G—see figure 8. Only the last vertex in the chain is called a gateway; all the others are called transit vertices.

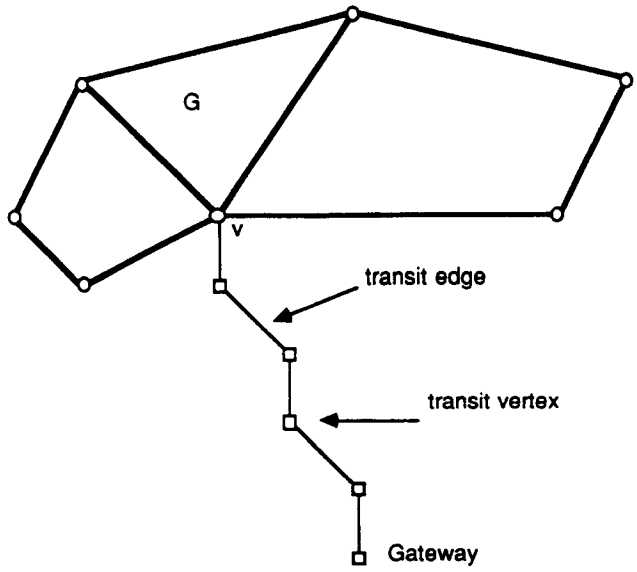


Figure 8. A gateway

It is clear that every time a new gateway is created, there are enough tokens around to pay for this creation. This is all the more obvious as the degree of a transit vertex is two. It is not hard to see that the length of a gateway will be roughly proportional to the logarithm of the size of the augmented catalog at its attachment to G. To answer a query we perform the initial search in A_v by starting at the gateway of v and proceeding to v. This will take $O(\log s)$ time.

As it turns out, a gateway chain is quite similar to a B-tree [K]. In a way it corresponds to a B-tree where all nodes at a given level have been combined into a supernode. The bridges between adjacent levels play the role of the inter-level links in the tree. The whole fractional cascading structure can be viewed as a generalization of B-trees. The upper and lower bounds that must be maintained on the gap size correspond naturally to the upper and lower bounds on the node size of a B-tree. The main difference, and one of the most intriguing aspects of fractional cascading as well, is that in the latter the gap splittings (or mergings) can cycle back to a node previously visited, and so go on for an unpredictable length of time.

7. Dynamic Fractional Cascading

We now examine how the fractional cascading structures can be made dynamic. When building the static structure as described in Section 3, we took advantage of inserting the keys present in each original catalog in increasing order. This sorting allowed us to use a simple linear scan to locate the position of each new key in the augmented catalog, and at the same time to set the value of the C-pointer of each augmented catalog element passed over. Both

the location problem, and the augmented-to-original correspondence problem are much more difficult in the dynamic case.

7.1. Insertions or Deletions Only

Let us first tackle insertions only. Suppose a new key K is to be inserted in C_v. We must (1) compute the position of K in A_v, (2) update whatever representation we are using to relate positions in A_v to positions in C_v, and (3) restore any gap invariants that have been violated by the new insertion. The similarity between gateways and B-trees makes dynamization a straightforward operation, at least as regards (1). Unfortunately, the static structure is grossly inadequate when it comes to problem (2). Too many C-pointers may need to be changed following a single insertion to allow any hope for a logarithmic update cost. In fact, what we must solve is an instance of the ordered set partition problem where we allow the operations *find*, *insert*, and *split*, as described in Section 2.2.1. The *find* operation replaces the C-pointers of the static structure, the *split* operation corresponds to the insertion of a new key in an original catalog, and the *insert* operation is used for the secondary insertions occasionally necessary for the restoration of the gap invariants. A recent paper by Imai and Asano [IA] has shown how to solve this particular case of the ordered set partition problem in constant amortized time per *insertion* or *split*, and constant actual time per *find*. The only assumption their argument requires is that we start with an empty structure. Finally problem (3) can be handled — for a change — exactly as in the static case. We must, however, use the Imai-Asano structure for the secondary insertions associated with stage 3 (the gap splitting). In conclusion:

Theorem 3. *If we allow only insertions, then fractional cascading can be made dynamic while preserving all the previous bounds for space, preprocessing, and query time. The cost of an insertion is $O(\log s)$ when amortized over a sequence of s insertions into an initially empty structure.*

It is possible to handle deletions (and only deletions) in a way analogous to that for insertions. The correspondence between augmented catalogs and original catalogs now requires a solution to the ordered set partition problem where the allowed operations are *merge*, *delete*, *insert*, and *find*. Although not explicitly stated as a result in their paper, Imai and Asano show in fact how to adapt the Gabow and Tarjan [GT] method to handle the first three of the operations above in constant amortized time and *find* in constant actual time. With deletions a new issue arises. It seems hopeless to try to eliminate all copies of an element being deleted from the structure at once. On the other hand, leaving these propagated copies lying around raises the possibility that the structure may no longer remain of linear size in the number of elements present in the original catalogs. But, as observed by Fries, Mehlhorn, and Näher, we can allay this fear if we simply impose a lower bound on the gap size as well [FMN].

Lemma 3. *If the minimum gap size is kept to γd for some $\gamma > 2$, then the size of the fractional cascading structure will be $O(\gamma s/(\gamma - 2))$.*

Proof:

Let (v, w) be an edge of G. We use lower case a's and c's to denote the size of the corresponding augmented and original catalogs, $b_{(v,w)}$ to designate the number of bridges between A_v and A_w, and $g_{(v,w)}$ to designate the number of elements in $A_v \cup A_w$ whose value falls in the range of (v, w). We then have $g_{(v,w)} \geq (b_{(v,w)} - 1)\gamma d + 2b_{(v,w)}$, and therefore $b_{(v,w)} \leq (g_{(v,w)} + \gamma d)/(\gamma d + 2)$.

Since we have made the convention that the original catalogs contain the endpoints of the ranges of their adjacent edges, we obtain $a_v \leq c_v + \sum_{(v,w)\in E}(b_{(v,w)} - 2)$. It follows that

$$\begin{aligned}\sum_{v\in V} a_v &\leq \sum_{v\in V} c_v + \sum_{v\in V}\sum_{(v,w)\in E}(b_{(v,w)} - 2)\\ &= s + 2\sum_{(v,w)\in E}(b_{(v,w)} - 2)\\ &\leq s + 2\sum_{(v,w)\in E}\left[\frac{g_{(v,w)} + \gamma d}{\gamma d + 2} - 2\right]\\ &\leq s + \frac{2d}{\gamma d + 2}\sum_{v\in V} a_v.\end{aligned}$$

From this the desired result follows. ∎

Maintaining both a lower and an upper bound on the gap size gets to be quite intricate. The accounting has to be modified to leave tokens in the piggy-banks for underflowing as well as overflowing gaps, following the method described by Fries, Mehlhorn, and Näher [FMN]. We will not give the details here, but simply state the end result.

Theorem 4. *If we allow only deletions, then fractional cascading can be made dynamic while preserving all the previous bounds for space, preprocessing, and query time. The cost of a deletion is $O(\log s)$ when amortized over a sequence of s deletions leading to an empty structure.*

Next, we will attack the general problem of handling *both* insertions and deletions at the same time. Instead of placing a lower bound on the gap size, we let the data structure degenerate gradually and rebuild it every now and then. The idea is just to mark the deleted elements, but not expend the effort to remove them right away from the structure. The obvious problem with this scheme is that since the data structure never decreases in size, it may become intolerably large compared to the number of *live* elements it contains after many deletions. To deal with this difficulty, we could stop the computation when the ratio of live elements to the total of those present drops below some threshold and re-insert every element still alive from scratch. But we now face the problem that although this scheme might have a good amortized performance, the occasional interruptions might be simply too long to be acceptable. Think for example of an on-line system where requests have to be handled immediately. The next section will bring an answer to this dilemma.

7.2. A General Scheme for Efficient Deletions

Consider a database reacting to three types of requests: insertions, deletions, and queries. Each insertion can be performed in ν amortized time, and each deletion can be recorded in δ actual time, where $\nu, \delta = O(1)$. The notion of recording a deletion as opposed to performing it is the following: an element can be marked off in δ time so that queries may go on and provide correct answers. Recording a deletion, however, does not free any storage, so it is not a viable alternative in the long run. To prove the following result, we use a dynamization technique originating in a paper by Bentley and Saxe [BSa], and one by Overmars [O].

Lemma 4. *Consider a data structure in which we can only insert new elements and answer queries. Let $M(s)$ be the storage used to store s elements, assumed polynomial in s, and let ν indicate the amortized time for an insertion, assumed to be constant. If the time δ*

to mark off an element to be deleted is also constant (thus ensuring that queries can still be answered correctly), it is then possible to implement each deletion in constant actual (non-amortized) time. The storage used is $O(M(s))$, and the time for inserting a new element or answering a query is the same as before, up to within a constant factor.

Proof: The idea, as mentioned earlier, is to "mark" the deleted elements as dead, then periodically garbage collect them to prevent the dead elements from swamping the live ones. Consider the situation at a generic time t. We always keep two identical copies of the data structure, so called the *query* copy and the *survival* copy. All requests are handled simultaneously (i.e., in a time-sharing fashion) in the two structures, except for queries, which are handled exclusively in the query structure. Recall that deletions are handled by simply recording the event, which will take a total of 2δ time. By keeping counters, we check that the live elements always outnumber the dead ones. As soon as this is not the case, we fork two concurrent processes, as described below; in the following, we let A denote the value of the two counters when they meet.

(1) Process 1 continues to handle all three types of requests in the query structure as though nothing was happening.

(2) Process 2 consists of two subprocesses, which can *pipe* information between them. Subprocess 2.1 will keep a transcript of all incoming requests during the entire lifetime of process 2. This includes all insertions and deletions but not queries. Subprocess 2.2 will go through three consecutive stages. In the first one, the subprocess re-inserts every element that is *alive* in the survival structure into a new survival structure. When this is done, the subprocess enters its second stage, where it makes a copy of the new survival structure; from then on the subprocess will work in double, performing the same operations in both copies of the new survival structure. We will not mention this duplication of effort in the following. In the third stage, the subprocess goes through the transcript maintained by subprocess 2.1 and starts responding to each request in chronological order. As soon as process 1 has spent $\delta A/2$ cycles in deletions and insertions (not counting queries), we complete the current request and immediately terminate all processes and subprocesses. The query structure is thrown away, and the two copies maintained by subprocess 2.2 become the query and survival structures. We will make sure that at this point the number of dead elements cannot exceed the number of live ones, so we are back to the initial conditions.

The idea is to have process 2 operate faster than process 1. First of all observe that after a while process 2 mimics process 1, although the duplicating task and the bookkeeping of subprocess 2.1 make this work about three times as hard. At any rate, giving a little more than three cycles to process 2 for every cycle of process 1 should be enough for process 2 eventually to catch up with process 1. However, this catching up should not be delayed too long or we may end up in a forbidden situation where more elements are dead than alive. Recall that from the moment processes 1 and 2 are triggered, no incoming deletion is effectively taken into account until the next process fork. [1]

We will show that setting the speed of process 2 to be $3 + \lceil 8\frac{\nu}{\delta} \rceil$ times the speed of process 1 satisfies all our conditions. Let I and D be respectively the number of insertions and

[1] Note also that the maintenance of multiple copies of the same structure makes the assumption that elements can be marked "dead" in constant time a bit tricky to implement. We cannot refer to an element to be deleted from both the copy and surival structures by a single pointer. We must instead access the element by naming an insertion or query operation record that referenced that element earlier. This "name" could, for example, be the serial number of the operation, which would be the same in the two structures.

deletions handled by process 1. We have

$$\nu I + \delta D \leq \delta \frac{A}{2}. \tag{1}$$

We must show that during these $\delta A/2$ cycles of process 1, process 2 has had time to go through its third stage and handle all I insertions and D deletions. The time necessary for these operations is

(1) *subprocess* 2.1: $I + D$ transcript operations which can be generously accomplished in $\nu I + \delta D$ time; these constants are chosen for convenience.

(2) *subprocess* 2.2 (stage 1): going through every element of the data structure cannot take more time than it would to rebuild it from scratch, so $2\nu A$ is an upper bound on the scanning time. Re-inserting the A elements alive will take νA time.

(3) *subprocess* 2.2 (stage 2): copying the data structure takes less time than rebuilding it, that is, at most νA cycles.

(4) *subprocess* 2.2 (stage 3): implementing the $I + D$ requests twice takes $2(\nu I + \delta D)$ time.

The total running time is dominated by $3(\nu I + \delta D) + 4\nu A$, which by (1) is less than $(\frac{3}{2}\delta + 4\nu)A$. This corresponds to a number of cycles in process 1 at most equal to

$$\frac{(\frac{3}{2}\delta + 4\nu)A}{3 + 8\frac{\nu}{\delta}} = \frac{\delta}{2}A.$$

Therefore, process 2 and its subprocesses will be complete when process 1 is. Note that during that time no more than $A/2$ requests for deletions can be accepted since process 1 lasts only $\delta A/2$ cycles. It follows that during the time the processes are active there are at least $A/2$ live elements and at most $A/2$ dead ones. Therefore the initial invariant is preserved: the dead count never exceeds the live count. Since the function $M(s)$ is polynomial in s, the space will be at all times proportional to what it could be at best, that is, $M(A/2)$. The proof is therefore complete. ∎

Lemma 4 provides a method for the general dynamization of fractional cascading. Whereas insertions are handled as usual, we use a lazy deletion mechanism to remove elements. This means ignoring deletions from augmented catalogs altogether, but reacting to deletion requests by just removing the appropriate elements from their catalogs. As in lemma 4 we will maintain a count of the elements alive and a count of those removed since the last cleanup operation. All the pieces of this process have been described above, except for the correspondence between original and augmented catalogs. Fries, Mehlhorn, and Näher [FMN] have shown how to modify the van Emde Boas priority queue [BKZ] so as to reduce the storage to linear and allow all five operations needed by the ordered set partition problem to be performed in time $O(\log\log s)$, where s is the total number of elements *present in the structure*. This implies that a fully dynamic version of fractional cascading is possible, but at the expense of increasing the cost per look-up to $\log\log s$ from constant:

Theorem 5. *If we allow both insertions and deletions, then a fractional cascading structure can be built whose size is $O(s)$ and where a multiple look-up along a generalized path of length p costs $O(p \log d \log\log s + \log s)$ in time. Both deletions and insertions can be handled in amortized time $O(\log s)$.*

8. General Remarks

In part II of this paper we give a large number of applications of fractional cascading to query problems. In fact, our discovery of this technique is due to noticing that tricks bearing a certain similarity had been used in a number of published algorithms [Ch1,Co,EGS] to deal with the problem of iterative search. Examples are the *hive-graph* of Chazelle [Ch1] and the chain refinement scheme of Edelsbrunner et al. [EGS]. These connections are developed more fully in part II.

The most unsatisfactory aspect of our treatment of fractional cascading is the handling of the dynamic situation. Is our method optimal? Whether it is or not, can it be simplified to the point of being useful in practice? Even in the insertion-only or deletion-only cases, our techniques are more of theoretical than practical interest, because of the large constants involved. We also feel that we do not fully understand the influence of high degree vertices in G on the method (see also [CG] for some additional comments on this). Can fractional cascading be applied to graphs, such as planar graphs, of bounded average degree — or does the presence of a small but non-constant number of high degree vertices really destroy the sampling/propagation?

We conclude by remarking that the philosophy of fractional cascading can be extended to other iterative search problems, beyond that of searching in a linearly ordered catalog. The three main requirements seem to be (1) that two search structures $\mathcal{A}$ and $\mathcal{B}$ can be merged into a joint structure efficiently (spec. in linear space), (2) that once the position of a "key" is known in the merged structure, its position in the component structures should be computable efficiently (spec. in constant time), and (3) that an appropriate notion of "sample" exist such that location of a key in the sample allows efficient (spec. constant time) location in the original. We hope that this paradigm will yield useful results in other areas as well.

Acknowledgments: We wish to thank Cynthia Hibbard, Ian Munro, Jorge Stolfi, and Robert Tarjan for many useful comments on the manuscript. We are also grateful to Marc Brown for programming a preliminary version of fractional cascading in the static case. The idea for the construction in the following appendix is due to Jorge Stolfi.

Appendix A. How gaps can get big.

It is possible to construct a catalog graph with any given degree d, $d \geq 3$, where insertion of a single record in one catalog ultimately propagates to many insertions into the same gap of another catalog. In the example we give below, we achieve secondary insertions into the same gap whose total number is $\Omega(s^\rho)$, where s is the size of our catalog graph and ρ is roughly $\log 2/\log(6d)$. We do not know if this is best possible. This example shows the need for the careful gap size counting we had to do in Section 3.1.

Our catalog graph will consist of two parts: a *multiplier* and a *concentrator*. The concentrator is just a linear chain of $k+1$ nodes, k to be determined later on. Across the last edge e_k of this chain there is only one gap (two bridges). This gap overlaps $6d-1$ gaps of the previous edge e_{k-1}; see figure 9 for an illustration. Now each of these gaps overlaps $6d-1$ gaps of the previous edge e_{k-2}, and so on. Therefore across the first edge e_1 there will be $(6d)^k+1$ bridges. The catalog of the first node contains enough additional records to bring all gaps across the first edge e_1 to saturation. The total size of this structure is $\Theta\big((6d)^k\big)$.

Consider what happens when we simultaneously insert one new record in each of the gaps of the first catalog. By simultaneously we mean during the same invocation of stage 3 as described in Section 3.1. All gaps of e_1 will split, causing $6d$ insertions into each gap of the second catalog. This will cause each gap over e_2 to overflow and reach size $12d-1$. In the

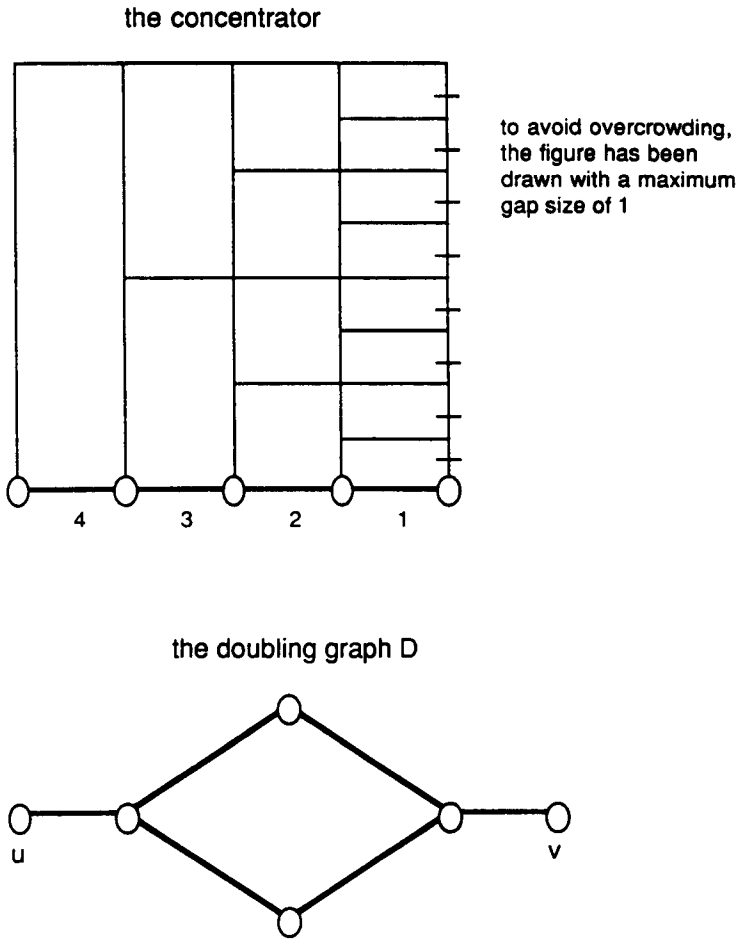


Figure 9. The concentrator and the multiplier

next iteration through stage 3 these gaps will split and will push *two* new records to be inserted into each gap of e_3 (because $12d - 1 = 3d + 3d + 6d - 1$). The gaps of e_3 now will reach a size of $16d - 1$ and will split the next time around, yielding *four* new insertions into each gap of e_4 (since $18d - 1 = 3d + 3d + 3d + 3d + 6d - 1$). This pattern continues, with the number of insertions into each gap doubling at each iteration. In the last splitting, 2^k secondary insertions will occur simultaneously in the one gap over e_k.

The job of the multiplier is to produce in the same stage all insertions needed to start the above process in the concentrator. It uses a doubling graph $D^{(1)}$ having an entry node u and an exit node v, and in addition four other nodes arranged as in figure 9. The catalog A_u has only two records, both bridges delimiting the same gap. The catalog A_v has three records, again all bridges delimiting two gaps. The catalogs of the other nodes are easily arranged so that an insertion into the gap of A_u causes an insertion into each of the gaps of A_v three stages later. We need a total of about $12d + O(1)$ records for this.

If we stack up m copies of these augmented catalogs on top of each other we obtain $D^{(m)}$, a graph where an insertion into each of the m gaps of A_u will cause an insertion into each of the $2m$ gaps of A_v. The multiplier is constructed by concatenating $D^{(1)}, D^{(2)}, \ldots, D^{(n)}$, where

n is chosen so that $2^n = (6d)^k + 1$. This compound graph produces the grouped insertions needed to feed the concentrator. The total size of our catalog graph is $O\big((6d)^k\big)$ and the number of insertions into a sigle gap it produces is 2^k. This proves the bound mentioned at the beginning if we fix k so that $s = \Theta\big((6d)^k\big)$ and thus completes our construction.

Fractional Cascading: II
Applications

1. Introduction

As we saw in part I, *fractional cascading* is an algorithmic technique for searching several sets at once. This generalized form of searching often arises in the solution of query problems. Imagine that you come upon a word of unknown origin, which you wish to identify. One solution is to look up the word in as many dictionaries as it will take to find it. Fractional cascading gives you a way out of this repetitive search. It offers you the following alternative: look up the word in one dictionary, and from then on jump directly into each of the other dictionaries in constant time. To make this happen, the dictionaries will have to be somehow reorganized, and linked together by some appropriate mechanism. We showed in part I that all this rearrangement can be done at fairly little cost.

The goal of this second part is to present a number of problems whose solutions can be significantly improved by using fractional cascading. Most of the algorithms presented are short and simple. We believe that fractional cascading is a speed-up mechanism of practical as well as theoretical relevance. One goal of this part will be to justify the first part of this belief. From now on, we will assume that the reader is familiar with the basic terminology of fractional cascading, such as *iterative search*, *catalogs*, *multiple look-ups*, etc. For convenience, let's recall the main findings of part I.

> **Fractional Cascading:** Let G be a catalog graph of size s and locally bounded degree d. In $O(s)$ space and time, it is possible to construct a data structure for solving the iterative search problem. The structure allows multiple look-ups along a generalized path of length p to be executed in time $O(p \log d + \log s)$. If d is a constant, this is optimal. The data structure is dynamic in the following sense. If only insertions are performed, the amortized time for each insertion will be $O(\log s)$; the same holds for deletions. Arbitrary insertions and deletions can also be done in $O(\log s)$ amortized time, but the query time becomes $O(p \log d \log \log s + \log s)$.

How is this part organized and what will we find in it? The applications we consider in this part all revolve around the notion of a *query problem*. In each case, one must design a database to answer efficiently certain types of queries relative to some given objects. This will lead us to examine problems of intersecting a line with a fixed polygonal path (Section 3), reporting points lying inside a trapezoidal region (Section 4) or a hyperrectangle (Section 5), performing range search in the past (Section 6), computing locus-functions (Section 7), compressing segment trees (Section 8), and extending query problems (Section 9). The reader puzzled by these rather vague descriptions can skip to the appropriate sections for clarification. On the last application, however, we wish to say a little more at this point. Section 9 concerns a fairly general principle which best illustrates the power of fractional cascading.

In a standard query problem, call it Π, a query specifies a certain subset of the given objects, and the goal is to compute this subset as fast and economically as possible. Often, however, the objects themselves are pointers to files which, once identified, must then be searched in a later stage[D. We call the resulting problem an *IS-extension* of Π (iterative search extension). What we will show in Section 9 is that with the use of fractional cascading (almost) any solution to a query problem can be transformed into a solution to its IS-extensions with little or no degradation of performance. This touches on a central aspect of fractional cascading: its use as a postprocessing device. Most often, fractional cascading is applied to a data structure at the very end stage of its development. What is remarkable is that its applicability depends on *syntactical* rather than *semantic* characteristics of the data structure. To have the basic appearance of a catalog graph is what really matters, and not so much the particular mathematical domain within which the data structure's semantics is defined. This feature grants fractional cascading great versatility.

The notion of iterative search comes in two flavors. It is called *explicit* if the problem to be solved makes explicit reference to a collection of catalogs. Queries are specified by a subset of this collection along with a search key. In the applications we just mentioned, however, iterative search is *implicit*. That is to say, the problems do not make mention of it in their statements; they don't even allude to it. It is only in the *specific* solutions chosen that iterative search shows its face. For practical reasons, implicit iterative search is what justifies the use of fractional cascading. We may still legitimately ask ourselves: how well understood is explicit iterative search? We study this problem in the next section. In particular, we examine the sensitivity of fractional cascading to the presence of high degree vertices in the catalog graph.

2. Explicit Iterative Search

Let $S = \{C_1, \ldots, C_p\}$ be a collection of p catalogs, and let $s = \sum_{1 \leq i \leq p} |C_i|$ be the combined size of the catalogs. ***Explicit iterative search*** is the following problem: given a query of the form (q, H), where q is a real number and H is a subset of $\{1, \ldots, p\}$, compute the successor of q in C_i for each $i \in H$ (recall that the successor of q in C_i is the smallest element in $C_i \cup \{+\infty\}$ larger than or equal to q). We solve this problem by setting up the conditions necessary for fractional cascading. Let G be a complete binary tree on p nodes, each associated with a distinct catalog: G is called an *emulation graph* of S. For convenience, we refer to the elements of H as nodes of G.

The idea is to apply fractional cascading to the emulation graph and answer the query by traversing the minimum spanning tree T of H (figure 1). Each node of G will have a flag for marking purposes. We compute T by iterating on the following process. Initially, all nodes of G are unmarked. For each node v of H, traverse the path from v to the root, marking each node along the way, and stopping as soon as a node already marked is encountered. At the end of this process, the set of marked nodes forms a spanning tree of H. It is not necessarily minimum since it *always* contains the root. We must now remove the branch joining the root of G to the lowest common ancestor of the nodes of H. Let v be the root of G; if v is not a node of H and has a single marked child w, then unmark v and iterate with respect to w, else stop. With T in hand, we can answer the query by performing multiple look-ups in the catalogs attached to the nodes of T.

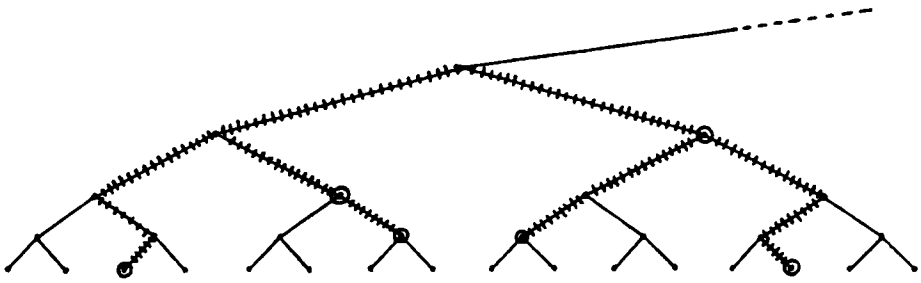


Figure 1. The emulation graph

A simple analysis shows that the time taken by the construction of T as well as the search in each catalog is $O(|T| + \log s)$. Let $v_1, \ldots, v_m$ be the vertices of H sorted by increasing inorder ranks. Let l_i be the lowest common ancestor of v_i and v_{i+1} $(1 \leq i < m)$, and let h_i denote the

number of ancestors of l_i in G. A rough analysis shows that $|T| \leq 2\sum_{1\leq i\leq m-1}(\log p - h_i)$. Since fewer than 2^j vertices among the l_i's can have fewer than j ancestors, we have

$$\sum_{1\leq i\leq m-1}(\log p - h_i) \leq \sum_{1\leq j\leq \lfloor \log m\rfloor} 2^j(\log p - j),$$

and therefore $|T| = O(m + m\log \frac{p}{m})$. We conclude that the running time of the algorithm is $O\big(|H|\log \frac{p}{|H|} + \log s\big)$.

The next question to decide is whether this result is optimal. After all, fractional cascading allows us to use *any* graph of bounded degree as a supporting search structure, so one might wonder whether a fancier catalog graph with, say, cycles to provide shortcuts can yield a better performance. We show that this is not the case.

Lemma 1. *Among all emulation graphs of S of bounded degree, the complete binary tree on p nodes is asymptotically optimal.*

Proof: Let G be an emulation graph of degree $\leq d$, and let H be a subset of m vertices carefully chosen so as to make the minimum spanning tree T of H as large as possible. Let the distance between two vertices v and w be defined as the number of edges on the shortest path between v and w. Let $l = \lfloor \log_d \frac{p}{m}\rfloor - 1$. Pick a vertex v in G and mark off all vertices at a distance less than or equal to l from v (this includes v). Next, pick a non-marked vertex and iterate on this process until all vertices are marked. Since G has bounded degree d, each iteration will mark at most d^{l+1} vertices, therefore at least m vertices will be picked in the process. Let $H = \{v_1, \ldots, v_m\}$ be the chosen vertices and let T be any spanning tree of H. For each v_i, there must exist at least one vertex w_i in T at a distance $\lfloor l/2\rfloor$ from v_i. Let p_i be the path in T between v_i and w_i. By construction of H, the paths $p_1, \ldots, p_m$ are vertex-disjoint, therefore the size of T is at least the added size of the p_i, that is, $\Omega\big(|H|\log \frac{p}{|H|}\big)$. ∎

Lemma 1 shows that our choice of G is adequate, but it still falls short of proving the optimality of the technique. Why can't a different method be used that perhaps bears no relation with fractional cascading? What we will show is that no improvement can be expected in a pointer machine model [T], if H is given as a set of indices and not as a set of addresses. Why is that so? Let's ask ourselves: how many different collections of dictionaries can be identified by taking t steps on a pointer machine? A single step gives a choice of at most c memory accesses, for some machine-dependent constant c. Therefore "at most c^t collections" is the answer. But there are $\binom{p}{m}$ possible sets H, so t must be at least on the order of $\log_c \binom{p}{m}$. As long as $m = o(p^{3/4})$, we have the elementary asymptotic formula

$$\binom{p}{m} = \frac{p^m e^{-\frac{m^2}{2p} - \frac{m^3}{6p^2}}}{m!}(1 + o(1)).$$

Using Stirling's approximation

$$m! = m^m e^{-m}\sqrt{2\pi m}(1 + o(1)),$$

we find that t must be at least on the order of

$$\frac{1}{\log c} m\log \frac{p}{m} + m\big(1 - \frac{1}{2p^{1/4}} - \frac{1}{6p^{1/2}}\big),$$

that is, $\Omega\left(m \log \frac{p}{m}\right)$. This shows that, at least for $m = o(p^{3/4})$, our algorithm is optimal. Keep in mind, however, that this argument assumes that the catalogs are referred to by indices and not by addresses.

Theorem 1. *Let Π be an explicit iterative search problem involving p catalogs of combined size s. There exists a data structure for solving Π such that any query can be answered in $O(m \log \frac{p}{m} + \log s)$ time, where m is the number of catalogs involved in the query. The data structure requires $O(s)$ space and can be constructed in $O(s)$ time. Within the context of fractional cascading, this result is optimal.*

The naive method requires $O(m \log s)$ response time, so the scheme of Theorem 1 is superior whenever the size of the catalogs exceeds their number ($s > p$), a situation of great likelihood in practice. The solution is optimal when the number of catalogs queried is at least a fixed fraction of the total number of catalogs ($p = \Omega(m)$).

We now turn our attention to implicit iterative search. Ironically, the problems for which fractional cascading seems the best suited do not even suggest the notion of iterative search in their statements. Their solutions, however, are inherently dependent on iterative search. This situation occurs in many query problems, as we will see.

3. Intersecting a Polygonal Path with a Line

In this section we investigate the following problem: we are given a polygonal path P and wish to preprocess it into a data structure so that, given any query line ℓ, we can quickly report all the intersections of P with ℓ. The obvious method for solving this problem simply checks each side of P for intersection with ℓ. This method requires storage $S = O(n)$, where n is the length of P, and has query time $Q = O(n)$. We desire a method with a query time of the form $Q = O(f(n) + k)$, where $f(n) = o(n)$ and k is the number of intersections reported. Using fractional cascading we are able to develop a technique that gives $Q = O\left((k+1) \log \frac{n}{k+1}\right)$. When k is a small constant, the running time is $O(\log n)$, which is optimal. When $k = \Omega(n)$, the running time is $O(k)$, and this is also optimal. For intermediate values of k, the expression of the query time suggests that the discovery of each intersection incurs the cost of a binary search. This is actually a fairly accurate reflection of the searching strategy. Our solution represents partial progress towards the desired goal.

The storage requirement of the method is $O(n \log n)$, but in the case where the polygonal path is *simple*, it can be reduced to $O(n)$. This is another instance of an interesting phenomenon in computational geometry, where the simplicity of a polygon reduces some required resource for an algorithm by a factor of $\log n$. Computing the convex hull is another well-known example.

The technique we propose in this section is based on the recursive application of the following observation:

Lemma 2. *A straight-line ℓ intersects a polygonal line path P if and only if ℓ intersects the convex hull $CH(P)$ of P.*

Proof: Obvious. ∎

Let $F(P)$ and $S(P)$ denote respectively the first and second halves of the path P, that is, the subpaths of P consisting of the first $\lfloor n/2 \rfloor$ and second $\lceil n/2 \rceil$ edges. Then our algorithm is expressed very simply recursively as:

```
Intersect(P, ℓ)
  begin
  if |P| = 1 { single edge } then
        compute P ∩ ℓ directly
  else if ℓ does not intersect CH(P) then exit
  else
    begin
        Intersect(F(P), ℓ)
        Intersect(S(P), ℓ)
    end
  end
```

Since we are allowed to preprocess P, it is to our advantage to precompute and store all the convex hulls we may need. We can do this by a recursion similar to that above, where, after obtaining $CH(F(P))$ and $CH(S(P))$, we compute $CH(P)$ by any one of a number of *linear-time* algorithms for computing the convex hull of two convex polygons [PH]. The overall data structure that we thus build is best thought of as a binary tree T whose n leaves are the edges of our path P (which coincide with their own convex hulls and from left to right occur in the same order as in P). Interior nodes of the tree correspond in an obvious way to subpaths of P and store the convex hull of their respective subpaths (figure 2).

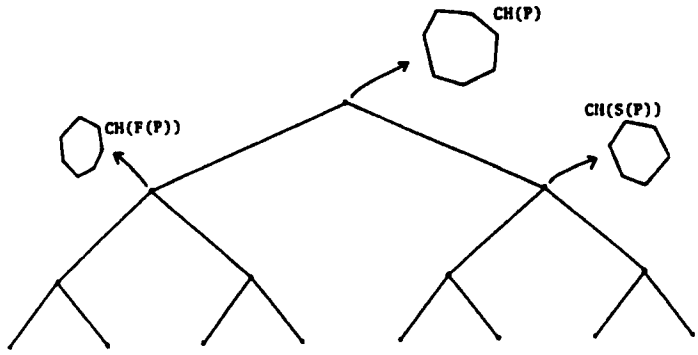


Figure 2. The convex hull decomposition

The tree T of convex hulls clearly takes $O(n \log n)$ space to store. The total time for computing it is also $O(n \log n)$ since, by the discussion above, this time satisfies a recurrence of the form

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + O(n).$$

We must now look more closely at the implementation of our intersection algorithm. We decide whether to descend into a subtree by testing for intersections between the convex hull stored in its root and the line ℓ. Even if we were to report only one intersection, the total cost of all these tests would be

$$\Omega(\sum_i \log \frac{n}{2^i}) = \Omega(\log^2 n),$$

since it costs $O(\log m)$ to test for intersection between a convex polygon of m sides and a line, and in T we must trace at least one path down to the intersected edge. This is already too expensive, so some additional weaponry must be brought into the battle. This is where fractional cascading comes in.

The underlying tree T is a perfectly good graph of bounded degree. However, how are we to view the "two-dimensional" (convex polygon, line) intersection problem as one of a look-up in a one-dimensional catalog? The answer is given by a simple observation. Let $c_1, \ldots, c_m$ be the vertices of a convex polygon given in clockwise order, and let $c_i x$ be the horizontal ray emanating from c_i towards $x = +\infty$. We define the slope of an edge $c_i c_{i+1}$ as the angle $\angle(c_i x, c_i c_{i+1}) \in [0, 2\pi)$. It is well-known that since C is convex there exists a circular permutation of the edges of C such that the sequence of slopes is non-decreasing. This sequence is unique, and is called the *slope-sequence* of C.

Lemma 3. *Let s and s' be the two slopes of ℓ obtained by giving the line its two possible orientations; if we know the positions of s and s' within the slope-sequence of a convex polygon C, we can determine whether C and ℓ intersect in constant time.*

Proof: In effect the positions in the slope-sequence tell us the vertices of C where the tangents parallel to ℓ occur. The line ℓ will intersect C if and only if it lies between these two tangents. ∎

Thus we view each node x of T as containing a catalog consisting of the slope-sequence of the convex polygon associated with x. To these catalogs over T we apply fractional cascading. The result is a more elaborate structure, but one still only requiring space $O(n \log n)$. The data structure allows us to implement *all* the (convex polygon, line) intersection tests required by our algorithm, except for the one at the root, in constant time per test. By the previous lemma, any time we need to decide whether to descend into a subtree, we just look up the slopes of ℓ in that subtree's root catalog and find the answer in constant time! There is, of course, an $O(\log n)$ cost at the root of T to get the whole process started.

As a net result, the cost of our intersection algorithm is now reduced to $O(\log n +$ size of subtree of T actually visited) since, once we pass the root, we spend only constant effort per node visited. Our claimed query time bound of $O((k+1)\log \frac{n}{k+1})$ now follows from the following lemma.

Lemma 4. *Let T be a perfectly balanced tree on n leaves and consider any subtree S of T with k leaves chosen among the leaves of T. Then,*

$$|S| \leq k\lceil \log n \rceil - k\lfloor \log k \rfloor + 2k - 1.$$

Proof: In S there are k leaves and $k-1$ branching nodes (outdegree 2). The size of S is maximized when all the branching nodes occur as high in T as possible. Then the number of remaining non-branching nodes in S is at most $k(\lceil \log n \rceil - \lfloor \log k \rfloor)$. ∎

We have finally shown,

Theorem 2. *Given a polygonal path P of length n, it is possible in time $O(n \log n)$ to build a data structure of size $O(n \log n)$, so that given any line ℓ, if ℓ intersects P in k edges, then these edges can be found and reported in time $O\big((k+1)\log \frac{n}{k+1}\big)$.*

We next show how the storage used can be reduced to $O(n)$ when P is known to be simple (i.e., non self-intersecting). The key lemma is

Lemma 5. *If P is simple, then $CH(F(P))$ and $CH(S(P))$ have at most two common tangents (figure 3).*

Proof: Consider $CH(P)$; the interior of this polygon is partitioned by the simple path P into a number of simply connected regions: $CH(P) \setminus P = \cup_i R_i$. The regions R_i are in one-to-one correspondence with the edges of $CH(P)$ that are not edges of P, except for possibly the interior of P, if P is closed. To see this, note that for any point in $CH(P) \setminus P$ (except for points inside P, if P is closed) there is a path to infinity that avoids P. Thus, regions containing such points must have on their boundary edges of $CH(P)$ that are not part of P. Furthermore, a particular region R can never have more than one such edge on its boundary because P is connected.

