Confluently Persistent Tries for Efficient Version Control*

Erik D. Demaine[†‡] Stefan Langerman[§] Eric Price[†]

Abstract

We consider a data-structural problem motivated by version control of a hierarchical directory structure in a system like Subversion. The model is that directories and files can be moved and copied between two arbitrary versions in addition to being added or removed in an arbitrary version. Equivalently, we wish to maintain a confluently persistent trie (where internal nodes represent directories, leaves represent files, and edge labels represent path names), subject to copying a subtree between two arbitrary versions, adding a new child to an existing node, and deleting an existing subtree in an arbitrary version.

Our first data structure represents an n-node degree-Δ trie with $O(1)$ "fingers" in each version while supporting finger movement (navigation) and modifications near the fingers (including subtree copy) in $O(\lg \Delta)$ time and space per operation. This data structure is essentially a locality-sensitive version of the standard practice—path copying—costing $O(d \lg \Delta)$ time and space for modification of a node at depth d, which is expensive when performing many deep but nearby updates. Our second data structure supporting finger movement in $O(\lg \Delta)$ time and no space, while modifications take $O(\lg n)$ time and space. This data structure is substantially faster for deep updates, i.e., unbalanced tries. Both of these data structures are functional, which is a stronger property than confluent persistence. Without this stronger property, we show how both data structures can be sped up to support movement in $O(\lg \lg \Delta)$, which is essentially optimal. Along the way, we present a general technique for global rebuilding of fully persistent data structures, which is nontrivial because amortization and persistence do not usually mix. In particular, this technique improves the best previous result for fully persistent arrays and obtains the first efficient fully persistent hash table.

1 Introduction

This paper is about a problem in persistent data structures motivated by an application in version control. We begin by describing the motivating application, then our model of the underlying theoretical problem, followed by our results.

---

*A preliminary version of this paper appears in *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory*, LNCS 5124, July 2008, pages 160–172.

†MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA. {edemaine,ecprice}@mit.edu

‡Supported in part by MADALGO — Center for Massive Data Algorithmics — a Center of the Danish National Research Foundation.

§Maître de recherches du F.R.S.-FNRS, Computer Science Department, Université Libre de Bruxelles, CP 212, Bvd. du Triomphe, 1050 Brussels, Belgium. stefan.langerman@ulb.ac.be

Version control. Increasingly many creative works on a computer are stored in a *version control system* that maintains the history of all past versions. The many motivations for such systems include the ability to undo any past mistake (in the simplest form, never losing a file), and the ability for one or many people to work on multiple parallel branches (versions) that can later be merged. Source code has been the driving force behind such systems, ranging from old centralized systems like RCS and CVS, to the increasingly popular centralized system Subversion, to recent distributed systems like Bazaar, darcs, GNU arch, Git, Monotone, and Mercurial. By now version control is nearly ubiquitous for source code and its supporting documentation. We also observe a rise in the use of the same systems in academic research for books, papers, figures, classes, etc.[1] In more popular computer use, Microsoft Word supports an optional form of version control ("change tracking"), Adobe Creative Suite (Photoshop, Illustrator, etc.) supports optional version control ("Version Cue"), and most Computer Aided Design software supports version control in what they call Product Data Management (e.g., Autodesk Productstream for AutoCAD and PDMWorks for SolidWorks). Entire modern file systems are also increasingly version controlled, either by taking periodic global snapshots (as in AFS, Plan 9, and WAFL), or by continuous change tracking (as in Apple's new Time Machine in HFS+, and in experimental systems CVFS, VersionFS, Wayback, and PersiFS [PCD05]). As repositories get larger, even to the point of entire file systems, high-performance version control is in increasing demand. For example, the Git system was built simply because no other free system could effectively handle the Linux kernel.

Requirements for version control. Most version control systems mimic the structure of a typical file system: a tree hierarchy of directories, each containing any number of linear files. Changes to an individual file can therefore be handled purely locally to that file. Conceptually these changes form a tree of versions of the file, though all systems represent versions implicitly by storing a delta ("diff") relative to the parent. In this paper, we do not consider such file version tracking, because linear files are relatively easy to handle.

The more interesting data structural challenge is to track changes to the hierarchical directory structure. All such systems support addition and removal of files in a directory, and creation and deletion of empty subdirectories. In addition, every system since Subversion's pioneering innovation supports moving or copying an entire subdirectory from one location to another, possibly spanning two different versions. This operation is particularly important for merging different version branches into a common version.

Persistent trie model. Theoretically, we can model version control of a hierarchical directory structure as a confluently persistent trie, which we now define.

A *trie* is a rooted tree with labeled edges. In the version-control application, internal nodes represent directories, leaves represent files, and edge labels represent file or directory names.[2] The natural queries on tries are navigation: placing a finger at the root, moving a finger along the edge with a specified label, and moving a finger from a node to its parent. We assume that there are $k = O(1)$ fingers in any single version of the trie; in practice, two fingers usually suffice. Each node has some constant amount of information which can be read or written via a finger; for example, each leaf can store a pointer to the corresponding file data structure. The structural

[1] For example, this paper is maintained using Subversion.

[2] We assume here that edge labels can be compared in constant time; in practice, this property is achieved by hashing the file and directory name strings.

changes supported by a trie are insertion and deletion of leaves attached to a finger (corresponding to addition and removal of files), copying the entire subtree rooted at one finger to become a new child subtree of another finger (corresponding to copying subdirectories), and deleting an entire subtree rooted at one finger (enabling moving of subdirectories). Subtree copying propagates any desired subset of the fingers of the old subtree into the new subtree, provided the total number of fingers in the resulting trie remains at most k.

The trie data structure must also be "confluently persistent". In general, persistent data structures preserve old versions of themselves as modifications proceed. A data structure is *partially persistent* if the user can query old versions of the structure, but can modify only the most recent version; in this case, the versions are linearly ordered. A data structure is *fully persistent* if the user can both query and modify past versions, creating new branches in a tree of versions. The strongest form of persistence is *confluent persistence*, which includes full persistence but also supports certain "meld" operations that take multiple versions of the data structure and produce a new version; then the version dependency graph becomes a directed acyclic graph (DAG). The version-control application demands confluent persistence because branch merging requires the ability to copy subdirectories (subtrees) from one version into another.

