Design of Data Structures for Mergeable Trees*

Loukas Georgiadis[1,2]   Robert E. Tarjan[1,3]   Renato F. Werneck[1]

Abstract

Motivated by an application in computational topology, we consider a novel variant of the problem of efficiently maintaining dynamic rooted trees. This variant allows an operation that merges two tree paths. In contrast to the standard problem, in which only one tree arc at a time changes, a single merge operation can change many arcs. In spite of this, we develop a data structure that supports merges and all other standard tree operations in $O(\log^2 n)$ amortized time on an n-node forest. For the special case that occurs in the motivating application, in which arbitrary arc deletions are not allowed, we give a data structure with an $O(\log n)$ amortized time bound per operation, which is asymptotically optimal. The analysis of both algorithms is not straightforward and requires ideas not previously used in the study of dynamic trees. We explore the design space of algorithms for the problem and also consider lower bounds for it.

1 Introduction and Overview

A *heap-ordered forest* is a set of node-disjoint rooted trees, each node v of which has a real-valued label $\ell(v)$ satisfying *heap order*: for every node v with parent $p(v)$, $\ell(v) \geq \ell(p(v))$. We consider the problem of maintaining a heap-ordered forest, initially empty, subject to a sequence of the following kinds of operations:

- *parent*(v): Given a node v, return its parent, or *null* if it is a root.
- *nca*(v, w): Given nodes v and w, return the nearest common ancestor of v and w, or *null* if v and w are in different trees.
- *make*(v, x): Create a new, one-node tree consisting of node v with label x.
- *link*(v, w): Given a root v and a node w such that $\ell(v) \geq \ell(w)$ and w is not a descendent of v, combine the trees containing v and w by making w the parent of v.

- *delete*(v): Delete leaf v from the forest.
- *merge*(v, w): Given nodes v and w, let P be the path from v to the root of its tree, and let Q be the path from w to the root of its tree. Restructure the tree or trees containing v and w by merging the paths P and Q in a way that preserves the heap order. The merge order is unique if all node labels are distinct, which we can assume without loss of generality: if necessary, we can break ties by node identifier. See Figure 1.
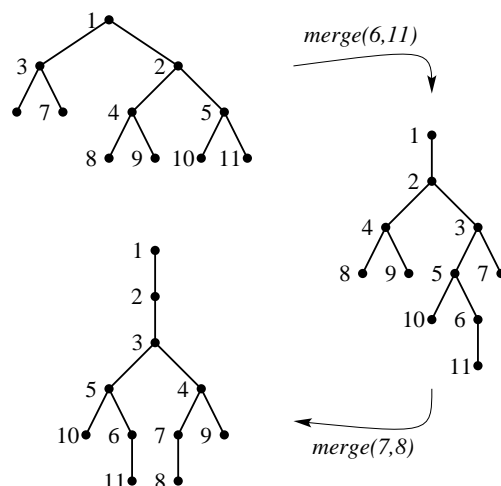

Figure 1: Two successive merges. The nodes are identified by label.

We call this the *mergeable trees problem*. In stating complexity bounds we denote by n the number of *make* operations (the total number of nodes) and by m the number of operations of all kinds; we assume $n \geq 2$. Our motivating application is an algorithm of Agarwal et al. [2] that computes the structure of 2-manifolds embedded in $\mathcal{R}^3$. In this application, the tree nodes are the critical points of the manifold (local minima, local maxima, and saddle points), with labels equal to their heights. The algorithm computes the critical points and their heights during a sweep of the manifold, and pairs up the critical points into so-called *critical pairs* using mergeable tree operations. This use of mergeable trees is actually a special case of the problem: *parent*, *nca*, and *merge* are applied only to leaves, *link* is used

only to attach a leaf to a tree, and each $nca(v, w)$ is followed immediately by $merge(v, w)$. None of these restrictions simplifies the problem significantly except for the restriction of merges to leaves. Since each such merge eliminates a leaf, there can be at most $n - 1$ merges. Indeed, in the 2-manifold application the total number of operations, as well as the number of merges, is $O(n)$. On the other hand, if arbitrary merges can occur, the number of merges can be $\Theta(n^2)$.

The mergeable trees problem is a new variant of the well-studied *dynamic trees problem*. This problem calls for the maintenance of a forest of rooted trees subject to *make* and *link* operations as well as operations of the following kind:

- *cut(v)*: Given a nonroot node v, make it a root by deleting the arc connecting v to its parent, thereby breaking its tree in two.

The trees are not (necessarily) heap-ordered and merges are not supported as single operations. Instead, each node and/or arc has some associated value or values that can be accessed and changed one node and/or arc at a time, an entire path at a time, or even an entire tree at a time. For example, in the original application of dynamic trees [21], which was to network flows, each arc has an associated real value representing a residual capacity. The maximum value of all arcs on a path can be computed in a single operation, and a constant can be added to all arc values on a path in a single operation. There are several known versions of the dynamic trees problem that differ in what kinds of values are allowed, whether values can be combined over paths or over entire trees at a time, whether the trees are unrooted, rooted, or ordered, and what operations are allowed. For all these versions of the problem, there are algorithms that perform a sequence of tree operations in $O(\log n)$ time per operation, either amortized [22, 25], worst-case [3, 12, 21], or randomized [1].

The main novelty, and the main difficulty, in the mergeable trees problem is the *merge* operation. Although dynamic trees support global operations on node and arc values, the underlying trees change only one arc at a time, by *links* and *cuts*. In contrast, a *merge* operation can delete and add many arcs (even $\Theta(n)$) simultaneously, thereby causing global structural change. Nevertheless, we have developed an algorithm that performs a sequence of mergeable tree operations in $O(\log n)$ amortized time per operation, matching the best bound known for dynamic trees. The time bound depends on the absence of arbitrary *cut* operations, but the algorithm can handle all the other operations that have been proposed for dynamic (rooted) trees. A variant of the algorithm can handle arbitrary *cuts* as well, but the

amortized time bound per operation becomes $O(\log^2 n)$. Both the design and the analysis of our algorithms require new ideas that have not been previously used in the study of the standard dynamic trees problem.

