FULLY PERSISTENT LISTS WITH CATENATION

James R. Driscoll
Daniel D. K. Sleator
Robert E. Tarjan

CS-TR-299-90

December 1990

Fully Persistent Lists with Catenation

James R. Driscoll*

Daniel D. K. Sleator†

Robert E. Tarjan‡

1. Introduction

In this paper we consider the problem of efficiently implementing a set of side-effect-free procedures for manipulating lists. These procedures are:

makelist(d):	Create and return a new list of length 1 whose first and only element is d.
first(X):	Return the first element of list X.
pop(X):	Return a new list that is the same as list X with its first element deleted. This operation does not effect X.
catenate(X, Y):	Return a new list that is the result of catenating list X and list Y, with X first, followed by Y (X and Y may be the same list). This operation has no effect on X or Y.

We show how to implement these operations so that *makelist*(d) runs in constant time and consumes constant space, *first*(X) runs in constant time and consumes no space, *pop*(X) runs in constant time and consumes constant space, and *catenate*(X, Y) runs in time (and consumes space) proportional to $\log\log k$, where k is the number of list operations done up to the time of the catenation. All of these time and space bounds are amortized over the sequence of operations.

NOTE. Another important operation on lists is *push*(d, X), which returns the list formed by adding element d to the front of list X. We treat this operation as a special case of catenation, since *catenate*(*makelist*(d), X) has the same effect as *push*(d, X). □

---

*Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH 03755. Research supported in part by the National Science Foundation under Grant No. CCR-8809573, and by DARPA as monitored by the Air Force Office of Scientific Research under Grant No. AFOSR-90-0292.

†School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. Research supported in part by the National Science Foundation under grant CCR-8658139, and by DIMACS, a National Science Foundation Science and Technology center, Grant No. NSF-STC88-09648.

‡Computer Science Department, Princeton University, Princeton, NJ 08544 and NEC Research Institute, Princeton, NJ 08540. Research at Princeton University partially supported by the Office of Naval Research, Contract No. N00014-87-K-0467, and by DIMACS.

This list manipulation problem arises naturally in list-based programming languages such as lisp or scheme. Incorporating our representation for lists into such a language would allow a programmer to use these conceptually simple procedures for manipulating lists without incurring a severe cost penalty. Another application is the implementation of continuations in a functional programming language [7].

In addition to solving these practical problems very efficiently, our result is the first non-trivial example of a fully persistent data structure that supports an operation that combines two versions. To put this work in context, we need to summarize previous work and terminology on persistence.

A typical data structure allows two types of operations: queries and updates. A query merely retrieves information, whereas an update changes the information represented. Such a data structure is said to be *ephemeral* if queries and updates may only be done on the current version. With an ephemeral structure, the sequence of versions (one for each update done) leading to the current version is lost.

In the *partially persistent* form of an ephemeral data structure, queries may be performed on any version that ever existed, while updates may be performed only on the most recent version. Thus, the collection of versions forms a linear sequence, each one (except the current one) being the predecessor of exactly one version. In the *fully persistent* form of an ephemeral data structure both queries and updates are allowed on any version of the structure. Therefore one version may be the predecessor of several versions (one for each update that was applied to it). The graph of relations between versions is a tree.

Several researchers have constructed methods for making specific data structures either partially or fully persistent [2,3,4,10,12,13,14,15,16,18,20] and some general techniques have been developed [5, 6,17,19]. In particular, our previous joint work with Neil Sarnak [5] gives efficient methods for transforming a pointer-based ephemeral data structure into one that is partially or fully persistent in a manner that satisfies ideal resource bounds: a constant factor in query time over that in the ephemeral structure, and a constant amount of space per change in the ephemeral structure. These techniques have several important limitations, however. One of these is that the updates in the ephemeral structure operate on a single version. Combining two versions together, such as catenating two lists, is not allowed. Another limitation is that the in-degree (number of pointers pointing to a particular node) in the ephemeral structure must be bounded by a fixed constant.

A fully persistent version of an ephemeral data structure that supports an update in which two different versions are combined is said to be *confluently persistent*. The relationship between versions of a confluently persistent data structure is that of a directed acyclic graph. The result of this paper, efficient confluently persistent catenatable lists, expands the range of data structures to which persistence can be applied.

As a starting point for thinking about this problem, we mention two simple approaches. If we represent the lists as singly-linked lists in the standard way, with a special *entry node* for each version that points to the first element of the version, then two lists X and Y can be catenated in time and space proportional to the length of X. This is done by duplicating the list X and calling the duplicate X', creating a new entry node that points to X', and making the last element of X' point to the beginning of Y. Popping the first element of a list is accomplished by creating a new entry node and making it point to the second element of the given list. This takes constant time and space.

This solution is an example of a more general technique known as *path-copying* [18]: when a node is to be changed, replace it by a new node, and change all nodes that point to it by applying the same idea recursively. (Such a method will obviously work best when the in-degree of the nodes is small and the paths in the structure are short.) By representing lists as balanced search trees and using the path-copying technique, pop and catenate can be made to run in $O(\log n)$ time and space (n is the length of the longest list involved in the operation). This improves the performance of catenate, but makes popping worse.

In our search for a more efficient solution we began by trying to apply the *node-copying* method of Driscoll *et al.* [5], which can make lists fully persistent using constant time and space per change. This idea breaks down because the method does not allow a pointer in a particular version to be changed to point to a node that is not part of the same version. This is because the node-copying method assigns a *version number* to each version, and maintains a data structure that contains version numbers throughout; to navigate (follow pointers) through a particular version requires comparing the version number of that version with those encountered in the traversal, and making decisions based on these comparisons. To accommodate the changing of a pointer to point into a different version would require a new navigation algorithm.