Related work in persistence. Partial and full persistence were mostly solved in the 1980's. In 1986, Driscoll et al. [DSST89] developed a technique that converts any pointer-based data structure with bounded in-degree into an equivalent fully persistent data structure with only constant-factor overhead in time and space for every operation. In 1989, Dietz [Die89] developed a fully persistent array supporting random access in $O(\lg \lg m)$ time, where m is the number of updates made to any version. This data structure enables simulation of an arbitrary RAM data structure with a log-logarithmic slowdown. Furthermore, this slowdown is essentially optimal, because fully persistent arrays have a lower bound of $\Omega(\lg \lg n)$ time per operation in the powerful cell-probe model.[3]

More relevant is the work on confluent persistence. The idea was first posed as an open problem in [DSST89]. In 1994, Driscoll et al. [DST94] defined confluence and gave a specific data structure for confluently persistent catenable lists. In 2003, Fiat and Kaplan [FK03] developed the first and only general methods for making a pointer-based data structure confluently persistent, but the slowdown is often suboptimal. In particular, their best deterministic result has a linear worst-case slowdown. Although their randomized result has a polylogarithmic amortized slowdown, it can take a linear factor more space per modification, and furthermore the answers are correct only with high probability; they do not have enough time to check the correctness of their random choices.

Fiat and Kaplan [FK03] also prove a lower bound on confluent persistence. They essentially prove that most interesting confluently persistent data structures require $\Omega(\lg p)$ space per operation in the worst case, even with randomization, where p is the number of paths in the version DAG from the root version to the current version. Note that p can be exponential in the number m of versions, as in Figure 1, resulting in a lower bound of $\Omega(m)$ space per operation. The lower bound follows from the possibility of having around p addressable nodes in the data structure; in particular, it is easy to build an exponential amount of data (albeit with significant repetition) using a linear number of confluent operations. However, their $\Omega(\lg p)$ lower bound requires a crucial and

[3]Personal communication with Mihai Pătraşcu, 2008. The proof is based on the predecessor lower bounds of [PT07].

unreasonable assumption: that all nodes of the structure can be addressed at any time. From the perspective of actually using a data structure, it is much more natural for the user to have to locate the data of interest using a sequence of queries. For this reason, our use of trie traversals by a constant number of fingers is both natural and critical to our success.

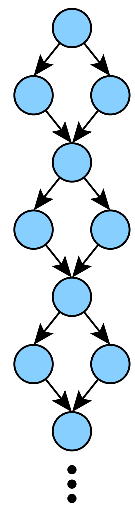


Figure 1: This version DAG has exponentially many paths from top to bottom, and can result in a structure with an exponential amount of data.

Functional data structures. Given the current lack of general transformations into confluently persistent data structures, efficient such structures seem to require exploiting the specific problem. One way to attain confluent persistence is to design a *functional* data structure, that is, a read-only (pointer-based) data structure. Such a data structure can create new cells with new initial contents, but cannot modify the contents of old cells. Each modifying operation requires a pointer to the new version of the data structure, described by a newly created cell. Functional data structures have many useful properties other than confluent persistence; for example, multiple threads can use functional data structures without locking. Pippenger [Pip97] proved a logarithmic-factor separation between the best pointer-based data structure and the best functional data structure for some problems. On the other hand, many common data structures can be implemented functionally with only a constant-factor overhead; see Okasaki [Oka98]. One example we use frequently is a functional *catenable deque*, supporting insertion and deletion at either end and concatenation of two deques in constant time per operation [Oka98].

Path copying. Perhaps the simplest technique for designing functional data structures is *path copying* [ST86, Oka98], an idea introduced independently by several groups [KM83, Mye82, Mye84, RTD83, Swa85]. This approach applies to any tree data structure where each node modification depends on only the node's subtree. Whenever we would modify a node v in the ephemeral (nonpersistent) structure, we instead create new copies of v and all ancestors of v. Because nodes depend on only their subtrees and the data structure becomes functional (read only), we can safely re-use all other nodes. Figure 2 shows an example of path copying in a binary search tree (which achieves logarithmic worst-case performance).

Version control systems including Subversion effectively implement path copying. As a result, modifications to the tree have a factor-$\Theta(d)$ overhead, where d is the depth of the modified node. More precisely, for a pointer-based data structure, we must split each node of degree Δ into a binary tree of height $O(\lg \Delta)$, costing $O(d \lg \Delta)$ time and space per update. In fact, Subversion copies a node representing a directory by copying the entire listing of files and directories contained within the directory, resulting in a substantially higher cost.

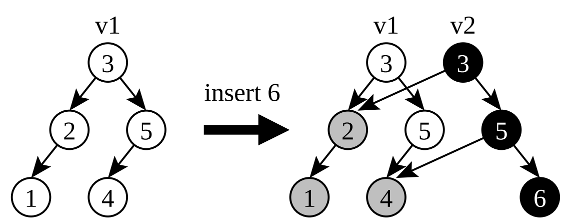


Figure 2: Path copying in a binary search tree. The old version (v1) consists of white and grey nodes; the new version (v2) consists of black and grey nodes.

Imbalance. The $O(d \lg \Delta)$ cost of path copying (and the larger cost incurred by Subversion) is potentially very large because the trie may be extremely unbalanced. For example, if the trie is a path of length n, and we repeatedly insert n leaves at the bottommost node, then path copying requires $\Omega(n^2)$ time and space.

Douceur and Bolosky [DB99] studied over 10,000 file systems from nearly 5,000 Windows PCs in a commercial environment, totaling 140 million files and 10.5 terabytes. They found that d roughly follows a Poisson distribution, with 15% of all directories having depth at least eight. Mitzenmacher [Mit03] studies a variety of theoretical models for file-system creation which all imply that d is usually logarithmic in n.

To better understand the case of actual version control, Figures 3 and 4 plot the distributions of depth, degree, and Subversion cost observed on August 29, 2008 in the Subversion repositories for all 185 Google Code projects and the 1,000 most active SourceForge projects.[4] Our experiment replays the log of every Subversion repository; for each file added, deleted, or modified, we measure the number of ancestors of the file (*depth*), the number of files in the directory containing the file (*degree*), and the sum of the number of files in that directory and all ancestors (*Subversion cost*). (True Subversion cost depends on the length of the filenames as well, but we choose this measure for having a similar scale to depth and degree.) This approach effectively measures the costs of what would be observed in the (past) use of the repository, in contrast to e.g. measuring the depths and degrees in the latest (trunk) version. We aggregate all counts over all repositories, which effectively weights toward larger (more interesting) projects. In general, we observe distributions with relatively low means but heavy tails. We believe that our worst-case results would mitigate the high cost incurred by the heavy tails.

