Blame Trees

Erik D. Demaine[1], Pavel Panchekha[1], David A. Wilson[1], Edward Z. Yang[2]

[1] Massachusetts Institute of Technology, Cambridge, Massachusetts
{edemaine,pavpan,dwilson}@mit.edu
[2] Stanford University, Stanford, California
ezyang@cs.stanford.edu

Abstract. We consider the problem of merging individual text documents, motivated by the single-file merge algorithms of document-based version control systems. Abstracting away the merging of conflicting edits to an external conflict resolution function (possibly implemented by a human), we consider the efficient identification of conflicting regions. We show how to implement tree-based document representation to quickly answer a data structure inspired by the "blame" query of some version control systems. A "blame" query associates every line of a document with the revision in which it was last edited. Our tree uses this idea to quickly identify conflicting edits. We show how to perform a merge operation in time proportional to the sum of the logarithms of the shared regions of the documents, plus the cost of conflict resolution. Our data structure is functional and therefore confluently persistent, allowing arbitrary version DAGs as in real version-control systems. Our results rely on concurrent traversal of two trees with short circuiting when shared subtrees are encountered.

1 Introduction

The document-level merge operation is a fundamental primitive in version control systems. However, most current implementations of this operation take linear time in the size of the document, and rely on the ability to identify the least common ancestor of the two revisions to be merged. For large documents, we can improve on this naïve bound by not spending time on non-conflicting portions of the document. More abstractly, the lowest common ancestor may not be unique, or even exist, in a fully confluent setting. In this paper, we describe the practical motivation for this problem, our model of the theoretical problem, and our solution.

Document merge. Single-document merging forms the core of many modern version control systems, as it is critical for reconciling multiple, concurrent branches of development.

Single-document merging has been implemented in a variety of different ways by different version control systems. The most basic merge strategy is the three-way merge, as implemented by Git, Mercurial, and many other version control systems. In a three-way merge, three revisions of the file are specified: the "source" version, the "target" version, and the "base" version, which is the least common ancestor of the source and target. Any intermediate history is thrown out, and the result of the merge relies on the diffs between the base and source and between the base and target. These diffs are split into changed and unchanged segments, which are then used to build the new document.

There are a few elaborations on the basic three-way merge. In the case of multiple least common ancestors, Git will recursively merge the common ancestors together, and use those as the base version.[3] Additionally, there is some question of whether two chunks that have applied an identical change (as opposed to conflicting ones) should silently merge together: most systems opt for not reporting a conflict. This behavior has lead to some highly publicized edge cases in the merge algorithm [3].

A more sophisticated merging algorithm is implemented by Darcs. It uses whether two patches commute as the test for whether a merge conflict should be generated; and it performs the merge patch by patch. Because Darcs uses the intermediate history, this often results in a higher quality merge, but requires time at least linear in the number of patches, and can result in exponential behavior in some cases.

An interesting but largely obsolete representation for an entire history which was implemented by the SCCS and BitKeeper systems is the "Weave" [9], where every document in the repository contains all lines of text present in any revision of the document, with metadata indicating what revisions they correspond to. Merging on this representation takes time proportional to the size of the entire history.

Prior work. Demaine, Langerman, and Price [4] considered the problem of efficient merge at the directory/file level, using confluently persistent data structures. They cite the document-level merge problem as "relatively easy to handle", assuming that the merge may take linear time. The goal of this work is to beat this bound.

In the algorithms community, merging usually refers to the combination of two sorted arrays into one sorted array. This problem can be viewed as similar, particularly if we imagine that equal-key items get combined by some auxiliary function. The standard solution to this problem (as in, e.g., mergesort) takes linear time in the input arrays. Adaptive merging algorithms [2,5,10,11] achieve the optimal bound of $\Theta(\sum_{i=1}^{k} \lg g_i)$ if the solution consists of g_1 items from the first set, then g_2 items from the second set, then g_3 items from the first set, etc. Our result is essentially a confluent data structure built around a dynamic form of this one-shot algorithm.

In this paper, we will refer repeatedly to the well-known results of the functional programming community [12], in particular confluence via purely functional data structures with path copying as the primary technique. The idea of

[3] This situation is rare enough in practice that very few other VCSes implement this behavior, although this strategy is reported to reduce conflicts in merges on the Linux kernel.

parametrizing an algorithm on a human-driven component is a basic technique of human-based computation; however, we do not refer to any of the results in that literature.