Rather than inventing a more powerful navigation scheme (which does not seem plausible), we instead constructed an efficient list representation that is itself made up of fully persistent (but not efficiently catenatable) lists. These lists are linked together to form a tree whose leaves are the elements of the list to be represented. We devised two solutions based on this underlying representation.

In our first method the subtrees of the root increase in size exponentially from left to right, and these subtrees are *3-collapsible* (a structural property to be defined later that roughly says that the leaves near the left are near the root). We call this the *increasing subtree* method. Because the subtrees increase in size exponentially, only $O(\log n)$ of them are required to represent a list of length n. Popping operates on the leftmost subtree of the root, and makes use of the 3-collapsibility property to run in constant time. Catenating moves the subtrees of the right list into the data structure of the left list one at a time in such a way as to preserve 3-collapsibility and the exponential growth of the subtrees.

Because catenatable lists can get quite large with very few operations (repeatedly catenating a singleton list to itself k times results in a list with length 2^k) we have devised a method that allows this $O(\log n)$ bound to be improved to $O(\log k)$, where k is the total number of list operations.

In our more sophisticated solution (called the *finger tree* method), we change the representation of the list of exponentially growing subtrees, replacing it by a finger search tree. This tree has $O(\log n)$ leaves, so its depth is $O(\log \log n)$. This structure allows popping in constant time and space and catenating two lists of total length n in $O(\log \log n)$ time and space. As before, the n in this bound can be replaced by k, where k is the total number of list operations. This is a dramatic improvement over the naive method which can require $\Omega(2^k)$ time and even over an $O(\log n)$-time catenation method, which can require $\Omega(k)$ time and space per catenate.

This paper is organized as follows: Section 2 outlines the fully persistent memory architecture we shall use, describes the basic representation of lists, and explains the fundamental operations of *delete*, *link*, and *pull* that will be applied to this representation. Section 3 describes the increasing subtree method and the trick for improving the time and space bound for catenation to $O(\log k)$. Section 4 outlines the finger tree method, and Section 5 concludes with open problems and directions for future research.

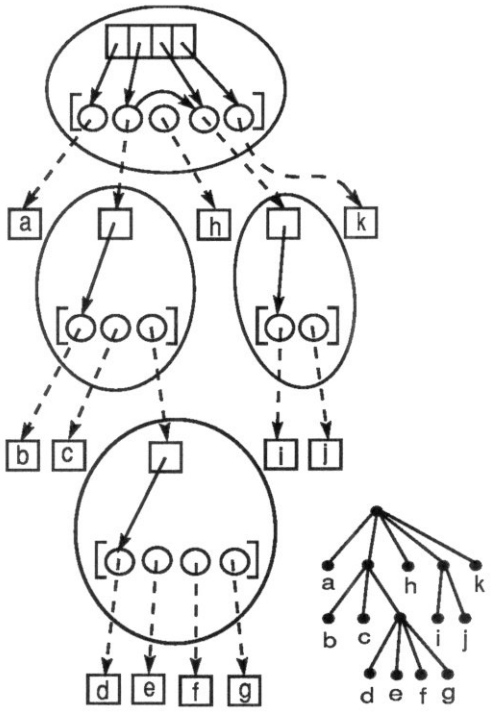

Figure 1: A representation of the list [a,b,c,d,e,f,g,h,i,j,k]. On the left is a picture of the structure. The bracketed circles represent singly-linked lists of version numbers. Version number pointers are shown as dotted lines. The portions of the structure corresponding to the four internal nodes of the tree are circled. Each of these is made fully persistent, and can be thought of as a single *memory element*. On the lower right corner is a picture of the tree.

2. Basic Representation

Our representation of a list is a rooted ordered tree with the list elements in the leaves, such that a left-to-right traversal of the tree visits the elements in the order they occur in the list. Each of the internal nodes of the tree has two or more children. No further structural constraint is imposed on these trees (except implicitly by the operations that manipulate them).

Each internal node of the tree is represented by a header node that has pointers to the first element of a singly-linked list of pointers to its children. The root node is slightly more complicated. Here the list elements for the children that are not leaves are singly linked together by additional pointers. The header node for the root also has pointers to the first and last of these non-leaf children, and to the last child. See Figure 1.

Since the representation of an internal node has a constant number of node types, bounded in-degree, and a single entry point, it can be made fully persistent at a cost of constant space per change by the node-copying method [5]. Each time such a change occurs, a new version of that node is created and is given a unique *version number*. The version number serves as a pointer to the node. It is natural to view the persistence machinery as providing *memory elements* (pointed to by version numbers), each of which contains all the data necessary to represent an internal node of the tree (a header node and a linked list). Such a memory element can be modified (creating a new memory element of this type) at a cost of constant additional space for each elementary change. While navigating pointers within a memory element, the version number of the element is used. When an edge from one memory element to another is traversed, a new version number is given to the navigation algorithm for use within the new memory element.

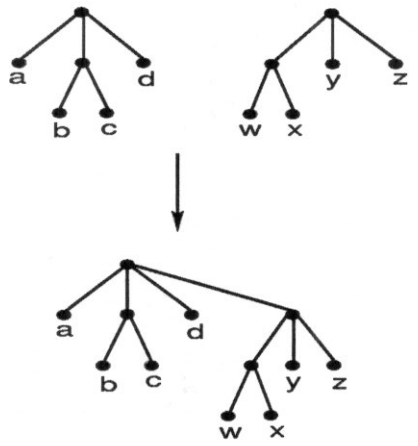

Figure 2: Linking two trees.