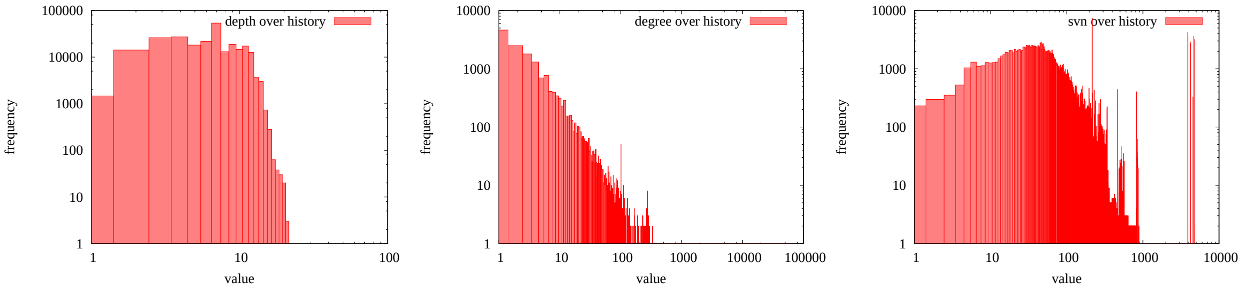
Our results. We develop four trie data structures, two of which are functional and two of which are efficient but only confluently persistent; see Table 1. All four structures effectively blast through the lower bound of Fiat and Kaplan.

Our first functional trie enables exploiting locality of reference among any constant number of fingers. Both finger movement (navigation) and modifications around the fingers (including subtree copy) cost $O(\lg \Delta)$ time and space per operation, where Δ is the total degree of the nodes directly involved. Note that navigation operations require space as well, though the space is permanent only if the navigation leads to a modification; stated differently, the space cost of a modification is effectively $O(t \lg \Delta)$ where t is the distance of the finger from its last modification. This data

[4] Projects listed on http://code.google.com/hosting/projects.html and http://sourceforge.net/softwaremap/trove_list.php?form_cat=18&limit=100, respectively.

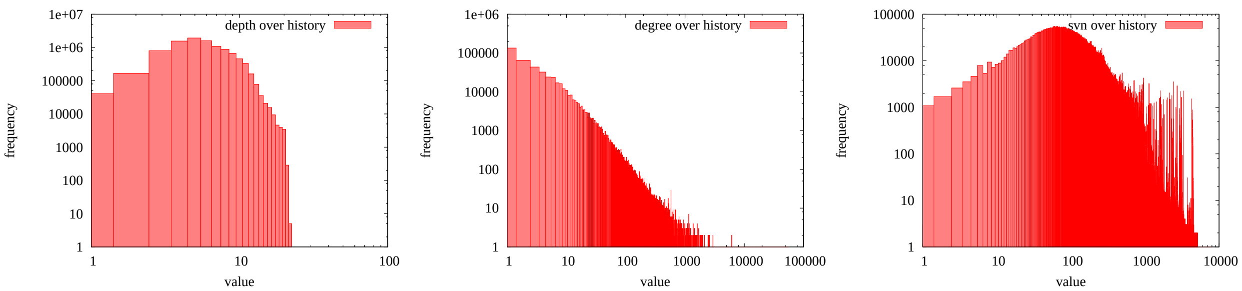
Method	Finger movement		Modifications	
	Time	Space	Time	Space
Path copying	$\lg \Delta$	0	d	d
Locality-sensitive (functional)	$\lg \Delta$	$\lg \Delta$	$\lg \Delta$	$\lg \Delta$
Locality-sensitive (confluently persistent)	$\lg \lg \Delta$	$\lg \lg \Delta$	$\lg \lg \Delta$	$\lg \lg \Delta$
Globally balanced (functional)	$\lg \Delta$	0	$\lg n$	$\lg n$
Globally balanced (confluently persistent)	$\lg \lg \Delta$	0	$\lg n$	$\lg n$

Table 1: Time and space complexity of data structures described in this paper. Operations are on an n-node trie at a node of depth d and degree Δ.



(a) Depth (mean 4.1). (b) Degree (mean 15.5). (c) Subversion cost (mean 54.3).

Figure 3: Log-log frequency distributions of observed parameters in all 185 Google Code projects' Subversion repositories, totaling 246,718 operations on 105,065 files.



(a) Depth (mean 5.1). (b) Degree (mean 35.8). (c) Subversion cost (mean 119.9).

Figure 4: Log-log frequency distributions of observed parameters in the 1,000 most active SourceForge projects' Subversion repositories, totaling 9,802,767 operations on 4,285,164 files.

structure is always at least as efficient as path copying, and much more efficient in the case of many deep but nearby modifications. In particular, the quadratic example of inserting n leaves at the bottom of a length-n path now costs only $O(n \lg \Delta)$ time and space. The main idea behind this relatively simple data structure is to split the trie at the fingers and all branching points in the Steiner tree on the fingers, represent the paths of the Steiner tree by deques, and explicitly maintain the subtries hanging off these paths. In combination, the data structure is a tree of deques of tries.

Our second functional trie guarantees $O(\lg n)$ time and space per modification, with no space required by navigation, while preserving $O(\lg \Delta)$ time per navigation. This data structure is substantially more space-efficient than the first data structure whenever modifications are deep and dispersed. For example, if we insert n leaves alternately at the top and at the bottom of a length-n path, then the time cost from navigation is $\Theta(n^2)$, but the space cost is only $O(n \lg n)$. The only disadvantage is that nearby modifications still cost $\Theta(\lg n)$ time and space, whereas the $O(t \lg \Delta)$ cost of the first data structure can be a bit smaller. The main idea behind this relatively complex data structure is to internally rebalance the unbalanced trie, based on techniques from worst-case link-cut trees, to put all nodes at effective depth $O(\lg n)$, and then use path copying. The main challenge is in maintaining the correspondence between fingers of deep nodes in the trie and fingers of shallow nodes in the representation.

Our two confluently persistent trie data structures are based on the functional data structures, replacing each height-$O(\lg \Delta)$ binary tree representation of a degree-Δ trie node with a new log-logarithmic fully persistent hash table. For the first structure, we obtain an exponentially improved bound of $O(\lg\lg\Delta)$ time and space per operation. For the second structure, we improve the movement cost to $O(\lg\lg\Delta)$ time (and no space). These operations have matching $\Omega(\lg\lg\Delta)$ time lower bounds because in particular they implement fully persistent arrays.

To our knowledge, efficient fully persistent hash tables have not been obtained before. The obvious approach is to use standard hash tables while replacing the table with the fully persistent array of Dietz [Die89]. There are two main problems with this approach. First, the time bound for fully persistent arrays is $O(\lg\lg m)$, where m is the number of updates to the array, but this bound can be substantially larger than even the size Δ of the hash table. Second, hash tables need to dynamically resize, and amortization does not mix well with persistence: the hash table could be put in a state where it is about to pay a huge cost, and then the user modifies that version repeatedly, each time paying the huge cost.