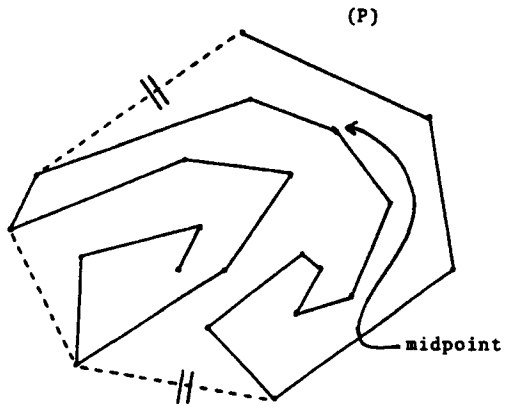


Figure 3. Sharing common tangents

Let us now examine the remaining boundary edges of this region R. Naturally, they are all edges of P. Since they form a connected set, they must form a subpath of P. The order of the edges along the subpath corresponds to the order of the same edges around R, with one exception. That arises when the initial or final vertex of P is interior to R. In these cases an initial or final segment of P may occur on the boundary of R twice.

Now let x be the midpoint of P that is the vertex separating $F(P)$ from $S(P)$. If an edge e of $CH(P)$ is a common tangent of $CH(F(P))$ and $CH(S(P))$, then e cannot be an edge of P. The boundary of the region R of $CH(P)$ bounded by e, with e removed, is a subpath of P joining $F(P)$ to $S(P)$. Therefore x is on the boundary of R. Since x can be on the boundary of at most two regions, there can be at most two common tangents. Note that the regions on either side of x can be the same region R. In that case the edge e of $CH(P)$ associated with R is an edge of either $CH(F(P))$ or $CH(S(P))$, since x is encountered twice when walking along the boundary of R. This implies that there are no common tangents of $CH(F(P))$ and $CH(S(P))$: one is fully enclosed in the other. Note also that one of the common tangents can be degenerate, in case x is on $CH(P)$. ∎

Since $CH(P)$ can be obtained from $CH(F(P))$ and $CH(S(P))$ by drawing the two common tangents (if any exist) and then throwing away the interior segments of the convex hulls of the parts, it follows that the total number of distinct edges used by all the convex hulls of T is at most $n + 2(n-1) = 3n - 2$. Therefore an algorithm with $O(n)$ storage may be feasible. Of course, a particular edge e may appear in many convex hulls. If we are to store it only once, where should we store it? The answer is: "at the highest node of T whose associated hull contains e". It is easy to check that this node is well-defined. A similar trick has been

used by Lee and Preparata for edges that appear on many separators in their classic point location paper [LP]. Thus, at each node of T, only a certain subset of the edges of its convex hull is stored, namely those that do not appear in hulls higher up in the tree. This particular choice has a fortunate consequence.

Lemma 6. *If we store each edge in the highest node in T in whose convex hull this edge appears, then all the edges stored at a particular node form a contiguous interval of the cycle of edges forming the convex hull of the node.*

Proof: The edges stored with a node v of T are exactly those which are not also edges of the parent of v in T. By the previous lemma, v and its brother have common hulls with at most two common tangents. The assertion follows. ∎

Thus we can view the stored edges at each node as a catalog of slopes, and apply fractional cascading. The lemma above implies that if the slope of a line ℓ we are looking up falls outside of the stored catalog of a node v, then the answer we want is the same as what we get for the parent of v. Again, we can in constant time per node locate the two tangents of the convex hull associated with the node and parallel to ℓ (root excepted). So we have shown,

Theorem 3. *Given a simple polygon path P of length n, it is possible in time $O(n \log n)$ to build a data structure of size $O(n)$, so that given any line ℓ, if ℓ intersects P in k edges, then these edges can be found and reported in time $O\big((k+1)\log \frac{n}{k+1}\big)$.*

4. Slanted Range Search

Let E^2 be the Euclidean plane endowed with a Cartesian system of axes (Ox, Oy). We will use the term *aligned rectangle* to refer to the Cartesian product $[a,b]\times[0,c]$, for some positive reals a, b, c. The *aligned range search* problem involves preprocessing a set V of n points so that for any aligned rectangle R, the set $V \cap R$ can be computed efficiently. McCreight [M2] has described a data structure, called a *priority search tree*, which allows us to solve this problem in optimal space and time. The data structure requires $O(n)$ space and offers $O(k + \log n)$ response time, where $k = |V \cap R|$ is the size of the output. Can the priority search tree be extended to solve a more general class of range search problems? For example, consider adding one degree of freedom to the previous problem. We define an *aligned trapezoid* as a trapezoid with corners $(a,0),(b,0)$ and $(a,c),(b,d)$, with $a < b$, $c > 0$, and $d > 0$. In the *slanted range search* problem, the set to be computed is of the form $V \cap R$, where R is an aligned trapezoid. Figure 4 illustrates the difference between the two problems. Note that slanted range search is strictly more general than aligned range search. Informally, the "roof" of the range is now of arbitrary slope. For this reason the priority search tree is inadequate. Instead, we turn to a slightly more complicated data structure, which we develop in two stages. First, we outline a data structure of linear size. Its response time is $O(\log^2 n + k \log n)$, where k is, as usual, the number of points to be reported. Then we show how to improve this solution by application of fractional cascading.

A special case of slanted search has been solved by Chazelle, Guibas and Lee [CGL]: given a query line L, report all points of V on one side of L. The algorithm, which is optimal in both space and time, is intimately based on the notion of *convex layers*, a structure obtained by repeatedly computing and removing the convex hull of V. This preprocessing partitions the point set into a hierarchy of subsets, each of which lends itself to efficient searching. For the purpose of slanted range search, we add a recursive component to the construction of layers. To begin with, observe that without loss of generality we can assume that all points in V have

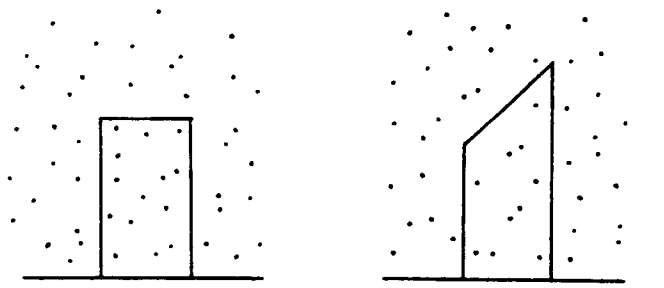


Figure 4. Two cases of range searching

distinct x-coordinates. If this is not the case, we store each group of points with the same x-coordinates in a linked list sorted by increasing y-coordinates. In this way, we may ignore every point that is not first in its list. Each time a point is reported, the corresponding list is scanned until we run into a point falling outside of the range. Of course, we can assume that all points with negative y-coordinates have been removed. Next, we introduce the notion of *lower hull* of the point set V, denoted $L(V)$. If $a_1, \ldots, a_i, a_{i+1}, \ldots, a_j$ are the vertices of the convex hull of V, given in counterclockwise order with a_1 (resp. a_i) the point with minimum (resp. maximum) x-coordinate, $L(V)$ is defined as the sequence of points $a_1, \ldots, a_i$. If V consists of a single point, then $L(V) = V$.

We are now ready to describe the data structure. It is constructed recursively by associating the list $D(v) = L(V)$ with the root v of a binary tree G. Let W be the points of V not in $L(V)$, and let $v.l$ and $v.r$ denote respectively the left and right children of node v. The data structure $D(v.l)$, associated with $v.l$, is defined as the sequence of points $L(W')$, where W' is the leftmost half of W. A data structure $D(v.r)$ is defined similarly with respect to the rightmost half of W (figure 5). The recursion stops as soon as W is empty, so G is finite: its size is trivially bounded above by n. The construction procedure is executed by calling $BUILD(V,\text{root})$.

Build(C, v)

begin
if $C = \phi$ **then** stop
$D(v) \leftarrow L(C)$
$W \leftarrow C \setminus L(C)$
Let α be the $\lceil \frac{|W|}{2} \rceil$th largest x-coordinate in W.
Build$\big(W \cap \{x \leq \alpha\}, l(v)\big)$
Build$\big(W \cap \{x > \alpha\}, r(v)\big)$
end

Each data structure $D(v)$ is now refined as follows: let $D(v) = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ be the lower hull at node v, with $x_1 < x_2 < \ldots < x_m$. The two pieces of information of interest at node v are:

(1) Abs$(v) = \{x_1, \ldots, x_m\}$, the sorted list of x-coordinates in $D(v)$;

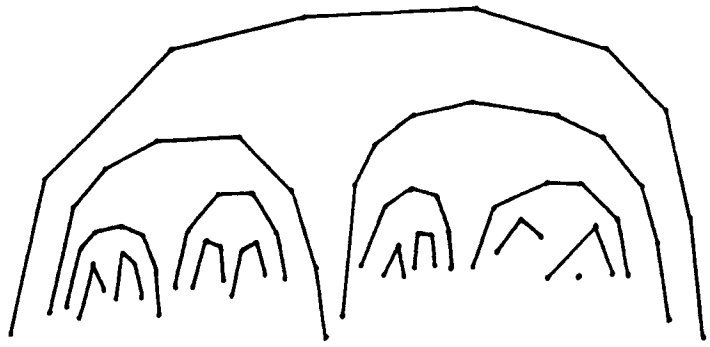


Figure 5. A tree of convex hulls

(2) Slope$(v) = \{\frac{y_2-y_1}{x_2-x_1}, \ldots, \frac{y_m-y_{m-1}}{x_m-x_{m-1}}\}$, the sorted list of edge-slopes in $D(v)$.

For explanatory purposes, we describe the query-answering process in two stages. A phase preliminary to the query-answering process marks selected vertices of G using two colors, *blue* and *red*. The red vertices are then used as starting points for the second stage of the algorithm, where the remaining candidate vertices are examined. We successively describe the algorithm, prove its correctness, and examine its complexity. As a convenient piece of terminology, we introduce the notion of an *L-peak*. Let L be the line passing through the two points (a, c) and (b, d), and let L^- be the half-plane below L. We define the L-peak of $D(v)$ as the point of $L^- \cap D(v)$ whose orthogonal distance to L is maximum (break ties arbitrarily). The L-peak of $D(v)$ is 0 if $L^- \cap D(v) = \phi$.

Stage 1: The algorithm is recursive and starts at the root v of G. In the following, $D(v)$ is regarded as the polygonal line with vertices $(x_1, y_1), \ldots, (x_m, y_m)$. The query trapezoid $R = \{(a,0), (b,0), (a,c), (b,d)\}$ falls in one of three positions with respect to $D(v)$:

(1) $D(v)$ intersects the vertical segment $r_a = \{(a, y) | 0 \leq y \leq c\}$ at some edge $[(x_{i-1}, y_{i-1}), (x_i, y_i)]$: as long as the point (x_i, y_i) is defined and lies in R, report it and increment i by one. If $D(v)$ does not intersect r_a but intersects the segment $r_b = \{(b, y) | 0 \leq y \leq d\}$ at some edge $[(x_j, y_j), (x_{j+1}, y_{j+1})]$, then perform a similar sequence of operations. As long as the point (x_j, y_j) is defined and lies in R, report it and decrement j by one. As a final step, mark v blue. If v is a leaf of G then return, else recur on its children (figure 6, case 1).

(2) $D(v)$ is completely to the left or to the right of R, that is, $x_m < a$ or $x_1 > b$: return (figure 6, case 2).

(3) None of the above: mark v red and return (figure 6, case 3).

Stage 2: As long as there are some unhandled red vertices left in G, pick any one of them, say v, mark it "handled" and compute (x_i, y_i), the L-peak of $D(v)$. Next, perform the following case-analysis:

(1) The L-peak of $D(v)$ lies in R: report it, and initialize j to $i + 1$. As long as the point (x_j, y_j) is defined and lies in R, report it and increment j by one. Next, re-initialize j to $i - 1$; as long as the point (x_j, y_j) is defined and lies in R, report it and decrement j by one.

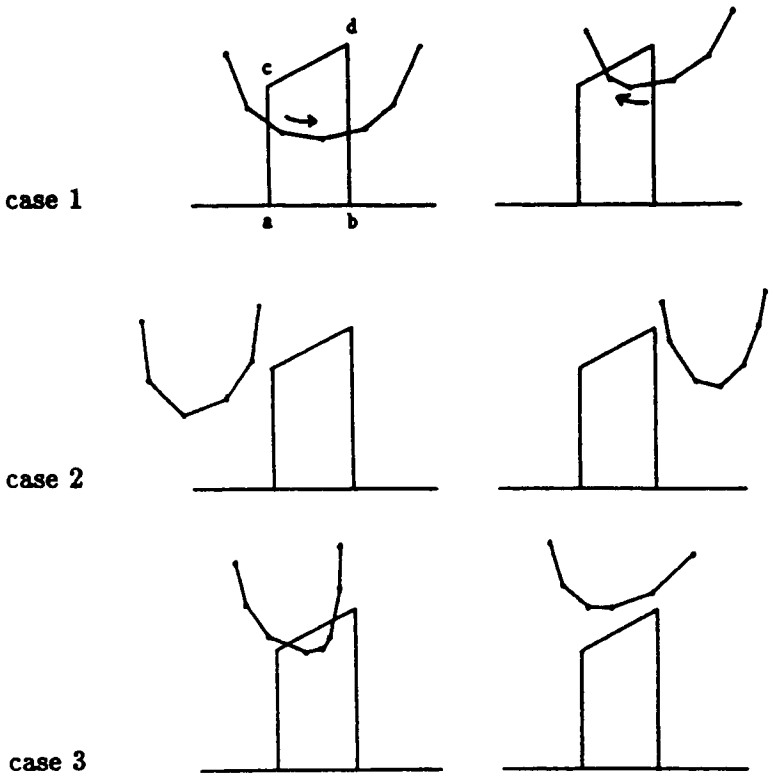


Figure 6. The various cases

(2) The L-peak of $D(v)$ does not lie in R: mark red the children of v (if any).

The description of the algorithm will be complete after a few words on the implementation of its basic primitives. The case-analysis of *Stage 1* is performed by binary search in Abs(v) with respect to a and b. In *Stage 2*, the L-peak of $D(v)$ is computed by performing a binary search in Slope(v) with respect to the slope of L.

The correctness of the algorithm is established with the following observations. First of all, it is clear that each point computed by the algorithm lies in R and is reported only once. Secondly, for each node v examined, all the points of $D(v) \cap V$ are reported. It then suffices to show that each lower hull contributing a point in R is indeed examined. Let U denote the set of vertices that must be examined by a correct algorithm, i.e., $U = \{v | D(v)$ contributes at least one point to $V \cap R\}$. Set

$$U_1 = \{v \in U | D(v) \cap r_a = \phi \text{ and } D(v) \cap r_b = \phi\}$$

and

$$U_2 = \{v \in G | D(v) \cap r_a \neq \phi \text{ or } D(v) \cap r_b \neq \phi\}.$$

The sets U_1 and U_2 contain respectively red and blue vertices. Clearly $(U \setminus U_1) \subseteq U_2$, and this inclusion may often be strict. We omit the proof that:

(1) the path from any vertex of U_1 to the root of G is a sequence of vertices in U_1 followed by a sequence of vertices in U_2 (the latter sequence possibly empty);

(2) the path from any vertex of U_2 to the root of G consists exclusively of vertices in U_2.

These two remarks show that the algorithm visits each vertex of U_1 and U_2, and is therefore correct. Note that the computation of U_2 may fail to contribute any point to the output, although it provides an important guiding mechanism, quite similar to the scheme followed by the priority search tree. In particular, if a red node v has no intersections with the trapezoid, then we never descend in G below v. This allows us to bound the number of such "fruitless" visits by $2(|U_1|+|U_2|)$. The recursive definition of nested lower hulls ensures that U_2 consists of at most two paths, each of length $O(\log n)$. Since each visit of a vertex in U_1 provides at least one output point, we easily bound the running time of the algorithm by $O(\log^2 n + k \log n)$, where k is the output size. The storage required by the algorithm is clearly $O(n)$. The preprocessing time can be kept down to $O(n \log n)$, provided that the points of V are sorted by x-coordinates at the outset of the computation. Repeated Graham scans will provide each lower hull in linear time.

But we now have the stage set for fractional cascading. Visiting vertex v of G involves a binary search in either Abs(v) or Slope(v). The keys to be searched are a, b, or the slope of L. The graph G is of bounded degree and its traversal always involves a subgraph whose vertices are examined in a connected sequence. We immediately conclude.

Theorem 4. *Given a slanted range search problem on n points, there exists a data structure of size $O(n)$ that allows us to answer any query in $O(k+\log n)$ time, where k is the size of the output. The data structure can be constructed in $O(n \log n)$ time and is optimal.*

5. Orthogonal Range Search

Let $\Re^d$ be the real d-dimensional Euclidean space endowed with a Cartesian system of reference $(Ox_1, \ldots, Ox_d)$. A d-range R is a set specified by two points $(a_1, \ldots, a_d)$ and $(b_1, \ldots, b_d)$, with $a_i \leq b_i$: we have

$$R = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d].$$

Let V be a set of n points in $\Re^d$. The *orthogonal range search problem* can be stated as follows: given a query d-range, report all points of $V \cap R$. We direct our interest here to data structures that require only $O(n \log^c n)$ space, for some constant c, and provide a response time of $O(\log n$+output size). As yet, such a data structure has been found only for the case $d = 2$ [Ch1,GBT,W]. We show that one also exists for the case $d = 3$. The algorithm relies on successive reductions to easier problems. We will proceed from the bottom, treating the easy cases first. The desired result is approached through a series of subproblems in which each new subproblem builds on the previous one.

Subproblem P1: Let V be a set of n points in $\Re^2$ and let (Ox, Oy) be a Cartesian system of reference. Consider the problem of computing the set $V(a,b) = \{(x,y) \in V | x \leq a$ and $y \leq b\}$, given any query point (a,b).

This problem can be solved by a number of known data structures, including the priority search tree of McCreight [M2]. To prepare the ground for fractional cascading, however, we must choose a different approach. Consider the set of n vertical rays emanating upward from the points of V. This set consists of the unbounded segments of the form $[(x,y),(x,+\infty)]$, obtained for each point (x,y) of V. To compute $V(a,b)$, it suffices to identify all intersections between these rays and the ray $H = [(-\infty,b),(a,b)]$. We accomplish this task by using the *hive-graph* structure described by Chazelle [Ch1]. This data structure allows us to compute all desired intersections in optimal time and space. Briefly, the hive-graph is a subdivision of the plane built by adding horizontal segments to the original set of rays. Figure 7 illustrates

this construction. Dashed lines correspond to added edges. Without going into the details of the structure, we must mention an essential feature of the query-answering process. To find the intersections between the rays and the segment s, the hive-graph will first ask us to compute the successor of b in some given catalog. The result of the search will then trigger the report of each intersection at unit cost per report. The data structure requires $O(n)$ space and can be constructed in $O(n \log n)$ time.

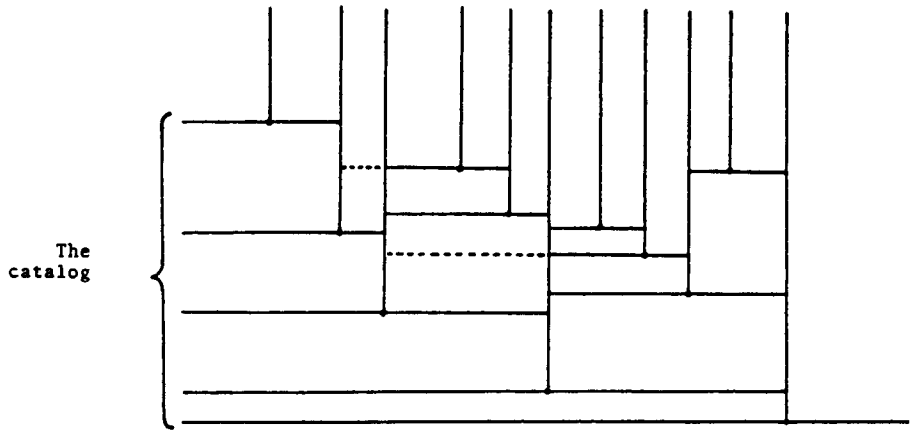


Figure 7. The hive-graph

Subproblem P2: Next, we turn to a restricted case of three-dimensional range search, one where the query R is of the form $[a_1, b_1] \times [0, b_2] \times [0, b_3]$ (figure 8.1). We say that subproblem *P2* is *based* on the two halfspaces $z \geq 0$ and $y \geq 0$.

Let V be a set of points $\{(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\}$, given by their coordinates in a Cartesian system of reference, (Ox, Oy, Oz). We use Bentley's notion of *range tree* [B] to reduce this problem to $O(\log n)$ instances of subproblem *P1*. In $O(n \log n)$ time, relabel the points of V so that $x_1 \leq x_2 \leq \ldots \leq x_n$, and set up a complete binary tree $\mathcal{T}$ whose n leaves correspond respectively to $(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)$ in left-to-right order. Each leaf of $\mathcal{T}$ has a *key*, which we define as the x-coordinate of its associate point. We organize $\mathcal{T}$ as a search tree, so that any successor of an arbitrary value among $\{x_1, \ldots, x_n\}$ can be computed in $O(\log n)$ time. For each vertex v of $\mathcal{T}$, let $U(v)$ be the subset of V induced by the leaves descending from v. Let $P(v)$ be the projection of $U(v)$ on the plane $x = 0$. For each set $P(v)$ we construct the data structure described in the solution of subproblem *P1*. Following the paradigm of the range tree, we can decompose R into a logarithmic number of canonical pieces. To do so, we search for a_1 and b_1 in $\mathcal{T}$. Let v_b be the leaf whose key is the successor of b_1. Symmetrically, consider the leaf whose key is the successor of a_1, and let v_a be its predecessor. For simplicity, we assume that all these nodes are well-defined (special cases can easily be integrated in a unified framework, but to preserve the continuity of the exposition, we will not attempt to do so). Let w_a and w_b be respectively the left and right children of the lowest common ancestor of v_a and v_b. We define W as the set of nodes of $\mathcal{T}$ that are either right children of nodes from v_a to w_a, or left children of nodes from v_b to w_b. Our original problem can be solved by solving it with respect to the point sets associated with the nodes of W. The benefit of this multiplication of work is that each subproblem is of lesser dimensionality. So, the original query can be answered by applying the solution of *P1* to each of the sets $\{P(v) | v \in W\}$. Note that the two-dimensional query for *P1* is specified by the point (b_2, b_3) in the yz-plane. Straightforward analysis shows that the time to preprocess the data

structure is $O(n \log^2 n)$, the space used is $O(n \log n)$, and the response time is $O(k + \log^2 n)$, where k is the size of the output.

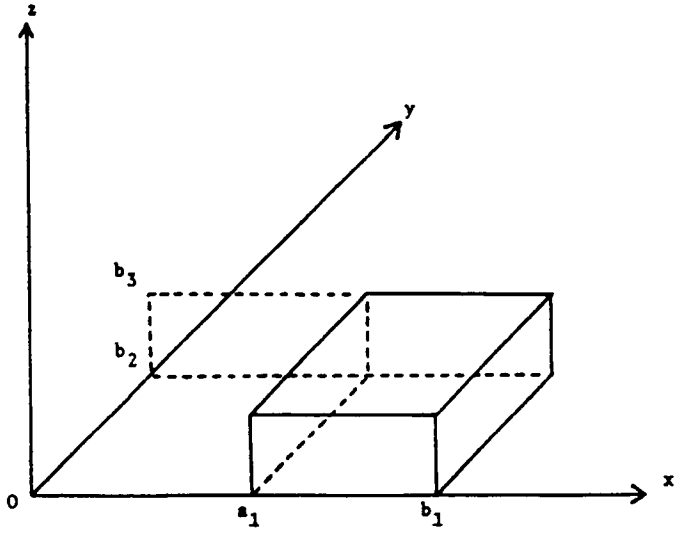


Figure 8.1

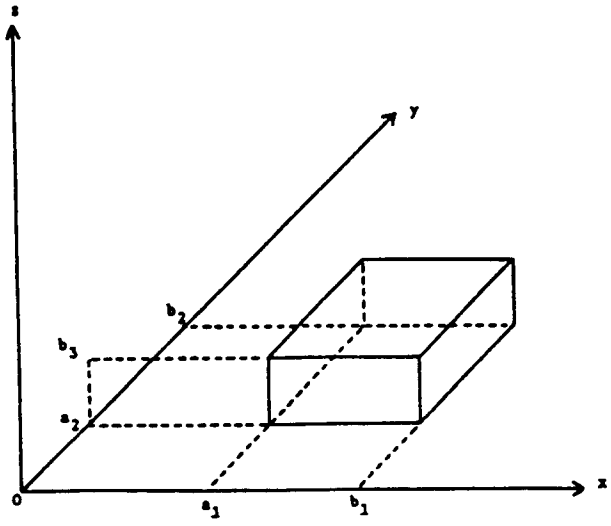


Figure 8.2

Figure 8. Reducing the dimensionality of the query

Subproblem P3: Next, we generalize subproblem *P2* by considering queries of the form $[a_1, b_1] \times [a_2, b_2] \times [0, b_3]$ (figure 8.2). *P3* is said to be *based* on the halfspace $z \geq 0$.

The same complete binary tree $\mathcal{T}$ defined in the previous paragraph is used here, but in a somewhat different way. Let v_1 (resp. v_2) be the left (resp. right) child of the internal node $v \in \mathcal{T}$, and let $d(v)$ be any number at least as large as any x-coordinate in $U(v_1)$ and at least as small as any in $U(v_2)$. Associate with v the data structure of *P2* defined with respect to $U(v_1)$ (resp. $U(v_2)$) and based on $(z \geq 0, x \leq d(v))$ (resp. $(z \geq 0, x \geq d(v))$). The data structure can be constructed in $O(n \log^3 n)$ time and requires $O(n \log^2 n)$ space. How do we answer a query? Starting at the root of $\mathcal{T}$, we compare $d(\text{root})$ against a_1 and a_2.

(1) If $a_1 \leq d(\text{root}) \leq a_2$, then we can apply the solution of *P2*, using the two data structures associated with v. The computation will thus be complete.

(2) If $d(\text{root}) < a_1$, then we iterate on this process by branching to the left child of the root.

(3) If $d(\text{root}) > a_2$, then we iterate on this process by branching to the right child of the root.

The running time of this algorithm will be $O(\log n) + t(n)$, where $t(n)$ is the time to solve two instances of subproblem *P2*. This brings the time complexity to $O(\log^2 n +$ output size).

Subproblem P4: We are ready to return to the original problem: R is now specified by two arbitrary points in $\Re^3$.

We modify the solution of *P3* in the obvious manner. Each node of $\mathcal{T}$ becomes associated with two data structures for solving not *P2* but, of course, *P3*. A similar analysis shows that the storage and preprocessing time leap up to $O(n \log^3 n)$ and $O(n \log^4 n)$, respectively. The response time remains $O(\log^2 n +$ output size).

Let's examine the data structure in its full expansion. What we have is essentially an interconnection of hive-graphs. If we ignore the hive-graphs for a moment, but just concern ourselves with their associated catalogs, we obtain a graph (actually a tree) of degree at most five; a typical node is adjacent to one parent, two children, and two roots of auxiliary structures. This gives us a perfect example of implicit iterative search. Or does it really? To be consistent, we must provide catalogs to all the nodes, and not just a happy few. We can do so by supplementing empty catalogs with a single key $(+\infty)$. We are now in a position to apply fractional cascading to this resulting catalog graph. An immediate savings of a factor $\log n$ in query time will follow.

Theorem 5. *There exists a data structure for three-dimensional orthogonal range search that allows us to answer any query in optimal $O(k + \log n)$ time, where k is the size of the output. The data structure requires $O(n \log^3 n)$ space and can be constructed in $O(n \log^4 n)$ time.*

6. Orthogonal Range Search in the Past

This section does not make use of fractional cascading per se but of its geometric counterpart, the hive-graph [Ch1], already mentioned in Section 5. As we will see in Section 10, however, a hive-graph is a special case of fractional cascading, so the relevance of this material makes its inclusion compelling. Consider the problem of querying a database about its present state as well as about configurations it held at previous times. This task, traditionally known as searching in the *past*, has already been well-researched [DM,O,Ch2,Co]. The question which we ask in this section, however, has not been addressed before. Briefly, it concerns the problem

of recording the previous states of a data structure for orthogonal range search. The question is not only of theoretical interest. Consider the case of a personnel database, where each point of V represents an employee's record. Coordinates indicate attributes such as sex, age, or salary. Over time, employees might be hired, fired, or simply have their records updated (not the first attribute, we hope). A query will then become a pair (R,t), with the meaning: report all points of V that were inside the d-range R at time t. The time range stretches from $-\infty$ to the present. The input is represented as a set V of n points in $\Re^d$; each point p is assigned an interval $[a_p, b_p]$ indicating its lifetime.

Our solution to this problem will be defined in two stages. For the time being, assume that $d = 2$ and disregard the notion of time. The query range R is the Cartesian product $[x_1, x_2] \times [y_1, y_2]$. Using Bentley's range tree [B], we can perform two-dimensional range searching in $O(n \log n)$ space and $O(k + \log^2 n)$ time, where k is the size of the output. The structure is similar to the one defined in the solution of subproblem *P2* (Section 5). As usual, each vertex v of $\mathcal{T}$ is associated with the set $U(v) \subseteq V$ formed by the leaves descending from v. The difference is that with each node v we associate a list $C(v)$ of the points in $U(v)$ sorted by increasing y-coordinates. To answer a query, we perform two binary searches in the tree and retrieve the nodes of the canonical decomposition of the query range R. For each such node v, we compute the points of $C(v)$ whose y-coordinates fall between y_1 and y_2.

All this is very well-known, so where is the novelty of our structure? The key observation is that since within each list examined only y-coordinates are relevant, we can free the x dimension and use it to represent the *lifetime* of each point. The lifetime of a point $p = (p_x, p_y)$ will be represented now by a horizontal segment $[(a_p, p_y), (b_p, p_y)]$. Instead of searching the list $C(v)$, we must now report all the segments of $\{[(a_p, p_y), (b_p, p_y)] | p \in C(v)\}$ that intersect the vertical segment $[(t, y_1), (t, y_2)]$. To do so, we use a hive-graph. This allows us to find all desired t_v intersections in time $O(\log n + t_v)$. Constructing a hive-graph for p segments takes $O(p \log p)$ time and $O(p)$ space [Ch1], so preprocessing time and storage for the overall data structure amount respectively to $O(n \log^2 n)$ and $O(n \log n)$. The query time is $O(\log n + t_v)$ per node, which gives a total of $O(\log^2 n + k)$, where k is the number of points to be reported. Generalization to higher dimensions is straightforward, using Bentley's technique for multidimensional divide-and-conquer [B].

Theorem 6. *It is possible to perform range searching in the past over a set of n d-dimensional points in $O(k + \log^d n)$ time and $O(n \log^{d-1} n)$ space, where k is the size of the output. The preprocessing time is $O(n \log^d n)$.*

7. Computing Locus-Functions

Let V be a set of n 2-ranges in the Euclidean plane $\Re^2$, which we assume endowed with a Cartesian system of reference (Ox, Oy). A 2-range is the Cartesian product of two closed intervals (recall the definition of a d-range in Section 5). We wish to compute functions of the form

$$f : p \in \Re^2 \mapsto f(p) \in \{0, \ldots, n\},$$

where $f(p)$ might be defined as the number of 2-ranges containing p or as the index of the largest (smallest) 2-range containing p; the notion of large or small refers to the area, perimeter, width/height ratio, or any other suitable function of 2-ranges. We characterize this class of functions as follows: a function $G : 2^V \mapsto \{0, \ldots, n\}$ is called *decomposable* if for any partition of a subset $X \subseteq V$ into Y and Z, $G(X)$ can be computed from $G(Y)$ and $G(Z)$ in constant time [BSa]. We restrict our attention to these so-called *locus-functions*. Let

$V(p)$ be the set of 2-ranges containing p; f is a locus-function if there exists a decomposable function G such that $f(p) = G(V(p))$, for any $p \in \Re^2$.

Note that the problem of computing $V(p)$, given any query point p, has been solved optimally in [Ch1]. The fact that f is single-valued makes the problem of computing locus-functions more difficult. For this reason, we resort to a slightly redundant data structure, inspired by Bentley and Wood's segment tree [BW]. We assume that the reader is familiar with this notion. Let $\{x_1, \ldots, x_{2n}\}$ be the x-coordinates of the 2-ranges of V, sorted in non-decreasing order. We construct a $(2n-1)$-leaf complete binary tree G, placing the ith leaf of G from the left in correspondence with the interval $[x_i, x_{i+1}]$. Each vertex v of G has a *span*, $I(v)$, defined as the union of all intervals associated with leaves descending from v. G induces a canonical decomposition of each 2-range of V into $O(\log n)$ canonical parts. With each node v distinct from the root, we associate the subset $\mathcal{R}(v) \subseteq V$ made of 2-ranges whose projections on the x-axis contain the span of v but not the span of v's parent. Vertex v is assigned a catalog $C(v)$ containing the y-coordinates, in sorted order, of all the 2-ranges in $\mathcal{R}(v)$. Note that each 2-range in $\mathcal{R}(v)$ contributes two entries to the catalog.

Let $p = (p_x, p_y)$ and let $f_v(p)$ denote the restriction of f to the subset of 2-ranges in $\mathcal{R}(v)$. Within the vertical slab $\{(x,y)|x \in I(v)\}$, $f_v(p)$ can be computed in $O(\log n)$ time by performing a binary search in $C(v)$ for the key p_y. To do so, it suffices to store the proper answer in each entry of $C(v)$ in preprocessing. We can now respond to any query as follows: in $O(\log n)$ time, compute the set $\pi(p) = \{v \in G|p \in I(v)\}$ by performing a binary search in G for the key p_x. The value of $f(p)$ is obtained by combining together the partial answers $\{f_v(p)|v \in \pi(p)\}$, at a total cost of $O(\log^2 n)$ operations. We omit the analysis of the preprocessing time because of its dependence on the particular function f we are dealing with. If $f(p)$ denotes the number of 2-ranges that contain p, then it is trivial to guarantee an $O(n \log n)$ preprocessing time by scanning each $C(v)$ linearly and updating partial counts on the fly. If f is more exotic, this on-line method might not work, however.

Once again, using Bentley's technique for multidimensional divide-and-conquer [B], we easily generalize this scheme to higher dimensions. All definitions, necessary facts, and algorithms are extended in a straightforward manner to the computation of locus-functions on d-ranges. Each increase of one in dimension adds a factor of $\log n$ in storage and search time.

With the algorithm now described, we identify its iterative search component and apply fractional cascading to improve its performance by a logarithmic factor. For the sake of generality, we consider the case where V consists of n d-ranges. The structure G consists of $d-1$ levels of nested binary trees. Each vertex is adjacent to at most four other vertices (one parent, two children, one root of a structure of lesser dimension). The trees at the lowest level do not have pointers to other tree structures but, instead, have a catalog associated with each of their vertices. For consistency, vertices with no catalogs are assigned dummy catalogs $\{+\infty\}$. Each traversal of G clearly satisfies the connectivity requirement of fractional cascading; we conclude.

Theorem 7. *Given a set of n d-ranges in $\Re^d$, it is possible to compute any locus-function in $O(\log^{d-1} n)$ time, using a data structure of size $O(n \log^{d-1} n)$.*

8. A Space-Compression Scheme

Data structures such as segment-trees [BW] and range trees [B] are suboptimal, space-wise. It is possible to eliminate some of their redundancy and thus save storage, but this entails some degradation in response time. Fractional cascading can be used, however, to slow down the rate of degradation. We illustrate this point by returning to the problem of computing

locus-functions in two dimensions (see Section 7). We will show that the storage can be reduced by a factor $\log\log n$, while increasing the query time by a factor $\log^{\epsilon} n$. We confess that this result is of rather academic interest, and we would not have included it, had it not illustrated the versatility of fractional cascading in such a simple way, as we will see now.

We borrow notation from Section 7. Let $\{x_1, \ldots, x_{2n}\}$ again be the x-coordinates of the 2-ranges of V, sorted in non-decreasing order, and let α be a positive integer. Construct a $(2n-1)$-leaf complete α-ary tree G by placing the ith leaf of G from the left in correspondence with the interval $[x_i, x_{i+1}]$. The *span* of vertex v is defined as before: $I(v)$ is the union of all intervals associated with leaves descending from v. As usual, $\mathcal{R}(v) \subseteq V$ designates the set of 2-ranges whose projections on the x-axis contain the span of v but not the span of v's parent. Unfortunately, storing all these sets is too expensive, so a redefinition of $\mathcal{R}(v)$ is in order. Let $v_1, \ldots, v_\alpha$ be the children of v from left to right and let R be any 2-range of V that appears in at least one $\mathcal{R}(v_k)$ $(k = 1, \ldots, \alpha)$. Note that the indices k such that $R \in \mathcal{R}(v_k)$ (if any) form a consecutive interval $[i, j]$. In general, we will have either $i = 1$ or $j = \alpha$. The inequalities $1 < i \leq j < \alpha$ can take place only at the highest node used in the canonical decomposition of R. For this reason, we can spend freely in the latter case, but we must show restraint in the others. We construct the sets $\mathcal{R}_l(v_k), \mathcal{R}_r(v_k), \mathcal{R}_t(v_k)$ as follows:

(1) If $i = 1$, include R in $\mathcal{R}_l(v_j)$.

(2) If $j = \alpha$, include R in $\mathcal{R}_r(v_i)$.

(3) If $1 < i \leq j < \alpha$, include R in $\mathcal{R}_t(v_i), \ldots, \mathcal{R}_t(v_j)$.

It is easy to understand the whys and wherefores of this construction. Given the interval-like occurrences of R among brother vertices, the collection of sets $\mathcal{R}_l(v_k), \mathcal{R}_r(v_k), \mathcal{R}_t(v_k)$ provides an implicit representation of the collection of sets $\mathcal{R}(v_k)$. With each set $\mathcal{R}_*(v_k)$, we associate the catalog $C_*(v_k)$ defined in Section 7. Note that except for one level each 2-range R can appear at most twice at each level of G. This contributes $O(n\frac{\log n}{\log \alpha})$ to the storage. The exception corresponds to the highest-level occurrences of R, which come in batches of at most α. Consequently, the data structure requires $O(n\frac{\log n}{\log \alpha} + \alpha n)$ space.

To answer a query $p = (p_x, p_y)$, we first collect all vertices whose spans contain p. For each such vertex, we consider the children of its parent in left-to-right order, $v_1, \ldots, v_\alpha$. Let v_i be the vertex in question. For obvious reasons, $f_{v_i}(p)$ can be computed by searching for p_y in the catalogs $C_r(v_1), C_r(v_2), \ldots, C_r(v_i)$, and $C_l(v_i), C_l(v_{i+1}), \ldots, C_l(v_\alpha)$, and if $1 < i < \alpha$, also $C_t(v_i)$. This scheme yields an overall $O(\alpha\frac{\log^2 n}{\log \alpha})$ response time.

