# Fast Decremental Tree Sums in Forests

**Benjamin Aram Berendsohn**
Max Planck Institute for Informatics, Saarbrücken, Germany

**Marek Sokołowski**
Max Planck Institute for Informatics, Saarbrücken, Germany

## Abstract

We study two fundamental decremental dynamic graph problems. In both problems, we need to maintain a vertex-weighted forest of size $n$ under edge deletions, weight updates, and a certain information-retrieval query. Both problems can be solved in $\mathcal{O}(\log n)$ time per update/query using standard dynamic forest data structures like top trees – even if additionally edge *insertions* are allowed. We investigate whether the deletion-only problem can be solved faster.

First, we consider `tree-sum` queries, where we ask for the sum of vertex weights in one of the connected components (i.e., trees) in the forest. We give a data structure with $\mathcal{O}(n)$ preprocessing time and $\mathcal{O}(\log^* n)$ time per operation, based on a micro-macro tree decomposition (Alstrup et al., 1997). If the forest is *unweighted* (i.e., all weights are 1 and cannot be changed), then the operation time can be improved to $\mathcal{O}(1)$.

Additionally, we give an asymptotically *universally optimal* algorithm. More specifically, our algorithm works in the *group model*, and processes $m$ operations on an initial forest $F$ in running time $\mathcal{O}(\mathrm{OPT}(F, m))$. Here $\mathrm{OPT}(F, m)$ is the number of weight additions and subtractions that a best possible algorithm performs to handle a worst-case instance for a fixed initial forest $F$ and a fixed number $m$ of operations. We achieve this with a combination of the aforementioned decomposition technique, precomputation of optimal data structures for very small instances, and some insights into the behavior of OPT. Note that even the *worst-case* complexity of this algorithm remains unknown to us.

Second, we consider `subtree-sum` queries. Here, the forest is rooted, and a query `subtree-sum`$(v)$ returns the sum of weights in the subtree rooted at $v$. An easy reduction from the well-known *prefix sum* problem shows that the general, weighted version of the problem requires $\Theta(n \log n)$ time for $n$ operations. Interestingly, we prove that the $\Omega(n \log n)$ complexity lower bound still holds even if weight updates are disallowed. On the other hand, we show that the unweighted version can be solved with $\mathcal{O}(\frac{\log n}{\log \log n})$ time per operation, and this is tight.

**2012 ACM Subject Classification** Theory of computation → Dynamic graph algorithms

**Keywords and phrases** dynamic graphs, connectivity, group model, universal optimality

**Digital Object Identifier** 10.4230/LIPIcs.ICALP.2026.26

**Category** Track A: Algorithms, Complexity and Games

**Related Version** *Full Version*: `http://arxiv.org/abs/2605.06555`

## 1 Introduction

Maintaining connectivity in dynamic graphs is one of the fundamental problems in the design of algorithms, with a plethora of major results in the area throughout the decades. An efficient data structure maintaining connectivity in dynamic *forests* was given by Sleator and Tarjan [31]: Their *link/cut trees* support connectivity queries under edge insertions and removals in an $n$-vertex forest in $\mathcal{O}(\log n)$ time. They also show that their data structure supports *weights* of vertices and various queries related to these weights in the same time complexity, such as the sum of weights of vertices in a tree component containing a given vertex, or the sum of weights of vertices in a subtree rooted at a specified vertex. The optimality of this data structure – even in the unweighted setting – was only shown much



later by Pătraşcu and Demaine [28]. In the more general setting of arbitrary dynamic graphs, the asymptotic behavior of an optimal data structure is still not fully understood, with the currently best algorithm of Huang et al. [23] being a factor of $\Theta((\log \log n)^2)$ away from the logarithmic lower bound.

Aiming to break through the logarithmic operation lower bound, we may decide to restrict the space of possible updates to the graph. The *incremental* setting, in which edges may only be inserted to the graph, is precisely the *disjoint set union* problem, which was already studied by Galler and Fischer [18] and subsequently improved: Hopcroft and Ullman gave a data structure supporting insertions and connectivity queries in $\mathcal{O}(\log^* n)$ time per operation [22], with the complexity analysis later refined by Tarjan to $\mathcal{O}(\alpha(n))$ per operation [32]. Here, $\log^* n$ and $\alpha(n)$ denote two very slow-growing functions, namely the *iterated logarithm* and the *inverse Ackermann function*. Again, the presented data structures can be easily adapted to support weights of vertices and report aggregate information on the connected components of the graphs. The bound of $\mathcal{O}(\alpha(n))$ was later shown to be optimal in various computational models by Tarjan [33] and Fredman and Saks [14], even under the promise that the graph always remains a forest.

In this work, we focus on the *decremental* setting, where a data structure is given an initial graph and edges may only be removed from the graph. This restriction has also received considerable attention over the years, and the decremental connectivity problem has been solved optimally: A data structure processing $m$ operations in an $n$-vertex forest in total $\mathcal{O}(n + m)$ time was given by Alstrup, Secher and Spork [4], with the result later strengthened to planar graphs by Łącki and Sankowski [25]. However, interestingly, even the decremental forest data structure employs various bit-optimization tricks to achieve the optimal time complexity, which renders it unable to handle weights of vertices efficiently. This suggests a natural question: How efficiently could we implement a data structure for *weighted* decremental forests that can also report summaries of tree components – such as the sum of the weights of the vertices in the component – dynamically? And can such a data structure also handle more complex queries, such as the sum of the weights of the vertices in a rooted subtree of a specified tree component?

**Setting.** We now formally introduce the problem. The task is to maintain a vertex-weighted *rooted* forest $(F, w)$, where weights come from some commutative group $G$, under some or all of the following operations:

- `cut`$(v)$: Delete the edge between $v$ and its parent (we assert that this edge exists).
- `update-weight`$(v, x)$: Set $w(v) \leftarrow x$.
- `tree-sum`$(v) \to x$: Return the sum $x$ of weights in the tree containing $v$.
- `subtree-sum`$(v) \to x$: Return the sum $x$ of weights in the subtree $F_v$ rooted at $v$.

We are mostly interested in the *group model* of computation, where the weights of the vertices are considered to be opaque objects. The implementation of a data structure may manipulate the weights (elements of the group) by adding them together or subtracting one from another, but no other operations – such as comparisons or the extraction of the representation of the weight in memory – are allowed.

All operations listed above can be implemented in $\mathcal{O}(\log n)$ worst-case time in the group model with well-known efficient fully dynamic forest data structures such as aforementioned link/cut trees [31] or top trees [2], so any sequence of $m$ operations can be processed in $\mathcal{O}((n + m) \log n)$ time. The question remains: Can (some or all) of the operations above be implemented more efficiently, knowing that the edges may only be removed from the forest?

## 1.1 Results and techniques

In the following, we formally present our results and give basic ideas on how to prove them. In the informal discussion following each result, we will assume for convenience that the number of operations $m$ is roughly equal to the number $n$ of vertices in the input forest. Thus the running times in the discussion will always be the total of the initialization time and the time required to process all $m \approx n$ operations. We will also assume the forest given initially is a tree.

**Tree-sums with recursive cluster decompositions.** Our first result concerns the basic `tree-sum` problem.

▶ **Theorem 1.1.** *There is a data structure that maintains a weighted rooted forest $(F, w)$ on $n$ vertices under* `cut`*,* `update-weight`*, and* `tree-sum`*, with running time $\mathcal{O}((m+n)\log^* n)$ for $m$ operations.*

We prove Theorem 1.1 in Section 3 using the well-known technique of *micro-macro decomposition* (defined in [4], though similar ideas already exist in [17]), although we adapt it to our purposes and call it *cluster decomposition*. Here, we define this decomposition for *rooted trees*, which simplifies the presentation considerably. The basic idea is to partition the input tree into $\Theta(\frac{n}{\log n})$ *clusters*, each of size $\mathcal{O}(\log n)$, and contract each of these parts into either a single vertex or two vertices connected by an edge to obtain a *cluster tree* of size $\Theta(\frac{n}{\log n})$.

Recall a basic $\mathcal{O}(n \log n)$ algorithm; call it $\mathsf{Alg}_1$. An $\mathcal{O}(n \log\log n)$ algorithm $\mathsf{Alg}_2$ can be obtained as follows: Construct the cluster tree in linear time, run $\mathsf{Alg}_1$ on this cluster tree, and the same algorithm on each cluster. The running time for the cluster tree algorithm is $\mathcal{O}(\frac{n}{\log n} \log \frac{n}{\log n}) = \mathcal{O}(n)$, and the running time for each cluster with $k \leq \log n$ vertices is $\mathcal{O}(k \log k)$, for a total of $\mathcal{O}(n \log\log n)$. This approach can be iterated: an algorithm $\mathsf{Alg}_t$ is implemented the same way, only that $\mathsf{Alg}_{t-1}$ is used for each of the clusters. Iterating this approach $\log^* n$ times gives an algorithm with $\log^* n$ "levels", each of which takes $\mathcal{O}(n)$ time in total; thus, the total running time of $\mathsf{Alg}_{\log^* n}$ is $\mathcal{O}(n \log^* n)$.

Of course, the above presentation is heavily simplified. We will note one interesting detail. Performing a `cut` within a cluster may affect the weights stored in the cluster tree, necessitating an `update-weight` operation on the cluster tree. Hence, the number of operations in the cluster tree can be $\Theta(n)$, instead of $\Theta(\frac{n}{\log n})$, as we assumed above. The solution to this is to use an algorithm where each `update-weight` operation takes only $\mathcal{O}(1)$ time (whereas `cut`s each take $\Theta(\log n)$ amortized time). Note that standard dynamic tree solutions require $\Theta(\log n)$ per operation, and thus are not suitable for this purpose. Instead, we design a simple decremental-only algorithm $\mathsf{Alg}_1$ with this property.

**Unweighted and 0-1-weighted tree-sums.** For unweighted forests, we can improve the above algorithm (Section 4). In fact, our solution works when the weight function only takes values zero and one.[1]

▶ **Theorem 1.2.** *There is a data structure in the Word RAM model of computation that maintains a 0-1-weighted rooted forest $(F, w)$ with $n$ vertices under* `cut`*,* `update-weight` *and* `tree-sum` *operations, in $\mathcal{O}(m+n)$ time for $m$ operations.*

[1] The underlying group is the set of integers, so if all weights are one, we recover the unweighted tree-sum problem.