Theoretical model. A natural model of a text document in a version control system is a confluently persistent sequence of characters. Persistent data structures [7, 8] preserve old versions of a document as modifications are made to a document. While a fully persistent data structure permits both queries and modifications to old versions of document, a confluently persistent structure additionally supports a merge operation. Any two documents can be merged, which means that the version dependencies can form a directed acyclic graph (DAG). Our paper presents an implementation of this data structure which admits an efficient implementation of this merge operation.

For the purposes of our treatment, we consider a more expressive model of text documents as confluently persistent sorted associative maps (dictionaries), whose values are arrays of characters, and whose keys are an ordered data type supporting constant-time comparison and a split operation with the property $a < \text{split}(a, b) < b$. (Data structures for maintaining order in this way are well known [1, 6], and also common in maintaining full persistence [7].) The user can decide to divide the document into one entry (key/value pair) per character, or one entry per line, or some other level of granularity. The supported operations are then insertion, deletion, and modification of entries in the map, which correspond to equivalent operations on characters, lines, etc., of the document.[4] These operations take one (unchanged) version as input and produce a new version (with the requested change) as output. In addition, the merge operation takes two maps (versions) and a conflict-resolution function, which takes two conflicting submaps ranging between keys i and j and combines them into a single submap ranging between keys i and j. In order to amortize some costs of our approach, we assume that conflict resolution requires at least $\Theta(|M| + |N|)$ time, where $|M|$ and $|N|$ are the number of entries in the input submaps (our bounds do not hold for a $\Theta(1)$ resolution function); in practice, the cost of conflict resolution is likely to be $\Theta(k(|M| + |N|))$, where k is the average length of the sequences of characters inside the submaps.

Entries of these maps have one extra piece of metadata: a unique ID identifying the revision that this entry was last updated. We also assume that we have a source of unique IDs (occupying $O(1)$ space each), which can be used to allocate new revision numbers for marking nodes. In practice, cryptographically secure hash functions are used to generate these IDs.

Sharing. The performance of our document merge operation depends on the underlying structure of our documents; if common and conflicting regions are interleaved $\Theta(n)$ times, then we cannot hope to do any better than a linear-time merge. Thus, we define the *disjoint shared regions* $\mathcal{S}$ of our sorted maps to be

[4] It is worth emphasizing that the keys do *not* correspond to character-indexes or line-numbers; aside from order, they are completely arbitrary.

the set of maximal disjoint ranges which have the same contents (matching keys, values, and revision ID): these can be thought as the non-conflicting regions of two documents. Note that, if two independent editors make the same revision to a document, the resulting two entries are considered in conflict, as the revision IDs will differ.

It will also be useful to refer to the *non-shared entries* $\mathcal{N}$ of two documents, which is defined to be the set of all entries in either document that are contained in no $S \in \mathcal{S}$.

Let $f(|\mathcal{N}|)$ denote the cost of conflict resolution among the $|\mathcal{N}|$ non-shared entries, which we assume to be $\Omega(|\mathcal{N}|)$. In our analyses, we will often charge the cost of traversing conflicting entries to the execution of the conflict resolution function, when considering the overall cost of a merge.

Main result. Our main result is a functional (and thus confluently persistent) data structure supporting insert, delete, modifying, and indexing on a map of size n in $O(\log n)$ time per operation; and merging two (versions of) maps in $O(f(|\mathcal{N}|)+\sum_{S\in\mathcal{S}} \log |S|)$ time. For example, for a constant number of non-shared nodes, the merge cost is logarithmic; or more generally, for a constant number of shared regions, the merge cost is logarithmic plus the conflict resolution cost. We expect that this adaptive running time will be substantially smaller than the standard linear-time merge in most practical scenarios.

2 Blame Trees, Version 1

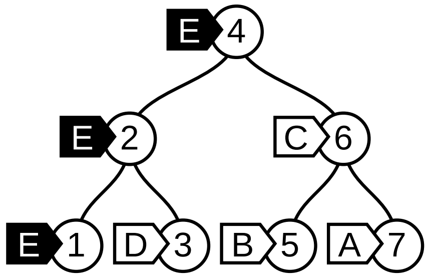

Fig. 1. An example blame tree, where the latest update was revision E made to key 1. The values on the leaves are omitted for clarity.