We now describe the three primitive operations on these trees: *link*, *delete*, and *pull*. The link operation takes two trees and makes the root of the second tree the last child of the root of the first. This requires adding a new element to the end of the root list of the first tree, making it point to the root of the second tree (the pointer being the version number of the list representing the root list of the second tree), updating the link from the last non-null child of the root to point to the new child (if this new child is not a leaf), and updating the last three fields of the header node. When a root node becomes a non-root node, certain information stored there is no longer needed (such as the pointer to the rightmost child, and the links among non-null child pointers). No harm is done by this extra information. All of these changes can be accomplished in constant time and space; and,

because the lists are persistent, the original trees are still available. See Figure 2. It is also possible to link the left tree below the root of the right tree, making the root of the left tree the first child of the root of the right tree. This form of linking is useful in the balanced linking method described in Section 3.

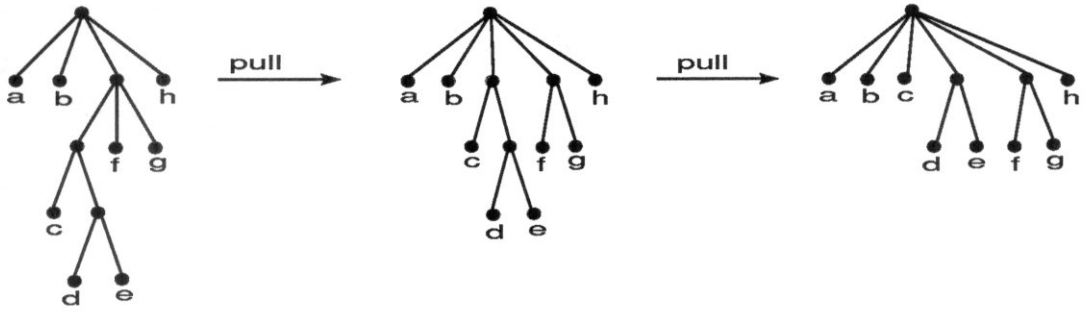

Figure 3: The pull operation.

It is important to note that the persistence mechanism used to represent the lists does not allow the root list of the first tree to point directly to the beginning of that of the second. If this were allowed it would solve the problem of efficient catenation. Instead we must settle for the less satisfactory method described above and work to insure that the way we link lists together keeps the beginning of the list easily accessible.

The delete operation removes the first child of the root, provided it is a leaf. It is invalid to apply delete to a tree whose first child is not a leaf, and in what follows we shall maintain the invariant that the first child of the root is always a leaf. The operation of deletion can be accomplished in constant time and space while maintaining persistence.

The objective of the pull operation is to bring subtrees up toward the root. It is the tool we shall use to maintain the invariant that the first child of the root is a leaf. Pull takes the first non-leaf child of the root, and "pulls" its leftmost child into the root list. See Figure 3. If this creates an internal node with only one child, this child is pulled up as well, and the now childless node is removed from the root list. This is shown in the second pull of Figure 3. If there is no non-leaf child of the root, the tree is said to be *flat*. The pull operation applied to a flat tree does nothing. The pointer to the first non-null child (and the links among the non-null children) allow this operation to be accomplished in constant time, and to be made fully persistent with only constant space consumption.

Lemma 2..1 *The number of pulls required to flatten a tree is the number of leaves minus the degree of the root.*

This lemma is a consequence of the observation that each pull increases the degree of the root by exactly one.

NOTE. In our basic representation any change in the tree requires changing all the nodes on the path from the change to the root. Therefore in designing our data structure, we must be wary of making changes deep in the tree. □

3. Preliminary Methods

The basic operations can be combined to give several different algorithms with varying efficiency tradeoffs. We first describe the *balanced linking* method. At the root of each tree we keep the size of the tree, which is the number of leaves it contains. We denote the size of a tree T by $|T|$. Catenation is implemented by linking the smaller tree below the larger tree. The pop operation repeatedly pulls the tree until the left child of the root is a leaf. This leaf is then deleted. Catenation takes constant time and space, and pop takes time and space at most proportional to the number of catenations done before the pop.

This algorithm is very closely related to the method of path compression with weighted union for maintaining disjoint sets [22]. In fact, it follows from the analysis of path compression that in an ephemeral version of the structure the amortized cost of a pop is $O(\alpha(k))$, where k is the number of list operations. Unfortunately an amortized bound of this form is simply not sufficient to give efficient full persistence. This is because a "bad" version, one in which pop is expensive, can generate arbitrarily many other bad versions. This would happen if, for example, several different short lists are catenated to the same bad version. Popping each of these new lists will be expensive and will ruin any global amortized bound. There are two ways to circumvent this problem. One is to find some way to make popping globally improve the representations of the lists (rather than locally improving a single version of a list). Another is to find a way to make the amortized bound worst-case. We pursue the latter avenue.

Call a tree *c-collapsible* if repeatly applying "delete, then pull c times" until the tree becomes empty preserves the invariant that the first child of the root is a leaf. Our goal is to arrange that all trees are c-collapsible for some constant c. We now show that c-collapsibility is preserved under linking with a sufficiently small tree.

Lemma 3..1 *Let T_1 and T_2 be two trees, and let T be the result of linking T_2 to the right of T_1, making the root of T_2 the last child of the root of T_1. If T_1 is c-collapsible and $|T_2| \leq (c-1) \cdot |T_1|$ then T is c-collapsible.*

Proof: Because T_1 is c-collapsible, $|T_1|$ collapsing steps (each one is a delete followed by c pulls) will work on T (the presence of T_2 does not interfere with the deletions). At this point $c|T_1|$ pulls have been applied to T. But $|T| = |T_1| + |T_2| \leq c|T_1|$, so $c|T_1|$ pulls are sufficient to flatten T. Therefore after $|T_1|$ collapsing steps T is flat; that is, all the leaves are children of the root. □

In particular, if we choose $c = 3$ then a tree of size up to $2n$ can be linked to a tree of size n while preserving 3-collapsibility. We now use this observation to provide a solution with $O(\log n)$ catenation time.