The solution to both problems is given by a new general technique for global rebuilding of a fully persistent data structure. The classic global rebuilding technique from ephemeral data structures, where the data structure rebuilds itself from scratch whenever its size changes by a constant factor, does not apply in the persistent context. Like the second problem above, we cannot charge the linear rebuild cost to the elements that changed the size, because such elements might get charged many times, even with de-amortized global rebuilding. Nonetheless, we show that careful choreography of specific global rebuilds works in the fully persistent context. As a result, we improve fully persistent arrays to support operations in $O(\lg\lg\Delta)$ time and space, where Δ is the current size of the array, matching the $\Omega(\lg\lg\Delta)$ lower bound. We also surmount the amortization and randomization issues with this global rebuilding technique.

2 Locality-Sensitive Functional Data Structure

Our first functional data structure represents a trie T with a set F of $O(1)$ fingers $f_1, f_2, \ldots, f_k$ while supporting finger movements, leaf insertion and deletion, and subtree copies and removals in $O(\lg \Delta)$ time and space per operation. This data structure is essentially a generalization of the *zipper* idiom from functional programming; see [Has08].

Theorem 1 *There is a functional trie data structure supporting movement of $O(1)$ fingers, leaf insertion and deletion, and subtree copies and removals in $O(\lg \Delta)$ time and space, where Δ is the total degree of the nodes directly involved.*

First we need some structural terminology; refer to Figure 5. Let T' be the Steiner tree with terminals f_i, that is, the union of shortest paths in T between all pairs of fingers. Let PF be the set of nodes with degree at least 3 in T' that are not in F, which we call *prosthetic fingers*. For convenience, we also define the root of the trie T to be a prosthetic finger in PF. Together, $F' = F \cup PF$ consists of all *pseudofingers*. Note that $|F'| \leq 2k + 1 = O(1)$, so we can afford to maintain all pseudofingers instead of just fingers. The *hand* T'' is the compressed Steiner tree obtained from the Steiner tree T' by contracting every vertex not in F' (each of degree 2).[5] A *tendon* is the shortest path in T' connecting two pseudofingers that are adjacent in T''. A *knuckle* is a connected component of T after removing all its pseudofingers and tendons.

[5] Our hand structure is unrelated to the hand data structure of [BMW03].

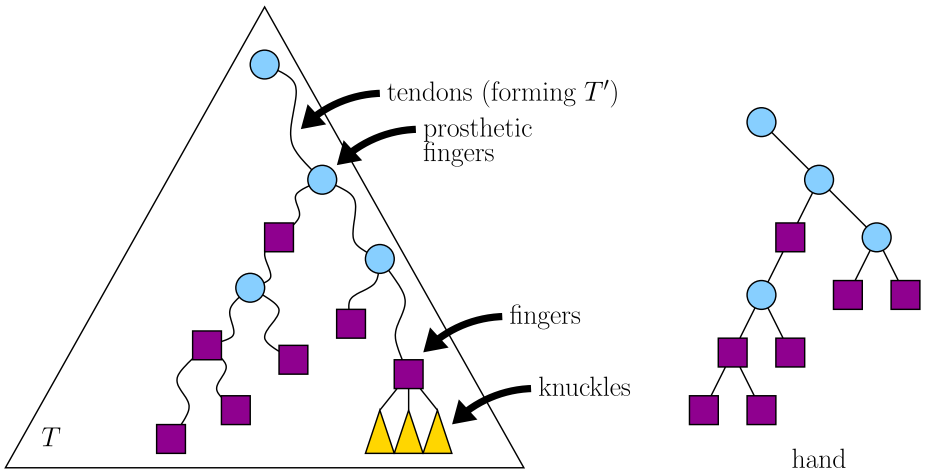


Figure 5: Fingers, prosthetic fingers, tendons, knuckles, and the hand.

At the top level, the functional trie data structure of Theorem 1 stores the constant-size hand T'', with each node representing a pseudofinger in F'. Each pseudofinger then stores (pointers to) the $O(1)$ incident tendons. Each tendon is represented by a deque of the nodes along the tendon, including the two pseudofingers at the ends. Each node of the Steiner tree T' (i.e., all pseudofingers and tendon nodes) stores a balanced search tree of its $O(\Delta)$ adjacent knuckles, called a *knuckle tree*. Each knuckle is represented as the rooted trie of its nodes, with each node storing its up to Δ children in a balanced search tree. (Here we use that knuckles are rooted subtrees, which follows from our inclusion of the root of T as a prosthetic finger.)

It remains to show how to perform update operations and how to move fingers. When performing an update at a finger, we modify its knuckle tree, adding a neighboring knuckle for a leaf insertion or subtree copy, and deleting a neighboring knuckle for a leaf deletion or subtree removal. In the case of subtree removal, we may also have to delete an incident tendon and any descendant pseudofingers. Then we use the path-copying technique on both that knuckle tree and the hand. To move a finger, we essentially transfer vertices between its neighboring knuckles and tendons:

1. If a finger enters a neighboring knuckle (in its knuckle tree), moving the finger to its new position might involve extracting the new finger from a deque and modifying a constant number of neighboring knuckles in its knuckle tree. We have two cases:

   (a) If the finger has degree 1 in T'', then it is attached to exactly one tendon. We insert into this tendon the vertex at the previous position of the finger.

   (b) If the finger has degree 2 or more in T'', then after the move that vertex has degree at least 3 and we create a new prosthetic finger at that position. We must then modify the hand so that the tendons that were incident to the finger now point to the prosthetic finger. This modification costs $O(1)$.

2. If a finger moves along a tendon, we proceed similarly, but now the previous finger either transfers to a neighboring knuckle or tendon, or becomes a new prosthetic finger. We extract the new vertex for the finger from the tendon, or if the tendon is empty, two fingers become equal.

The operations modify the hand, which has constant size, and change only $O(1)$ nodes from the knuckle trees and the deques, for a total cost of $O(\lg \Delta)$. By traversing the old and new hands, we can upgrade all of the $O(1)$ pseudofingers to point to their representations in the new version, ensuring that pseudofingers are always current.