Blame trees represent a text document as a balanced binary search tree containing strings, augmented with length annotations, to facilitate efficient indexing into arbitrary locations of the document. (Because these annotations are not relevant for merges, we omit them from our presentation.) Blame trees are

```
Tree is either:
1: NODE(rev, key, val, left, right)
2: LEAF
```

Fig. 2. Our trees are classic binary search trees annotated with an extra revision field, indicating the last edit which affected a node or any node in its subtree.

further augmented with a revision `rev` annotation, which tracks the latest revision to the data structure which affected this node. Updates generate a fresh revision for the edit and record it on all nodes they touch.

For binary trees, disjoint shared regions correspond directly to disjoint shared subtrees, e.g., the set of maximum-sized disjoint subtrees which exist identically (matching keys, values and annotations) in both trees. Any shared region S can be represented by $O(\log |S|)$ shared subtrees. To simplify analysis, we will instead consider disjoint shared subtrees $\tilde{\mathcal{S}}$, and then translate our bounds back into disjoint shared regions. A useful fact which we will refer to repeatedly is that $|\tilde{\mathcal{S}}| \leq 2 \sum_{S \in \mathcal{S}} \log |S|$.

As any modifications to a node must modify all of its parent nodes, the definition of two shared subtrees in Figure 3 is equivalent (e.g., we only need to check roots of shared subtrees for equality).

```
1: function SHARED($a, b$)
2:     return $a$.key = $b$.key $\wedge$ $a$.rev = $b$.rev
3: end function
```

Fig. 3. The definition of two shared subtrees.

When the trees in question have identical structure, merging two blame trees is trivial: traverse both structures in-order and simultaneously. Because the structures are identical, traversals will be in lock-step, and we can immediately identify a shared subtree when we first encounter it, and skipping it entirely. We pay only the cost of visiting the root of every shared subtree, so the cost of traversal is $O(|\mathcal{N}| + |\tilde{\mathcal{S}}|)$, which in particular is $O(|\mathcal{N}| + \sum_{S \in \mathcal{S}} \log |S|)$. The overall cost is $O(f(|\mathcal{N}|) + \sum_{S \in \mathcal{S}} \log |S|)$, with the cost of traversing conflicting nodes charged to the conflict resolution function f.

This naïve traversal doesn't work, however, when the two tree have differing structures. Additionally, the conflict resolution may return a new subtree to be spliced in, and thus we need to manage rebalancing the resulting trees. Consequently, our general strategy for merging balanced search trees of different shapes will be to identify the disjoint shared subtrees of the two trees, split the trees into conflicting and shared regions, resolve the conflicting regions, and concatenate the trees back together. The core of our algorithm is this:

Lemma 1. *It is possible to determine the disjoint shared subtrees $\tilde{\mathcal{S}}$ of two balanced blame trees in time* $O(|\mathcal{N}| + \sum_{S \in \tilde{\mathcal{S}}} \log |S|) \subseteq O(|\mathcal{N}| + \sum_{S \in \mathcal{S}} \log^2 |S|)$.

```
function InOrder($a$)
    if $a$ is Node then
        InOrder($a$.left)
        $skip \leftarrow$ yield $a$                ▷ Suspend the coroutine to visit the node
        if $skip \neq$ Skip then
            InOrder($a$.right)
        end if
    end if
end function
```

Fig. 4. In-order traversal as a coroutine. Execution of this function proceeds normally until the **yield** a statement is reached; at this point, execution of the function is suspended and the value a is returned to the caller of the coroutine. When the coroutine is initially invoked, it returns a resumption continuation, which the caller can use to resume the execution of the coroutine. In our case, the resumption continuation requires the caller to provide a value *skip*, which indicates whether or not to skip traversal of the right subtree.

```
function Traverse($a, b$)
    $n_a, k_a \leftarrow$ InOrder(a)
    $n_b, k_b \leftarrow$ InOrder(b)
    while $n_a$, $n_b$ not Null do
        if Shared($n_a, n_b$) then                          ▷ $n_a = n_b$
            yield $n_a$                   ▷ Add $n_a$ to list of shared subtrees
            $n_a \leftarrow k_a$(Skip)                      ▷ Skip the right subtree
            $n_b \leftarrow k_b$(Skip)
        else if $n_a$.key $\leq n_b$.key then
            $n_a \leftarrow k_a$(NoSkip)
        else if $n_a$.key $> n_b$.key then
            $n_b \leftarrow k_b$(NoSkip)
        end if
    end while
end function
```

