Nearly Optimal List Labeling

Michael A. Bender
Stony Brook University and RelationalAI
Stony Brook, NY, USA

Alex Conway
Cornell Tech
New York, NY, USA

Martín Farach-Colton
New York University
New York, NY, USA

Hanna Komlós
New York University
New York, NY, USA

Michal Koucký
Charles University
Prague, Czech Republic

William Kuszmaul
Carnegie Mellon University
Pittsburg, PA, USA

Michael Saks
Rutgers University
Piscataway, NJ, USA

***Abstract*—The list-labeling problem captures the basic task of storing a dynamically changing set of up to n elements in sorted order in an array of size $m = (1+\Theta(1))n$. The goal is to support insertions and deletions while moving around elements within the array as little as possible.**

Until recently, the best known upper bound stood at $O(\log^2 n)$ amortized cost. This bound, which was first established in 1981, was finally improved two years ago, when a randomized $O(\log^{3/2} n)$ expected-cost algorithm was discovered. The best randomized lower bound for this problem remains $\Omega(\log n)$, and closing this gap is considered to be a major open problem in data structures.

In this paper, we present the See-Saw Algorithm, a randomized list-labeling solution that achieves a nearly optimal bound of $O(\log n \operatorname{polyloglog} n)$ amortized expected cost. This bound is achieved despite at least three lower bounds showing that this type of result is impossible for large classes of solutions.

***Index Terms*—Data structures, probabilistic algorithms, combinatorial algorithms**

I. Introduction

In this paper, we revisit one of the most basic problems in data structures: maintaining a sorted array, as elements are inserted and deleted over time [39]. Suppose we are given an array of size $m = (1 + \Theta(1))n$, and a sequence of insertions and deletions, where up to n elements can be present at a time. As the set of elements changes over time, we must keep the elements in sorted order within the array. Sometimes, to support an insertion, we may need to move around elements that are already in the array. The ***cost*** of an insertion or deletion is the number of elements that we move, and the goal is to achieve as small a cost as possible.[1]

Since it was introduced in 1981 [39], this problem has been rediscovered in many different contexts [3], [36], [56], [63], and has gone by many different names (e.g., a sparse-array priority queue [39], the file-maintenance problem [16], [63]–[66], the dynamic sorting problem [43], etc). In recent decades, it has become most popularly known as the ***list-labeling problem*** [10], [23], [27], [30], [57].

In the decades since it was introduced, the list-labeling problem has amassed a large literature on algorithms [7], [9], [10], [12], [16], [17], [19], [36], [38]–[40], [64]–[66], lower bounds [10], [23], [24], [27], [28], [30], [57], [67], applications to both theory and practice [8], [9], [12]–[14], [16], [17], [22], [26], [44], [46], [55], [60]–[66], other parameter regimes for m and n [4], [6], [20], [24], [67], and open problems [35], [57]. We focus here on some of the major milestones and defer a more in-depth discussion of related work to Section VIII.

Past upper and lower bounds. The list-labeling problem was introduced in 1981 by Itai, Kohheim, and Rodeh [39], who gave a simple deterministic solution with amortized cost $O(\log^2 n)$. Despite a great deal of interest [9], [12], [16], [17], [19], [36], [38]–[40], [64]–[66], this bound would remain the state of the art for four decades.

Starting in the early 1990s, much of the theoretical progress was on lower bounds. The first breakthrough came from Dietz and Zhang [27], [30], who showed $\Theta(\log^2 n)$ to be optimal for any ***smooth*** algorithm, that is, any algorithm that spreads elements out evenly whenever it rebuilds some subarray. Later work by Bulánek, Koucký, and Saks [23] established an even more compelling claim—that the $\Theta(\log^2 n)$ bound is optimal for any deterministic algorithm. At this point it seemed likely that $\Theta(\log^2 n)$ should be optimal across all algorithms, including randomized ones, but the best lower bound known for randomized solutions, also due to Bulánek, Koucký, and Saks [24], remained $\Omega(\log n)$.

Recent work by Bender et al. [10] showed that there is, in fact, a surprising separation between deterministic and randomized solutions. They construct a list-labeling algorithm with $O(\log^{3/2} n)$ expected cost per operation. Their algorithm satisfies a notion of *history independence*, in which the set of array-slot positions occupied at any given moment reveals nothing about the input sequence except for the current number of elements. This history independence property ends up being crucial to the algorithm design and analysis [10],[2] but the property also comes with a limitation: Bender et al. show that any algorithm satisfying this type of history independence *must* incur amortized expected cost $\Omega(\log^{3/2} n)$ [10].

[1]One might prefer to simply analyze time complexity rather than cost. It turns out that, for the algorithms in this paper, these two metrics will be asymptotically equivalent.

[2]Roughly speaking, the authors use history independence as a mechanism to avoid the possibility of a clever input sequence somehow "degrading" the state of the data structure over time.

It remained an open question whether there might exist a list-labeling algorithm that achieves $o(\log^{3/2} n)$ cost, or even $O(\log n)$ cost. Such an algorithm would necessarily need to be non-smooth, randomized, *and* history dependent—and it would need to employ these properties in algorithmically novel ways.

This paper: nearly optimal list labeling. In this paper, we present a list-labeling algorithm that achieves amortized expected cost $\widetilde{O}(\log n)$ per operation. This matches the known $\Omega(\log n)$ lower bound [24] up to a $\operatorname{poly}\log\log n$ factor. We refer to our list-labeling algorithm as the *See-Saw Algorithm*.

Perhaps the most surprising aspect of the See-Saw Algorithm is how it employs history dependence. The algorithm breaks the array into a recursive tree, and attempts (with the help of randomization) to predict which parts of the tree it thinks more insertions will go to. It then gives more slots to the subproblems that it thinks are more likely to get more insertions.

The idea that such predictions could be helpful would be very natural if we were to assume that our input were either *stochastic* [18], [19] or came with some sort of *prediction oracle* [47]. What is remarkable about the See-Saw Algorithm is that the predictions it makes, and the ways in which it uses them, end up leading to near-optimal behavior on *all possible input sequences*. In fact, to the best of our knowledge, the See-Saw Algorithm is the first example of a dynamic data structure using *adaptivity* to improve the best worst-case (amortized expected) bound on cost for a problem.[3]

Of course, randomization is also important. If the See-Saw Algorithm were deterministic, then the input sequence could easily trick it into making bad decisions. Thus, it is not just the fact that the algorithm makes predictions based on the past, but also the way in which those predictions interact with the randomness of the algorithm that together make the result possible.[4]

Our result puts the complexity of maintaining a sorted array almost on par with the complexities of other classical sorting problems [2], [21], [48]. Whether or not the See-Saw Algorithm has a practical real-world counterpart remains to be seen. Such a result could have extensive applications [44], [46], [55], [60]–[62] to systems that use list labeling as a locality-friendly alternative to binary search trees.

A remark on other parameter regimes. In addition to the setting where $m = (1 + \Theta(1))n$, list labeling has also been studied in other parameter regimes, both where $m = (1+\delta)n$ for some $\delta = o(1)$ (our results naturally extend to this regime with cost $\widetilde{O}(\delta^{-1}\log n)$), and where $m \gg n$. An interesting feature of the $m \gg n$ regime is that, when $m = n^{1+\Theta(1)}$, the optimal cost becomes $\Theta(\log n)$, even for randomized solutions [24]. Thus, a surprising interpretation of our result is that there is *almost no complexity gap* between the setting where $m = (1+\Theta(1))n$ and the setting where $m = \operatorname{poly}(n)$. In both cases, the optimal amortized expected complexity is $\widetilde{\Theta}(\log n)$.

Implications to other algorithmic problems. We remark that there are several algorithmic problems whose best known solutions rely directly on list labeling, and for which list-labeling improvements immediately imply stronger results.

One significant application is to cache-oblivious B-trees [12]–[14], [17], [22], where our list-labeling algorithm can be used to reduce the best known I/O complexity from $O(\log_B N + (\log^{3/2} N)/B)$ [10] to $O(\log_B N) + \widetilde{O}((\log N)/B)$, which, in turn, reduces to the optimal bound of $O(\log_B N)$ so long as $B = (\log\log n)^c$ for some sufficiently large $c = \Theta(1)$.[5]

Another application is to the variation of list labeling in which $n = m$ elements are inserted [4], [10], [20], [67] without deletion, that is, an array is filled all the way from empty to full. Here, our results imply an overall amortized expected bound of $\widetilde{O}(\log^2 n)$ cost per insertion (see Corollary 4), improving over the previous state-of-the-art of $\widetilde{O}(\log^{5/2} n)$ [10].

Paper outline. The rest of the paper proceeds as follows. We begin in Sections II and III with preliminaries and statements of our main results. We present the See-Saw Algorithm in Section IV. We then present the analysis of the algorithm, modulo two central technical claims, in Section V—these technical claims are proven in Sections VI and VII. Section VIII gives a detailed review of related work and Section IX provides concluding remarks and discusses open problems. Finally, Appendix A gives reductions for our main theorem, and Appendix B gives pseudocode for the See-Saw Algorithm.

II. Preliminaries

Defining the list-labeling problem. In the list-labeling problem, there are two parameters, the array size m, and the maximum number of elements n. We will be most interested in the setting where $m = (1+\Theta(1))n$, but to be fully general, we will also allow $m = (1+\delta)n$ for $\delta = o(1)$.

We are given an (online) sequence of insertions and deletions, where at any given moment there are up to n elements present. The elements are assumed to have a total order, and our job is to keep the current set of elements in sorted order within the size-m array. As insertions and deletions occur, we may choose (or need) to move elements around within the array. The ***cost*** of an insertion or deletion is defined to be the number of elements that get moved.

[3]Indeed, one can formalize this claim—it has remained an open question whether there exists *any* data structural problem for which history dependence is necessary to achieve optimal edit-cost bounds [50]. Our paper does not quite resolve this problem for the following technical reason: the *lower bound* for history-independent list labeling [10] applies only to a weaker notion of history independence than the one in [50].

[4]Interestingly, despite the importance of randomness in our algorithm, the actual *amount* of randomness is relatively small. In fact, one can straightforwardly implement the algorithm using $O((\log\log n)\log n)$ random bits, where $O(\log\log n)$ random bits are used to generate the randomness used within each level of the recursion tree.

[5]Cache-oblivious B-trees make use of so-called packed-memory arrays [11], [12], [17], which are list-labeling solutions with the additional property that the array never contains more than $O(1)$ free slots between consecutive elements. As discussed in Section IV, our results can be extended to also offer this additional property.

When discussing randomized solutions, one assumes that the input sequence is generated by an oblivious adversary. In other words, the input sequence is independent of the random bits used by the list-labeling algorithm.

Conventions. To simplify discussion throughout the paper, we will generally ignore rounding issues. Quantities that are fractional but should be integral can be rounded to the closest integer without affecting the overall analysis by more than a negligible error.

We will always be interested in bounding *amortized expected cost*. A bound of C on this quantity means that, for all i, the expected total cost of the first i operations is $O(iC)$.

III. Main Results

Formally, the main result of this paper is that:

Theorem 1. *For $\delta \in (0,1)$, and $m = (1+\delta)n$, there is a solution to the list-labeling problem on an array of size m, and with up to n elements present at a time, that supports amortized expected cost $O(\delta^{-1}(\log n)(\log\log n)^3)$ per insertion and deletion.*

Corollary 2. *If $m = (1+\Theta(1))n$, then there is a solution to the list-labeling problem with amortized expected cost $O((\log n)(\log\log n)^3)$ per operation.*

In Appendix A, we present a series of standard w.l.o.g. reductions that together reduce the task of proving Theorem 1 to the task of proving the following equivalent but simpler-to-discuss result:

Theorem 3. *Let m be sufficiently large, and $n = m/2$. The See-Saw Algorithm is a list-labeling algorithm that, starting with $m/4$ elements, can support $m/4$ insertions with amortized expected cost $O((\log n)(\log\log n)^3)$.*

Note that, compared to Theorem 1, Theorem 3 is able to assume an insertion-only workload, a relationship of $m = 2n$, and a starting-state of $m/4 = n/2$ elements. The rest of the paper will be spent proving Theorem 3.

Finally, we remark that, in Appendix A, we also arrive at the following corollary:

Corollary 4. *There is a list-labeling algorithm that inserts n items into an initially empty array of size n with amortized expected cost $O((\log^2 n)(\log\log n)^3)$.*

IV. The See-Saw Algorithm

In this section, we present the ***See-Saw Algorithm***, which we subsequently prove achieves $\widetilde{O}(\log n)$ amortized expected cost per insertion on an array $\mathcal{A}$ of size m. We also present detailed pseudocode for the algorithm in Appendix B.

As discussed in Section III, we will consider, without loss of generality, that we have an insertion-only workload, that the initial number of elements is $m/4$, and that we are handling $m/4$ total insertions. The algorithm will make use of parameters $\alpha = C_\alpha(\log\log n)^2$ and $\beta = C_\beta(\log\log n)^2$, where C_α and C_β are positive constants selected so that C_α, C_β, and C_α/C_β are all sufficiently large.

Defining a subproblem tree. At any given moment, we will break the array into a recursive ***subproblem tree***. Each subproblem π in the tree is associated with a subarray $\mathcal{A}_\pi$ whose size is denoted by $m_\pi = |\mathcal{A}_\pi|$. For the subproblem π at the root of the tree, $\mathcal{A}_\pi$ is the entire array. Each non-leaf node π has left and right children, $L(\pi)$ and $R(\pi)$ respectively, such that $\mathcal{A}_\pi = \mathcal{A}_{L(\pi)} \oplus \mathcal{A}_{R(\pi)}$ (the concatenation of $\mathcal{A}_{L(\pi)}$ and $\mathcal{A}_{R(\pi)}$). In contrast with the classical $O(\log^2 n)$ algorithm (and, indeed, all previous algorithms that we are aware of), the structure of the subproblem tree used by the See-Saw Algorithm will be *non-uniform*, meaning that sibling subproblems $L(\pi)$ and $R(\pi)$ will not necessarily satisfy $m_{L(\pi)} = m_{R(\pi)}$.

As we shall see, the structure of the tree will evolve over time, with subproblems getting terminated and then replaced by new ones. When a subproblem π is first created, we will use $\mathcal{S}^0_\pi$ to refer to the set of elements stored in $\mathcal{A}_\pi$ when π is created.

Because subproblems are created and destroyed over time, the children $L(\pi)$ and $R(\pi)$ of a given subproblem π may get replaced many times during π's own lifetime. Thus one should think of $L(\pi)$ and $R(\pi)$ as time-dependent variables, referring to π's *current* left and right children at any given moment.

How an insertion decides its root-to-leaf path. Given an insertion x that goes to a subproblem π, the protocol for determining which child $L(\pi)$ or $R(\pi)$ the insertion x goes to can be described as follows: if $L(\pi)$ contains at least one element, and if $\max_{y\in L(\pi)} y > x$, then x is placed in $L(\pi)$; otherwise, it goes to $R(\pi)$. [6] This rule determines the root-to-leaf path that a given insertion takes.

Implementing leaves. When a subproblem is created, there are two conditions under which it is declared to be a leaf: subproblems π whose initial density $|\mathcal{S}^0_\pi|/m_\pi$ is greater than $3/4$ are ***expensive leaves***; and subproblems π whose subarray satisfies $m_\pi \le 2^{\sqrt{\log n}}$ are ***tiny leaves***.

In both cases, leaf subproblems π are implemented using the classical algorithm of Itai, Konheim and Rodeh [39], whose cost per operation is $O(\log^2 m_\pi)$. For tiny leaves, this results in $O(\log n)$ amortized cost per operation. For expensive leaves, this could result in as much as $O(\log^2 m) = O(\log^2 n)$ cost per operation. One of the major tasks in analyzing the algorithm will be to bound the total cost incurred in expensive leaves over all operations.

Initializing a subtree. When a subproblem π is first initialized, it is always initialized to be ***balanced***. This means: (1) that the elements in π are evenly distributed across $\mathcal{A}_\pi$; and (2) that, within each level of the subtree rooted at π, all of the subproblems within that level have arrays that are the same sizes as each other.

Thus we define the `CreateSubtree`$(\mathcal{A}', \mathcal{S}')$ procedure as follows. The procedure takes as input a subarray $\mathcal{A}'$ and a set $\mathcal{S}'$ of elements, and it produces a tree of balanced subproblems,

[6] Assuming we start with $m/4$ elements in the array, it will turn out that the left children are never empty—we will not need to formally prove this fact, however, as it does not end up being necessary for our analysis.

where the root of the tree π satisfies $\mathcal{A}_\pi = \mathcal{A}'$ and $\mathcal{S}^0_\pi = \mathcal{S}'$. This can be accomplished by first spreading the elements $\mathcal{S}'$ evenly across the array $\mathcal{A}$, and then creating a subproblem π satisfying $\mathcal{A}_\pi = \mathcal{A}$. If π is a leaf, then this is the entire procedure. Otherwise, if π is not a leaf, then we create children for π with sub-arrays of size $m_\pi/2$; and if those are not leaves, we create grandchildren for π with subarrays of size $m_\pi/4$, and so on. Pseudocode for `CreateSubtree` (and all other components of the See-Saw Algorithm) is presented in Appendix B.

Implementing non-leaf subproblems. Now consider a non-leaf subproblem π and let $\mathcal{I}_\pi$ denote the sequence of insertions that π receives.

The first thing that π does is select a ***rebuild window size*** w_π—this window size is selected from a carefully constructed probability distribution that we will describe later on. The subproblem π then treats the insertion sequence $\mathcal{I}_\pi$ as being broken into equal-sized ***rebuild windows*** $\mathcal{I}_{\pi,1}, \mathcal{I}_{\pi,2}, \ldots$, where each rebuild window $\mathcal{I}_{\pi,i}$ consists of up to w_π insertions. (Only the final rebuild window may be smaller.)