Represent a list by a fully persistent list of trees satisfying the invariant that every tree is 3-collapsible and the i^{th} tree has more than twice as many leaves as the $(i-1)^{\text{st}}$ tree. With each tree in the list, store the size of the tree. The doubling property implies that a list with $t \geq 2$ trees in its representation must have at least 2^t elements.

To pop the first element of such a list, apply delete followed by 3 pulls to the first tree. If this leaves an empty tree, then delete the empty tree from the front of the list of trees. This operation uses constant time and space, and the resulting list has the doubling property.

Catenation is more complicated. To catenate two lists of trees, process each tree of the right list in order from left to right. For each such tree T, compare the size of T with the current size of the rightmost tree T' of the left list. If $|T| \leq 2 \cdot |T'|$ then link T onto T'. Otherwise make T the new rightmost tree of the left list. (After this happens no further links will occur in the catenation.) See Figure 4. This process preserves the doubling property and 3-collapsibility. Catenation takes $O(\log n)$ time and space because the number of trees representing a list of n elements is at most $\log_2(n+1)$.

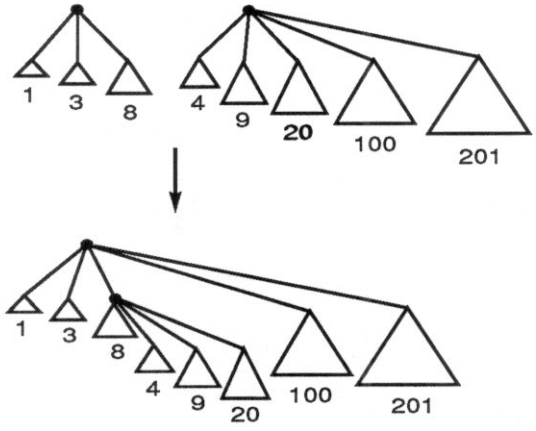

Figure 4: Catenating two lists in $O(\log n)$ time.

Although catenation has a cost that is logarithmic in the size of the lists involved, a sequence of k catenates can generate a list of length 2^k. Thus, in terms of the number of operations k, the previous result only gives us an $O(k^2)$ bound on the total time and space used to process the sequence.

We can do better in an amortized sense with the following trick. Make an initial guess $k_0 = 2$ of the total number of list operations to be performed. Carry out the operations normally, but keep track of the total size of all trees comprising a list. Keep a record of all the list operations performed as well. If a list becomes longer than $\log_2 k_0$ during a catenation then truncate it so that the length of the resulting list of trees is exactly $\log_2 k_0$. This insures that catenation will cost $O(\log k_0)$ time and space, and that at least the first k_0 elements of each list will be correctly represented. Should the actual number of operations exceed the guess k_0, then throw away the lists, pick a new guess $k_1 = k_0^2$, and rebuild the lists using the recorded sequence of operations and this new guess. That the cost of a delete remains constant in the amortized sense follows from the fact that the guess at least doubles at every phase (choose a potential function of the form $\Phi = c(k - k_i)$, where k_i is the previous guess). That the amortized cost of a catenate is $O(\log k)$ follows from the fact that at each phase the cost of a catenate is $O(\log k_i)$, $\log_2 k_{i+1} = 2 \log_2 k_i$, and the final guess k_f is at most k^2.

The same trick can be used to transform the $O(\log n)$ pop and catenate path-copying algorithm mentioned in the introduction into one with cost $O(\log k)$ per operation. This will even work if the length bound is doubled ($k_{i+1} = 2k_i$) rather than being squared at each iteration.

4. The Finger Tree Method

The idea of the finger tree method is to modify the list-of-trees method of Section 3 by replacing the list of collapsible trees[1] by a balanced binary tree. Each leaf of this balanced tree is the root of one of the collapsible trees. We retain the invariant that each successive collapsible tree is more than twice the size of its predecessor. In the list-of-trees method the catenation operation linked one collapsible tree at a time until the next one was too big, and then catenated the remainder of the $O(\log n)$-length list of collapsible trees by copying the list. The point in the list of collapsible trees at which the algorithm stops linking and starts catenating is called the *split point*. The finger tree method uses the same split point, but takes advantage of the balanced tree structure to streamline the process of transferring the collapsible trees.

The algorithm to catenate a structure B to the right of a structure A works roughly as follows: Find the split point in B. Split the balanced binary tree of B in two at the split point, creating B_l and B_r. Link B_l onto the rightmost collapsible tree of A. Join the binary trees of A and B_r together. All of the steps will be done in a fully persistent fashion, and they all will be shown to take time and space proportional to the depth of the balanced binary tree, which is $O(\log \log n)$.

4.1. Red-Black Trees

A *red-black tree* [8,22] is a binary tree in which each internal node[2] has pointers to both of its children[3] and a one bit field storing the *color* of the node, which is either red or black.

The colors of the nodes satisfy the following three conditions: Every leaf is black. Every red node has a black parent (or is the root). Every path from the root to a leaf contains the same number of black nodes. For purposes of analyzing red-black trees it is convenient to assign to each node an integral *rank* which is a function of the coloring of the tree. The rank of each leaf is 0, and the rank of an internal node is equal to the rank of its red children (if any) and one greater than that of its black children (if any). We can interpret the constraints on the coloring of the tree with respect to the ranks of the nodes as follows: (1) The rank of each leaf is 0. (2) The rank of each node that is the parent of a leaf is 1. (3) In traversing from a node to its parent the rank increases by 0 or 1. (4) In traversing from a node to its grandparent the rank increases by 1 or 2.

The *depth* of a red-black tree is the maximum number of internal nodes on a path from the root to a leaf. The following lemma (which is a slightly modified form of Lemma 4.1 of Tarjan's monograph [22]) shows that the depth of a red-black tree with n leaves is $O(\log n)$, and will be useful to us later for other purposes.