3 Globally Balanced Functional Data Structure

Our second functional data structure represents the trie as a balanced binary tree, then makes this tree functional via path copying. Specifically, we will use a balanced representation of tries similar to link-cut trees of Sleator and Tarjan [ST83]. This representation is natural because the link and cut operations are essentially subtree copy and delete.[6] Sleator and Tarjan's original formulation of link-cut trees cannot be directly implemented functionally via path copying, and we explain how to address these issues in Section 3.1. In addition to being able to modify the trie, we need to be able to navigate this representation as we would the original trie. We discuss how to navigate in Section 3.2. In the end, we obtain the following result:

Theorem 2 *There is a functional trie data structure that supports movement of $O(1)$ fingers in $O(\lg \Delta)$ time (and zero space), where Δ is the total degree of the nodes directly involved; and supports leaf insertion and deletion and subtree copies and removals in $O(\lg n)$ time and space, where n is the number of nodes in the modified version.*

A key element of our approach is the *finger*. In addition to the core data structure representing a trie, we also maintain a constant number of locations in the trie called fingers. A finger is a data structure in itself, storing more than just a pointer to a node. Roughly speaking, a finger consists of pointers to all ancestors of that node in the balanced tree, organized to support several operations for computing nearby fingers. These pointers contrast with nodes in the balanced tree representation, which do not even store parent pointers. Modifications to our data structure must preserve fingers to point to the same location in the new structure, but fortunately there are only a constant number of fingers to maintain. Section 3.4 details the implementation of fingers.

3.1 Functional Link-Cut Trees

In the original link-cut trees [ST83], nodes store pointers to other nodes outside of their subtree, which prevents us from applying path copying. We show how to modify link-cut trees to avoid such pointers and thereby obtain functional link-cut trees via path copying.

There are multiple kinds of link-cut trees; we follow the worst-case logarithmic link-cut trees of [ST83, Section 5]. These link-cut trees decompose the trie into a set of "heavy" paths, and represent each heavy path by a globally biased binary tree [BST85], tied together into one big tree which we call the *representation tree*. An edge is *heavy* if more than half of the descendants of the parent are also descendants of the child. A *heavy path* is a contiguous sequence of heavy edges. Because any node has at most one heavy child, we can decompose the trie into a set of heavy paths, connected by *light* (nonheavy) edges. Following a light edge decreases the number of nodes in the subtree by at least a factor of two, so any root-to-leaf path intersects at most $\lg n$ heavy paths. The *weight* w_v of a node v is 1 plus the number of descendants of v through a light child of v. The depth of a

[6] Euler-tour trees are simpler, but linking multiple occurrences of a node in the Eulerian tour makes path copying infeasible.

node v in its globally biased search tree T is at most $\lg\left[\left(\sum_{u \in T} w_u\right)/w_v\right]$. If the root-to-node path to a node v intersects k heavy paths and w_i is the weight of the last ancestor of v in the ith heavy path down to v, then the total depth of v is $\lceil \lg(n/w_1) \rceil + \lceil \lg(w_1/w_2) \rceil + \cdots + \lceil \lg(w_{k-1}/w_k) \rceil$, which is $O(\lg n)$ because $w_k \geq 1$ and $k \leq \lg n$.

Link-cut trees augment each node in a globally biased search tree to store substantial extra information. Most of this information depends only on the subtree of the node, which fits the path-copying framework. The one exception is the parent of the node, which we cannot afford to store. Instead, we require a parent operation on fingers, which returns a finger pointing to the parent node in the representation tree. For this operation to suffice, we must always manipulate fingers, not just pointers to nodes. This restriction requires one other change to the augmentation in a link-cut tree. Namely, instead of storing pointers to the minimum (leftmost) and maximum (rightmost) descendants in the subtree, each node stores *relative fingers* to these nodes. Roughly speaking, such a relative finger consists of the nodes along the path from the node to the minimum (maximum). We require the ability to concatenate partial fingers, in this case, the finger of the node with a relative finger to the minimum or maximum, resulting in a (complete) finger to the latter.

To tie together the globally biased search trees for different heavy paths, the link-cut tree stores, for each node, an auxiliary globally biased tree of its light children. More precisely, leaves of the auxiliary tree point to (the root of) the globally biased search tree representing the heavy path starting at such light children. In addition, the top node in a heavy path stores a pointer to its parent in the trie, or equivalently, the root of the tree of light children of that parent. We cannot afford to store this parent pointer, so instead we define the finger to ignore the intermediate nodes of the auxiliary tree, so that the parent of the current finger gives us the desired root. Nodes in an auxiliary tree also contain pointers to their maximum-weight leaf descendents; we replace these pointers with relative fingers. Sleator and Tarjan's link-cut trees order auxiliary trees by decreasing weight; this detail is unnecessary, so we instead order the children by key value to speed up navigation.

With these modifications, the link-cut trees of Sleator and Tarjan can be implemented functionally with path copying. To see that path copying induces no slowdown, note that although the finger makes jumps down, the finger is shortened only one node at a time. Each time we take the parent of a finger, the node at the end can be rebuilt from the old node and the nodes below it in constant time. Furthermore, because the maximum finger length is $O(\lg n)$, at the end of the operation, one can repeatedly take the parent of the finger to get the new root. Another way to see that path copying induces no slowdown is that the link-cut trees in the original paper also involved augmentation, so every ancestor of a modified node was already being rebuilt.

These functional link-cut trees let us support modification operations in $O(\lg n)$ time given fingers to the relevant nodes in our trie. It remains to see how to navigate a finger, the topic of Section 3.2, and how to maintain other fingers as the structure changes, the topic of Section 3.3.

3.2 Finger Movement

In this section, we describe three basic finger-movement operations: finding the root, navigating to the parent, and navigating to the child with a given label. In all cases, we aim simply to determine the operations required of our finger representation.

The root of the trie is simply the minimum element in the topmost heavy path, a finger for which is already stored at the root of the representation tree. Thus we can find the root of the trie in constant time.

The parent of a node in the trie is the parent of the node within its heavy path, unless it is the top of its heavy path, in which case it is the node for which this node is a light child. Equivalently, the parent of a node in the trie is the predecessor leaf in the globally biased search tree, if such a predecessor exists, or else it is the root of the auxiliary tree containing the node. We also defined the parent operation on fingers to solve the latter case. For the former case, we require a predecessor operation on a finger that finds the predecessor of the node among all ancestors of the node. If this operation takes constant time, then we can find the predecessor leaf within the globally biased search tree by taking the maximum descendant of the predecessor ancestor found by the finger.

A child of a node is either the node immediately below in the heavy path or one of its light children. The first case corresponds to finding the successor of the node within its globally biased search tree. Again we can support this operation in constant time by requiring a successor operation on a finger that finds the successor of the node among all ancestors of the node. Thus, if a child stores the label of the edge above it in the trie, we can test whether the desired child is the heavy child. Otherwise, we binary search in the auxiliary search tree of light children in $O(\lg \Delta)$ worst-case time. Because the total depth of a node among all trees of light children is $O(\lg n)$, the total time to walk to a node at depth d can be bounded by $O(d + \lg n)$ as well as $O(d \lg \Delta)$.