This paper is organized as follows. Section 2 introduces some terminology used to describe our algorithms. Section 3 shows how to extend Sleator and Tarjan's data structure for dynamic trees to solve the mergeable trees problem in $O(\log^2 n)$ amortized time per operation. Section 4 presents an alternative data structure that supports all operations except *cut* in $O(\log n)$ time. Section 5 discusses lower bounds, and Section 6 contains some final remarks.

2 Terminology

We consider forests of rooted trees, heap-ordered with respect to distinct node labels. We view arcs as being directed from child to parent, so that paths lead from leaves to roots. A vertex v is a *descendent* of a vertex w (and w is an *ancestor* of v) if the path from v to the root of its tree contains w. (This includes the case $v = w$.) If v is neither an ancestor nor a descendent of w, then v and w are *unrelated*. We denote by $size(v)$ the number of descendents of v, including v itself. The path from a vertex v to a vertex w is denoted by $P[v, w]$. We denote by $P[v, w)$ the subpath of $P[v, w]$ from v to the child of w on the path; if $v = w$, this path is empty. By extension, $P[v, null)$ is the path from v to the root of its tree. Similarly, $P(v, w]$ is the subpath of $P[v, w]$ from its second node to w, empty if $v = w$. Along a path P, labels strictly decrease. We denote by $bottom(P)$ and $top(P)$ the first and last vertices of P, respectively. The *length* of P, denoted by $|P|$, is the number of arcs on P. The *nearest common ancestor* of two vertices v and w is $bottom(P[v, null) \cap P[w, null))$; if the intersection is empty (that is, if v and w are in different trees), then their nearest common ancestor is *null*.

3 Mergeable Trees via Dynamic Trees

The *dynamic tree* (or *link-cut tree*) data structure of Sleator and Tarjan [21, 22] represents a forest to which arcs can be added (by the *link* operation) and removed (by the *cut* operation). These operations can happen in any order, as long as the existing arcs define a forest. The primary goal of the data structure is to allow efficient manipulation of paths. To this end, the structure implicitly maintains a partition of the arcs into *solid* and *dashed*. The solid arcs partition the forest into node-disjoint paths, which are interconnected by the dashed arcs. To manipulate a specific path $P[v, null)$, this path must first be *exposed*, which is done by making all its arcs solid and all its incident arcs dashed. Such an operation is called *exposing* v.

An *expose* operation is performed by doing a sequence of *split* and *join* operations on paths. The operation $split(Q, x)$ is given a path $Q = P[v, w]$ and a node x on Q. It splits Q into $R = P[v, x]$ and $S = P[x, w]$ and then returns the pair (R, S). The inverse operation, $join(R, S)$, is given two node-disjoint paths R and S. It catenates them (with R preceding S) and then returns the catenated path. Sleator and Tarjan proved that a sequence of *expose* operations can be done in an amortized number of $O(\log n)$ *joins* and *splits* per *expose*. Each operation on the dynamic tree structure requires a constant number of *expose* operations.

To allow *join* and *split* operations (and other operations on solid paths) to be done in sublinear time, the algorithm represents the actual forest by a *virtual forest*, containing the same nodes as the actual forest but with different structure. Each solid path P is represented by a binary tree, with bottom-to-top order along the path corresponding to symmetric order in the tree. Additionally, the root of the binary tree representing P has a virtual parent equal to the parent of $top(P)$ in the actual forest (unless $top(P)$ is a root in the actual forest). The arcs joining roots of binary trees to their virtual parents represent the dashed arcs in the actual forest. For additional details see [22].

The running time of the dynamic tree operations depends on the type of binary tree used to represent the solid paths. Ordinary balanced trees (such as red-black trees [15, 23]) suffice to obtain an amortized $O(\log^2 n)$ time bound per dynamic tree operation: each *join* or *split* takes $O(\log n)$ time. Splay trees [23, 22] give a better amortized bound of $O(\log n)$, because the splay tree amortization can be combined with the *expose* amortization to save a log factor. Sleator and Tarjan also showed [21] that the use of biased trees [5] gives an $O(\log n)$ time bound per dynamic tree operation, amortized for locally biased trees and worst-case for globally biased trees. The use of biased trees results in a more complicated data structure than the use of either balanced trees or splay trees, however.

Merge operations can be performed on a virtual tree representation in a straightforward way. To perform $merge(v, w)$, we first expose v and then expose w. As Sleator and Tarjan show [21], the nearest common ancestor u of v and w can be found during the exposure of w, without changing the running time by more than a constant factor. Having found u, we expose it. Now $P[v, u]$ and $P[w, u]$ are solid paths represented as single binary trees, and we complete the *merge* by combining these trees into a tree representing the merged path. In the remainder of this section we focus on the implementation and analysis of the path-merging process.

3.1 Merging by Insertions. A simple way to merge two paths represented by binary trees is to delete the nodes from the shorter path one-by-one and successively insert them into (the tree representing) the longer path, as proposed in a preliminary journal version of [2]. This can be done with only insertions and deletions on binary trees, thereby avoiding the extra complication of *joins* and *splits*. (The latter are still needed for *expose* operations, however.) Let p_i and q_i be the numbers of nodes on the shorter and longer paths combined in the i-th merge. If the binary trees are balanced, the total time for all the merges is $O(\sum p_i \log(q_i + 1))$. Even if insertions could be done in constant time, the time for this method would still be $\Omega(\sum p_i)$. In the preliminary journal version of [2], the authors claimed an $O(n \log n)$ bound on $\sum p_i$ for the case in which no cuts occur. Unfortunately this is incorrect: even without cuts, $\sum p_i$ can be $\Omega(n^{3/2})$, and this bound is tight (see Section A.1). We therefore turn to a more-complicated but faster method.