A standard binary representation of G allows us to apply fractional cascading (see Knuth [K], for example). Let $v_1, \ldots, v_\alpha$ be the children of v from left to right. We remove all pointers from v to $v_2, \ldots, v_k$, and replace them by pointers from v_i to v_{i+1}, for $i = 1, 2, \ldots, \alpha - 1$ (figure 9). To each node v, we now attach a little chain of three consecutive nodes, assigned to the catalogs $C_t(v)$, $C_r(v)$, and $C_l(v)$, whenever these are well-defined. The data structure forms a catalog graph of bounded degree. Application of fractional cascading immediately takes the running time down to $O(\alpha\frac{\log n}{\log \alpha} + \log n)$. Setting $\alpha = \lfloor (\log n)^{\epsilon} \rfloor$, we obtain the following result.

Theorem 8. *Given a set of n 2-ranges in $\Re^2$ and any positive real ϵ, it is possible to compute a locus-function in $O(\log^{1+\epsilon} n)$ time, using $O(n\frac{\log n}{\log\log n})$ space.*

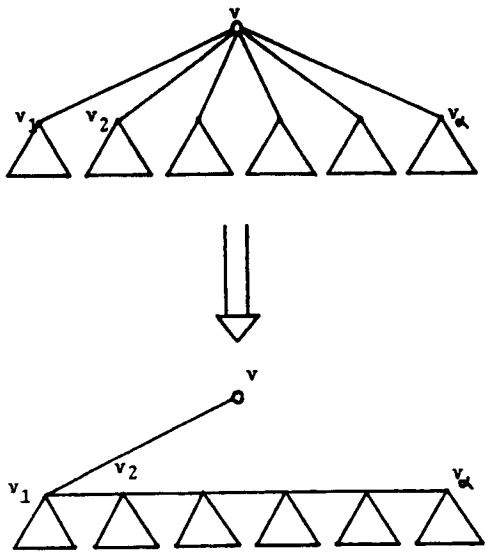


Figure 9. Putting each node in normal form

9. Iterative Search Extensions of Query Problems

In practice one is often faced with query problems which are not quite the standard problems studied in the literature, but natural generalizations thereof. A typical occurrence of orthogonal range search (Section 5) can be found in a personnel division's database. A query involves retrieving the names of all employees whose attributes fall in a certain range. What is often desired, however, is not so much the names of the employees but additional information about them. To satisfy this request will involve looking up some files (or catalogs) associated with each employee. Unfortunately, this extra work cannot be nicely integrated within a more general range search problem. The only recourse is then to search separately the files of each employee selected by the range search. In the best case, this may multiply the running time of the algorithm by a logarithmic factor.

We will show that with a little care *asymptotically no extra work* need be done in order to retrieve the complementary information desired. This result does not apply only to range search but to a host of other query problems. One advantage of our approach is its generality. We investigate a number of algorithms for query problems and show that by a *generic* modification each can be made to accommodate the additional requests mentioned above. Before proceeding any further, we must formalize this notion of *additional request.*

Consider the following class of problems: let V be a data set, Q a (finite or infinite) query domain, and P a predicate defined for each pair in $V \times Q$. Preprocess the set V so that the function g defined as follows can be computed efficiently:

$$g : q \in Q \mapsto g(q) \in 2^V; \quad g(q) = \{v \in V | P(v, q) \text{ is true } \}.$$

In the orthogonal range search problem, V is a set of points in $\Re^d$, q is a *d-range* and $g(q)$ is the set $V \cap q$. For any query problem Π we define an iterative search problem Π^*: each element $v \in V$ is associated with a distinct catalog $C(v)$ defined over a totally ordered set X. A query for Π^* is a pair (q, x) in $Q \times X$; the problem is to compute the successor of x in each catalog of $\{C(v) | v \in g(q)\}$.

Definition . Problem Π^* is called the *IS-extension* of problem Π.

The term "IS-extension" is a short-hand for iterative search extension. One nice feature shared by many algorithms for query problems is that they operate on graph structures. The memory is often organized as a tree, a dag, or more generally a graph of bounded degree, which the algorithm traverses in a connected manner when answering a query. This feature allows us to transform these algorithms generically via fractional cascading. We next characterize the class of algorithms to which these transformations apply. This leads to the definition of a *retrieval reference algorithm* or *RRA* for short. Let $\mathcal{A}$ be an algorithm for problem Π; we say that $\mathcal{A}$ is an RRA if and only if:

(1) The underlying data structure of $\mathcal{A}$ is a graph G of bounded degree. Each vertex of G is associated with at most one element of V, but elements of V may appear in several vertices.

(2) The output of $\mathcal{A}$, i.e., $\{v \in V | P(v,q)$ is true $\}$, is a subset of the data stored at the vertices visited during the computation.

(3) The computation is modelled by a sequence of *stages*, each of which corresponds to one or several actual steps of the algorithm. To each stage t corresponds a vertex $v(t) \in G$; for each $v(t)$ (except for at most a constant number of them) there exists an edge of the form $(v(t'), v(t))$ with $t' < t$.

(4) The mapping between $v(0), v(1), \ldots$ and the steps of the algorithm is trivial. Transforming the algorithm so that it outputs the name of the current vertex $v(t)$ at each step can always be done without slowing down the algorithm by more than a constant factor.

Note that these requirements do not in any way define a model of computation. These are only necessary and sufficient requirements for an algorithm to be an *RRA*. We will find that although a number of algorithms for query problems can be immediately seen as *RRA*'s, many others have to undergo minor transformations in order to be readily recognized as such. Here are some examples of query problems which admit of *RRA*'s. This list is given for illustrative purposes and is not meant to be comprehensive.

(a) **Interval Overlap**: Given a set V of intervals and a query interval q, report the intervals of V that intersect q [Ch1,E,M1,M2].

(b) **Segment Intersection**: Given a set V of segments in the plane and a query segment q, report the segments of V that intersect q [Ch1,DE,EKM].

(c) **Point Enclosure**: Given a set V of d-ranges and a query point q in $\Re^d$, report the d-ranges of V that contain q [Ch1,E].

(d) **Orthogonal Range Search**: Given a set V of points in $\Re^d$ and a query d-range q, report the points of V that lie inside q [B,Ch3,GBT,M2,W].

(e) **Rectangle Search**: Given a set V of d-ranges and a query d-range q, report the d-ranges of V that intersect q [Ch3,GBT].

(f) **Triangle Retrieval**: Given a set V of points in E^2 (resp. E^3) and a query triangle (resp. tetrahedron) q, report the points of V that lie within x [CY,EH,EW,Y].

(g) **Circular Range Query**: Given a set V of points in E^2 and a query circle q, report the points of V that lie within q [CCP].

(h) **k-Nearest-Neighbor**: Given a set V of points in E^2 and a query of the form (q,k); $q \in E^2$, k integral ≥ 0, report the k points of V closest to q [CCP].

Retrieval reference algorithms are best understood in the broader context of the pointer machine model [T]. This model includes most algorithms free of address calculations: this rules out, for example, hashing, radix sort, and operations on dense matrices. In the pointer machine model, the memory is represented by a directed graph with one vertex per piece of data and one edge per pointer. The computation involves visiting vertices of the graph in such a way that going from one vertex to another requires the presence of a directed edge from the origin to the destination. New pointers are provided by requesting new memory cells from a free list; they cannot be created by arithmetic operations. Sometimes, solutions to query problems do require address calculations to perform binary search in linear arrays. This is not a major handicap, however, since it is easily fixed by substituting balanced search trees for arrays. With these remarks, checking each of the references accompanying the problems listed above leads to the straightforward conclusion:

Lemma 7. *All solutions to the eight problems referenced above (which include the most efficient known to date) are of the type RRA.*

The main result of this section states that any *RRA* for a query problem Π can always be generically transformed into an algorithm for solving its IS-extension. To alleviate the notation, we make the simplifying assumption that the catalogs are each of the same size m.

Theorem 9. *Let Π be a query problem defined over a set V of size p, and let $\mathcal{A}$ be an RRA for solving Π. Assume that $\mathcal{A}$ requires $O(f(p))$ space and has $O(g(p)+k)$ response time, where k is the size of the output. Let Π^* be the IS-extension of Π obtained by associating a catalog of size m with each element in V. Then there exists a data structure for solving Π^*, which requires $O(mf(p))$ space and $O(\log m + g(p) + k)$ response time.*

Proof: Let G be the graph used in modelling $\mathcal{A}$ as an *RRA*. To each vertex of G corresponds at most one element of V, hence one catalog (possibly reduced to $+\infty$ if the vertex does not store any element). Since T has bounded degree, we can apply fractional cascading to its associated set of catalogs. To answer a query, look up the search key x in the catalog associated with $v(0)$; at any subsequent step $t > 0$ retrieve the relevant successor in the catalog associated with $v(t)$. ∎

Since both interval overlap and point enclosure can be solved in optimal space and time, so can their IS-extensions [Ch1]. If $m = O(n)$, the algorithms for each of the other problems mentioned above have the same complexity as the algorithms for their IS-extensions. In general, note that since the function f grows at least linearly, the storage used for solving Π^* is also $O(f(n))$, where $n = pm$ is the size of the input. The naive algorithm for solving Π^* consists of applying $\mathcal{A}$ and looking up the search key x in each of the k catalogs found. This scheme uses only $O(n + f(p))$ space but may need as much as $O(g(p) + k \log m)$ time.

10. Other Applications

To illustrate the wide applicability of fractional cascading, we wish to report briefly on other related work. The idea of propagating fractional samples has already been used in a number of different specific contexts [Ch1,Co,EGS]. Interestingly, in all three cases, fractional cascading provides a unifying framework in which to understand these results. Let's take the case of the *hive-graph*, for example. We briefly recall this technique (see [Ch1] for details). Given a

set of horizontal segments, construct a planar subdivision by adding, for each endpoint p, the longest vertical segment passing through p that does not properly intersect any horizontal segment. This is our base subdivision (figure 10). We refine it by adding new vertical segments, so that every face ends up with at most a constant number of vertices. As we can see, it is not immediate that such a property can be ensured without adding a quadratic number of segments. The novelty of [Ch1] was to show that by propagating only *every other* vertical segment, the size of the planar subdivision remains linear.

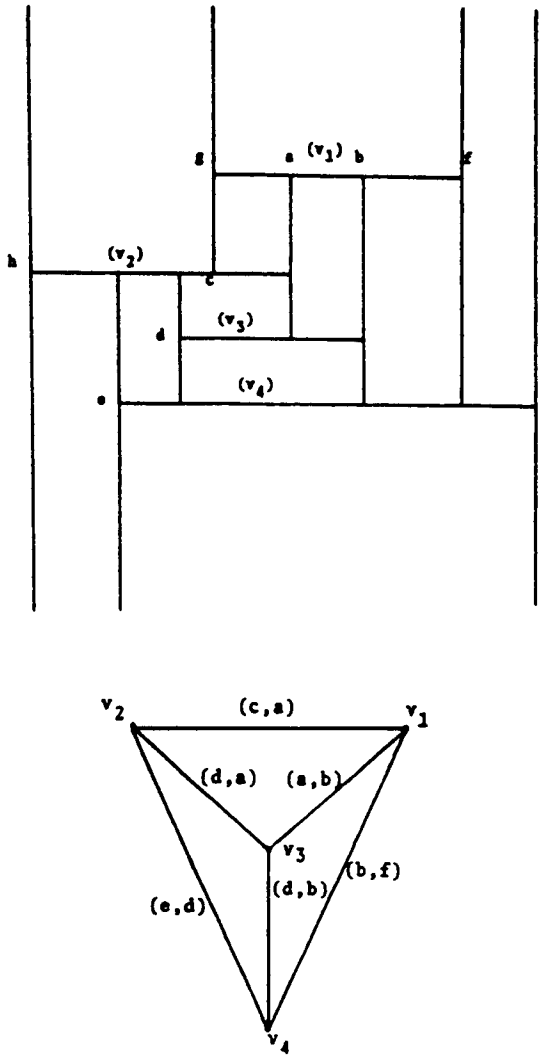


Figure 10. The base subdivision

How can we interpret this result in terms of fractional cascading? Every horizontal segment corresponds to a node of the catalog graph; catalogs are made of the x-coordinates of the

vertices on each segment; edges connect nodes whose corresponding segments are visible from each other (where segment a is visible from segment b if there exists a vertical segment that connects a and b, and does not intersect any other segment). In figure 10, for example, node v_1 is adjacent to v_2, v_3, and v_4. Its catalog is the list of x-coordinates $\{g, a, b, f\}$.

Other results that can be interpreted in terms of fractional cascading or that make explicit use of it include algorithms for

(a) *Planar point location*: locate a point in a planar subdivision [Co,EGS].

(b) *Point enclosure*: find d-ranges containing a query point [Ch1].

(c) *Homothetic range search*: report the points falling in a query 2-range of fixed aspect-ratio [CE].

(d) *3d-Domination search*: range search in $\Re^3$ for queries of the form $[0, a] \times [0, b] \times [0, c]$ [CE].

(e) *Intersection search*: find the intersection of a polygon with a query segment [CG].

11. Concluding Remarks

The contribution of this paper has been to show the versatility of a new data structuring technique, called *fractional cascading*. The technique seems simple and general enough to have many practical applications. Besides those studied in this paper, one should mention the relevance of fractional cascading to external searching in general. Since it works on a pointer machine, fractional cascading can handle situations where the collection of catalogs is very large, but where each of them can be stored on one or a small number of pages. It would be interesting to determine if such a scheme can outperform hashing techniques in practice.

One of the most interesting open problems is to determine whether fractional cascading extends to higher dimensions. Imagine that a catalog is a planar subdivision, and the "successor" of a query point is the name of the face that contains it. Can iterative search be speeded up? As usual, we may try to merge all the subdivisions into one master subdivision. The catch is that merging together two subdivisions of respective size l and m may result in a subdivision of size $\Theta(lm)$. This contrasts with the nice property of linear lists: merging two of them only adds their sizes. Why is this extension so important, anyway? Various data structures for near-neighbor problems involve a hierarchy of Voronoi diagrams. A query involves selecting a few of them and performing repeated point locations. Results similar to the ones we have obtained with fractional cascading would bring about dramatic improvements to the best solutions known to date.

Acknowledgments: We wish to thank Bob Tarjan for his many helpful comments and suggestions. The proof of Lemma 1, in particular, is due to him. We also thank Cynthia Hibbard for her many suggestions that improved the exposition.

References

[B] Bentley, J.L. *Multidimensional divide-and-conquer*, Comm. ACM, 23, 4 (1980), 214–229.

[BKZ] van Emde Boas, P., Kaas, B., and Zijlstra, E. *Design and implementation of an efficient priority queue*, Math. Syst. Theory **10**, 1977, pp. 99–127.

[BSa] Bentley, J.L., Saxe, J.B. *Decomposable searching problems I: static to dynamic transformations*, J. of Algorithms 1 (1980), 301–358.

[BS] Bentley, J.L., Shamos, M.I. *A problem in multivariate statistics: Algorithms, data structures and applications*, Proc. 15th Allerton Conf. Comm., Contr., and Comp. (1977), 193–201.

[BW] Bentley, J.L., Wood, D. *An optimal worst-case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., Vol. C–29 (1980), 571–577.

[Ch1] Chazelle, B. *Filtering search: A new approach to query-answering*, Proc. 24th Ann. Symp. Found. Comp. Sci. (1983), 122–132. To appear in SIAM J. on Computing, 1986.

[Ch2] Chazelle, B. *How to search in history*, Information and Control, 1985.

[Ch3] Chazelle, B. *A functional approach to data structures and its use in multidimensional searching*, Brown Univ. Tech. Rept, CS–85–16, Sept. 1985 (preliminary version in 26th FOCS, 1985).

[CCP] Chazelle, B., Cole, R., Preparata, F.P., Yap, C.K. *New upper bounds for neighbor searching*, Tech. Rept. CS–84–11 (1984), Brown Univ.

[CE] Chazelle, B., Edelsbrunner, H. *Linear space data structures for a class of range search*, to appear in Proc. 2nd ACM Symposium on Computational geometry, 1986.

[CG] Chazelle, B., Guibas, L.J. *Visibility and intersection problems in plane geometry*, Proc. 1st ACM Symposium on Computational Geometry, Baltimore, MD, pp. 135–146, June 1985.

[CGL] Chazelle, B., Guibas, L.J., Lee, D.T. *The power of geometric duality*, BIT, 25 (1), 1985. Also, in Proc. 24th Ann. Symp. Found. Comp. Sci. (1983), 217–225.

[Co] Cole, R. *Searching and storing similar lists*, Tech. Report No. 88, Courant Inst., New York Univ. (Oct. 1983). To appear in J. Algorithms.

[CY] Cole, R., Yap, C.K. *Geometric retrieval problems*, Proc. 24th Ann. Symp. Found. Comp. Sci. (1983), 112–121.

[DE] Dobkin, D.P., Edelsbrunner, H. *Space searching for intersection objects*, Proc. 25th Ann. Symp. Found. Comp. Sci. (1984).

[DM] Dobkin, D.P., Munro, J.I. *Efficient uses of the past*, Proc. 21st Ann. Symp. Found. Comp. Sci. (1980), 200–206.

[E] Edelsbrunner, H. *Intersection problems in computational geometry*, Ph.D. Thesis, Tech. Report, Rep. 93, IIG, Univ. Graz, Austria (1982).

[EGS] Edelsbrunner, H., Guibas, L.J., Stolfi, J. *Optimal point location in a monotone subdivision*, to appear in SIAM J. Comp. Also DEC/SRC research report no. 2, 1984.

[EH] Edelsbrunner, H., Huber, F. *Dissecting sets of points in two and three dimensions*, forthcoming technical report, IIG, Univ. Graz, Austria, 1984.

[EKM] Edelsbrunner, H., Kirkpatrick, D.G. Maurer, H.A. *Polygonal intersection search*, Inform. Process. Lett. 14 (1982), 74–79.

[EW] Edelsbrunner, H., Welzl, E. *Halfplanar range search in linear space and $O(n^{0.695})$ query time*, Tech. Report, F–111, IIG, Univ. Graz, Austria (1983).

[FMN] Fries, O., Mehlhorn, K., and Näher, St. *Dynamization of geometric data structures*, Proc. 1st ACM Computational Geometry Symposium, 1985, pp. 168–176.

[GBT] Gabow, H.N., Bentley, J.L., Tarjan, R.E. *Scaling and related techniques for geometry problems*, Proc. 16th Ann. SIGACT Symp. (1984), 135–143.

[GT] Gabow, H. N., and Tarjan, R. E. *A linear-time algorithm for a special case of disjoint set union*, Proc. of 24-th FOCS Symposium, 1983, pp. 246–251.

[IA] Imai, H. and Asano, T. *Dynamic segment intersection search with applications*, Proc. of 25-th FOCS Symposium, 1984, pp. 393–402.

[K] Knuth, D.E. *The art of computer programmming, sorting and searching*, Vol. 3, Addison-Wesley, 1973.

[LP] Lee, D.T., Preparata, F.P. *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., Vol. 6, No. 3, pp. 594–606, Sept. 1977.

[M1] McCreight, E.M. *Efficient algorithms for enumerating intersecting intervals and rectangles*, Tech. Rep., Xerox PARC, CSL–80–9 (June 1980).

[M2] McCreight, E.M. *Priority search trees*, Tech. Rep., Xerox PARC, CSL–81–5 (1981).

[O] Overmars, M.H. *The design of dynamic data structures*, PhD Thesis, University of Utrecht, The Netherlands, 1983.

[PH] Preparata, F.P., Hong, S.J. *Convex hulls of finite sets of points in two and three dimensions*, Comm. ACM, vol 20, (1977), 87–93.

[Ta] Tarjan, R.E. *Amortized computational complexity*, SIAM J. on Comp., to appear.

[T] Tarjan, R.E. *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), 110–127.

[VW] Vaishani, V.K., and Wood, D. *Rectilinear line segment intersection, layered segment trees, and dynamization*, J. Algorithms, vol. 3, 1982, pp. 160–176.

[W] Willard, D.E. *New data structures for orthogonal queries*, to appear in SIAM J. Comput.

[Y] Yao, F.F. *A 3–space partition and its applications*, Proc. 15th Annual SIGACT Symp. (1983), 258–263.

Index

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Report #1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Report #2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Report #3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Report #4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Report #5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Report #6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Report #7, November 14, 1985.

"On Interprocess Communication."
Leslie Lamport.
Report #8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Report #9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Report #10, May 1st, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Report #11, May 5, 1986.

by Bernard Chazelle and Leonidas J. Guibas

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Ordered hash tables

O. Amble and D. E. Knuth

University of Oslo and Stanford University, Computer Science Department, Stanford, California 94305, USA

Some variants of the traditional hash method, making use of the numerical or alphabetical order of the keys, lead to faster searching at the expense of a little extra work when items are inserted. This paper presents the new algorithms and analyses their average running time.

(Received June 1973)

Keywords and phrases: Searching, hash tables, analysis of algorithms, address calculation.

Traditional methods of search are usually based either on the numerical or alphabetical ordering of keys (e.g. binary search), or on the keys' arithmetical properties (e.g. hashing). By combining these two approaches it is possible to obtain methods which are often superior to the traditional algorithms.

In this paper we shall discuss a new class of search procedures which use both the idea of ordering and the idea of 'open' hash addressing. A mathematical analysis of the expected running time is also given.

Definitions

Given a file or *table* of data containing N distinct *keys* $K_1, K_2, \ldots, K_N$, the search problem consists of taking a given *argument* K and determining whether or not $K = K_i$ for some i. In practice the key K_i is part of a larger record of information, R_i, which is being retrieved via its key; but for the purposes of our discussion we may concentrate solely on the keys themselves, since they are the only things which significantly enter into the search algorithms. If the search argument K is not in the table, we sometimes want to put it in; therefore we are generally interested in two algorithms, one for searching and one for insertion. The recent book by Knuth (1973) contains an extensive account of the algorithms which are commonly used for searching and insertion.

One of the important families of search algorithms is the so-called method of 'open addressing with double hashing', which works as follows. The table is stored in a larger array of M positions, numbered 0 through $M - 1$. If U is the universe of all possible keys that might ever be sought (e.g. U might be all n-bit numbers or all n-character identifiers, for some n), we define two functions for each K in U, namely

$$h(K) = \text{the 'hash address' of } K,$$
$$i(K) = \text{the 'hash increment' of } K.$$

These functions are constrained so that $0 \leq h(K) < M$ and $1 \leq i(K) < M$ and $i(K)$ is relatively prime to M, for all K. Thus if $M = 2^m$, $i(K)$ is allowed to be any odd positive number less than M; alternatively if M is prime, $i(K)$ is allowed to be any positive number less than M. For best results these functions are usually chosen to be efficiently computable, yet with the property that distinct keys will tend to have different hash addresses.

Some of the M positions of the hash table are unoccupied, while N of the positions contain keys. For convenience we shall assume that all keys have a strictly positive numeric value. The entries of the hash table will be denoted by $T_0, T_1, \ldots, T_{M-1}$, where $T_j = 0$ if that position is empty and $T_j > 0$ if T_j is the key stored in position j.

Algorithms

Using these definitions, it is possible to describe the conventional algorithm for open addressing with double hashing as follows.

Algorithm A. Let K be the search argument.
Step A1. Set $j \leftarrow h(K)$.
Step A2. If $T_j = K$, the algorithm terminates 'successfully'.
Step A3. If $T_j = 0$, the algorithm terminates 'unsuccessfully'.
Step A4. Set $j \leftarrow j - i(K)$. If now $j < 0$, set $j \leftarrow j + M$. Return to step A2. □

The search is said to be 'successful' or 'unsuccessful' according as K has been found or not. After a successful search, it is possible to fetch the entire record having the given key.

A new record may be inserted into such a table by first searching for its key K; when the algorithm terminates unsuccessfully in step A3, the new record may be placed into the jth position of the table. Subsequent searches for this key will follow the same path to position j.

The fact that $i(K)$ is relatively prime to M ensures that no part of the table is examined twice, until all M locations have been probed. Since we assume that there is at least one empty position, the search must terminate if K is not present.

The above algorithm includes several noteworthy special cases. If $i(K)$ is identically 1 for all K, it is the well-known method of *linear probing*. If $i(K) = 1$ and $h(K) = M - 1$ for all K, it reduces to the straightforward method of *sequential scanning*. If $i(K) = f(h(K))$ where f is a more-or-less random function, the algorithm is called double hashing with *secondary clustering*. On the other hand, if the probability that $h(K) = h(K')$ and $i(K) = i(K')$, for distinct keys K and K' in U, is $1/M\varphi(M)$, i.e. if each of the possible values of the pair $(h(K), i(K))$ is equally likely, the method is called *independent double hashing*.

Algorithm A makes decisions only by testing for equality vs. inequality. By using the numerical order of keys we obtain a new algorithm which is almost identical to the other:

Algorithm B. (Searching in an ordered hash table.)
Step B1. Set $j \leftarrow h(K)$.
Step B2. If $T_j = K$, the algorithm terminates 'successfully'.
Step B3. If $T_j < K$, the algorithm terminates 'unsuccessfully'.
Step B4. Set $j \leftarrow j - i(K)$. If now $j < 0$, set $j \leftarrow j + M$. Return to step B2. □

Only step B3 has changed, and in a trivial way. Unsuccessful searches will now be faster.

Of course we cannot use Algorithm B unless the positions of the hash table have been filled in a suitable way. If the keys have been inserted in decreasing order by the ordinary method (i.e. if we start with an empty table, then insert the largest key, then the second-largest, etc.), it is easy to see that Algorithm

The preparation of this paper was supported in part by Norges Almenvitenskapelige Forskningsrad, and in part by the US Office of Naval Research under grant number ONR 00014-67-A-0112-0057 NR 044-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

B will work properly. This proves that there is always an arrangement of keys such that Algorithm B is valid.

Of course in practice we need to be able to insert keys in arbitrary order, as they arrive 'on line'. The following method can be used:

Algorithm C. (Insertion into an ordered hash table.)
Assume that $K \neq T_j$ for $0 \leq j < M$, and that $N \leq M - 2$.
Step C1. Set $j \leftarrow h(K)$.
Step C2. If $T_j = 0$, set $T_j \leftarrow K$ and terminate.
Step C3. If $T_j < K$, interchange the values of $T_j \leftrightarrow K$.
Step C4. Set $j \leftarrow j - i(K)$. If now $j < 0$, set $j \leftarrow j + M$.
Return to step C2. □

During this algorithm, the variable K takes on a decreasing sequence of values, and the increments in step C4 will vary (in general). This is a rather peculiar state of affairs, in spite of the innocuous appearance of Algorithm C, so it is helpful to look at an example.

Suppose that $M = 11$ and that there are $N = 8$ keys

145, 293, 397, 458, 553, 626, 841, 931,

where the middle digit is the h-value and the rightmost digit is the i-value; thus, $h(293) = 9$ and $i(293) = 3$. Then the keys may be distributed in the T table as follows:

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
0	0	626	931	841	553	293	0	458	397	145

The reader may verify that Algorithm B will indeed retrieve each of these keys properly. Now if we wish to insert the new key 759, Algorithm C first replaces T_5 by 759 and sets $K \leftarrow 553$; after examining $T_2 = 626$, it sets $T_{10} \leftarrow 553$, $K \leftarrow 145$; and eventually $T_0 \leftarrow 145$. The table for all nine keys is therefore

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
145	0	626	931	841	759	293	0	458	397	553

To verify that Algorithm C is correct, consider the *path* corresponding to key K, namely the sequence of table position numbers

$$h(K),\ h(K) - i(K),\ h(K) - 2i(K),\ \ldots,\ h(K) - (M-1)\,i(K) \pmod{M}.$$

Since $i(K)$ is relatively prime to M, this sequence consists of the numbers $0, 1, \ldots, M-1$ in some order. Algorithm B works properly if and only if, for every key $K = T_j$ in the table, we do not have $K > T_{j'}$ for some j' which appears *earlier* than j in the path corresponding to K. (This is the essential 'invariant' which is relevant to formal proofs of Algorithm B.) Since Algorithm C never decreases the value of any table position, it preserves this condition.

Analyses

Now let us attempt to determine how much faster (if at all) the new algorithms will go. The following uniqueness theorem is very helpful in this regard.

Theorem:
A set of N keys $K_1, \ldots, K_N$ can be arranged in a table $T_0, T_1, \ldots, T_{M-1}$ of $M > N$ positions in one and only one way such that Algorithm B is valid.

Proof:
We have observed that at least one arrangement is possible. Suppose that there are at least two, and let K_j be the largest key which appears in different positions in two different arrangements. Thus, all keys larger than K_j occupy fixed positions in all possible arrangements. If we look at the path corresponding to K_j, as defined above, the positions of keys larger than K_j are predetermined; and all keys smaller than K_j must occur later than K_j. Therefore K_j must occupy the first vacant place in its path, after the larger keys, contradicting the assumption that K_j can appear in different places. □

To know the behaviour of these search algorithms, we want to know the corresponding average number of iterations or *probes* in the table, i.e. the average number of times steps A2, B2, or C2 are performed respectively. (Only the average number is generally considered in discussions of hashing, since the worst case is too horrible to contemplate.)

The classical Algorithm A has been extensively investigated (see Knuth, 1973 for a review of the literature), and the results can be summarised as follows. Let $\alpha = N/M$ be the 'load factor' of the hash table. Let A_N be the average number of times step A2 is performed in a random successful search, and let A'_N be the corresponding number in a random unsuccessful search. By 'random' and 'average' we mean that the hash addresses of the keys are assumed to be independent and uniformly distributed in the range 0 through $M - 1$, and that each of the N keys of the table is equally likely in a successful search. Then the following approximate formulas have been derived, as M and N approach infinity:

Increment method	A_N	A'_N
linear probing	$\frac{1}{2}(1 + (1-\alpha)^{-1})$	$\frac{1}{2}(1 + (1-\alpha)^{-2})$
secondary clustering	$1 - \ln(1-\alpha) - \frac{1}{2}\alpha$	$(1-\alpha)^{-1} - \ln(1-\alpha) - \alpha$
independent double hashing	$-\alpha^{-1}\ln(1-\alpha)$	$(1-\alpha)^{-1}$

Since the number of probes needed to retrieve an item with Algorithm A is the same as the number needed to insert it, the average number of probes needed to find the kth item inserted is A'_{k-1}. It follows that

$$A_N = (A'_0 + A'_1 + \ldots + A'_{N-1})/N. \qquad (1)$$

Now let us consider the performance of Algorithm B. We shall assume that there is no significant correlation between the hash addresses and the numerical ordering of the keys. Since the position of any fixed set of keys in the table is unique, we may as well assume that they have been inserted in decreasing order. Then the insertion algorithm is identical to that used with Algorithm A, and the average number of probes needed to find the kth largest item is A'_{k-1}. It follows that

$$B_N = (A'_0 + A'_1 + \ldots + A'_{N-1})/N = A_N. \qquad (2)$$

In other words, Algorithm B is equivalent to Algorithm A with respect to successful searching, on the average.

In an unsuccessful search with Algorithm B, the number of probes is the same as would be required in a *successful* search if the keys were $\{K_1, K_2, \ldots, K_N, K\}$ instead of $\{K_1, K_2, \ldots, K_N\}$. Therefore

$$B'_N = B_{N+1} = A_{N+1}. \qquad (3)$$

The above formulas for A_N and A'_N show that this is indeed an improvement. For example, when $\alpha = 0{\cdot}90$ (i.e. when the table is 90% full), the quantities for unsuccessful search are

increment method	A'_N	B'_N
linear probing	50·5	5·500
secondary clustering	11·4	2·853
independent double hashing	10·0	2·558

As $\alpha \to 1$, the ratio B'_N/A'_N approaches 0.

Finally let us investigate the new cost of insertion with Algorithm C. Let C_N be the average number of times step C2 is performed when inserting the Nth item. Each time we execute

step C2, we increase by one the total number of probes needed to find one of the keys. Thus, if we sum over N insertions, we must have

$$C_1 + \ldots + C_N = NA_N .$$

This equation together with (1) implies that

$$C_N = A'_{N-1} . \qquad (4)$$

In other words, the average number of probes needed to insert a new item is exactly the same as it was with Algorithm A.

It is worth noting that the probability distribution of C_N is not in general the same as that of A'_{N-1}, although the average value is the same. In fact, a single insertion with Algorithm C might take up to order N^2 iterations (although such an event is extremely rare). Consider again the case of three-digit keys whose middle digit is the h-value and whose rightmost digit is the i-value; and let $M = 10$. Then the insertion of 949 into the table

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
109	319	529	739	841	651	461	271	0	0

is amazingly slow, as the reader may verify. In general, the table might contain n keys in 'organ-pipe order',

$$0 = T_n < T_0 < T_{n-1} < T_1 < \ldots < T_{\lfloor n/2 \rfloor} ,$$

and we might have

$$i(T_j) = \begin{cases} M - 1 & \text{for } 0 \le j < \lceil n/2 \rceil , \\ +1, \ 1, & \text{for } \lceil n/2 \rceil \le j < n; \end{cases}$$

then the insertion of a new largest key whose hash address is $\lfloor n/2 \rfloor$ will take maximum time, namely $(n + 1)n/2 + 1$ iterations of step C2.

We have now analysed the average number of iterations in both Algorithms B and C. The analysis isn't complete, however, because we have not determined the average number of *interchanges* performed in step C3. This is an important consideration, since it is the number of times we need to compute an increment $i(K)$; with Algorithm A, the increment needs to be computed only once. Therefore let D_N be the average number of times the operation $T_j \leftrightarrow K$ is performed in step C3 while inserting the Nth item.

Unfortunately the analysis of D_N is complicated, and we must defer the calculations to Appendix 1. It turns out that D_N is approximately $(1 - \alpha)^{-1} + \alpha^{-1} \ln (1 - \alpha)$ for linear probing, and approximately equal to $A_N - 1$ for independent double hashing.

Further development

The above algorithms can be extended in various ways, to gain further improvements. For example, it is easy to see that the ideas can immediately be generalised to the case of external searching, where each of the M table positions is a 'bucket' containing b or less keys for some given b.

Another type of extension will make unsuccessful searching still faster, at the expense of M more bits of memory. Let $B_0, B_1, \ldots, B_{M-1}$ be a vector of bits with all B_j initially 0. Suppose that we set $B_j \leftarrow 1$ in step C3 of the insertion algorithm, so that $B_j = 1$ if and only if some successful search 'passes through' position j. Then if the search algorithm ever gets to step B3 and finds $B_j = 0$, the search must be unsuccessful.

This extra-bit approach applies, of course, to unordered hash tables as well as ordered ones, but it is especially attractive in the ordered case because the extra testing can be done with almost no cost. We can combine the bit test with the ordinary test if we assume that each bit B_j appears at the left of T_j as a new significant bit. Then Algorithm B can be rewritten as follows.

Step B1. Set $j \leftarrow h(K)$.

Step B2. If $(B_j, T_j) < (1, K)$, then the algorithm terminates successfully or unsuccessfully according as $T_j = K$ or not.

Step B3. If $(B_j, T_j) = (1, K)$, then the algorithm terminates successfully.

Step B4. Set $j \leftarrow j - i(K)$. If now $j < 0$, set $j \leftarrow j + M$. Return to step B2. □

Only steps B2 and B3 have changed, and the change is such that the computer time per iteration is the same as before; there is just a little more calculation at the end of a successful search, plus the cost of attaching a 1 at the left of the input argument K when the search begins.

The average number of probes per unsuccessful search with this modified algorithm appears to be difficult to analyse, but the empirical data in **Table 1** shows that the idea can be worthwhile. Of course the number of probes per successful search is unaffected by the extra bits.

So far none of the ideas mentioned have been of any use in the case of successful search. One possibility which suggests itself is to start searching one place ahead (i.e. to start at position $h(K) - i(K)$), because this will save one probe if K is not at its hash address, and because we will be able to test whether K is in position $h(K)$ if the first search is unsuccessful. Since we have greatly improved the ability to detect unsuccessful searches, we can perhaps use some of this capability in connection with successful searches.

Unfortunately, a more careful analysis shows that such an idea is unsound; it actually *increases* the average number of probes for both successful and unsuccessful searching (see Appendix 2). There is, however, a case in which it does work, namely if we force $h(K)$ to be *correlated* with the magnitude of the table entry for K. Suppose we have a hash function such that

Table 1 Average number of probes required by the algorithms, as a function of the load factor $\alpha = N/M$

METHOD	SUCCESSFUL SEARCH							UNSUCCESSFUL SEARCH						
100α (% full)	25	50	75	80	85	90	95	25	50	75	80	85	90	95
Alg. A, linear probing	1·167	1·500	2·500	3·000	3·833	5·500	10·500	1·389	2·500	8·500	13·000	22·722	50·500	200·500
Alg. A, secondary clustering	1·163	1·443	2·011	2·209	2·472	2·853	3·521	1·371	2·193	4·636	5·810	7·715	11·402	22·045
Alg. A, indep. double hashing	1·151	1·386	1·848	2·012	2·232	2·558	3·153	1·333	2·000	4·000	5·000	6·667	10·000	20·000
Alg. B, linear probing	1·167	1·500	2·500	3·000	3·833	5·500	10·500	1·167	1·500	2·500	3·000	3·833	5·500	10·500
Alg. B, secondary clustering	1·163	1·443	2·011	2·209	2·472	2·853	3·521	1·163	1·443	2·011	2·209	2·472	2·853	3·521
Alg. B, indep. double hashing	1·151	1·386	1·848	2·012	2·232	2·558	3·153	1·151	1·386	1·848	2·012	2·232	2·558	3·153
Alg. B, linear, with pass bits	1·167	1·500	2·500	3·000	3·833	5·500	10·500	1·0	1·2	2·0	2·4	3·6	5·4	10·3
Alg. B, indep., with pass bits	1·151	1·386	1·848	2·012	2·232	2·558	3·153	1·0	1·1	1·3	1·4	1·6	1·7	2·2
Alg. B, linear, correlated, one ahead	1·871	1·797	2·245	2·613	3·306	4·825	9·667	2·0	2·2	2·9	3·3	4·0	5·8	11·0
Alg. X, bidirectional linear	1·1	1·3	1·7	2·0	2·3	2·9	4·2	1·3	1·5	2·1	2·3	2·6	3·1	4·4

$$K \le K' \text{ implies } h(K) \le h(K'),$$

and suppose further that we are using linear probing (i.e. that $i(K)$ is identically 1). Then it is not hard to see that the correlation causes the number of probes for successful search in an ordered hash table to have a much smaller variance; there will be fewer keys requiring very small or very large numbers of probes, although the average number will remain unchanged. Appendix 2 shows that this 'start one ahead' approach will lead to less probes per successful search when the table is more than about 64·38% full. (The limiting value $\alpha = 0{\cdot}643797758$, where the one-ahead method begins to excel, is the root of $2(1 - \alpha)(e^{\alpha} - 1) = \alpha$.)

An obvious problem arises, however, if we want the hash function to correlate with the keys in this way. Our options for the choice of hash function will be so drastically reduced that it will probably be impossible to find an efficiently computable $h(K)$ that works well with typical sets of keys. A solution to this dilemma is achieved if we store *transformed keys* in the T table, instead of the keys themselves. Thus, let $t(K)$ be any function which scrambles keys without loss of information:

$$t(K) = t(K') \text{ implies that } K = K'.$$

Then we can store $t(K_1), t(K_2), \ldots$ in the table, and search for $t(K)$ instead of K. We can now achieve the desired correlation between $h(K)$ and $t(K)$ by letting $h(K)$ be the leading bits of $t(K)$.