We describe the algorithm for the unweighted case. The idea is to apply the cluster decomposition above only twice; then the bottom-level clusters form a decomposition of the input tree into $\mathcal{O}(\frac{n}{\log\log n})$ small trees of size at most $k \leq \mathcal{O}(\log\log n)$. Note that the number of distinct (labeled) trees of size at most $k$ is no more than $k^{\mathcal{O}(k)} \ll \frac{n}{\log\log n}$. Thus, we can essentially precompute all answers for every possible such tree, and then do a lookup for each cluster.

This technique is commonly used in algorithm design [5, 10, 6, 7]. For a *data structure* this is not quite as straightforward, since the queries are not known at the beginning. Still, we can compute something akin to a *state graph* that stores all necessary information, and can be navigated efficiently.

**An optimal weighted tree-sum algorithm.** We can take the idea of precomputation even further (Section 5), based on a technique of *precomputing decision trees* used by Pettie and Ramachandran [29] (also see e.g. [26, 27] for different applications of this approach). Perform the cluster decomposition three times, obtaining trees of size at most $k \leq \mathcal{O}(\log\log\log n)$. For these very small trees, we can precompute the *optimal data structure* in time $o(n)$. If $\mathrm{OPT}(n, m)$ is the optimal total running time for $m$ operations on a forest with $n$ vertices, then our algorithm has running time

$$\mathcal{O}(n+m) + \sum_i \mathrm{OPT}(n_i, m_i),$$

where $n_i \leq k$ is the number of vertices in the $i$th cluster, and $m_i$ is the number of operations applied to that cluster. Note that $\sum n_i \leq n$ and $\sum m_i \leq m$.

With some effort (Section 5.1), we can argue that the above formula is upper bounded by $\mathcal{O}(n + m + \mathrm{OPT}(n, m))$, since the function OPT satisfies a weak, asymptotic form of *super-additivity*: $\sum_i \mathrm{OPT}(n_i, m_i) \leq \mathcal{O}(\mathrm{OPT}(\sum_i n_i, \sum_i m_i) + \sum_i n_i)$ for any pair of positive integer sequences $\{n_i\}$, $\{m_i\}$. Interestingly, even this weak variant of super-additivity is quite challenging to prove. The proof proceeds via a series of reductions involving data structures for `tree-sum` that are initialized with forests with *zero weights*. The rationale is that $\mathrm{OPT}_0(n, m)$ – the function denoting the optimal total running time for $m$ operations on a *zero-initialized* forest with $n$ vertices – can be directly shown to be super-additive, i.e., $\sum_i \mathrm{OPT}_0(n_i, m_i) \leq \mathrm{OPT}_0(\sum_i n_i, \sum_i m_i)$.

We finally obtain

▶ **Theorem 1.3.** *There exists a data structure with running time $\mathcal{O}(\mathrm{OPT}(n, m) + n + m)$ for any forest on $n$ vertices and any sequence of $m$ operations.*

Note that our data structure is as fast (up to constant) as any other data structure, even if the latter has $n$ and $m$ hard-coded into it. It turns out we can prove something even stronger: Our data structure is optimal even compared to other data structures with a hard-coded *initial forest* $F$. For such data structures, the only non-hard-coded parts of the input are the initial weights and the sequence of operations. This is precisely the notion of *universal optimality*, popularized by works of Haeupler, Wajc, and Zuzic [21] and Haeupler, Hladík, Rozhon, Tarjan, and Tetek [20], and we adopt it here as well. Universally optimal algorithms have been recently studied for various *static* problems, such as Single-Source Shortest Paths [21, 20, 36], planar convex hull [1, 34] and sorting [19, 35]. A recent work of the first author presents a universally optimal data structure for the related *tree-min* problem [9].

We now discuss some details of Theorem 1.3, in particular in comparison with Pettie and Ramachandran's famous optimal minimum spanning tree (MST) algorithm [29]. They similarly reduce a large instance to many smaller instances, though their reduction is much

more complex. On the other hand, for them, computing the optimal algorithm for small instances is quite straightforward, since every MST algorithm in the comparison model can be represented as a decision tree. Thus, they only need to enumerate all decision trees for the given small graph, and choose the best correct one.

In contrast, the tree-sum problem is an *online* problem in the *group model*, which presents some interesting challenges. For example, testing correctness of a decision tree computing the MST of a static $k$-vertex graph is easy (just check all possible graphs, and for each graph, all possible input weight orders), but testing correctness of an algorithm (or a data structure) in the group model is harder. Another interesting problem is that the number operations applied to a single cluster is initially unknown, and can be much larger than the size of the cluster. Dealing with these problems requires some new insights.[2]

Finally, Pettie and Ramachandran note that an explicit provably optimal algorithm for MST was actually known before. An argument of Jones [24] transforms any optimal verification algorithm for a computational problem (which is known for MST) into an optimal algorithm. Interestingly, Jones's argument does not work for the online data structure problems like ours. Thus, our optimality result in itself is novel. In any case, this approach yields massive constants in the running time and does not imply anything about the *decision tree complexity* of MST. In contrast, the running time of Pettie and Ramachandran's algorithm matches the decision tree complexity. Theorem 1.3 similarly matches the optimal number of additions and subtractions, up to constant factors. That means that any algorithm that only performs a linear number of additions and subtractions in total (with arbitrary running time) implies a data structure with linear total running time.

**Subtree-sums.** Finally, we consider the harder subtree-sum problem. In its full generality, with support for `subtree-sum`, `update-weight`, and `cut`, it is a generalization of the well-known partial-sum problem [8, 11, 13, 15, 14, 37]. In the partial-sum problem, we are tasked to maintain an array of weights under the operations of computing the sum of a prefix and changing an entry. Clearly, we can interpret the array as a degenerate path-like rooted tree. Then partial sums are `subtree-sum` operations and entry changes are `update-weight` operations (`cut` operations are not used). For the partial-sum problem, an $\Omega(n \log n)$ lower bound was shown by Pătraşcu and Demaine [28], so we have:

▶ **Observation 1.4.** *Each data structure that maintains a weighted forest with $n$ vertices under the operations* `subtree-sum` *and* `update-weight` *requires $\Omega(n \log n)$ time for $n$ operations, even in the Word RAM model of computation and assuming $G = \mathbb{Z}$.*

Much more interesting is the case where we do not allow `update-weight` operations, only `subtree-sum` and `cut`. In Section 6, we show a randomized reduction from the lower bound of [28], asserting the $\Omega(n \log n)$ lower bound in this restricted setting:

▶ **Theorem 1.5.** *Each data structure that maintains a weighted forest with $n$ vertices under the operations* `subtree-sum` *and* `cut` *requires $\Omega(n \log n)$ time for $\Theta(n)$ operations in the Word RAM model of computation and assuming $G = \mathbb{Z}$.*

On a high level, we design a hard instance for the problem as follows. We show that the $\Omega(n \log n)$ lower bound of Pătraşcu and Demaine also holds under the assumption that the underlying array contains $\sqrt{n}$ integers ranging from 0 to $\Theta(\sqrt{n})$, with each element of the

[2] In a different paper, Pettie and Ramachandran also give a provably optimal algorithm for an online data structure problem [30, Appendix B]. This is in the comparison model, so some of our challenges do not apply. Additionally, there appears to be an error in that paper that hides some of the complexity; see the full paper for a discussion.

array updated $\Theta(\sqrt{n})$ times. The weighted forest in our proof consists then of a spine of $\sqrt{n}$ vertices – each representing an element of the array – with $\sqrt{n}$ leaves of various weights attached to each vertex of the spine, so that a partial sum of $i$ initial elements of the array can be inferred from the `subtree-sum` of the subtree rooted at the $i$th vertex of the spine. We also show a Las Vegas randomized subroutine that efficiently translates an update of the $i$th element of the array to a sequence of $\mathcal{O}(1)$ `cut`s of leaves that are still attached to the $i$th vertex of the spine. This way, any sequence of $\Theta(n)$ operations in a `partial-sum` data structure is interpreted as a sequence of $\Theta(n)$ operations in the `subtree-sum` structure, and the $\Omega(n \log n)$ lower bound for the former implies the same lower bound for the latter.

With Observation 1.4 and Theorem 1.5 in mind, we study the *unweighted* variant of the problem (or, more precisely, the *0-1-weighted* variant with no weight updates). Here, we are able to obtain an improved (amortized) running time of $\mathcal{O}(\frac{\log n}{\log \log n})$ per operation (Section 7.2). Again, we state the slightly more general case of 0-1 weights:

▶ **Theorem 1.6.** *There is a data structure in the Word RAM model maintaining a 0-1-weighted forest with $n$ vertices under the operations* `subtree-sum` *and* `cut`*, with $\mathcal{O}((m+n)\frac{\log n}{\log \log n})$ total time for $m$ operations.*

Theorem 1.6 again uses recursive cluster decompositions, but with larger clusters: Instead of $\mathcal{O}(\frac{n}{\log n})$ clusters of size $\mathcal{O}(\log n)$, we use roughly $k = \sqrt{\log n}$ clusters of size $\mathcal{O}(\frac{n}{k})$. Recursively applying this decomposition until clusters have constant size yields a decomposition of depth $\mathcal{O}(\log_k n) = \mathcal{O}(\frac{\log n}{\log \log n})$. Each cluster tree obtained in this process has size at most $k$, which notably implies that it can be encoded with $\mathcal{O}(\log n)$ bits, which fits into a single word. This allows us to combine a delayed-updating technique of Dietz [11] with some precomputation to obtain a fast data structure for these very small cluster trees.

Note that the paper of Dietz [11] cited above solves a variant of our problem where the initial tree is a 0-1-weighted path updated by `update-weight` rather than `cut`, with the same $\mathcal{O}((m+n)\frac{\log n}{\log \log n})$ total time; this data structure is known to be optimal due to Fredman and Saks [14]. On the other hand, the path variant of Theorem 1.6 (reporting `subtree-sum`s in an unweighted or 0-1-weighted path with $n$ vertices updated by `cut`) is much easier – it can be solved in $\mathcal{O}(m+n)$ time, e.g. via a reduction to the *marked ancestor problem* [3]. In contrast, we show in Section 7.1 that for general forests, our problem is as hard as the path variant with weight updates.

▶ **Theorem 1.7.** *Each data structure in the Word RAM model that maintains an unweighted forest with $n$ vertices under the operations* `subtree-sum` *and* `cut` *requires $\Omega(n\frac{\log n}{\log \log n})$ time for $n$ operations.*

In other words, the data structure of Theorem 1.6 is asymptotically optimal.