Fig. 5. Concurrent in-order traversal of two trees which reports shared subtrees. The algorithm begins by initiating in-order traversal on a and b, retrieving the left-most nodes n_a and n_b and the resumption continuations of the traversals k_a and k_b. The algorithm then repeatedly checks for shared subtrees, advancing the traversal with the lowest key, skipping right subtrees when a shared subtree is found.

Proof. Perform an in-order traversal concurrently on both trees, advancing the traversal on the tree with the lower key. This traversal can easily be expressed as a pair of coroutines, as seen in Figure 4 and Figure 5 and illustrated in Figure 6. If the two nodes being traversed are roots of shared subtrees, record the node as a shared subtree and skip traversal of the right child of both trees; continue traversal from the parent.

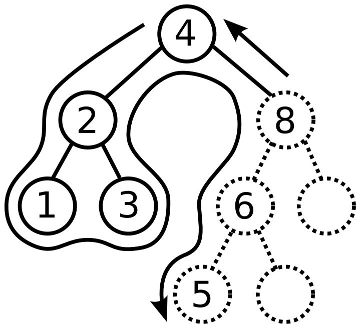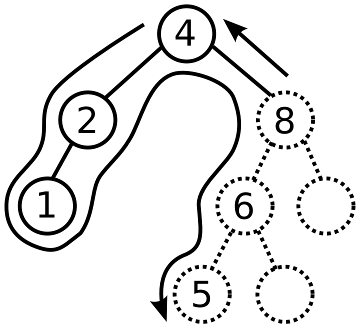

Fig. 6. An illustration of concurrent in-order traversal, where the dashed portion of the tree is shared. Both traversals start at 1 and progressively increase (the left tree stepping twice at key 3) until node 5 is reached: at this point, we discover the tree is shared. At this point, we walk back up the tree (5, 6, 8); each node is shared, so the right subtrees are skipped.

The resulting list r of shared subtrees will contain "runs" of shared nodes where $r[i] = r[i+1]$.left (that is, the parent of the shared subtree was also shared). Discard all shared subtrees except the final subtree of each run (as any such tree i is strictly contained in $i+1$), and return the resulting list.

Because in-order traversal returns nodes with monotonically increasing keys, it is easy to see that, without short circuiting, this traversal will discover all shared subtrees. Furthermore, because we only skip subtrees of shared subtrees (which must also be shared), it is easy to see that no shared subtrees are skipped.

An ordinary in-order traversal will take $O(n)$ time, so we need to show that with our short-circuiting we spend only $O(\log|S|)$ per shared subtree S. Suppose that we have accessed the leftmost node of a shared subtree, with cost $O(\log|S|)$; this must occur before any other nodes of the shared subtree are traversed, as we are doing in-order traversal. This node is itself a shared subtree (though not the maximal node), and we will short circuit to the parent. This will occur repeatedly for the entire path contained within the maximal shared subtree, so after another $O(\log|S|)$ steps, we reach the shared node rooting the maximal shared subtree, and continue traversal of the rest of the tree. Clearly only $O(\log|S|)$ total is spent through a shared subtree, for $O(\sum_{S\in\mathcal{S}}\log S)$.