3.2 Interleaved Merges. To obtain a faster algorithm, we must merge paths represented by binary trees differently. Instead of inserting the nodes of one path into the other one-by-one, we find subpaths that remain contiguous after the merge, and insert each of these subpaths as a whole. This approach leads to an $O(\log^2 n)$ time bound per operation.

This algorithm uses the following variant of the *split* operation. If Q is a path with strictly decreasing node labels and k is a label value, the operation $split(Q, k)$ splits Q into the bottom part R, containing nodes with labels greater than or equal to k, and the top part S, containing nodes with labels less than k, and then returns the pair (R, S).

We implement $merge(v, w)$ as before, first exposing v and w and identifying their nearest common ancestor u, and then exposing u itself. We now need to merge the (now-solid) paths $R = P[v, u]$ and $S = P[w, u]$, each represented as a binary tree. Let Q be the merged path (the one we need to build). Q is initially empty. As the algorithm progresses, Q grows, and R and S shrink. The algorithm proceeds top-to-bottom along R and S, repeatedly executing the appropriate one of the following steps until all nodes are on Q:

1. If R is empty, let $Q \leftarrow join(S, Q)$.

2. If S is empty, let $Q \leftarrow join(R, Q)$.

3. If $\ell(top(R)) > \ell(top(S))$, remove the top portion of S and add it to the bottom of Q. More precisely, perform $(S, A) \leftarrow split(S, \ell(top(R)))$ and then $Q \leftarrow join(A, Q)$.

4. If $\ell(top(R)) < \ell(top(S))$, perform the symmetric

steps, namely $(R, A) \leftarrow split(R, \ell(top(S)))$ and then $Q \leftarrow join(A, Q)$.

To bound the running time of this method, we use an amortized analysis [24]. Each state of the data structure has a non-negative potential, initially zero. We define the amortized cost of an operation to be its actual cost plus the net increase in potential it causes. Then the total cost of a sequence of operations is at most the sum of their amortized costs.

Our potential function is a sum of two parts. The first is a variant of the one used in [21] to bound the number of *joins* and *splits* during *exposes*. The second, which is new, allows us to bound the number of *joins* and *splits* during *merges* (not counting the three *exposes* that start each *merge*).

We call an arc (v, w) *heavy* if $size(v) > size(w)/2$ and *light* otherwise. This definition implies that each node has at most one incoming heavy arc, and any tree path contains at most $\lg n$ light arcs (where $\lg$ is the binary logarithm). The *expose potential* of the actual forest is the number of heavy dashed arcs. We claim that an operation $expose(v)$ that does k splits and joins increases the expose potential by at most $2 \lg n - k + 1$. Each *split* and *join* during the *expose*, except possibly one *split*, converts a dashed arc along the exposed path to solid. For each of these arcs that is heavy, the potential decreases by one; for each of these arcs that is light, the potential can go up by one, but there are at most $\lg n$ such arcs. This gives the claim. The claim implies that the amortized number of *joins* and *splits* per *expose* is $O(\log n)$. An operation $link(v, w)$ can be done merely by creating a new dashed arc from v to w, without any *joins* or *splits*. This increases the expose potential by at most $\lg n$: the only dashed arcs that can become heavy are those on $P[v, null]$ that were previously light, of which there are at most $\lg n$. A *cut* of an arc (v, w) requires at most one *split* and can also increase the expose potential only by $\lg n$, at most one for each newly light arc on the path $P[w, null]$. Thus the amortized number of *joins* and *splits* per *link* and *cut* is $O(\log n)$. Finally, consider a *merge* operation. After the initial exposes, the rest of the *merge* cannot increase the expose potential, because all the arcs on the merged path are solid, and arcs that are not on the merged path can only switch from heavy to light, and not vice-versa.

The second part of the potential, the *merge potential*, we define as follows. Given all the *make* operations, assign to each node a fixed ordinal in the range $[1, n]$ corresponding to the order of the labels. Identify each node with its ordinal. The actual trees are heap-ordered with respect to these ordinals, as they are with respect to node labels. Assign each arc (x, y) a potential of $2 \lg(x - y)$. The *merge potential* is the sum of all the arc potentials. Each arc potential is at most $\lg n$, thus a *link* increases the merge potential by at most $\lg n$; a *cut* decreases it or leaves it the same. An *expose* has no effect on the merge potential. Finally, consider the effect of a *merge* on the merge potential. If the *merge* combines two different trees, it is convenient to view it as first linking the tree root of larger label to the root of smaller label, and then performing the merge. Such a *link* increases the merge potential by at most $\lg n$. We claim that the remainder of the *merge* decreases the merge potential by at least one for each arc broken by a *split*.

To verify the claim, we reallocate the arc potentials to the nodes, as follows: given an arc (x, y), allocate half of its potential to x and half to y. No such allocation can increase when a new path is inserted in place of an existing arc during a merge, because the node difference corresponding to the arc can only decrease. Suppose a merge breaks some arc (x, y) by inserting some path from v to w between x and y. Since $x > v \geq w > y$, either $v \geq (x + y)/2$ or $w \leq (x + y)/2$, or both. In the former case, the potential allocated to x from the arc to its parent (y originally, v after the merge) decreases by at least one; in the latter case, the potential allocated to y from the arc from its child (x originally, w after the merge) decreases by at least one. In either case, the merge potential decreases by at least one.

Combining these results, we obtain the following lemma:

LEMMA 3.1. *The amortized number of* joins *and* splits *is* $O(\log n)$ *per* link, cut, *and* expose, $O(\log n)$ *per* merge *that combines two trees, and* $O(1)$ *for a* merge *that does not combine two trees.*

Lemma 3.1 gives us the following theorem:

THEOREM 3.1. *If solid paths are represented as balanced trees, biased trees or splay trees, the amortized time per mergeable tree operation is* $O(\log^2 n)$.