## 1.2 Outlook to the offline setting

We can consider the decremental `tree-sum` problem in an *offline* setting, where the entire sequence of $m$ operations is provided to the algorithm together with the initial weighted $n$-vertex forest at the time of initialization. It turns out that the offline variant of the problem can be solved in $\mathcal{O}(n+m)$ time: Simulating the sequence of operations backwards, we reduce `tree-sum` to the incremental variant – that is, the disjoint set union problem – where the *merge forest* (the forest containing an edge $uv$ for every merge of sets containing vertices $u$ and $v$) is known in advance. This simplified variant of the problem is solvable in linear time in the Word RAM model by a data structure of Gabow and Tarjan [17]. Moreover, reporting `tree-sum`s requires additionally only $n-1$ operations in the group $G$: Whenever two sets (i.e., vertex sets of two tree components of a forest) are merged, the tree sum of the resulting tree components is simply the sum of the tree sums of the components being merged.

This discussion unveils an interesting phenomenon: In the online incremental setting of `tree-sum`, it is the maintenance of connected tree components that is strictly more computationally expensive (requiring on average $\Omega(\alpha(n))$ time per edge insertion) than the additional bookkeeping of the tree sums ($\mathcal{O}(1)$ time per insertion). However, the opposite appears to be the case in the decremental setting: The tree components can be maintained in amortized constant time per edge removal, but it is not clear at all whether tree sums can be tracked this efficiently.

## 2 Preliminaries

For $k \in \mathbb{N}$, define $[k] = \{1, 2, \dots, k\}$. All logarithms in this work are binary, unless specified otherwise. For convenience, assume that $\log x = 0$ for all $x \leq 1$. Then for $k \in \mathbb{N}_+$, we define the $k$-fold logarithmic function $\log^{[k]} x$: $\log^{[1]} x = \log x$ and $\log^{[k]} x = \log(\log^{[k-1]} x)$ for $k > 1$. Finally, the iterated logarithm, $\log^* x$, is defined as the smallest $k$ such that $\log^{[k]} x \leq 1$.

All trees and forests in this work are rooted. In a tree $T$ and vertices $u$ and $v$, we say that $u$ is an ancestor of $v$ if $u$ lies on the unique path from the root of $T$ to $v$. In this case, we also say that $v$ is a descendant of $u$. Strict ancestors and descendants are defined analogously, only that we additionally require that $u \neq v$. For a tree $T$ and its vertex $v$, we denote by $T_v$ the subtree of $T$ rooted at $v$, i.e., the subtree induced by all descendants of $v$. The *depth* of a vertex $v$, denoted by $\mathrm{depth}_T(v)$, is its distance to the root (0 for the root itself), and the height of $T$ is the maximum depth of a vertex.

We assume the standard Word RAM model of computation with memory cells (*words* or *registers*) of size $b = \Theta(\log n)$ [16]. Formally, the memory is represented by an array $R$ of $b$-bit integers, with access to standard arithmetic and bitwise operations, as well as comparisons and indirect accesses (of the form $R[R[i]] \leftarrow R[j]$). Each such access takes constant time.

Most of the algorithms in this paper work in the *group model*. In this model, some memory cells may contain – instead of a $b$-bit integer – an element of a commutative group $G$. The elements of $G$ can be manipulated in constant time via addition and subtraction, and the comparisons between the elements of $G$ (including the zero-comparisons of the elements of $G$) are disallowed. In particular, it is forbidden to examine or modify the memory representation of any memory cell containing an element of $G$. The weights of vertices of $F$ come from the group $G$. For convenience, we will denote by $W[v]$ the memory cell occupied by the current weight $w(v)$ of a vertex $v$.

If $F$ is a forest with a weight function $w : V(F) \to G$, then for $U \subseteq V(F)$ we define $w(U) = \sum_{v \in U} w(v)$. Similarly, if $T$ is a (sub)tree of $F$, then $w(T) = w(V(T))$. If $S \subseteq V(F)$, then we denote by $F[S]$ the subgraph of $F$ induced by $S$, and by $w|_S : S \to G$ the restriction of the weight function $w$ to the vertices in $S$.

**Decremental connectivity.** Our data structures rely heavily on the Word RAM data structures supporting connectivity queries in decremental forests. Such a data structure, when initialized with a forest $F$, supports the following updates and queries:

- `cut`$(v)$: Delete the edge between $v$ and its parent (we assert that this edge exists).
- `root`$(v) \to r$: Return the root of the tree containing $v$.
- `connected`$(u, v) \to$ `bool`: Indicate whether $u$ and $v$ are in the same component of $F$.
- `ancestor`$(u, v) \to$ `bool`: Indicate whether $u$ is an ancestor of $v$ in $F$ (which implies they are connected).

As mentioned before, in the Word RAM model, there exists a linear-time data structure for decremental connectivity of forests supporting `cut` and `root` [4]. It is easily adapted to our needs (see the full version for a proof).

▶ **Lemma 2.1** ([4]). *There is a Word RAM data structure that maintains a forest with $n$ vertices, and supports* `cut`*,* `root`*,* `connected`*, and* `ancestor` *in $\mathcal{O}(n+m)$ time for $m$ operations.*

**Binarization and auxiliary vertices.** In our algorithms, it will be convenient to transform the input trees into *binary* trees, in which each vertex has at most two children. It is well-known that this can be done by transforming high-degree vertices into binary trees, which introduces a linear number of additional vertices. For most applications, the fact that the number of vertices increases by only a constant factor is sufficient to retain desired running times. However, for our optimal algorithm in Section 5, inserting additional vertices causes technical problems. To address this, we introduce the concept of *auxiliary vertices*.

Let $F$ be the input forest for one of our data structures. If a vertex $v$ in $F$ is marked as auxiliary, then no operations may be applied to $v$ (e.g., `cut`$(v)$ and `update-weight`$(v,\cdot)$ are illegal). Moreover, if a weight function is associated to $F$, then $w(v)=0$. In the interest of brevity, we still write just $F$ for a forest with auxiliary vertices; it will be made clear at the start of each relevant section whether algorithms allow auxiliary vertices or not.

Using auxiliary vertices, we can transform any forest into a binary forest that is equivalent for our data structure problems.

▶ **Theorem 2.2.** *Let $(F,w)$ be a weighted forest. Then, in linear time, we can compute a* binary *weighted forest $(F',w')$, such that $V(F)\subseteq V(F')$, $|V(F')|\leq 2|V(F)|$, every sequence of operations* `cut`*,* `update-weight`*,* `tree-sum`*, and* `subtree-sum` *is legal in $(F,w)$ if and only if it is legal in $(F',w')$, and every such sequence yields the same results on $(F,w)$ and $(F',w')$.*

*Moreover, $w'(v)=w(v)$ for each $v\in V(F)$ and $w'(v)=0$ for each $v\in V(F')\setminus V(F)$.*

**Proof.** For every vertex $v\in V(F)$ with $d_v\geq 3$ children, we replace $v$ with a rooted path $P_v$ consisting of a root $v$ and $d_v-1$ auxiliary vertices of weight 0. Each child of $v$ then becomes a child of a different vertex in $P_v$. This transformation can be done in linear time, and it clearly satisfies all requirements of the theorem. ◀

## 2.1 The cluster decomposition

We now formally describe a decomposition of a binary tree $T$ into small disjoint well-structured subgraphs of $T$, which we call the *cluster decomposition* of $T$. The decomposition is inspired by the work of Alstrup et al. [4], however our variant of the decomposition will be particularly convenient to use in the upcoming data structures.

Let $T$ be a tree, and let $C\subseteq V(T)$ induce a connected subtree of $T$. A vertex $v\in C$ is called an *upper boundary vertex* if the parent of $v$ is not contained in $C$, or $v$ is the root of $T$. Note that each connected set $C\subseteq V(T)$ has precisely one upper boundary vertex, which we denote by $\mathrm{ub}(C)$. A vertex in $C$ is called a *lower boundary vertex* if it has at least one child that is not contained in $C$. We call $C$ a *cluster* if it has at most one lower boundary vertex, and moreover, no child of the lower boundary vertex is in $C$. Let $\mathrm{lb}(C)$ denote the lower boundary vertex of a cluster $C$, if it exists, or let $\mathrm{lb}(C)=\perp$ otherwise. Note that a single vertex is always a valid cluster.

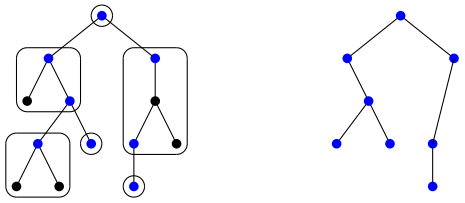

**Figure 1** A cluster decomposition (left) and the associated cluster tree (right).

A *cluster decomposition* (see Figure 1) of a tree $T$ is a partition of $V(T)$ into a set $\mathcal{C}$ of disjoint clusters. The *cluster tree* of $\mathcal{C}$ is a tree $S$ where $V(S)$ are all boundary vertices of the clusters in $\mathcal{C}$. A vertex $u \in V(S)$ is a child of $v \in V(S)$ if one of the following is true:

- There is a cluster $C \in \mathcal{C}$ with $u = \mathrm{lb}(C)$ and $v = \mathrm{ub}(C)$, or
- $u$ and $v$ are in different clusters and $u$ is a child of $v$ in $T$ (then $u$ must be an upper boundary vertex and $v$ must be a lower boundary vertex).

Note that every vertex in $V(S)$ has a parent, except the root $r$ of $T$. Thus, the cluster tree is indeed a (rooted) tree. Note that the cluster tree contains no auxiliary vertices, regardless of whether $T$ contains auxiliary vertices or not.

▶ **Lemma 2.3.** *Let $T$ be a binary tree with $n$ vertices, and let $k \leq n$. Then, in linear time, we can compute a cluster decomposition of $T$ into at most $6\frac{n}{k}$ clusters, each of size at most $k$, along with the corresponding cluster tree.*

A proof of Lemma 2.3 is found in the full version of the paper.

Both of our main algorithms use a cluster decomposition $\mathcal{C}$ of the input tree. The decomposition $\mathcal{C}$ itself is never modified, even when clusters become disconnected. However, we will modify the cluster tree by removing edges; for this, the following definition will be useful.