To translate this bound from shared subtrees into shared regions, we observe that for any shared region S, the shared subtrees $\tilde{S}$ have the following property $O(\sum_{s\in\tilde{S}}\log|s|) \subseteq O(\log^2|S|)$ (recalling that only $O(\log|S|)$ subtrees are necessary to encode a region S, each with maximum size $|S|$). The bound $O(|\mathcal{N}| + \sum_{S\in\mathcal{S}}\log^2|S|)$ follows. □

From here, it is easy to implement merge in general:

Theorem 1. *Given the ability to split and concatenate a sequence M of blame trees in $O(\sum_M \log|M|)$ time, it is possible to merge two balanced blame trees with shared disjoint regions $\mathcal{S}$ in time $O(f(|\mathcal{N}|) + \sum_{S\in\mathcal{S}}\log^2|S|)$.*

Proof. We need to show that the cost of traversal and conflict resolution dominates the cost of splitting and concatenating trees; that is, that $O(\sum_M \log|M|) \subseteq O(f(|\mathcal{N}|) + \sum_{S\in\tilde{\mathcal{S}}} \log|S|)$. We split this bound into shared regions $\mathcal{S}$ and unshared regions $\mathcal{U}$. For $\mathcal{S}$, the cost contributed by each shared subtree $|S|$ is a $\log|S|$ factor better than the cost of traversal as stated in the lemma. For $\mathcal{U}$, we observe $\sum_{U\in\mathcal{U}} \log|U| \leq |\mathcal{N}|$, so the cost of splitting and concatenating unshared nodes can be charged to conflict resolution. □

3 Faster Traversal

We now show how to achieve a logarithmic speedup when considering a specific type of balanced binary tree, namely red-black trees. Our traversal time improves from $O(|\mathcal{N}| + \sum_{S\in\mathcal{S}} \log^2|S|)$ to $O(|\mathcal{N}| + \sum_{S\in\mathcal{S}} \log|S|)$. This method relies on level-order traversal. We first describe the algorithm for perfect binary trees, and then sketch how to apply this to red-black trees, which are not perfectly balanced, but are perfectly balanced on black nodes.

Lemma 2. *It is possible to determine the shared subtrees $\tilde{\mathcal{S}}$ of two perfect binary trees annotated with precise max-depth in time $O(|\mathcal{N}| + |\tilde{\mathcal{S}}|)$, e.g. $O(|\mathcal{N}| + \sum_{S\in\mathcal{S}} \log|S|)$.*

Proof. We maintain two queues, one for tree A and one for tree B. Each queue starts containing the root node of its respective tree. Without loss of generality, assume the max-depth of the two trees are equal (if they are not, recursively deconstruct the tree until you have a forest of correct depth; the nodes removed by the deconstruction are guaranteed not to be shared, because they have the wrong height.)

Claim. The list of elements extracted from each queue consists of the nodes of depth d whose parents were non-shared and had depth $d+1$, sorted in order of their key.

If the claim is true, we can perform a merge of sorted lists in linear time, and for any matching keys we check if the nodes are shared subtrees. (This is sufficient, as nodes with different max-depths or non-equal keys cannot be shared subtrees.) Finally, for all non-shared nodes add their children to their corresponding queue in order of the lists and repeat. The full algorithm is presented in Figure 8.

We first show that our claim is true by induction on d. The base case is trivial. Suppose that the algorithm has fulfilled the claim up until the current round d. We consider the possible sources of nodes of depth d; by the properties of a perfect binary tree, the parent of a node with max-depth d must have max-depth $d+1$. Furthermore, the node would not have been added if the parent were shared, as desired.

Observing that only non-shared nodes, or the roots of maximal shared subtrees ever enter the queue, if queue pop and push operations take $O(1)$, then

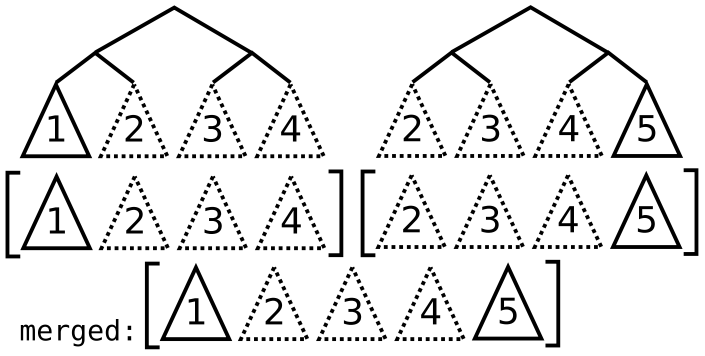

Fig. 7. An illustration of level-order traversal. Given the two trees shown above, this diagram shows the state of the two queues (in brackets) after finishing traversing the second level of the tree, and the resulting merged list of trees. Subsequent traversals will only traverse over subtrees 1 and 5. (As an optimization, we can note when subtrees are disjoint and immediately mark them as conflicts.)