Whenever one rebuild window $\mathcal{I}_{\pi,i}$ ends and another $\mathcal{I}_{\pi,i+1}$ begins, π performs a ***rebuild***. The rebuild terminates all of π's descendant subproblems, spends $O(m_\pi)$ cost on rearranging the elements within $\mathcal{A}_\pi$, and then creates new descendant subproblems for π.

To describe this rebuild process, let us refer to π's children before the rebuild as $\overline{L}(\pi), \overline{R}(\pi)$ and to π's new children after the rebuild as $L(\pi), R(\pi)$. The most interesting step in the rebuild is to select the sizes $m_{L(\pi)}$ and $m_{R(\pi)}$ for $\mathcal{A}_{L(\pi)}$ and $\mathcal{A}_{R(\pi)}$—we will describe this step later in the section. After selecting the size $m_{L(\pi)}$, the new subproblem $L(\pi)$, along with its descendants, are created by calling `CreateSubtree`$(\mathcal{A}_{L(\pi)}, \mathcal{S}^0_{L(\pi)})$, where $\mathcal{S}^0_{L(\pi)}$ is the same set of elements that were stored in $\overline{L}(\pi)$ prior to the rebuild. Similarly, $R(\pi)$ and its descendants are created by calling `CreateSubtree`$(\mathcal{A}_{R(\pi)}, \mathcal{S}^0_{R(\pi)})$, where $\mathcal{S}^0_{R(\pi)}$ is the set of elements that was stored in $\overline{R}(\pi)$ prior to the rebuild.

It is worth emphasizing that the rebuild changes the *sizes* of the subarrays used to implement each of π's children, but does not change the *sets* of elements stored within the two children. Furthermore, although π's new child subtrees are initialized to be balanced, the subtree rooted at π need not be balanced: $m_{L(\pi)}$ need not be equal to $m_{R(\pi)}$, nor does $|\mathcal{S}^0_{L(\pi)}|$ need to equal $|\mathcal{S}^0_{R(\pi)}|$. The cost of such a rebuild is $O(m_\pi)$.

It remains to specify how to choose w_π (the rebuild-window size), and how to choose $m_{L(\pi)}$ and $m_{R(\pi)}$ during each rebuild. For this second point, rather than setting $m_{L(\pi)} = m_{R(\pi)} = m_\pi/2$, π will (sometimes) try to *predict* which of $L(\pi)$ or $R(\pi)$ will need more free slots in the future, and will potentially give a different number of slots to each of them. Successfully predicting which subproblem will get more insertions is key to our algorithm and is described more fully below.

A key component: selecting window sizes. We now describe how π selects its rebuild-window size w_π. This turns out to be the *only* place in the algorithm where randomization is used. We start by generating a random variable K_π, taking values in $[0, k_{\max}]$ where $k_{\max} = 2 \log\log n$ and where $p_k = \Pr[K_\pi = k]$ is given by:

$$\begin{aligned} p_k &= 2^{-(k+1)} \left(1 + \frac{k}{k_{\max}}\right) \qquad \text{for } k \in [1, k_{\max}] \\ p_0 &= 1 - \sum_{k=1}^{k_{\max}} p_k \leq 1/2. \end{aligned}$$

Having drawn K_π from this distribution, we then set

$$w_\pi = \frac{m_\pi}{\alpha 2^{K_\pi}},$$

where $\alpha = \Theta((\log\log n)^2)$ is the parameter defined at the beginning of the section. As we shall see, the specific details of this probability distribution will end up being central to the analysis of the algorithm.

A key component: selecting array skews. Next we describe how π selects the sizes of the subarrays $m_{L(\pi)}$ and $m_{R(\pi)}$ to be used by its children $L(\pi)$ and $R(\pi)$ within a given rebuild window. Here, critically, π *adapts* to the history of how insertions in past rebuild windows behaved.

The rebuilds of π will behave differently at the beginning of odd-numbered windows than even-numbered ones. At the beginning of odd-numbered rebuild windows, π will not make any attempt to adapt; it will simply set $m_{L(\pi)} = m_{R(\pi)} = m_\pi/2$. (This, incidentally, is why π need not do any rebuilding at the beginning of the first rebuild window.) To adapt at the beginning of an even-numbered rebuild window j, π will count the total number of insertions that went right minus the total number that went left during rebuild window $(j-1)$—call this quantity the ***insertion skew*** $D_{\pi,j-1}$. Then, at the beginning of rebuild window j, π will set rebuild-window j's ***array skew***

$$Q_{\pi,j} = m_\pi \cdot \frac{D_{\pi,j-1}}{\beta w_\pi},$$

where $\beta = \Theta((\log\log n)^2)$ is the parameter defined at the beginning of the section. Finally, using this array skew, π sets $m_{L(\pi)} = m_\pi/2 - Q_{\pi,j}$ and $m_{R(\pi)} = m_\pi/2 + Q_{\pi,j}$.

In other words, π uses odd-numbered rebuild windows for *learning*, and then uses even-numbered rebuild windows for *making use of what it has learned*. Within the even-numbered rebuild windows, π is essentially trying to predict which subproblem will get more insertions, and then giving that subproblem more slots.

In the same way that the window-size selection is the only place the in the algorithm that makes use of randomization, the selection of array skews is the only place that *adapts* to the historical behavior of the input. Of course, if the See-Saw Algorithm were deterministic, it would be easy to construct an insertion sequence that would defeat this type of adaptivity. Thus, it is not just the fact that the See-Saw Algorithm tries to make predictions based on the past, but also the way in which this interacts with the (randomly chosen) window-size

w_π, that together make the algorithm work. The analysis of how adaptivity and randomization work together to minimize expensive leaves will require a number of deep technical ideas, and will constitute the main technical contribution of the paper.

One final source of cost: subproblem resets. So far, we have encountered one way in which a subproblem π's life can end, namely, that one of its ancestors begins a new rebuild window. There will also be another way in which π's life can end: If π receives a total of $m_\pi/\alpha = \Theta(m_\pi/(\log\log n)^2)$ insertions, then π will be ***reset***.[7] This threshold m_π/α for the maximum number of insertions that π can handle before being reset is referred to as its ***quota***. What it means for a subproblem π to be reset is that it (and its descendants) are terminated, and that a new balanced subproblem tree is created in π's place using the *same* subarray and the *same* set of elements as π did. The new subtree is created using the `CreateSubtree` protocol.

One should think of resets as, in some sense, being a technical detail. They are just there to ensure that each subproblem has a bounded number of insertions. The real engine of the algorithm, however, is in the implementation of rebuilds.

A remark on non-smoothness, randomness, and history dependence. As discussed in the introduction, there are three properties that past work has already shown to be necessary if one is to achieve $o(\log^{1.5} n)$ overall amortized expected cost. These properties are *non-smoothness* [28], [30], *randomness* [23], and *history dependence* [10]. It is therefore worth remarking on their roles in the See-Saw Algorithm.

The randomness in the algorithm is used to select the rebuild window size w_π for each subproblem. We will see that, although the input sequence can *attack* the algorithm for one specific choice of w_π, there is no way for it to systematically attack w_π across the entire distribution from which it is selected.

The fact that our algorithm is non-history-independent, and the fact that the rebuilds it performs are non-smooth, are both due to the same step in the algorithm: the step where, at the beginning of each even-numbered rebuild window j, π selects the array skew $Q_{\pi,j}$ adaptively based on what occurred during the previous rebuild window. This adaptivity is fundamentally history dependent, and then the rebuild that it performs on $\mathcal{A}_\pi$ is fundamentally non-smooth (since, for a given rebuild, there is only one possible value for the array skew that would result in the rebuild being smooth).

A remark on how to think about the range of values for array skews. It is worth taking a moment to understand intuitively the range of possible values for the array skew $Q_{\pi,j}$. Since the insertion skew $D_{\pi,j-1}$ satisfies $|D_{\pi,j-1}| \le w_\pi$, the array skew will always satisfy $|Q_{\pi,j}| \le m_\pi/\beta = O(m_\pi/(\log\log n)^2)$. So, perhaps surprisingly, there is a sense in which the array skew is always a *low-order term* compared to the size of the array m_π. On the other hand, the window size w_π is also at most $m_\pi/\alpha = O(m_\pi/(\log\log n)^2)$, so one should think of the maximum possible window size w_π as being comparable to the maximum possible array skew $Q_{\pi,j}$ (and, in fact, the former quantity is the smaller because $\alpha > \beta$).

A remark on packed-memory arrays. Many data-structural applications of list-labeling require the additional property that there are at most $O(1)$ free slots between any two consecutive elements in the array. A list-labeling solution with this property is typically referred to as a ***packed-memory array*** [11], [12], [17]. We remark that the See-Saw Algorithm can be turned into a packed-memory array with the following modification: Whenever the initial density of a non-leaf subproblem π is less than, say, 0.25, we automatically set all of the array skews $Q_{\pi,1}, Q_{\pi,2}, \ldots$ to 0. This turns out to not interfere with the analysis of the See-Saw Algorithm in any way, since as we shall see, the analysis only cares about the array skews in cases where the initial density is at least 0.5 (Lemma 10). On the other hand, with this modification in place, no subproblem π is ever given fewer than $(0.25 - o(1))m_\pi$ elements, which implies that we have a packed-memory array.

V. Algorithm Analysis

In this section, we prove Theorem 3, which, as discussed in Section III, implies the main result of the paper, Theorem 1. We begin by restating Theorem 3 below.

Theorem 3. *Let m be sufficiently large, and $n = m/2$. The See-Saw Algorithm is a list-labeling algorithm that, starting with $m/4$ elements, can support $m/4$ insertions with amortized expected cost $O((\log n)(\log\log n)^3)$.*

The proof of Theorem 3 occupies this section and the next two. In this section we prove the theorem assuming two results, Lemma 10 and Claim 14. These are proved in the following two sections.

In the algorithm description, at any point in time there is a binary tree of subproblems. It is important to keep in mind that the tree is dynamic; subproblems are terminated and new ones are created in their place. As a convention, we will refer to the subproblems that, over time, serve as the roots of the tree as the ***global subproblems***.

Throughout the section, we will make use of the following notation for discussing a subproblem π, some of which were also defined in Section IV:

- $m_\pi = |\mathcal{A}_\pi|$, the size of π's subarray.
- $s_\pi = |\mathcal{S}_\pi^0|$, where $\mathcal{S}_\pi^0$ is the set of elements in π at the *beginning* of its lifetime.
- $\mathcal{I}_\pi$ is the full sequence of inserts that arrive to subproblem π during its lifetime.
- $\mathcal{I}_{\pi,j} \subseteq \mathcal{I}_\pi$ is the subsequence of inserts that arrive to π during its j-th rebuild window.
- $Q_{\pi,j}$ is the value of the array skew used for π's j-th rebuild window.
- $D_\pi(v)$ where $v \in \mathcal{I}_\pi$ is equal to 1 if π sends v right and -1 if π sends v left.

[7] We remark that, in our pseudocode in Appendix B, the *parent* of π is responsible for implementing resets (with the exception of the case where π is the root, which is handled separately).

- $D_\pi(J)$, where $J \subseteq \mathcal{I}_\pi$ is equal to $\sum_{v \in J} D_\pi(v)$, which is the number of elements of J that π sent right minus the number that were sent left.
- $D_\pi = D_\pi(\mathcal{I}_\pi)$ is the total number of inserts to π that went right minus the number that went left.
- $D_{\pi,j} = D_\pi(\mathcal{I}_{\pi,j})$.
- $\sigma_\pi = |\mathcal{I}_\pi|$ is the total number of elements that are inserted into π during its lifetime (not including the s_π elements initially present). Note that, by design, $\sigma_\pi \leq m_\pi/\alpha$.
- $F_\pi = 1 - \frac{\sigma_\pi + s_\pi}{m_\pi}$ is the density of free slots in $\mathcal{A}_\pi$ at the *end* of π.
- $F^0_\pi = 1 - \frac{s_\pi}{m_\pi}$ is the density of free slots in $\mathcal{A}_\pi$ at the *beginning* of π.
- K_π is the value of the random integer that determines the rebuild window size $w_\pi = m_\pi/(\alpha 2^{K_\pi})$.
- t_π is the total number of rebuild windows that π starts over its lifetime.

We will often drop the subscript π, e.g., on $\mathcal{A}_\pi, \mathcal{S}^0_\pi, \mathcal{I}_{\pi,j}$ and $Q_{\pi,j}$, when the subproblem is clear from context. However, when we write m, we always mean the full array size.

We organize the set of all subproblems that exist throughout the algorithm into a nonbinary tree called the ***history tree***. For a given subproblem π, its children will be all left and right subproblems that it ever creates. (The root of the tree is a fictitious root subproblem, and the children of the root are the global subproblems.) A subproblem π will have at least t_π different left subproblems and t_π right subproblems, since it starts a new left and right subproblem at the beginning of each rebuild window. (Recall that t_π is the total number of rebuild windows of π.) A subproblem may have more than one left or right child subproblem per rebuild window because a child subproblem may reach its quota, which causes it to reset, causing it to get replaced by a new subproblem. The leaves of the history tree are the expensive-leaf and tiny-leaf subproblems.

Note that, for a given subproblem π in the history tree, the number of children π has is not fixed in advance but depends both on the random choices of window sizes by π and its ancestors, and also on the specific insertion sequence (the set of which subproblems get terminated because they reach their quotas may depend on the specific insertion sequence).

A. The Basics: Proving Correctness, and Bounding the Costs of Rebuilds, Resets, and Tiny Leaves

We start with some basic observations:

Proposition 5. *For any subproblem π:*

1) *If π is non-global, then $m_\pi \in [0.49m_\rho, 0.51m_\rho]$, where ρ is the parent of π.*
2) *The total number of items $s_\pi + \sigma_\pi$ that π must store in its subarray is at most $0.8m_\pi$.*

Proof. For the first part, if π is inside the j-th rebuild window of ρ then the size of π's array is $\frac{1}{2}m_\rho \pm |Q_{\rho,j}|$ and $|Q_{\rho,j}| \leq \frac{|D_{\rho,j-1}|m_\rho}{w_\rho \beta} \leq m_\rho/\beta \leq 0.01m_\rho$ (since $\beta \geq 100$).

For the second part, the assertion is true for any global subproblem since the total number of elements in the array never exceeds $m_\pi/2$. For a non-global subproblem π with parent ρ, it must be that ρ is not an expensive leaf (since expensive leaves don't initiate subproblems), so $s_\rho \leq 0.75m_\rho$. Assume without loss of generality that π is a left subproblem of ρ. Recall that, when ρ is created, it gives half of the elements in $\mathcal{S}^0_\rho$ to its left child, and that whenever ρ rebuilds its children, it does not move any elements between them (it just changes the sizes of their arrays); thus, the number of elements from $\mathcal{S}^0_\rho$ that π contains is just $|\mathcal{S}^0_\rho|/2 = s_\rho/2$. So the number of elements $s_\pi + \sigma_\pi$ in π at the end of π's lifetime is at most

$$s_\rho/2 + \sigma_\rho \leq 0.38m_\rho + m_\rho/\alpha \leq 0.39m_\rho.$$

By the first part of the proposition, we have $m_\rho \leq m_\pi/0.49$, so our bound on $s_\pi + \sigma_\pi$ is at most

$$0.39(m_\pi/0.49) \leq 0.8m_\pi.$$

□

As a corollary, we can establish the correctness of the See-Saw Algorithm.

Corollary 6. *The See-Saw Algorithm is a valid list-labeling algorithm.*

Proof. In the algorithm, each successive insert is passed down the current subproblem tree to a leaf subproblem which inserts the item into its subarray using the classical algorithm. The classical algorithm at leaf subproblem π will fail to carry out an insertion only if the total number of items assigned to π exceeds m_π, but the final part of Proposition 5 ensures that this does not happen. The only other times that items are moved in the array are when a non-leaf problem does a rebuild of one or both of its subproblems. Such a rebuild will fail only if for a created subproblem ρ the number of items s_ρ initially assigned to ρ exceeds m_ρ, which again is impossible by the last part of Proposition 5.

The above guarantees that after each insertion, all items inserted so far are placed in the array. It remains to verify that the ordering of the items in the array is consistent with the intrinsic ordering on items. At any point in the execution, if π is an active leaf subproblem then the items in the subarray of π are in order by the correctness of the classical algorithm. If two items are assigned to different leaves π and ρ with π to the left of ρ (under the usual left-to-right ordering of leaves) then it is easy to see from the definition of the algorithm that the subarray of π is entirely to the left of the subarray of ρ and the items assigned to π are all less than the items assigned to ρ, so the two items will be in correct order. □

It remains to bound the cost of the algorithm. Define the ***level*** of a subproblem to be its depth in the history tree, where global subproblems are said to have level 1. The first part of Proposition 5 implies that the maximum level of any subproblem is at most $1.5 \log m \leq 2 \log n$.

The cost incurred by the data structure can be broken into four groups: (1) The cost of rebuilds, which occur every time

that a non-leaf subproblem π finishes a rebuild window and begins a new one; (2) the cost of resets, which which occur whenever a subproblem reaches its quota for the total number of insertions it can process; (3) the cost of tiny subproblems; and (4) the cost of expensive leaf subproblems.