Lemma 4..1 *The subtree rooted at a node of rank r in a red-black tree contains at least 2^r leaves.*

Proof: If $r = 0$ then the node is a leaf, and the lemma holds. If $r > 0$, then the node is internal and has two children of rank at least $r - 1$. By induction each of these is the root of a subtree with at

---

[1] For brevity we suppress the "c" of "c-collapsibility" when we wish to avoid specifying a constant. We shall show later that all the trees that we call collapsible are actually 4-collapsible.

[2] The *internal nodes* of a binary tree have two children, and the *external nodes* or *leaves* of the tree have no children. Every node of a binary tree is either an internal node or a leaf.

[3] Parent pointers may also be included if needed. They are not needed in our application.

least 2^{r-1} leaves. □

The fundamental operations that we shall need to apply to these trees are inserting a new leaf, deleting a leaf, splitting a tree into two, and joining two trees together. The split operation takes as input a red-black tree along with a particular leaf l, and produces two red-black trees, one containing all the leaves to the left of l, and one containing l and all the leaves to the right of l. The join operation is the inverse of split. It produces a new red-black tree whose leaves are those of the first input tree, followed by those of the second input tree. Algorithms exist to perform all of these operations in time $O(\log n)$ (where n is the number of leaves in all of the trees involved in the operation).

The bottom-up algorithms for insertion and deletion [21,22] can roughly be described as follows: A constant amount of restructuring occurs in the vicinity of the change, then color changes are made on some initial part of the path from the change to the root of the tree, then up to three rotations[4] are done at the top of this path of color changes.

A *finger search tree* is a data structure for representing a list of elements — one of which is a special element called the *finger* — that allows very efficient accesses, insertions and deletions in the vicinity of the finger [1,9,11,23]. In particular, these three operations can be performed in time $O(\log d)$, where d is the distance in the list between the finger and the location of the operation. The method of Tsakalidis [23] is particularly useful for our purposes.

In Tsakalidis's method, each element of the list is a leaf in a red-black tree. We assume throughout this discussion that the finger is the leftmost leaf of the tree[5]. The leftmost path of internal nodes of the tree (along with the internal nodes adjacent to this path) are represented in a special way, but the rest of the tree is just as it would be in a red-black tree.

Tsakalidis's representation of the leftmost path and adjacent internal nodes is called the *spine*. We will make use of this data structure without explicitly describing it here. It has the following important properties: (1) The spine consists of nodes with a bounded number of fields. (2) Each node of the spine (except for one entry node) is pointed to only by nodes of the spine. (3) The indegree of any node (the number of pointers to it) is bounded by a constant (independent of the size of the tree). (4) An insertion or deletion operation in the finger search tree causes only a constant number of pointers and fields in the spine to change. This is a consequence of an implicit representation of the colors, and of the $O(1)$ restructuring insertion and deletion algorithms in red-black trees.

These four properties allow us to make this data structure fully persistent by the node-copying method [5] using only constant space per insertion and deletion, while preserving the $O(\log d)$ running time for these operations.

4.2. The representation

We are now ready to describe our representation of lists. The list is partitioned into consecutive blocks of elements of sizes $b_1, b_2, \ldots, b_k$. These sizes are arbitrary subject to the constraint that $b_{i+1} > 2b_i$ for all $1 \leq i < k$. Each block of elements is stored in the leaves of a 4-collapsible tree.

[4] A rotation is a local restructuring operation in a binary tree that maintains the symmetric order of the leaves and changes the depths of various nodes.

[5] In our application we only need this special case. Furthermore, such a solution can be adapted to allow arbitrary finger locations by maintaining two such trees.

Each of these collapsible trees is stored according to the basic representation described in Section 2, and the root of each tree stores its size. A persistent version of Tsakalidis's structure — which we shall call the finger tree — connects the collapsible trees together. The root of the collapsible trees are the leaves of the finger tree. The internal nodes (except those in the spine data structure) of the finger tree are represented as standard persistent memory elements. As usual, a pointer from one of these nodes to another (or to the root of a collapsible tree) is actually a version number. The spine is stored as fully persistent version of Tsakalidis' ephemeral representation of the spine.

To find the split point efficiently we store additional information in the internal nodes of the finger tree (except those in the spine). In such a node x, whose descendants are collapsible trees of sizes $b_i, b_{i+1}, \ldots, b_j$, we store the following information: $S_x = \sum_{i \le l \le j} b_l$ (the number of leaves that are descendants of this node) and $Q_x = \max_{i \le l \le j} \{b_l - 2 \sum_{i \le m < l} b_m\}$. These quantities can be maintained efficiently under any restructuring of the tree since, if l and r are the left and right child, respectively, of a node x, we have $S_x = S_l + S_r$ and $Q_x = max\{Q_l, Q_r - 2S_l\}$.

This completes the static description of our data structure. It remains to describe and analyze pop and catenate.

4.3. Pop

We first observe that the leftmost collapsible tree can be deleted from the structure (or a new leftmost collapsible tree can be inserted into the structure) persistently in constant time and space. This is because the operation on the finger tree is taking place within a constant distance of the finger (0 or 1 for deletion and insertion respectively), and because the values of S_x and Q_x can be updated in constant time. (Since we do not store these values for nodes in the spine, no updating of these is needed. It is possible, however, that because of the rotations that occur we may introduce a new node outside of the spine for which we must compute these values. This can be done using the update rule above. There are only $O(1)$ such nodes.)

To perform pop, delete the leftmost collapsible tree from the finger tree. Delete the leftmost leaf from it, and apply four pull operations. If the resulting tree is empty, the operation is complete. If it is not empty, then insert the collapsible tree back into the finger tree. All of these steps take constant time and use constant space.

The resulting structure still satisfies all of the required structural constraints; decreasing b_1 maintains $b_2 > 2b_1$, and deleting one element from a 4-collapsible tree and applying four pulls leaves a 4-collapsible tree.