For example, if M is a prime number and if $h(K) = K \bmod M$, we can let $t(K)$ be a packed binary number whose leftmost bits are $h(K)$ and whose rightmost bits represent the quotient $\lfloor K/M \rfloor$. This transformed key $t(K)$ is one bit larger than the original key. Alternatively if $M = 2^m$ is a power of 2, we may let $t(K) = (aK) \bmod 2^w$, where w is the key length and a is any odd number; then $h(K)$ may be chosen as the leading m bits of $t(K)$.

The reader may justifiably feel at this point that the method is getting 'baroque'. The last few paragraphs have discussed detailed refinements which are mildly interesting, but they can obviously never save more than one probe per search. Therefore the reader may wonder why we are going on and on, 'beating a dead horse'. The answer is that it was precisely the above train of thought, together with hand simulations on random numbers, which led us to consider another algorithm which *does* offer a substantial improvement. We shall now discuss this improved algorithm, which uses the correlation between hash addresses and table entries in a somewhat different fashion.

Bidirectional linear probing

Let $t(K)$ be any one-to-one transformation of keys:

$$t(K) = t(K') \text{ implies } K = K'.$$

Furthermore let $h(K)$ be a hash function such that

$$t(K) \le t(K') \text{ implies } h(K) \le h(K').$$

We have already discussed practical ways of finding such functions; and it is natural to assume that a hash method using such transformations would keep the nonempty positions of the hash table in sorted order:

$$T_i \ne 0 \text{ and } T_j \ne 0 \text{ and } i < j \text{ implies } T_i < T_j.$$

Consider now the following straightforward search procedure:

Algorithm X. (Bidirectional linear probing.)

Step X1. Set $j \leftarrow h(K)$, and set $K \leftarrow t(K)$.

Step X2. If $T_j = K$, the algorithm terminates 'successfully'. If $T_j > K$, go to step X5 (downward search). If $T_j = 0$, the algorithm terminates 'unsuccessfully'. Otherwise go to step X3 (upward search).

Step X3. (At this point, $0 < T_j < K$.) Set $j \leftarrow j + 1$.

Step X4. If $T_j = K$, the algorithm terminates 'scucessfully'. If $T_j = 0$ or $T_j > K$, the algorithm terminates 'unsuccessfully'. Otherwise return to step X3.

Step X5. (At this point, $T_j > K$.) Set $j \leftarrow j - 1$.

Step X6. If $T_j = K$, the algorithm terminates 'successfully'. If $T_j < K$, the algorithm terminates 'unsuccessfully'. Otherwise return to step X5.

This algorithm searches either up or down depending on the result of the first comparison. Its validity depends on having a table T_j whose nonempty entries are ordered as stated above, having the additional property that no empty space occurs between the location of any transformed key and its hash address. Furthermore there must be empty positions at the ends of the table; we can take care of this by extending the boundaries so that $T_{-1} = T_M = 0$.

In this case there are, in general, many configurations of the T's which will guarantee correct retrieval. For example, suppose that $M = 10$ and consider the transformed keys 614, 621, 637, 641, 647, 698, 841, where $h(K)$ is the leading digit. (It is not typical to have so many keys with the same hash address, but our intent is to give a small example which exhibits some of the more interesting things that can happen.) If we use the ordinary method of linear probing (Algorithm B), the table is filled thus:

$j =$	0	1	2	3	4	5	6	7	8	9
$T_j =$	0	614	621	637	641	647	698	0	841	0
probes		6	5	4	3	2	1		1	

The bottom line shows how many table entries are examined when searching for T_j; i.e. it takes four probes to find '637', since we start at T_6. Algorithm X allows us to rearrange the T_j's so that many of the keys will be found sooner:

$j =$	0	1	2	3	4	5	6	7	8	9
$T_j =$	0	0	0	614	621	637	641	647	698	841
probes				4	3	2	1	2	3	2

The search for '841' goes upwards now, but we save two probes when searching for '614'. The average number of probes per successful search is reduced from

$$(6 + 5 + 4 + 3 + 2 + 1 + 1)/7 = 22/7$$

to

$$(4 + 3 + 2 + 1 + 2 + 3 + 2)/7 = 17/7.$$

Appendix 3 shows how to characterise the *optimum* arrangements of the T_j's, for any given set of keys, i.e. those arrangements which minimise the average number of probes per successful search by Algorithm X. As a consequence of the theory developed there, we may use the following algorithm to insert into a bidirectional hash table, maintaining optimum arrangements at all times.

Algorithm Y. (Optimum insertion for bidirectional linear probing.) In this algorithm, let $h'(T_j)$ be $h(t^{-1}(T_j))$; thus if $T_j = t(K_j)$ then $h'(T_j) = h(K_j)$.

Step Y1. Set $j \leftarrow h(K)$, $K \leftarrow t(K)$.

Step Y2. If $T_j = 0$, set $T_j \leftarrow K$ and terminate the algorithm.

Step Y3. Set p to the largest index $<j$ such that $T_p = 0$. Set q to the smallest index $>j$ such that $T_q = 0$.

Step Y4. Set $j \leftarrow q$. Then if $T_{j-1} > K$, repeatedly set $T_j \leftarrow T_{j-1}$ and $j \leftarrow j - 1$, until $T_{j-1} < K$. Finally set $T_j \leftarrow K$. (Thus, K has been sorted into the proper place with respect to the other transformed keys.)

Step Y5. Set $d \leftarrow 0$. Then for $j \leftarrow p + 1, p + 2, \ldots, q$ (in this order), repeatedly set

$$d \leftarrow d + 1 \text{ if } h'(T_j) \ge j,$$
$$d \leftarrow d - 1 \text{ if } h'(T_j) < j.$$

If at any time during this process d becomes negative, go immediately to step Y6 without finishing the loop. But if d

remains ≥ 0 throughout the entire loop, terminate the algorithm.

Step Y6. Set $T_j \leftarrow T_{j+1}$ for $p \leq j < q$, and set $T_q \leftarrow 0$.

Algorithm Y finds the smallest block of consecutive nonempty locations containing position $h(K)$, and inserts $t(K)$ into this block by shifting the transformed keys which are larger. Then step Y5 is used to decide whether or not it would have been better to shift the transformed keys which are smaller; if so, step Y6 moves the whole block down. (Empirical tests show that step Y6 is required only about 1/4 as often as step Y5.)

To use this algorithm, a dozen or so extra table positions T_j should be included for $j < 0$ and for $j \geq M$, to avoid end effects. (There are several ways to make the algorithm cyclically symmetric modulo M, but these are more complicated and time-consuming than simply to provide extra 'breathing space' at both ends. The optimum arrangement rarely spills over very far; in our experiments with $M = 4096$ and tables 95% full, no more than five locations were needed at either end.)

The theory of linear probing shows that this insertion method isn't extremely slow; the average size $q - p$ of the block of keys considered when the $(N + 1)$st key is being inserted will be $2A'_N - 2 \approx (1 - \alpha)^{-2} - 1$ when $N/M = \alpha$. (*Cf.* Knuth, 1973, exercise 6.4–47.) When this size is averaged over N insertions, it reduces to $2A_N - 2 \approx \alpha/(1 - \alpha)$. Thus, insertion by Algorithm Y is only four or five times slower than insertion by the classical linear probing algorithms. On the other hand, empirical results (see Table 1) show that retrieval by Algorithm X is significantly better than classical linear probing.

Conclusions

Traditional hash methods are comparatively slow with respect to unsuccessful search. By extending them to make use of the inherent ordering of keys, we have shown that the time for unsuccessful search can be significantly reduced.

Two main algorithms have been presented in this paper. First we discussed Algorithm B, and the corresponding Algorithm C for insertion. This method reduced the time for unsuccessful search to the time for successful search, without significantly increasing the cost per insertion. Therefore it is attractive for applications in which unsuccessful searches are common. A refinement, adding 'pass bits', makes unsuccessful search even faster. However, the method is never useful in typical compiler or assembler applications, where unsuccessful searches are almost always followed by insertions.

The second method we have discussed is Algorithm X, together with the corresponding Algorithm Y for insertion. Here both successful and unsuccessful search times are reduced, at the expense of greater insertion time and slightly more complex programs. (The method may be compared with a scheme recently published by Brent (1973); his method requires less probes than ours on successful searches, but it does not reduce the unsuccessful search time.)

Table 1 presents the behavioural characteristics of the algorithms discussed here, assuming random hash functions. Some of the results have been derived by theoretical analyses; these are shown to three decimal places. The other results, for which only one decimal place of accuracy appears in the table, have not yet been verified theoretically. Every entry in Table 1 is the number of probes per search, i.e. the number of T_j entries examined. This information can be used to predict the behaviour of each algorithm; but it should be emphasised that the time per probe and the setup time will vary from one method to another. For example, linear probing and Algorithm X will have faster inner loops than independent double hashing, while the latter (especially with 'pass bits') involves fewer probes. Thus the number of probes is not an absolute measure of goodness, the entire algorithm must be considered when making comparisons.

Appendix 1

Analysis of step C3

To analyse the quantity D_N defined in the text, let us assume that the keys are $K_1 < K_2 < \ldots < K_N$. Let D'_N be the average number of times, during N random insertions, that the variable K is set to the *smallest* key K_1 at some time during the insertion process. In other words, D'_N is the average number of times K_1 is 'moved'. Then D'_{N-1} is the average number of times K_2 is moved, since the behaviour of the algorithm on $\{K_2, \ldots, K_N\}$ is essentially independent of K_1. Similarly, D'_{N+1-i} is the average number of times K_i is moved. Therefore

$$D_1 + \ldots + D_N = (D'_1 - 1) + \ldots + (D'_1 - 1)$$

for all N, and we have

$$D_N = D'_N - 1 .$$

Consider now the case of independent double hashing. Experience shows (but it has not been rigorously proved) that this case is satisfactorily approximated by *uniform* hashing, where each key's path is a random permutation of $\{0, 1, \ldots, M - 1\}$, independent of all other keys. Under this assumption, which has been tacitly made in the text, the analysis of hashing algorithms usually becomes quite easy. However, the 'organ pipe' example of the text indicates some of the complexities of Algorithm C, and a rather indirect approach to the analysis of D_N (or D'_N) seems to be necessary.

Let D''_N be the probability that the smallest key K_1 is moved during the insertion of the Nth key. It follows that

$$D'_N = \frac{D''_1 + 2D''_2 + \ldots + ND''_N}{N} ,$$

since the probability that K_1 is moved on the jth insertion is D''_j times j/N, the probability that K_1 appears among the first j keys inserted.

Consider the entire sequence of actions which occur when the keys $\{K_1, \ldots, K_N\}$ are inserted into the table in decreasing order. This sequence of actions states for example that, when K_j was inserted, a certain sequence of larger keys were encountered before an empty place was found. We shall call the elements of the latter sequence the *dominators* of K_j. Knowing all the sequences of dominators, in the decreasing-order case, we can deduce what actions will occur when the keys are inserted in any other specified order. Define the function p on the indices $\{2, \ldots, N\}$ such that, if K_j is the last of $\{K_2, \ldots, K_N\}$ to be inserted, then $K_{p(j)}$ will be the last of $\{K_2, \ldots, K_N\}$ to be moved. Now the very last insertion moves K_1 if and only if either (*a*) K_1 was the last element inserted, or (*b*) K_j was the last element inserted, for some $j \geq 2$, and $K_{p(j)}$ is one of the dominators of K_1.

For example, suppose $N = 3$, so that $K_3 > K_2 > K_1$. If K_3 is a dominator of K_2, we have $p(3) = p(2) = 2$, and K_1 is moved on the third insertion if and only if it is the last to be inserted or it is dominated by K_2. On the other hand if K_3 does not dominate K_2, then $p(3) = 3$ and $p(2) = 2$; hence K_1 is moved on the third insertion if and only if it is either the last to be inserted, or it is dominated by the last to be inserted.

For any fixed choice of dominator sequences on $\{K_2, \ldots, K_N\}$, and for fixed $j \geq 2$, the probability that $K_{p(j)}$ dominates K_1 is a function only of M and N, independent of j and the given actions, because of the assumptions of uniform hashing. This probability may be expressed as

$$\frac{1}{N-1} \sum_{r \geqslant 1} (r - 1) P_r ,$$

where P_r is the probability that K_1 has $r - 1$ dominators, since exactly

$$\binom{N-2}{r-2}\Big/\binom{N-1}{r-1} = (r-1)/(N-1)$$

of the possible choices of $r-1$ dominators include the given key $K_{p(j)}$. Since P_r is also the probability that r probes are needed to insert the Nth item by Algorithm A, we have

$$\frac{1}{N-1}\sum_{r\geqslant 1}(r-1)\,P_r = \frac{1}{N-1}\,(A'_{N-1}-1)\,.$$

The probability that K_j is inserted last is $1/N$; summing for $2 \leq j \leq N$, and adding $1/N$ for the case that K_1 comes last, gives

$$D'_N = \frac{N-1}{N}\left(\frac{1}{N-1}\,(A'_{N-1}-1)\right) + \frac{1}{N} = \frac{1}{N}\,A'_{N-1}\,.$$

The above formulas now yield the desired answer,

$$D_N = A_N - 1\,.$$

Such a simple result deserves a simpler proof; however, it is surprisingly easy to derive this formula by plausible but fallacious arguments, and the above approach is the only reliable one for this analysis that is known to the authors.

We come finally to the case of linear probing. This is much more complicated, and the derivation will only be sketched here. Consider the M^n 'hash sequences' $a_1 \ldots a_n$ to be equally likely, where the kth key inserted has $h(K) = a_k$. Then the probability that the $(n+1)$st key inserted moves K_1 and is not itself K_1 is

$$\frac{1}{N}\sum_{\substack{1\leqslant k\leqslant n\\ 1\leqslant i\leqslant n}} ka(M,n,k,i)M^{-n}\,, \qquad (*)$$

where $a(M,n,k,i)$ denotes the number of hash sequences $a_1, \ldots, a_n$ which cause T_0 through T_{k-1} to be occupied, T_k to be empty, *and* $T_0 = K_1$ if the smallest key K_1 is the ith to be inserted. Let $g(M,n,k)$ be the number of hash sequences which cause T_0 through T_{k-1} to be occupied and $T_{M-1} = T_k = 0$, and let $f(M,n)$ be the number of hash sequences which cause $T_{M-1} = 0$. Then the formulas

$$f(m,n) = (m-n)m^{n-1},$$

$$g(m,n,k) = \binom{n}{k} f(k+1,k)\,f(m-k-1,n-k)$$

can be derived by simple arguments (see Knuth, 1973, p. 529). Also let b_n be the number of hash sequences $a_1 \ldots a_n$ which cause $T_0, \ldots, T_{n-1}$ to be occupied, and for which the 'pass bit' B_j is set to 1 for $0 < j \leq n-1$. From the relation

$$f(n+1,n) = \sum_{0\leqslant k<n}\binom{n}{k} f(k+1,k)b_{n-k}$$

and Abel's binomial formula, we deduce that $b_n = (n-1)^{n-1}$. Now the value of (*) may be expressed as

$$\frac{1}{N}\sum_{1\leqslant j\leqslant n}\binom{n}{j} jb_j \sum_{l\geqslant 0}(j+l)\,g(M-j+1,n-j,l)\,M^{-n} \qquad (**)$$

because we obtain each sequence enumerated by $a(M,n,k,i)$ by piecing together, in $\binom{n}{j}$ ways, a sequence enumerated by b_j and a sequence enumerated by $g(M-j+1,n-j,l)$, where $1 \leq i \leq j$ and $k = j+l$. The sum (**) can be evaluated as described in Knuth (1973), page 691, exercise 27; the result is

$$\frac{1}{N}\left(\frac{n}{M} + 2\,\frac{n(n-1)}{M^2} + 3\,\frac{n(n-1)(n-2)}{M^3} + \ldots\right),$$

essentially an incomplete gamma function. Summing for $0 \leq n < N$, and adding 1 for when K_1 is inserted, yields the desired result

$$D'_N = 1 + \frac{1}{2}\,\frac{N-1}{M} + \frac{2}{3}\,\frac{(N-1)(N-2)}{M^2} + \frac{3}{4}\,\frac{(N-1)(N-2)(N-3)}{M^3} + \ldots\,.$$

Appendix 2

Starting one place ahead

Consider the case of linear probing in an ordered hash table, when $h(K)$ is uncorrelated with the magnitude of K. Let P_r be the probability that exactly r probes are needed to find the $(n+1)$st largest key, for some fixed value of n. Then P_r is the probability that the positions occupied by the n largest keys include $h-1, h-2, \ldots, h-r+1$, but not position $h-r$, given any h; and P_{r+1} is the probability that $h, h-1, \ldots, h-r+1$ (but not $h-r$) are included. Hence $P_r - P_{r+1}$ is the probability that $h-1, \ldots, h-r+1$ are occupied, but neither h nor $h-r$, for any given h. It follows that the expected number of probes needed to locate the $(n+1)$st largest key K, if we begin searching at location $h(K)-1$ instead of $h(K)$, is

$$\sum_{r\geqslant 2}(r-1)\,P_r + \sum_{r\geqslant 1}(r+1)\,(P_r - P_{r+1}) = P_1 + \sum_{r\geqslant 1} r\,P_r\,.$$

This always exceeds $\sum r\,P_r$, which is the corresponding average if we begin searching at $h(K)$.

Essentially the same argument applies to uniform hashing. So we may conclude that it is not a good idea to start probing at location $h(K) - i(K)$.

However, the situation is considerably different when $h(K)$ is correlated with K, so that $h(K) \leq h(K')$ whenever $K < K'$, since then T_{h-1} is almost always less than T_h. To analyse this situation, let us look first at the case that j never goes from 0 to $M-1$ during a successful search. (In other words, the 'pass bit' B_0 is 0.) Then the nonzero T_j's are sorted; hence if we start a search at $h(K)-1$, we will lose only one probe when K is in its 'home position' $h(K)$ while we save one probe whenever K is not. It follows that the one-ahead method is favourable, for successful searching, whenever the number of keys in home position is less than $\frac{1}{2}N$.

An assumption which greatly simplifies the analysis when $B_0 \neq 0$ is to restore cyclic symmetry, by assuming that keys which passed from position 0 to position $M-1$ behave subsequently as if they are larger than keys which haven't. Under this assumption we shall prove below that the average number of keys in home position is exactly

$$(M-N)\left(\left(1+\frac{1}{M}\right)^{N-1} - 1\right) + \left(1+\frac{1}{M}\right)^{N-1}.$$

For $N/M = \alpha$ as $M \to \infty$, this number is approximately

$$(1-\alpha)\,\frac{e^{\alpha}-1}{\alpha}\,N\,;$$

curiously as $N \to M$ it drops to approximately e.

Without the above symmetry assumption, the number of keys in home position might be drastically different. For example, when $M = 10$ and $N = 8$, the hash sequence 9 8 7 6 5 1 1 1 leaves only one element in home position under cyclic symmetry, but there will be six keys in home position in the true ordered hash table. However, the average effect of this correction is bounded by the length of search A'_N for the smallest element, and for $N/M = \alpha < 1$ the correction is asymptotically negligible. Similarly we may ignore the fact that the search for a key in home position T_0 might be adversely affected by the presence of larger keys in T_{M-1}, T_{M-2}, etc.

To prove the formula for keys in home position under cyclic symmetry we can observe that the number of hash sequences $a_1 \ldots a_N$ which leave an element in home position $M-1$ is exactly

$$f(M+1,N) - f(M,N)\,,$$

in the notation of Appendix 1. For if we add the $f(M, N)$ hash sequences which leave T_{M-1} empty, we obtain all the $f(M+1, N)$ hash sequences which would leave T_M empty in a linearly-probed hash table of size $M+1$. Therefore the average total number of elements in home position, under cyclic symmetry, is

$$M(f(M+1, N) - f(M, N))/M^N .$$

The formula for f, given in Appendix 1, completes the proof.

It is interesting to study the cyclically symmetric algorithm further, to find the average number of elements displaced exactly d locations from their home position when $h(K)$ correlates with K. Let $h(M, n, k)$ be the number of hash sequences $a_1 \ldots a_n$ for which $\leq k$ elements pass from position 0 to $M-1$ when they are inserted. Then, by considering the number of such sequences containing exactly j zeroes, we obtain the recurrence

$$h(m, n, k) = \sum_j h(m-1, n-j, k+1-j)$$

for all $m, n, k \geq 0$. Furthermore we have the initial conditions

$$h(m, n, 0) = f(m+1, n) = (m+1-n)(m+1)^{n-1} ,$$

from which it is possible to derive the general formula

$$h(m, n, k) = (m+1+k-n) \times \sum_{0 \leqslant r \leqslant k} \binom{n}{r} (m+k+1-r)^{n-1-r} (r-k-1)^r$$

for all $m, n, k \geq 0$. (Abel's binomial identity shows that this sum equals m^n whenever $k \geq n$.)

The hash sequence $a_1 \ldots a_N$ produces a key with home address 0 and displacement d if and only if it is a sequence with $\geq d$ keys passing from 0 to $M-1$ but $\leq d$ passing from 1 to 0. The number of such hash sequences, when $d > 0$, is

$$h(M, N, d) - h(M, N, d-1) ,$$

because $h(M, N, d)$ is the number of hash sequences with $\leq d$ keys passing from 1 to 0, while $h(M, N, d-1)$ is the number with $< d$ passing from 0 to $M-1$ *and* (consequently) $\leq d$ from 1 to 0. It follows that the average total number of keys with displacement $d > 0$ is

$$M(h(M, N, d) - h(M, N, d-1))/M^N .$$

It would be interesting to obtain asymptotic data about this probability distribution. When $M = N$, the same formulas arise in connection with the classical Kolmogorov–Smirnov tests for random numbers: the quantity $h(n, n, k-1)/n^n$ is the probability that the so-called statistic K_n^+ is $\leq k/\sqrt{n}$. According to a theorem of N. V. Smirnov in 1939, we have

$$\lim_{n \to \infty} \frac{h(n, n, s\sqrt{n})}{n^n} = 1 - e^{-2s^2} ;$$

cf. Knuth (1969), p. 51.

Appendix 3

Optimum bidirectional linear probing

Given N keys $K_1 < \ldots < K_N$ and corresponding hash addresses $0 \leq h(K_1) \leq \ldots \leq h(K_N) < M$, we wish to place them into table positions so that K_j appears in $T_{p(j)}$ for $1 \leq j \leq N$, where $p(1) < \ldots < p(N)$. Writing $h_j = h(K_j)$, we wish to find a placement which is *optimum*, in the sense that the sum

$$\sum_{1 \leqslant j \leqslant N} |h_j - p(j)|$$

is minimised. We shall call this sum the 'cost' of the placement. For convenience in exposition, we shall allow the positions $p(j)$ to be negative or greater than M, although the proofs could easily be extended to characterise the optimum arrangements subject to $p(1) \geq x$ and $p(N) \leq y$, for any desired bounds $x \leq 0$ and $y \geq M-1$. Algorithm X requires a placement such that all positions between h_j and $p(j)$ are occupied, for each j; however, we may ignore this condition, because all optimum placements automatically satisfy it.

Given any placement $p(1) < \ldots < p(N)$, we shall say that a *block* $[a, b]$ is a set of consecutive positions which are occupied by K_a through K_b (i.e. $p(j+1) = p(j) + 1$ for $a \leq j < b$). An *up-block* is a block followed by an empty position, which would lead to less cost if it were shifted one place higher; in other words, it is a block $[a, b]$ such that the shifted placement p' has less cost, where

$$p'(j) = \begin{cases} p(j) + 1 \text{ if } a \leq j \leq b; \\ p(j), \text{ otherwise.} \end{cases}$$

By the definition of cost, we find that $[a, b]$ is an up-block if and only if

(*a*) either $b = N$ or $p(b+1) > p(b) + 1$; and
(*b*) the number of j in the range $a \leq j \leq b$, for which $h_j > p(j)$, exceeds the number for which $h_j \leq p(j)$.

Thus it is easy to test a given placement for the presence of up-blocks. A *down-block* is defined similarly.

An optimum placement will, of course, contain neither up-blocks nor down-blocks. Conversely, this condition of local optimality is sufficient for global optimality:

Theorem:
Given N hash addresses $h_1 \leq \ldots \leq h_N$, a placement $p(1) < \ldots < p(N)$ is optimum if and only if it contains no up-blocks and no down-blocks.

Proof:
Let p be an arbitrary placement; we want to prove that p either contains an up-block, or a down-block, or is optimum. Let p' be an optimum placement. If $p(j) = p'(j)$ for all j, we are done. Otherwise suppose that $p(j_0) \neq p'(j_0)$; by symmetry we may assume that $p(j_0) p' < (j_0)$.

It would be nice if we could prove that $p(j_0)$ is part of an up-block, under these hypotheses. However, the following example shows that the argument cannot be quite so trivial:

$k =$	1	2	3	4	5	6	7	8
$h_k =$	4	4	4	4	4	6	6	6
$p(k) =$	0	1	2	3	4	6	7	8
$p'(k) =$	2	3	4	5	6	7	8	9

If $j_0 = 6$, we have $p(j_0) < p'(j_0)$, but $p(j_0)$ is actually part of a *down-block*.

We can circumvent such difficulties by arguing as follows. Let a' be minimal so that $[a', j_0]$ is a block in placement p'. Then let b be maximal so that $[a', b]$ is a block in placement p. Then let a be minimal so that $[a, b]$ is a block in placement p'. (In the above example, when $j_0 = 6$, we will have $a' = 1$ and $b = 5$, and $a = 1$.) In general we will always have $p(a') < p'(a')$, $p(b) < p'(b)$, and $[a, b]$ will always be a block in both placements; thus, $p'(j) - p(j)$ has a constant value $t \geq 1$ for $a \leq j \leq b$. Furthermore, position $p(b) + 1$ is empty in placement p, while $p'(a) - 1$ is empty in placement p'.

Let d_+ be the number of displacements $h_j - p(j)$ in block $[a, b]$ that are positive for placement p; also let d_- be the number of negative displacements in the block, and let d_k be the number of displacements which equal k. Define d'_+, d'_-, and d'_k similarly for placement p'. It follows that $d_k = d'_{k-t}$ for all k.

Now $[a, b]$ is an up-block for p if and only if $d_+ > d_0 + d_-$, and it is a down-block for p' if and only if $d'_- > d'_0 + d'_+$. Our proof would be complete if $[a, b]$ were an up-block for p, hence we may assume that

$$d_+ \leq d_0 + d_- .$$

The optimality of p' implies that

$$d'_- \leq d'_0 + d'_+ .$$

Now the latter inequality is equivalent to

$$d_- + d_0 + d_1 + \ldots + d_{t-1} \leq d_+ - (d_1 + \ldots + d_{t-1}),$$

hence we have

$$d_+ \leq d_0 + d_- \leq d_+ - 2(d_1 + \ldots + d_{t-1}).$$

This can be true only if $d_+ = d_0 + d_-$ and $d'_- = d'_0 + d'_+$. If we shift block $[a, b]$ one position down from where it was in p', we obtain a new placement p'' of cost equal to p', hence p'' is optimum. Furthermore p'' is *closer* to the given placement, in an obvious sense, so the proof will eventually terminate.

It is interesting to note that the above proof does not use the hypothesis $h_1 \leq \ldots \leq h_N$; it characterises the optimum placements for arbitrary (even non-integral) h_j. If $N = 2$, with $h_1 = 100$ and $h_2 = 1$, there are actually one hundred optimum placements, namely $p_k(j) = k + j$ for $-1 \leq k \leq 99$. The additional hypothesis $h_1 \leq \ldots \leq h_N$ leads to a slightly stronger theorem, showing that the optimum placements are more constrained: When $h_j \leq h_{j+1}$, we have

$$h_j - p(j) \leq h_{j+1} - p(j) = h_{j+1} - p(j+1) + 1,$$

for j and $j + 1$ in the same block. The above proof can now be strengthened to show that $d_1 > 0$ (and hence $t = 1$) whenever p has no up-blocks. Thus, two optimum placements p and p' must have $|p(j) - p'(j)| \leq 1$ for all j, whenever the h_j form a nondecreasing sequence.

References

BRENT, R. P. (1973). 'Reducing the retrieval time of scatter storage techniques,' *CACM*, Vol. 16, No. 2, February 1973.

KNUTH, D. E. (1969). *Seminumerical Algorithms*; The Art of Computer Programming, Vol. 2, Addison-Wesley Publishing Company.

KNUTH, D. E. (1973). *Sorting and Searching*; The Art of Computer Programming, Vol. 3, Addison-Wesley Publishing Company.

Book reviews

The Management of Problem-Solving, Positive Results from Productive Thinking, by G. Tarr, 1973; 160 pages. (*Macmillan*, £3·95)

Management today is either self-consciously scientific or nervously aware that it ought to be: it is inclined to snatch at instant techniques ready-made for the amateur—the unqualified in pursuit of the unquantifiable. Or, it may resort to experts, but too rarely enquires whether the consultancy is itself well managed, or from what source it draws the experience it proposes to dispense. Of course, rather than expecting to have to take the advice the client may be buying an alibi, so that he can support his claim to have left unturned no stone under which efficiency might have lurked. Their clients as well as those who manage the management-problem solvers would do well to read Graham Tarr's little book. His presentation is easy but not condescending, and his prose very readable.

The book 'is intended to be wise rather than learned'; these are the author's own words and ever so slightly off-putting, but he makes good his claim, for while accepting that it is not possible to teach experience he has most helpfully distilled his own. But, what are the signs of wisdom? First, there is the author's practical scepticism about techniques-mongering, then his advocacy of insight rather than processing, his emphasis on the need to quit the trees to see the wood, the value he attaches to common sense and practical men as touchstones for the reputed jewels discovered by analysis, and his reminder of the supreme danger of believing that the solution found is the one best, instead of, at best, one of the best. Again, there is more than a whiff of experience about his comments on managing a project team, on conducting a choir of soloists, on the earnest technical dogmatism that blinkers the young analyst, on the necessity of measuring progress but the impossibility of measuring output; above all, on the need for project leadership and its importance for the quality as well as the quantity of the team's output.

But, this book is not all broad philosophical generalisations, it contains many shrewd comments on the day-to-day conduct of work and the crises of life in a group of problem solvers. It would be a particularly good present for a young man or woman recently, and rather too rapidly, promoted to team leader. Leadership cannot be learned from textbooks, it has to be caught rather than taught, but this book is no textbook, it is a sharing of experience and as such well worth reading and discussing over a pipe or a pint.

F. J. M. LAVER (Sidmouth)

Information, Computers, Machines and Man, edited by A. E. Karbowiak and R. M. Huey, 1971; 347 pages. (*John Wiley and Sons*, £4·50)

This book consists of a series of papers written by the staff of the University of New South Wales with the object of illustrating the methods and concepts that can be applied in the field of systems engineering. It is based upon courses run for 1st and 2nd year students of Applied Science and Engineering, but has been extensively broadened and modified to make it pertinent for those with a more general interest.

The book is structured into three sections which may be referenced independently. Initially the elements and concepts of systems engineering design are covered, with some useful definitions and explanations of the logic and mathematics involved. References are made to analogous biological systems, and there is a valuable chapter on Human Systems, outlining some of the problems in the control of multi-discipline teams and the interaction between professional non-technical management and specialist teams.

The second section covers the more scientific and technical aspects of engineering systems, with detailed accounts of some of the more important materials and components used in control systems. It also includes two chapters on computer architecture and programming which—although unduly IBM 360 oriented—give a useful summary of these topics to the professional engineer who wishes to use or even program a computer, but without the obligation of becoming a computer man.

The final section consists of a discussion of actual systems, including biological, and endeavours to draw together the concepts and material technology developments previously covered. It outlines the likely growth in the application of technology to our way of life, and emphasises the importance of proper planning for change, insisting upon the overwhelming need to consider human factors when inducing such change.

Information, Computers, Machines and Man, despite its over-comprehensive and somewhat pretentious title, is in fact a useful introductory text on systems theory, enhanced by comprehensive examples and an explanation of the fundamentals of systems engineering. It does however suffer from a lack of co-ordination and relevance in many places. This might not be important when it is placed within the wider environment of a degree course, but will be troublesome to the general reader who might justifiably think, from both the fly-leaf and the introduction, that he has acquired a consistent study text on systems engineering. Nevertheless the approach taken is sufficiently broad that both computer professionals and engineers will find it helpful for the analogies drawn, the future developments suggested, and the human and organisational problems outlined.

J. WOODS (London)

Short notice

Computers and Society, by A. S. Douglas.

This is the text of Professor Douglas's inaugural lecture to the London School of Economics and Political Science, on his appointment to the Chair of Computational Method at the LSE.

Randomized Search Trees

Conference Paper · November 1989

DOI: 10.1109/SFCS.1989.63531 · Source: DBLP

2 authors, including:

Cecilia R. Aragon
University of Washington Seattle
211 PUBLICATIONS **4,441** CITATIONS

Randomized Search Trees

Raimund Seidel*

Computer Science Division
University of California Berkeley
Berkeley CA 94720

Fachberich Informatik
Universität des Saarlandes
D-66041 Saarbrücken, GERMANY

Cecilia R. Aragon†

Computer Science Division
University of California Berkeley
Berkeley CA 94720

Abstract

We present a randomized strategy for maintaining balance in dynamically changing search trees that has optimal *expected* behavior. In particular, in the expected case a search or an update takes logarithmic time, with the update requiring fewer than two rotations. Moreover, the update time remains logarithmic, even if the cost of a rotation is taken to be proportional to the size of the rotated subtree. Finger searches and splits and joins can be performed in optimal expected time also. We show that these results continue to hold even if very little true randomness is available, i.e. if only a logarithmic number of truely random bits are available. Our approach generalizes naturally to weighted trees, where the expected time bounds for accesses and updates again match the worst case time bounds of the best deterministic methods.

We also discuss ways of implementing our randomized strategy so that no explicit balance information is maintained. Our balancing strategy and our algorithms are exceedingly simple and should be fast in practice.

This paper is dedicated to the memory of *Gene Lawler*.

1 Introduction

Storing sets of items so as to allow for fast access to an item given its key is a ubiquitous problem in computer science. When the keys are drawn from a large totally ordered set the method of choice for storing the items is usually some sort of search tree. The simplest form of such a tree is a binary search tree. Here a set X of n items is stored at the nodes of a rooted binary tree as follows: some item $y \in X$ is chosen to be stored at the root of the tree, and the left and right children of the root are binary search trees for the sets $X_< = \{x \in X \mid x.key < y.key\}$ and

*Supported by NSF Presidential Young Investigator award CCR-9058440. Email: seidel@cs.uni-sb.de
†Supported by an AT&T graduate fellowship

$X_> = \{x \in X \mid y.key > x.key\}$, respectively. The time necessary to access some item in such a tree is then essentially determined by the depth of the node at which the item is stored. Thus it is desirable that all nodes in the tree have small depth. This can easily be achieved if the set X is known in advance and the search tree can be constructed off-line. One only needs to "balance" the tree by enforcing that $X_<$ and $X_>$ differ in size by at most one. This ensures that no node has depth exceeding $\log_2(n+1)$.

When the set of items changes with time and items can be inserted and deleted unpredictably, ensuring small depth of all the nodes in the changing search tree is less straightforward. Nonetheless, a fair number of strategies have been developed for maintaining approximate balance in such changing search trees. Examples are AVL-trees [1], (a,b)-trees [4], $BB(\alpha)$-trees [25], red-black trees [13], and many others. All these classes of trees guarantee that accesses and updates can be performed in $O(\log n)$ worst case time. Some sort of balance information stored with the nodes is used for the restructuring during updates. All these trees can be implemented so that the restructuring can be done via small local changes known as "rotations" (see Fig. 1). Moreover, with the appropriate choice of parameters (a,b)-trees and $BB(\alpha)$-trees guarantee that the average number of rotations per update is constant, where the average is taken over a sequence of m updates. It can even be shown that "most" rotations occur "close" to the leaves; roughly speaking, for $BB(\alpha)$-trees this means that the number of times that some subtree of size s is rotated is $O(m/s)$ (see [17]). This fact is important for the parallel use of these search trees, and also for applications in computational geometry where the nodes of a primary tree have secondary search structures associated with them that have to be completely recomputed upon rotation in the primary tree (e.g. range trees and segment trees; see [18]).

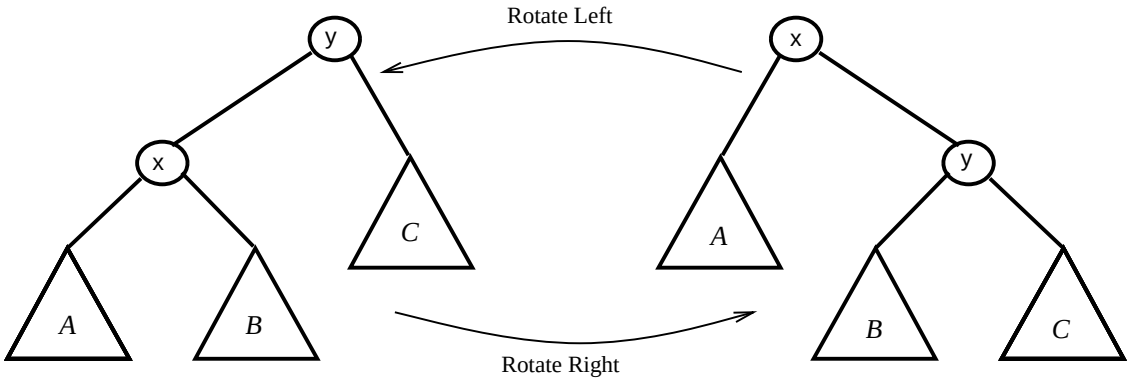


Figure 1:

Sometimes it is desirable that some items can be accessed more easily than others. For instance, if the access frequencies for the different items are known in advance, then these items should be stored in a search tree so that items with high access frequency are close to the root. For the static case an "optimal" tree of this kind can be constructed off-line by a dynamic programming technique. For the dynamic case strategies are known, such as biased 2-3 trees [5] and D-trees [17], that allow accessing an item of "weight" w in worst case time $O(\log(W/w))$, which is basically optimal. (Here W is the sum of the weights of all the items in the tree.) Updates can be performed in time $O(\log(W/\min\{w-, w, w+\})$, where $w-$ and $w+$ are the weights of the items that precede and succeed the inserted/deleted item (whose weight is w).

All the strategies discussed so far involve reasonably complex restructuring algorithms that require some balance information to be stored with the tree nodes. However, Brown [8] has pointed out that some of the unweighted schemes can be implemented without storing any balance infor-

mation explicitly. This is best illustrated with schemes such as AVL-trees or red-black trees, which require only one bit to be stored with every node: this bit can be implicitly encoded by the order in which the two children pointers are stored. Since the identities of the children can be recovered from their keys in constant time, this leads to only constant overhead to the search and update times, which thus remain logarithmic.

There are methods that require absolutely no balance information to be maintained. A partiucarly attractive one was proposed by Sleator and Tarjan [30]. Their "splay trees" use an extremely simple restructuring strategy and still achieve all the access and update time bounds mentioned above both for the unweighted and for the weighted case (where the weights do not even need to be known to the algorithm). However, the time bounds are not to be taken as worst case bounds for individual operations, but as *amortized* bounds, i.e. bounds averaged over a (sufficiently long) sequence of operations. Since in many applications one performs long sequences of access and update operations, such amortized bounds are often satisfactory.