Let $T$ be a tree, let $\mathcal{C}$ be a cluster decomposition of $T$, and let $S$ be the corresponding cluster tree. For every forest $F$ obtained by removing some set of edges from $T$, we define the *cluster forest induced by $F$* as the forest $G$ with $V(G) = V(S)$, such that $u$ is a child of $v$ in $G$ if:

- There is a cluster $C \in \mathcal{C}$ with $u = \mathrm{lb}(C)$ and $v = \mathrm{ub}(C)$, and $u, v$ are connected in $F$, or
- $u$ and $v$ are in distinct clusters and $u$ is a child of $v$ in $F$.

Note that $G$ can be obtained from $S$ by removing edges. More specifically, consider a sequence of edge removals within $T$ that produces $F$. To construct $G$, we do the following: Whenever an edge between the boundary vertices of two clusters is removed, the corresponding two vertices in $G$ are likewise disconnected. Whenever an edge within a cluster $C$ is removed, if this disconnects $\mathrm{lb}(C)$ and $\mathrm{ub}(C)$, the corresponding edge in $G$ is deleted. Since connectivity can be maintained in constant time per operation (Lemma 2.1), it is easy to construct the following data structure:

▶ **Lemma 2.4.** *Suppose we are given a tree $T$ on $n$ vertices, and a cluster decomposition $\mathcal{C}$ of $T$. There is a data structure that maintains the induced cluster forest under (up to $n-1$) edge deletions in $\mathcal{O}(n)$ total time.*

## 3 Tree sums in $\mathcal{O}(n \log^* n)$ time

In this section, we prove:

▶ **Theorem 1.1.** *There is a data structure that maintains a weighted rooted forest $(F, w)$ on $n$ vertices under* `cut`*,* `update-weight`*, and* `tree-sum`*, with running time $\mathcal{O}((m+n)\log^* n)$ for $m$ operations.*

We start with a simple $\mathcal{O}(n \log n + m)$-time data structure (Section 3.1), which we then iterate using cluster decompositions (Section 3.2).

For technical reasons, our data structures need to support a slightly modified `cut` operation: `cut-report`$(v)$ performs a `cut`$(v)$, but additionally returns the results of `tree-sum`$(v)$ and `tree-sum`$(u)$, where $u$ is the (former) parent of $v$. This will be useful to avoid extra calls to `tree-sum` that complicate the running time analysis.

In this section, let a *tree-sum data structure* refer to a data structure that maintains a *binary* weighted forest, potentially with auxiliary vertices, under `tree-sum`, `update-weight`, and `cut-report`. By Theorem 2.2, this also implies an analogous data structure for general weighted forests.

## 3.1 A simple algorithm with constant-time queries and weight updates

As mentioned in the introduction, well-known dynamic forest data structures can solve the tree-sum problem with $\mathcal{O}(\log n)$ time per operation. The data structure presented next takes $\Theta(\log n)$ (amortized) time per `cut`, but only $\mathcal{O}(1)$ time for `tree-sum` and `update-weight`, which is crucial to prove Theorem 1.1. The data structure is a straightforward adaptation of [12, Section 2]. We provide a proof in the full version of the paper.

▶ **Lemma 3.1.** *There is a tree-sum data structure with the following running times for $n$ vertices: $\mathcal{O}(n)$ for initialization, $\mathcal{O}(1)$ for each* `tree-sum` *and* `update-weight`*, and $\mathcal{O}(n \log n)$ in total for all* `cut-report` *operations.*

## 3.2 Iterating the algorithm

We now show how to build data structures with running times of roughly $\log \log n$, $\log \log \log n$, etc. per operation. The following reduction using cluster decompositions is the main necessary ingredient.

▶ **Lemma 3.2.** *Suppose there is a tree-sum data structure $X$ with total running time $f(n', m')$ for $n'$ vertices and $m'$ operations. Then, there exists a tree-sum data structure with the following property. For each number of operations $m$ and number of vertices $n$, there exists an integer $k \in \mathbb{N}_+$ and integers $m_i \in \mathbb{N}_0$, $n_i \in \mathbb{N}_+$ for $i \in [k]$ with $\sum_{i=1}^{k} n_i \leq n$, $\sum_{i=1}^{k} m_i \leq m$, and $n_i \leq \log n$ for all $i \in [k]$, such that the total running time on any instance with $n$ vertices and $m$ operations is at most*

$$\mathcal{O}(n+m) + \sum_{i=1}^{k} f(n_i, m_i).$$

The numbers $n_i$ in Lemma 3.2 refer to the cluster sizes, and $m_i$ to the number of operations applied to a particular cluster. The lemma then essentially states that each operation is delegated to a single cluster, with overall linear overhead.

In the remainder of this section, we prove Lemma 3.2. For simplicity, we assume the initial forest is a tree; otherwise, if the forest is disconnected, we can apply the data structure separately for each connected component, and use Lemma 2.1 to find the proper data structure to query or modify.

Let $(T^0, w_0)$ be the initial weighted binary tree. We first use Lemma 2.3 with the maximum cluster size $k = \log n$ to compute a cluster decomposition $\mathcal{C}$ of $T^0$ with cluster tree $S$. We maintain the induced cluster forest $G$ as described by Lemma 2.4.

We also maintain a weight function $w'$ on the vertices in the cluster forest $G$, as follows. Suppose $(F, w)$ is the current weighted forest. Let $C$ be some cluster, and let $u = \mathrm{ub}(C) \in V(G)$. Then $w'(u)$ is defined as the sum of weights of all vertices in $C$ that are connected to $u$. On the other hand, if $v = \mathrm{lb}(C)$ exists, then $w'(v)$ is defined as the sum of weights of all vertices in $C$ that are connected to $v$, but not to $u = \mathrm{ub}(C)$. In particular, note that $w'(v) = 0$ as long as $v$ is connected to $u$.

▶ **Observation 3.3.** *Let $v \in V(G)$ be a boundary vertex of some cluster, let $T$ be the tree in $F$ containing $v$, and let $U$ be the tree in $G$ containing $v$. Then $w(T) = w'(U)$.*

In the following, let $F$ denote the current forest, and let $G$ denote the cluster forest induced by $F$. Let $w$ denote the current weight function on $F$, and let $w'$ denote the weight function on $G$ defined as above. Our data structure maintains $(F, w)$ and $(G, w')$ explicitly (with child and parent pointers), and additionally:

- A data structure $R$ for `connected` queries in $F$ (Lemma 2.1).
- An instance $D$ of the data structure from Lemma 3.1 on $(G, w')$.
- A *cluster object* for each cluster $C$, which stores $\mathrm{ub}(C)$ and $\mathrm{lb}(C)$.
- For each cluster $C$, an instance $X_C$ of the given data structure $X$ (defined in Lemma 2.3) on the induced subforest $F[C]$, with the weight function $w$ restricted to $C$.
- For each vertex, a pointer to the cluster object containing it.

Initialization works as follows. First compute $\mathcal{C}$ and $G$ using Lemma 2.3, with $k = \log n$. Create the cluster objects and point each vertex towards its cluster. Then initialize $R$, $D$, and all instances $X_C$. Finally, compute $w'$ by summing the weights of each cluster and assigning this sum to the upper boundary vertex (recall that all clusters are still connected). We now describe the three operations.

- Consider an operation `tree-sum`$(v)$. We first try to find a boundary vertex that is connected to $v$, by determining its cluster $C$, and then checking connectivity between $v$ and $\mathrm{ub}(C)$, resp. $\mathrm{lb}(C)$, using $R$. If $v$ is not connected to either, then the component of $F$ containing $v$ is entirely within $C$. Thus, we can return $X_C$.`tree-sum`$(v)$. Otherwise, if $v$ is connected to some $u \in \{\mathrm{ub}(C), \mathrm{lb}(C)\}$, then we can return $D$.`tree-sum`$(u)$ by Observation 3.3.
- Consider `update-weight`$(v, x)$. Say $y$ is the weight $w(v)$ before the operation. We again first determine the cluster $C$ containing $v$, and call $X_C$.`update-weight`$(v, x)$. Further, the change to $w$ may affect $w'$ for either $\mathrm{ub}(C)$ or $\mathrm{lb}(C)$. We again check connectivity between $v$ and the two boundary vertices. If $v$ is connected to $\mathrm{ub}(C)$, then we increase $w'(\mathrm{ub}(C))$ by $x - y$. Otherwise, if $v$ is connected to $\mathrm{lb}(C)$, we increase $w'(\mathrm{lb}(C))$ by $x - y$. If neither is true, $w'$ does not change for any vertex.
- Consider `cut-report`$(v)$. Let $u$ be the parent of $v$ (before the operation). Suppose first that $u$ and $v$ are in distinct clusters. Then $u$ is a lower boundary vertex and $v$ is an upper boundary vertex. Thus, we can simply call $D$.`cut-report`$(v)$ and pass along the two returned values. No cluster is changed, so $w'$ and $X_C$ do not need to be touched.
  Second, suppose that $u$ and $v$ are both contained in a single cluster $C$. This still may remove an edge from $G$; we check this as in Lemma 2.4 and call $D$.`cut` if necessary. We then call $X_C$.`cut-report`$(v)$, which returns the sums $x_u$ and $x_v$ of weights of vertices in $C$ connected to $u$, resp. $v$. Now $w'$ could have changed for $\mathrm{ub}(C)$ or $\mathrm{lb}(C)$, and we

need to update it in $D$. Recalling the definition of $w'$, it is straightforward to determine $w'(\mathrm{ub}(C))$ and $w'(\mathrm{lb}(C))$ by checking connectivity between $\mathrm{ub}(C)$, $\mathrm{lb}(C)$, $u$, and $v$, and then using $x_u$ or $x_v$ if necessary. If $w'$ changes, we call $D$.`update-weight` appropriately. Finally, we need to return the results for `tree-sum`$(u)$ and `tree-sum`$(v)$. This is again straightforward to compute: If connected to $\mathrm{ub}(C)$ or $\mathrm{lb}(C)$, we can use $D$.`tree-sum`; if not, we can use $x_u$ or $x_v$, respectively.

Note that we never call operations on auxiliary vertices in $X_C$, since by assumption we are never given an auxiliary vertex as parameter $v$.

We now bound the running time. Initialization, aside from the instances $X_C$, takes time $\mathcal{O}(n)$. Besides calls to $X_C$ and $D$, each operation performs a constant amount of work, for a total of $\mathcal{O}(m)$ across $m$ operations. By Lemma 3.1, the time spent in $D$ is $\mathcal{O}(m + |V(G)| \log |V(G)|) \leq \mathcal{O}(m + n)$.