We can bound the first three of these with the following lemma:

Lemma 7. *The total expected amortized cost (across all subproblems) from rebuilds, resets, and tiny subproblems is* $O((\log n)(\log\log n)^3)$ *per insertion.*

Proof. First we bound the cost of all resets. A reset is done when a subproblem π has reached its quota of m_π/α insertions, and the cost of the reset is m_π. We can bound the sum of these reset costs by charging α to each insertion that went through π. Overall, each insertion travels through $O(\log n)$ total subproblems, and therefore gets charged $O(\alpha \log n) = O((\log n)(\log\log n)^2)$. Thus the amortized expected cost of resets is $O((\log n)(\log\log n)^2)$.

Next we bound the cost of rebuilds. The number of rebuilds that a subproblem π performs is $t_\pi - 1 = \lfloor(\sigma_\pi - 1)/w_\pi\rfloor$ where again t_π is its number of rebuild windows, σ_π is the total number of elements inserted into π during its lifetime, and w_π is the rebuild window size. (Remember that, crucially, a subproblem does not perform a rebuild at the beginning of its first window, as the elements in $\mathcal{A}_\pi$ are already evenly spread out at that point in time, which is the state that π initially wants.) Recall that $w_\pi = m_\pi/(\alpha 2^K)$ where $K = K_\pi$. If $K = 0$ then $w_\pi = m_\pi/\alpha$ which is precisely π's quota, so the number of rebuilds is $\lfloor(\sigma_\pi - 1)/w_\pi\rfloor \le \lfloor(m_\pi/\alpha - 1)/w_\pi\rfloor = 0$. For $K \ge 1$ the number of rebuilds performed by π is at most $\sigma_\pi/(m_\pi/(\alpha 2^K)) = \frac{2^K \alpha \sigma_\pi}{m_\pi}$ and each rebuild has cost m_π, so the total cost is at most $2^K \alpha \sigma_\pi$.

Recalling that, for $k \in [1, k_{\max}]$, we have $\Pr[K = k] = p_k = 2^{-(k+1)}(1 + k/k_{\max}) \le 2^{-k}$, we can take the expected value over all choices for K to bound the expected total cost of π's rebuilds by

$$\sum_{k=1}^{k_{\max}} p_k 2^k \alpha \sigma_\pi \le \sum_{k=1}^{k_{\max}} 2^{-k} 2^k \alpha \sigma_\pi = k_{\max} \alpha \sigma_\pi = O((\log\log n)^3 \sigma_\pi),$$

where the last equality follows since $k_{\max} = O(\log\log n)$ and $\alpha = O((\log\log n)^2)$.

Since the insertion sets for the subproblems at any fixed level of recursion are disjoint, the total expected cost of rebuilds at each level is $O(n(\log\log n)^3)$. Summing over the $O(\log n)$ levels yields an amortized expected cost of $O((\log n)(\log\log n)^3)$ per operation.

Finally, because tiny subproblems have size at most $2^{O(\sqrt{\log n})}$, they incur amortized expected cost at most $O(\log^2(2^{O(\sqrt{\log n})})) = O(\log n)$ per insertion. □

B. Bounding the Costs of Expensive Leaves

It remains to bound the cost of expensive leaves. These leaves may incur amortized cost as large as $O(\log^2 n)$ per insertion. So we want to show that the expected number of insertions that reach expensive leaves is $O(n/\log n)$. This is indeed true, and the proof occupies most of the rest of the paper.

Each insertion follows a unique root-to-leaf path in the history tree. We now define some notation for how to think about this path for a specific insertion v:

- $d(v)$ is the number of subproblems in v's path.
- $\pi_1(v), \ldots, \pi_{d(v)}(v)$ is the path of subproblems that v follows.
- $F_j^0(v) = F^0_{\pi_j(v)}$.
- $F_j(v) = F_{\pi_j(v)}$.
- For $\pi = \pi_j(v)$, $j < d(v)$, define $\delta_\pi(v) = F_{j+1}(v) - F_j(v)$, so that $F_j(v) - F_1(v) = \sum_{i=1}^{j-1} \delta_{\pi_i(v)}(v)$.

All of the above are random variables that depend on both the sequence of insertions that has occurred prior to v and the random choices of the algorithm, i.e., the parameters K_ρ for all subproblems ρ.

By definition, the leaf subproblem $\pi_{d(v)}(v)$ is an expensive leaf if and only if $F^0_{d(v)}(v) \le 1/4$ which implies $F_{d(v)}(v) \le 1/4$. On the other hand, since $\pi_1(v)$ is a global subproblem and the total number of elements ever present is at most $m/2$, $F_{\pi_1(v)} \ge 1/2$. Thus, we obtain the following necessary condition for v to reach an expensive leaf:

$$\sum_{i=1}^{d(v)-1} \delta_{\pi_i(v)}(v) \le -1/4. \tag{1}$$

For a non-leaf subproblem π and insertion $v \in \mathcal{I}_{\pi,i}$ (recall that $\mathcal{I}_{\pi,i}$ denotes the insertions in the i-th rebuild window of π), define:

$$\Delta_\pi(v) = \begin{cases} \dfrac{D_\pi - 2(1-F_\pi)Q_i}{m_\pi - 2Q_i} & \text{if } v \text{ is sent left by } \pi \\ \dfrac{-D_\pi + 2(1-F_\pi)Q_i}{m_\pi + 2Q_i} & \text{if } v \text{ is sent right by } \pi. \end{cases}$$

This definition comes out of the following lemma, which shows that $\Delta_\pi(v)$ lower bounds $\delta_\pi(v)$.

Lemma 8. *Let π be a non-leaf subproblem and let $v \in \mathcal{I}_\pi$. Then,*

$$\delta_\pi(v) \ge \Delta_\pi(v).$$

Proof. Suppose v occurs during the j-th rebuild window of π. Let ρ be the child subproblem of π (active during $\mathcal{I}_{\pi,j}$) that v is assigned to. We will assume that ρ is a left subproblem of π; the other case follows by a symmetric argument with the appropriate changes of sign.

By definition, $m_\rho = m_\pi/2 - Q_{\pi,j}$. At the beginning of the first rebuild window of π, the left child of π starts with $s_\pi/2$ items, where s_π is the number of items initially stored in $\mathcal{A}_\pi$, and these items will be assigned to every left subproblem created in subsequent windows of π. The total number of

inserts received by π that go left is $(|\mathcal{I}_\pi| - D_\pi)/2$, so the total number of elements ever stored in ρ is at most

$$s_\pi/2 + (|\mathcal{I}_\pi| - D_\pi)/2,$$

which implies that the total number of free slots in $\mathcal{A}_\rho$ is always at least

$$\begin{aligned} &m_\rho - s_\pi/2 - (|\mathcal{I}_\pi| - D_\pi)/2 \\ &\quad = m_\pi/2 - Q_{\pi,j} - s_\pi/2 - (|\mathcal{I}_\pi| - D_\pi)/2 \\ &\quad = (m_\pi - s_\pi - |\mathcal{I}_\pi|)/2 + D_\pi/2 - Q_{\pi,j} \\ &\quad \geq F_\pi m_\pi/2 + D_\pi/2 - Q_{\pi,j}. \end{aligned}$$

The free-slot density in ρ at the end of its lifetime therefore satisfies

$$\begin{aligned} F_\rho &\geq \frac{F_\pi m_\pi/2 + D_\pi/2 - Q_{\pi,j}}{m_\rho} \\ &= \frac{F_\pi(m_\pi/2 - Q_{\pi,j}) + D_\pi/2 - (1-F_\pi)Q_{\pi,j}}{m_\pi/2 - Q_{\pi,j}} \\ &= F_\pi + \frac{D_\pi/2 - (1-F_\pi)Q_{\pi,j}}{m_\pi/2 - Q_{\pi,j}} \\ &= F_\pi + \frac{D_\pi - 2(1-F_\pi)Q_{\pi,j}}{m_\pi - 2Q_{\pi,j}} = F_\pi + \Delta_{\pi(v)}. \end{aligned}$$

□

Before continuing, it is worth remarking on two features of $\Delta_\pi(v)$, for $v \in \mathcal{I}_{\pi,i}$, that make it nice to work with (and that, at least in part, shape its definition).

The first property is that, if D_π and Q_i were both zero (which would, happen, for example, if the insertions in π alternated evenly between π's left and right children), then $\Delta_\pi(v)$ would also be zero. This means that one should think of $\Delta_\pi(v)$ as having a "default" value of zero, which is why later on (in Lemma 10), when we want to bound $\mathrm{Var}(\Delta_\pi(v))$, we will be able to get away with bounding $\mathbb{E}[(\Delta_\pi(v))^2]$ instead.[8]

The second property is that $\Delta_\pi(v)$ is the *same* for all v in a given rebuild window $\mathcal{I}_{\pi,i}$. In fact, if $\pi = \pi_j(v)$ for some j, then all $u \in \mathcal{I}_\pi$ agree on the values of $\Delta_{\pi_1(u)}(u), \ldots, \Delta_{\pi_{j-1}(u)}(u)$. This property will be critical for our analysis (in Lemma 13), and later on, of how the sequence $\Delta_{\pi_1(v)}(v), \Delta_{\pi_2(v)}(v), \ldots$ behaves. This property is also the reason why all of the quantities used to define $\Delta_\pi(v)$ (i.e., $D_\pi(v), F_j(v), Q_{\pi,i}$) are based only on the window $\mathcal{I}_{\pi,i}$ that contains v, rather than on anything more specific about the insertion v.

For an insert v, define $\Delta_1(v), \ldots, \Delta_{d(v)-1}(v)$ by $\Delta_i(v) = \Delta_{\pi_i(v)}(v)$ for $i < d(v)$. Combining Lemma 8 with (1), we get a new necessary condition for v to reach an expensive leaf:

$$\sum_{i=1}^{d(v)-1} \Delta_i(v) \leq -1/4.$$

[8]Note that, no matter what, we have $\mathrm{Var}(\Delta_\pi(v)) \leq \mathbb{E}[(\Delta_\pi(v))^2]$, so one can always use the latter as an upper bound for the former. What is important here, is that the latter quantity is actually a *good* upper bound for the former.

We will bound the fraction of v's that arrive at an expensive leaf by showing that at most an expected $O(1/\log n)$ fraction of insertions v satisfy the above condition. To analyze this fraction, we adopt a probabilistic point of view with regard to the insertions themselves. In particular, rather than analyzing the probability that any *specific* insertion reaches an expensive leaf, we will select a uniformly random insertion v from the entire insertion sequence (this randomness is for the sake of analysis, only, and is not coming from the randomness in the algorithm), and we will analyze the probability that this randomly selected insertion reaches an expensive leaf. Thus, the underlying probability space will be over both randomly chosen v and the randomness of the algorithm.

To analyze a random insertion v, our main task will be to analyze the (random) sequence $\Delta_1(v), \Delta_2(v), \ldots$. We first make the observation (Proposition 9) that each $\Delta_i(v)$ is bounded in absolute value by $\frac{3}{\beta}$. The more significant result is then Lemma 10, which says that under fairly general conditions on a subproblem π, the variance of $\Delta_\pi(v)$, for a uniformly random $v \in \mathcal{I}_\pi$, is (deterministically) bounded above by a small (sub-constant) multiple of its expectation (up to a negligible additive term). Once we have these properties, we will argue that they force $\Delta_1(v), \Delta_2(v), \ldots$ to evolve according to a well-behaved process, which we will then analyze using (mostly) standard results from the theory of random walks.

Proposition 9. *For any non-leaf subproblem π and any $v \in \mathcal{I}_\pi$, we have $|\Delta_\pi(v)| \leq \frac{3}{\beta}$.*

Proof. Let $\mathcal{I}_i$ be the rebuild window of $\mathcal{I}_\pi$ such that $v \in \mathcal{I}_i$. Note that the array skew Q_i satisfies $|Q_i| \leq m_\pi \cdot \frac{|D_{\pi,i-1}|}{\beta w_\pi} \leq m_\pi/\beta$. Using this, we can conclude that

$$\begin{aligned} |\Delta_\pi(v)| &\leq \frac{|D_\pi| + 2|Q_i|}{m_\pi - 2|Q_i|} \\ &\leq (1 + 3/\beta) \cdot \left(\frac{|D_\pi| + 2|Q_i|}{m_\pi}\right) \\ &\leq (1 + 3/\beta) \cdot \left(\frac{|D_\pi|}{m_\pi}\right) + (1 + 3/\beta) \cdot 2/\beta \\ &\leq (1 + 3/\beta)/\alpha + (1 + 3/\beta) \cdot 2/\beta \\ &\qquad\qquad \text{(since } |D_\pi| \leq |\mathcal{I}_\pi| \leq m_\pi/\alpha\text{)} \\ &\leq 3/\beta, \end{aligned}$$

where the final inequality uses that $\beta = \omega(1)$ and $\alpha > 100\beta$. □

We now come to the main technical lemma, which we will prove in Section VI.

Lemma 10. *(The See-Saw Lemma) Let π be a non-leaf subproblem with insertion set $\mathcal{I}_\pi$, and suppose that F_π^0, the free-slot density of π when it starts, satisfies $F_\pi^0 \leq 0.5$. Then, for a uniformly random $v \in \mathcal{I}_\pi$,*

$$\mathbb{E}[\Delta_\pi(v)^2] \leq \frac{100 k_{\max}}{\beta} \mathbb{E}[\Delta_\pi(v)] + 2^{-k_{\max}},$$

where the expectations are taken over both the random choice of v and the algorithm's random choice of w_π.

Lemma 10 is the part of the analysis that captures the role of *adaptivity* in our algorithm. If the algorithm were not adaptive (i.e., always set $Q_i = 0$), then $\Delta_\pi(v)$ would simply be D_π/m_π. The insertion sequence would then be able to force $\Delta_\pi(v)^2 = (D_\pi/m_\pi)^2$ (and, more importantly, $\mathrm{Var}(\Delta_\pi(v))$) to be large by sending more insertions to one child of π than to the other. (This would also cause $\mathbb{E}[\Delta_\pi(v)]$ to be slightly negative, which would also be bad for us.) The key insight in Lemma 10 is that we cannot hope to prevent $\Delta_\pi(v)^2$ from being large—but we *can hope* to use adaptivity in order to create a "see-saw" relationship between $\mathbb{E}[\Delta_\pi(v)^2]$ and $\mathbb{E}[\Delta_\pi(v)]$. In particular, if the insertion sequence chooses to send far more insertions to one child than the other, then this creates an opportunity for us to employ adaptivity, which we can then use to put more free slots on the side that receives more insertions, which allows for us to create a positive expected value for $\mathbb{E}[\Delta(v)]$. This, in turn, is a good thing, since $\Delta(v)$ being positive means that, on average, insertions experience a free-slot density *increase* when traveling from π to π's child. Thus, we create a situation where, no matter what, we win: either $\mathbb{E}[\Delta(v)]$ is large (which is good), or $\mathrm{Var}(\Delta_\pi(v))$ is small (which is also good!).

Thus, the "magic" of the See-Saw Algorithm will be in how it uses adaptivity to guarantee the See-Saw Lemma. A priori, the adaptive behavior of the algorithm (i.e, the way in which it selects $Q_{\pi,i}$ based on the insertion behavior in the previous rebuild window) would seem to be quite difficult to analyze. Intuitively, the algorithm is attempting to observe when there are "trends" in the insertion-sequence's behavior. However, if we are not careful, the insertion sequence may be able to trick us into observing a "trend" in one rebuild window, even though the next rebuild window will behave in the opposite way. The main contribution of Section VI, where we prove Lemma 10, is that if the window size w_π and the array skews $Q_{\pi,1}, Q_{\pi,2}, \ldots$ are selected in just the right way (as in the See-Saw Algorithm), then it is possible to perform a telescoping argument that holds for *any input*. The argument shows that, even if the insertion sequence causes the algorithm to perform badly for some choices of w_π, this creates "opportunities" for the algorithm to perform better on other choices of w_π, so that on average the algorithm always does well.

Since Lemma 10 only considers subproblems π satisfying $F^0_\pi \le 0.5$, it will be useful to define a modified version of Δ_π, where for any non-leaf subproblem π, we have:

$$\widehat{\Delta}_\pi(v) = \begin{cases} \Delta_\pi(v) & \text{if } F^0_\pi \le 0.5 \\ 0 & \text{otherwise.} \end{cases}$$

Trivially, we then have:

Corollary 11. *For any non-leaf subproblem π, and for a uniformly random $v \in \mathcal{I}_\pi$, we have*

$$\mathbb{E}[\widehat{\Delta}_\pi(v)^2] \le \frac{100 k_{\max}}{\beta}\mathbb{E}[\widehat{\Delta}_\pi(v)] + 2^{-k_{\max}}.$$

We define $\widehat{\Delta}_1(v), \widehat{\Delta}_2(v), \ldots$ by $\widehat{\Delta}_i(v) = \widehat{\Delta}_{\pi_i(v)}(v)$. Earlier we gave a necessary condition for reaching an expensive leaf, based on summations of $\{\Delta_i(v)\}$. We now give a similar condition based on the sequence $\{\widehat{\Delta}_i(v)\}$:

Proposition 12. *For any insert v, if the path of v ends at an expensive leaf, then there is an interval $[a, b] \subseteq [1, d(v) - 1]$ such that*

$$\widehat{\Delta}_a(v) + \cdots + \widehat{\Delta}_b(v) \le -0.23.$$

Proof. Suppose that the leaf $\pi_{d(v)}(v)$ is an expensive leaf. So $F^0_{d(v)}(v) \le 1/4$, which implies $F_{d(v)}(v) \le 1/4$. Let $b = d(v) - 1$ and let ℓ be the largest index such that $F^0_\ell(v) \ge 1/2$; this is well-defined because $\pi_1(v)$ is global and so $F^0_1(v) \ge 1/2$. Since $\alpha \ge 100$, and since $\pi_\ell(v)$ gets at most $m_{\pi_\ell(v)}/\alpha$ insertions, we have $F_\ell(v) \ge F^0_\ell(v) - 0.01 \ge 0.49$. By the definition of ℓ, $\widehat{\Delta}_i(v) = \Delta_i(v)$ for all $i \in [\ell + 1, b]$. Letting $a = \ell + 1$, and letting $\delta_i(v)$ denote $\delta_{\pi_i(v)}$, we have by Lemma 8 that

$$\begin{aligned} \sum_{i=a}^{b} \widehat{\Delta}_i(v) &= -\Delta_\ell(v) + \sum_{i=\ell}^{b} \Delta_i(v) \\ &\le -\Delta_\ell(v) + \sum_{i=\ell}^{b} \delta_i(v) \\ &= -\Delta_\ell(v) + F_{b+1}(v) - F_\ell(v) \\ &\le -\Delta_\ell(v) + 1/4 - 0.49. \end{aligned}$$

Finally, by Proposition 9, this is at most $3/\beta + 1/4 - 0.49 \le 0.01 + 1/4 - 0.49$, since $\beta \ge 300$. □

We will now show how to bound the probability that a random insertion v (out of the *entire* input stream) encounters an expensive leaf.