3.3 Multiple Fingers

This section describes how to migrate the constant number of fingers to the new version of the data structure created by an update, in $O(\lg n)$ time. We distinguish the finger at which the update takes place as *active*, and call the other fingers *passive*. Just before performing the update, we decompose each passive finger into two relative fingers: the longest common subpath with the active finger, and the remainder of the passive finger. This decomposition can be obtained in $O(\lg n)$ time given constant-time operation to remove the topmost or bottommost node from a relative finger. First, repeatedly remove the topmost nodes from both the active and passive fingers until the removed nodes differ, resulting in the remainder part of the passive finger. Next, repeatedly remove the bottommost node of (say) the active finger until reaching this branching point, resulting in the common part of the passive finger. Now perform the update, and modify the nodes along the active finger according to path copying. We can find the replacement top part of the passive finger again by repeatedly removing the bottommost node of the active finger; the remainder part does not need to change. Then we use the concatenate operation to join these two relative fingers to restore a valid passive finger. Of course, we perform this partition, replacement, and rejoin trick for each relative finger.

3.4 Finger Representation

Recall that each finger to a node must be a list of the ancestors in the representation tree of that node supporting: push and pop (to move up and down); concatenate (to follow relative fingers); inject (to augment relative fingers based on a node's children); eject (to allow for multiple fingers); parent outside path (for faster parent query); and predecessor and successor among the ancestors of the node (to support parent and child in the trie). Our finger must also be functional, because nodes store relative fingers.

As a building block, catenable deques support all the operations we want except for predecessor or successor. Moreover, functional catenable deques have been well researched, with Okasaki [Oka98] giving a fairly simple $O(1)$ method that is amortized but in a way that permits confluent usage. Furthermore, Kaplan and Tarjan [KT95] have shown a complicated $O(1)$ worst-case purely functional implementation of catenable deques.

In order to implement predecessor and successor queries, we decompose the path in the representation tree into a sequence of right paths (sequence of nodes where the next element on the path is the right child) and left paths (sequence of nodes where the next element on the path is the left child). Then, the predecessor of a node among its ancestors is the last element of the last right path. The successor of a node among its ancestors is the last element of the last left path.

Instead of maintaining the finger as one catenable deque, we represent the finger as a deque of catenable deques, alternating right and left paths; see Figure 6. Then, because the last right path (or left path, respectively) is either the ultimate or penultimate deque in the sequence, its last element can be retrieved in $O(1)$ time. All other operations of the standard catenable deque can be simulated on the deque of deques with only $O(1)$ overhead. We thus obtain a structure that supports all of the operations of normal catenable deques of nodes, predecessor and successor in $O(1)$ time.

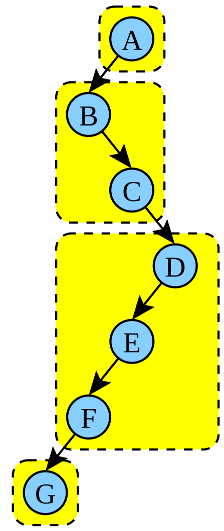


Figure 6: A finger to G is represented by G and a deque of (outlined) deques of ancestors of G.

This suffices to describe the basic data structure for functional tries. Their query time is $O(\lg \Delta)$ for navigation downward, $O(1)$ for navigation up, and $O(\lg n)$ for updates.

4 Adding Hash Tables

Our data structures above take $O(\lg \Delta)$ to move a finger to a child node. We can improve this bound to $O(\lg \lg \Delta)$ by introducing fully persistent hash tables. The resulting data structures are confluently persistent, but no longer functional.

For our first structure (Section 2), we show in Section 5.3 below how to construct a fully persistent hash table on n elements that performs insertions, deletions, and searches in $O(\lg \lg n)$ expected amortized time. Using this structure instead of the balanced trees of neighboring vertices at every vertex, the time and space cost of updates and finger movements improves to $O(\lg \lg \Delta)$ expected amortized.

Theorem 3 *There is a confluently persistent trie data structure supporting movement of $O(1)$ fingers, leaf insertion and deletion, and subtree copies and removals in $O(\lg \lg \Delta)$ amortized expected time and space, where Δ is the total degree of the nodes directly involved.*

We can also use this method to improve our second structure (Section 3.1). Given a set of elements with weights $w_1, \ldots, w_n$, we develop in Section 5.4 below a weight-balanced fully persistent hash table with $O(\lg \frac{\sum_j w_j}{w_e})$ expected amortized time modification, $O(\lg \lg n)$ find, and $O(\lg n)$ insert and delete, where n is the number of elements in the version of the hash table being accessed or modified and w_e is the weight of the element being accessed.

To use a hash table to move a finger down the trie, we modify each node of our data structure to include a hash table of relative fingers to light children in addition to the binary tree of light children. The binary tree of light children is necessary to support quickly recomputing the heavy child on an update. Except for inserts and deletes, the hash table achieves at least as good time bounds as the weight-balanced tree, and each trie operation involves at most $O(1)$ inserts or deletes, so we can maintain both the table and tree in parallel without overhead. As a result, updates still take $O(\lg n)$ time, moving the finger up still takes $O(1)$ time, and moving the finger down now takes $O(\lg \lg \Delta)$ time where Δ is the degree of the node before the move. The hash tables depend on fully persistent arrays, whose bounds are amortized expected, so updates and moving a finger down become amortized expected bounds.

Theorem 4 *There is a confluently persistent trie data structure that supports movement of $O(1)$ fingers in $O(\lg \lg \Delta)$ amortized expected time (and zero space), where Δ is the total degree of the nodes directly involved; and supports leaf insertion and deletion and subtree copies and removals in $O(\lg n)$ amortized expected time and space, where n is the number of nodes in the modified version.*

5 Fully Persistent Hash Tables

A natural starting point for fully persistent hash tables is Dietz's fully persistent array [Die89]. This data structure stores n distinct integers between 1 and u, viewed as an array of size u with special *empty* entries that occupy no space. If m is the total number of operations performed on the array, the structure occupies $O(m)$ space, and supports inserting, deleting, modifying, or reading the item at a given index, and finding the predecessor nonempty cell from a given index, in $O(\lg \lg(um))$ expected time.

A fully persistent hash table data structure follows immediately by storing items in a fully persistent array with 2-universal hashing [CW77, CLRS01] (where the probability of two elements colliding is $O(1/u)$). If we set $u = \Omega(n^4)$, then the probability of any of the n^2 pairs colliding is $O(n^2/n^4) = O(1/n^2)$. Thus globally rebuilding whenever any pair collides induces an expected cost of $O(1/n)$ time per operation (with an oblivious adversary), even for initializing the table with n items.