Proof. The time per binary tree operation for balanced trees, biased trees or splay trees is $O(\log n)$. The theorem follows from Lemma 3.1 and the fact that $O(1)$ *exposes* are needed for each mergeable tree operation. □

For an implementation using balanced trees, our analysis is tight: for any n, there is a sequence of $\Theta(n)$ *make* and *merge* operations that take $\Omega(n \log^2 n)$ time. For an implementation using splay trees or biased trees, we do not know whether our analysis is tight; it is possible that the amortized time per operation is $O(\log n)$. We can obtain a small improvement for splay

trees by using the amortized analysis for dynamic trees given in [22] in place of the one given in [21]. With this idea we can get an amortized time bound of $O(\log n)$ for every mergeable tree operation, except for *links* and *merges* that combine two trees; for these operations, the bound remains $O(\log^2 n)$. The approach presented here can also be applied to other versions of dynamic trees, such as top trees [3], to obtain mergeable trees with an amortized $O(\log^2 n)$ time bound per operation.

4 Mergeable Trees via Partition by Rank

The path decomposition maintained by the algorithm of Section 3.2 is structurally unconstrained; it depends only on the sequence of tree operations. If arbitrary *cuts* are disallowed, we can achieve a better time bound by maintaining a more-constrained path decomposition. The effect of the constraint is to limit the ways in which solid paths change. Instead of arbitrary *joins* and *splits*, the constrained solid paths are subject only to arbitrary insertions of single nodes, and deletions of single nodes from the top.

We define the *rank* of a vertex v by $rank(v) = \lfloor \lg size(v) \rfloor$. To decompose the forest into solid paths, we define an arc (v, w) to be solid if $rank(v) = rank(w)$ and dashed otherwise. Since a node can have at most one solid arc from a child, the solid arcs partition the forest into node-disjoint solid paths. See Figure 2.
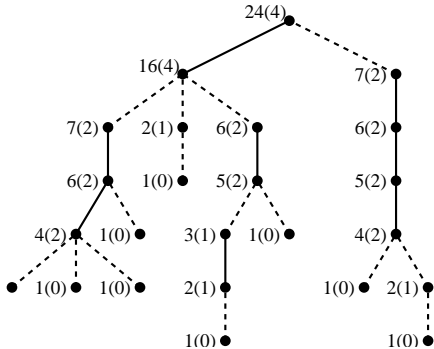

Figure 2: A tree partitioned by size into solid paths, with the corresponding sizes and, in parentheses, ranks.

The algorithm works essentially as before. To merge v and w, we first ascend the paths from v and w to find their nearest common ancestor u; then we merge the traversed paths in a top-down fashion. We can no longer use the *expose* operation, however, since the partition of the forest into solid paths cannot be changed. Also, we must update the partition as the merge takes place. This requires keeping track of node ranks.

Section 4.1 describes the algorithm in more detail, without specifying the data structure used to represent solid paths. Interestingly, the set of operations needed

on solid paths is limited enough to allow their representation by data structures as simple as heaps (see Section A.3). The data structures that give us the best running times, however, are finger trees and splay trees. We discuss their use within our algorithm in Sections 4.2 and 4.3.

4.1 Basic Algorithm

Nearest common ancestors. To find the nearest common ancestor of v and w, we concurrently traverse the paths from v and w bottom-up, rank-by-rank, and stop when we reach the same solid path P (or *null*, if v and w are in different trees). If x and y are the first vertices on P reached during the traversals from v and w, respectively, then the nearest common ancestor of v and w is whichever of x and y has smaller label.

Each of these traversals visits a sequence of at most $1 + \lg n$ solid paths linked by dashed arcs. To perform *nca* in $O(\log n)$ time, we need the ability to jump in constant time from any vertex on a solid path P to $top(P)$, and from there follow the outgoing dashed arc. We cannot afford to maintain an explicit pointer from each node to the top of its solid path, since removing the top node would require updating more than a constant number of pointers, but we can use one level of indirection for this purpose. We associate with each solid path P a record P^* that points to the top of P; every node in P points to P^*. When $top(P)$ is removed, we just make P^* point to its (former) predecessor on P. When an element is inserted into P, we make it point to P^*. With the solid path representations described in Sections 4.2 and 4.3, both operations take constant time.

Merging. The first pass of $merge(v, w)$ is similar to $nca(v, w)$: we follow the paths from v and w bottom-up until their nearest common ancestor u is found. Then we do a second pass, in which we traverse the paths $P[v, u]$ and $P[w, u]$ in a top-down fashion and merge them. To avoid special cases in the detailed description of the merging process, we assume the existence of dummy leaves v' and w', children of v and w respectively, each with label ∞ and rank -1. For a node x on $P[v, u]$ or $P[w, u]$, we denote by $pred(x)$ the node preceding x on $P[v, u]$ or $P[w, u]$, respectively. To do the merging, we maintain two current nodes, p and q. Initially, $p = top(P[v, u])$ and $q = top(P[w, u])$; if $rank(p) < rank(q)$, we swap p and q, so that $rank(p) \geq rank(q)$. We also maintain r, the bottommost node for which the merge has already been performed. Initially $r = u$. The following invariants will hold at the beginning of every step: (1) the current forest is correctly partitioned into

solid paths; (2) $rank(p) \geq rank(q)$; (3) r is either *null* or is the parent of both p and q. The last two invariants imply that the arc from q to r, if it exists, is dashed.

To perform the *merge*, we repeat the appropriate one of steps 1 and 2 below until $p = v'$ and $q = w'$. The choice of step depends on whether p or q goes on top.