4.4. Catenate

We describe a catenation method that has the property that the time (and space) required is proportional to the depth of the finger trees involved (which is $O(\log\log n)$ for forming a list of length n), and that maintains all the structural constraints above.

The first step of the algorithm is to transform the two finger trees into red-black trees, in which each internal node is represented by a standard persistent memory element and is endowed with a color and values for S_x and Q_x. This transformation can be done in time and space proportional to the size of the spine of a finger tree, which is proportional to the depth of the finger tree. The

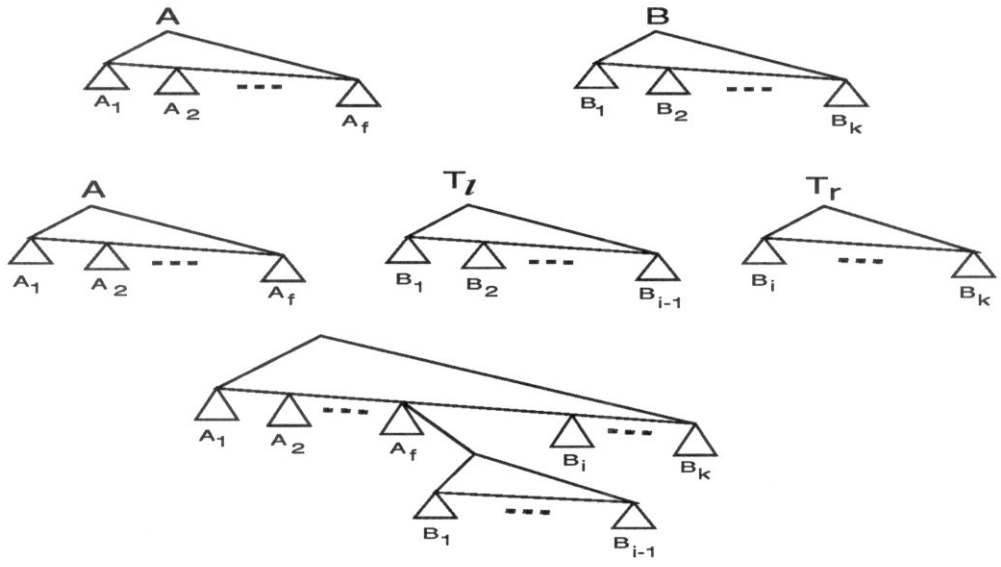

Figure 5: The top picture above shows the initial trees A and B. The center picture is an intermediate point in catenating A and B. The bottom picture shows the final result.

colors can be extracted from their implicit representation in the spine, and the data values can be computed from those of their children, which are available.

Let the two structures to be catenated be A and B, and let the collapsible trees of A be $A_1, A_2, \ldots, A_f$ of sizes $a_1, a_2, \ldots, a_f$, and the collapsible trees of B be $B_1, B_2, \ldots, B_k$ of sizes $b_1, b_2, \ldots, b_k$. The split point is defined to be the least $i \geq 1$ such that $b_i > 2\left(a_f + \sum_{1 \leq j \leq i-1} b_i\right)$ (or equivalently $2a_f < b_1 - 2\sum_{1 \leq j \leq i-1} b_i$). If this inequality is not true for any $i \leq k$ then the split point is defined to be $k+1$. We can now describe how to determine if the split point is in a subtree T rooted at the node x given S_l (the total size of the trees to the left of the subtree T) and a_f. If $2a_f < Q_x - 2S_l$ then a point i satisfying the above inequalities is in T, otherwise there is no such point in T. Using this test it is easy to locate the split point in time proportional to the depth of the red-black tree: merely walk down from the root (updating S_l) always choosing the leftmost option that contains a point satisfying the inequality. The leaf we reach in this way is the split point.

Once the split point i has been found, we split the red-black tree at this point, creating a new red-black tree T_l with collapsible trees $B_1, B_2, \ldots, B_{i-1}$, and a new red-black tree T_r with collapsible trees $B_i, B_{i+1}, \ldots, B_k$. (If the split point is $k+1$ then T_r is empty. If the split point is 1 then T_l is empty.) We now change the representation of the leftmost path of T_l into a single persistent memory element, link the resulting structure to the right of A_f, and call the result A'_f. This link occurs deep in the red-black tree of A. Making such a change persistent requires that all the nodes on the path from A_f to the root be changed (just as in path-copying). Furthermore, the S_x and Q_x fields of all the nodes on this path must be updated. The cost of these changes is proportional to the depth of the red-black tree. Finally we join the new version of the red-black tree of A with the red-black tree of T_r, and transform the resulting red-black tree into a finger tree (by building the spine).[6] The

[6]In linking T_l into A_f, we have in a sense mixed apples and oranges, since the memory elements of the nodes of T_l contain color and size fields. After the link is done, the data in these fields is no longer needed. No harm is done by the presence of this extra useless information in the tree. A slightly different approach is to completely avoid the use of the

process is shown schematically in Figure 5.

The $O(\log\log n)$ bound on the time and space cost of catenate follows immediately from the structural constraints on these trees. It remains to prove that these constraints are preserved by our algorithm.

In the data structure resulting from a catenation, each collapsible tree is more than twice the size of its predecessor. To verify this we need only check the two places where this constraint might be violated. These are between a_{f-1} and a'_f and between a'_f and b_i. The former place satisfies the constraint since $a'_f \geq a_f$. By our choice of the split point, we know that $b_i > 2(a_f + b_1 + \cdots + b_{i-1}) = 2a'_f$, which proves that the latter point also satisfies the constraint.