In spite of their elegant simplicity and their frugality in the use of storage space, splay trees do have some drawbacks. In particular, they require a substantial amount of restructuring not only during updates, but also during accesses. This makes them unusable for structures such as range trees and segment trees in which rotations are expensive. Moreover, this is undesirable in a caching or paging environment where the writes involved in the restructuring will dirty memory locations or pages that might otherwise stay clean.

Recently Galperin and Rivest [12] proposed a new scheme called "scapegoat trees," which also needs basically no balance information at all and achieves logarithmic search time even in the worst case. However logarithmic update time is achieved only in the amortized sense. Scapegoat trees also do not seem to lend themselves to applications such as range trees or segment trees.

In this paper we present a strategy for balancing unweighted or weighted search trees that is based on randomization. We achieve *expected case* bounds that are comparable to the deterministic worst case or amortized bounds mentioned above. Here the expectation is taken over all possible sequences of "coin flips" in the update algorithms. Thus our bounds do not rely on any assumptions about the input. Our strategy and algorithms are exceedingly simple and should be fast in practice. For unweighted trees our strategy can be implemented without storage space for balance information.

Randomized search trees are not the only randomized data structure for storing dynamic ordered sets. Bill Pugh [26] has proposed and popularized another randomized scheme called *skip lists.* Although the two schemes are quite different they have almost identical expected performace characteristics. We offer a brief comparison in the last section.

Section 2 of the paper describes *treaps*, the basic structure underlying randomized search trees. In section 3 unweighted and weighted randomized search trees are defined and all our main results about them are tabulated. Section 4 contains the analysis of various expected quantities in randomized search trees, such as expected depth of a node or expected subtree size. These results are then used in section 5, where the various operations on randomized search trees are described and their running times are analyzed. Section 6 discusses how randomized search trees can be implemented using only very few truly random bits. In section 7 we show how one can implement randomized search trees without maintaining explicit balance information. In section 8 we offer a short comparison of randomized search trees and skip lists.

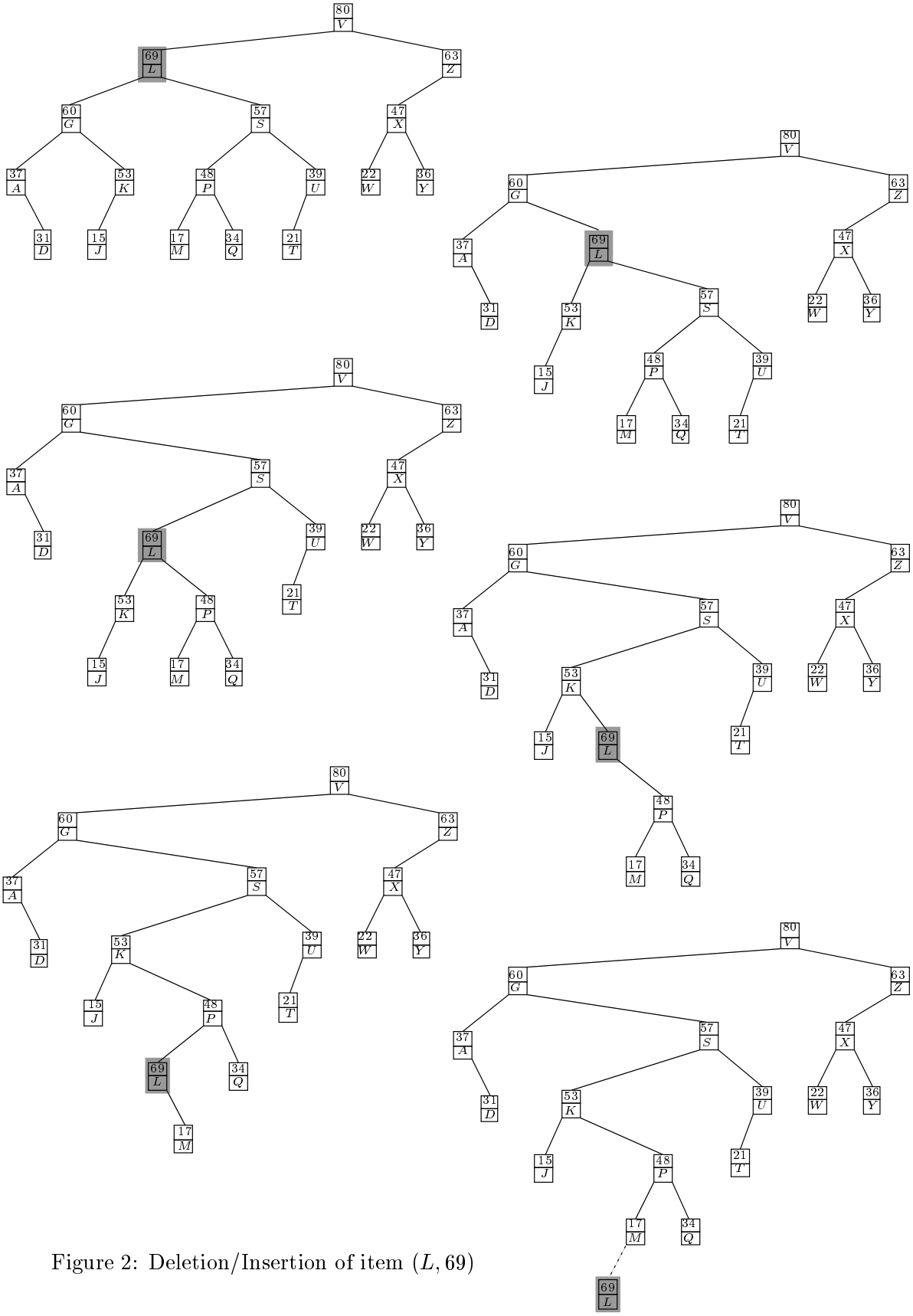


Figure 2: Deletion/Insertion of item $(L, 69)$

2 Treaps

Let X be a set of n items each of which has associated with it a *key* and a *priority*. The keys are drawn from some totally ordered universe, and so are the priorities. The two ordered universes need not be the same. A *treap* for X is a rooted binary tree with node set X that is arranged in in-order with respect to the keys and in heap-order with respect to the priorities.[1] "In-order" means that for any node x in the tree $y.key \leq x.key$ for all y in the left subtree of x and $x.key \leq y.key$ for y in the right subtree of x. "Heap-order" means that for any node x with parent z the relation $x.priority \leq z.priority$ holds. It is easy to see that for any set X such a treap exists. With the assumption that all the priorities and all the keys of the items in X are distinct — a reasonable assumption for the purposes of this paper — the treap for X is unique: the item with largest priority becomes the root, and the allotment of the remaining items to the left and right subtree is then determined by their keys. Put differently, the treap for an item set X is exactly the binary search tree that results from successively inserting the items of X in order of decreasing priority into an initially empty tree using the usual leaf insertion algorithm for binary search trees.

Let T be the treap storing set X. Given the key of some item $x \in X$ the item can easily be located in T via the usual search tree algorithm. The time necessary to perform this access will be proportional to the depth of x in the tree T. How about updates? The insertion of a new item z into T can be achieved as follows: At first, using the key of z, attach z to T in the appropriate leaf position. At this point the keys of all the nodes in the modified tree are in in-order. However, the heap-order condition might not be satisfied, i.e. z's parent might have a smaller priority than z. To reestablish heap-order simply rotate z up as long as it has a parent with smaller priority (or until it becomes the root). Deletion of an item x from T can be achieved by "inverting" the insertion operation: First locate x, then rotate it down until it becomes a leaf (where the decision to rotate left or right is dictated by the relative order of the priorites of the children of x), and finally clip away the leaf (see Figure 2).

At times it is desirable to be able to *split* a set X of items into the set $X_1 = \{x \in X \mid x.key < a\}$ and the set $X_2 = \{x \in X \mid x.key > a\}$, where a is some given element of the key universe. Conversely, one might want to *join* two sets X_1 and X_2 into one, where it is assumed that the keys of the items in X_1 are smaller than the keys from X_2. With treap representations of the sets these operations can be performed easily via the insertion and deletion operations. In order to *split* a treap storing X according to some a, simply insert an item with key a and "infinite" priority. By the heap-order property the newly inserted item will be at the root of the new treap. By the in-order property the left subtree of the root will be a treap for X_1 and the right subtree will be a treap for X_2. In order to *join* the treaps of two sets X_1 and X_2 as above, simply create a dummy root whose left subtree is a treap for X_1 and whose right subtree is a treap for X_2, and perform a delete operation on the dummy root.

Recursive pseudocode implementations[2] of these elementary treap update algorithms are shown in Figure 3.

Sometimes "handles" or "fingers" are available that point to certain nodes in a treap. Such handles permit accelerated operations on treaps. For instance, if a handle on a node x is available, then deleting x reduces just to rotating it down into leaf position and clipping it; no search is

[1] Herbert Edelsbrunner pointed out to us that Jean Vuillemin introduced the same data structure in 1980 and called it "Cartesian tree" [V]. The term "treap" was first used for a different data structure by Ed McCreight, who later abandoned it in favor of the more mundane "priority search tree" [Mc].

[2] In practice it will be preferable to approach these operations the other way round. Joins and splits of treaps can be implemented as iterative top-down procedures; insertions and deletions can then be implemented as accesses followed by splits or joins. These implementations are operationally equivalent to the ones given here.

```
function EMPTY-TREAP() : treap
    tnull→[priority,lchild,rchild] ← [−∞,tnull,tnull]
    return( tnull )


procedure TREAP-INSERT( (k,p) : item, T : treap )
    if T = tnull then T ← NEWNODE()
                      T→[key,priority,lchild,rchild] ← [k,p,tnull,tnull]
    else if k < T→key then TREAP-INSERT( (k,p) ,T→lchild )
                           if T→lchild →priority > T→priority then ROTATE-RIGHT( T )
    else if k > T→key then TREAP-INSERT( (k,p) ,T→rchild )
                           if T→rchild →priority > T→priority then ROTATE-LEFT( T )
    else (* key k already in treap T *)


procedure TREAP-DELETE ( k : key, T : treap )
    tnull→key ← k
    REC-TREAP-DELETE( k,T )
procedure REC-TREAP-DELETE ( k : key, T : treap )
    if k < T→key then REC-TREAP-DELETE( k,T→lchild )
    else if k > T→key then REC-TREAP-DELETE( k,T→rchild )
    else ROOT-DELETE( T )
procedure ROOT-DELETE( T : treap )
    if IS-LEAF-OR-NULL( T ) then T ← tnull
    else if T→lchild→priority > T→rchild→priority then ROTATE-RIGHT( T )
                                                       ROOT-DELETE( T→rchild )
                                                  else ROTATE-LEFT( T )
                                                       ROOT-DELETE( T→lchild )


procedure TREAP-SPLIT( T : treap, k : key, T1, T2 : treap )
    TREAP-INSERT( (k,∞), T )
    [T1,T2] ← T→[lchild,rchild]

procedure TREAP-JOIN( T1,T2, T : treap )
    T ← NEWNODE()
    T→[lchild,rchild] ← [T1,T2]
    ROOT-DELETE( T )


procedure ROTATE-LEFT( T : treap )
    [T,T→rchild,T→rchild→lchild] ← [T→rchild,T→rchild→lchild,T]
procedure ROTATE-RIGHT( T : treap )
    [T,T→lchild,T→lchild→rchild] ← [T→lchild,T→lchild→rchild,T]
function IS-LEAF-OR-NULL( T : treap ) : Boolean
    return( T→lchild = T→rchild )
```

Figure 3: Simple routines for the elementary treap operations of creation, insertion, deletion, splitting, and joining. We assume call-by-reference semantics. A treap node has fields *key, priority, lchild, rchild.* The global variable *tnull* points to a sentinel node whose existence is assumed. $[\ldots] \leftarrow [\ldots]$ denotes parallel assignment.

necessary. Similarly the insertion of an item x into a treap can be accelerated if a handle to the successor (or predecessor) s of x in the resulting treap is known: start the search for the correct leaf position of x at node s instead of at the root of the treap. So-called "finger searches" are also possible where one is to locate a node y in a treap but the search starts at some (hopefully "nearby") node x that has a handle pointing to it; essentially one only needs to traverse the unique path between x and y. Also, splits and joins of treaps can be performed faster if handles to the minimum and maximum key items in the treaps are available. These operations are discussed in detail in sections 5.7 and 5.8.

Some applications such as so-called Jordan sorting [15] require the efficient *excision* of a subsequence, i.e. splitting a set of X of items into $Y = \{x \in X \mid a \leq x.key \leq b\}$ and $Z = \{x \in X \mid x.key < a \text{ or } x.key > b\}$. Such an excision can of couse be achieved via splits and joins. However treaps also permit a faster implementation of excisions, which is discussed in section 5.9.

3 Randomized Search Trees

Let X be a set of items, each of which is uniquely identified by a key that is drawn from a totally ordered set. We define a *randomized search tree* for X to be a treap for X where the priorities of the items are independent, identically distributed continuous random variables.

Theorem 3.1 *A randomized search tree storing n items has the expected performance characteristics listed in the table below:*

Performance measure	**Bound on expectation**
access time	$O(\log n)$
insertion time	$O(\log n)$
*insertion time for element with handle on predecessor or successor	$O(1)$
deletion time	$O(\log n)$
*deletion time for element with handle	$O(1)$
number of rotations per update	2
†time for finger search over distance d	$O(\log d)$
†time for fast finger search over distance d	$O(\log \min\{d, n-d\})$
time for joining two trees of size m and n	$O(\log \max\{m, n\})$
time for splitting a tree into trees of size m and n	$O(\log \max\{m, n\})$
‡time for fast joining or fast splitting	$O(\log \min\{m, n\})$
†time for excising a tree of size d	$O(\log \min\{d, n-d\})$
update time if cost of rotating a subtree of size s is $O(s)$	$O(\log n)$
update time if cost of rotating a subtree of size s is $O(s \log^k s)$, $k \geq 0$	$O(\log^{k+1} n)$
update time if cost of rotating a subtree of size s is $O(s^a)$ with $a > 1$	$O(n^{a-1})$
update time if rotation cost is $f(s)$, with $f(s)$ non-negative	$O\left(\frac{f(n)}{n} + \sum_{0<i<n} \frac{f(i)}{i^2}\right)$

*requires parent pointers
†requires parent pointers and some additional information, see section 5.4
‡requires some parent pointers, see section 5.7 and 5.8

Now let X be a set of items identifiable by keys drawn from a totally ordered universe. Moreover, assume that every item $x \in X$ has associated with it an integer weight $w(x) > 0$. We define the *weighted randomized search tree* for X as a treap for X where the priorities are independent

continuous random variables defined as follows: Let F be a fixed continuous probability distribution. The priority of element x is the maximum of $w(x)$ independent random variables, each with distribution F.

Theorem 3.2 *Let T be a weighted randomized search tree for a set X of weighted items. The following table lists the expected performance characteristics of T. Here W denotes the sum of the weights of the items in X; for an item x, the predecessor and the successor (by key rank) in X are denoted by x^- and x^+, respectively; Tmin and Tmax denote the items of minimal and maximal key rank in X.*

Performance measure	**Bound on expectation**
time to acces item x	$O(1+\log\frac{W}{w(x)})$
time to insert item x	$O\left(1+\log\frac{W+w(x)}{\min\{w(x^-),w(x),w(x^+)\}}\right)$
time to delete item x	$O\left(1+\log\frac{W}{\min\{w(x^-),w(x),w(x^+)\}}\right)$
*insertion time for item x with handle on predecessor	$O\left(1+\log(1+\frac{w(x)}{w(x^-)}+\frac{w(x)}{w(x^+)}+\frac{w(x^-)}{w(x^+)})\right)$
*time to delete item x with handle	$O\left(1+\log(1+\frac{w(x)}{w(x^-)}+\frac{w(x)}{w(x^+)})\right)$
number of rotations per update on item x	$O\left(1+\log(1+\frac{w(x)}{w(x^-)}+\frac{w(x)}{w(x^+)})\right)$
†time for finger search between x and y, where V is total weight of items between and including x and y	$O\left(\log\frac{V}{\min\{w(x),w(y)\}}\right)$
time to join trees T_1 and T_2 of weight W_1 and W_2	$O\left(1+\log\frac{W_1}{w(T_1max)}+\log\frac{W_2}{w(T_2min)}\right)$
time to split T into trees T_1, T_2 of weight W_1, W_2	
‡time for fast joining or fast splitting	$O\left(1+\log\min\{\frac{W_1}{w(T_1max)},\frac{W_2}{w(T_2min)}\}\right)$
*time for increasing the weight of item x by Δ	$O\left(1+\log\frac{w(x)+\Delta}{w(x)}\right)$
*time for decreasing the weight of itme x by Δ	$O\left(1+\log\frac{w(x)}{w(x)-\Delta}\right)$

*requires parent pointers
†requires parent pointers and some additional information, see section 5.4
‡requires some parent pointers, see section 5.7 and 5.8

Several remarks are in order. First of all, note that no assumptions are made about the key distribution in X. All expectations are with respect to randomness that is "controlled" by the update algorithms. For the time being we assume that the priorities are kept hidden from the "user." This is necessary to ensure that a randomized search tree is transformed into a randomized search tree by any update. If the user knew the actual priorities it would a simple matter to create a very "non-random" and unbalanced tree by a polynomial number of updates.

The requirement that the random variables used as priorities be continuous is not really necessary. We make this requirement only to ensure that with probability 1 all priorities are distinct. Our results continue to hold for i.i.d. random variables for which the probability that too many of them are equal is sufficiently small. This means that in practice using integer random numbers drawn uniformly from a sufficiently large range (such as 0 to 2^{31}) will be adequate for most applications.

Comparing with other balanced tree schemes some of which require only one bit of balance information per node, it might seem disappointing that randomized search trees require to store such "extensive" balance information at each node. However, it is possible to avoid this. In

section 7 we discuss various strategies of maintaining randomized search trees that obviate the explicit storage of the random priorities.

Next note that weighted randomized search trees can be made to adapt naturally to observed access frequencies. Consider the following strategy: whenever an item x is accessed a new random number r is generated (according to distribution F); if r is bigger than the current priority of x, then make r the new priority of x, and, if necessary, rotate x up in the tree to reestablish the heap-property. After x has been accessed k times its priority will be the maximum of k i.i.d. random variables. Thus the expected depth of x in the tree will be $O(\log(1/p))$, where p is the access frequency of x, i.e. $p = k/A$, with A being the total number of accesses to the tree.

How would one insert an item x into a weighted randomized search tree with fixed weight k? This can most easily be done if the distribution function F is the identity, i.e. we start with random variables uniformly distributed in the interval $[0, 1]$. The distribution function G_k for the maximum of k such random variables has the form $G_k(z) = z^k$. From this it follows that x should be inserted into the tree with priority $r^{1/k}$, where r is a random number chosen uniformly from the interval $[0, 1]$. Since the only operations involving priorities are comparisons and the logarithm function is monotonic, one can also store $(\log r)/k$ instead. Adapting the tree to changing weights is also possible, see section 5.11 .

Finally there is the question of how much "randomness" is required for our method. How many random bits does one need to implement randomized search trees? There is a rather simple minded strategy that shows that an expected *constant* number of random bits per update can suffice in the unweighted case. This can be achieved as follows: Let the priorities be real random numbers drawn uniformly from the interval $[0, 1]$. Such numbers can be generated piece-meal by adding more and more random bits as digits to their binary representations. The idea is, of course, to generate only as much of the binary representation as needed. The only priority operation in the update algorithms are comparisons between priorities. In many cases the outcome of such a comparison will already be determined by the existing partial binary representations. When this is not the case, i.e. one representation happens to be a prefix of the other, one simply refines the representations by appending random bits in the obvious way. It is easy to see that the expected number of additional random bits needed to resolve the comparison is not greater than 4. Since in our update algorithms priority comparisons happen only in connection with rotations, and since the expected number of rotations per update is less than 2, it follows that the expected number of random bits needed is less than 12 for insertions and less than 8 for deletions.

Surprisingly, one can do much better than that. Only $O(\log n)$ truly random bits suffice overall. This can be achieved by letting a pseudo random number generator supply the priorities, where this generator needs $O(\log n)$ truly random seed bits. One possible such generator works as follows. Let $U \geq n^3$ be a prime number. Randomly choose 8 integers in the range between 0 and $U - 1$ and make them the coefficients of a degree-7 polynomial q (this requires the $O(\log n)$ random bits). As i-th priority produce $q(i) \bmod U$. Other generators are possible. The essential property they need to have is that they produce "random" numbers that are 8-wise independent. (Actually a somewhat weaker property suffices; see section 6.)

Theorem 3.3 *All the results of Theorem 3.1 continue to hold, up to constant factors, if the priorities are 8-wise independent random variables from a sufficiently large range, as for instance produced by the pseudo random number generator outlined above.*

In order to achieve logarithmic expected access, insertion, and deletion time, 5-wise independence suffices.

4 Analysis of Randomized Search Trees

In this section we analyze a number of interesting quanitities in randomized search trees and derive their expected values. These quantities are:

$D(x)$, the *depth* of node x in a tree, in other words the number of nodes on the path from x to the root;

$S(x)$, the *size* of the subtree rooted at node x, i.e. the number of nodes contained in that subtree;

$P(x, y)$, the length of the unique *path* between nodes x and y in the tree, i.e. the number of nodes on that path;

$SL(x)$ and $SR(x)$, the length of the right *S*pine of the *L*eft subtree of x and the length of the left *S*pine of the *R*ight subtree of x; by *left spine* of a tree we mean the root together with the left spine of the root's left subtree; the *right spine* is defined analogously.

Throughout this section we will deal with the treap T for a set $X = \{x_i = (k_i, p_i) | 1 \leq i \leq n\}$ of n items, numbered by increasing key rank, i.e. $k_i < k_{i+1}$ for $1 \leq i < n$. The priorities p_i will be random variables. Since for a fixed set of keys k_i the shape of the treap T for X depends on the priorities, the quantities of our interest will be random variables also, for which it makes sense to analyze expectations, distributions, etc.

Our analysis is greatly simplified by the fact that each of our quantities can be represented readily by two types of indicator random variables:

$A_{i,j}$ is 1 if x_i is an ancestor of x_j in T and 0 otherwise

$C_{i;\ell,m}$ is 1 if x_i is a common ancestor of x_ℓ and x_m in T and 0 otherwise

We consider each node an ancestor of itself. In particular we have:

Theorem 4.1 *Let $1 \leq \ell, m \leq n$, and let $\ell < m$.*

(i) $D(x_\ell) = \sum_{1 \leq i \leq n} A_{i,\ell}$

(ii) $S(x_\ell) = \sum_{1 \leq j \leq n} A_{\ell,j}$

(iii) $P(x_\ell, x_m) = 1 + \sum_{1 \leq i < m} (A_{i,\ell} - C_{i;\ell,m}) + \sum_{\ell < i \leq n} (A_{i,m} - C_{i;\ell,m})$

(iv) $SL(x_\ell) = \sum_{1 \leq i < \ell} (A_{i,\ell-1} - C_{i;\ell-1,\ell})$

$SR(x_\ell) = \sum_{\ell < i \leq n} (A_{i,\ell+1} - C_{i;\ell,\ell+1})$

Proof. (i) and (ii) are clear, since the depth of a node is nothing but the number of its ancestors, and the size of its subtree is nothing but the number of nodes for which it is ancestor.

For (iii) note that the path from x_ℓ to x_m is divided by their lowest common ancestor v into two parts. The part from x_ℓ to v, which consists exactly of the ancestors of x_ℓ minus the common ancestors of x_ℓ and x_m, is accounted for by the first sum. (This would be clear if the index range of that sum were between 1 and n. The smaller index range is justifed by the fact that for $\ell < m \leq i$ the node x_i is an ancestor of x_ℓ iff it is a common ancestor of x_ℓ and x_m, i.e. $A_{i,\ell} = C_{i;\ell,m}$ in that range.) Similarly the second sum accounts for the path between x_m and v. Since v is not counted in either of those two parts, the $+1$ correction factor is needed.

For (iv) it suffices to consider $SL(x_\ell)$, by symmetry. If x_ℓ has a left subtree, then the lowest node on its right spine must be $x_{\ell-1}$. It is clear that in that case the spine consists exactly of the ancestors of $x_{\ell-1}$ minus the ancestors of x_ℓ; but the latter are the common ancestor of $x_{\ell-1}$ and x_ℓ. Also, no x_i with $i \geq \ell$ can lie on that spine.

If x_ℓ has no left subtree, then either $\ell = 1$, in which case the formula correctly evaluates to 0, or $x_{\ell-1}$ is an ancestor of x_ℓ, in which case every ancestor of $x_{\ell-1}$ is a common ancestor of $x_{\ell-1}$ and x_ℓ, and the formula also correctly evaluates to 0. □

If we let $a_{i,j} = \mathrm{Ex}[A_{i,j}]$ and $c_{i;\ell,m} = \mathrm{Ex}[C_{i;\ell,m}]$, then by the linearity of expectations we immediately get the following:

Corollary 4.2 *Let $1 \le \ell, m \le n$, and let $\ell < m$.*

(i) $\mathrm{Ex}[D(x_\ell)] = \sum_{1\le i\le n} a_{i,\ell}$

(ii) $\mathrm{Ex}[S(x_\ell)] = \sum_{1\le j\le n} a_{\ell,j}$

(iii) $\mathrm{Ex}[P(x_\ell, x_m)] = 1 + \sum_{1\le i<m}(a_{i,\ell} - c_{i;\ell,m}) + \sum_{\ell<i\le n}(a_{i,m} - c_{i;\ell,m})$

(iv) $\mathrm{Ex}[SL(x_\ell)] = \sum_{1\le i<\ell}(a_{i,\ell-1} - c_{i;\ell-1,\ell})$
$\mathrm{Ex}[SR(x_\ell)] = \sum_{\ell<i\le n}(a_{i,\ell+1} - c_{i;\ell,\ell+1})$

In essence our whole analysis has now been reduced to determining the expectations $a_{i,j}$ and $c_{i;\ell,m}$. Note that since since we are dealing with indicator 0-1 random variables, we have

$$a_{i,j} = \mathrm{Ex}[A_{i,j}] = \Pr[A_{i,j} = 1] = \Pr[x_i \text{ is ancestor of } x_j], \quad \text{and}$$

$$c_{i;\ell,m} = \mathrm{Ex}[C_{i;\ell,m}] = \Pr[C_{i;\ell,m} = 1] = \Pr[x_i \text{ is common ancestor of } x_\ell \text{ and } x_m]\,.$$

Determining the probabilities and hence the expectations is made possible by the following *ancestor lemma*, which completely characterizes the ancestor relation in treaps.

Lemma 4.3 *Let T be the treap for X, and let $1 \le i, j \le n$.*

Then, assuming that all priorities are distinct, x_i is an ancestor of x_j in T iff among all x_h, with h between and including i and j, the item x_i has the largest priority.

Proof. Let x_m be the item with the largest priority in T, and let $X' = \{x_\nu | 1 \le \nu < m\}$ and $X'' = \{x_\mu | m < \mu \le n\}$. By the definition of a treap, x_m will be the root of T and its left and right subtrees will be treaps for X' and X'' respectively.

Clearly our ancestor characterization is correct for all pairs of nodes involving the root x_m. It is also correct for all pairs $x_\nu \in X'$ and $x_\mu \in X''$: they lie in different subtrees, and hence are not ancestor-related. But indeed the largest priority in the range between ν and μ is realized by x_m and not by x_ν or x_μ.

Finally, by induction (or recursion, if you will) the characterization is correct for all pairs of nodes in the treap for X' and for all pairs of nodes in the treap for X''. □

As an immediate consequence of this ancestor lemma we obtain an analogous *common ancestor lemma.*

Lemma 4.4 *Let T be the treap for X, and let $1 \le \ell, m, i \le n$ with $\ell < m$. Let p_ν denote the priority of item x_ν.*

Then, assuming that all priorities are distinct, x_i is a common ancestor of x_ℓ and x_m in T iff

$$\begin{aligned} p_i &= \max\{p_\nu | i \le \nu \le m\} && \text{if } 1 \le i \le \ell \\ p_i &= \max\{p_\nu | \ell \le \nu \le m\} && \text{if } \ell \le i \le m \\ p_i &= \max\{p_\nu | \ell \le \nu \le i\} && \text{if } m \le i \le n, \end{aligned}$$

which can be summarized as

$$p_i = \max\Big\{p_\nu |\ \min\{i,\ell,m\} \le \nu \le \max\{i,\ell,m\}\Big\}\,,$$

Now we have all tools ready to analyze our quantities of interest. We will deal with the case of unweighted and weighted randomized search trees separately.

4.1 The unweighted case

The last two lemmas make it easy to derive the ancestor probabilities $a_{i,j}$ and $c_{i;\ell,m}$.

Corollary 4.5 *In an (unweighted) randomized search tree x_i is an ancestor of x_j with probability $1/(|i-j|+1)$, in other word we have*

$$a_{i,j} = 1/(|i-j|+1)\,.$$

Proof. According to the ancestor lemma we need the priority of x_i to be the largest among the priorities of the $|i-j|+1$ items between x_i and x_j. Since in an unweighted randomized search tree these priorities are independent identically distributed continuous random variables this happens with probability $1/(|i-j|+1)$. □

Corollary 4.6 *Let $1 \le \ell < m \le n$.*
In the case of unweighted randomized search trees the expectation for common ancestorship is given by

$$c_{i;\ell,m} = \begin{cases} 1/(m-i+1) & \text{if } 1 \le i \le \ell \\ 1/(m-\ell+1) & \text{if } \ell \le i \le m \\ 1/(i-\ell+1) & \text{if } m \le i \le n, \end{cases}$$

which can be summarized as

$$c_{i;\ell,m} = 1/(\max\{i,\ell,m\} - \min\{i,\ell,m\} + 1)\,.$$

Proof. Analogous to the previous proof. □

Now we can put everything together to obtain exact expressions and closed form upper bounds for the expectations of the quantities of interest. The expressions involve harmonic numbers, defined as $H_n = \sum_{1 \le i \le n} 1/i$. Note the standard bounds $\ln n < H_n < 1 + \ln n$ for $n > 1$.

Theorem 4.7 *Let $1 \le \ell, m \le n$, and let $\ell < m$. In an unweighted randomized search tree of n nodes the following expectations hold:*

(i) $$\begin{aligned} \mathrm{Ex}[D(x_\ell)] &= H_\ell + H_{n+1-\ell} - 1 \\ &< 1 + 2 \cdot \ln n \end{aligned}$$

(ii) $$\begin{aligned} \mathrm{Ex}[S(x_\ell)] &= H_\ell + H_{n+1-\ell} - 1 \\ &< 1 + 2 \cdot \ln n \end{aligned}$$

(iii) $$\begin{aligned} \mathrm{Ex}[P(x_\ell, x_m)] &= 4H_{m-\ell+1} - (H_m - H_\ell) - (H_{n+1-\ell} - H_{n+1-m}) - 3 \\ &< 1 + 4 \cdot \ln(m-\ell+1) \end{aligned}$$

(iv) $$\mathrm{Ex}[SL(x_\ell)] = 1 - 1/\ell$$
$$\mathrm{Ex}[SR(x_\ell)] = 1 - 1/(n+1-\ell)$$

Proof. Just use Corollary 4.2 and plug in the values from Corollaries 4.5 and 4.6. □

It is striking that in an unweighted randomized search tree the expected number of ancestors of a node x exactly equals the expected number of its descendants. However, it is reminiscent, although apparently unrelated, to the following easily provable fact: in any rooted binary tree T the average node depth equals the average size of a subtree.

We want to point out a major difference between the random variables $D(x_\ell)$ and $S(x_\ell)$. Although both have the same expectation, they have very different distributions. $D(x_\ell)$ is very tightly concentrated around its expectation, whereas $S(x_\ell)$ is not. For instance, one can easily see that in an n-node tree $\Pr[D(x_1) = n] = 1/n!$, whereas $\Pr[S(x_1) = n] = 1/n$. Using the ancestor lemma it is not too hard to derive the exact distribution of $S(x_\ell)$: one gets $\Pr[S(x_\ell) = n] = 1/n$ and $\Pr[S(x_\ell) = s] = O(1/s^2)$ for $1 \leq s < n$. We leave the details as an exercise to the reader. Here we briefly prove a concentration result for $D(x_\ell)$.

Lemma 4.8 *In an unweighed randomized search tree with $n > 1$ nodes we have for index $1 \leq \ell \leq n$ and any $c > 1$*

$$\Pr[D(x_\ell) \geq 1 + 2c \ln n] < 2(n/e)^{-c \ln(c/e)}.$$

Proof. Recall that $D(x_\ell) = \sum_{1 \leq i \leq n} A_{i,\ell}$, where $A_{i,\ell}$ indicates whether x_i is an ancestor of x_ℓ. Let $L = \sum_{1 \leq i < \ell} A_{i,\ell}$ count the "left ancestors" of x_ℓ and let $R = \sum_{\ell < i \leq n} A_{i,\ell}$ count the "right ancestors." Since $D(x_\ell) = 1 + L + R$, in order for $D(x_\ell)$ to exceed $1 + 2c \ln n$ at least one of L and R has to exceed $c \ln n$. It now suffices to prove that the probability of either of those events is bounded by $(n/e)^{-c \ln(c/e)}$. We will just consider the case of R. The other case is symmetric.

The crucial observation is that for $i > \ell$ the 0-1 random variables $A_{i,\ell}$ are independent and one can therefore apply so-called Chernoff bounds [14, 9] which, in one form, state that if random variable Z is the sum of independent 0-1 variables, then $\Pr[Z \geq c \cdot \mathrm{Ex}[Z]] < e^{-c \ln(c/e) \mathrm{Ex}[Z]}$.

In order to make the bound independent of ℓ we consider random variable $R' = \sum_{\ell < i < \ell + n} A_{i,\ell}$, where we use additional independent 0-1 variables $A_{i,\ell}$ for $i > n$, with $\mathrm{Ex}[A_{i,\ell}] = 1/(i - \ell + 1)$. Obviously for any $k > 0$ we have $\Pr[R \geq k] \leq \Pr[R' \geq k]$. Using $\mathrm{Ex}[R'] = H_n - 1$, and using $\ln n - 1 < H_n - 1 < \ln n$ and applying the Chernoff bound we get the desired

$$\begin{aligned} &\Pr[R \geq c \ln n] \leq \Pr[R' \geq c \ln n] < \Pr[R' \geq c(H_n - 1)] \\ &< e^{-c \ln(c/e)(H_n - 1)} < e^{-c \ln(c/e)(\ln n - 1)} = (n/e)^{-c \ln(c/e)}. \end{aligned}$$

□

Note that this lemma implies that the probability of least one of the n nodes in an unweighted randomized search tree having depth greater than $2c \ln n$ is bounded by $n(n/e)^{-c \ln(c/e)}$. In other words, the probability that the height of a randomized search tree is more than logarithmic is exceedingly small, and hence the tree's expected height is logarithmic also. In contrast to the random variables studied in this section the random variable h_n, the height of an n-node unweighted randomized search tree, is quite difficult to analyze exactly. Devroye [10], though, has shown that $h_n / \ln n \to \gamma$ almost surely, as $n \to \infty$, where $\gamma = 4.31107...$ is the unique solution of $\gamma \ln(2e/\gamma) = 1$ with $\gamma \geq 2$.

4.2 The weighted case

Recall that in the weighted case every item x_i has associated with it a positive integer weight w_i, and the weighted randomized search tree for a set of items uses as priority for x_i the maximum of w_i independent continuous random variables, each with the same distribution.

For $i \le j$ let $w_{i:j}$ denote $\sum_{i \le h \le j} w_h$, and for $i > j$ define $w_{i:j} = w_{j:i}$. Let $W = w_{1:n}$ denote the total weight.

Corollary 4.9 *In an weighted randomized search tree x_i is an ancestor of x_j with probability $w_i/w_{i:j}$, in other word we have*

$$a_{i,j} = w_i/w_{i:j}\,.$$

Proof. According to the ancestor lemma we need the priority of x_i to be the largest among the priorities of the items between x_i and x_j. This means one of the w_i random variables "drawn" for x_i has to be the largest of the $w_{i:j}$ random variables "drawn" for the items between x_i and x_j. But since these random variables are identically distributed this happens with the indicated probability. □

Corollary 4.10 *Let $1 \le \ell < m \le n$.*
In the case of unweighted randomized search trees the expectation for common ancestorship is given by

$$c_{i;\ell,m} = \begin{cases} w_i/w_{i:m} & \text{if } 1 \le i \le \ell \\ w_i/w_{\ell:m} & \text{if } \ell \le i \le m \\ w_i/w_{\ell:i} & \text{if } m \le i \le n \end{cases}$$

Proof. Analogous to the previous proof, but based on the common ancestor lemma. □

Now we just need to plug our values into Corollary 4.2 to get the following:

Theorem 4.11 *Let $1 \le \ell, m \le n$, and let $\ell < m$. In an weighted randomized search tree with n nodes of total weight W the following expectations hold:*

$$\begin{array}{lllll}
\textit{(i)} & \mathrm{Ex}[D(x_\ell)] & = & \sum_{1\le i\le n} w_i/w_{i:\ell} \\
 & & < & 1 + 2\cdot\ln(W/w_\ell) \\
\textit{(ii)} & \mathrm{Ex}[S(x_\ell)] & = & \sum_{1\le i\le n} w_\ell/w_{i:\ell} \\
\textit{(iii)} & \mathrm{Ex}[P(x_\ell, x_m)] & = & 1 + \sum_{1\le i<\ell}(w_i/w_{i:\ell} - w_i/w_{i:m}) \\
 & & & \quad + \sum_{\ell\le i\le m}(w_i/w_{\ell:i} + w_i/w_{i:m} - 2w_i/w_{\ell:m}) \\
 & & & \quad + \sum_{m<i\le n}(w_i/w_{m:i} - w_i/w_{\ell:i}) \\
 & & < & 1 + 2\cdot\ln(w_{\ell:m}/w_\ell) + 2\cdot\ln(w_{\ell:m}/w_m) \\
\textit{(iv)} & \mathrm{Ex}[SL(x_\ell)] & = & \sum_{1\le i<\ell}\Big(w_i/w_{i:\ell-1} - w_i/w_{i:\ell}\Big) \\
 & & < & 1 + \ln(1 + w_\ell/w_{\ell-1}) \\
 & \mathrm{Ex}[SR(x_\ell)] & = & \sum_{\ell<i\le n}\Big(w_i/w_{\ell+1:i} - w_i/w_{\ell:i}\Big) \\
 & & < & 1 + \ln(1 + w_\ell/w_{\ell+1})
\end{array}$$

Proof. The exact expressions follow from Corollaries 4.9 and 4.10.

The quoted upper bounds are consequences of the following two inequalities that can easily be verified considering the area underneath the curve $f(x) = 1/x$:

$$\alpha/A \le \ln A - \ln(A-\alpha) \quad \text{for } 0 \le \alpha < A \tag{1}$$

$$\alpha/A - \alpha/B \le (\ln A - \ln(A-\alpha)) - (\ln B - \ln(B-\alpha)) \quad \text{for } 0 \le \alpha < A \le B \tag{2}$$

For instance, to prove (i) we can apply inequality (1) and use the principle of telescoping sums to derive

$$\sum_{1\le i<\ell} w_i/w_{i:\ell} < \sum_{1\le i<\ell} (\ln w_{i:\ell} - \ln w_{i+1:\ell}) = \ln w_{1:\ell} - \ln w_\ell = \ln(w_{1:\ell}/w_\ell) < \ln(W/w_\ell)$$

and analogously $\sum_{\ell<i\le n} w_i/w_\ell < \ln(w_{\ell:n}/w_\ell) \le \ln(W/w_\ell)$, which, adding in the 1 stemming from $i=\ell$, yields the stated bound.