1. $\ell(p) > \ell(q)$. The node with lower rank (q) will be on top, with new rank $rank'(q) = \lfloor \lg(size(p) + size(q)) \rfloor$. Let $t = pred(q)$. Since $rank'(q) > rank(q)$, we remove q from the top of its solid path and insert it between p and r. There are two possibilities:

   (a) If $rank'(q) < rank(r)$, the arc from q to r, if it exists, will be dashed. If $rank'(q) > rank(p)$, the arc from p to q will also be dashed, and q forms a one-node solid path. Otherwise, we add q to the top of the solid path containing p.

   (b) If $rank'(q) = rank(r)$, we insert q into the solid path containing r (just below r itself); the arc from q to r will be solid.

   In both cases, we set $r \leftarrow q$ and $q \leftarrow t$.

2. $\ell(p) < \ell(q)$. The node with higher rank (p) goes on top. Let $rank'(p) = \lfloor \lg(size(p) + size(q)) \rfloor$ be the rank of p after the merge. We must consider the following subcases:

   (a) $rank'(p) > rank(p)$: This case happens only if p is the top of its solid path (otherwise the current partition would not be valid). We remove it from this path, and make the arc from $t = pred(p)$ to p dashed, if it is not dashed already.

      i. If $rank'(p) = rank(r)$, we insert p into the solid path containing r (note that r cannot have a solid predecessor already, or else its rank would be greater than $rank'(p)$).

      ii. If $rank'(p) < rank(r)$, we keep the arc from p to r, if it exists, dashed.

      We then set $r \leftarrow p$ and $p \leftarrow t$.

   (b) $rank'(p) = rank(p)$: p remains on the same solid path (call it P). Let e be the first (lowest) vertex on P reached during the traversal from v and w and let $t = pred(e)$. The arc (t, e) is dashed. There are two subcases:

      i. $\ell(e) < \ell(q)$: e will be above q after the merge. We set $r \leftarrow e$ and $p \leftarrow t$.

      ii. $\ell(e) > \ell(q)$: we perform a search on P for the bottommost vertex x whose label is smaller than $\ell(q)$. We set $r \leftarrow x$ and set $p \leftarrow pred(x)$. The new node p is on P.

In all cases above, we make q a (dashed) child of the new r. If now $rank(p) < rank(q)$ (which can only happen if the arc from p to r is dashed), we swap pointers p and q to preserve Invariant (2).

Maintaining sizes. The algorithm makes some decisions based on the value of $\lfloor \lg(size(p) + size(q)) \rfloor$. We cannot maintain all node sizes explicitly, since many nodes may change size during each *merge* operation. Fortunately, the only sizes actually needed are those of the top vertices of solid paths. This is trivially true for q. For p, we claim that, if the arc from p to r is solid, then $\lfloor \lg(size(p) + size(q)) \rfloor = rank(p)$. This follows from $rank(p) \leq \lfloor \lg(size(p) + size(q)) \rfloor \leq rank(r) = rank(p)$, since p is a child of r, and p and r are on the same solid path.

These observations are enough to identify when cases 1(a) and 2(a) apply, and one can decide between the remaining cases (1(b) and 2(b)) based only on node labels. Therefore, we keep explicitly only the sizes of the top nodes of the solid paths. To update these efficiently, we also maintain the *dashed size* of every vertex x, defined to be the sum of the sizes of the dashed children of x (i.e., the children connected to x by a dashed arc) plus one (which accounts for x itself). These values can be updated in constant time whenever the merging algorithm makes a structural change to the tree. Section A.2 explains how.

Complexity. To simplify our analysis, we assume that there are no leaf deletions. We discuss leaf deletions in Section 4.4. The running time of the merging procedure depends on which data structure is used to represent solid paths. But at this point we can at least count the number of basic operations performed. There will be at most $O(n \log n)$ removals from the top of a solid path, since they only happen when a node rank increases. Similarly, there will be at most $O(n \log n)$ insertions. The number of searches (case 2(b)ii) will also be $O(n \log n)$, since every search precedes an insertion of q, and therefore an increase in rank. The only case unaccounted for is 2(b)i, which will happen at most $O(\log n)$ times per operation, taking $O(m \log n)$ time in total. This is also the time for computing nearest common ancestors.

4.2 Solid Paths as Finger Trees. We show how to achieve an $O(\log n)$ amortized bound per operation by representing each solid path as a *finger search tree* [7, 26] (*finger tree* for short). A finger tree is a balanced search tree that supports fast access in the vicinity of certain preferred positions, indicated by pointers called *fingers*. Specifically, if there are d items between the target

and starting point (i.e., an item with finger access), then the search takes $O(\log(d+2))$ time. Furthermore, given the target position, finger trees support insertions and deletions in constant amortized time. (Some finger trees achieve constant-time finger updates in the worst case [11, 6].) When solid paths are represented as ordinary balanced binary trees, the cost of inserting a node w into a path P is proportional to $\log|P|$. With finger trees, this cost is reduced to $O(\log(|P[v,x]|+2))$, where w is inserted just below v and x is the most-recently-accessed node of P (unless w is the first node inserted during the current merge operation, in which case $x = top(P)$). We refer to $P[v,x]$ as the *insertion path* of w into P. After w is inserted, it becomes the most-recently-accessed node. To bound the running time of the merging algorithm, we first give an upper bound on the sum of the logarithms of the lengths of all insertion paths during the sequence of merges. We denote this sum by L.

LEMMA 4.1. *For any sequence of merge operations,* $L = O(n \log n)$.