Lemma 13. *Consider a uniformly random insertion v out the entire insertion stream. The probability that v reaches an expensive leaf is $O(1/\log n)$.*

Proof. Since v is a random variable, the sequences $\pi_1(v), \pi_2(v), \ldots$ and $\widehat{\Delta}_1(v), \widehat{\Delta}_2(v), \ldots$ are also random variables. When discussing our randomly chosen v, we will use $\widehat{\Delta}_j$ as a shorthand for $\widehat{\Delta}_j(v)$. For convenience of notation, we let $\widehat{\Delta}_j = 0$ if $j \ge d(v)$.

By Proposition 12, it suffices to upper bound the probability that there is a pair $a \le b$ satisfying $\widehat{\Delta}_a + \cdots + \widehat{\Delta}_b \le -.23$. The maximum depth of the history tree is $2\log n$ so there are at most $4\log^2 n$ pairs with $a \le b < 2\log n$. So it suffices to fix $a \le b < 2\log n$ and show that $\Pr[\sum_{i=a}^{b} \widehat{\Delta}_i(v) \le -0.23] = O(1/\log^3 n)$.

In Section VII we will use standard concentration bounds for random processes to show:

Claim 14. *Let $k_{\max} = 2\log\log n$, and let $\beta = C_\beta(\log\log n)^2$ for some sufficiently large positive constant C_β. Let $X_1, X_2, \ldots, X_r$ be random variables with $r \le 2\log n$ such that for $i \in [1, r]$:*

1) $|X_i| \le 3/\beta$;

2) $\mathbb{E}[X_i^2 \mid X_1, \ldots, X_{i-1}] \leq \frac{100k_{\max}}{\beta}\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}] + 2^{-k_{\max}}$.

Then, $\Pr[\sum_i X_i < -0.2] \leq O\left(1/\log^3 n\right)$.

Although we will defer the proof of Claim 14 to Section VII, it may be worth taking a moment to explain the intuition behind the claim. For this, it is helpful to substitute X_i with $X_i' := X_i \cdot \log\log n$. Under this substitution (and with a bit of algebra) one can reduce the the hypotheses of the claim to (1) $|X_i'| \leq O(1/\log\log n) \leq 1$, and (2) $\mathbb{E}[X_i'^2 \mid X_1', \ldots, X_{i-1}'] \leq O(1)\cdot\mathbb{E}[X_i' \mid X_1', \ldots, X_{i-1}'] + \widetilde{O}(1/\log^2 n)$; and the conclusion of the claim becomes that, with probability $1 - 1/\log^3 n$, we have $\sum_i X_i' \geq -O(\log\log n)$. In other words, the essence of the claim is simply that, if a random walk has steps of size at most, say 1, and if each step has mean at least a constant factor larger than its variance (modulo some small additive error), then the random walk will not be able to become substantially negative with any substantial probability.

We would like to apply Claim 14 to $X_1, X_2, \ldots = \widehat{\Delta}_a, \widehat{\Delta}_{a+1}, \ldots, \widehat{\Delta}_b$. Proposition 9 implies that each $\widehat{\Delta}_i$ satisfies the first hypothesis of the claim, and the second hypothesis *almost* follows from Corollary 11. The only issue is that Corollary 11 tells us how to think about $\widehat{\Delta}_\pi(v)$ for a random v *out of those in* $\mathcal{I}_\pi$, but what we actually want to reason about is $\widehat{\Delta}_i(v) \mid \widehat{\Delta}_a(v), \ldots, \widehat{\Delta}_{i-1}(v)$ for a random v *out of all insertions*. Fortunately, these two probability distributions end up (by design) being closely related to one another, allowing us to establish the following variation of Corollary 11:

Corollary 15. *For each* $i \in [1, 2\log n]$,

$$\mathbb{E}[\widehat{\Delta}_i^2 \mid \widehat{\Delta}_1, \ldots, \widehat{\Delta}_{i-1}] \leq \frac{100k_{\max}}{\beta}\mathbb{E}[\widehat{\Delta}_i \mid \widehat{\Delta}_1, \ldots, \widehat{\Delta}_{i-1}] + 2^{-k_{\max}}.$$

Proof. Because, in this proof, we will use π_i as a formal random variable, it is helpful to think of each π_i as formally being given by the triple $(\mathcal{A}_{\pi_i}, \mathcal{S}^0_{\pi_i}, \mathcal{I}_{\pi_i})$.

Recall that our probability space consists of the selection of the parameters w_π during the algorithm (which, along with the insertion sequence, fully determine the history tree) and the selection of a uniformly random v that determines the path $\pi_1, \pi_2, \ldots, \pi_j$ down the tree. We will need an alternative incremental description of the probability space. Keep in mind that the full sequence of insertions is fixed. First note that the global subproblems are completely determined by the insertion sequence (i.e., there is no randomness) and the insertion sets for these subproblems partition the full set of insertions. Select π_1 from among the global subproblems with probability proportional to the size of its set of insertions. Next select the parameter $w_1 = w_{\pi_1}$ according to the algorithm specification. The parameter w_1 determines the windows and the set of subproblems of π_1, and the insertion sets for these subproblems partition the insertion set of π_1. Next we select π_2 from among these subproblems with probability proportional to the size of its insert set. We continue in this way selecting $\pi_1, w_1, \pi_2, w_2, \ldots$ until we arrive either at a tiny leaf or expensive leaf. This process gives the same distribution over paths $\pi_1, \pi_2, \ldots$, as the distribution that first runs the algorithm to determine the full history tree and then selects a random insert and follows its path.

Now let us consider the random variable $\widehat{\Delta}_i \mid \pi_1, w_1, \ldots, \pi_i$, for a given $i \in [1, 2\log n]$. So that this is well defined for all i, we can artificially define $\pi_j(v)$ and $w_j(v)$ to be null, for $j > d(v)$ and $j \geq d(v)$, respectively. If π_i is a leaf (or null), then $\widehat{\Delta}_i \mid \pi_1, w_1, \ldots, \pi_i$ is defined to be identically zero, so we have trivially that

$$\mathbb{E}[\widehat{\Delta}_i^2 \mid \pi_1, w_1, \ldots, \pi_i] \leq \frac{100k_{\max}}{\beta}\mathbb{E}[\widehat{\Delta}_i \mid \pi_1, w_1, \ldots, \pi_i] + 2^{-k_{\max}}.$$

The interesting case is what happens if π_i is a non-leaf subproblem.

For any given set of outcomes for $\pi_1, w_1, \ldots, \pi_i$, where π_i is a non-leaf subproblem, the probabilistic rule for selecting w_i and π_{i+1} (which together determine $\widehat{\Delta}_i$) is completely determined by π_i. Therefore, if we fix any set of outcomes for $\pi_1, w_1, \ldots, \pi_i$, and if we use $\widehat{\Delta}_\pi$ to denote the random variable $\widehat{\Delta}_\pi(u)$ for a uniformly random $u \in \mathcal{I}_\pi$, then we have

$$\mathbb{E}[\widehat{\Delta}_i \mid \pi_1, w_1, \ldots, \pi_i] = \mathbb{E}[\widehat{\Delta}_i \mid \pi_i] = \mathbb{E}[\widehat{\Delta}_{\pi_i}],$$
$$\mathbb{E}[\widehat{\Delta}_i^2 \mid \pi_1, w_1, \ldots, \pi_i] = \mathbb{E}[\widehat{\Delta}_i^2 \mid \pi_i] = \mathbb{E}[\widehat{\Delta}_{\pi_i}^2].$$

Now we can combine this with Corollary 11 to obtain

$$\mathbb{E}[\widehat{\Delta}_i^2 \mid \pi_1, w_1, \ldots, \pi_i] \leq \frac{100k_{\max}}{\beta}\mathbb{E}[\widehat{\Delta}_i \mid \pi_1, w_1, \ldots, \pi_i] + 2^{-k_{\max}}.$$

Since, earlier in the proof, we also established this identity for the case where π_i is a leaf (or null), we can conclude that the identity holds for all options of $\pi_1, w_1, \ldots, \pi_i$.

Finally note that $\pi_1, w_1, \ldots, \pi_i$ determines $\widehat{\Delta}_1, \ldots, \widehat{\Delta}_{i-1}$. Therefore for any fixing $d_1, \ldots, d_{i-1}$ of $\widehat{\Delta}_1, \ldots, \widehat{\Delta}_{i-1}$, we can average the previous inequality with respect to the conditional distribution on $\pi_1, w_1, \ldots, \pi_i$ given $\widehat{\Delta}_1 = d_1, \ldots, \widehat{\Delta}_{i-1} = d_{i-1}$ and this gives exactly the desired result. □

An immediate consequence of Corollary 15 is that, for any interval $[a, b]$, and for $i \in [a, b]$, we have $\mathbb{E}[\widehat{\Delta}_i^2 \mid \widehat{\Delta}_a, \ldots, \widehat{\Delta}_{i-1}] \leq \frac{100k_{\max}}{\beta}\mathbb{E}[\widehat{\Delta}_i \mid \widehat{\Delta}_a, \ldots, \widehat{\Delta}_{i-1}] + 2^{-k_{\max}}$. We also have by Proposition 9 that $|\widehat{\Delta}_i| \leq 3/\beta$, so we can apply Claim 14, using $X_1, X_2, \ldots = \widehat{\Delta}_a, \widehat{\Delta}_{a+1}, \ldots, \widehat{\Delta}_b$ to complete the proof that $\Pr[\widehat{\Delta}_a + \cdots + \widehat{\Delta}_b \leq -0.23] = O(1/\log^3 n)$, which, in turn, completes the proof of the lemma. □

Given Lemma 13, we can complete the proof of Theorem 3 as follows.

Theorem 3. *Let* m *be sufficiently large, and* $n = m/2$*. The See-Saw Algorithm is a list-labeling algorithm that, starting with* $m/4$ *elements, can support* $m/4$ *insertions with amortized expected cost* $O((\log n)(\log\log n)^3)$.

Proof. Lemma 7 bounds the amortized expected costs of rebuilds, resets, and tiny leaves by

$$O((\log n)(\log\log n)^3).$$

Lemma 13 bounds the probability of an insertion encountering an expensive leaf by $O(1/\log n)$. If an insertion does encounter an expensive leaf, it incurs $O(\log^2 n)$ amortized expected cost within the leaf. Thus, the amortized expected cost per insertion from expensive leaves is $O(\log n)$. □

It remains to prove Lemma 10 and Claim 14. These are given in Sections VI and VII.

VI. Proof of The See-Saw Lemma

In this section, we prove Lemma 10, restated below:

Lemma 10. *(The See-Saw Lemma) Let π be a non-leaf subproblem with insertion set $\mathcal{I}_\pi$, and suppose that F_π^0, the free-slot density of π when it starts, satisfies $F_\pi^0 \le 0.5$. Then, for a uniformly random $v \in \mathcal{I}_\pi$,*

$$\mathbb{E}[\Delta_\pi(v)^2] \le \frac{100k_{\max}}{\beta}\mathbb{E}[\Delta_\pi(v)] + 2^{-k_{\max}},$$

where the expectations are taken over both the random choice of v and the algorithm's random choice of w_π.

Let us also recall the definition of $\Delta_\pi(v)$. For a subproblem π and insertion $v \in \mathcal{I}_{\pi,j}$ (recall that $\mathcal{I}_{\pi,j}$ denotes the insertions in the j-th rebuild window of π):

$$\Delta_\pi(v) = \begin{cases} \dfrac{D_\pi - 2(1-F_\pi)Q_j}{m_\pi - 2Q_j} & \text{if } v \text{ is sent left by } \pi \\ \dfrac{-D_\pi + 2(1-F_\pi)Q_j}{m_\pi + 2Q_j} & \text{if } v \text{ is sent right by } \pi. \end{cases}$$

Since π is the only subproblem that will be mentioned in this proof, we'll often omit the subscript π.

To prove the lemma, we will prove a lower bound on $\mathbb{E}[\Delta(v)]$ and an upper bound on $\mathbb{E}[\Delta(v)^2]$ and then compare them. These expectations would be easier to deal with if we could change the denominator in $\Delta(v)$ to m_π. In particular, the expressions for insertions v that go left vs right would then be negatives of each other. Recalling that $D(v)$ is $+1$ if v goes right and -1 if v goes left, define

$$\Lambda(v) = \Lambda_\pi(v) = D(v)\frac{-D_\pi + 2(1-F_\pi)Q_j}{m_\pi}.$$

We will use $\Lambda(v)$ to estimate $\Delta(v)$. Define the error function

$$\varepsilon(v) = \Delta(v) - \Lambda(v).$$

The following claim will inform how we think about $\varepsilon(v)$:

Claim 16. *If $v \in \mathcal{I}_{\pi,j}$ for some j, then we have*

$$\varepsilon(v) = \begin{cases} \dfrac{2Q_j}{m_\pi - 2Q_j}\Lambda(v) & \text{if } v \text{ is sent left by } \pi \\ \dfrac{-2Q_j}{m_\pi + 2Q_j}\Lambda(v) & \text{if } v \text{ is sent right by } \pi, \end{cases}$$

and that

$$|\varepsilon(v)| \le \frac{8|Q_j|}{3m_\pi}|\Lambda(v)| \le |\Lambda(v)|/3.$$

Proof. We have $\frac{\Delta(v)}{\Lambda(v)} = \frac{m_\pi}{m_\pi + D(v)\cdot 2Q_j}$, which implies

$$\begin{aligned}\frac{\Delta(v)-\Lambda(v)}{\Lambda(v)} &= \frac{m_\pi - (m_\pi + D(v)\cdot 2Q_j)}{m_\pi + D(v)\cdot 2Q_j} \\ &= \frac{-D(v)\cdot 2Q_j}{m_\pi + D(v)\cdot 2Q_j}.\end{aligned}$$

To prove the second part of the claim, observe that $|Q_j| = \frac{|D_{j-1}|m_\pi}{\beta w_j} \le \frac{m_\pi}{\beta} \le \frac{m_\pi}{8}$, and so:

$$|\varepsilon(v)| \le \frac{2|Q_j|}{\frac{3}{4}m_\pi}|\Lambda(v)| = \frac{8|Q_j|}{3m_\pi}|\Lambda(v)| \le |\Lambda(v)|/3.$$

□

We can bound $\mathbb{E}[\Delta(v)]$ and $\mathbb{E}[\Delta(v)^2]$ as a function of $\Lambda(v)$ and $\varepsilon(v)$ as follows:

Proposition 17. *For a subproblem π,*

$$\begin{aligned}\mathbb{E}[\Delta(v)] &\ge \mathbb{E}[\Lambda(v)] - \mathbb{E}[|\varepsilon(v)|] \\ \mathbb{E}[\Delta(v)^2] &\le 2\cdot\mathbb{E}[\Lambda(v)^2],\end{aligned}$$

where expectations are with respect to the randomness of the algorithm and v chosen uniformly from $\mathcal{I}_\pi$.

Proof. The first inequality is immediate from the definition of $\varepsilon(v)$ and the triangle inequality. For the second, using the bound $|\varepsilon(v)| \le |\Lambda(v)|/3$ from Claim 16, we have:

$$\begin{aligned}\mathbb{E}[\Delta(v)^2] &= \mathbb{E}[(\Lambda(v)+\varepsilon(v))^2] \\ &\le \mathbb{E}[(|\Lambda(v)| + |\Lambda(v)|/3)^2] \\ &\le \frac{16}{9}\mathbb{E}[\Lambda(v)^2].\end{aligned}$$

□

In what follows, we will compute a lower bound on $\mathbb{E}[\Lambda(v)]$ and upper bounds on $\mathbb{E}[\Lambda(v)^2]$ and $\mathbb{E}[|\varepsilon(v)|]$. We will then be able to use Proposition 17 to complete the proof of Lemma 10.

Recall that, for the subproblem π, the algorithm chooses its rebuild window size w_π based on the random variable $K_\pi \in [0, k_{\max}]$. It will often be helpful to condition on $K_\pi = k$ for some k. Thus we use the following notation to refer to the values that variables take when $K_\pi = k$:

- w^k is the window size $m_\pi/(\alpha 2^k)$.
- t^k is the number of rebuild windows.
- The partition of $\mathcal{I}_\pi$ into windows is denoted $\mathcal{I}_1^k, \mathcal{I}_2^k, \ldots, \mathcal{I}_{t^k}^k$.
  (We have $t^k \le 2^k$, since $\frac{m_\pi}{\alpha} \ge |\mathcal{I}_\pi| = \sum_j |\mathcal{I}_j^k| > (t^k-1)w^k \ge (t^k-1)\frac{m_\pi}{\alpha 2^k}$.)
- D_j^k is an abbreviation for $D(\mathcal{I}_j^k)$.
- Q_j^k is the value used by the algorithm for Q_j. It is $\frac{m_\pi D_{j-1}^k}{\beta w^k}$ if j is even, and is 0 if j is odd.