time bound $O(|\mathcal{N}| + |\tilde{\mathcal{S}}|)$ follows easily; the alternate formulation of the bound falls easily out of the fact that any $S \in \mathcal{S}$ can only contribute $\log |S|$ roots of shared subtrees. $\square$

In general, most practical self-balancing binary search trees will not be perfect. However, in the case of red-black trees, the number of black nodes down any path of the tree is constant. So we can adapt the algorithm for perfect trees to only count black nodes towards depth: when we would add a red node into the queue, we instead push its two black children. This means that we do not ever check red nodes to see if they are shared subtrees, but this only adds a constant factor extra time on the analysis. Note that this technique does not work if the tree is not perfectly balanced in some fashion: without perfect balance, we will encounter nodes whose max-depths are much lower than the current max-depth which still must be queued. Switching our queue to a priority queue to accommodate these nodes would result in a logarithmic slowdown, destroying our bound.

Our final result follows easily:

Theorem 2. *It is possible to merge two red-black blame trees with shared disjoint regions $\mathcal{S}$ in time $O(f(|\mathcal{N}|) + \sum_{S\in\mathcal{S}} \log |S|)$.*

Proof. The analysis proceeds identically as our previous Theorem 1, except that the cost of splitting and concatenating the red-black trees is the same as the traversal in the case of shared regions $\mathcal{S}$. $\square$

Acknowledgments

This work began during an open-problem solving session for MIT's Advanced Data Structures class (6.851) in Spring 2012. Thanks to Anders Kaseorg, An-

```
function PushChildrenPop($Q$)
    $x \leftarrow$ Pop($Q$)
    Push($Q, x$.left)
    Push($Q, x$.right)
end function
function LevelTraverse($a, b$)
    $Q_a, Q_b \leftarrow$ Singleton($a$), Singleton($b$)
    while $\neg$(Empty($Q_a$) $\vee$ Empty($Q_b$)) do
        $n_a, n_b \leftarrow$ Peek($Q_a$), Peek($Q_b$)
        if Shared($n_a, n_b$) then
            yield $n_a$
            Pop($Q_a$)
            Pop($Q_b$)
        else if $n_a$.depth $> n_b$.depth then
            PushChildrenPop($Q_a$)
        else if $n_a$.depth $< n_b$.depth then
            PushChildrenPop($Q_b$)
        else if $n_a$.key $\leq n_b$.key then
            PushChildrenPop($Q_a$)
        else if $n_a$.key $> n_b$.key then
            PushChildrenPop($Q_b$)
        end if
    end while
end function
```

Fig. 8. Level-order traversal of two perfect trees.

drea Lincoln, and any unnamed participants of the session for contributing to this solution. This research is partially funded by the DARPA Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH) program, BAA-10-70, under contract #N66001-10-2-4088 (*Bridging the Security Gap with Decentralized Information Flow Control*).

References

1. Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164, Rome, Italy, September 2002.
2. Svante Carlsson, Christos Levcopoulos, and Ola Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9:629–648, 1993.
3. Bram Cohen. Git can't be made consistent. `http://bramcohen.livejournal.com/74462.html`, April 2011.
4. Erik D. Demaine, Stefan Langerman, and Eric Price. Confluently persistent tries for efficient version control. *Algorithmica*, 57(3):462–483, 2010.
5. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM*

Symposium on Discrete Algorithms, pages 743–752, San Francisco, California, January 2000.

6. Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, New York City, May 1987.
7. James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
8. Amos Fiat and Haim Kaplan. Making data structures confluently persistent. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms*, pages 537–546, Washington, DC, January 2001.
9. Greg Hudson. Notes on keeping version histories of files. `http://web.mit.edu/ghudson/thoughts/file-versioning`, October 2002.
10. Kurt Mehlhorn. *Data Structures and Algorithms*, volume 1 (Sorting and Searching), pages 240–241. Springer-Verlag, 1984.
11. Alistair Moffat, Ola Petersson, and Nicholas C. Wormald. A tree-based Mergesort. *Acta Informatica*, 35(9):775–793, August 1998.
12. Chris Okasaki. *Purely functional data structures.* Cambridge University Press, New York, NY, USA, 1998.