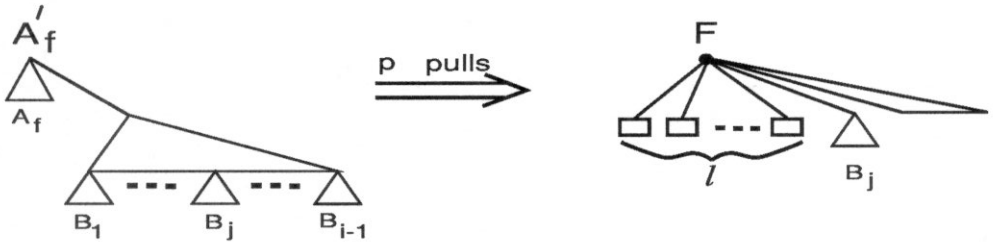

Figure 6: The process of transforming A'_f to F by applying p pulls.

The only remaining detail is to show that the tree A'_f is 4-collapsible. This requires a more detailed analysis of the structure of red-black trees and properties of the pull operation.

Lemma 4..2 *Let l be the ith leaf ($i \geq 2$) of a red-black tree. The distance between l and the nearest internal node on the left path is at most $2\lceil \log i \rceil$.*

Proof: Let $r = \lceil \log i \rceil$. Let x be the deepest node on the left path of the red-black tree that is of rank r. By Lemma 4..1 the subtree rooted at x contains at least 2^r leaves. These leaves are consecutive starting from the leftmost leaf. Since $2^r \geq i$ this subtree must contain l, so x is an ancestor of l.

In traversing the path from l toward the root of the tree, the first step goes from a node of rank 0 to one of rank 1. Subsequently every pair of steps increases the rank by at least 1. Therefore after $2r + 1$ steps a node of rank at least $r + 1$ must be reached. Since x is on this path and has rank r it must have been passed. Therefore x is reached after at most $2r$ steps. □

Lemma 4..3 *For any j ($1 \leq j \leq i - 1$), let l be the number of leaves to the left of B_j in A'_f. The number of pulls that must be applied to A'_f until a state is reached in which B_j and all the leaves to the left of B_j are adjacent to the root of the tree is at most $2l$.*

Proof: Let F be the tree (reached by applying pulls to A'_f) in which B_j and all of the l leaves to the left of B_j are adjacent to the root. (See Figure 6.) Let p be the number of pulls required to reach this state starting from tree A'_f. Our goal is to bound p by $2l$.

---

persistent memory architecture in the nodes of the red-black tree. This approach works, but a different non-uniformity results when the link is done. Now the resulting tree contains a mixture of nodes, some of which are ordinary red-black tree nodes, while others are persistent nodes. It is too expensive to clean up the entire tree immediately. Instead the pop algorithm must replace the red-black nodes by persistent nodes gradually as they are encountered.

The *stripped tree* S is obtained by starting with A'_f and deleting all nodes that are not ancestors of a leaf in B_j or a leaf to the left of B_j. The path from the root of B_j to the root of S is a rightmost path of S. (See Figure 7.)

Let d be the distance between the root of B_j and the root of S (or equivalently the root of A'_j). Let q be the number of nodes on this path that have only one child. (Although all internal nodes in A'_f have at least two children, this may not be true in S.)

Let F' be the stripped form of F. We shall now evaluate p', the number of pulls required to transform S into F'. Each pull either increases the degree of the root by one or deletes an internal node with one child. Therefore

$$\begin{aligned} p' &= (\text{degree of root of } F') - (\text{degree of root of } S) + q \\ &= 1 + l + q - (\text{degree of root of } S) \\ &\leq 1 + l + q - 2 \;=\; l + q - 1. \end{aligned}$$

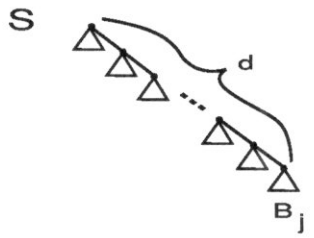

Figure 7: The stripped tree S.

What is the relationship between p and p'? Stripping accelerates the pulling process by at most one for every node on the path from the parent of the root of B_j to the root of S, except for the root and the nodes on this path with one child. This is because for the nodes with more than one child, the pull that promotes the subtree immediately to the left of the path also promotes the subtree containing B_j. This extra promotion may not happen in the pull sequence in the unstripped tree. (It depends on whether stripping removed a subtree to the right of the node or not.) To summarize, we have

$$p \;\leq\; p' + d - 1 - q \;\leq\; l + d - 2.$$

Now we observe that $d \leq j + 2$. Lemma 4..2 shows that the distance between the root of B_j and the root of T_l (after the leftmost path of T_l is made into a single memory element) is at most $2\lceil \log j \rceil$. It is easy to verify that $2\lceil \log j \rceil \leq j + 1$ for all $j \geq 2$. For $j = 1$, the distance between the root of B_1 and the root of A'_f is 2. This completes the proof of the observation.

Also note that $j \leq l$, since each of $A_f, B_1, B_2, \ldots, B_{j-1}$ has at least one leaf. Combining these observations gives

$$p \;\leq\; l + d - 2 \;\leq\; l + j + 2 - 2 \;\leq\; l + j \;\leq\; 2l\,. \;\square$$

Lemma 4..4 *The tree A'_f is 4-collapsible.*

Proof: The number of pulls required to transform F (of the previous lemma) into a tree in which all the leaves of B_j are adjacent to the root is at most b_j. By the choice of the split point, we know that $b_j \leq 2l$.

When repeatedly applying pop to the tree A'_f, by the time the first l leaves have been deleted, $4l$ pulls have been done. By Lemma 4..3, the first $2l$ of these pulls suffice to create the tree F, and the remaining ones suffice to collapse B_j. Thus, by the time any leaves of B_j need to be popped, all of the leaves of B_j are adjacent to the root.