This data structure has two related flaws that make it unsuitable for speeding up trie nodes. First, we need to know n ahead of time in order to set u correctly. If n changes by more than a constant factor, we cannot afford to globally rebuild the data structure, because this nonrandom event can be exploited by an adversary; in general, persistence and amortization do not mix, as mentioned in the introduction. Second, even if we can surmount dynamically changing n (perhaps by randomly choosing when to rebuild), the $O(\lg \lg(um))$ time bound depends not only on the current size n (and u) but also on the total number m of operations. Thus, if the table has changed many times (e.g., n is large in some version), then operations are expensive even if n is small in the version of interest.

In this section we show how to fix both of these problems with an improved fully persistent hash table that supports dynamically changing n, and executes operations on a version storing n items in $O(\lg \lg n)$ amortized expected time. Our data structure uses the simple fixed-size hash table described above, combined with a new technique for speeding up fully persistent data structures via controlled rebuilding. We begin by describing the general technique, and then see how it can be applied to hashing.

5.1 Persistent Rebuilding Technique

Suppose we have a fully persistent data structure that can only be built for a specific *capacity* N, meaning that it can handle only $O(N^2)$ versions each of *size* $n = \Theta(N)$. We call such a data structure *capacitated*. Here we assume that each update operation changes the size n of a version by at most ± 1.

Suppose also that we can implement an operation $rebuild(D, v)$ that builds a new data structure D' representing just a single (root) version, specified by version v of existing data structure D, of capacity N equal to the size of that version. For example, the rebuild operation can simply create a new data structure of appropriate capacity and insert the N items from v into the structure; we allow other possible implementations of rebuild, in case the data structure must really have $n = \Theta(N)$ elements at all times (forbidding an empty structure) or the data structure has no insert operation. We require that the rebuilt data structure D' is deterministically identical to D in the sense that performing any sequence of operations on D starting at version v, including further rebuilds but forbidding measuring the capacity N, results in the same query outputs as performing that sequence on D' from the root version.

Theorem 5 *Suppose we are given a capacitated fully persistent data structure that, for storing $m = O(N^2)$ versions with capacity N, uses $O(s(m))$ space and supports updates in $O(f(N))$ amortized expected time, queries in $O(q(N))$ time, and rebuilds in $O((m + N)\, f(m + N))$ expected time, for some increasing functions f, q, s. Then we can construct an equivalent (noncapacitated) fully persistent data structure for storing m versions that uses $O(s(m))$ space and supports updates in $O(f(\alpha\, n))$ expected time and queries in $O(q(\alpha\, n))$ time in a version of size n, for some constant α.*

Proof: The data structure we construct consists of several capacitated data structures, called *substructures*; refer to Figure 7. Substructures come in pairs, one *pristine* and one *active*. A pristine substructure of capacity N stores (copies of) $\Theta(N)$ versions specifically for later rebuilding; they are not used for updates or queries. The corresponding active substructure of capacity N stores (copies of) the same $\Theta(N)$ versions, called *primary versions*, and in addition $O(N^2)$ *secondary versions*, forming a tree of $O(N)$ versions rooted at each primary version. Each secondary version in such a tree stores a pointer to the corresponding primary version, which in turn stores a pointer to the corresponding version in the pristine substructure. Each primary version also stores a linked list of the updates, along with the version numbers they update, that form the secondary versions in the attached subtree (enabling future replay). Each version of the overall data structure stores a pointer to the active substructure representing it, as well as the local version number within that substructure.

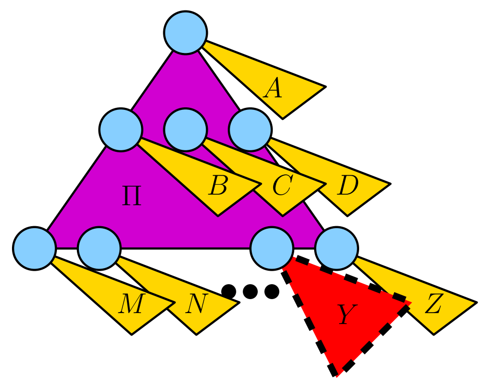


Figure 7: The tree of versions in an active substructure. Here Π is the tree of primary versions, as stored by the pristine substructure, and does not grow during updates. Each node in Π is the root of a tree (A, B, $\ldots$, Z) of secondary versions that grows during updates. When a secondary tree grows large enough (as represented by Y), a new active substructure is created to replace it.

An update to the overall data structure first updates the active substructure representing the version being updated, creating a new secondary version. We can set the primary pointer for the new secondary version to match that of the updated version (where primary versions point to themselves). Then we also append this operation and its local version number to the list associated with that primary version. If the length of this list becomes at least some constant $c > 0$ times the size of the primary version, then we trigger an overflow of that primary version.

When a primary version overflows, we create two new identical substructures by the following process (executed twice): rebuild the corresponding version in the pristine substructure, and replay the updates stored in the linked list. We mark one of these new substructures pristine and the other active, and we cross-link corresponding versions. The root version of the new active substructure replaces the old primary version, so we change the pointers from the corresponding version of the overall data structure; similarly, the other versions in the new active substructure replace the old descendant secondary versions. In particular, we can empty the linked list associated with the old primary version, and we could delete those now-obsolete versions, except that we do not assume that substructures support version deletion. The new versions in the active substructure are all primary versions, with initially empty linked lists; in effect, we are promoting secondary versions to primary versions. As a result, the number of primary versions in a substructure of capacity N is always $c\,N + O(1)$.

The cost of overflowing in a substructure of capacity N is $O(N\,f(N))$, because the pristine substructure from which we rebuild stores only $O(N)$ versions (unlike the active substructure) and the linked list of secondary updates has size $\Theta(N)$. We can charge this cost to the decrease in the total size of all linked lists, which is $\Theta(N)$, resulting in an amortized cost of $O(f(N))$. The query cost is identical up to the additive constant cost of finding the appropriate substructure. Furthermore, if we choose $c < \frac{1}{2}$, then the maximum depth of any version in a substructure is at most $2c\,N$ which is a constant factor less than N, implying that all versions have size $n = \Theta(N)$. Thus all bounds can be rewritten in terms of n instead of N. The space grows by only a constant factor because each version appears in at most two active substructures, once as a secondary version and one as a primary version, and in at most two corresponding pristine substructures. □

5.2 Fully Persistent Arrays

Applying Theorem 5 to the fully persistent arrays of Dietz [Die89], we obtain the following improvement from $O(\lg\lg(um))$ expected time per operation:

Corollary 6 *There is a fully persistent array data structure storing n distinct integers between 1 and $u = n^{O(1)}$ that uses $O(m)$ space, where m is the total number of operations performed on the array, and supports inserting, deleting, modifying, or reading the item at a given index, and finding the predecessor nonempty cell from a given index, in $O(\lg\lg n)$ expected time.*

5.3 Fully Persistent Hash Tables

Applying Theorem 5 to the fully persistent hash tables described at the beginning of Section 5, using 2-universal hashing on top of fully persistent arrays, we obtain the following result:

Corollary 7 *There is a fully persistent hash table data structure supporting insertion, deletion, and search in $O(\lg\lg n)$ amortized expected time, where n is the number of items in the queried version. The data structure occupies $O(m)$ space, where m is the total number of operations performed on the data structure.*

Here the rebuild operation designs a hash function for a table of size $\Theta(N)$.