Observe that the rebuild windows for $K_\pi = k+1$ are obtained by splitting each rebuild window for $K_\pi = k$ into two parts $\mathcal{I}_j^k = \mathcal{I}_{2j-1}^{k+1} \cup \mathcal{I}_{2j}^{k+1}$. The two sets $\mathcal{I}_{2j-1}^{k+1}$ and $\mathcal{I}_{2j}^{k+1}$ will both be of size w^{k+1}, unless $j = t^k$, in which case the sizes may be less than w^{k+1} or even 0; indeed the rebuild window $\mathcal{I}_{2t^k}^{k+1}$ may not even exist, in which case we treat it as empty.

The following two sums play a key role in the computation of $\mathbb{E}[\Lambda(v)]$ and $\mathbb{E}[\Lambda(v)^2]$.

$$\begin{aligned} S^k &= \sum_{j \le t^k} (D_j^k)^2. \\ R^k &= \sum_{\text{even } j \le t^k} D_{j-1}^k D_j^k. \end{aligned}$$

In the case that there is only one window (e.g. $k = 0$), we have $S^k = (D_\pi)^2$ and $R^k = 0$.

The following upper bound on S^k will be helpful later on in the proof, in particular, when we wish to bound $S^{k_{\max}}$. Recall, $\sigma_\pi = |\mathcal{I}_\pi|$.

Proposition 18. *For any $k \in [1, k_{\max}]$,*

$$S^k \le \frac{m_\pi \sigma_\pi}{\alpha 2^k}.$$

Proof. We have that

$$S^k = \sum_{j=1}^{t^k} (D_j^k)^2 \le \sum_{j=1}^{t^k} |\mathcal{I}_j^k|^2 \le \sum_{j=1}^{t^k} |\mathcal{I}_j^k| \frac{m_\pi}{\alpha 2^k} = \frac{\sigma_\pi m_\pi}{\alpha 2^k}.$$

□

We now turn to our bounds on $\mathbb{E}[\Lambda(v)]$, $\mathbb{E}[(\Lambda(v))^2]$, and $\mathbb{E}[|\varepsilon(v)|]$.

Lemma 19. *For any subproblem π, we have:*

$$\begin{aligned} \mathbb{E}[\Lambda(v)] &= \frac{\alpha(1-F_\pi)}{m_\pi \sigma_\pi \beta} \sum_{k=1}^{k_{\max}} \left(1 + \frac{k}{k_{\max}}\right) R^k - \frac{S^0}{\sigma_\pi m_\pi} \\ \mathbb{E}[(\Lambda(v))^2] &\le \frac{8\alpha}{m_\pi \sigma_\pi \beta^2} \sum_{k=0}^{k_{\max}} S^k \\ \mathbb{E}[|\varepsilon(v)|] &\le \frac{8\alpha}{m_\pi \sigma_\pi \beta^2} \sum_{k=0}^{k_{\max}} S^k, \end{aligned}$$

where the expectations are taken with respect to random $v \in \mathcal{I}_\pi$ and the choice of K_π.

The proof of this lemma follows from straightforward calculations:

Proof. To bound $\mathbb{E}[\Lambda(v)]$, we first analyze $\mathbb{E}[\Lambda(v)]$ conditioned on $K_\pi = k$. We write $\Lambda^k(v)$ (resp. $\varepsilon^k(v)$) for $\Lambda(v)$ (resp. $\varepsilon(v)$) conditioned on $K_\pi = k$. After this conditioning, the only remaining randomness is the uniform random choice of $v \in \mathcal{I}_\pi$. For each window $\mathcal{I}_j^k$, and for all $v \in \mathcal{I}_j^k$, we have by definition that $\Lambda^k(v) = D_\pi(v) \frac{2(1-F_\pi)Q_j^k - D_\pi}{m_\pi}$, so:

$$\sum_{v \in \mathcal{I}_j^k} \Lambda^k(v) = D_j^k \frac{2(1-F_\pi)Q_j^k - D_\pi}{m_\pi}.$$

Therefore, for v selected uniformly at random from $\mathcal{I}_\pi$, we have

$$\begin{aligned} \mathbb{E}[\Lambda^k(v)] &= \frac{1}{\sigma_\pi} \sum_{j=1}^{t^k} D_j^k \frac{2(1-F_\pi)Q_j^k - D_\pi}{m_\pi} \\ &= \frac{2(1-F_\pi)}{\sigma_\pi m_\pi} \sum_{j=1}^{t^k} D_j^k Q_j^k - \frac{(D_\pi)^2}{\sigma_\pi m_\pi} \\ &\qquad \text{(since } \textstyle\sum_{j=1}^{t^k} D_j^k = D_\pi\text{)} \\ &= \frac{2(1-F_\pi)}{\sigma_\pi m_\pi} \sum_{\text{even } j \le t^k} \frac{m_\pi}{\beta w_\pi^k} D_{j-1}^k D_j^k - \frac{(D_\pi)^2}{\sigma_\pi m_\pi} \\ &\qquad \text{(by definition of } Q_j^k\text{)} \\ &= \frac{(1-F_\pi)2^{k+1}\alpha}{\sigma_\pi m_\pi \beta} R^k - \frac{S^0}{\sigma_\pi m_\pi}. \\ &\qquad \text{(since } |w^k| = \tfrac{m_\pi}{\alpha 2^k} \text{ and } S^0 = (D_\pi)^2\text{)} \end{aligned}$$

Now averaging over the options for k, each of which occurs with probability $p_k = 2^{-(k+1)} \cdot (1 + k/k_{\max})$,

$$\begin{aligned} \mathbb{E}[\Lambda(v)] &= \sum_{k=0}^{k_{\max}} \frac{p_k(1-F_\pi)2^{k+1}\alpha}{\sigma_\pi m_\pi \beta} R^k - \left(\sum_k p_k\right) \cdot \frac{S^0}{\sigma_\pi m_\pi} \\ &= \sum_{k=1}^{k_{\max}} \frac{p_k(1-F_\pi)2^{k+1}\alpha}{\sigma_\pi m_\pi \beta} R^k - \frac{S^0}{\sigma_\pi m_\pi} \\ &\qquad \text{(since } R^0 = 0 \text{ and } \textstyle\sum p_k = 1\text{)} \\ &= \frac{\alpha(1-F_\pi)}{\sigma_\pi m_\pi \beta} \sum_{k=1}^{k_{\max}} \left(1 + \frac{k}{k_{\max}}\right) R^k - \frac{S^0}{\sigma_\pi m_\pi}, \end{aligned}$$

as claimed.

Next, we analyze $\mathbb{E}[(\Lambda(v))^2]$. As above we start by analyzing the conditional expectation $\mathbb{E}[(\Lambda^k(v))^2]$. For each window $\mathcal{I}_j^k$, and for each $v \in \mathcal{I}_j^k$ we have:

$$\begin{aligned} (\Lambda^k(v))^2 &= \frac{(2(1-F_\pi)Q_j^k - D_\pi)^2}{(m_\pi)^2} \\ &\le 2\frac{(2(1-F_\pi)Q_j^k)^2 + (D_\pi)^2}{(m_\pi)^2} \\ &\qquad \text{(by the inequality } (a+b)^2 \le 2a^2 + 2b^2\text{)} \\ &\le \frac{8(Q_j^k)^2 + 2(D_\pi)^2}{(m_\pi)^2}, \end{aligned} \tag{2}$$

where the final step uses $F_\pi \in [0,1]$. Therefore,

$$\sum_{v \in \mathcal{I}_j^k} (\Lambda^k(v))^2 \le |\mathcal{I}_j^k| \frac{8(Q_j^k)^2 + 2(D_\pi)^2}{(m_\pi)^2}.$$

It follows that, for a uniformly random $v \in \mathcal{I}_\pi$, we have

$$\begin{aligned}
\mathbb{E}[(\Lambda^k(v))^2] &= \frac{1}{\sigma_\pi}\sum_{j=1}^{t^k}|\mathcal{I}_j^k|\frac{8(Q_j^k)^2+2(D_\pi)^2}{(m_\pi)^2} \\
&= \frac{8}{\sigma_\pi(m_\pi)^2}\sum_{j=1}^{t^k}|\mathcal{I}_j^k|(Q_j^k)^2 + \frac{2(D_\pi)^2}{\sigma_\pi(m_\pi)^2}\sum_{j=1}^{t^k}|\mathcal{I}_j^k| \\
&= \frac{8}{\sigma_\pi(m_\pi)^2}\sum_{\text{even } j\le t^k}|\mathcal{I}_j^k|\frac{(D_{j-1}^k)^2(m_\pi)^2}{(w^k)^2\beta^2} + \frac{2(D_\pi)^2}{(m_\pi)^2} \\
&\qquad\qquad \text{(by defn of } Q_j^k \text{ and } \textstyle\sum|\mathcal{I}_j^k| = \sigma_\pi) \\
&\le \frac{8}{\sigma_\pi\beta^2 w^k}\sum_{\text{even } j\le t^k}(D_{j-1}^k)^2 + \frac{2(D_\pi)^2}{(m_\pi)^2} \\
&\qquad\qquad \text{(since } |\mathcal{I}_j^k| \le w^k) \\
&\le \frac{8\alpha 2^k}{m_\pi\sigma_\pi\beta^2}S^k + \frac{2(D_\pi)^2}{(m_\pi)^2} \qquad \text{(since } w^k = \tfrac{m_\pi}{\alpha 2^k}) \\
&\le \frac{8\alpha}{m_\pi\sigma_\pi\beta^2}\left(2^k S^k + \frac{1}{4}(D_\pi)^2\right). \\
&\qquad\qquad \text{(since } \alpha \ge \beta \text{ and thus } m_\pi \ge \sigma_\pi\alpha \ge \sigma_\pi\beta^2/\alpha)
\end{aligned}$$

Averaging over k we get,

$$\begin{aligned}
\mathbb{E}[(\Lambda(v))^2] &\le \frac{8\alpha}{\sigma_\pi m_\pi\beta^2}\left(\sum_{k=0}^{k_{\max}}p_k 2^k S^k + \sum_{k=0}^{k_{\max}}p_k\frac{(D_\pi)^2}{4}\right) \\
&\le \frac{8\alpha}{\sigma_\pi m_\pi\beta^2}\left(\sum_{k=1}^{k_{\max}}S^k + \frac{S^0}{2} + \frac{(D_\pi)^2}{4}\right) \\
&\qquad \text{(since } p_k \le 2^{-k}\text{, } p_0 \le 1/2 \text{ and } S^k \ge 0) \\
&\le \frac{8\alpha}{m_\pi\sigma_\pi\beta^2}\sum_{k=0}^{k_{\max}}S^k. \qquad \text{(since } S^0 = (D_\pi)^2)
\end{aligned}$$

Finally, we analyze $\mathbb{E}[|\varepsilon(v)|]$. For each window $\mathcal{I}_j^k$, and for each $v \in \mathcal{I}_j^k$, we have by Claim 16 that

$$\begin{aligned}
|\varepsilon^k(v)| &\le \frac{8|Q_j^k|}{3m_\pi}|\Lambda^k(v)| = \frac{8|Q_j^k|\cdot|2(1-F_\pi)Q_j^k - D_\pi|}{3(m_\pi)^2} \\
&\le 8\frac{2(Q_j^k)^2 + |Q_j^k||D_\pi|}{3(m_\pi)^2} \qquad \text{(since } F_\pi \in [0,1]) \\
&\le 8\frac{\frac{5}{2}(Q_j^k)^2 + \frac{1}{2}(D_\pi)^2}{3(m_\pi)^2} \\
&\qquad \text{(by the inequality } ab \le (a^2+b^2)/2) \\
&\le \frac{8(Q_j^k)^2 + 2(D_\pi)^2}{(m_\pi)^2},
\end{aligned}$$

which is, quite fortuitously (and partly by design), the same as the upper bound on $(\Lambda^k(v))^2$ shown in (2). Therefore, the exact same computation as for $\mathbb{E}[(\Lambda^k(v))^2]$ yields the claimed bound. □

We now come to the critical part of the proof. We have lower bounds on $\mathbb{E}[\Lambda(v)]$ in terms of the sums R^k and upper bounds on $\mathbb{E}[(\Lambda(v))^2]$ and $\mathbb{E}[|\varepsilon(v)|]$ in terms of the sums S^k. In order to complete the proof we need to relate the quantities R^k to the quantities S^k. This connection is provided by the following simple but crucial identity:

Proposition 20. *For any $h < \ell$,*

$$S^h - S^\ell = 2\sum_{k=h+1}^{\ell}R^k.$$

Proof. First we compute $S^k - S^{k+1}$. For each rebuild window $\mathcal{I}_j^k = \mathcal{I}_{2j-1}^{k+1} \cup \mathcal{I}_{2j}^{k+1}$, since $D_j^k = D_{2j-1}^{k+1} + D_{2j}^{k+1}$, we have

$$(D_j^k)^2 = (D_{2j-1}^{k+1})^2 + (D_{2j}^{k+1})^2 + 2D_{2j-1}^{k+1}D_{2j}^{k+1}.$$

Summing both sides over j yields $S^k = S^{k+1} + 2R^{k+1}$, so $S^k - S^{k+1} = 2R^{k+1}$. The desired equality follows by summing this equality for k from h to $\ell - 1$. □

We note that this lemma is the reason that, in the specification of the algorithm, Q_j^k is defined to be 0 for odd j. Had we applied the definition for even j also to odd j, then in the lower bound of $\mathbb{E}[\Delta]$, instead of $R^k = \sum_{\text{even } j} D_{j-1}^k D_j^k$ we would have to use $R^k = \sum_j D_{j-1}^k D_j^k$, where the sum is over all j, not just even j. This change to the definition of R^k would, in turn, cause the telescoping argument in Proposition 20 to fail, and we would no longer be able to relate the bound on $\mathbb{E}[\Lambda(v)]$ to the bounds on $\mathbb{E}[(\Lambda(v))^2]$ and $\mathbb{E}[|\varepsilon(v)|]$.

We now manipulate the bound on $\mathbb{E}[\Delta(v)]$ to finish the proof of the lemma. A key step is given by:

$$\begin{aligned}
\sum_{k=1}^{k_{\max}}kR^k &= \sum_{k=1}^{k_{\max}}\sum_{h=k}^{k_{\max}}R^h = \sum_{k=0}^{k_{\max}-1}\frac{S^k - S^{k_{\max}}}{2} \\
&= \frac{1}{2}\sum_{k=0}^{k_{\max}-1}S^k - \frac{k_{\max}}{2}S^{k_{\max}},
\end{aligned}$$

where the second equality uses Proposition 20. As we will see below, this identity is the reason why p_k was defined to be $2^{-(k+1)}\cdot(1 + k/k_{\max})$ rather than, say, 2^{-k}.