Finally, consider the time spent in a single instance $X_C$. By definition, this is $f(n_i, m_i)$, where $m_i$ is the number of operations called on $X_C$, and $n_i = |C|$. It remains to show that the stated conditions on $m_i$ and $n_i$ are true. Clearly, the total size of all clusters is $n$, so $\sum n_i = n$. Also, each cluster has size at most $\log n$, so $n_i \leq \log n$. Further, observe that each operation (in the data structure we just described) results in at most one call to an operation of $X_C$. Thus, we have $\sum m_i \leq m$. This concludes the proof of Lemma 3.2.

▶ **Corollary 3.4.** *There is a tree-sum data structure, parameterized with $t \in \mathbb{N}_+$, with total running time $\mathcal{O}(t(m+n) + n \log^{[t]} n)$ for $m$ operations and $n$ vertices.*

**Proof.** Use Lemma 3.1 for the case $t = 1$. For $t \geq 2$, use Lemma 3.2 and the data structure constructed for $t - 1$. By induction, the running time of the latter is, up to an absolute multiplicative constant, bounded by $f(m', n') = t(m' + n') + n' \log^{[t]} n'$ for $m'$ operations and $n'$ vertices. The running time of the new data structure then is, up to an absolute multiplicative constant, at most

$$m + n + \sum_{i=1}^{k} (t-1)(m_i + n_i) + n_i \log^{[t-1]} n_i \leq t(m+n) + n \log^{[t]} n,$$

using that $\sum m_i = m$, $\sum n_i = n$, and $n_i \leq \log n$. ◀

With $t = \log^* n$, Corollary 3.4 implies Theorem 1.1.

## 4 Tree sizes in linear time

In this section, we prove:

▶ **Theorem 1.2.** *There is a data structure in the Word RAM model of computation that maintains a 0-1-weighted rooted forest $(F, w)$ with $n$ vertices under* `cut`, `update-weight` *and* `tree-sum` *operations, in $\mathcal{O}(m+n)$ time for $m$ operations.*

Let a *tree-size data structure* be a data structure as in Theorem 1.2, which supports auxiliary vertices in the input. The following lemma is the centerpiece of the data structure. See the full version of the paper for a proof.

▶ **Lemma 4.1.** *Fix a machine word size $b$ and some $\ell \leq \frac{b}{3+\log b}$. There is a tree-size data structure that maintains a 0-1-weighted forest on up to $\ell$ vertices, with $\mathcal{O}(\ell)$ initialization time and $\mathcal{O}(1)$ time per operation.*

*The data structure requires a global table, only depending on $\ell$, that can be precomputed in time $\mathcal{O}((5\ell)^\ell)$.*

We can now combine Lemma 4.1 with Lemma 3.2 to prove Theorem 1.2. The only additional observation necessary is that the reduction of Lemma 3.2 preserves the property that $w$ is a 0-1 weight function. Internally, the more general weight function $w'$ is used only for the data structure $D$ (which uses Lemma 3.1), not for the data structures $X_C$.

**Proof of Theorem 1.2.** Start by precomputing the global table of Lemma 4.1, with $\ell = \lfloor \log \log n \rfloor$. This takes $o(n)$ time. We now have a data structure with running time $f(m', n') \in \mathcal{O}(m' + n')$ for $m'$ operations and $n'$ vertices, but only if $n' \leq \log \log n$. Applying Lemma 3.2 gives us a data structure with running time $\mathcal{O}(m' + n')$ for all forests with $n' \leq \log n$ vertices, and applying it the second time yields the claim. ◀

## 5 Tree sums in optimal time

We now present an explicit data structure with unknown, but optimal running time. In this section, let a *tree-sum* data structure support operations `cut`, `tree-sum`, and `update-weight`; auxiliary vertices are again allowed. Unlike in Section 3, we are not restricted to binary input trees, and do not require support for `cut-report`.

Let $D$ be a tree-sum data structure and let $F$ be a forest. Let $\mathbb{T}(D, F, m)$ be the maximum number of additions and subtractions that $D$ performs when executing a legal sequence of $m \in \mathbb{N}_0$ operations with the initial forest $F$. The maximum is thus taken over the operation sequences and the initial weights. Let $\mathrm{OPT}(F, m)$ be the minimum $\mathbb{T}(D, F, m)$ over all data structures $D$. Note that the quantity $\mathrm{OPT}(n, m)$ defined in the introduction is the maximum $\mathrm{OPT}(F, m)$ over all forests $F$ on $n$ vertices.[3] The main result of this section is:

▶ **Theorem 5.1.** *There exists a single data structure with running time $\mathcal{O}(\mathrm{OPT}(F, m)+m+n)$ for any forest $F$ on $n$ vertices and any sequence of $m$ operations. Note that the data structure does not know $F$, $n$, or $m$ upfront.*

This means that our data structure performs asymptotically as good as even the data structures that have the number of operations $m$ and the initial forest $F$ hard-coded. In particular, Theorem 5.1 implies Theorem 1.3.

As stated in the introduction, the main idea is to precompute an optimal data structure for very small trees.

▶ **Lemma 5.2.** *Given a forest $F$ with $n$ vertices, in time $2^{2^{n^{\mathcal{O}(1)}}}$, we can precompute a tree-sum data structure $D$ that performs $m \in \mathbb{N}_0$ operations with the initial forest $F$ (and arbitrary initial weights) in time $\mathcal{O}(\mathrm{OPT}(F, m) + m + n)$.*

Lemma 5.2 is shown in the full version of the paper. Our data structure (for large trees) works as follows. First, for each (ordered) binary tree $T$ on at most $k = \lfloor \log \log \log n \rfloor$ vertices, compute an optimal data structure $D_T$ for $T$ with Lemma 5.2. As is well known, there are no more than $\mathcal{O}(4^k) \leq \mathcal{O}(\log n)$ such trees. We can combine them into a single data structure $X$ with optimal running time (with linear overhead) for all binary trees of size at most $\log \log \log n$.[4]

Now take the components $S^1, S^2, \ldots, S^k$ of the input forest $F$. Binarize each of them using Theorem 2.2, obtaining binary trees $T^1, T^2, \ldots, T^k$. As usual, we build the helper data structure of Lemma 2.1 to identify the affected tree when performing an operation on the

[3] In the introduction, we did not allow auxiliary vertices; however, it is easy to see that marking a vertex as auxiliary cannot make the problem easier.

[4] When given an initial binary tree $T$, the combined data structure $X$ can identify the respective $D_T$ in $\mathcal{O}(|V(T)|)$ time, using a table containing each $D_T$.

overall forest. For each $T^i$, apply Lemma 3.2 three times, using $X$ as the data structure for small trees. Note that Lemma 3.2 requires $X$ to support `cut-report`, which we simulate with one `cut` and two `tree-sum` operations.

The running time for $m'$ operations on one of the trees $T^i$ is as follows, for some $k \in \mathbb{N}_+$, some partition of $T^i$ into (cluster) subtrees $T^{i,1}, T^{i,2}, \ldots, T^{i,k}$, and some $m_1, m_2, \ldots, m_k \in \mathbb{N}_0$ with $\sum_{j=1}^{k} m_j \leq m'$:

$$\mathcal{O}(|V(T^i)| + m') + \sum_{j=1}^{k} \mathrm{OPT}(T^{i,j}, 3m_j).$$

The factor 3 in $\mathrm{OPT}(T^{i,j}, 3m_j)$ is due to the fact that each `cut-report` operation results in three actual operations on $X$.

We now need the following lemma, which we will show in a moment in Section 5.1.

▶ **Lemma 5.3.** *Let $F$ be a forest, and let $T_1, T_2, \ldots, T_k$ be the trees induced by a partition of $V(F)$ into connected subsets. Further let $m_1, m_2, \ldots, m_k \in \mathbb{N}_0$. Then:*

$$\sum_{i=1}^{k} \mathrm{OPT}(T_i, 3m_i) \leq \mathcal{O}(\mathrm{OPT}(F, m) + n).$$

This bounds the running time for $m'$ operations on $T^i$ by $\mathcal{O}(|V(T^i)| + m' + \mathrm{OPT}(T^i, m'))$. Now observe that $\mathrm{OPT}(S^i, m') = \mathrm{OPT}(T^i, m')$, since by Theorem 2.2, all operation sequences return exactly the same results on $S^i$ and $T^i$. Applying Lemma 5.3 again on the components $S^1, S^2, \ldots, S^k$ yields the running time $\mathcal{O}(n + m + \mathrm{OPT}(F, m))$ for $m$ operations on $F$. Thus, we have shown Theorem 1.3.

## 5.1 Weak super-additivity of OPT

In this section, we prove Lemma 5.3. We need the following definitions. An *initial-zero tree-sum data structure* is a tree-sum data structure that only accepts initial forests where all weights are zero. (Thus, weights need to be changed before obtaining anything useful from `tree-sum` queries). Define $\mathrm{OPT}_0$ in the same way as OPT, but for initial-zero tree-sum data structures. It turns out that $\mathrm{OPT}_0$ is easier to work with than OPT. Indeed, we can show that $\mathrm{OPT}_0$ is *super-additive*.

In the following statement, we say that $U \subseteq V(F)$ is *convex* if for each $u, v \in U$, if $u$ and $v$ are connected in $F$, then they are also connected in $F[U]$. In other words, $U$ includes from each tree $T$ of $F$ either a connected subgraph of $T$, or nothing at all. In the initial-zero setting, this in particular implies that any sequence of operations in an instance $(F, w)$ that is legal in $(F[U], w|_U)$ yields the same results in both instances.

▶ **Lemma 5.4.** *Let $F$ be a forest, $U_1, U_2$ be a partition of $V(F)$ into convex subsets, and let $m_1, m_2 \in \mathbb{N}_0$. Then $\mathrm{OPT}_0(F[U_1], m_1) + \mathrm{OPT}_0(F[U_2], m_2) \leq \mathrm{OPT}_0(F, m_1 + m_2)$.*

**Proof.** Let $F_1 = F[U_1]$, let $F_2 = F[U_2]$, and let $m = m_1 + m_2$. Let $D$ be an optimal initial-zero tree-sum data structure for $F, m$, that is, the maximum number of additions and subtractions $D$ performs when handling any $m$ operations with the initial forest $F$ and zero weights is precisely $\mathrm{OPT}_0(F, m)$. We now give two data structures $D_1$ and $D_2$ for the forests $F_1$ and $F_2$, respectively, and then show that they perform at most $\mathrm{OPT}_0(F, m)$ additions and subtractions in total when presented with $m_1$ and $m_2$ operations, respectively.