5.4 Weight-Balanced Hash Tables

This section describes a fully persistent weight-balanced hash table, which form a suitable replacement for the auxiliary globally biased trees of light children in the globally balanced functional data structure. This data structure uses the fully persistent hash tables from the previous section combined with ideas from the working-set structure of Iacono [Iac01].

Theorem 8 *There is a fully persistent weight-balanced hash table storing n elements, each with a key and a weight, subject to looking up an element by key in $O(\lg \lg r)$ time, where r is the number of elements at least as heavy as the element in the query version; modifying an element with specified key in $O(\lg r)$ time; and inserting or deleting an element in $O(\lg n)$ time.*

First we build a hybrid data structure by maintaining, in parallel, a weight-balanced tree keyed on weight and a hash table by element key. This fully persistent data structure stores n elements subject to inserting or deleting an element with specified key and weight in $O(\lg \lg n + \lg r)$ time, where r is the number of elements at least as heavy as the specified element; finding an element with specified key in $O(\lg \lg n)$ time; and finding the minimum and maximum values in $O(1)$ time. Our weight-balanced hash table is a list of these hybrid structures, where the ith structure of the list contains $2^{2^{2^i}}$ elements (except the last, which may contain fewer) and every element in the ith structure is lighter than every element in the preceding structures.

To find an element with specified key, we search in each successive structure in the list until we find the desired element. If we find the element in the ith structure, the running time is $\sum_{j=1}^{i} O(\lg \lg 2^{2^{2^j}}) = O(\lg \lg 2^{2^{2^i}})$. Equivalently, the operation takes $O(\lg \lg r)$ time where r is the number of elements at least as heavy as the desired element.

Next we argue that changing the weight of an element with specified key takes $O(\lg r)$ time, where r is the number of elements heavier than the element either before or after the modification. To decrease the weight rank from r to r', where rank r' appears in the ith structure and rank r appears in the jth structure, we insert the new element into the ith structure, delete the old element from the jth structure, then walk down the list from i to $j-1$, removing the maximum-weight element from the structure and inserting it into the next. Deleting the old element and inserting the new element take $O(\lg r)$ and $O(\lg r')$ time, respectively; walking down the list to restore the sizes takes $\sum_{k=i}^{j-1} O\left(\lg 2^{2^{2^k}}\right) + O(1) = O\left(\lg 2^{2^{2^{j-1}}}\right) = O(\lg r)$ time, because inserting a maximum-weight element into any structure takes $O(1)$ time; and walking along the list to find the element takes $O(\lg \lg r)$ time; for a total of $O(\lg r)$ time. Increasing the rank of an element works analogously, except that elements are shifted in the other direction to restore structure sizes.

Next we consider insertions and deletions. To insert an element with specified key and weight, we first insert it with weight $-\infty$, which takes $O(\lg n)$ time: $O(\lg \lg n)$ to get to the last structure in the list, then $O(\lg n)$ to insert into this structure or $O(1)$ to create a new structure. Then, as argued above, reweighting to the correct weight takes $O(\lg n)$ time. Similarly, to delete an element with specified key and weight, we first reweight it to have weight $-\infty$, and then extract it from the last structure, in $O(\lg n)$ time.

Finally, because $\sum_j w_j \geq r_i w_i$, modifications to element e take place in $O\left(\lg \frac{\sum_j w_j}{w_e}\right)$ time, so this data structure can do modifications at least as fast as weight-balanced trees.

6 Open Problems

It would be interesting to combine our two functional data structures into one that achieves the minimum of both performances. In particular, it would be nice to be able to modify a node at depth d in $O(\min\{d \lg \Delta, \lg n\})$ time and space. One approach is to develop modified globally biased binary tree where the depth of the ith smallest node is $O(\min\{i, w_i\})$ and supporting fast splits and joins.

Acknowledgments

We thank Patrick Havelange for initial analysis of and experiments with version control systems such as Subversion.

References

[BMW03] Guy E. Blelloch, Bruce M. Maggs, and Shan Leung Maverick Woo. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 373–383, Baltimore, Maryland, January 2003.

[BST85] Samuel W. Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[CW77] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proc. 9th Annual ACM Symposium on Theory of Computing*, pages 106–112, 1977.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. *SIGMETRICS Perform. Eval. Rev.*, 27(1):59–70, 1999.

[Die89] Paul F. Dietz. Fully persistent arrays (extended abstract). In *Proc. Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74, 1989.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989. Originally appeared in STOC'86.

[DST94] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *J. ACM*, 41(5):943–959, 1994.

[FK03] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *J. Algorithms*, 48(1):16–58, 2003.

[Has08] HaskellWiki. Zipper. http://www.haskell.org/haskellwiki/Zipper, 2008.

[Iac01] John Iacono. Alternatives to splay trees with o(log n) worst-case access times. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 516–522, 2001.

[KM83] Timo J. G. Krijnen and Lambert Meertens. Making B-trees work for B. Mathematical Centre Report IW 219, Mathematisch Centrum, Amsterdam, The Netherlands, 1983.

[KT95] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 93–102, 1995.

[Mit03] Michael Mitzenmacher. Dynamic models for file sizes and double Pareto distributions. *Internet Math.*, 1(3):305–333, 2003.

[Mye82] Eugene W. Myers. AVL dags. Technical Report 82-9, Department of Computer Science, University of Arizona, Tucson. Arizona, 1982.

[Mye84] Eugene W. Myers. Efficient applicative data types. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, Salt Lake City, Utah, January 1984.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[PCD05] Dan R. K. Ports, Austin T. Clements, and Erik D. Demaine. PersiFS: A versioned file system with an efficient representation. In *Proc. 20th ACM Symposium on Operating Systems Principles*, October 2005.

[Pip97] Nicholas Pippenger. Pure versus impure lisp. *ACM Trans. Program. Lang. Syst.*, 19(2):223–238, 1997.

[PT07] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 555–564, 2007.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[ST86] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–677, July 1986.

[Swa85] Garrett Swart. Efficient algorithms for computing geometric intersections. Technical Report 85-01-02, Department of Computer Science, University of Washington, Seattle, Washington, 1985.