Similarly we can use inequality (2) to derive (iv); we just show the case of $\mathrm{Ex}[SL(x_\ell)]$:

$$\begin{aligned}
\textstyle\sum_{1\le i<\ell}\Big(w_i/w_{i:\ell-1} - w_i/w_{i:\ell}\Big) &< 1 + \textstyle\sum_{1\le i<\ell-1}\Big((\ln w_{i:\ell-1} - \ln w_{i+1:\ell-1}) - (\ln w_{i:\ell} - \ln w_{i+1:\ell})\Big)\\
&= 1 + (\ln w_{1:\ell-1} - \ln w_{\ell-1:\ell-1}) - (\ln w_{1:\ell} - \ln w_{\ell-1:\ell})\\
&< 1 + \ln w_{\ell-1:\ell} - \ln w_{\ell-1:\ell-1}\\
&= 1 + \ln(1 + w_\ell/w_{\ell-1})
\end{aligned}$$

For proving (iii) one uses inequality (2) and telescoping to bound the first sum by $\ln(w_{\ell:m}/w_\ell)$ and the third sum by $\ln(w_{\ell:m}/w_m)$. The middle sum can be broken into three pieces. The third piece evaluates to -2, and using inequality (1) the first piece is bounded by $1+\ln(w_{\ell:m}/w_\ell)$ and the second piece by $1+\ln(w_{\ell:m}/w_m)$. Together this yields the indicated bound. □

With sufficiently uneven weight assignments the bounds listed in this theorem can become arbitrarily bad, in particular they can exceed n, the number of nodes in the tree, which is certainly an upper bound for every one of the discussed quantities. One can obtain somewhat stronger bounds by optimizing over all possible weight assignments while leaving the total weight W and the weight of x_ℓ (and possibly x_m) fixed. Although this is possible in principle it seems technically quite challenging in general. We just illustrate the case $D(x_1)$.

Which choice of w_i maximizes $\mathrm{Ex}[D(x_1)] = \sum_{1\le i\le n} w_i/w_{1:i}$ while leaving w_1 and $W = w_{1:n}$ fixed? Rewrite the sum as

$$1 + \sum_{1<i\le n} \frac{w_{1:i} - w_{1:i-1}}{w_{1:i}} = 1 + \sum_{1<i\le n} (1 - w_{1:i-1}/w_{1:i}) = n - \sum_{1<i\le n} w_{1:i-1}/w_{1:i}\,.$$

A little bit of calculus shows that the last sum is minimized when all its summands are the same, which, using our boundary conditions, implies that each of them is $(w_1/W)^{1/(n-1)}$. Thus it follows that $S = \mathrm{Ex}[D(x_1)]$ is bounded by

$$n - (n-1)(w_1/W)^{1/(n-1)} = 1 + (n-1)\Big(1 - (w_1/W)^{1/(n-1)}\Big)\,.$$

which, however, is not a particularly illuminating expression, except that it is easily seen never to exceed n.

5 Analysis of Operations on Randomized Search Trees

In this section we discuss the various operations on unweighted and weighted randomized search trees and derive their expected efficiency, thus proving all bounds claimed in Theorem 3.1 and Theorem 3.2.

5.1 Searches

A succesful search for item x in a treap T commences at the root of T and, using the key of x traces out the path from the root to x. Clearly the time taken by such a search is proportional to the length of the path or, in other words, the number of ancestors of x. Thus the expected time for a search for x is propotional to the expected number of its ancestors, which is $O(\log n)$ in the unweighted and $O(1 + \log(W/w(x)))$ in the weighted case by Theorems 4.7 and 4.11.

An unsuccessful search for a key that falls between the keys of successive items x^- and x^+ takes expected time $O(\log n)$ in the unweighted and $O(\log(W/\min\{w(x^-), w(x^+)\}))$ in the weighted case, since such a search must terminate either at x^- or x^+ (ignoring the two cases where x^- or x^+ does not exist).

5.2 Insertions and Deletions

An item is inserted into a treap by first attaching it at the proper leaf position which is located by an (unsuccessful) search using the item's key; then the item is rotated up the treap, as long as it has a parent with smaller priority. Clearly the number of those rotations can be at most the length of the search path that was just traversed. Thus the insertion time is proportional to the time needed for an unsuccessful search, which in expectation is $O(\log n)$ in the unweighted and $O(\log(W/\min\{w(x^-), w(x^+)\}))$ in the weighted case, where x^- and x^+ are the predecessor and successor of x in the resulting tree.

The deletion operation requires essentially the same time since it basically reverses an insertion operation: first the desired item is located in the tree using its key; then it is rotated down until it is in leaf position, at which point it is clipped away; during those downward rotations the decision whether to rotate right or left is dictated by the priorities of the two current children, as the one with larger priority must be rotated up.

What is still interesting is the expected number of rotations that occur during an update. Since in terms of rotations a deletion is an exact reversal of an insertion it suffices to analyze the number of rotations that occur during a deletion.

So let x be an item in a treap T to be deleted from T. Although we cannot tell just from the tree structure of T which downwards rotations will be performed, we can tell how many there will be: their number is the sum of the lengths of the right spine of the left subtree of x and the left spine of the right subtree of x. The correctness of this fact can be seen by observing that a left-rotation of x has the effect of shortening the left spine of its right subtree by one; a right-rotation of x shortens the right spine of the left subtree.

Thus we know from Theorems 4.7 and 4.11 that in expectation the number of rotations per update is less than 2 in the unweighted case and less than $2+\ln(1+w(x)/w(x^-))+\ln(1+w(x)/w(x^+)) = O\Big(1 + \log(1 + w(x)/w(x^-) + w(x)/w(x^+))\Big)$ in the weighted case.

5.3 Insertion and Deletions using Handles

Having a handle, i.e. a pointer, to a node to be deleted from a treap of course obviates the need to search for that node. Only the downward rotations to leaf position, and the clipping away needs to be performed, which results in expected time bound of $O(1)$ in the unweighted and $O\Big(1 + \log(1 + w(x)/w(x^-) + w(x)/w(x^+))\Big)$ in the weighted case.

Similarly, if we know the leaf position at which a new node x is to be inserted (which is always either the resulting predecessor x^- or successor x^+ of x), then, after attaching the new leaf, only the upward rotations need to be performed, which also results in the expected time

bounds just stated. If just a pointer to x^- is available, and x cannot be attached as a leaf to x^- since it has a nonempty right subtree, then the additional cost of locating x^+ is incurred. Note that in this case x^+ is the bottom element of the left spine of that right subtree of x^- and locating it amounts to traversing the spine. The expeced cost for this is therefore $O(1)$ in the unweighted and $O\Big(1+\log(1+w(x^-)/w(x^+))\Big)$, which results in a total expected cost of $O(1)$ and $O\Big(1+\log(1+w(x)/w(x^-)+w(x)/w(x^+)+w(x^-)/w(x^+))\Big)$ in the respective cases.

Note that in contrast to ordinary insertions, where the sequence of ancestors can be computed during the search for the leaf position, insertions using handles require that ancestor information is stored explicitly in the tree, i.e. parent pointers must be available, so that the upward rotations can be performed. Deletions based on handles also require parent pointers, since when the node x to be deleted is rotated down one needs to know its parent y so that the appropriate child pointer of y can be reset.

5.4 Finger searches

In a finger search we assume that we have a "finger" or "handle" or "pointer" on item x in a tree and using this information we quickly want to locate the item y with given key k. Without loss of generality we assume that $x = x_\ell$ and $y = x_m$ with $\ell < m$. The most natural way to perform such a search is to follow the unique path between x_ℓ and x_m in the tree. The expected length of this path is given in Theorems 4.7 and 4.11 for weighted and unweighted randomized search trees, which immediately would yield the bounds of Theorems 3.1 and 3.2. However, things are not quite that easy since it is not clear that one can traverse the path between x_ℓ and x_m in time proportional to its length. Such a traversal would, in general, entail going up the tree, starting at x_ℓ, until the lowest common ancestor u of x_ℓ and x_m is reached, and then going down to x_m. But how can one tell that u has been reached (For all we know, x_ℓ may be u already.)?

If one reaches during the ascent towards the root a node z whose key is larger than the "search key" k, then one has gone too far (similarly, if one reaches the root of the entire tree via its right child). The desired u was the last node on the traversed path that was reached via its left child, or, if no such node exists, the start node x. This *excess path* between u and z may be much longer than the path between x and y and traversing this path may dominate the running time of the finger search.

There are several ways of dealing with this excess path problem. One is to try to argue that in expectation such an excess path has only constant length. However, this turns out to be true for the unweighted case only. Other methods store extra information in the tree that either allows to shortcut such excess paths or obviates the need to traverse them.

Extra pointers

Let us first discuss a method of adding extra pointers. Define the *left parent* of v is the first node on the path from v to the root whose key is smaller than the key of v, and the *right parent* of v is the first node on the path with larger key. Put differently, a node is the right parent of all nodes on the right spine of its left subtree and the left parent of all nodes on the left spine of its right subtree. We store with every node v in the tree besides the two children pointers also two pointers set to the left parent and right parent of v, respectively, or to nil if such an parent does not exist. Such parent pointers have been used before in [3]. Please note that during a rotation or when adding or clipping a leaf these parent pointers can be maintained at constant cost. Thus no significant increase in update times is incurred.

A finger search now proceeds as follows: starting at x chase right parent pointers until either a nil pointer is reached or the next node has key larger than k (using a sentinel with key $+\infty$ obviates this case distinction). At this point one has reached the lowest common ancestor u of x and y, and using the key k one can find the path from u to y following children pointers in the usual fashion. Note that the number of links traversed is at most one more than the length of the path between x and y, and that it can be much smaller because of the shortcuts provided by the right parent pointers.

Extra keys

With every node u in the tree one stores $\min(u)$ and $\max(u)$, the smallest and largest key of any item in the subtree rooted at u. With this information one can tell in constant time whether an item can possibly be in the subtree rooted at u: its key has to lie in the range between $\min(u)$ and $\max(u)$. The common ancestor of x_ℓ and x_m is then the first node z on the path from x_ℓ to the root with $\min(z) \leq k \leq \max(z)$.

Maintaining this min()/max() information clearly only takes constant time during rotations. However, when a leaf is added during an insertion or is clipped away at the end of a deletion the min()/max() information needs to be updated on an initial portion of the path toward the root. To be specific, consider the change when leaf x is removed. The nodes for which the min() information changes are exactly the nodes on the left spine of the right subtree of the *predecessor* of x. The max() information needs to be adjusted for all nodes on the right spine of the left subtree of the *successor* of x. The expected lengths of those spines are given in Theorems 4.7 and 4.11. They need to be added to the expected update times, which asymptotically is irrelevant in the case of unweighted randomized search trees, but can be the dominant factor in the case of weighted trees.

Finally, to make this argument applicable to the items with smallest and largest keys in the tree, one needs to maintain treaps with two sentinel nodes that are never removed: one with key $-\infty$ and one with key $+\infty$ (both with random priorities).

Relying on short excess paths

Finally let us consider the method, suggested to us by Mehlhorn and Raman [22], that traverses the excess path and relies on the fact that in expectation this path is short, at least for unweighted trees. (In the weighted case one can concoct examples where this expectation can become arbitrarily large.) The advantage of this method is that only the one usual parent pointer needs to be stored per node, and not two parent pointers. However, as we shall see soon, one also need to store one additional bit per node.

As before let u be the lowest common ancestor between $x = x_\ell$ and $y = x_m$ and let the right parent of u be z, the *endnode* of the excess path. Call the nodes on the path between, but not including u and z the *excess nodes*. Note that $u = x_\nu$ for some ν with $\ell \leq \nu \leq m$ (namely the one with largest priority in that range), and $z = x_j$ for some j with $m < j \leq n$, and any excess node must be some x_i with $1 \leq i < \ell$. We estimate the expected number of excess nodes by introducing for each i with $1 \leq i < \ell$ a 0-1 random variable X_i indicating whether or not x_i is an excess node and summing the expectations. Now $\mathrm{Ex}[X_i]$ is the probability that x_i is an excess node, which we represent as the sum of the probabilities of the disjoint events E_{ij} that x_i is an excess node and z is x_j. Now, for E_{ij} to happen x_j must be the right parent of x_i and of u, or in other words, x_i and u must both be on the right spine of the left subtree of x_j. This happens if in the range between i and j, item x_j has the largest priority and x_i has the second largest, and if u has the largest priority in the range from ℓ to $j-1$. By the definition of u this last condition can be reformulated

that the largest priority in the range ℓ to $j-1$ occur in the range ℓ to m. By the assumption that all priorities are independently, identically distributed, it follows that

$$\Pr[E_{ij}] = \frac{1}{(j-i+1)} \cdot \frac{1}{(j-i)} \cdot \frac{m-\ell+1}{(j-\ell)}$$

and the expected number of excess nodes is

$$\sum_{1 \le i < \ell} \mathrm{Ex}[X_i] = \sum_{1 \le i < \ell} \sum_{m < j \le n} \Pr[E_{ij}] = \sum_{1 \le i < \ell} \sum_{m < j \le n} \frac{1}{(j-i+1)} \cdot \frac{1}{(j-i)} \cdot \frac{m-\ell+1}{(j-\ell)}.$$

Summing first over i and applying the bound $\sum_{a \le k < b} 1/k(k+1) < 1/a$ one can see that the double sum is upper bounded by

$$\sum_{m < j \le n} \frac{m-\ell+1}{(j-\ell)(j-\ell+1)},$$

which in turn by applying the same bound can be seen to be upper bounded by 1. Thus we have proved that in the case of unweighted trees the expected number of excess nodes is always less than one.

The above analysis relies on the existence of z, i.e. the lowest common ancestor u must have a right parent. This need not be the case: u can lie on the right spine of the entire tree. In this case a naive finger search would continue going up past u along the spine until the root is reached. This could be very wasteful. Consider for instance a finger search from x_{n-1} to x_n which with probability $1/2$ would this way traverse the entire spine, which has $O(\log n)$ expected length. To prevent this we require that every node has stored with it information indicating whether it is on the right or left spine of the entire tree. With this information a finger search can stop going up as soon as the right spine of the entire tree is reached, and thus in effect only the path between x and y is traversed.

5.5 Fast finger searches

Now assume we have a "finger" or "pointer" on item x and also one on item z and we want to quickly locate item y with key k. A natural thing to do is to start finger searches both at x and at z that proceed in lockstep in parallel until the first search reaches y. If X and Z are the lengths of the paths from x and z to y, respectively, then by the previous subsection the time necessary for such a parallel finger search would be proportional to $\min\{X, Z\}$, and the expected time would be proportional to $\mathrm{Ex}[\min\{X, Z\}] \le \min\{\mathrm{Ex}[X], \mathrm{Ex}[Z]\}$.

This can be exploited as follows: Always maintain a pointer to $Tmin$ and $Tmax$, the smallest and largest key items in the tree. Note that this can be done with constant overhead per update. If one now wants to perform a finger search starting at some node $x = x_\ell$ for some node $y = x_m$ and, say, y's key is larger than the one of x, then start in parallel a finger search from $Tmax = x_n$. The expected time for this search is now proportional to the minimum of the expected path lengths from x and $Tmax$ to y, which by Theorem 4.7 is $O(\min\{\log(m-\ell+1), \log(n-m+1)\}$. If we let $d = m-\ell+1$ be the key distance between x and y, then this is at most $O(\min\{\log d, \log(n-d)\}) = O(\log \min\{d, n-d\})$, as claimed in Theorem 3.1, since $n-m+1 \le n-d+1$.

5.6 Joins and splits

Two treaps T_1 and T_2 with all keys in T_1 smaller than all keys in T_2 can be joined by creating a new tree T whose root r has as left child T_1 and right child T_2, and then deleting r fromt T.

Creating T takes constant time. Deleting r, as we observed before, takes the time proportional to the length of the right spine of T_1 plus the length of the left spine of T_2, i.e. the depth of the largest key item of T_1 plus the depth of the smallest key item of T_2. By Theorem 4.7 this is in expectation $O(\log m + \log n) = O(\log \max\{m, n\})$ if T_1 and T_2 are unweighted randomized search trees with m and n items, respectively. In the weighted case Theorem 4.11 implies an expected bound of $O(1 + \log \frac{W_1}{w(T_1 max)} + \log \frac{W_2}{w(T_2 min)})$, if we let W_i denote the total weight of T_i.

Splitting a treap T into a treap T_1 with all keys smaller than k and treap T_2 with all keys larger than k is achieved by essentially reversing the join operation: An item x with key k and infinite priority is inserted into T; because of its large priority the new item will end up as the root; the left and right subtrees of the root will be T_1 and T_2, respectively. This insertion works by first doing a search starting at the root to locate the correct leaf position for key k, adding item x as a leaf, and then rotating x up until it becomes the root. Note that the length of the search path is the same as the number of rotations performed, which in turn is the sum of the lengths of the right spine of T_1 and the left spine of T_2. Thus the time necessary to perform this split is proportional to the sum of the lengths of those two spines, and therefore the same expected time bounds hold as for joining T_1 and T_2.

5.7 Fast splits

How can one speed up the split operation described and analyzed above? One needs to shorten the search time for the correct leaf position and one needs to reduce the number of subsequent rotations. Let us assume the size (or total weight) of the eventual T_1 is small. Then it makes sense to determine the correct leaf position for x in T by a finger search starting at $Tmin$, the item in T with minimum key. Let z be the lowest common ancestor of $Tmin$ and the leaf added for x. Note that z is part of the path between $Tmin$ and x and that z must be on the left spine of T. Now, when rotating x up the tree one can stop as soon as x becomes part of the left spine of the current tree, or, in other words, as soon as z becomes the left child of x: The current left subtree of x will then contain exactly all items of T with key smaller than the splitting key k and forms the desired T_1. The tree T_2 is obtained by simply replacing the subtree rooted at x by the right subtree of x.

The time to do all this is proportional to the length L of the path in the original T between $Tmin$ and the leaf added for x: as argued before, the finger search takes this much time, and the number of subsequent rotations is exactly the number of edges on this path between x and z. In the unweighted case the expected value of L is by Theorem 4.7 $O(\log m)$ where m is the number of nodes that end up in T_1. This is not particularly good if T_1 is almost all of T. In this case it would be much better to proceed symmetrically: start the finger search at $Tmax$, the item in T with largest priority, and identify the path form $Tmax$ to x. The expectation of the length R of that path, and hence of the splitting time would then be $O(\log n)$, where n is the size of T_2. Of course in general one does not know in advance whether T_1 or T_2 is smaller. But one can circumvent this problem by starting both searches in lockstep in parallel, terminating both as soon as one succeeds. This would take $O(\min\{L, R\})$ time. And since $\mathrm{Ex}[\min\{L, R\}] \leq \min\{\mathrm{Ex}[L], \mathrm{Ex}[R]\}$, the total expected time for the fast split would be $O(\min\{\log m, \log n\}) = O(\log \min\{m, n\})$.

For the weighted case it now would seem to suffice to observe that by Theorem 4.11 the expectation of L is $O(1 + \log(W_1/w(T_1 min) + W_1/w(T_1 max)))$, where $T_1 max$ is the item with largest key in T_1, which is of course nothing but x^-, the predecessor of x in T. However, this is not quite true, since the new leaf x is not necessarily a child of x^-, but could be a child of x^+, the successor of x. In that case the path in T from x^- to x^+ (in other words the right spine of the left subtree of x^-) that needs to be traversed additionally after x^- has been reached could be quite long, namely $O(1 + \log(1 + w(x^-)/w(x^+))$. This additional traversal and cost can be avoided if one uses more

space and stores with every item in the tree a predecessor and succesor pointer. (Note that those additional pointers could be maintained with constant overhead during all update operations.)

What kind of information needs to be maintained so that such a split procedure can be implemented? One needs $Tmin$ and $Tmax$; since the finger searches start at $Tmin$ and $Tmax$ and the upward parts of the searches only proceed along the spines of T, one needs to maintain parent pointers only for spine nodes and not for all nodes.

5.8 Fast joins

In a fast join of T_1 and T_2 one tries to reverse the operations of a fast split. Starting at T_1max and T_2min one traverses the right spine RS of T_1 and the left spine LS of T_2 in a merge like fashion until either a node x of RS is found whose priority is larger than the one of the root of T_2 or, symmetrically, a node y of LS is found whose priority exceeds the one of the root of T_1. In the first case the subtree T_x of T_1 rooted at x is replaced by the join of T_x and T_2, computed using the normal join algorithm, and the root of T_1 becomes the root of the result. In the other case one proceeds symmetrically and the root of T_2 becomes the root of the result.

This in effect reverses the operation of a fast split, except that no finger searches had to be made. Therefore the time required for fast joining T_1 and T_2 into T is upper bounded by the time necessary to fast split T into T_1 and T_2.

5.9 Excisions

Let x and y be two items in an n-node treap T and assume we have a "finger" on at least one of them. We would like to extract from T the treap T' containing all d items with keys between and including the keys of x and y, leaving in T all the remaining items. First we show that we can do this in time proportional to $P(x, y)$ the length of the path from x to y plus $SL(x)$ and $SR(y)$, the lengths of the right spine of the left subtree of x and the left spine of the right subtree of y. By Theorem 4.7 the expectation of this would be $O(\log d)$.

With a finger search first identify the path connecting x and y, and with this the lowest common ancestor u of x and y, all in time $O(P(x, y))$. Let T_u be the subtree rooted at u. It suffices to show how to excise T' from T_u. Let LL be the sequence of nodes on the path from x to u with keys less than the key of x and let A be the sequence of the remaining nodes on that path, not including u. Symmetrically, let RR be the sequence of nodes on the path from y to u with keys greater than the key of y and let B be the sequence of the remaining nodes not including u. Now the desired treap T' has as its root u; its left subtree is formed by stringing together the nodes in A along with their right subtrees; its right subtree is formed by stringing together the nodes in B together with their left subtrees. Clearly T' can thus be formed in time $O(P(x, y))$.

The remaining pieces of T_u can be put back into a treap as follows: Form a treap L by stringing together the nodes in LL together with their left subtrees, adding the left subtree of x at the bottom of the right spine. Symmetrically, form a treap R by stringing together the nodes in RR, adding the right subtree of y at the bottom of the left spine. The remainder of T_u is constructed by stringing the members of LL into a tree L and RR into a tree R (see Figures 4 and 5). Clearly L and R can be constructed in time $O(P(x, y))$. In section 5.7 we showed that the time necessary for the join operation is proportional to sum of the lenghts of the right spine of L and the left spine of R. But this sum is at most $P(x, y) + SL(x) + SR(y)$.

The excision of T' from T can also be performed in expected $O(\log(n - d)$ time using the following method: Split T into treaps L and T'', where L contains all items with key less than the key of x; split T'' into the desired T' and R, where R contains all items with key greater then the

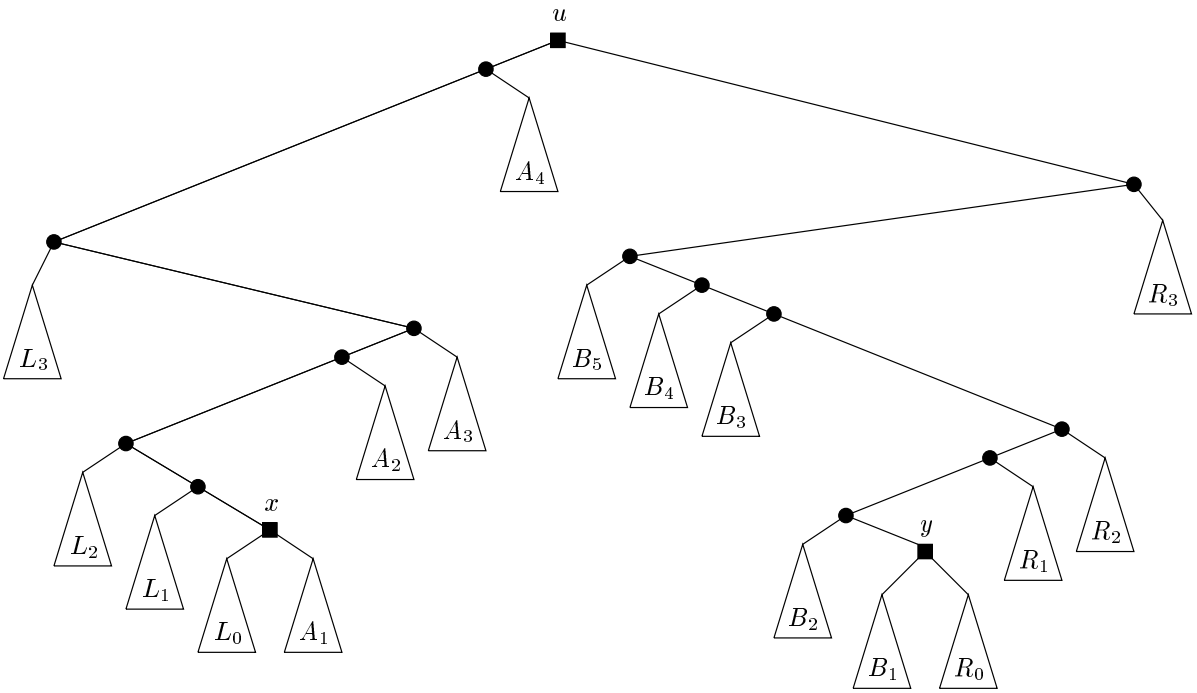


Figure 4: Subtree T_u before the excision

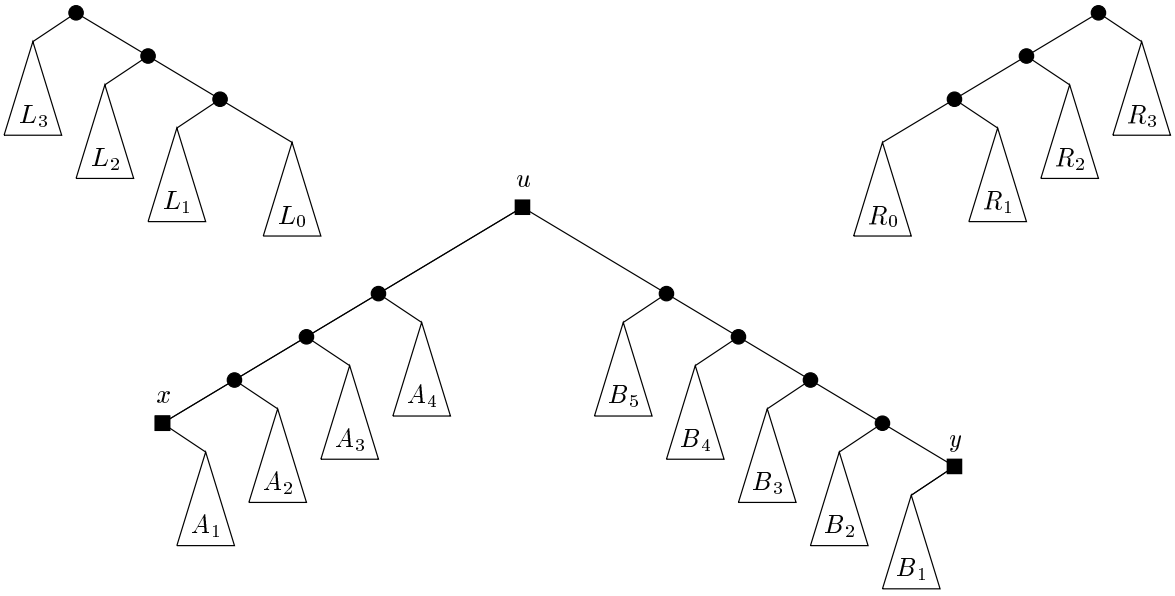


Figure 5: The trees L, T', and R

one of y; finally join L and R to form the remainder treap. L and R each contains at most $n-d$ nodes. Thus using the fast split method of section 5.7 and the normal join method this can all be performed in expected $O(\log(n-d))$ time.

Of course usually d is not know in advance. Thus we face the usual dilemma of which method to choose. This can be resolved as follows: In lockstep in parallel perform a finger search from x to y and perform a finger search from $Tmin$ to x followed by a finger search from $Tmax$ to y. If the search from x to y is completed first, use the first method for the excision, otherwise use the second method.

5.10 Expensive rotations

How expensive is it to maintain unweighted randomized search trees under insertions and deletions when the cost of a rotation is not constant but depends as $f(s)$ on the size s of the subtree that is being rotated?

Since an insertion operation is just the reverse of a deletion operation it suffices to analyze just deletions. Recall that in order to delete a node x it first has to be located and then it has to be rotated down into leaf position, where it is then clipped away. Since the search time and the clipping away is unaffected by the cost of rotations we only need to analyze the expected cost $R_f(x)$ of rotating x into leaf position.

As before let $x_1, \ldots, x_n$ be the items in the treap numbered by increasing key rank. Assume we want to delete x_k.

For $i \le k \le j$ let $E_{k;i,j}$ denote the event that at some point during this deletion x_k is the root of a subtree comprising the $j-i+1$ items x_i through x_j. Then the total expected cost of the deletion is given by

$$R_f(x_k) = \sum_{\substack{1 \le i \le k \\ k \le j \le n}} \Pr[E_{k;i,j}] \cdot f(j-i+1)\,.$$

We need to evaluate $\Pr[E_{k;i,j}]$. We can do this by characterizing those configurations of priorities that cause event $E_{k;i,j}$ to happen.

We claim that in the generic case $1 < i \le k \le j < n$ this event occurs exactly if among the $j-i+3$ items x_{i-1} through x_{j+1} the items x_{i-1}, x_k, x_{j+1} have the largest three priorities (the order among those three is irrelevant).

As a consequence of the ancestor lemma x_k is root of a subtree comprising x_i through x_j exactly if x_k has largest priority in that range and x_{i-1} and x_{j+1} each have larger priority than x_k, i.e. x_{i-1} and x_{j+1} have the largest two priorities among x_{i-1} through x_{j+1} and x_k has third largest priority. Now the deletion of x_k can be viewed as continuously decreasing its priority and rotating x_k down whenever its priority becomes smaller than the priority of one of its children, thus always maintaining a legal treap. This means that if x_{i-1}, x_k, x_{j+1} have the largest three priorities among x_{i-1} through x_{j+1}, at some point during the decrease x_k will have third largest priority and thus will be root of a tree comprising x_i through x_j as claimed. Similarly, x_k can never become the root of such a tree if x_{i-1}, x_k, x_{j+1} do not initially have the largest three priorities among x_{i-1} through x_{j+1}.

Using the same type of argument it is easy to see that the left-marginal event $E_{k;1,j}$ happens iff x_k and x_{j+1} have the largest two priorities among the $j+1$ itmes x_1 through x_{j+1}. The right-marginal event $E_{k;i,n}$ happens iff x_k and x_{i-1} have the largest two priorities among x_{j-1} through x_n. Finally $E_{k;1,n}$ of course occurs exaclty if x_k has the largest of all n priorities.

Since in an unweighted randomized search tree the priorities are independent identically dis-

tributed continuous random variables we can conclude from these characterizations that[3]

$$\Pr[E_{k;i,j}] = \begin{cases} 6/(j-i+1)^{\overline{3}} & \text{for } 1 < i \le k \le j < n \text{ (generic case)} \\ 2/j^{\overline{2}} & \text{for } i = 1 \text{ and } k \le j < n \text{ (left-marginal case)} \\ 2/(n-i+1)^{\overline{2}} & \text{for } 1 < i \le k \text{ and } j = n \text{ (right-marginal case)} \\ 1/n & \text{for } i = 1 \text{ and } j = n \text{ (full case).} \end{cases}$$

Substituting these values now yields

$$R_f(x_k) = \frac{f(n)}{n} + \sum_{k \le j < n} 2\frac{f(j)}{j^{\overline{2}}} + \sum_{1 < i \le k} 2\frac{f(n-i+1)}{(n-i+1)^{\overline{2}}} + \sum_{1 < i \le k} \sum_{k \le j < n} 6\frac{f(j-i+1)}{(j-i+1)^{\overline{3}}}.$$

In this form this expression is not particularaly illuminating. Rewriting it as a linear combination of $f(1), \ldots, f(n)$ yields for $k \le (n+1)/2$

$$R_f(x_k) = \frac{f(n)}{n} + \sum_{1 \le s < k} \frac{6}{(s+1)^{\overline{2}}} f(s) + \sum_{k \le s \le n-k} \left(\frac{6(k-1)}{s^{\overline{3}}} + \frac{2}{s^{\overline{2}}} \right) f(s) + \sum_{n-k < s < n} \left(\frac{6(n+1)}{s^{\overline{3}}} - \frac{2}{s^{\overline{2}}} \right) f(s)$$

and for $k > (n+1)/2$ we can exploit symmetry and get $R_f(x_k) = R_f(x_{n-k+1})$. This is the *exact* expectation and applies to any arbitrary real valued function f. For non-negative f it is easy to see that for any k this expression is upper bounded by

$$R_f(x_k) = O\Big(f(n)/n + \sum_{1 \le s < n} f(s)/s^2\Big).$$

From this the bounds of Theorem 3.1 about expensive rotations follow immediately.

There is a slightly less cumbersome way to arrive at this asymptotic bound. For any k and any $s < n$ item x_k can participate in at most s generic events $E_{k;i,j}$ with $j - i + 1 = s$, each having a probability of $O(1/s^3)$, which yields a contribution of $\mathrm{O}(f(s)/s^2)$ to $R_f(x_k)$. Similarly x_k can participate in at most two marginal events $E_{k;i,j}$ with $j - i + 1 = s$ each having a probability of of $O(1/s^2)$, which also yields a contribution of $O(f(s)/s^2)$ to $R_f(x_k)$. Finally x_k participates in exactly one "full" event $E_{k;1,n}$, which has probability $1/n$ and gives the $f(n)/n$ contribution to $R_f(x_k)$.

A different rotation cost model

The above analysis hinges on the fact that the probability that a particular three random variables are the smallest in a set of s independent identically distributed continuous random variables is $O(1/s^3)$. In the next section, which deals with limited randomness, we will see that it is advantageous if one only needs to consider two out of s variables, and not three.

In order to achieve this we will slightly change how the cost of a rotation is measured. If node x is rotated, then the cost will be $f(\ell) + f(r)$, where ℓ and r are the sizes of the left and right subtrees of x. This cost model is asymptotically indistinguishable from the previous one as long as there exists a constants c_1 and c_2 so that $c_1(f(\ell) + f(r)) \le f(\ell + r + 1) \le c_2(f(\ell) + f(r))$ for all $\ell, r \ge 0$. This is the case for all non-decreasing functions that satisfy $f(n+1) < c \cdot f(n/2)$, which is true essentially for all increasing functions that grow at most polynomially fast.

We will distribute this cost of a rotation at x to the individual nodes of the subtrees as follows: a node y that differs in key rank from x by j is charged $\Delta f(j) = f(j) - f(j-1)$, with the convention

[3] We use the notations $x^{\overline{m}} = x(x+1)\cdots(x+m-1)$ and $x^{\underline{m}} = x(x-1)\cdots(x-m+1)$.

that $f(0) = 0$. Since the right subtree of x contains the first r nodes that succeed x in key rank, the charge distributed in the right subtree is thus $\sum_{1 \le j \le r} \Delta f(j)$ which evaluates to $f(r)$ as desired. Symmetrically, the total charge in the left subtree adds up to $f(\ell)$.

Now let $x = x_k$ be the node to be deleted and let $y = x_{k+j}$ be some other node. The node y may participate in several rotations during the deletion of x. What are the roots $z = x_{k+i}$ of the (maximal) subtrees that are moved up during those rotations? It is not hard to see that before the deletion began both y and z must have been descendants of x and after completion of the deletion y must be a descendant of z. This charaterizes the z's exactly and corresponds to the following condition on the priorities: In the index range between the minimum and the maximum of $\{k, k+i, k+j\}$ the node $x = x_k$ must have the largest priority and $z = x_{k+i}$ must have the second largest. Note that with uniform and independent random priorities this condition hold with probability $1/s^{\overline{2}} = 1/s(s+1)$ if the size of the range is $s+1$.

If $D_{i,j}$ denotes the event that $y = x_{k+j}$ participated in a down rotation of $x = x_k$ against $z = x_{k+i}$, then the expected cost of the rotations $R_f(x_k)$ incurred when x_k is deleted from an n-node tree using cost function f can be written as

$$R_f(x_k) = \sum_{\substack{-k \le j \le n-k \\ j \neq 0}} \sum_{\substack{-k \le i \le n-k \\ i \neq 0}} \Pr[D_{i,j}] \Delta f(|j|)$$

Since $\Pr[D_{i,j}] = 1/(\max\{k+i, k+j, k\} - \min\{k+i, k+j, k\})^{\overline{2}}$, the inner sum evaluates for a fixed $j > 0$ to

$$\sum_{-k \le i < 0} 1/(j-i)^{\overline{2}} + \sum_{0 < i \le j} 1/j^{\overline{2}} + \sum_{j < i \le n-k} 1/i^{\overline{2}},$$

which using $\sum_{a \le \nu < b} 1/\nu^{\overline{2}} = 1/a - 1/b$ evaluates to

$$\Big[1/(j+1) - 1/(j+k)\Big] + \Big[1/(j+1)\Big] + \Big[1/(j+1) - 1/(n-k+1)\Big] < 3/(j+1) < 3/j\,.$$

For $y = x_{k-j}$ a symmetric argument shows that the inner sum is also upper bounded by $3/j$. When f is non decreasing, i.e. Δf is non-negative, we therefore get

$$R_f(x_k) \le \sum_{1 \le j < k} \frac{3\Delta f(j)}{j} + \sum_{1 \le j < n+1-k} \frac{3\Delta f(j)}{j} < 6 \sum_{1 \le j \le n} \frac{\Delta f(j)}{j} = 6\Big(\frac{f(n)}{n} + \sum_{1 \le j < n} \frac{f(j)}{j^{\overline{2}}}\Big)$$

from which again the bounds on expensive rotations stated in Theorem 3.1 follow. Note that this method actually also allows the *exact* computation of $R_f(x_k)$ for any arbitrary real valued function f.

5.11 Changing weights

If the priority p of a single item x in a treap is changed to a new priority p', then the heap property of the treap can be reestablished by simply rotating x up or down the treap as is done during the insertion and deletion operations. The cost of this will be proportional to the number of rotations performed, which is $|D(x) - D'(x)|$, where $D(x)$ and $D'(x)$ are the depth of x in the old and new tree, respectively.

Now assume the weight w of an item x in a weighted randomized search tree of total weight W is changed to w' and after the required change in the random priorities the tree is restructured as outlined above. In the old tree x had expected depth $O(1 + \log(W/w))$, in the new tree it has expected depth $O(1 + \log(W'/w'))$, where $W' = W - w + w'$ is the total weight of the new tree.

One is now tempted to claim that the expected depth difference and hence the expected number of rotations to achieve the change is $O(|\log(W/w) - \log(W'/w')|)$, which is about $O(|\log(w'/w)|)$ if the weight change is small relative to W. There are two problems with such a quick claim: (a) in general it is not true that $\mathrm{Ex}[|X - Y|] = |\mathrm{Ex}[X] - \mathrm{Ex}[Y]|$; (b) one cannot upper bound a difference $A - B$ using only upper bounds for A and B.