We now lower bound $\mathbb{E}[\Lambda(v)]$ by

$$\begin{aligned}
\mathbb{E}[\Lambda(v)] &= \frac{\alpha(1-F_\pi)}{\sigma_\pi m_\pi \beta}\sum_{k=1}^{k_{\max}}\left(1+\frac{k}{k_{\max}}\right)R^k - \frac{S^0}{\sigma_\pi m_\pi} && \text{(Prop. 18)}\\
&= \frac{\alpha(1-F_\pi)}{\sigma_\pi m_\pi \beta}\left(\sum_{k=1}^{k_{\max}} R^k + \frac{1}{k_{\max}}\sum_{k=1}^{k_{\max}} kR^k\right) - \frac{S^0}{\sigma_\pi m_\pi}\\
&= \frac{\alpha(1-F_\pi)}{\sigma_\pi m_\pi \beta}\left(\frac{S^0 - S^{k_{\max}}}{2} + \frac{1}{2k_{\max}}\sum_{k=0}^{k_{\max}-1} S^k\right.\\
&\quad \left. - \frac{S^{k_{\max}}}{2}\right) - \frac{S^0}{\sigma_\pi m_\pi} && \text{(Prop. 20 and (Prop. 18))}\\
&= \frac{\alpha(1-F_\pi)}{\sigma_\pi m_\pi \beta}\left(\frac{S^0 - S^{k_{\max}}}{2} + \frac{1}{2k_{\max}}\sum_{k=0}^{k_{\max}} S^k\right.\\
&\quad \left. - \frac{S^{k_{\max}}}{2k_{\max}} - \frac{S^{k_{\max}}}{2}\right) - \frac{S^0}{\sigma_\pi m_\pi}\\
&\geq \frac{\alpha(1-F_\pi)}{\sigma_\pi m_\pi \beta}\left(\frac{S^0}{2} + \frac{1}{2k_{\max}}\sum_{k=0}^{k_{\max}} S^k\right.\\
&\quad \left. - \frac{3}{2}S^{k_{\max}}\right) - \frac{S^0}{\sigma_\pi m_\pi}\\
&\geq \frac{\alpha}{2\sigma_\pi m_\pi \beta}\left(\frac{S^0}{2} + \frac{1}{2k_{\max}}\sum_{k=0}^{k_{\max}} S^k\right)\\
&\quad - \frac{3\alpha}{2\sigma_\pi m_\pi \beta}S^{k_{\max}} - \frac{S^0}{\sigma_\pi m_\pi} && \text{(since } F_\pi \in [0,1/2],\ S^k \geq 0)\\
&\geq \frac{\alpha}{2\sigma_\pi m_\pi \beta}\left(\frac{S^0}{2} + \frac{1}{2k_{\max}}\sum_{k=0}^{k_{\max}} S^k\right)\\
&\quad - \frac{3\alpha}{2\sigma_\pi m_\pi \beta}S^{k_{\max}} - \frac{\alpha S^0}{8\beta\sigma_\pi m_\pi} && \text{(since } \alpha \geq 8\beta)\\
&\geq \frac{\alpha}{2\sigma_\pi m_\pi \beta}\left(\frac{1}{2k_{\max}}\sum_{k=0}^{k_{\max}} S^k\right) - \frac{3\alpha}{2\sigma_\pi m_\pi \beta}S^{k_{\max}} && \text{(since } S^0 \geq 0)\\
&\geq \frac{\alpha}{2\sigma_\pi m_\pi \beta}\left(\frac{1}{2k_{\max}}\sum_{k=0}^{k_{\max}} S^k\right) - \frac{3}{2\beta}2^{-k_{\max}} && \text{(Prop. 18)}\\
&\geq \frac{\alpha}{4\sigma_\pi m_\pi \beta k_{\max}}\sum_{k=0}^{k_{\max}} S^k - 2^{-k_{\max}}.
\end{aligned}$$

We are now ready to complete the proof of Lemma 10. We have

$$\begin{aligned}
\mathbb{E}[\Delta(v)] &\geq \mathbb{E}[\Lambda(v)] - \mathbb{E}[|\varepsilon(v)|] && \text{(Prop. 17)}\\
&\geq \frac{\alpha}{4\sigma_\pi m_\pi \beta k_{\max}}\sum_{k=0}^{k_{\max}} S^k - 2^{-k_{\max}} - \frac{8\alpha}{m_\pi \sigma_\pi \beta^2}\sum_{k=0}^{k_{\max}} S^k\\
&\qquad \text{(by (Prop. 17) and Lemma 19)}\\
&= \left(\frac{\beta}{32k_{\max}} - 1\right)\frac{8\alpha}{m_\pi \sigma_\pi \beta^2}\sum_{k=0}^{k_{\max}} S^k - 2^{-k_{\max}}\\
&\geq \left(\frac{\beta}{32k_{\max}} - 1\right)\mathbb{E}[\Lambda(v)^2] - 2^{-k_{\max}} && \text{(Lemma 19)}\\
&\geq \left(\frac{\beta}{32k_{\max}} - 1\right)\mathbb{E}[\Delta(v)^2]/2 - 2^{-k_{\max}} && \text{(Prop. 17)}\\
&\geq \left(\frac{\beta}{100k_{\max}}\right)\mathbb{E}[\Delta(v)^2] - 2^{-k_{\max}},\\
&\qquad \text{(since } \beta \geq 1000k_{\max})
\end{aligned}$$

which completes the proof of Lemma 10.

VII. Proof of Claim 14

In this section, we prove Claim 14, restated below:

Claim 14. *Let $k_{\max} = 2\log\log n$, and let $\beta = C_\beta(\log\log n)^2$ for some sufficiently large positive constant C_β. Let $X_1, X_2, \ldots, X_r$ be random variables with $r \leq 2\log n$ such that for $i \in [1,r]$:*

1) $|X_i| \leq 3/\beta$;
2) $\mathbb{E}[X_i^2 \mid X_1, \ldots, X_{i-1}] \leq \frac{100k_{\max}}{\beta}\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}] + 2^{-k_{\max}}$.

Then, $\Pr[\sum_i X_i < -0.2] \leq O\left(1/\log^3 n\right)$.

As notation, for a sequence $Z = Z_1, \ldots, Z_s$ of random variables, define $\mu_1(Z), \ldots, \mu_s(Z)$ and $V_1(Z), \ldots, V_s(Z)$ by:

$$\begin{aligned}
\mu_j(Z) &= \mathbb{E}[Z_j \mid Z_1, \ldots, Z_{j-1}]\\
V_j(Z) &= \mathrm{Var}[Z_j \mid Z_1, \ldots, Z_{j-1}].
\end{aligned}$$

Also define $\mu(Z) = \sum \mu_j(Z)$, $V(Z) = \sum_j V_j(Z)$ and $\Sigma(Z) = \sum_j Z_j$. Note that $\mu_j(Z)$ and $V_j(Z)$ are random variables that are determined by the values of $Z_1, \ldots, Z_{j-1}$. We will prove:

Lemma 21. *Let $Z = Z_1, Z_2, \ldots, Z_s$ be random variables and suppose that A, B, and C are positive real numbers with $4A \leq B \leq 1$ such that, with probability 1, we have for all $j \in [1,s]$ that:*

1) $|Z_j| \leq A$;
2) $V_j(Z) \leq B\mu_j(Z) + C$.

Then for $q \geq \max(2sC/B, 16B)$, $\Pr[\Sigma(Z) < -q] \leq 3e^{-q/16B}$.

Supposing n is sufficiently large, Claim 14 follows from the lemma with $q = 0.2$, $s = r \leq 2\log n$, $A = \frac{3}{\beta}$, $B = 100k_{\max}/\beta$ and $C = 2^{-k_{\max}} = \frac{1}{\log^2 n}$. The hypothesis $4A \leq B \leq 1$ is satisfied since $\beta = \Omega((\log\log n)^2)$ and $k_{\max} = 2\log\log n$, and the hypothesis $q \geq \max(2sC/B, 16B)$ is satisfied since $2sC/B = O((\log\log n)/\log n)$ and $16B =$

$O(1/\log\log n)$ so both are smaller than $q = 0.2$. The resulting probability upper bound is $3e^{-.2\beta/1600k_{\max}} = 3e^{-C_\beta \log\log n/16000} \le 3(\log n)^{-C_\beta/16000}$ and taking C_β large enough this is less than $1/\log^3 n$, as required.

Recall that a martingale difference sequence is a sequence $Y = Y_1, Y_2, \ldots, Y_s$ of random variables such that we have $\mathbb{E}[|Y_i|] < \infty$ for all i and for any outcomes of $Y_1, \ldots, Y_{j-1}$, $\mu_j(Y) = \mathbb{E}[Y_j \mid Y_1, \ldots, Y_{j-1}] = 0$; this is equivalent to the condition that the sequence with terms $X_j = \sum_{i=1}^{j} Y_i$ is a martingale. Lemma 21 will be deduced from the following theorem of Freedman:

Theorem 22 (Proposition 2.1 of [34])**.** *Let $s \in \mathbb{N}$ and let $Y = Y_1, \ldots, Y_s$ be a martingale difference sequence. Suppose D and v are positive real numbers such that, with probability 1:*

1) *$|Y_j| \le D$ for all j;*
2) *$\sum_j V_j(Y) \le v$.*

Then, for $\ell > 0$, $\Pr[\Sigma(Y) > \ell] \le e^{-\ell^2/2(v+D\ell)}$.

A natural approach to proving Lemma 21 is to define the martingale difference sequence X by $X_j = \mu_j(Z) - Z_j$. This would imply $V_j(X) = V_j(Z)$ for all j, and it would then suffice to upper bound $\Pr[\Sigma(Z) < -q] = \Pr[\Sigma(X) > \mu(Z) + q]$, which we might hope to do with Theorem 22. However, the theorem cannot be applied directly because the upper bound on $V_j(X)$ of $B\mu_j(Z) + sC$ implied by Hypothesis (2) of Lemma 21 is itself a random variable, and the quantity $\mu(Z) + q$ to which $\Sigma(X)$ is compared in the conclusion is also a random variable, while Theorem 22 requires both to be fixed quantities. To get around this we first prove:

Proposition 23. *With Z satisfying the hypotheses of Lemma 21, let $\rho_2 > \rho_1$ be fixed real numbers. For $q \in (-\rho_1, 2\rho_2 - \rho_1 - \frac{2sC}{B}]$,*

$$\Pr[\Sigma(Z) < -q \text{ and } \mu(Z) \in [\rho_1, \rho_2]] \le e^{-(\rho_1+q)^2/(4B\rho_2)}.$$

Proof. Assume that Z satisfies the hypotheses of Lemma 21 and $q \in (-\rho_1, 2\rho_2 - \rho_1 - \frac{2sC}{B}]$. We define sequences Z' and X' which are modified versions of Z and X. We'll be able to apply Theorem 22 to X'. Let $C_j(Z)$ denote the event:

$$\sum_{i=1}^{j} V_i(Z) \le B\rho_2 + sC,$$

and define $Z'_1, \ldots, Z'_s$ to be given by:

$$Z'_j = \begin{cases} Z_j & \text{if } C_j(Z) \text{ holds,} \\ 0 & \text{otherwise.} \end{cases}$$

Claim 24. *Z' has the following properties:*

1) *If $C_j(Z)$ holds then $Z'_1, \ldots, Z'_j = Z_1, \ldots, Z_j$ and $V_1(Z'), \ldots, V_j(Z') = V_1(Z), \ldots, V_j(Z)$.*
2) *$V(Z') \le \rho_2 B + sC$.*
3) *If $\mu(Z) \le \rho_2$ then $C_j(Z)$ holds for all $j \in [1, s]$ and $Z' = Z$.*

Proof. For the first claim, note that if $C_j(Z)$ holds, then conditioned on $Z_1, \ldots, Z_{j-1}$, Z'_j and Z_j are equal as random variables and consequently $V_j(Z') = V_j(Z)$. Also, if $C_j(Z)$ holds then $C_i(Z)$ holds for all $i \le j$, since $V_i(Z) \ge 0$ for all i. Therefore if $C_j(Z)$ holds $Z'_1, \ldots, Z'_j = Z_1, \ldots, Z_j$ and $V_1(Z'), \ldots, V_j(Z') = V_1(Z), \ldots, V_j(Z)$.

For the second claim, let h be the least index for which $C_h(Z)$ fails, setting $h = s + 1$ if $C_j(Z)$ holds for all $j \le s$. Then $C_i(Z)$ fails for $i \ge h$, and so conditioned on $Z_1, \ldots, Z_{h-1}$, for $i \ge h$, Z_i is identically 0, and so $V_i(Z) = 0$. Therefore by part 1 of the claim:

$$\begin{aligned} V(Z') &\le \sum_{i \le s} V_i(Z') = \sum_{i \le h-1} V_i(Z') \\ &= \sum_{i \le h-1} V_i(Z) \le \rho_2 B + sC, \end{aligned}$$

since $C_{h-1}(Z)$ holds.

For the third claim, if $\mu(Z) \le \rho_2$ then by Hypothesis (2) of Lemma 21, $\sum_{i \le s} V_i(Z) \le \sum_{i=1}^{s} (B\mu_i(Z) + C) = B\mu(Z) + sC \le B\rho_2 + sC$ and so condition $C_s(Z)$ holds, which implies by the first part of the claim that $C_1(Z), \ldots, C_s(Z)$ all hold and that $Z' = Z$. □

Now define the martingale difference sequence X' by $X'_j = \mu_j(Z') - Z'_j$. Then:

$$\begin{aligned} &\Pr[\Sigma(Z) < -q \text{ and } \mu(Z) \in [\rho_1, \rho_2]] \\ &\quad \le \Pr[\Sigma(Z') < -q \text{ and } \mu(Z') \in [\rho_1, \rho_2]] \\ &\qquad\qquad \text{(Part 3 of Claim 24)} \\ &\quad = \Pr[\Sigma(X') > \mu(Z') + q \text{ and } \mu(Z') \in [\rho_1, \rho_2]] \\ &\quad \le \Pr[\Sigma(X') > \rho_1 + q]. \end{aligned}$$

We claim that the hypotheses of Theorem 22 are satisfied with $Y = X'$, $D = 2A$, and $v = B\rho_2 + sC$. For the first hypothesis of Theorem 22, $|Z_j| \le A$ which implies $|\mu_j(Z)| \le A$ and so $|X_j| \le |\mu_j(Z)| + |Z_j| \le 2A$. For the second hypothesis, we have

$$\begin{aligned} V_j(X') &= \mathbb{E}[(\mu_j(X') - X'_j)^2 \mid X'_1, \ldots, X'_{j-1}] \\ &= \mathbb{E}[(\mu_j(Z') - Z'_j)^2 \mid X'_1, \ldots, X'_{j-1}] \\ &= \mathbb{E}[(\mu_j(Z') - Z'_j)^2 \mid Z'_1, \ldots, Z'_{j-1}] \\ &= V_j(Z') \le B\rho_2 + sC, \qquad \text{(Part 2 of Claim 24)} \end{aligned}$$

where the 3rd equality holds because the sequences $Z'_1, \ldots, Z'_{j-1}$ and $X'_1, \ldots, X'_{j-1}$ determine each other.

Since $\ell = \rho_1 + q > 0$ (by the hypothesis on q in the current proposition), we can apply Theorem 22 to get:

$$\begin{aligned} &\Pr\big[\Sigma(Z) < -q \text{ and } \mu(Z) \in [\rho_1, \rho_2]\big] \le \Pr\big[\Sigma(X') > \rho_1 + q\big] \\ &\quad \le \exp\left(-\frac{(\rho_1+q)^2}{2(B\rho_2 + sC + 2A(q+\rho_1))}\right) \quad \text{(Theorem 22)} \\ &\quad \le \exp\left(-\frac{(\rho_1+q)^2}{2B(\rho_2 + sC/B + (q+\rho_1)/2)}\right) \\ &\qquad\qquad \text{(since } 4A \le B\text{)} \\ &\quad \le \exp\left(-\frac{(\rho_1+q)^2}{4B\rho_2}\right), \quad \text{(since } q \le 2\rho_2 - \rho_1 - 2sC/B\text{)} \end{aligned}$$

as required. □

We can now finish the proof of Lemma 21.

Proof of Lemma 21. Hypothesis 2 of the lemma implies $V(Z) \le B\mu(Z) + sC$ and therefore $\mu(Z) \ge -sC/B$, since variance is nonnegative. Cover the interval $[-sC/B, \infty]$ by $[-sC/B, q] \cup \bigcup_{i\ge 1}[2^{i-1}q, 2^i q]$. For each of these intervals we want to apply Proposition 23 with $[\rho_1, \rho_2]$ set to that interval. For $\rho_1 = -sC/B$ and $\rho_2 = q$, the hypothesis $q > -\rho_1$ holds because, by assumption, $q \ge 2sC/B > sC/B$; and the hypothesis $q \le 2\rho_2 - \rho_1 - 2sC/B = 2q + sC/B - 2sC/B$ holds because $2sC/B \le q$ by assumption. For the interval $[\rho_1, \rho_2] = [2^{i-1}q, 2^i q]$, the hypotheses of Proposition 23 hold because $q > 0 > -\rho_1$ and because $2\rho_2 - \rho_1 - 2sC/B \ge 2\rho_2 - \rho_1 - q = (2^{i+1} - 2^{i-1} - 1)q \ge q$. So, applying Proposition 23 to each interval, we can conclude that:

$$
\begin{aligned}
&\Pr[\Sigma(Z) < -q] \le \Pr\left[\Sigma(Z) < -q \text{ and } \mu(Z) \in [-sC/B, q]\right] \\
&+ \sum_{i\ge 1} \Pr\left[\Sigma(Z) < -q \text{ and } \mu(Z) \in [q2^{i-1}, q2^i]\right] \\
&\le \exp\left(-\frac{(q - sC/B)^2}{4Bq}\right) + \sum_{i\ge 1} \exp\left(-\frac{(q2^{i-1} + q)^2}{4 \cdot 2^i qB}\right) \\
&\qquad\qquad \text{(Prop. 23)} \\
&\le \exp\left(-\frac{(q - sC/B)^2}{4Bq}\right) + \sum_{i\ge 1} \exp\left(-\frac{(q2^{i-1})^2}{4 \cdot 2^i qB}\right) \\
&\le \exp\left(-\frac{(q/2)^2}{4Bq}\right) + \sum_{i\ge 1} \exp\left(-\frac{2^i q}{16B}\right) \qquad (q \ge 2sC/B) \\
&\le \exp\left(-\frac{q}{16B}\right) + \sum_{j\ge 1} \exp\left(-\frac{jq}{16B}\right) \\
&\le \exp\left(-\frac{q}{16B}\right) + \sum_{j\ge 1} \exp\left(-\frac{q}{16B} - (j-1)\right) \quad (q > 16B) \\
&\le \exp\left(-\frac{q}{16B}\right) + \exp\left(-\frac{q}{16B}\right) \sum_{j\ge 0} e^{-j} \\
&\le 3\exp\left(-\frac{q}{16B}\right). \qquad (q > 16B)
\end{aligned}
$$

□

VIII. Related work

In this section, we give a detailed discussion of related work on the list-labeling problem. To distinguish the different regimes in which one can study the problem, we will refer to $m = (1+\Theta(1))n$ as the ***linear regime***, to $m = (1+o(1))n$ as the ***dense regime***, to $m = n^{1+\Theta(1)}$ as the ***polynomial regime***, and to $m = n^{\omega(1)}$ as the ***super-polynomial regime***. Although list labeling was originally formulated in the linear regime [39], the other regimes end up also being useful in many settings.