First, $D_1$ is the same as $D$. In particular, $D_1$ behaves as if initialized with the original forest $F$, although it only accepts `cut`$(v)$, `update-weight`$(v, \cdot)$ or `tree-sum`$(v)$ if $v \in U_1$. Now take an operation sequence $X_1^*$ of length $m_1$ on $F_1$ such that the number of additions

and subtractions $D$ performs to execute $X_1^*$ is *maximized*. Run $D$ with $X_1^*$, and capture the final state of the data structure. Hard-code this state into the data structure $D_2$, except set the weights of all vertices in $F_1$ and all group elements stored by the data structure to zero. When presented with an operation sequence $X_2$ on $F_2$ (one operation at the time), $D_2$ continues the execution of $D$ from that final state.[5]

The correctness of $D_1$ is easy to see: Suppose $X_1$ is a sequence of $m_1$ operations on $F_1$. Consider a `tree-sum`$(v)$ query in $X_1$ (possibly after some `cuts`). In $D_1$, the query returns the sum of weights of some set $U \subseteq V(F_1)$. In $D$, it returns the sum of weights of some superset $U' \supseteq U$. However, since $U$ and $U'$ induce connected subgraphs of $F$, and $V(F_1)$ is convex, we have $U' \setminus U \subseteq V(F_2)$. Moreover, weights in $V(F_2)$ are zero at the start and never changed by $X_1$. Thus, the query returns the same result in both cases.

We can show correctness of $D_2$ in a similar, but slightly more complicated way. Consider a `tree-sum`$(v)$ query from $X_2$ applied to $D$, assuming $X_1^*$ and all previous queries in $X_2$ have been applied already. The return value $x$ of this query is computed by a series of additions and subtractions involving (previous or current) weights from $F_1$ and $F_2$. By ignoring all such additions and subtractions performed during the execution of $X_1^*$, we get that $x$ is a linear combination of (1) weights from $F_1$ and $F_2$, all read during the execution of $X_2$, and (2) values stored in the data structure directly after execution of $X_1$. Now recall that $D_2$ sets all those stored values to zero, and also sets all weights in $F_1$ to zero. Thus, what $D_2$ returns is the reduced linear combination with only the weights from $F_2$. As above, by convexity, this is precisely the correct result.

Having shown correctness, we now argue the stated running times. Let $X_1$ and $X_2$ be some operation sequences on $F_1$ and $F_2$, respectively. Recall that $\mathbb{T}(D', F', m')$ is the maximum number of additions and subtractions that a data structure $D'$ performs when executing $m' \in \mathbb{N}_0$ operations with an initial forest $F'$. For convenience, let also $\mathbb{T}(D', F', X')$ be the number of additions and subtractions that $D'$ performs when executing a specific sequence $X'$ of operations on $F'$. Note that $\mathbb{T}(D', F', m')$ is the maximum of $\mathbb{T}(D', F', X')$ over all sequences $X'$ of length $m'$.

By construction, we have $\mathbb{T}(D_1, F_1, X_1) = \mathbb{T}(D, F, X_1)$, and moreover $\mathbb{T}(D, F, X_1^*) + \mathbb{T}(D_2, F_2, X_2) = \mathbb{T}(D, F, X_1^* + X_2)$, and further $\mathbb{T}(D, F, X_1) \leq \mathbb{T}(D, F, X_1^*)$. These three inequalities imply that

$$\mathbb{T}(D_1, F_1, X_1) + \mathbb{T}(D_2, F_2, X_2) \leq \mathbb{T}(D, F, X_1^* + X_2) \leq \mathrm{OPT}_0(F, m).$$

Since this is true for any pair of sequences $X_1$ and $X_2$, we have

$$\mathrm{OPT}_0(F_1, m_1) + \mathrm{OPT}_0(F_2, m_2) \leq \mathrm{OPT}_0(F, m),$$

as desired. ◀

We will now generalize Lemma 5.4 to normal tree-sum data structures without the initial-zero assumption. We need two technical lemmas about OPT and $\mathrm{OPT}_0$. Note that both are far from trivial to show; see the full version of the paper for proofs.

▶ **Lemma 5.5.** *We have* $\mathrm{OPT}(F, m) \leq \mathrm{OPT}_0(F, 5m) + 2n$ *for each forest* $F$ *on* $n$ *vertices and* $m \in \mathbb{N}_0$.

[5] This formulation assumes that data structures have access to the zero element of the weight group. However, note that we can easily simulate additions with zero instead of actually storing zeroes.

▶ **Lemma 5.6.** *Let $F$ be a forest. For each constant $c > 1$, there is another constant $c'$ such that $\mathrm{OPT}(F, \lceil cm \rceil) \leq c' \cdot \mathrm{OPT}(F, m)$ for all $m \in \mathbb{N}_+$.*

With this, we are finally ready to prove:

▶ **Lemma 5.3.** *Let $F$ be a forest, and let $T_1, T_2, \ldots, T_k$ be the trees induced by a partition of $V(F)$ into connected subsets. Further let $m_1, m_2, \ldots, m_k \in \mathbb{N}_0$. Then:*

$$\sum_{i=1}^{k} \mathrm{OPT}(T_i, 3m_i) \leq \mathcal{O}(\mathrm{OPT}(F, m) + n).$$

**Proof.** We have

$$\begin{aligned}
\sum_{i=1}^{k} \mathrm{OPT}(T_i, 3m_i) &\leq \sum_{i=1}^{k} \mathrm{OPT}_0(T_i, 15m_i) + \mathcal{O}(n) && \text{(Lemma 5.5)} \\
&\leq \mathrm{OPT}_0(F, 15m) + \mathcal{O}(n) && \text{(Lemma 5.4)} \\
&\leq \mathrm{OPT}(F, 15m) + \mathcal{O}(n) && \text{(obvious)} \\
&\leq \mathcal{O}(\mathrm{OPT}(F, m) + n). && \text{(Lemma 5.6)}
\end{aligned}$$

◀

Observe that, unsurprisingly, the constant 3 in Lemma 5.3 can be replaced by an arbitrary constant. We believe that the function OPT is properly super-additive. That is, we make the following conjecture:

▶ **Conjecture 5.7.** *Let $F$ be a forest, $U_1, U_2$ be a partition of $V(F)$ into convex subsets, and let $m_1, m_2 \in \mathbb{N}_0$. Then $\mathrm{OPT}(F[U_1], m_1) + \mathrm{OPT}(F[U_2], m_2) \leq \mathrm{OPT}(F, m_1 + m_2)$.*

## 6 Hardness of weighted subtree-sum

In this section we prove Theorem 1.5, restated here for convenience.

▶ **Theorem 1.5.** *Each data structure that maintains a weighted forest with $n$ vertices under the operations* `subtree-sum` *and* `cut` *requires $\Omega(n \log n)$ time for $\Theta(n)$ operations in the Word RAM model of computation and assuming $G = \mathbb{Z}$.*

We remark that the lower bound holds even in the more general *cell-probe* model of computation, where the time complexity of an operation is only measured in the number of memory accesses (probes) it makes, regardless of the computation performed between probes. Moreover, we allow for Las Vegas randomization. Auxiliary vertices are not allowed in this section.

In our hardness proof, we invoke the Pătraşcu and Demaine's lower bound for the `partial-sum` problem [28]. Recall that in this problem, we maintain an array $A$ of length $n$ with values from 0 to $U - 1$ for some $U \geq n^{\Omega(1)}$, supporting operations of the form `update`$(i, x)$ (set $A[i] \leftarrow x$) and `partial-sum`$(p)$ (return $\sum_{i=1}^{p} A[i]$). We will say that a sequence of operations performed on $A$ is *epoch-based* if all operations can be partitioned into some $E$ epochs, such that within each epoch, each element of $A$ is `updated` exactly once and the sum of each prefix is queried exactly once. We will use the following version of the lower bound:

▶ **Theorem 6.1.** *Consider any cell-probe data structure for the* `partial-sum` *problem on an array $A$ of length $n$ and values from 0 to $U \geq n^{\Omega(1)}$ that may use Las Vegas randomization. Let $E \leq n^{\mathcal{O}(1)}$. The data structure requires $\Omega(En \log n)$ time to process $\Theta(En)$ operations, even if we assume that the sequence of operations is epoch-based and comprises $E$ epochs.*

We remark that Theorem 6.1 is not explicitly stated in [28], but it can be derived from their techniques; we provide a proof in the full version of the paper.

Towards Theorem 1.5, assume we have a data structure $X$ supporting `subtree-sum` and `cut`. Using $X$, we will construct a data structure $Y$ for `partial-sum` from Theorem 6.1, as follows. Suppose $Y$ is initialized with an array $A$ of length $n'$ containing integers from 0 to $n'-1$. Let $n = n'(9n'+2)$. We construct a weighted forest $F$ comprising two trees: $T_+$ on $n_+ = n'(8n'+1)$ vertices and $T_-$ on $n_- = n'(n'+1)$ vertices. $T_+$ contains a path $v_1^+ v_2^+ \dots v_{n'}^+$ of $n'$ vertices, where each $v_i^+$ has weight 0. For each $i \in [n']$, we create $8n'$ additional vertices, denoted $u_{i,1}, u_{i,2}, \dots, u_{i,8n'}$, where $u_{i,j}$ has weight $j$. We connect each $u_{i,j}$ and $v_i^+$ with an edge. $T_-$ is constructed similarly, with a path $v_1^- v_2^- \dots v_{n'}^-$ of $n'$ vertices of weight 0, and for each $i \in [n']$, $n'$ additional vertices $w_{i,1}, w_{i,2}, \dots, w_{i,n'}$ of weight 1 connected to $v_i^-$. We root $T_+$ at $v_{n'}^+$ and $T_-$ at $v_{n'}^-$, so that $\texttt{subtree-sum}(v_i^+)$ (respectively, $\texttt{subtree-sum}(v_i^-)$) returns the sum of weights of all vertices $u_{i',j}$ (resp., $w_{i',j}$) with $i' \le i$. We remark that $F$ contains no auxiliary vertices. The data structure $X$ is initialized with $F$ and the given weights.