For the sake of definiteness let us deal with the case of a weight increase, i.e. $w' > w$. The case of a decrease can be dealt with analogously. Let us first address problem (a). In section 3 we briefly outlined how to realize weighted randomized search trees. As priority p for an item x of weight w use $u^{1/w}$ (or equivalently $(\log u)/w$), where u is a random number uniformly distributed in $[0, 1]$. This simulates generating the maximum of w random numbers drawn independently and uniformly from $[0, 1]$. If the new weight of x is w' and one chooses as new priority $p' = v^{1/w'}$, where v is a new random number drawn from $[0, 1]$, then p' is not necessarily larger than p. This means that $D'(x)$, the new depth of x, could be larger than the old depth $D(x)$, inspite of the weight increase, which is expected to make the depth of x smaller. Thus, since the relative order of $D(x)$ and $D'(x)$ is unknown, $\mathrm{Ex}[|D(x) - D'(x)|]$ becomes difficult to evaluate.

This difficulty does not arise if one chooses as new priority $p' = u^{1/w'}$ (or equivalently $(\log u)/w'$), where u is the random number originally drawn from $[0, 1]$. Note that although the random variable p and p' are highly dependent, each has the correct distribution, and this is all that is required. Since $w' > w$ we have $u^{1/w'} > u^{1/w}$, i.e. $p' > p$. Thus we have $D(x) \geq D'(x)$, and therefore $\mathrm{Ex}[|D(x) - D'(x)|] = \mathrm{Ex}[D(x) - D'(x)] = \mathrm{Ex}[D(x)] - \mathrm{Ex}[D'(x)]$.

Addressing problem (b) pointed out above, we can bound the difference of those two expectations using the fact that we know exact expressions for each of them. Assume that x has key rank ℓ, i.e. $x = x_\ell$, and let $\Delta = w' - w$ be the weight increase. From Theorem 4.11 we get (using the notation from there)

$$\begin{aligned}
\mathrm{Ex}[D(x_\ell)] - \mathrm{Ex}[D'(x_\ell)] &= \Big(1 + \sum_{\substack{1 \leq i \leq n \\ i \neq \ell}} \frac{w_i}{w_{i:\ell}}\Big) - \Big(1 + \sum_{\substack{1 \leq i \leq n \\ i \neq \ell}} \frac{w_i}{w_{i:\ell} + \Delta}\Big) \\
&= \sum_{1 \leq i < \ell} \Big(\frac{w_i}{w_{i:\ell}} - \frac{w_i}{w_{i:\ell} + \Delta}\Big) + \sum_{\ell < i \leq n} \Big(\frac{w_i}{w_{\ell:i}} - \frac{w_i}{w_{\ell:i} + \Delta}\Big).
\end{aligned}$$

Applying the methods used in the proof of part (iv) of Theorem 4.11 it is easy to show that each of the last two sums is bounded from above by $\ln \frac{w_\ell + \Delta}{w_\ell} = \ln \frac{w'}{w}$. From this bound on the expected difference of old and new depth of x it follows that the expected time to adjust the tree after the weight of x has been increased from w to w' is $O(1 + \log(w'/w))$. Using the same methods one can show that decreasing w to w' can be dealt with in time $O(1 + \log(w/w'))$.

When implementing the method outlined here, one needs to store for every item the priority implicitly in two pieces (w, u), where integer w is the weight and u is a random number from $[0, 1]$. When two priorities (w, u) and $(\overline{w}, \overline{u})$ are to be compared one has to compare $u^{1/w}$ with $\overline{u}^{1/\overline{w}}$. Alternatively one could store the pieces $(w, \log u)$ and use $(\log u)/w$ for the explicit comparison.

This raises the issue of the cost of arithmetic. We can postulate a model of computation where the evaluation of an expression like $u^{1/w}$ or $\log u$ takes constant time and thus dealing with priorities does not become a significant overhead to the tree operations. We would like to argue that such a model is not that unrealistic. This seems clear in practice, since there one would definitely use a floating point implementation. (This is not to say that weighted trees are necessarily practical.) From the theoretical point of view, the assumption of constant time evaluation of those functions is not that unrealistic since Brent [7] has shown that, when measured in the bit model, evaluating such functions up to a relative error of 2^{-m} is only slightly more costly than multyplying two m bit numbers.

Thus we assume a *word model* where each of the four basic arithmetic operations and evaluating functions such as $\log u$ using operands specified by logarithmically many bits costs constant time. It seems natural to assume here that "logarithmically" means $O(\log W)$. We now need to deal with one issue: it is not clear that a word size of $O(\log W)$ suffices to represent our random priorities so that their relative order can be determined.

Here is a simple argument why $O(\log W)$ bits per word should suffice for our purposes. Following the definition of a weighted randomized search tree the priorities of an n node tree of total weight W can be viewed as follows: W random numbers are drawn independently and uniformly from the interval $[0, 1]$ and certain n of those chosen numbers are selected to be the priorities. Now basic arguments show that with probability at most $1/W^{k-2}$ the difference of any two of the W chosen numbers is smaller than $1/W^k$. This means that with probability at least $1 - 1/W^{k-2}$ all comparisons between priorities can be resolved if the numbers are represented using more than $k \log_2 W$ bits, i.e. a word size of $O(\log W)$ suffices with high probability.

6 Limited Randomness

The analyses of the previous sections crucially relied on the availability of perfect randomness and on the complete independence of the random priorities. In this section we briefly discuss how one can do with much less, thus proving Theorem 3.3. We will show that for unweighted trees all the asymptotic results about expectations proved in the previous sections continue to hold, if the priorities in the tree are integer random variables that are of only limited independence and are uniformly distributed over a sufficiently large range. In particular we will show that 8-wise independence and range size $U \geq n^3$ suffice. The standard example of a family of random variables satisfying these properties is $X_i = q(i) \bmod U$ for $1 \leq i \leq n$, where $U > n^3$ is a prime number and q is a degree 7 polynomial whose coefficients are drawn uniformly and independently from $\{0, \ldots, U-1\}$. Thus q acts as a pseudo random number generator that needs $O(\log n)$ truely random bits as seeds to specify its eight coefficients.

It is quite easy to see why one would want $U \geq n^3$. Ideally all priorites occuring in a randomized search tree should be distinct. Our algorithms on treaps can easily be made to handle the case of equal priorities. However for the analysis and for providing guarantees on the expectations it is preferably that all priorities be distinct. Because of the pairwise independence implied by 8-wise independence, for any two distinct random variables X_i, X_j we have $\Pr[X_i = X_j] = 1/U$. Thus the probability that any two of the n random variables happen to agree is upper bounded by $\binom{n}{2}/U$ and, as the birthday paradox illustrates, not much smaller than that. With $U \geq n^3$ the probability of some two priorities agreeing thus becomes less than $1/n$. We can now safely ignore the case of agreeing priorities since in that event we could even afford to rebuild the entire tree which would incur only $O(\log n)$ expected cost.

Why 8-wise independence? Let $\mathcal{X}$ be a finite set of random variables and let d be some integer constant. We say that $\mathcal{X}$ has the *d-max property* iff there is some constant c so that for any enumeration $X_1, X_2, \ldots X_m$ of the elements of any subset of $\mathcal{X}$ we have

$$\Pr[X_1 > X_2 > \cdots > X_d > \{X_{d+1}, X_{d+2}, \ldots, X_m\}] \leq c/m^{\underline{d}}.$$

Note that identically distributed independent continous random variables have the d-max property for any d with a constant $c = 1$.

It turns out that all our results about expected properties of randomized search trees and of their update operations can be proved by relying only on the 2-max property of the random priorities. Moreover, the 2-max property is implied by 8-wise independence because of the following remarkable lemma that is essentially due to Mulmuley [23].

Lemma 6.1 *Let $\mathcal{X}$ be a set of n random variables, each uniformly distributed over a common integer range of size at least n.*
$\mathcal{X}$ has the d-max property if its random variables are $(3d+2)$-wise independent.

A proof of this lemma (or rather, of a related version) can be found in Mulmuley's book [23, section 10.1].

We now need to show that results of section 4 about unweighted trees continue to hold up to a constant factor if the random priorities satisfy the 2-max property. This is clear for the central Corollaries 4.5 and 4.6. As a consequence of the ancestor and common ancestor lemma they essentially just give the probability that a certain one in a set of priorities achieves the maximum. For those two corollaries the 1-max property would actually suffice. From the continued validity of Corollary 4.5 the asymptotic versions of points (i) and (ii) of Theorem 4.7 about expected depth of a node and size of its subtree follow immediately, also relying only on the 1-max property. Note that this means that if one is only interested in a basic version of randomized search trees where the expected search and update times are logarithmic (although more than an expected constant number of rotations may occur per update), then 5-wise independence of the random priorities provably suffices.

Points (iii) and (iv) of Theroem 4.7 do not follow immediately. They consider expectations of random variables that were expressed in Theorem 4.1 as the sum of differences of indicator variables $(A_{i,\ell} - C_{i;\ell,m})$ and upper bounds for the expectations of ($A_{i,\ell}$ and $C_{i;\ell,m}$) yield no upper bound for the expectation of their difference. Now $(A_{i,\ell} - C_{i;\ell,m})$ really indicates the event $E_{i;\ell,m}$ that x_i is an ancestor of x_ℓ but not an ancestor of x_m. We need to show that if the priorities have the 2-max property, $\Pr[E_{i;\ell,m}]$ is essentially the same as if the priorities were completely independent.

Without loss of generality let us assume $\ell < m$. In the case $i \leq \ell$ the event $E_{i;\ell,m}$ happens exactly when the following constellation occurs among the priorities: x_i has the largest priority among the items with index between and including i and ℓ, but not the largest priority among the items with index between i and m. For $\ell < i < m$ event $E_{i;\ell,m}$ occurs iff x_i has the largest priority among the items with indices in the range between ℓ and i, but not the largest in the range ℓ to m. (For the case $i > m$ the event is empty.)

Thus in both cases we are dealing with an event $E_{X,\mathcal{Y},\mathcal{Z}}$ of the following form: For a set $\mathcal{Z}$ of random variables and $X \in \mathcal{Y} \subset \mathcal{Z}$ the random variable X is largest among the ones in $\mathcal{Y}$ but not the largest among the ones in $\mathcal{Z}$. In the case of identically distributed, independent random variables clearly we get $\Pr[E_{X,\mathcal{Y},\mathcal{Z}}] = 1/|\mathcal{Y}| - 1/|\mathcal{Z}|$. The following claim shows that essentially the same is true if $\mathcal{Z}$ has the 2-max property.

Claim 1 *If $\mathcal{Z}$ has the 2-max property, then* $\Pr[E_{X,\mathcal{Y},\mathcal{Z}}] = O(1/|\mathcal{Y}| - 1/|\mathcal{Z}|)$.

Proof. Let $a = |\mathcal{Y}|$ and $b = |\mathcal{Z}|$ and let $X_1, X_2, \ldots, X_b$ be an enumeration of $\mathcal{Z}$ so that $X_1 = X$ and $\mathcal{Y} = \{X_1, \ldots, X_a\}$. For $a < i \leq b$ let F_i denote the event that X_i is largest among $\{X_1, \ldots, X_i\}$ and X_1 is second largest. Because of the 2-max property of $\mathcal{Z}$ we have $\Pr[F_i] = O(1/i^2)$. Since $E_{X,\mathcal{Y},\mathcal{Z}} = \bigcup_{a<i\leq b} F_i$ we therefore get

$$\Pr[E_{X,\mathcal{Y},\mathcal{Z}}] \leq \sum_{a<i\leq b} \Pr[F_i] = O\Big(\sum_{a<i\leq b} 1/i^2\Big) = O(1/a - 1/b)\,,$$

as desired. □

Thus points (iii) and (iv) of Theorem 4.7 hold, up to constant factors, if priorities have the 2-max property. This immediately means that all results listed in Theorem 3.1, except for the ones

on expensive rotations, continue to hold, up to constant factors, if random priorities satisfying the 2-max property are used. The results on expensive rotations rely on the 3-max property. However, if one uses the alternate cost model as explained in section 5.10, then the 2-max property again suffices. This cost model is equivalent to the first one for all rotation cost functions of practical interest.

The only result for unweighted trees that seems to require something more than the 2-max property is the one on short excess paths for finger searches in section 5.4. This is not too serious since other methods for implementing finger searches are available. We leave it as an open problem to determine weak conditions on priorities under which excess paths remain short in expectation.

7 Implicit Priorities

In this section we show that it is possible to implement unweighted randomized search trees so that no priorities are stored explicitly. We offer three different methods. One uses hash functions to generate or regenerate priorities on demand. The other stores the nodes of the tree in a random permutation and uses node addresses as priorities. The last method recomputes priorities from subtree sizes.

7.1 Priorities from hash functions

This method is based on an initial idea of Danny Sleator [28]. He suggested to choose and associate with a randomized search tree a hash function h. For every item in the tree the priority is then declared to be $h(k)$, where k is the key of the item. This priority need not be stored since it can be recomputed whenever it is needed. The hope is, of course, that a good hash function is "random enough" so that the generated priorities behave like random numbers.

Initially it was not clear what sort of hash function would actually exhibit enough random behaviour. However, the results of the previous section show that choosing for h the randomly selected degree-7 polynomial q mentioned at the beginning of the previous section does the trick. If one is only interested in normal search and update times, then a randomly selected degree-4 polynomial suffices, as discussed in the previous section. In order to make this scheme applicable for any sort of key type we apply q not to the key but to the address of the node where the respective item is stored.

One may argue that from a practical point of view it is too expensive to evaluate a degree-7 polynomial whenever a priority needs to be looked at. Note, however, that priorities are compared only during updates, and that priority comparisons are coupled with rotations. This means that the expected number of priorities looked at during a deletion is less than 2 and during an insertion it is less than 4.

Bob Tarjan [29] pointed out to us that this method also yields a good randomized method for the so-called unique representation problem where one would like subsets of a finite universe to have unique tree representations (see e.g. [2] and [27]).

7.2 Locations as priorities

Here we store the nodes of the tree in an array $L[]$ in random order. Now the addresses of the nodes can serve as priorities, i.e. the node $L[i]$ has priority i. We will assume here that the underlying treap has the min-heap property, and not the max-heap property. Thus $L[1]$ will the root of the tree.

How does one insert into or delete from an n-node tree stored in this fashion? Basically one needs to update a random permutation. In order to insert an item x some i with $1 \leq i \leq n+1$ is chosen uniformly at random. If $i = n+1$ then x is stored in location $L[n+1]$, i.e. it is assigned priority $n+1$, and it is then inserted in the treap. If $i \leq n$ the node stored at $L[i]$ is moved to $L[n+1]$, i.e. its priority is changed to $n+1$. This means in the tree it has to be rotated down into leaf position. The new item x is placed into location $L[i]$ and it is inserted in the treap with priority i.

When item $x = L[i]$ is to be removed from the tree, it is first deleted in the usual fashion. Since this vacates location $L[i]$ the node stored at $L[n]$ is moved there. This means its priority was changed from n to i and it has to be rotated up the tree accordingly.

This scheme is relatively simple, but it does have some drawbacks. Per update some extra node changes location and has to be rotated up or down the tree. This is not a problem timewise since the expected number of those rotations is constant. However, changing the location of a node y means that one needs to access its parent so that the relevant child pointer can be reset. For accessing the parent one either needs to maintain explicit parent pointers, which is costly in space or one needs to find it via a search for y, which is costly in time. Explicit parent pointers definitely are necessary if a family of trees is to be maintained under joins and splits. One can keep the nodes of all the trees in one array. However, when an extra node is moved during an update, one does not know which tree it belongs to and hence one cannot find its parent via search. Also note the book keeping problem when the root of a tree is moved.

Finally, there is the question what size the array $L[]$ should have. This is no problem if the maximum size of the tree is known a priori. If this is not the case, one can adjust the size dynamically by, say, doubling it whenever the array becomes filled, and halving it whenever it is only 1/3 full. With a strategy of this sort the copying cost incurred through the size changes is easy to be seen constant in the amortized sense.

7.3 Computing priorities from subtree sizes

This method was suggested by Bent and Driscoll [6]. It assumes that for every node x in the tree the size $S(x)$ of its subtree is known. In a number of applications of search trees this information is stored with every node in any case.

During the deletion of a node y for every down rotation one needs to decide whether to rotate left or right. This decision is normally dictated by the priorities of the two children x and z of y: the one with larger priority is rotated up. The priority of x is the largest of the $S(x)$ priorities stored in its subtree. The priority of z is the largest of the $S(z)$ priorities in its subtree. Thus the probability that the priority of x is larger than the priority of z is $p = S(x)/(S(x) + S(z))$. This means that p should also be the probability that x is rotated up. Thus the decision which way to rotate x can be probabilistically correctly simulated by flipping a coin with bias $S(x)/(S(x)+S(z))$. It is amusing that one can actually do this without storing the sizes. Before the rotation one could determine $S(x)$ and $S(z)$ in linear time by traversing the two subtrees. This would make the cost of the rotation linear in the size of the subtree rotated, and our results about costly rotations imply that the expected deletion time is still logarithmic.

Unfortunately this trick does not work for insertions. How does one perform them? Note that when a node x is to be inserted into a a tree rooted at y it becomes the root of the new tree with probability $1/(S(y) + 1)$. This suggests the following strategy: Flip a coin with bias $1/(S(y) + 1)$. In case of success insert x into the tree by finding the correct leaf position and rotating it all the way back up to the root. In case of failure apply this strategy recursively to the appropriate child of y.

We leave the implementation of joins and splits via this method as an exercise.

8 Randomized search trees and Skip lists

Skip lists are a probabilistic search structure that were proposed and popularized by Pugh [26]. They can be viewed as a hierarchy of coarser and coarser lists constructed over an initial linked list, with a coarser list in the hierarchy guiding the search in the next finer list in the hierarchy. Which list items are used in the coarses lists is determined via random choice. Alternatively, skip lists can also be viewed as a randomized version of (a, b)-trees.

The performance characteristics of skip lists are virtually identical to the ones of unweighted randomized search trees. The bounds listed in Theorem 3.1 all hold if the notion of rotation is changed to the notion of pointer change. The only possible exception are the results about costly rotations: here, it seems, only partial results are known for skip lists (see [21],[23, section 8.1.1]).

Just as with randomized search trees it is possible to generalize skip lists to a weighted version [20]. This is done by appropriately biasing the random choice that determines how far up the hierachy a list item is to go. Apparently again the expected performance characteristics of weighted skip lists match the ones of weighted randomized search trees listed in Theorem 3.2.

No analogue of Theorem 3.3 about limited randomness for skip lists has appeared in the literature. However, Kurt Mehlhorn [19] has adapted the approach taken in this paper to apply to skip lists also. Also Martin Dietzfelbinger [11] apparently has proved results in this direction.

Comparing skip lists and randomized search trees seems a fruitless exercise. They are both conceptually reasonably simple and both are reasonably simply to implement. Both have been implemented and are for instance availabe as part of LEDA [21, 24]. They seem to be almost identical in their important performance characteristics. Differences such as randomized search trees can be implemented using only exactly n pointers, whereas this appears to be impossible for skip lists, are not of particularly great practical significance.

There seems to be ample room for both skip lists and randomized search trees. We would like to refer the reader to chapter 1 of Mulmuley's book [23], where the two structures are used and discussed as two prototypical randomization strategies.

9 Acknowledgements

We would like to thank Kurt Mehlhorn for his constructive comments and criticism.

References

[1] G.M. Adel'son-Velskii and Y.M. Landis, An algorithm for the organization of information. *Soviet Math. Dokl. 3* (1962) 1259–1262.

[2] A. Andersson and T. Ottmann, Faster uniquely represented dictionaries. *Proc. 32nd FOCS* (1991) 642–649.

[3] H. Baumgarten, H. Jung, and K. Mehlhorn, Dynamic point location in general subdivision. *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms (SODA)* (1992) 250–258.

[4] R. Bayer and E. McCreight, Organization and maintenance of large ordered indices. *Act. Inf. 1* (1972) 173–189.

[5] S.W. Bent, D.D. Sleator, and R.E. Tarjan, Biased search trees. *SIAM J. Comput. 14* (1985) 545–568.

[6] S.W. Bent and J.R. Driscoll, Randomly balanced search trees. *Manuscript* (1991).

[7] R.P. Brent, Fast Multiple Precision Evaluation of Elementary Functions. *J. of the ACM 23* (1976) 242–251.

[8] M. Brown, Addendum to "A Storage Scheme for Height-Balanced Trees." *Inf. Proc. Letters 8* (1979) 154–156.

[9] K.L. Clarkson, K. Mehlhorn, and R. Seidel, Four results on randomized incremental construction. *Comp. Geometry: Theory and Applications 3* (1993) 185–212.

[10] L. Devroye, A note on the height of binary search trees. *J. of the ACM 33* (1986) 489–498.

[11] M. Dietzfelbinger, (private communication).

[12] I. Galperin and R.L. Rivest, Scapegoat Trees. Proc. 4th ACM-SIAM Symp. on Discrete Algorithms (SODA) (1993) 165–174.

[13] L.J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees. *Proc. 19th FOCS* (1978) 8–21.

[14] T. Hagerup and C. Rüb, A guided tour of Chernoff bounds. *Inf. Proc. Letters 33* (1989/90) 305–308.

[15] K. Hoffman, K. Mehlhorn, P. Rosenstiehl, and R.E. Tarjan, Sorting Jordan sequences in linear time using level linked search trees. *Inform. and Control 68* (1986) 170–184.

[16] E. McCreight, Priority search trees. *SIAM J. Comput. 14* (1985) 257–276.

[17] K. Mehlhorn, **Sorting and Searching.** Springer (1984).

[18] K. Mehlhorn, **Multi-dimensional Searching and Computational Geometry.** Springer (1984).

[19] K. Mehlhorn, (private communication).

[20] K. Mehlhorn and S. Näher, Algorithm Design and Software Libraries: Recent Developments in the LEDA Project. *Algorithms, Software, Architectures, Information Processing 92,* Vol. 1, Elsevier Science Publishers B.V., 1992

[21] K. Mehlhorn and S. Näher, LEDA, a Platform for Combinatorial and Geometric Computing. To appear in *Commun. ACM*, January 1995.

[22] K. Mehlhorn and R. Raman (private communication).

[23] K. Mulmuley, **Computational Geometry: An Introduction through Randomized Algorithms.** Prentice Hall (1994).

[24] S. Näher, LEDA User Manual Version 3.0. Tech. Report MPI-I-93-109, Max-Planck-Institut für Informatik, Saarbrücken (1993).

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

[26] W. Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM 33* (1990) 668–676.

[27] W. Pugh and T. Teitelbaum, Incremental Computation via Function Caching. *Proc. 16th ACM POPL* (1989) 315–328.

[28] D.D. Sleator (private communication).

[29] R.E. Tarjan (private communication).

[30] D.D. Sleator and R.E. Tarjan, Self-adjusting binary search trees. *J. of the ACM 32* (1985) 652–686.

[31] J. Vuillemin, A Unifying Look at Data Structures. *Commun. ACM 23* (1980) 229–239.

EERTREE: An Efficient Data Structure for Processing Palindromes in Strings

Mikhail Rubinchik and Arseny M. Shur

Ural Federal University, Ekaterinburg, Russia
mikhail.rubinchik@gmail.com, arseny.shur@urfu.ru

Abstract. We propose a new linear-size data structure which provides a fast access to all palindromic substrings of a string or a set of strings. This structure inherits some ideas from the construction of both the suffix trie and suffix tree. Using this structure, we present simple and efficient solutions for a number of problems involving palindromes.

1 Introduction

Palindromes are one of the most important repetitive structures in strings. During the last decades they were actively studied in formal language theory, combinatorics on words and stringology. Recall that a palindrome is any string $S = a_1a_2\cdots a_n$ equal to its reversal $\overleftarrow{S} = a_n\cdots a_2a_1$.

There are a lot of papers concerning the palindromic structure of strings. The most important problems in this direction include the search and counting of palindromes in a string and the factorization of a string into palindromes. Manacher [13] came up with a linear-time algorithm which can be used to find all maximal palindromic substrings of a string, along with its palindromic prefixes and suffixes. The problem of counting and listing distinct palindromic substrings was solved offline in [6] and online in [11]. Knuth, Morris, and Pratt [10] gave a linear-time algorithm for checking whether a string is a product of even-length palindromes. Galil and Seiferas [4] asked for such an algorithm for the *k-factorization* problem: decide whether a given string can be factored into exactly k palindromes, where k is an arbitrary constant. They presented an online algorithm for $k = 1, 2$ and an offline one for $k = 3, 4$. An online algorithm working in $O(kn)$ time for the length n string and any k was designed in [12]. Close to the k-factorization problem is the problem of finding the *palindromic length* of a string, which is the minimal k in its k-factorization. This problem was solved by Fici et al. in $O(n \log n)$ time [3]. In this paper we present a new tree-like data structure, called eertree[1], which allows one to simplify and speed up solutions to search, counting and factorization problems as well as to several other palindrome-related algorithmic problems. This structure can also cope with Watson–Crick palindromes [8] and other palindromes with involution and may

[1] This structure can be found, with the reference to the first author, in a few IT blogs under the name "palindromic tree". See, e.g., http://adilet.org/blog/25-09-14/.

be interesting for the RNA studies along with the affix trees [14] and affix arrays [19].

In Sect. 2 we first recall the problem of counting distinct palindromic substrings in an online fashion. This was a motive example for inventing eertree. This data structure contains the digraph of all palindromic factors of an input string S and supports the operation $\mathsf{add}(c)$ which appends a new symbol to the end of S. Thus, the number of nodes in the digraph equals the number of distinct palindromes inside S. Maintaining an eertree for a length n string with σ distinct symbols requires $O(n \log \sigma)$ time and $O(n)$ space (for a random string, the expected space is $O(\sqrt{n\sigma})$). After introducing the eertree we discuss some of its properties and simple applications.

In Section 3 we study advanced questions related to eertrees. We consider joint eertree of several strings and name a few problems solved with its use. Then we design two "smooth" variations of the algorithm which builds eertree. These variations require at most logarithmic time for each call of $\mathsf{add}(c)$ and then allow one to support an eertree for a string with two operations: appending and deleting the last symbol. Using one of these variations, we design a fast backtracking algorithm enumerating all *rich* strings over a fixed alphabet up to a given length. (A string is rich if it contains the maximum possible number of distinct palindromes.) Finally, we show that eertree can be efficiently turned into a persistent data structure.

The use of eertrees for factorization problems is described in Sect. 4. Namely, new fast algorithms are given for the k-factorization of a string and for computing its palindromic length. We also conjecture that the palindromic length can be found in linear time and provide some argument supporting this conjecture.

Definitions and Notation. We study finite strings, viewing them as arrays of symbols: $w = w[1..n]$. The notation σ stands for the number of distinct symbols of the processed string. We write ε for the empty string, $|w|$ for the length of w, $w[i]$ for the ith letter of w and $w[i..j]$ for $w[i]w[i+1]\dots w[j]$, where $w[i..i-1] = \varepsilon$ for any i. A string u is a *substring* of w if $u = w[i..j]$ for some i and j. A substring $w[1..j]$ (resp., $w[i..n]$) is a *prefix* [resp. *suffix*] of w. If a substring (prefix, suffix) of w is a palindrome, it is called a *subpalindrome* (resp. *prefix-palindrome*, *suffix-palindrome*). A subpalindrome $w[l..r]$ has *center* $(l+r)/2$, and *radius* $\lceil (r-l+1)/2 \rceil$. Throughout the paper, we do not count ε as a palindrome.

Trie is a rooted tree with some nodes marked as terminal and all edges labeled by symbols such that no node has two outgoing edges with the same label. Each trie represents a finite set of strings, which label the paths from the root to the terminal nodes.

2 Building An Eertree

2.1 Motive problem: distinct subpalindromes online

Well known online linear-time Manacher's algorithm [13] outputs maximal radiuses of subpalindromes in a string for all possible centers, thus encoding all

subpalindromes of a string. Another interesting problem is to find and count all distinct subpalindromes. Groult et al. [6] solved this problem offline in linear time and asked for an online solution. Such a solution in $O(n \log \sigma)$ time and $O(n)$ space was given in [11], based on Manacher's algorithm and Ukkonen's suffix tree algorithm [20]. As was proved in the same paper, this solution is asymptotically optimal in the comparison-based model. But in spite of a good asymptotics, this algorithm is based on two rather "heavy" data structures. In is natural to try finding a lightweight structure for solving the analyzed problem with the same asymptotics. Such a data structure, eertree, is described below. Its further analysis revealed that it is suitable for coping with many algorithmic problems involving palindromes.

2.2 Eertree: structure, interface, construction

The basic version of eertree supports a single operation $\mathsf{add}(c)$, which appends the symbol c to the processed string (from the right), updates the data structure respectively, and returns the number of new palindromes appeared in the string. According to the next lemma, $\mathsf{add}(c)$ returns 0 or 1.

Lemma 1 ([2]). *Let S be a string and c be a symbol. The string Sc contains at most one subpalindrome which is not a substring of S. This new palindrome is the longest suffix-palindrome of Sc.*

From inside, eertree is a directed graph with some extra information. Its nodes, numbered with positive integers starting with 1, are in one-to-one correspondence with subpalindromes of the processed string. Below we denote a node and the corresponding palindrome by the same letter. We write $\mathsf{eertree}(S)$ for the state of eertree after processing the string S letter by letter, left to right.

Remark 1. To report the number of distinct subpalindromes of S, just return the maximum number of a node in $\mathsf{eertree}(S)$.

Each node v stores the length $\mathsf{len}[v]$ of its palindrome. For the initialization purpose, two special nodes are added: with the number 0 and length 0 for the empty string, and with the number -1 and length -1 for the "imaginary string".

The edges of the graph are defined as follows. If c is a symbol, v and cvc are two nodes, then an edge labeled by c goes from v to cvc. The edge labeled by c goes from the node 0 (resp. -1) to the node labeled by cc (resp., by c) if it exists. This explains why we need two initial nodes. The outgoing edges of a node v are stored in a dictionary which, given a symbol c, returns the edge $\mathsf{to}[v][c]$ labeled by it. Such a dictionary is implemented as a binary balanced search tree.

An unlabeled *suffix link* $\mathsf{link}[u]$ goes from u to v if v is the longest proper suffix-palindrome of u. By definition, $\mathsf{link}[c] = 0$, $\mathsf{link}[0] = \mathsf{link}[-1] = -1$. The resulting graph, consisting of nodes, edges, and suffix links, is the eertree; see Fig. 1 for an example.

Lemma 2. *A node of positive length in an eertree has exactly one incoming edge.*

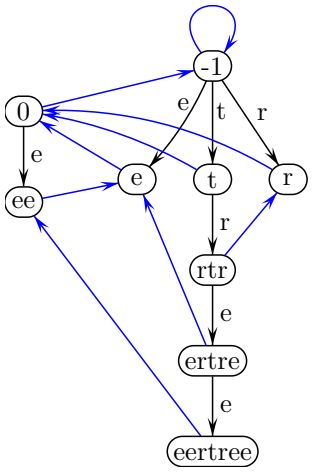


Fig. 1. The eertree of the string *eertree*. Edges are black, suffix links are blue.

Proof. An edge leading to a node u is labeled by $c = u[1]$. Then its origin must be the node v such that $u = cvc$ or the node -1 if $u = c$. □

Proposition 1. *The eertree of a string S of length n is of size $O(n)$.*

Proof. The eertree of S has at most $n+2$ nodes, including the special ones (by Lemma 1), at most n edges (by Lemma 2), and at most $n+2$ suffix links (one per node). □

Proposition 2. *For a string S of length n, $\mathsf{eertree}(S)$ can be built online in $O(n \log \sigma)$ time.*

Proof. We start defining $\mathsf{eertree}(\varepsilon)$ as the graph with two nodes (0 and -1) and two suffix links. Then we make the calls $\mathsf{add}(S[1]), \ldots, \mathsf{add}(S[n])$ in this order. By Lemma 1 and the definition of add, after each call we know the longest suffix-palindrome $\mathsf{maxSuf}(T)$ of the string T processed so far. We support the following invariant: after a call to add, all edges and suffix links between the existing nodes are defined. In this case, adding a new node u one must build exactly one edge (by Lemma 2) and one suffix link: any suffix-palindrome of u is its prefix as well, and hence the destination node of the suffix link from u already exists.

Consider the situation after i calls. We have to perform the next call, say $\mathsf{add}(a)$, to $T = S[1..i]$. We need to find the maximum suffix-palindrome P of Ta. Clearly, $P = a$ or $P = aQa$, where Q is a suffix-palindrome of T. Thus, to determine P we should find the longest suffix-palindrome of T preceded by a. To do this, we traverse the suffix-palindromes of T in the order of decreasing length, starting with $\mathsf{maxSuf}(T)$ and following suffix links. For each palindrome we read its length k and compare $T[i-k]$ against a until we get an equality or arrive at the node -1. In the former case, the current palindrome is Q; we check whether it has an outgoing edge labeled by a. If yes, the edge leads to $aQa = P$, and P is not new; if no, we create the node P of length $|Q| + 2$ and the edge

(Q, P). In the latter case, $P = a$; as above, we check the existence of P in the graph from the current node (which is now -1) and create the node if necessary, together with the edge $(-1, P)$ and the suffix link $(P, 0)$.

It remains to create the suffix link from P if $|P| > 1$. It leads to the second longest suffix-palindrome of Ta. This palindrome can be found similar to P: just continue traversing suffix-palindromes of T starting with the suffix link of Q.

Now estimate the time complexity. During a call to $\mathsf{add}(a)$, one checks the existence of the edge from Q with the label a in the dictionary, spending $O(\log \sigma)$ time. The path from the old to the new value of maxSuf requires one transition by an edge (from Q to P) and $k \geq 0$ of transitions by suffix links, and is accompanied by $k+1$ comparisons of symbols. In order to estimate k, follow the position of the first symbol of maxSuf: a transition by a suffix link moves it to the right, and a transition by an edge moves it one symbol to the left. During the whole process of construction of $\mathsf{eertree}(S)$, this symbol moves to the right by $\leq n$ symbols. Hence, the total number of transitions by suffix links is $\leq 2n$. The same argument works for the second longest suffix-palindrome, which was used to create suffix links. Thus, the total number of graph transitions and symbol comparisons is $O(n)$, and the time complexity is dominated by checking the existence of edges, $O(n \log \sigma)$ time in total. □

2.3 Some properties of eertrees

We call a node *odd* (resp., even) if it corresponds to an odd-length (resp., even-length) palindrome. By *suffix path* we mean a path consisting of suffix links.

Lemma 3. 1) *Nodes and edges of an eertree form two weakly connected components: the tree of odd (resp., of even) nodes rooted at* -1 *(resp., at* 0*).*
2) *The tree of even (resp., odd) nodes is precisely the trie of right halves of even-length palindromes (resp., the trie of right halves, including the central symbol, of odd-length palindromes).*
3) *Nodes and inverted suffix links of an eertree form a tree with a loop at its root* -1.

Proof. 1) If an edge (u, v) exists, then $|v| = |u|+2$. Hence, the edges of an eertree constitute no cycles, odd nodes are unreachable from even ones, and vice versa. Further, Lemma 2 implies that each even (resp., odd) node can be reached by a unique path from 0 (resp., -1). So we have the two required trees.

2) This is immediate from the definitions of a trie and an edge.

3) The suffix link decreases the length of a node, except for the node -1. So the only cycle of suffix links is the loop at -1. Each node has a unique suffix link and is connected by a suffix path to the node -1. So the considered graph is a tree (with a loop on the root) by definition.

Remark 2. Tries are convenient data structures, but a trie built from the set of all suffixes (or all factors) of a length n string is usually of size $\Omega(n^2)$. For a linear-space implementation, such a trie should be compressed into a more complicated

and less handy structure: suffix tree or suffix automaton (DAWG). On the other hand, eertrees are linear-size tries and do not need any compression. Moreover, the size of an eertree is usually much smaller than n, because the expected number of distinct palindromes in a length n string is $O(\sqrt{n\sigma})$ [17]. This fact explains high efficiency of eertrees in solving different problems.

Remark 3. A *θ-palindrome* is a string $S = a_1 \cdots a_n$ equal to $\theta(a_n \cdots a_1)$, where θ is a symbol-to-symbol function and θ^2 is the identity (see, e.g., [8]). Clearly, an eertree containing all θ-palindromes of a string can be built in the way described in Proposition 2 (the comparisons of symbols should take θ into account).

2.4 First applications

We demonstrate the performance of eertrees on two test problems taken from student programming contests. The first problem is Palindromic Refrain [21, Problem A], stated as follows: for a given string S find a subpalindrome P maximizing the value $|P| \cdot \mathsf{occ}(S, P)$, where $\mathsf{occ}(S, P)$ is the number of occurrences of P in S. The solution to this problem, suggested by the jury of the contest, included a suffix data structure and Manacher's algorithm.

Proposition 3. Palindromic Refrain *can be solved by an eertree with the use of $O(n)$ additional time and space.*

Proof. In order to find $\mathsf{occ}[v]$ for each node of $\mathsf{eertree}(S)$, we store an auxiliary parameter $\mathsf{occAsMax}[v]$, which is the number of i's such that $\mathsf{maxSuf}(S[1..i]) = v$. This parameter is easy to compute online: after a call to add, we increment occAsMax for the current maxSuf. After building $\mathsf{eertree}(S)$, we compute the values of occ as follows:

$$\mathsf{occ}[v] = \mathsf{occAsMax}[v] + \sum_{u:\mathsf{link}[u]=v} \mathsf{occ}[u]\,. \tag{1}$$

Indeed, if v is a suffix of $S[1..i]$ for some i, then either $v = \mathsf{maxSuf}(S[1..i])$ and this occurrence is counted in $\mathsf{occAsMax}[v]$, or $v = \mathsf{maxSuf}(u)$ for some suffix-palindrome u of $S[1..i]$; in the latter case, $\mathsf{link}[u] = v$, and this occurrence of v is counted in $\mathsf{occ}[u]$. To compute the values of occ in the order prescribed by (1), one can traverse the tree of suffix links bottom-up:

```
for (v = size; v ≥ 1; v--)
        occ[v] = occAsMax[v]
for (v = size; v ≥ 1; v--)
        occ[ link[v] ] += occ[v]
```

Here size is the maximum number of a node in $\mathsf{eertree}(S)$. Note that the node $\mathsf{link}[v]$ always has the number less than v, because $\mathsf{link}[v]$ exists at the moment of creation of v. After computing occ for all nodes, $P = \mathsf{argmax}(\mathsf{occ}[v] \cdot \mathsf{len}[v])$. □

The second problem is Palindromic Pairs [22, Problem B]: for a string S, find the number of triples i, j, k such that $1 \le i \le j < k \le |S|$ and the strings $S[i..j]$, $S[j+1..k]$ are palindromes.

Proposition 4. Palindromic Pairs *can be solved by an eertree with the use of* $O(n \log \sigma)$ *additional time and* $O(n)$ *space.*

Proof. Let $\mathsf{maxSuf}[j] = \mathsf{maxSuf}(S[1..j])$ and $\mathsf{sufCount}[v]$ be the number of suffix-palindromes of the subpalindrome v of S, including v itself. Note that $\mathsf{sufCount}[v] = 1 + \mathsf{sufCount}[\mathsf{link}[v]]$. Hence, $\mathsf{sufCount}[v]$ can be stored in the node v of $\mathsf{eertree}(S)$ and computed when this node is created. In addition, we memorize the values $\mathsf{maxSuf}[1], \ldots, \mathsf{maxSuf}[n]$ in a separate array. The number of palindromes ending in position j of S is the number of suffix-palindromes of $S[1..j]$ or of $\mathsf{maxSuf}(S[1..j])$. So this number equals $\mathsf{sufCount}[\mathsf{maxSuf}[j]]$.