Independent Formulations. There have been many independent formulations of list labeling under a variety of different names. The problem encapsulates several other scenarios beyond the maintenance of elements from an ordered universe in a sorted array. Instead of elements coming from an ordered universe, one can think of elements coming from an unordered universe whose rank is determined relative to the elements in the current set at the moment of their insertion. This was the original formulation of Itai, Konheim, and Rodeh [39] who devised a sparse table scheme to implement priority queues. Willard [63] independently studied the ***file-maintenance problem*** for maintaining order in a file as records are inserted and deleted. Even more abstractly, one does not have to think of an array but of a linked list of items that are assigned labels from $\{1, \ldots, m\}$, and the natural order of the labels should correspond to the relative order of the items. This view becomes relevant when m is large relative to n (the polynomial and super-polynomial regimes), and it was taken by Dietz [26], Tsakalidis [59], and Dietz and Sleator [29], and Bender et al. [9] who (in some cases implicitly) applied both the polynomial and exponential regimes to the so-called ***order-maintenance problem***, which studies the abstract data-structural problem of maintaining ordered items in a linked list. A problem similar to list labeling (in the polynomial regime) was studied in the context of balanced binary search trees by Andersson [3] and Andersson and Lai [4], as well as by Galperin and Rivest [36] under the name ***scapegoat trees***. Raman [56] formulated the problem in the linear regime in the context of building locality preserving dictionaries. Hofri and Konheim [37] studied a sparse table structure that supports search, insert and deletion by keys in the linear and dense regimes. Devanny, Fineman, Goodrich, and Kopelowitz [25] studied the ***online house numbering problem***, a version of list labeling where the goal is to minimize the maximum number of times that any one element gets moved (i.e., has its label changed).

Upper bounds. The most studied setting of the list-labeling problem is the linear regime, in which $m = (1 + \Theta(1))n$. Itai, Konheim, and Rodeh [39], showed an upper bound of $O(\log^2 n)$ amortized cost per operation. This was later deamortized to $O(\log^2 n)$ worst-case cost per operation by Willard [64]–[66]. Simplified algorithms for these upper bounds were provided by Katriel [40], and Itai and Katriel [38] for the amortized bound and Bender, Cole, Demaine, Farach-Colton, and Zito [9] and Bender, Fineman, Gilbert, Kopelowitz, and Montes [16] for the worst-case bound. The upper bound of $O(\log^2 n)$ stood unimproved for four decades until Bender, Conway, Farach-Colton, Komlós, Kuszmaul, and Wein [10] showed an amortized $O(\log^{3/2} n)$ expected cost algorithm. The same paper also proved an upper bound of $O(\log^{3/2} n/(\log^{1/2} \tau))$ for the *sparse* regime where $m = \tau n$ for $\tau \le n^{o(1)}$. The algorithm by Bender et al. [10] is history independent, and builds on techniques developed by an earlier $O(\log^2 n)$ expected-cost history-independent solution due to Bender, Berry, Johnson, Kroeger, McCauley, Phillips, Simon, Singh, and Zage [7].

In the polynomial regime, where $m = n^{1+\Theta(1)}$, upper bounds of $O(\log n)$ have been shown [3], [36], [42]. In the superpolynomial regime, where $m = n^{\omega(1)}$, Babka, Bulánek,

Čunát, Koucký, and Saks [6] gave an algorithm with amortized $O(\log n/\log\log m)$ cost when $m = \Omega(2^{\log^k n})$, which implies a constant amortized cost algorithm in the pseudo-exponential regime where $m = 2^{n^{\Omega(1)}}$.

For the regime where $m = n$, Andersson and Lai [4], Zhang [67], and Bird and Sadnicki [20] showed an $O(n\log^3 n)$ upper bound for filling an array from empty to full (i.e., an insertion-only workload). This bound was subsequently improved to $O(n\log^{2.5} n)$ by Bender et al. [15], and then to $\widetilde{O}(\log^2 n)$ in the current paper (Corollary 4).

Finally, several papers (in the linear regime) have also studied forms of beyond-worst-case analysis. Bender and Hu [19] provided an *adaptive* solution, which has $O(\log n)$ amortized expected cost on certain common classes of instances while maintaining $O(\log^2 n)$ amortized worst-case cost. McCauley, Moseley, Niaparast, and Singh [47] study a setting in which one has access to a (possibly erroneous) prediction oracle, and give a solution that is parameterized by the oracle's error.

Lower bounds. In the linear regime, Dietz and Zhang [30] proved a lower bound of $\Omega(\log^2 n)$ for ***smooth*** algorithms, which are restricted to rearrangements that spread a set of elements evenly across some subarray. Bulánek, Koucký, and Saks [23] later showed an $\Omega(\log^2 n)$ lower bound for deterministic algorithms. Bender, Conway, Farach-Colton, Komlós, Kuszmaul, and Wein [10] showed a lower bound of $\Omega(\log^{3/2} n)$ for history-independent algorithms, where the notion of history independence that they used is that the set of slots occupied, at any given moment, should reveal nothing about the input sequence beyond the current number of elements.

In the polynomial regime, Dietz and Zhang [30] proved a lower bound of $\Omega(\log n)$ for smooth algorithms. Dietz, Seiferas, and Zhang [28], and a later simplification by Babka, Bulánek, Čunát, Koucký, and Saks [5], extended this to a lower bound of $\Omega(\log n)$ for general deterministic algorithms. Finally, Bulánek, Koucký, and Saks [24] proved an $\Omega(\log n)$ lower bound for general (including randomized) algorithms. This is also by extension the best known lower bound for randomized algorithms in the linear regime.

In other regimes, Bulánek, Koucký, and Saks [23] showed a deterministic lower bound of $\Omega(n\log^3 n)$ for n insertions into an initially empty array of size $m = n + n^{1-\varepsilon}$. In the superpolynomial regime, Babka, Bulánek, Čunát, Koucký, and Saks [6] gave a deterministic lower bound of $\Omega\left(\frac{\log n}{\log\log m - \log\log n}\right)$ for m between n^{1+C} and 2^n, which reduces to a bound of $\Omega(\log n)$ for $m = n^{1+C}$.

Other Theory Applications. In addition to the applications discussed above, list labeling has found many algorithmic applications in areas such as cache-oblivious data structures and computational geometry. Many of these applications use ***packed-memory arrays***, which are list-labeling solutions in the linear (and dense) regimes with the added requirement that there are never more than $O(1)$ free slots in a row between consecutive elements. Various works show bounds of $O(\delta^{-1}\log^2 n)$ for this version of the problem [11], [12], [17]. Improvements to list labeling in both [10] and in this paper imply analogous improvements for packed-memory arrays (with our result bringing the bound down to $\widetilde{O}(\delta^{-1}\log n)$). These improvements, in turn, imply immediate improvements to the bounds in many of the applications below.

Packed-memory arrays have found extensive applications to the design of efficient cache-oblivious data structures. Bender, Demaine, and Farach-Colton [12] used the packed-memory array to construct a ***cache-oblivious B-tree***. Simplified algorithms for cache-oblivious B-trees were provided by Brodal, Fagerberg, and Jacob [22] and Bender, Duan, Iacono, and Wu [13]. Bender, Fineman, Gilbert, and Kuszmaul [17] presented ***concurrent*** cache-oblivious B-trees and Bender, Farach-Colton, and Kuszmaul [14] presented ***cache-oblivious string B-trees***. All of these data structures use packed memory arrays. In each case, the list-labeling improvements in the current paper improve the range of parameters for which the above constructions are optimal, so that the restriction on the block-size B goes from $B \geq \widetilde{\Omega}(\log\sqrt{\log n})$ (using the list-labeling solution from [10]) to $B \geq \operatorname{poly}\log\log n$.

List labeling has also found applications in data structures for computational geometry problems. Nekrich used the technique to design data structures for orthogonal range reporting [51], [52] (these use the polynomial regime), the stabbing-max problem [54] (this uses the linear regime), and a related problem of searching a dynamic catalog on a tree [53] (this uses the linear regime). Similarly, Mortensen [49] used the technique (in the linear regime) for the orthogonal range and dynamic line segment intersection reporting problems.

Additionally, Fagerberg, Hammer, and Meyer [33] use list labeling (implicitly, and in the linear regime) for a rebalancing scheme that maintains optimal height in a balanced B-tree. And Kopelowitz [42] uses a generalization of the list-labeling problem (in the polynomial regime) to design an efficient algorithm for constructing suffix trees in an online fashion.

On the lower-bound side, Emek and Korman [32] show how to make use of lower bounds for list labeling to derive lower bounds for the *distributed controller problem*, which is a resource allocation problem in the distributed setting [1].

Practical Applications. Additionally, many practical applications make use of packed-memory arrays. Durand, Raffin and Faure [31] use a packed-memory array in particle movement simulations to maintain sorted order for efficient searches. Khayyat, Lucia, Singh, Ouzzani, Papotti, Quiané-Ruiz, Tang and Kalnis [41] handle dynamic database updates in inequality join algorithms using packed-memory arrays. Toss, Pahins, Raffin and Comba [58] constructed a ***packed-memory quadtree***, which supports large streaming spatiotemporal datasets. De Leo and Boncz [45] implement a ***rewired memory array***, which improves the practical performance of packed-memory arrays. ***Parallel*** packed-memory arrays have been implemented in several works [44], [46], [55], [60]–[62] to store dynamic graphs with fast updates and range queries.

IX. Conclusion and Open Questions

We conclude the paper by discussing some of the central remaining open questions.

The biggest open question is whether it is possible to achieve an expected cost bound of $O(\log n)$, or whether one necessarily needs to incur extra $\operatorname{poly}\log\log n$ factors. This appears to be a quite difficult question, but it is also an important one, as the answer may well determine the *practical* impact of the list labeling in the future. If an $O(\log n)$ solution turns out to be possible, then it is likely that list labeling (and, specifically, packed memory arrays) could serve as practical and more cache-friendly alternatives to binary search trees in many settings.

Another important open question has to do with high-probability guarantees. It is not yet known whether one can construct a randomized list-labeling algorithm that achieves $o(\log^2 n)$ cost per operation, not just in *expectation*, but also with high probability (i.e., probability $1 - 1/\operatorname{poly}(n)$). This question is especially important for applications that are latency sensitive.

Finally, our results suggest that it is time to revisit a classic open problem due to Naor and Teague [50]: Does there exist a problem for which weakly history-independent algorithms are provably worse (in terms of write cost) than their non-history-independent alternatives? Our results suggest that list labeling is likely such a problem. The main challenge here is in constructing a lower bound: Can one prove that weakly-history-independent list-labeling solutions must incur $\Omega(\log^{3/2} n)$ expected cost? The lower bound from [10] offers a natural starting point: they prove that, if the probability distribution governing the data structure's state is always fully determined by the the array size and the current *number* of elements, then the lower bound of $\Omega(\log^{3/2} n)$ holds. In contrast, a lower bound for all weakly-history-independent solutions would need to make a weaker assumption: that the probability distribution governing the data structure's state is always fully determined by the the array size and the current *set* of elements. This distinction renders many of the arguments in the known lower bound inoperable, and it appears that a stronger lower bound would require significant additional ideas beyond the techniques in [10].

Acknowledgments

This work was supported by NSF grants CCF-2106999, CCF-2118620, CNS-1938180, CCF-2118832, CCF-2106827, CNS-1938709, and CCF-2247577.

Michael Bender was supported in part by the John L. Hennessy Chaired Professorship.

Martín Farach-Colton was supported in part by the Leonard J. Shustek professorship.

Hanna Komlós was partially supported by the Graduate Fellowships for STEM Diversity.

Michal Koucký carried out part of the work during an extended visit to DIMACS, with support from the National Science Foundation under grant number CCF-1836666 and from The Thomas C. and Marie M. Murray Distinguished Visiting Professorship in the Field of Computer Science at Rutgers University. He was also partially supported by the Grant Agency of the Czech Republic under the grant agreement no. 24-10306S. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 823748 (H2020-MSCA-RISE project CoSP).

William Kuszmaul was partially supported by a Harvard Rabin Postdoctoral Fellowship and by a Harvard FODSI fellowship under NSF grant DMS-2023528.

References

[1] Yehuda Afek, Baruch Awerbuch, Serge A. Plotkin, and Michael E. Saks. Local management of a global resource in a communication network. *J. ACM*, 43(1):1–19, 1996.

[2] Noga Alon, Phillip G Bradford, and Rudolf Fleischer. Matching nuts and bolts faster. *Information processing letters*, 59(3):123–127, 1996.

[3] Arne Andersson. Improving partial rebuilding by using simple balance criteria. In *Proc. Workshop on Algorithms and Data Structures (WADS)*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989.

[4] Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In John R. Gilbert and Rolf G. Karlsson, editors, *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121, July 1990.

[5] Martin Babka, Jan Bulánek, Vladimír Čunát, Michal Koucký, and Michael E. Saks. On online labeling with polynomially many labels. In *ESA*, volume 7501 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 2012.

[6] Martin Babka, Jan Bulánek, Vladimír Čunát, Michal Koucký, and Michael E. Saks. On online labeling with large label set. *SIAM J. Discret. Math.*, 33(3):1175–1193, 2019.

[7] Michael A. Bender, Jon Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In *Proc. 35th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 289–302, June 2016.

[8] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151, 2002.

[9] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 152–164. Springer, 2002.

[10] Michael A. Bender, Alex Conway, Martin Farach-Colton, Hanna Komlós, William Kuszmaul, and Nicole Wein. Online list labeling: Breaking the $\log^2$n barrier. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 980–990. IEEE, 2022.

[11] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 399–409. IEEE Computer Society, 2000.

[12] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[13] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

[14] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 233–242. ACM, 2006.

[15] Michael A Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. On the optimal time/space tradeoff for hash tables. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1284–1297, 2022.

[16] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. File maintenance: When in doubt, change the layout! In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1503–1522, January 2017.

[17] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, 2005.

[18] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 20–29, 2006.

[19] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. *ACM Trans. Database Syst.*, 32(4):26:1–26:43, November 2007.

[20] Richard S. Bird and Stefan Sadnicki. Minimal on-line labelling. *Inf. Process. Lett.*, 101(1):41–45, 2007.

[21] Gerth Stølting Brodal. A survey on priority queues. In *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 150–163. Springer, 2013.

[22] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[23] Jan Bulánek, Michal Koucký, and Michael Saks. Tight lower bounds for the online labeling problem. In *Proc. 44th Annual Symposium on Theory of Computing (STOC)*, pages 1185–1198, 2012.

[24] Jan Bulánek, Michal Koucký, and Michael E. Saks. On randomized online labeling with polynomially many labels. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 7965 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2013.

[25] William E Devanny, Jeremy T Fineman, Michael T Goodrich, and Tsvi Kopelowitz. The online house numbering problem: Min-max online list labeling. In *Proc. 25th European Symposium on Algorithms (ESA)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[26] Paul F. Dietz. Maintaining order in a linked list. In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC)*, pages 122–127, New York, NY, USA, 1982.

[27] Paul F Dietz, Joel I Seiferas, and Ju Zhang. A tight lower bound for online monotonic list labeling. In *Scandinavian Workshop on Algorithm Theory*, pages 131–142. Springer, 1994.

[28] Paul F Dietz, Joel I Seiferas, and Ju Zhang. A tight lower bound for online monotonic list labeling. *SIAM Journal on Discrete Mathematics*, 18(3):626–637, 2004.

[29] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Annual Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[30] Paul F Dietz and Ju Zhang. Lower bounds for monotonic list labeling. In *Scandinavian Workshop on Algorithm Theory*, pages 173–180. Springer, 1990.

[31] Marie Durand, Bruno Raffin, and François Faure. A packed memory array to keep moving particles sorted. In *VRIPHYS*, pages 69–77. Eurographics Association, 2012.

[32] Yuval Emek and Amos Korman. New bounds for the controller problem. *Distributed Computing*, 24(3-4):177–186, 2011.

[33] Rolf Fagerberg, David Hammer, and Ulrich Meyer. On optimal balance in B-trees: What does it cost to stay in perfect shape? In *ISAAC*, volume 149 of *LIPIcs*, pages 35:1–35:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[34] David A. Freedman. On Tail Probabilities for Martingales. *The Annals of Probability*, 3(1):100 – 118, 1975.

[35] Anna Gál, Meena Mahajan, Rahul Santhanam, and Till Tantau. Computational complexity of discrete problems (dagstuhl seminar 21121). In *Dagstuhl Reports*, volume 11. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[36] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 165–174. ACM/SIAM, 1993.

[37] Micha Hofri and Alan G. Konheim. Padded lists revisited. *SIAM Journal on Computing*, 16(6):1073–1114, 1987.

[38] Alon Itai and Irit Katriel. Canonical density control. *Inf. Process. Lett.*, 104(6):200–204, 2007.

[39] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Proc. 8th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, 1981.

[40] Irit Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Israel Inst. of Tech., Haifa, May 2002.

[41] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Fast and scalable inequality joins. *The VLDB Journal*, 26(1):125–150, 2017.

[42] Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proc. 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 283–292, 2012.

[43] William Kuszmaul. *Randomized Data Structures: New Perspectives and Hidden Surprises*. PhD thesis, Massachusetts Institute of Technology, 2023.

[44] Dean De Leo and Peter A. Boncz. Fast concurrent reads and updates with PMAs. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 8:1–8:8. ACM, 2019.

[45] Dean De Leo and Peter A. Boncz. Packed memory arrays - rewired. In *35th IEEE International Conference on Data Engineering (ICDE)*, pages 830–841. IEEE, 2019.

[46] Dean De Leo and Peter A. Boncz. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endowment 14*, 14(6):1053–1066, 2021.

[47] Samuel McCauley, Ben Moseley, Aidin Niaparast, and Shikha Singh. Online list labeling with predictions. *Advances in Neural Information Processing Systems*, 36, 2024.

[48] Fahd Mustapha Meguellati and Djamel Eddine Zegour. A survey on balanced binary search trees methods. In *2021 International Conference on Information Systems and Advanced Technologies (ICISAT)*, pages 1–5. IEEE, 2021.

[49] Christian Worm Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 618–627. ACM/SIAM, 2003.