We will now show how to use $X$ to process an epoch-based sequence of operations on $A$ comprising $E = n'$ epochs, with a total of $\Theta((n')^2) = \Theta(n)$ operations. Suppose an `update` is performed on $A$ during the $p$th epoch, replacing $A[i]$ with a new value from range $[0, n'-1]$ and thus increasing $A[i]$ by some $\Delta \in [-n'+1, n'-1]$. Then $X$ performs three `cut`s: it cuts $u_{i,j_1}$ and $u_{i,j_2}$ from $T_+$, where $j_1$ and $j_2$ are chosen such that $j_1 + j_2 = 7n' - \Delta$ and both $u_{i,j_1}$ and $u_{i,j_2}$ are still connected to $v_i^+$, and it cuts $w_{i,p}$ from $T_-$. Later we will show that such $j_1$ and $j_2$ always exist and can be found by a randomized algorithm in expected constant time. Observe that after these cuts, the value of the expression

$$\texttt{subtree-sum}(v_k^+) - 7n' \cdot \texttt{subtree-sum}(v_k^-)$$

remains unchanged for $k < i$, and increases by $\Delta$ for $k \ge i$. Thus, given suitable preprocessing, we can compute any partial sum of $A$ in constant time using two `subtree-sum` queries – one in each of $T_+$ and $T_-$. The details follow.

At the time of the initialization, we construct a table $B$ of size $n'$, so that

$$\sum_{i=1}^{k} A[i] = \texttt{subtree-sum}(v_k^+) - 7n' \cdot \texttt{subtree-sum}(v_k^-) + B[k] \quad \text{for all } k \in [n']. \tag{1}$$

Note that initially, $\texttt{subtree-sum}(v_k^+) = 4kn'(8n'+1)$ and $\texttt{subtree-sum}(v_k^-) = kn'$ for all $k \in [n']$, so $B$ can be computed in $\mathcal{O}(n')$ time. The invariant (1) will be maintained throughout the execution of the data structure. We will also preserve the following invariant: after $p$ `update`s to the value of $A[i]$, exactly $2p$ vertices $u_{i,j}$ have been cut from $T_+$, and exactly vertices $w_{i,p'}$ for $p' \le p$ have been cut from $T_-$.

Consider now an operation during the $p$th epoch. A `partial-sum` query for some $k \in [n']$ can be answered by computing the right-hand side of (1), which requires two `subtree-sum` queries and a constant amount of additional work. Now consider an `update` changing $A[i]$ by $\Delta \in [-n'+1, n'-1]$, and define $s = 7n' - \Delta \in [6n'+1, 8n'-1]$.

▶ **Lemma 6.2.** *There exist $j_1, j_2 \in [8n']$ such that $j_1 + j_2 = s$ and both $u_{i,j_1}$ and $u_{i,j_2}$ are still connected to $v_i^+$. Furthermore, such $j_1$ and $j_2$ can be found by a Las Vegas randomized algorithm in expected constant time.*

**Proof.** Consider the $3n'$ pairs of integers $(j, s-j)$ for $j \in [3n']$. Since $s \in [6n'+1, 8n'-1]$, all $6n'$ integers in these pairs are distinct and belong to $[8n']$. Since $p < n'$, by the invariant at most $2(p-1) < 2n'$ of these integers correspond to vertices $u_{i,j}$ that have already been

cut from $T_+$. Therefore, at least $n'$ pairs $(j, s-j)$ remain such that both $u_{i,j}$ and $u_{i,s-j}$ are still connected to $v_i^+$. Thus, the sought pair $(j_1, j_2)$ exists, and it can be found by randomly sampling pairs as above until a valid one is found, which takes expected constant time. ◀

After finding such $j_1$ and $j_2$, $X$ performs $\texttt{cut}(u_{i,j_1})$, $\texttt{cut}(u_{i,j_2})$, and $\texttt{cut}(w_{i,p})$. This preserves all the invariants, as argued above, completing the description of the data structure $Y$. However, by Theorem 6.1, $Y$ requires $\Omega(En' \log n') = \Omega(n \log n)$ time to process $E = n'$ epochs of operations. $Y$ manages to perform the entire sequence of operations via $\Theta((n')^2) = \Theta(n)$ operations on $X$ and a total of $\mathcal{O}((n')^2) = \mathcal{O}(n)$ additional work (for maintaining $B$ and finding $j_1, j_2$). Thus, $X$ must require $\Omega(n \log n)$ time to process the $\Theta(n)$ operations, proving Theorem 1.5.

We also observe that this proof can be rewritten to also apply in the group model setting:

▶ **Corollary 6.3.** *Each data structure that maintains a weighted forest under $n$ vertices under the operations* `subtree-sum` *and* `cut` *requires $\Omega(n \log n)$ time for $\Theta(n)$ operations in the group model of computation.*

**Proof.** Let $B = o(n)$ be such that the weights of the vertices in the forest $F$ above are strictly smaller than $B$, and let $D$ be the considered data structure in the group model of computation. Since $D$ works correctly in any commutative group, it can report the correct subtree sums in the group $\mathbb{Z}_{nB}$ of integers modulo $nB$; these are equal in value to the subtree sums in $\mathbb{Z}$, as all subtree sums are strictly smaller than $nB$. Meanwhile, all operations in $\mathbb{Z}_{nB}$ can be simulated by a Word RAM machine in constant time; therefore there exists a data structure $D'$ in the Word RAM model of computation with the same complexity guarantees that maintains $F$ under `subtree-sum` and `cut`. Theorem 1.5 applied to $D'$ finishes the proof. ◀

# 7 Subtree sizes

In this section, we consider the `subtree-sum` problem on *unweighted* and *0-1-weighted* forests, where `update-weight` is not available and $\texttt{subtree-sum}(v)$ returns the number of vertices (resp., the number of vertices with weight 1) in the subtree rooted at $v$. Our algorithms (Section 7.2) support auxiliary vertices. We determine the complexity of the problem to be $\Theta(n \frac{\log n}{\log \log n})$ for $\Theta(n)$ operations.

## 7.1 Lower bound

We prove the lower bound (Theorem 1.7) below:

▶ **Theorem 1.7.** *Each data structure in the Word RAM model that maintains an unweighted forest with $n$ vertices under the operations* `subtree-sum` *and* `cut` *requires $\Omega(n \frac{\log n}{\log \log n})$ time for $n$ operations.*

Similarly to Theorem 1.5, we reduce from the `partial-sum` problem; however, we find it more convenient to invoke a result from an earlier work by Fredman and Saks [14] that concerns the `partial-sum-parity` problem. In `partial-sum-parity`, we are given an array $A$ of length $n$ with values from $\{0, 1\}$, and we should support two operations: `update` (flipping the value of $A[i]$) and `partial-sum-parity` (returning the parity of the sum of the first $i$ values of $A$).

▶ **Theorem 7.1** ([14, Theorem 3]). *Consider any cell-probe data structure for the problem of* `partial-sum-parity` *on an array $A$ of length $n$ that may use amortization and Las Vegas randomization. The data structure requires $\Omega(n\frac{\log n}{\log\log n})$ time to process $\Theta(n)$ operations, even if each element of the array is* `update`*d at most once.*

**Proof of Theorem 1.7.** Let $X$ be a data structure for `subtree-sum` on unweighted forests. Using $X$, we will construct a data structure $Y$ for `partial-sum-parity` on an array $A$ of length $n'$. On initialization of $Y$ with an array $A$, we construct a tree $T$ of size $n \leq 3n'$ as follows. Let $v_1 v_2 \dots v_{n'}$ be a path of $n'$ vertices. For each $i \in [n']$, if $A[i] = 0$, we create one additional vertex $u_{i,1}$ connected to $v_i$; if $A[i] = 1$, we create two additional vertices $u_{i,1}$ and $u_{i,2}$ connected to $v_i$. We root $T$ at $v_{n'}$. This construction ensures the following invariant: For each $i \in [n']$, the parities of `subtree-sum`$(v_i)$ and $\sum_{j=1}^{i} A[j]$ are the same. The data structure $X$ is initialized with $T$. Then, to process an `update` of $A$ flipping the parity of $A[i]$, we simply invoke `cut`$(u_{i,1})$; this is legal (since each element of $A$ is `update`d at most once) and preserves the invariant. Therefore, a `partial-sum-parity` query for some $k \in [n']$ can be answered by simply invoking `subtree-sum`$(v_k)$ and returning its parity.

By Theorem 7.1, $Y$ requires $\Omega(n'\frac{\log n'}{\log\log n'})$ time to process $\Theta(n')$ operations. On the other hand, our implementation of $Y$ performs $\Theta(n')$ operations on $X$ and a total of $\mathcal{O}(n')$ additional work. Since $n' = \Theta(n)$, this finishes the proof. ◀

## 7.2 Upper bound

We start with a data structure for forests that are very small relative to the size of a machine word, which we denote by $b$. This is somewhat similar to Lemma 4.1. However, here we actually need to maintain small *weighted* forests, which means that we need a different approach. The weights are assumed to fit in one word each, i.e., to be an integer in $[0, 2^b - 1]$. We only allow a very limited way of changing weights: the operation `decrement-weight`$(v)$, which decreases the weight $w(v)$ of $v$ by one (only allowed if $w(v) \geq 1$). Note that this data structure is merely used as a subroutine; in the end, we give a data structure for *unweighted* forests of size $n = \Theta(2^b)$. The proof can be thought of as an adaptation of Dietz's data structure for *Subset Rank* [11] – essentially, partial sums of 0-1-arrays – to forests.

▶ **Lemma 7.2.** *Fix a word size $b$, and let $k \leq \frac{1}{4}\sqrt{b}$ be a power of two. There is a data structure maintaining a weighted forest with $k$ vertices under the operations* `subtree-sum`*,* `cut`*, and* `decrement-weight`*, provided that all weights are non-negative integers, and the sum $W$ of all weights in the initial forest is at most $2^b - 1$.*

*The data structure requires a certain global table only depending on $k$, which can be computed in time $\mathcal{O}((3k)^k)$. After precomputing that table, initialization takes $\mathcal{O}(k)$ time, each* `subtree-sum` *operation takes constant time, each* `cut` *operation takes $\mathcal{O}(k)$ time, and all* `decrement-weight` *operations together take $\mathcal{O}(W)$ time.*

**Proof.** Let $(F, w)$ denote the current weighted forest, and say $V(F) = [k]$.

We follow Dietz [11] in using two arrays $A$ and $B$ of length $k$. $A$ stores the actual subtree-sum for each vertex in $F$, but is only updated every $k$ operations (such an update is called a *flush*). $B$ stores for each vertex the number of decrements since the last flush. Since this number can be no more than $k$, the entire array $B$ fits in $k(1 + \log k) \leq b$ bits, so it fits into a single machine word.