Proof. Let P and Q be two solid paths being merged, with $rank(P) = r_p \geq rank(Q) = r_q$ (i.e., nodes of Q are inserted into P). Let J be the set of integers in $[0, \lceil \lg \lg n \rceil]$, and let j be any integer in J. Define $\beta(j) = 2^j$. We consider insertions that satisfy:

$$(4.1) \qquad \beta(j) \leq r_p - r_q < \beta(j+1).$$

Every element of Q that gets inserted into P increases its rank by at least $\beta(j)$; hence there can be at most $(n \lg n)/\beta(j)$ such insertions. Let $\delta(j)$ be the fraction of these insertions that actually occur, and let $\lambda(j) = (\delta(j)\, n \lg n)/\beta(j)$ be the number of actual insertions that satisfy condition (4.1). Since the total number of rank increases is at most $n \lg n$, we have

$$(4.2) \qquad \sum_{j \in J} \lambda(j)\beta(j) \leq n \lg n \Rightarrow \sum_{j \in J} \delta(j) \leq 1.$$

Next we calculate an upper bound for the total length of the insertion paths that satisfy (4.1). Whenever a node x is included in such an insertion path, it acquires at least 2^{r_q} descendants.[1] Since $r_p - r_q < \beta(j+1)$, x can be on at most $2^{\beta(j+1)}$ such paths while maintaining rank r_p. Then the total number of occurrences of x on all insertion paths that satisfy (4.1) is at most $(\lg n)2^{\beta(j+1)}$, since the rank of a node can change at most $\lg n$ times. This implies an overall bound of $(n \lg n)2^{\beta(j+1)}$. This length is distributed over $\lambda(j)$ insertions and, because the log function is concave, the

---
[1] In the SODA'06 proceedings, due to a typo, it is mistakenly mentioned that x acquires at least $\beta(j)$ descendants.

sum $S(j)$ of the logarithms of the lengths of the insertion paths is bounded above by

$$
\begin{aligned}
S(j) \quad &\leq \quad \lambda(j) \lg \frac{(n \lg n)2^{\beta(j+1)}}{\lambda(j)} \\
&= \quad \delta(j) \frac{n \lg n}{\beta(j)} \lg \frac{\beta(j)2^{2\beta(j)}}{\delta(j)} \\
&< \quad \delta(j) \frac{n \lg n}{\beta(j)} \lg \frac{2^{3\beta(j)}}{\delta(j)} \\
&\leq \quad \frac{n \lg n}{\beta(j)} \delta(j) \lg \frac{1}{\delta(j)} + 3\delta(j) n \lg n \\
&\leq \quad n \lg n \left(\frac{1}{\beta(j)} + 3\delta(j) \right).
\end{aligned}
$$

We need to bound the sum of this expression for all j. The $1/\beta(j)$ terms form a geometric series that sums to at most 2. The $3\delta(j)$ terms add up to at most 3 by Inequality 4.2. Therefore, $L \leq 5n \lg n$. $\qquad \square$

LEMMA 4.2. *Any sequence of m merge and nca operations takes $O((n+m)\log n)$ time.*

Proof. We have already established that each *nca* query takes $O(\log n)$ time, so we only have to bound the time necessary to perform merges. According to our previous analysis, there are at most $O(n \log n)$ insertions and deletions on solid paths. Finger trees allow each to be performed in constant amortized time, as long as we have a finger to the appropriate position. Lemma 4.1 bounds the total time required to find these positions by $O(n \log n)$, which completes the proof. $\qquad \square$

This lemma implies an $O(\log n)$ amortized time bound for every operation but *cut*. The analysis for the remaining operations is trivial; in particular, *link* can be interpreted as a special case of *merge*. This bound is tight for data structures that maintain parent pointers explicitly: there are sequences of operations in which parent pointers change $\Omega(n \log n)$ times.

4.3 Solid Paths as Splay Trees. The bound given in Lemma 4.2 also holds if we represent each solid path as a splay tree. To prove this, we have to appeal to Cole's *dynamic finger theorem for splay trees*:

THEOREM 4.1. [9, 8] *The total time to perform m_t accesses (including inserts and deletes) on an arbitrary splay tree t, initially of size n_t, is $O(m_t + n_t + \sum_{j=1}^{m_t} \log(d_t(j)+2))$, where, for $1 \leq i \leq m_t$, the j-th and $(j-1)$-st accesses are performed on items whose distance in the list of items stored in the splay tree is $d_t(j)$. For $j=0$, the j-th item is interpreted to be the item originally at the root of the splay tree.*

To bound the total running time of our data structure we simply add the contribution of each solid path using the bound given in the theorem above. (We also have to consider the time to find the nearest common ancestors, but this has already been bounded by $O(m \log n)$.) Moreover, $\sum_t (n_t + m_t) = O(n \log n)$, where the sum is taken over all splay trees (i.e., solid paths) ever present in the virtual tree. Finally, we can use Lemma 4.1 to bound $\sum_t \sum_j (\log (d_t(j) + 2))$ by $O(n \log n)$. Note, however, that we cannot apply the lemma directly: it assumes that the most-recently-accessed node at the beginning of a *merge* is the top vertex of the corresponding solid path. In general, this does not coincide with the root of the corresponding splay tree, but it is not hard to prove that the bound still holds.

4.4 Leaf Deletions. Although the results of Sections 4.2 and 4.3 do not (necessarily) hold if we allow arbitrary *cuts* (because the ranks can decrease), we can in fact allow leaf deletions. The simplest way to handle leaf deletions is merely to mark leaves as deleted, without otherwise changing the structure. Then a leaf deletion takes $O(1)$ time, and the time bounds for the other operations are unaffected.

5 Lower Bounds

A data structure that solves the mergeable trees problem can be used to sort a sequence of numbers. Given a sequence of n values, we can build a tree in which each value is associated with a leaf directly connected to the root. If we merge pairs of leaves in any order, after $n - 1$ operations we will end up with the values in sorted order. As a result, any lower bound for sorting applies to our problem. In particular, in the comparison and pointer machine models, a worst-case lower bound of $\Omega(n \log n)$ for executing n operations holds, and the data structures of Sections 4.2 and 4.3 are optimal.