Further, let $\mathsf{prefCount}[v]$ be the number of prefix-palindromes of v and $\mathsf{maxPref}[j]$ be the longest prefix-palindrome of $S[j..n]$. The values of $\mathsf{prefCount}$ and $\mathsf{maxPref}$ can be found when building $\mathsf{eertree}(\overleftarrow{S})$[2]. Similar to the above, the number of palindromes beginning in position j of S is $\mathsf{prefCount}[\mathsf{maxPref}[j]]$. Note that all additional computations take $O(1)$ time for each call of add, except for the second eertree, which requires $O(n \log \sigma)$ time.

For a fixed j, the number of triples (i, j, k) defining a palindromic pair is the number of palindromes ending at position i times the number of palindromes beginning at position $j+1$. Hence, the answer to the problem is

$$\sum_{j=1}^{n-1} \mathsf{sufCount}[\mathsf{maxSuf}[j]] \cdot \mathsf{prefCount}[\mathsf{maxPref}[j+1]].$$

Since this is also a linear-time computation, we are done with the proof. □

3 Advanced Modifications of Eertrees

3.1 Joint eertree for several strings

When a problem assumes the comparison of two or more strings, it may be useful to build a joint data structure. For example, a variety of problems can be solved by joint ("generalized") suffix trees, see [7]. Here we introduce the *joint eertree* of a set of strings and name several problems it can solve.

A joint eertree $\mathsf{eertree}(S_1, \ldots, S_k)$ is built as follows. We build $\mathsf{eertree}(S_1)$ in a usual fashion; then reset the value of maxSuf to 0 and proceed with the string S_2, addressing the add calls to the currently built graph; and so on, until all strings are processed. Each created node stores an additional k-element boolean array flag. After each call to add, we update flag for the current maxSuf node, setting

[2] The strings S and $\overleftarrow{S}$ have exactly the same subpalindromes, so there is no need to build the second eertree. We just perform calls to add on $\mathsf{eertree}(S)$ and fill $\mathsf{prefCount}$ and $\mathsf{maxPref}$.

its ith bit to 1, where S_i is the string being processed. As a result, $\mathsf{flag}[v][i]$ equals 1 if and only if v is contained in S_i.

Some problems easily solved by a joint eertree are gathered below.

Problem	Solution
Find the number of subpalindromes, common to all k given strings.	Build $\mathsf{eertree}(S_1, \ldots, S_k)$ and count the nodes having only 1's in the flag array.
Find the longest subpalindrome contained in all k given strings.	Build $\mathsf{eertree}(S_1, \ldots, S_k)$. Among the nodes having only 1's in the flag array, find the node of biggest length.
For strings S and T find the number of palindromes P having more occurrences in S than in T.	Build $\mathsf{eertree}(S, T)$, computing occ_S and occ_T in its nodes (see Palindromic Refrain in Sect. 2.4). Return the number of nodes v such that $\mathsf{occ}_S[v] > \mathsf{occ}_T[v]$.
For strings S and T find the number of equal palindromes, i.e., of triples (i, j, k) such that $S[i..i+k] = T[j..j+k]$ is a palindrome.	Build $\mathsf{eertree}(S, T)$, computing the values occ_S and occ_T in its nodes. The answer is $\sum_v \mathsf{occ}_S[v] \cdot \mathsf{occ}_T[v]$.

3.2 Coping with deletions

In the proof of Proposition 2, an $O(n \log \sigma)$ algorithm for building an eertree is given. Nevertheless, in some cases one call of add requires $\Omega(n)$ time, and this kills some possible applications. For example, we may want to support an eertree for a string which can be changed in two ways: by appending a symbol on the right ($\mathsf{add}(c)$) and by deleting the last symbol ($\mathsf{pop}()$). Consider the following sequence of calls:

$$\underbrace{\mathsf{add}(a), \ldots, \mathsf{add}(a)}_{n/3 \text{ times}}, \underbrace{\mathsf{add}(b), \mathsf{pop}(), \mathsf{add}(b), \mathsf{pop}(), \ldots, \mathsf{add}(b), \mathsf{pop}()}_{n/3 \text{ times}}$$

Since each appending of b requires $n/3$ suffix link transitions, the algorithm from Proposition 2 will process this sequence in $\Omega(n^2)$ time independent of the implementation of the operation $\mathsf{pop}()$.

Below we describe two algorithms which build eertrees in a way that provides an efficient solution to the problem with deletions.

Searching suffix-palindromes with quick links. Consider a pair of nodes $v, \mathsf{link}[v]$ in an eertree and the symbol $b = v[|v| - |\mathsf{link}[v]|]$ preceding the suffix $\mathsf{link}[v]$ in v. In addition to the suffix link, we define the *quick link*: let $\mathsf{quickLink}[v]$ be the longest suffix-palindrome of v preceded in v by a symbol different from b.

Lemma 4. *As a node v is created, the link $\mathsf{quickLink}[v]$ can be computed in $O(1)$ time.*

Proof. The two longest suffix-palindromes of v are $u = \mathsf{link}[v]$ and $u' = \mathsf{link}[\mathsf{link}[v]]$. Assume that v has suffixes bu and cu'. If $c \neq b$, then $\mathsf{quickLink}[v] = u'$

by definition. If $c = b$, then clearly $\mathsf{quickLink}[v] = \mathsf{quickLink}[u]$. Thus we need a constant number of operations. The code computing the quick link of v is given below. □

```
if ( S[n - len[link[v]]] == S[n - len[link[link[v]]]] )
        quickLink[v] = quickLink[link[v]]
else
        quickLink[v] = link[link[v]]
```

Recall that appending a letter c to a current string S, we scan suffix-palindromes of S to find the longest suffix-palindrome Q preceded by c; then $\mathsf{maxSuf}(Sc) = cQc$. (If cQc is a new palindrome, then this scan continues until $\mathsf{link}[cQc]$ is found.) The use of quick links reduces the number of scanned suffix-palindromes as follows. When the current palindrome is v, we check both v and $\mathsf{link}[v]$. If both are not preceded by c, then all suffix-palindromes of S longer than $\mathsf{quickLink}[v]$ are not preceded by c either; so we skip them and check $\mathsf{quickLink}[v]$ next.

Example 1. Let us call $\mathsf{add}(b)$ to the eertree of the string $S = aabaabaaba$. The longest suffix-palindrome of S is the string $v = abaabaaba$. Since the symbols preceding v and $\mathsf{link}[v] = abaaba$ in S are distinct from b, we jump to $\mathsf{quickLink}[v] = a$, skipping the suffix-palindrome aba preceded by the same letter as $\mathsf{link}[v]$. Now $\mathsf{quickLink}[v]$ is preceded by b, so we find $\mathsf{maxSuf}(Sb) = bab$. Note that v "does not know" which symbol precedes its particular occurrence, and different occurrences can be preceded by different symbols. So there is no way to avoid checking the symbol preceding $\mathsf{link}[v]$.

Constructing an eertree with quick links, on each step we add $O(1)$ time and space for maintaining these links and possibly reduce the number of processed suffix-palindromes. So the overall time and space bounds from Proposition 2 are in effect. Let us estimate the number of operations per step. The statements on "series" of palindromes, analogous to the next proposition, were proved in several papers (see, e.g., [12, Lemmas 5,6] and [3, Lemma 5]).

Proposition 5. *In an eertree, a path consisting of quick links has length* $O(\log n)$.

Corollary 1. *The algorithm constructing an eertree using quick links spends* $O(\log n)$ *time and* $O(1)$ *space for any call to* add.

Using direct links. Now we describe the fastest algorithm for constructing an eertree which, however, uses more than $O(1)$ space for creating a node. Still, the space requirements are quite modest, so the algorithm is highly competitive:

Proposition 6. *There is an algorithm which constructs an eertree spending* $O(\log \sigma)$ *time and* $O(\min(\log \sigma, \log\log n))$ *space for any call to* add.

Proof. For each node we create σ *direct links*: $\mathsf{directLink}[v][c]$ is the longest suffix-palindrome of v preceded in v by c.

Let Q be the longest suffix-palindrome of a string S, preceded by c in S. Then either $Q = \mathsf{maxSuf}(S)$ or $Q = \mathsf{directLink}[\mathsf{maxSuf}(S)][c]$, and the longest suffix-palindrome of Q, preceded by c, is $\mathsf{directLink}[Q][c]$. Thus, we scan suffixes in constant time, and the time per step is now dominated by $O(\log \sigma)$ for searching an edge in the dictionary plus the time for creating direct links for a new node.

Note that the arrays $\mathsf{directLink}[v]$ and $\mathsf{directLink}[\mathsf{link}[v]]$ coincide for all symbols except for the symbol c preceding $\mathsf{link}[v]$ in v. Hence, creating a node v we first find $\mathsf{link}[v]$, then copy $\mathsf{directLink}[\mathsf{link}[v]]$ to $\mathsf{directLink}[v]$ and assign $\mathsf{directLink}[v][c] = \mathsf{link}[v]$. However, storing or copying direct links explicitly would cost a lot of space and time. So we do this implicitly, using fully persistent balanced binary search tree (*persistent tree* for short; see[1]). We will not fall into details of the internal of the persistent tree, taking it as a blackbox. The persistent tree provides full access to any of m its *versions*, which are balanced binary search trees. The versions are ordered by the time of their creation. An update of any version results in creating a new $(m+1)$th version, which is also fully accessible; the updated version remains unchanged. Such an update as adding a node or changing the information in a node takes $O(\log k)$ time and space, where k is the size of the updated version.

We store direct links from all nodes of the eertree in a single persistent tree. Each version corresponds to a node. Direct links $\mathsf{directLink}[v][c]$ in a version v are stored as a search tree, with the letter c serving as the key for sorting (we assume an ordered alphabet). Creation of a node v requires an update of the version corresponding to the node $\mathsf{link}[v]$. It remains to estimate the size of a single search tree. It is at most σ by definition, and it is $O(\log n)$ by Proposition 5. Thus, the update time and space is $O(\min(\log \sigma, \log \log n))$, as required. □

Comparing different implementations. The three methods of building an eertree are gathered in the following table.

Method	Time for n calls	Time for one call	Space for one node
basic	$\Theta(n \log \sigma)$	$\Omega(\log \sigma)$ but $O(n)$	$\Theta(1)$
quickLink	$\Theta(n \log \sigma)$	$\Omega(\log \sigma)$ but $O(\log n)$	$\Theta(1)$
directLink	$\Theta(n \log \sigma)$	$\Theta(\log \sigma)$	$O(\min(\log \sigma, \log \log n))$

The basic version is the simplest one and uses the smallest amount of memory. Quick and direct links work somewhat faster, but their main advantage is that any single call is cheap, and thus can be reversed without much pain. Hence, one can easily maintain an eertree for a string with both operations $\mathsf{add}(c)$ and $\mathsf{pop}()$. Indeed, let $\mathsf{add}(c)$ push to a stack the node containing $P = \mathsf{maxSuf}(Sc)$ and, if P is a new palindrome, the node containing Q such that $P = cQc$. This takes $O(1)$ additional time and space. Then $\mathsf{pop}()$ reads this information from the stack and restores the previous state of the eertree in constant time.

The table above also suggests the question whether some further optimization of the obtained algorithms is possible.

Question 1. Is there an online algorithm which builds an eertree spending $O(\log \sigma)$ time and $O(1)$ space for any call to add?

3.3 Enumerating rich strings

By Lemma 1, the number of distinct subpalindromes in a length n string is at most n. Such strings with exactly n palindromes are called *rich*. Rich strings possess a number of interesting properties; see, e.g., [2,5]. The sequence A216264 in the Online Encyclopedia of Integer Sequences [18] is the growth function of the language of binary rich strings, i.e., the nth term of this sequence is the number of binary rich strings of length n. J. Shallit computed this function up to $n = 25$, thus enumerating several millions of rich strings. Using the results of Sect. 3.2, we were able to raise the upper bound to $n = 60$, enumerating several *trillions* of rich strings in 10 hours on an average laptop. The new numerical data shows that this sequence grows much slower than it was expected before.

Proposition 7 below serves as the theoretic basis for such a breakthrough in computation. It is based on the following obvious corollary of Lemma 1.

Lemma 5. *Any prefix of a rich string is rich.*

Proposition 7. *Suppose that R is the number of k-ary rich strings of length $\le n$, for some fixed k and n. Then the trie built from all these strings can be traversed in time $O(R)$.*

Proof. For simplicity, we give the proof for the binary alphabet. The extension to an arbitrary fixed alphabet is straightforward. Consider the following code, using an eertree on a string with deletions.

```
void calcRichString(i)
       ans[i]++
       if (i < n)
              if (add('0') )
                     calcRichString(i + 1)
              pop()
              if (add('1') )
                     calcRichString(i + 1)
              pop()
```

Here i is the length of the currently processed rich string. Recall that add(c) appends c to the current eertree and returns the number of new palindromes, which is 0 or 1. Hence the modified string is rich if and only if add returns 1. Note that any added symbol will be deleted back with pop(). So we exit every particular call to calcRichString with the same string as the one we entered this call. As a result, the call calcRichString(0) traverses depth-first the trie of all binary rich strings of length $\le n$.

As was mentioned in Sect. 3.2, the pop operation works in constant time. For add we use the method with direct links. Since the alphabet is constant-size, the

array directLink$[v]$ can be copied in $O(1)$ time. Hence, add also works in $O(1)$ time. The number of pop's equals the number of add's, and the latter is twice the number of rich strings of length $< n$. The number of other operations is constant per call of calcRichString, so we have the total $O(R)$ time bound. □

Remark 4. Visit `http://pastebin.com/4YJxVzep` for an implementation of the above algorithm. In 10 hours, it computed the first 58 terms of the sequence A216264. To increase the number of terms to 60, we used a few optimization tricks which reduce the constant in the O-term. We do not discuss these tricks here, because they make the code less readable.

3.4 Persistent eertrees

In Sect. 3.2 we build an eertree supporting deletions from a string. A natural generalization of this approach leads to *persistent* eertrees. Recall that a persistent data structure is a set of "usual" data structures of the same type, called *versions* and ordered by the time of their creation. A call to a persistent structure asks for the access or update of any specific version. Existing versions are neither modified nor deleted; any update creates a new (latest) version.

Consider a *tree of versions* $\mathcal{T}$ whose nodes, apart from the root, are labeled by symbols. The tree represents the set of versions of some string S: each node v represents the string read from the root to v. Recall that we denote a node of a data structure by the same letter as the string related to it. Note that some versions can be identical except for the time of their creation (i.e., for the number of a node). The problem we study is maintaining an eertree for each version of S. More precisely, the function addVersion(v, c) to be implemented adds a new child u labeled by c to the node v of $\mathcal{T}$ and computes eertree(u). The data structure which performs the calls to addVersion, supporting the eertrees for all nodes of $\mathcal{T}$, will be called a *persistent eertree*. Surprisingly enough, this complicated structure can be implemented efficiently in spite of the fact that the current string cannot be addressed directly for symbol comparisons.

Proposition 8. *The persistent eertree can be implemented to perform each call to* addVersion(v, c) *in* $O(\log |v|)$ *time and space.*

Proof. We use the method with direct links and build, as in Sect. 3.1, a joint eertree for all versions. Each node of the tree $\mathcal{T}$ stores links to the palindromes of the corresponding version of S. Overall, the node v of $\mathcal{T}$ contains the following information: a binary search tree searchTree$[v]$, containing links to all subpalindromes of v; link maxSuf$[v]$ to the maximal suffix-palindrome of v; array pred$[v]$, whose ith element is the link to the predecessor z of v such that the distance between z and v in $\mathcal{T}$ is 2^i $(i \geq 0)$; and the symbol symb$[v]$ added to the parent of v to get v. All listed parameters except for searchTree$[v]$ use $O(\log |v|)$ space. For search trees we use, as in Sect. 3.2, the persistent tree [1], reducing both time and space for copying the tree and inserting one element to $O(\log |v|)$. (Recall that another persistent tree is used inside the eertree for storing direct links of all nodes.)

Now we implement addVersion(v, c) in time $O(\log |v|)$. Note that for any i the symbol $v[i]$ can be found in $O(\log |v|)$ time. Indeed, this symbol is symb$[z]$, where z is the predecessor of v such that the distance between z and v is $h = |v| - i$. Using the binary representation of h, we can reach z from v in at most $\log |v|$ steps following the appropriate pred links.

Let V be the current number of versions (at any time). Creating a new version u with the parent v, we increment V by one and compute all parameters for u. First we compute pred$[u]$. This can be done in $O(\log |v|)$ time because pred$[u][0] = v$ and pred$[u][i] =$ pred[pred$[u][i-1]][i-1]$ for $i > 0$.

To compute the palindrome $y =$ maxSuf$[u]$, we call add(c) for the string v. Let x be the parent of y in the eertree. Then $x =$ maxSuf$[v]$ if maxSuf$[v]$ is preceded by c in v and $x =$ directLink[maxSuf$[v]][c]$ otherwise. Hence, to compute y we access exactly one symbol of v. Further, if y is not in the eertree, a new node of the eertree should be created for y. It is easy to see that link$[y] =$ to[directLink$[x][c]][c]$. Next, directLink$[u]$ is copied from directLink[link$[u]]$, with one element replaced by link$[u]$. To find this element, we need to know the letter of v preceding x. Therefore, to find maxSuf$[u]$ and modify eertree if necessary, we need $O(\log |v|)$ time for accessing a constant number of symbols in v and $O(\log \sigma)$ time for the rest of computation in add(c). Finally, we create a version of the search tree for u, updating the version for v with y (if y is in the search tree for v, this tree is copied to the new version without changes). This operation takes $O(\log |v|)$ as well. The proposition is proved. The code for addVersion(v, c) is given below. □

```
void getpred(v, par)
       pred[v][0] = par
       i = 1
       while (pred[v][i] > 0)
              pred[v][i + 1] = pred[ pred[v][i] ][i]
              i++
int addVersion(v, c)
       t++ // the number of versions, initialized by 0
       u = t
       symb[u] = c
       pred[u] = getpred(u, v)
       if (c == v[len[v] - len[maxSuf[v]]])
              x = maxSuf[v]
       else
              x = directLink[maxSuf[v]][c]
       maxSuf[u] = to[x][c] //created if does not exist
       searchTree[u] = insert(searchTree[v], maxSuf[u])
       return u
```

4 Factorizations into Palindromes

As was mentioned in the introduction, the k-factorization problem can be solved online in $O(kn)$ time for the length n string and any k [12]. In this section we are aimed at solving this problem in time independent of k. This setting is motivated by the fact that the expected palindromic length of a random string is $\Omega(n)$ [16], and the $O(kn)$ asymptotics is quite bad for such big values of k. On the positive side, the palindromic length of a string S, which is the minimum k such that a k-factorization of S exists, can be found in $O(n \log n)$ time [3].

4.1 Palindromic length vs k-factorization

Lemma 6. *Given a k-factorization of a length n string S, it is possible, in $O(n)$ time, to factor S into $k+2t$ palindromes for any positive integer t such that $k+2t \le n$.*

Proof. Let $P_1, \dots, P_k$ be palindromes, $S = P_1 \cdots P_k$, $k \le n-2$. It is sufficient to show how to factor S into $k+2$ palindromes. If $|P_i| \ge 3$ for some i, then we split P_i into three palindromes: the first letter, the last letter, and the remaining part. Otherwise, there are some P_i, P_j of length 2, each of which can be split into two palindromes. □

Thus, k-factorization problem is reduced in linear time to two similar problems: factor a string into the minimum possible odd (resp. even) number of palindromes. We solve these two problems using an eertree. To do this, we first describe an algorithm, based on an eertree and finding the palindromic length in time $O(n \log n)$. While its asymptotics is the same as of the algorithm of [3], its constant under the O-term is much smaller (see Remark 6 below) and its code is simpler and shorter.

Proposition 9. *Using an eertree, the palindromic length of a length n string can be found online in time $O(n \log n)$.*

Proof. For a length n string S we compute online the array ans such that $\mathsf{ans}[i]$ is the palindromic length of $S[1..i]$. Note that any k-factorization of $S[1..i]$ can be obtained by appending a suffix-palindrome $S[j+1..i]$ of $S[1..i]$ to a $(k-1)$-factorization of $S[1..j]$. Thus, $\mathsf{ans}[i] = 1+\min\{\mathsf{ans}[j] \mid S[j+1..i] \text{ is a palindrome}\}$.

To compute ans efficiently, we store two additional parameters in the nodes of the eertree: *difference* $\mathsf{diff}[v] = \mathsf{len}[v] - \mathsf{len}[\mathsf{link}[v]]$ and *series link* $\mathsf{seriesLink}[v]$, which is the longest suffix-palindrome of v having the difference unequal to $\mathsf{diff}[v]$. Series links are similar to quick links, which are not suitable for the problem studied. Clearly, the difference is computable in $O(1)$ time and space on the creation of a node; the following code shows that the same is true for the series link.

```
if (diff[v] == diff[link[v]])
      seriesLink[v] = seriesLink[link[v]]
```

```
else
        seriesLink[v] = link[v]
```

The following "naive" implementation computes $\mathsf{ans}[n]$ in $O(n)$ time.

```
ans[n] = ∞
for (v = maxSuf; len[v] > 0; v = link[v])
        ans[n] = min(ans[n], ans[n - len[v]] + 1)
```

With series links, the same idea can be rewritten as follows:

```
int getMin(u)
        res = ∞
        for (v = u; len[v] > len[seriesLink[u]]; v = link[v])
                res = min(res, ans[n - len[v]] + 1)
        return res
ans[n] = ∞
for (v = maxSuf; len[v] > 0; v = seriesLink[v])
        ans[n] = min(ans[n], getMin(v))
```

The getMin function has linear-time worst-case complexity, and we are going to speed it up to a constant time. By the *series* of a palindrome u we mean the sequence of nodes in the suffix path of u from u (inclusive) to $\mathsf{seriesLink}[u]$ (exclusive). Note that $\mathsf{getMin}[u]$ loops through the series of u. Comparing $\mathsf{diff}[u]$ and $\mathsf{diff}[\mathsf{link}[u]]$, we can check whether the series of u contains just one palindrome. If this is the case, then $\mathsf{res} = \mathsf{ans}[n - \mathsf{len}[u]] + 1$ can be computed in $O(1)$ time. Hence, below we are interested in series of at least two elements. A suffix-palindrome u of S is called *leading* if either $u = \mathsf{maxSuf}(S)$ or $u = \mathsf{seriesLink}[v]$ for some suffix-palindrome v of S. We need four auxiliary lemmas.

Lemma 7. *If a palindrome v of length $l \geq n/2$ is both a prefix and a suffix of a string $S[1..n]$, then S is a palindrome.*

Proof. Let $i \leq n/2$. Then $S[i] = v[i] = v[l - i + 1] = S[n - i + 1]$, i.e., S is a palindrome by definition. □

Lemma 8. *Suppose v is a leading suffix-palindrome of a string $S[1..n]$ and $u = \mathsf{link}[v]$ belongs to the series of v. Then u occurs in v exactly two times: as a suffix and as a prefix.*

Proof. Let $i = n - |v| + 1$. Then $v = S[i..n], u = S[i{+}\mathsf{diff}[v]..n] = S[i..n{-}\mathsf{diff}[v]]$. Since $\mathsf{diff}[u] = \mathsf{diff}[v]$, we have $\mathsf{diff}[v] \leq |v|/2$, so that the two mentioned occurrences of u touch or overlap. If there exist k, t such that $i < k < i{+}\mathsf{diff}[v]$ and $S[k..t] = u$, then $S[k..n]$ is a palindrome by Lemma 7. This palindrome is a proper suffix of v and is longer than $\mathsf{link}[v]$, which is impossible. □

Lemma 9. *Suppose v is a leading suffix-palindrome of a string $S[1..n]$ and $u = \mathsf{link}[v]$ belongs to the series of v. Then u is a leading suffix-palindrome of $S[1..n{-}\mathsf{diff}[v]]$.*

Proof. If u is not leading, then the string $S[1..n-\mathsf{diff}[v]]$ has a suffix-palindrome $z = S[j..n-\mathsf{diff}[v]]$ with $\mathsf{link}[z] = u$ and $\mathsf{diff}[z] = \mathsf{diff}[u]$. Since u is both a prefix and a suffix of z and $|z| = |v| \le 2|u|$, clearly $z = v$. Then $w = S[j..n]$ is a palindrome by Lemma 7. Assume that w has a suffix-palindrome v' which is longer than v. Then v' begins with u, and this occurrence of u is neither prefix nor suffix of $z = S[j..n-\mathsf{diff}[v]]$, contradicting Lemma 8. Therefore, $v = \mathsf{link}[w]$ and $\mathsf{diff}[w] = \mathsf{diff}[v]$, which is impossible because v is leading. This contradiction proves that u is leading. □

Lemma 10. *In an eertree, a path consisting of series links has length $O(\log n)$.*

Proof. Follows from [12, Lemma 6], since any leading suffix-palindrome is also leading in terms of [12].

By Lemma 10, the function $\mathsf{ans}(n)$ calls getMin $O(\log n)$ times. Now consider an $O(1)$ time implementation of getMin. Recall that it is enough to analyze non-trivial series of palindromes; they look like in Fig. 2. The first positions of all palindromes in the depicted series of v and $\mathsf{link}[v]$ match (because $\mathsf{diff}[v] = \mathsf{diff}[\mathsf{link}[v]]$) except for the last palindrome in the series of v.

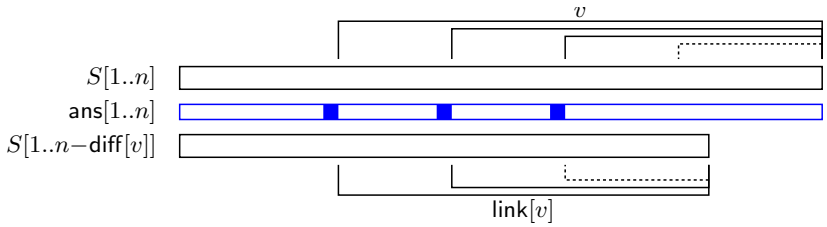


Fig. 2. Series of a palindrome v in $S[1..n]$ and of $\mathsf{link}[v]$ in $S[1..n-\mathsf{diff}[v]]$. Leading palindromes of the next series are shown by dash lines. The function $\mathsf{getMin}(v)$ returns the minimum of the values of ans in the marked positions, plus one.

We see that $\mathsf{diff}[v]$ steps before we already computed the minimum of all but one required numbers. If we memorize the minimum at that moment, we can use it now to obtain getMin in constant time. We store such a minimum as an additional parameter dp of the node of the eertree, updating it each time the palindrome represented by a node becomes a leading suffix-palindrome. Lemmas 8 and 9 ensure that when we access $\mathsf{dp}[\mathsf{link}[v]]$ to compute $\mathsf{getMin}[v]$, it is exactly the value computed $\mathsf{diff}[v]$ steps before. The computations with dp can be performed inside the getMin function:

```
int getMin(v)
       dp[v] = ans[n - (len[seriesLink[v]] + diff[v])] //last
       if (diff[v] == diff[link[v]])// non-trivial series
              dp[v] = min(dp[v], dp[link[v]])
       return dp[v] + 1
```

Here $\mathsf{dp}[v]$ is initialized by the value of ans in the position preceding the last element of the series of v. It is nothing to do if this series does not have other elements; if it has, the minimum value of ans in the corresponding positions is available in $\mathsf{dp}[\mathsf{link}[v]]$.

Remark 5. Series links can replace quick links in the construction of eertrees. Recall that in the method of quick links (Sect. 3.2) after checking the symbols in S preceding v and $\mathsf{link}[v]$ we assign $\mathsf{quickLink}[v]$ to v and repeat the process until the required symbol is found or the node -1 is reached. With series links, the termination condition is the same, but the process is a bit different. We first put $v = \mathsf{maxSuf}(S)$ and check the symbol before v. Then we keep repeating two operations: check the symbol preceding $\mathsf{link}[v]$ and assign $\mathsf{seriesLink}[v]$ to v. In this way, all "skipped" symbols, including the symbol preceding v, equal the symbol preceding the previous value of $\mathsf{link}[v]$. (This is due to periodicity of v; for details see, e.g., [12, Sect. 2].) The number of iterations of the cycle equals the number of series of suffix-palindromes of S, which is $O(\log n)$ by Lemma 10.

Remark 6. Let t_i be the number of series of suffix-palindromes for the string $S[1..i]$. Our computation of palindromic length[3] performs, on each step, the following operations. For the eertree: at most t_i+1 symbol comparisons (Remark 5) and one $(\log \sigma)$-time access to a dictionary. For palindromic length: t_i calls to getMin, which fills one cell in dp and one cell in ans.

The algorithm by Fici et al. [3, Figure 8] on each step builds three arrays (G, G', G''), each containing t_i triples of numbers; totally $9t_i$ cells to be filled. So, our algorithm should work significantly faster.

Now we return to the k-factorization problem.

Proposition 10. *Using an eertree, the k-factorization problem for a length n string can be solved online in time $O(n \log n)$.*

Proof. The above algorithm for palindromic length can be easily modified to obtain both minimum odd number of palindromes and minimum even number of palindromes needed to factor a string. Instead of ans and dp, one can maintain in the same way four parameters: ans_o, ans_e, dp_o, dp_e, to take parity into account. Now ans_o (resp., ans_e) uses dp_e (resp., dp_o), while dp_o (resp., dp_e) uses ans_o (resp., ans_e). The reference to Lemma 6 finishes the proof.

4.2 Towards a linear-time solution

A big question is whether palindromic length can be found faster than in $O(n \log n)$ time. First of all, it may seem that the bound $O(n \log n)$ for our algorithm is imprecise. Indeed, for building an eertree we scan only $O(n)$ suffix palindromes even when we use just suffix links (see the proof of Proposition 2). For palindromic length, on each step we run through all suffix-palindromes, but

[3] See `http://ideone.com/xE2k6Y` for an implementation.

possibly skipping many of them due to the use of series links. Can this number of scanned palindromes be $O(n)$ as well? As was observed in [3], the answer is "yes" on average, but "no" in the worst case: processing any length n prefix of the famous *Zimin word*, one should analyze $\Theta(n \log n)$ series of palindromes (all of them 1-element, but this does not help).

Below we design an $O(n)$ offline algorithm for building an eertree of a length n string S over the alphabet $\{1, \ldots, n\}$, getting rid of the $\log \sigma$ factor in online algorithms. Then we discuss ideas which may help to obtain the palindromic length from an eertree in linear time. The offline algorithm consists of four steps.

1. Using Manacher's algorithm, compute arrays oddR and evenR, where oddR$[i]$ (resp. evenR$[i]$) is the radius of the longest subpalindrome of S with the center i (resp., $i{+}1/2$).

2. Compute the longest and the second longest suffix-palindromes for any prefix of S. We use variables ℓ, ℓ', and r such that after rth iteration the string $S[\ell..r]$ (resp., $S[\ell'..r]$) is the longest (resp., second longest) suffix-palindrome of $S[1..r]$.

```
ℓ = 2
for (r = 1; r ≤ n; r++)
        ℓ--
        while ( !isPal(S[ℓ..r] )
                ℓ++
        ℓ' = max(ℓ' - 1 , ℓ + 1)
        while ( !isPal(S[ℓ'..r] ) && (ℓ' ≤ r) )
                ℓ'++
        C[(ℓ + r) / 2].push(1, r)
        C[(ℓ' + r) / 2].push(2, r)
```

The function isPal, checking whether a given substring is a palindrome, works in $O(1)$ time, using the value obtained on step 1 for the center $(\ell{+}r)/2$. Each element of the array C is a connected list; the indices are both integers and half-integers. The internal cycles make at most $2n$ increments of each of the variables ℓ, ℓ'; hence, the whole step works in linear time.

3. Build the suffix array SA and the LCP array for S; for the alphabet $\{1, \ldots, n\}$, this can be done in linear time [15]. Recall that $LCP[i]$ is the length of the longest common prefix of $S[SA[i]..n]$ and $S[SA[i{-}1]..n]$.

4. Recall from Sect. 2.3 that an eertree consists of two tries, containing right halves of odd-length and even-length palindromes, respectively. Build each of them using a variation of the algorithm, constructing a suffix tree from a suffix array and its LCP array [9]. The algorithm for odd-length palindromes is given below; the algorithm for even lengths is essentially the same, so we omit it.

```
path = (-1) // stack for the current branch of the trie
for (i = 1; i ≤ n; i++)
        k = SA[i] // start processing palindromes centered at k
        while (path.size() > LCP[i] + 1)
```

```
        path.pop()
for (j = path.size(); j ≤ oddR[k]; j++) //can be empty
        path.push( newNode(path.top(), S[k + j - 1]) )
for (j = 1; j ≤ C[k].size(); j++)
        (rank, r) = C[k][j]
        node[rank][r] = path[r - k + 1]
```

The function newNode(v, a) returns a new node attached to the node v with the edge labeled by a. Array node[1][1..n] (resp., node[2][1..n]) contains links to the longest (resp., second longest) palindromes ending in given positions. Now estimate the working time of this algorithm. The outer cycle works $O(n)$ time plus the time for the inner cycles. The number of pop operations is bounded by the number of pushes, and the latter is the same as the number of nodes in the resulting eertree, which is $O(n)$. The total number of iterations of the third inner cycle is the number of palindromes stored in the whole array C; this is exactly $2n$, see step 2. Thus, the algorithm works in $O(n)$ time.

After running both the above code and its modification for even-length palindromes, we obtain the eertree without suffix links and the arrays node[1], node[2]. From these arrays the suffix links can be computed trivially:

```
for (i = 1; i ≤ n; i++)
        link[ node[1][i] ] = node[2][i];
```

Thus we have proved

Proposition 11. *The eertree of a length n string over the alphabet $\{1, \dots, n\}$ can be built offline in $O(n)$ time.*

Now return to the palindromic length. Even with an $O(n)$ preprocessing for building the eertree, we still need $O(n \log n)$ time for factorization. Note that in [12] an $O(kn \log n)$ algorithm for k-factorization was transformed into a $O(kn)$ algorithm using bit compression (the so-called *method of four Russians*). That algorithm produced a $k \times n$ bit matrix (showing whether a jth prefix of the string is i-factorable), so such a speed up method was natural. In our case we work with integers, so the direct application of a bit compression is impossible. However, we have the following property.

Lemma 11. *If S is a string of palindromic length k and c is a symbol, then the palindromic length of Sc is $k-1, k$, or $k+1$.*

Proof. Any k-factorization of S plus the substring c give a $(k+1)$-factorization of Sc. Suppose Sc has a t-factorization $P_1 \cdots P_t$ for a smaller t. Then $P_t = Pc$ has length > 1. Hence, either $P = c$ and S has the t-factorization $P_1 \cdots P_{t-1}c$ or $P = cQ$ for a palindrome Q and S has the $(t+1)$-factorization $P_1 \cdots P_{t-1}cQ$. The result now follows. □

Consider a $n \times n$ bit matrix M such that $M[i, j] = 1$ if and only if $S[1..j]$ is i-factorable. For jth column, we have to compute just two values: in the rows $k-1$

and k, where k is the palindromic length of $S[1..j-1]$ (if $M[k-1,j] = M[k,j] = 0$, we write $M[k+1,j] = 1$ by Lemma 11). For each value we should apply the OR operation to $\log n$ bit values, to the total of $2n \log n$ bit operations. If we will be able to arrange these operations naturally in groups of size $\log n$, we will use the bit compression to get just $O(n)$ operations. So we end the paper with the following conjecture.

Conjecture 1. Using Lemma 11, eertree and the method of four Russians it is possible to find palindromic length of a string in $O(n \log \sigma)$ time online and in $O(n)$ time offline.

5 Conclusion

In this paper, we proposed a new tree-like data structure, named eertree, which stores all palindromes occurring inside a given string. The eertree has linear size (even sublinear on average) and can be built online in nearly linear time. We proposed some advanced modifications of the eertree, including the joint eertree for several strings, the version supporting deletions from a string, and the persistent eertree.

Then we provided a number of applications of the eertree. The most important of them are the new online algorithms for k-factorization, palindromic length, the number of distinct palindromes, and also for computing the number of rich strings up to a given length.

For further research we formulated a conjecture on the linear-time factorization into palindromes and an open problem about the optimal construction of the eertree.

Acknowledgments. The authors thank A. Kul'kov, O. Merkuriev and G. Nazarov for helpful discussions.

References

1. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. System Sci. 38(1), 86–124 (1989)
2. Droubay, X., Justin, J., Pirillo, G.: Episturmian words and some constructions of de Luca and Rauzy. Theoret. Comput. Sci. 255, 539–553 (2001)
3. Fici, G., Gagie, T., Kärkkäinen, J., Kempa, D.: A subquadratic algorithm for minimum palindromic factorization. Journal of Discrete Algorithms 28, 41–48 (2014)
4. Galil, Z., Seiferas, J.: A linear-time on-line recognition algorithm for "Palstar". J. ACM 25, 102–111 (1978)
5. Glen, A., Justin, J., Widmer, S., Zamboni, L.: Palindromic richness. European J. Combinatorics 30(2), 510–531 (2009)
6. Groult, R., Prieur, E., Richomme, G.: Counting distinct palindromes in a word in linear time. Inform. Process. Lett. 110, 908–912 (2010)
7. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Computer Science and Computational Biology. Cambridge University Press (1997)

8. Kari, L., Mahalingam, K.: Watson-Crick palindromes in DNA computing. Natural Computing 9, 297–316 (2010)
9. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Combinatorial pattern matching. LNCS, vol. 2089, pp. 181–192. Springer, Berlin (2001)
10. Knuth, D.E., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. 6, 323–350 (1977)
11. Kosolobov, D., Rubinchik, M., Shur, A.M.: Finding distinct subpalindromes online. In: Proc. Prague Stringology Conference. PSC 2013. pp. 63–69. Czech Technical University in Prague (2013)
12. Kosolobov, D., Rubinchik, M., Shur, A.M.: Pal^k is linear recognizable online. In: Proc. 41th Int. Conf. on Theory and Practice of Computer Science (SOFSEM 2015). LNCS, vol. 8939, pp. 289–301. Springer (2015)
13. Manacher, G.: A new linear-time on-line algorithm finding the smallest initial palindrome of a string. J. ACM 22(3), 346–351 (1975)
14. Mauri, G., Pavesi, G.: Algorithms for pattern matching and discovery in RNA secondary structure. Theoret. Comput. Sci. 335, 29–51 (2005)
15. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. 39(2) (2007)
16. Ravsky, O.: On the palindromic decomposition of binary words. Journal of Automata, Languages and Combinatorics 8(1), 75–83 (2003)
17. Rubinchik, M., Shur, A.M.: The number of distinct subpalindromes in random words. arXiv:1505.08043 [math.CO] (2015)
18. Sloane, N.J.A.: The on-line encyclopedia of integer sequences. Available at `http://oeis.org`
19. Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis. Theoret. Comput. Sci. 389, 278–294 (2007)
20. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
21. Problems of Asia–Pacific Informatics Olympiad 2014 (2014), available at `http://olympiads.kz/apio2014/apio2014_problemset.pdf`
22. Problems of the MIPT Fall Programming Training Camp 2014. Contest 12 (2014), available at `https://drive.google.com/file/d/0B_DHLY8icSyNUzRwdkNFa2EtMDQ`