We now consider the entire process of popping A'_f. The first a_f pops work because the tree A_f is 4-collapsible. It follows from the discussion above that by the time all of the leaves of A_f have been popped, all of the leaves of B_1 are adjacent to the root. By the time these leaves have been deleted, all the leaves of B_2 are adjacent to the root. This argument applies for all B_j, so we conclude that A'_f is 4-collapsible. □

As before, we can convert the $O(\log\log n)$ catenation time to amortized $O(\log\log k)$. In the previous version each k_i was squared, thereby doubling $\log_2 k_i$. Here we choose $k_i = 2^{2^{2^i}}$, thereby doubling $\log\log k_i$ at each new guess. Truncation can be used to ensure that the height of the finger tree is $O(\log\log k_i)$. (From a practical perspective, an additional detail to be considered is that the size of the guesses grows so rapidly that some guess may become too large to represent and manipulate efficiently in a real computer. If this happens, just truncate the lists at the maximum representable size. In such a setting it is unlikely that the number of operations would exceed the size of the largest representable number.)

5. Remarks

We have shown how to make lists persistent under the operations of pop and catenate, where any two versions of a list may be catenated. Although the time bounds for the operations are low, we have left a variety of interesting open problems to explore.

A large amount of machinery was used to achieve the $O(\log\log k)$ result, which means that it is probably not very useful in practice. Is there a simpler method to achieve the same bound? From a practical perspective the best method to use is probably path-copying in a binary search tree, with the doubling trick mentioned at the end of section 3.

There is no non-trivial lower bound known for the this problem. Our intuition is that there is probably a way of catenating in constant time, or failing that in time that is $O(\log^*(k))$ or $O(\alpha(k))$. It is an intriguing open problem to find such a method.

Another direction of future work is to make other data structures confluently persistent. A close relative of our lists is the catenatable deque, which allows pops from both ends. We believe that our techniques can be modified to handle this case efficiently as well.

We have devised an alternative way of representing catenatable lists that extends to the problem of catenatable deques, with somewhat worse resource bounds. The idea is to use red-black trees with lazy recoloring [5] to represent the deques, with certain paths in the trees themselves represented by balanced search trees. The resulting data structure supports pop and catenate in time and space $O(\log\log n)$, or amortized time and space $O(\log\log k)$ using a variant of the list truncation trick. The amortization in all our $O(\log\log k)$ results can be eliminated by using incremental rebuilding of the

data structures. Space constraints preclude us from giving the details of these results in this version of the paper.

A bigger challenge is to devise a general technique for automatically making an ephemeral combinable data structures confluently persistent. Such a result would be very useful in the implementation of high-level languages. It would allow, for instance, the efficient passing of large linked data structures as call-by-value parameters since the data structure would not need to be copied, but could be modified by means of persistence at low cost per change.

We were motivated to attack this problem after it was suggested by Mark Krentel. The problem is also listed by Driscoll *et al.*

References

[1] M. R. Brown, R. E. Tarjan, Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing,* **9**, 594-614.

[2] B. Chazelle, How to search in history, *Information and Control* **77** (1985), 77-99.

[3] R. Cole, Searching and storing similar lists, *Journal of Algorithms* **7** (1986), 202-220.

[4] D. P. Dobkin and J. I. Munro, Efficient uses of the past, *Journal of Algorithms* **6** (1985), 455-465.

[5] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, Making data structures persistent, *Journal of Computer and Systems Science* **38** (1989), 86-124.

[6] P. F. Dietz, Fully persistent arrays, Proceedings of the 1989 Workshop on Algorithms and Data Structures, Ottawa Canada, Lecture Notes in Computer Science No. 382, Springer-Verlag, 1989, 67-74.

[7] M. Felleisen, M. Wand, D. P. Friedman, B. F. Duba, Abstract continuations: a mathematical semantics for handling full functional jumps, Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, ACM Order No. 552880, 52-62.

[8] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees, in Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science, 1980, pages 8-21.

[9] L. J. Guibas, E. M. McCreight, M. F. Plass, J. R. Roberts, A new representation for linear lists, in Proceedings of the 9th Annual ACM Symposium on Theory of Computing, 1977, pages 49-60.

[10] R. Hood and R. Melville, Real-time queue operations in pure LISP, *Information Processing Letters.* **13** (1981). 50-54.

[11] R. Kosaraju, Localized search in sorted lists, in Proceedings of the 14th Annual ACM Symposium on Theory of Computing, 1981, pages 62-69.

[12] T. Krijnen and L. G. L. T. Meertens, "Making B-Trees Work for B," IW 219 83. The Mathematical Centre, Amsterdam, The Netherlands, 1983.

[13] E. W. Myers, "AVL Dags," TR 82-9, Department of Computer Science, The University of Arizona, Tucson, AZ, 1982.

[14] E. W. Myers, An applicative random-access stack. *Information Processing Letters.* **17** (1983), 241-248.

[15] E. W. Myers, Efficient applicative data type, *in* "Conference Record Eleventh Annual ACM Symposium on Principles of Programming Languages, 1984," 66-75.

[16] M. H. Overmars, Searching in the past, I, *Information and Computation*, in press.

[17] 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.

[18] T. Reps, T. Teitelbaum, and A. Demers, Incremental context-dependent analysis for language based editors, *ACM Transactions on Programming Languages and Systems,* **5** (1983), 449-477.

[19] N. Sarnak, "Persistent Data Structures," Ph. D. thesis, Department of Computer Science, New York University, New York, 1986.

[20] G. F. Swart, "Efficient Algorithms for Computing Geometric Intersections," Technical Report 85-01-02, Department of Computer Science, University of Washington, Seattle, WA, 1985.

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

[22] R. E. Tarjan, "Data structures and network algorithms," Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[23] A. K. Tsakalidis, "An Optimal Implementation for Localized Search," A 84/06, Fachbereich Angewandte Mathematik und Informatik, Universität des Saarlandes, Saarbrüken, West Germany, 1984.