[50] Moni Naor and Vanessa Teague. Anti-persistence: history independent data structures. In *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 492–501, 2001.

[51] Yakov Nekrich. Space efficient dynamic orthogonal range reporting. *Algorithmica*, 49(2):94–108, 2007.

[52] Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.

[53] Yakov Nekrich. Searching in dynamic catalogs on a tree. *Computing Research Repository (CoRR)*, abs/1007.3415, 2010.

[54] Yakov Nekrich. A dynamic stabbing-max data structure with sub-logarithmic query time. *Computing Research Repository (CoRR)*, abs/1109.3890, 2011.

[55] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluç. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proc. 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1372–1385, 2021.

[56] Vijayshankar Raman. Locality preserving dictionaries: Theory and application to clustering in databases. In *Proc. 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 337–345, 1999.

[57] Michael Saks. Online labeling: Algorithms, lower bounds and open questions. In *International Computer Science Symposium in Russia (CSR)*, volume 10846, pages 23–28. Springer, 2018.

[58] Julio Toss, Cicero Augusto de Lara Pahins, Bruno Raffin, and João Luiz Dihl Comba. Packed-memory quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. *Computers & Graphics*, 76:117–128, 2018.

[59] Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21:101–112, 1984.

[60] Brian Wheatman and Randal Burns. Streaming sparse graphs using efficient dynamic sets. In *IEEE BigData*, pages 284–294. IEEE, 2021.

[61] Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *HPEC*, pages 1–7. IEEE, 2018.

[62] Brian Wheatman and Helen Xu. A parallel packed memory array to store dynamic graphs. In *Proc. Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 31–45. SIAM, 2021.

[63] Dan E. Willard. Inserting and deleting records in blocked sequential files. Technical Report TM81-45193-5, Bell Labs Tech Reports, 1981.

[64] Dan E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proc. 14th Annual Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.
[65] Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 251–260, 1986.
[66] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.
[67] Ju Zhang. *Density control and on-line labeling problems*. PhD thesis, University of Rochester, 1993.

APPENDIX A
REDUCING THEOREM 1 TO 3

To reduce Theorem 1 to Theorem 3, we will make a series of (standard) simplifications that are each without loss of generality.

Ignoring deletions. We may assume without loss of generality that the sequence of operations includes only insertions.

Proposition 25. *Any list-labeling solution that can start with (up to) $(1-3\gamma)m$ elements and support γm insertions with amortized expected cost $O(t(m,\gamma))$, can be modified to handle an arbitrary sequence of insertions/deletions, with up to $n=(1-\delta)m$ elements present at a time, and with amortized expected cost $O(t(m,\delta/3)+1/\delta)$ per operation.*

Proof. Set $\gamma=\delta/3$. We can collect operations into batches of size γm. As a batch forms, we "pretend" that the elements in the batch have not yet been deleted (i.e., we replace the deleted elements with tombstones, which we think of as elements).

Once a batch is fully formed, we rebuild the entire data structure from scratch, so that the deleted elements are cleared out. This rebuild increases the amortized expected cost by $O(1/\delta)$ per operation. During each batch, we are supporting an insertion-only workload that starts with (up to) $(1-\delta)m=(1-3\gamma)m$ elements and performs up to $\delta n/3=\gamma m$ insertions. The amortized expected cost per batch is therefore $O(t(m,\gamma)+1/\delta)$ □

Reducing to $n=m/2$. Our new task is to support a sequence of insertions that starts with up to $(1-3\delta)n$ elements, and performs up to δn insertions.

The following lemma reduces this problem to the problem of performing $n=m/2$ insertions in an initially empty size-m array.

Lemma 26. *Let there be a list labeling algorithm A' that for every $n'\geq 1$, it can insert n' items into an initially empty array of size $m'=2n'$ for amortized expected cost $t(n')$, where $t(n')\geq 1$ is a non-decreasing function. Then for every fixed $\delta\in(0,1/2]$ there is a list labeling algorithm A that for every $m\geq 1$ can insert $\lceil\delta m/3\rceil$ items into an array of size m that already contains $\lfloor(1-\delta)m\rfloor$ items, and where the amortized expected cost is at most cost $O(t(m)/\delta+1/\delta)$.*

As the proof of Lemma 26 requires some care, we defer it to the end of the section.

Starting with $m/4$ elements. So far, we have reduced Theorem 1 to the setting in which we wish to perform $n=m/2$ insertions in an initially-empty array of size m. However, we can break these insertions into batches, where we fill the array from from $1/2^i$ full to $1/2^{i-1}$ full, for some i, and we can implement each batch on an array of size $m/2^{i-2}\leq m$. Thus, if we focus just on the task of implementing a batch, our final problem is: perform $m/4$ insertions in an array that initially contains $m/4$ elements. This is precisely the problem considered by Theorem 3, which completes the reduction from Theorem 1.

Proving Lemma 26. We now prove Lemma 26.

Proof. For $\delta<12/m$ the claim is trivial so we assume that $\delta\geq 12/m$. Our algorithm A with a ***real*** array of size m will simulate algorithm A' on a ***virtual*** array of size $m'=2n'$, where $n'=2\lfloor\delta m/3\rfloor$. Algorithm A' will get to insert n' items into its virtual array. The first $n'/2$ items that will A' get are selected from the initial items that are in the real array of A. The next $n'/2$ items will be the items that A should insert into its real array (except for the very last one depending on rounding). The state of the real array during the latter $n'/2$ insertions will reflect the state of the virtual array.

A will classify each of its items as either ***visible*** or ***invisible***. All items that will be inserted into the virtual array will be visible, all the other items will be invisible. In particular, all the items newly inserted into the real array will be visible. Initially, A selects from the real array $n'/2$ items as the visible items and declares the remaining items as invisible. The algorithm selects as visible each initial item of rank $1+\lceil 3/\delta\rceil i$, for $i=0,1,\ldots,$ together with additional items of the smallest rank so to have exactly $n'/2$ visible items. (As the number of initial items is at least $\lfloor m/2\rfloor\geq n'/2$, there are enough items to chose from.)

Algorithm A will maintain the following two invariants: (1) No free slot in the real array can be immediately to the left of an invisible item, and (2) If we remove the invisible items together with their slots from the real array we get a copy of the current state of the virtual array. Since the left-most item in A will be always visible, invariant (1) means that invisible items form blocks of invisible items that follow immediately a visible item. After each block of invisible items there might be free slots followed by a visible item. Because we initially select each item of rank $1+\lceil 3/\delta\rceil i$, for $i=0,1,\ldots,$ as visible and newly inserted items will be also visible, each block of invisible items will always be of size at most $\lceil 3/\delta\rceil-1$.

To start the simulation, A inserts the initial set of visible items into the virtual array using A'. Then it will rearrange the real array to satisfy the two invariants. This will move at most m items in the real array. Then we process new insertions into A.

For each newly inserted item b, A proceeds as follows. It passes b to A' as a new insertion. In response to the insertion request, A' might rearrange its items in the virtual array to prepare an appropriate free slot for b. Then A' inserts b into the free slot. Before A' inserts b into the free slot, A rearranges

its real array to satisfy invariant (2) (and also invariant (1)) as items in the virtual array might have moved. Notice, the position of a particular visible item in the real array is given by the number of visible items to its left, together with the number of empty slots to its left, and the number of invisible items to its left. Similarly, the position of the same visible item in the virtual array is given by the number of visible items to its left together with the number of empty slots to its left. This implies that if an item in the virtual array retains its position during the rearrangement by A', it should retain its position also in the real array during the rearrangement by A. Also the block of invisible items following such an item will stay in place.

Thus A will have to move at most $\lceil 3/\delta \rceil$-times many items as A' did in the virtual array in order to re-establish the invariants. (It has to move the same number of visible items and each is followed by a block of at most $\lceil 3/\delta \rceil - 1$ invisible items.)

After the rearrangement of the real array, A will proceed to insert the item b. Let a be the closest visible item before b in the virtual array. Let b be put into i-th empty slot following a in the virtual array. Let there be ℓ invisible items following a in the real array. Let ℓ' of those invisible items be smaller than b. Algorithm A will move the last $\ell - \ell'$ invisible items following immediately after a in the real array i positions to the right. Then A inserts b into $(\ell' + i)$-th position after a, that is in the free slot immediately to the left of the moved invisible items. This will re-establish the correspondence between the virtual and real array. The cost of the additional moves is at most $\lceil 3/\delta \rceil$.

The total number of moves done by A' during its n' insertions is $n' \cdot t(n')$. (Although only half of the inserted items are new.) Hence, the total number of moves done by A during $n'/2$ new insertions is bounded by $m + \lceil 3/\delta \rceil \cdot n' \cdot t(n') + \lceil 3/\delta \rceil \cdot n'/2$. Since $\lceil 3/\delta \rceil \cdot \lfloor \delta m/3 \rfloor \leq \frac{\delta m}{3} \cdot \frac{3+\delta}{\delta} \leq 2m$, the total cost can be bounded by $4mt(m) + 3m$.

We can accommodate an additional insert into A for the cost of at most m, hence inserting at least $\delta m/3$ items for the amortized expected cost $3(4mt(m) + 4m)/\delta m = 12(t(m) + 1)/\delta$ as needed. □

Finally, it is worth pointing out one corollary of the lemma, which is the following claim about filling an array from empty to full:

Corollary 27. *If there is a list labeling algorithm A' that for every $n' \geq 1$ can insert n' items into an initially empty array of size $m' = 2n'$ for amortized expected cost $t(n')$, where $t(n') \geq 1$ is a non-decreasing function then there is a list labeling algorithm A that can insert n items into an initially empty array of size n with amortized expected cost $O(t(n) \log n)$ per insertion.*

Proof. First, apply the algorithm A' to insert $\lfloor n/2 \rfloor$ items into the array for the total cost at most $n \cdot t(n)$. Then proceed in phases. Each phase $i = 1, \ldots,$ starts with $e_i \geq 1$ remaining empty slots. It applies algorithm A from Lemma 26 for $\varepsilon_i = e_i/n$ to insert next $\lceil e_i/3 \rceil$ items. The algorithm stops once n items are inserted. The cost of each phase $i \geq 1$ is at most $\lceil \frac{e_i}{3} \rceil \cdot 12(t(n)+1)/\delta = \lceil \frac{e_i}{3} \rceil \cdot \frac{n}{e_i} \cdot 12(t(n)+1) \leq 12n(t(n)+1)$. Since $e_{i+1} \leq 2e_i/3$, there are at most $\log_{3/2} n$ phases. Thus the total cost to fill in the array is bounded by $O(n \cdot t(n) \log n)$. The lemma follows. □

Thus, one immediate consequence of Theorem 3 is:

Corollary 4. *There is a list-labeling algorithm that inserts n items into an initially empty array of size n with amortized expected cost $O((\log^2 n)(\log \log n)^3)$.*

APPENDIX B
PSEUDOCODE FOR THE SEE-SAW ALGORITHM

In this section, we give pseudocode for the See-Saw Algorithm. We assume parameters $\alpha = C_\alpha (\log \log n)^2$ and $\beta = C_\beta (\log \log n)^2$, where C_α and C_β are positive constants selected so that C_α, C_β, and C_α / C_β are all sufficiently large.

Variables to be used in pseudocode. Before presenting the algorithm pseudocode, we list the relevant variables for ***subproblem*** π. We emphasize that many of these variables are dynamically changing over time, i.e., are updated dynamically within the pseudocode.

- $L(\pi)$ and $R(\pi)$ are the ***left and right child*** of π, respectively.
- $\mathcal{A}_\pi$ is an array such that $\mathcal{A}_\pi = \mathcal{A}_{L(\pi)} \oplus \mathcal{A}_{R(\pi)}$ (the concatenation of the arrays).
- Q_π is the ***array skew***, such that $|\mathcal{A}_{L(\pi)}| = |\mathcal{A}_\pi|/2 - Q_\pi$ and $|\mathcal{A}_{R(\pi)}| = |\mathcal{A}_\pi|/2 + Q_\pi$.
- The ***pivot*** τ_π partitions the insertions that go to the left and right children of π. Upon creation of a subproblem, τ_π will be set to be the largest element stored in the subarray of its left child. [9] This element will remain the pivot until π ends or is reset.
- The ***rebuild window size*** w_π is the number of insertions permitted between rebuilds.
- σ_π is the number of insertions that have occurred during the current rebuild window.
- υ_π is the number of insertions that have occurred during the lifetime of π.
- δ_π is the number of insertions that occurred during the current window that are greater than the pivot minus those that are less than the pivot. This is called the ***insertion skew*** of the window.
- μ_π is a counter specifying which window we are in, starting with window 1.

Pseudocode. Below, we give pseudocode for both insertions and the subroutines used within an insertion. We assume that the subproblem tree is initialized (at the beginning of time, with $m/4$ initial elements) by a call to CREATESUBTREE.

[9] It turns out that $L(\pi)$ is guaranteed to have at least one element, so τ_π is guaranteed to exist. Since we are not going to prove this explicitly, one can think of there as being an extra edge case (that will never occur) in which, if $L(\pi)$ has no elements, then insertions to π always go to $R(\pi)$.

CREATESUBTREE($\mathcal{A}', \mathcal{S}'$)

Move the items in $\mathcal{S}'$ so that they are uniformly spread out in array $\mathcal{A}'$
return ALLOCATEBALANCEDSUBPROBLEMS($\mathcal{A}'$)
▷ Builds tree of subproblems on $\mathcal{A}'$

ALLOCATEBALANCEDSUBPROBLEMS($\mathcal{A}'$):

Create a new subproblem π
$\mathcal{A}_\pi \leftarrow \mathcal{A}'$
if density(π) > 0.75 or $|\mathcal{A}_\pi| \leq 2^{\sqrt{\log n}}$ **then**
    Declare π to be a leaf
    return π
$w_\pi \leftarrow$ PICKWINDOWLENGTH(π)
$\sigma_\pi \leftarrow 0$; $\delta_\pi \leftarrow 0$; $\mu_\pi \leftarrow 0$; $\upsilon_\pi \leftarrow 0$
$L \leftarrow$ the left half of $\mathcal{A}_\pi$, $R \leftarrow$ the right half of $\mathcal{A}_\pi$
$\tau_\pi \leftarrow$ the largest element in L
$L(\pi) \leftarrow$ ALLOCATEBALANCEDSUBPROBLEMS($\mathcal{A}_{L(\pi)}$)
$R(\pi) \leftarrow$ ALLOCATEBALANCEDSUBPROBLEMS($\mathcal{A}_{R(\pi)}$)
return π

INSERT(x, π):

if π is a leaf **then**
    Insert x into π using the classical algorithm
    return
if $x \leq \tau_\pi$ **then**
    INSERT($x, L(\pi)$)
    $\delta_\pi \leftarrow \delta_\pi - 1$
    if $\upsilon_{L(\pi)} \geq |\mathcal{A}_{L(\pi)}|/\alpha$ **then**
        $L(\pi) \leftarrow$ CREATESUBTREE($\mathcal{A}_{L(\pi)}$, SET($L(\pi)$))
        ▷ Reset $L(\pi)$
else
    INSERT($x, R(\pi)$)
    $\delta_\pi \leftarrow \delta_\pi + 1$
    if $\upsilon_{R(\pi)} \geq |\mathcal{A}_{R(\pi)}|/\alpha$ **then**
        $R(\pi) \leftarrow$ CREATESUBTREE($\mathcal{A}_{R(\pi)}$, SET($R(\pi)$))
        ▷ Reset $R(\pi)$
$\sigma_\pi \leftarrow \sigma_\pi + 1$
$\upsilon_\pi \leftarrow \upsilon_\pi + 1$
if $\sigma_\pi = w_\pi$ **then** ▷ End of rebuild window
    SKEWREBUILD(π)
    $\mu_\pi \leftarrow \mu_\pi + 1$; $\sigma_\pi \leftarrow 0$; $\delta_\pi \leftarrow 0$
if $\pi =$ root and $\sigma_\pi = m/\alpha$ **then**
    root $\leftarrow$ CREATESUBTREE($\mathcal{A}$, SET(root))
    ▷ Reset the root

SKEWREBUILD(π):

$Q_\pi \leftarrow$ PICKARRAYSKEW(π)
$\mathcal{S}_L \leftarrow$SET($L(\pi)$)
$\mathcal{S}_R \leftarrow$SET($R(\pi)$) ▷ Keep the items stored in the left and right children the same
$L \leftarrow$ the array consisting of the first $|\mathcal{A}_\pi| - Q_\pi$ slots in $\mathcal{A}_\pi$
$R \leftarrow$ the array consisting of the remaining $|\mathcal{A}_\pi| + Q_\pi$ slots in $\mathcal{A}_\pi$
$L(\pi) \leftarrow$ CREATESUBTREE(L, $\mathcal{S}_L$)
$R(\pi) \leftarrow$CREATESUBTREE(R, $\mathcal{S}_R$)

PICKARRAYSKEW(π):

if μ_π is odd **then**
    return 0
else
    return $|\mathcal{A}_\pi| \cdot \frac{\delta_\pi}{\beta w_\pi}$

PICKWINDOWLENGTH(π):

$k_{\max} \leftarrow 2 \log\log n$
For $k \in [1, k_{\max}]$, $p_k \leftarrow 2^{-(k+1)}(1 + k/k_{\max})$
$p_0 \leftarrow 1 - \sum_{k=1}^{k_{\max}} p_i$
Draw K_π so that $\Pr[K_\pi = k] = p_k$
return $|\mathcal{A}_\pi|/(\alpha 2^{K_\pi})$

SET(π):

return $\{y \mid y$ is stored in $\mathcal{A}_\pi\}$