This lower bound is not necessarily valid if we are willing to forgo the *parent* operation (as in the case of Agarwal et al.'s original problem), or consider more powerful machine models. Still, we can prove a superlinear lower bound in the cell-probe model of computation [31, 13], which holds even for the restricted case in which only the *nca* and *merge* operations are supported, and furthermore the arguments of *merges* are always leaves. This result follows from a reduction of the *boolean union-find* problem to the mergeable trees problem. The boolean union-find problem is that of maintaining a collection of disjoint sets under two operations: $union(A, B, C)$, which creates a new set C representing the union of sets A and B (which are destroyed), and $find(x, A)$, which returns *true* if x

belongs to set A and *false* otherwise. Kaplan, Shafrir, and Tarjan [17] proved an $\Omega(m\alpha(m, n, \log n))$ lower bound for this problem in the cell probe model with cells of size $\log n$. Here n is the total number of elements and m is the number of *find* operations. Function $\alpha(m, n, \ell)$ is defined as $\min\{k : A_k(\lceil m/n \rceil) > \ell\}$, where $A_k(\cdot)$ is the k-th row of Ackermann's function.

We can solve this problem using mergeable trees as follows. Start with a tree with n leaves directly connected to the root. Each leaf represents an element (the root does not represent anything in the original problem). As the algorithm progresses, each path from a leaf to the root represents a set whose elements are the vertices on the path (except the root itself). Without loss of generality, we use the only leaf of a set as its unique identifier. To perform $union(A, B, C)$, we execute $merge(A, B)$ and set $C \leftarrow \max\{A, B\}$. To perform $find(x, A)$, we perform $v \leftarrow nca(x, A)$. If $v = x$, we return *true*; otherwise v will be the root, and we return *false*. This reduction also implies some tradeoffs for the worst-case update (*merge*) time t_u and the worst-case query (*nca*) time t_q. Again it follows from [17] that for $t_u \geq \max\{(\frac{140b}{\log n})^\epsilon, 140^\epsilon\}$ we have $t_q = \Omega(\frac{\epsilon \log n}{(\epsilon+1)\log t_u})$.

Finally, we note that [19] gives an $\Omega(m \log n)$ lower bound for any data structure that supports a sequence of m *link* and *cut* operations. This lower bound is smaller by a log factor than our upper bound for any sequence of m mergeable tree operations that contains arbitrary *links* and *cuts*.

For completeness, we compare these lower bounds to the complexity of m on-line nearest common ancestor queries on a static forest of n nodes. For pointer machines, Harel and Tarjan [16] give a worst-case $\Omega(\log \log n)$ bound per query. An $O(n + m \log \log n)$-time pointer-machine algorithm was given in [28]; the same bound has been achieved for the case where arbitrary *link*s are allowed [4]. On RAMs, the static on-line problem can be solved in $O(n + m)$ time [16, 20]; when arbitrary links are allowed, the best known algorithm runs in $O(n + m\alpha(m + n, n))$ time [14].

6 Final Remarks

There are several important questions related to the mergeable trees problem that remain open. Perhaps the most interesting one is to devise a data structure that supports all the mergeable tree operations, including arbitrary *cuts*, in $O(\log n)$ amortized time. In this most general setting only the data structure of Section 3.2 gives an efficient solution, supporting all operations in $O(\log^2 n)$ amortized time. The data structure of Section 4 relies on the assumption that the rank of each node is nondecreasing; in the presence of arbitrary *cuts* (which can be followed by arbitrary *links*) explicitly

maintaining the partition by rank would require linear time per *link* and *cut* in the worst case. We believe, however, that there exists an optimal data structure (at least in the amortized sense) that can support all operations in $O(\log n)$ time. Specifically, we conjecture that Sleator and Tarjan's link-cut trees implemented with splay trees do support m mergeable tree operations (including *cut*) in $O(m \log n)$ time using the algorithm of Section 3.2.

Another issue has to do with the definition of n. We can easily modify our algorithms so that n is the number of nodes in the current forest, rather than the total number of nodes ever created. Whether the bounds remain valid if n is the number of nodes in the tree or trees involved in the operation is open.

Another direction for future work is devising data structures that achieve good bounds in the worst case. This would require performing merges implicitly.

The mergeable trees problem has several interesting variants. When the data structure needs to support only a restricted set of operations, it remains an open question whether we can beat the $O(\log n)$ bound on more powerful machine models (e.g. on RAMs). When both *merge* and *nca* but not *parent* operations are allowed, the gap between our current upper and lower bounds is more than exponential.

References

[1] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms*, pages 524–533, 2004.

[2] P. K. Agarwal, H. Edelsbrunner, J. Harer, and Y. Wang. Extreme elevation on a 2-manifold. In *Proc. 20th Symp. on Comp. Geometry*, pages 357–365, 2004.

[3] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining diameter, center, and median of fully-dynamic trees with top trees. Unpublished manuscript, *http://arxiv.org/abs/cs/0310065*, 2003.

[4] S. Alstrup and M. Thorup. Optimal algorithms for finding nearest common ancestors in dynamic trees. *Journal of Algorithms*, 35:169–188, 2000.

[5] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14(3):545–568, 1985.

[6] G. S. Brodal, G. Lagogiannis, C. Makris, A. Tsakalidis, and K. Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences*, 67(2):381–418, 2003.

[7] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[8] R. Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[9] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[10] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 365–372, 1987.

[11] P. F. Dietz and R. Raman. A constant update time finger search tree. *Information Processing Letters*, 52(3):147–154, 1994.

[12] G. N. Frederickson. Data structures for on-line update of minimum spanning trees, with applications. *SIAM Journal of Computing*, 14:781–798, 1985.

[13] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21th ACM Symp. on Theory of Computing*, pages 345–354, 1989.

[14] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 434–443, 1990.

[15] L. J. Guibas and R. Sedgewick. A dichromatic framwork for balanced trees. In *Proc. 19th Symp. on Foundations of Computer Science*, pages 8–21, 1978.