To adapt the Dietz's idea to trees with `cut`s, we need the following two extra data structures. First, we want to store, for each vertex $v$, the set of vertices $V(F_v)$ in the subtree rooted at $v$ (i.e., the ones whose weights count towards `subtree-sum`$(v)$). We store the mapping $v \to V(F_v)$ in an array $C$. Each $V(F_v)$ is represented as a bitstring, so $C$ has size $k^2 \leq b$ and fits into a single machine word.

Second, to actually obtain the sum $w(F_v)$, we need the *global table* mentioned in the statement of the lemma. We denote it by $Q$ and it maps a pair $(B', U)$ to an integer. Here $B'$ is an array of length $k$, storing $k$ numbers in $[0, k]$ (as $B$ above), and $U$ is a subset of $V(F) = [k]$, stored as a bitstring. Both $B'$ and $U$ fit into single machine words. The value $Q[B', U]$ is $\sum_{i \in U} B'[i]$. Note that $Q$ has $(k+1)^k \cdot 2^k$ entries, each entry fits into a machine word, and $Q$ can be computed in time $|Q| \cdot \mathcal{O}(k) = \mathcal{O}((3k)^k)$. Observe that $Q$ only depends on $k$, as required.

We now show how to implement the operations. We maintain the invariant that $A$ is correct for a previous point in time (the last *flush*), and since then at most $k-1$ `decrement-weight` and no `cut` operations have been performed. In turn, $B[v]$ is the number of `decrement-weight`$(v)$ operations since the last flush.

- Initially, we compute $A$ and $C$ in a bottom-up fashion, in $\mathcal{O}(k)$ time. We initialize $B$ as an all-zero vector, and assume $Q$ has been already precomputed.
- To answer a `subtree-sum`$(v)$ query, we calculate $A[v] - Q[B, C[v]]$, in constant time.
- To perform `decrement-weight`$(v)$, we first increment $B[v]$. Then, if $k$ `decrement-weight` operations have been performed since the last flush, we perform a flush as follows. For each $v \in V(F)$, set $A[v] \leftarrow A[v] - Q[B, C[v]]$, and then reset $B$ to the all-zero array. The amortized time for the flushes is clearly constant per `decrement-weight` operation. Since we can perform no more than $W$ `decrement-weight` operations, the total time is at most $\mathcal{O}(W)$.
- To perform `cut`$(v)$, we first perform a flush, then recover all vertex weights from $A$, then delete the edge from $v$ to its parent and recompute $A$ and $C$. This requires $\mathcal{O}(k)$ time, as desired. ◀

We now show how to iterate Lemma 7.2, again using a cluster decomposition (though with different cluster size than before).

We introduce a new operation that simplifies the analysis. When maintaining a weighted forest which is initially a tree, the parameterless operation `root-sum`() returns the sum of weights of vertices still connected to the root of the *initial* tree. Note that `root-sum` can and will be implemented with a single call `subtree-sum`$(v)$, where $v$ is the initial tree root (which can be stored at the beginning).

We present a sequence of data structures $X_1, X_2, \ldots$, which support increasingly larger inputs, but also have successively worse running time. For now, we assume the input is a binary tree. For technical reasons related to binarization (Theorem 2.2), we consider 0-1-weighted forests instead of unweighted forest. Below, big-$\mathcal{O}$ notation only hides constants not depending on the values $n$, $b$, or $t$.

▶ **Lemma 7.3.** *Fix a word size $b$ and let $\ell$ be a power of two with $\ell \leq \frac{1}{48}\sqrt{b}$. For each parameter $t \in \mathbb{N}_+$, there is a data structure maintaining an 0-1-weighted binary forest, initially a tree, with $n \leq \min\{\ell^t, 2^b - 1\}$ vertices under the operations* `cut` *and* `subtree-sum`*.*

*The data structure requires a certain global table only depending on $\ell$, which can be computed in time $\mathcal{O}((36\ell)^{12\ell})$. After precomputing that table, the initialization takes $\mathcal{O}(tn)$ time, each* `subtree-sum` *operation takes $\mathcal{O}(t)$ time, each* `root-sum` *operation takes $\mathcal{O}(1)$ time, and all* `cut` *operations together take $\mathcal{O}(t(n + \ell^2))$ time.*

As described in the introduction, the proof again uses cluster decompositions, but with a different cluster size. The proof is found in the full version of the paper.

With Lemma 7.3, we are ready to prove:

▶ **Theorem 1.6.** *There is a data structure in the Word RAM model maintaining a 0-1-weighted forest with $n$ vertices under the operations* `subtree-sum` *and* `cut`*, with $\mathcal{O}((m+n)\frac{\log n}{\log\log n})$ total time for $m$ operations.*

**Proof.** We assume the machine word size is at least $b = 1+\lceil\log(n+1)\rceil$.[6] Let $b' = \lceil\log n\rceil \leq b$. As usual, we can assume the initial weighted forest is a tree $(T^0, w_0)$. First, binarize the tree with Theorem 2.2 to obtain $(T^1, w_1)$. Now use Lemma 7.3 with parameters $\ell = \lfloor\frac{1}{48}\sqrt{b'}\rfloor$ and $t = \lceil\log_\ell(2n)\rceil$. Note that $|V(T^1)| \leq 2n < 2^b - 1$ and $|V(T^1)| \leq \ell^t$, so the conditions for Lemma 7.3 are satisfied. The total running time is $\mathcal{O}(t(m+n+\ell^2))$. Observe that $\ell^2 \leq b' \leq n$. Moreover, we have $t \leq \mathcal{O}(\frac{\log n}{\log\log n})$ by an easy calculation. The statement follows. ◀

## References


1. Peyman Afshani, Jérémy Barbay, and Timothy M. Chan. Instance-optimal geometric algorithms. *J. ACM*, 64(1):3:1–3:38, 2017. doi:10.1145/3046673.
2. Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. doi:10.1145/1103963.1103966.
3. Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS 1998, Palo Alto, California, USA, November 8-11, 1998*, pages 534–544. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743504.
4. Stephen Alstrup, Jens P. Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Inf. Process. Lett.*, 64(4):161–164, 1997. doi:10.1016/S0020-0190(97)00170-1.
5. V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. On economical construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*, 194(3):1209–1210, 1970. Russian.
6. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
7. Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/J.TCS.2003.05.002.
8. Jon Louis Bentley and Hermann A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980. doi:10.1007/BF00263991.
9. Benjamin Aram Berendsohn. Universally optimal decremental tree minima, 2026. doi:10.48550/arXiv.2602.15977.
10. Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994. doi:10.1016/S0022-0000(05)80002-9.
11. Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 39–46. Springer, 1989. doi:10.1007/3-540-51542-9_5.
12. Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981. doi:10.1145/322234.322235.
13. Peter M. Fenwick. A new data structure for cumulative frequency tables. *Softw. Pract. Exp.*, 24(3):327–336, 1994. doi:10.1002/SPE.4380240306.


[6] The assumption that $b \geq \Omega(\log n)$ is standard, and if it is a constant factor less than necessary, we use the usual simulation of $b$-bit-word operations with a constant number of words.

14 M. Fredman and M. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, pages 345–354, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/73007.73040.

15 Michael L. Fredman. The complexity of maintaining an array and computing its partial sums. *J. ACM*, 29(1):250–260, 1982. doi:10.1145/322290.322305.

16 Michael L. Fredman and Dan E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 1–7. ACM, 1990. doi:10.1145/100216.100217.

17 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of Disjoint Set Union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

18 Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964. doi:10.1145/364099.364331.

19 Bernhard Haeupler, Richard Hladík, John Iacono, Václav Rozhon, Robert E. Tarjan, and Jakub Tetek. Fast and simple sorting using partial information. In Yossi Azar and Debmalya Panigrahi, editors, *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025, New Orleans, LA, USA, January 12-15, 2025*, pages 3953–3973. SIAM, 2025. doi:10.1137/1.9781611978322.134.

20 Bernhard Haeupler, Richard Hladík, Václav Rozhon, Robert E. Tarjan, and Jakub Tetek. Universal optimality of Dijkstra via beyond-worst-case heaps. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 2099–2130. IEEE, 2024. doi:10.1109/FOCS61266.2024.00125.

21 Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1166–1179. ACM, 2021. doi:10.1145/3406325.3451081.

22 John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM J. Comput.*, 2(4):294–303, 1973. doi:10.1137/0202024.

23 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. *TheoretiCS*, 2, 2023. doi:10.46298/THEORETICS.23.6.

24 Neil D. Jones. *Computability and complexity - from a programming perspective*. Foundations of computing series. MIT Press, 1997.

25 Jakub Łącki and Piotr Sankowski. Optimal decremental connectivity in planar graphs. *Theory Comput. Syst.*, 61(4):1037–1053, 2017. doi:10.1007/S00224-016-9709-X.

26 Lawrence L. Larmore. An optimal algorithm with unknown time complexity for convex matrix searching. *Inf. Process. Lett.*, 36(3):147–151, 1990. doi:10.1016/0020-0190(90)90084-B.

27 Jirí Matousek. Range searching with efficient hierarchical cuttings. In David Avis, editor, *Proceedings of the Eighth Annual Symposium on Computational Geometry, Berlin, Germany, June 10-12, 1992*, pages 276–285. ACM, 1992. doi:10.1145/142675.142732.

28 Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006. doi:10.1137/S0097539705447256.

29 Seth Pettie and Vijaya Ramachandran. An Optimal Minimum Spanning Tree Algorithm. *J. ACM*, 49(1):16–34, January 2002. doi:10.1145/505241.505243.

30 Seth Pettie and Vijaya Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, 34(6):1398–1431, 2005. doi:10.1137/S0097539702419650.

31 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

32 Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

**33** Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18(2):110–127, 1979. doi:10.1016/0022-0000(79)90042-4.

**34** Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. A combinatorial proof of universal optimality for computing a planar convex hull. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025*, volume 351 of *LIPIcs*, pages 102:1–102:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ESA.2025.102.

**35** Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. Simpler optimal sorting from a directed acyclic graph. In Ioana Oriana Bercea and Rasmus Pagh, editors, *2025 Symposium on Simplicity in Algorithms, SOSA 2025, New Orleans, LA, USA, January 13-15, 2025*, pages 350–355. SIAM, 2025. doi:10.1137/1.9781611978315.26.

**36** Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. Simpler universally optimal Dijkstra. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025*, volume 351 of *LIPIcs*, pages 71:1–71:9. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ESA.2025.71.

**37** Andrew Chi-Chih Yao. On the complexity of maintaining partial sums. *SIAM J. Comput.*, 14(2):277–288, 1985. doi:10.1137/0214022.