[16] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[17] H. Kaplan, N. Shafrir, and R. E. Tarjan. Meldable heaps and boolean union-find. In *Proc. 34th ACM Symp. on Theory of Computing*, pages 573–582, 2002.

[18] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[19] Mihai Pătraşcu and Erik D. Demaine. Lower bounds for dynamic connectivity. In *Proc. 36th ACM Symp. on Theory of Computing*, pages 546–553, 2004.

[20] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[21] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[22] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[23] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM Press, Philadelphia, PA, 1983.

[24] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, 1985.

[25] R. E. Tarjan and R. F. Werneck. Self-adjusting top trees. In *Proc. 16th ACM-SIAM Symp. on Discrete Algorithms*, pages 813–822, 2005.

[26] R. E. Tarjan and C. J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon.

SIAM Journal on Computing, 17(1):143–173, 1988.

[27] M. Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[28] A. K. Tsakalides and J. van Leeuwen. An optimal pointer machine algorithm for finding nearest common ancestors. Technical Report RUU-CS-88-17, U. Utrecht Dept. of Computer Science, 1988.

[29] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[30] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[31] A. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.

A Appendix

A.1 Bounds on Iterated Insertions. This section shows that there exists a sequence of operations on which Agarwal et al.'s algorithm makes $\Theta(n\sqrt{n})$ insertions on binary trees. We start with a tree consisting of $2k + 1$ nodes, $v_0, v_1, \ldots, v_{2k}$. For every i in $[1, k]$, let v_{i-1} be the parent of both v_i and v_{k+i}.

Consider a sequence of $k - \sqrt{k}$ merges such that the i-th merge combines v_{k+i} with $v_{k+i+\sqrt{k}}$ (for simplicity, assume k is a perfect square). In other words, we always combine the topmost leaf with the leaf that is $\sqrt{k}$ levels below. On each merge, the longest path is the one starting from the bottom leaf: it will have $1 + \sqrt{k}$ arcs. The shortest path will have size 1 for the first $\sqrt{k}$ merges, 2 for the following $\sqrt{k}$, 3 for the next $\sqrt{k}$, and so on, up to $\sqrt{k} - 1$. Let p_i be the length of the shortest path during the i-th merge. Over all merges, these lengths add up to

$$\sum_{i=1}^{k-\sqrt{k}} p_i = \sum_{i=1}^{\sqrt{k}-1} (i\sqrt{k}) = \sqrt{k} \sum_{i=1}^{\sqrt{k}-1} i = \frac{k\sqrt{k} - k}{2} = \Theta(n\sqrt{n}).$$

A simple potential function argument (counting the number of unrelated vertices) shows that this is actually the worst case of the algorithm. For any sequence of n merges in an n-node forest, $\sum_{i=1}^{n} p_i < n\sqrt{n}$.

A.2 Maintaining sizes. This section explains how to update the $s(\cdot)$ (size) and $d(\cdot)$ (dashed size) fields during the *merge* operation when the tree is partitioned by rank. There are three basic operations we must handle: removal of the topmost node of a solid path; insertion of an isolated node (a node with no adjacent solid arcs) into a solid path; and replacement of the (dashed) parent of q (moving the subtree rooted at q down the tree). We consider each case in turn.

Consider removals first. Let x be a node removed from a solid path and let c be its solid child. Node c will become the top of a solid path, so we keep its size

explicitly: $s'(c) \leftarrow s(x) - d(x)$. Also, because all arcs entering x become dashed, we must update its dashed size: $d'(x) \leftarrow s(x)$. All other fields (including $s(x)$ and $d(c)$) remain unchanged.

Now consider the insertion of a node x into a solid path S. Whenever this happens, the parent of x does not change (it is always r). There are two cases to consider. If $x = top(S)$, it will gain a solid child c, the former top of the path; we set $s'(x) \leftarrow s(c) + s(x)$ and $s'(c)$ becomes undefined (the values of $d(x)$ and $d(c)$ remain unchanged). If $x \neq top(S)$, we set $d'(r) \leftarrow d(r) - s(x)$ (both $s(r)$ and $d(x)$ remain unchanged, and $s(x)$ becomes undefined).

The third operation we must handle is moving q. Let r_0 be its original parent and r_1 be the new one (recall that r_1 is always a descendent of r_0). We must update their dashed sizes: $d'(r_0) \leftarrow d(r_0) - s(q)$ and $d'(r_1) \leftarrow d(r_1) + s(q)$. Furthermore, if r_0 and r_1 belong to different solid paths, we set $s'(r_1) \leftarrow s(r_1) + s(q)$ (r_1 will be the topmost vertex of its path in this case).

A.3 Solid Paths as Heaps. In the special case where we do not need to maintain parent pointers, we can use the algorithm described in Section 4 and represent each solid path as a heap, with higher priority given to elements of smaller label. To insert an element into the path, we use the standard *insert* heap operation. Note that it is not necessary to search for the appropriate position within the path: the position will not be completely defined until the element becomes the topmost vertex of the path, when it can be retrieved with the *deletemin* operation. If each heap operation can be performed in $f(n)$ time, the data structure will be able to execute m operations in $O((m + nf(n))\log n)$ time. Thus we get an $O(n\log^2 n + m\log n)$-time algorithm using ordinary heaps, and an $O(n\log n\log\log n + m\log n)$-time algorithm using the fastest known RAM priority queues [27].

We can also represent each solid path with a van Emde Boas tree [29, 30]. They support insertions, deletions and predecessor queries in $O(\log\log n)$ amortized time if the labels belong to a small universe (polynomial in n). Arbitrary labels can be mapped to a small universe with the Dietz-Sleator data structure [10] (and a balanced search tree) or, if they are known offline, with a simple preprocessing step. We must also use hashing [18] to ensure that the total space taken by all trees (one for each solid path) is $O(n)$. With these measures, we can solve the mergeable trees problem (including *parent* queries) in $O(n\log n\log\log n + m\log n)$ total time.