2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS) | 978-1-6654-5519-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/FOCS54457.2022.00096

Online List Labeling: Breaking the $\log^2 n$ Barrier

Michael A. Bender, *Stony Brook University*
Alex Conway, *VMWare Research*
Martín Farach-Colton, *Rutgers University*
Hanna Komlós, *Rutgers University*
William Kuszmaul, *MIT*
Nicole Wein, *DIMACS*

***Abstract*—The online list-labeling problem is an algorithmic primitive with a large literature of upper bounds, lower bounds, and applications. The goal is to store a dynamically-changing set of n items in an array of m slots, while maintaining the invariant that the items appear in sorted order, and while minimizing the *relabeling* cost, defined to be the number of items that are moved per insertion/deletion.**

For the linear regime, where m is $(1+\Theta(1))n$, an upper bound of $O(\log^2 n)$ on the relabeling cost has been known since 1981. A lower bound of $\Omega(\log^2 n)$ is known for deterministic algorithms and for so-called smooth algorithms, but the best general lower bound remains $\Omega(\log n)$. The central open question in the field is whether $O(\log^2 n)$ is optimal for all algorithms.

In this paper, we give a randomized data structure that achieves an expected relabeling cost of $O(\log^{3/2} n)$ per operation. More generally, if m is $(1+\varepsilon)n$ for ε as $O(1)$, the expected relabeling cost becomes $O(\varepsilon^{-1}\log^{3/2} n)$.

Our solution is *history-independent*, meaning that the state of the data structure is independent of the order in which items are inserted/deleted. For history-independent data structures, we also prove a matching lower bound: for all ε between $1/n^{1/3}$ and some sufficiently small positive constant, the optimal expected cost for history-independent list-labeling solutions is $\Theta(\varepsilon^{-1}\log^{3/2} n)$.

I. Introduction

The online *list-labeling problem* is one of the most basic and well-studied algorithmic primitives in data structures, with an extensive literature spanning upper bounds [1]–[14], lower bounds [15]–[20], variants [8], [9], [11], [21]–[25], and open-problem surveys [19], [26]. The problem has been independently re-discovered in many different contexts [9], [24], [25], [27], and it has found extensive applications to areas such as ordered maintenance [5], [6], [23], [28], cache-oblivious data structures [13], [14], [29]–[31], dense file maintenance [2]–[4], [27], applied graph algorithms [32]–[37], etc.

The list-labeling problem was originally formulated [1] as follows. An algorithm must store a set of n elements (where n changes over time) in sorted order in an array of $m \geq n$ slots. Elements are inserted and deleted over time, with each insertion specifying the new element's *rank* $r \in \{1, 2, \ldots, n+1\}$ among the other elements that are present (e.g., inserting at rank 1 means that the inserted element is the new smallest element). To keep the elements in sorted order in the array, the algorithm must sometimes move elements around. The cost of an algorithm is the number of elements moved during the insertions/deletions.[1]

The list-labeling problem is well understood in the regime where $m \gg n$. In the *pseudo-exponential regime*, when $\frac{m}{n} = 2^{\omega(\log n)}$, it is possible to achieve $O(1)$ amortized cost per operation [10]. In the *polynomial regime*, when $\frac{m}{n} = n^{\Theta(1)}$, the amortized cost becomes $O(\log n)$ [9], [24], [38]. These bounds are known to be tight for both deterministic and randomized algorithms [10], [39], [40].

It has remained an open problem, however, what happens in the *linear regime*, where $m = (1+\varepsilon)n$ for some $\varepsilon = \Theta(1)$. In 1981, Itai, Konheim, and Rodeh [1] showed how to achieve amortized cost $O(\log^2 n)$, and posed as an open question whether any algorithm could do better. Despite a great deal of subsequent work on alternative solutions (including deterministic, randomized, and deamortized algorithms) for the same problem [2]–[7], [11], [12], [21], the bound of $O(\log^2 n)$ has remained unimproved for four decades.

Starting in 1990, there has been a long line of work towards establishing a matching $\Omega(\log^2 n)$ lower bound [15]–[17], [20], [39]. It is known that any deterministic algorithm requires $\Omega(\log^2 n)$ amortized cost per insertion [20]. And the same lower bound holds for *smooth* algorithms, where the relabelings are restricted to evenly rebalance elements across a contiguous subarray [15]. This second lower bound is surprisingly strong: it applies even to randomized algorithms and even to the offline problem, where the entire sequence of operations is known *a priori*. However, the best general lower bound remains $\Omega(\log n)$ [39].

These lower bounds tell us that, if an algorithm is to beat the $O(\log^2 n)$ bound, then the algorithm must be both randomized and non-smooth. Whether or not any such algorithm is possible has remained the central open question [1], [15]–[17], [20], [39] in this research area (see also discussion of the problem in open-problem surveys and textbooks [19], [26], [41]). Several sources [15]–[17] have conjectured that $\Theta(\log^2 n)$ cost is optimal in general.

A. Breaking through the $\log^2 n$ barrier.

We present a randomized list-labeling algorithm that achieves expected cost $O(\log^{3/2} n)$ per insertion/deletion in the linear regime (Corollary 15). In breaking through the

[1] To accommodate the many ways in which list labeling is used, some works describe the problem in a more abstract (but equivalent) way: the list-labeling algorithm must dynamically assign each element $x \in S$ a label $\ell(x) \in \{1, 2, \ldots, m\}$ such that $x \prec y \iff \ell(x) < \ell(y)$, and the goal is to minimize the number of elements that are relabeled per insertion/deletion—hence the name of the problem.

978-1-6654-5519-0/22/$31.00 ©2022 IEEE
DOI 10.1109/FOCS54457.2022.00096

$\log^2 n$ barrier, we establish that there is a fundamental gap between deterministic and randomized algorithms for online list labeling. Our result is the first asymptotic improvement in the linear regime in the 40-year history of the problem.

The original $O(\log^2 n)$ upper bound by Itai et al. [1] also extends to the *dense regime* of $\varepsilon = o(1)$, where the bound on amortized cost becomes $O(\varepsilon^{-1}\log^2 n)$ [8], [18], [42]. Extending our algorithm to the same regime, we achieve expected cost $O(\varepsilon^{-1}\log^{3/2} n)$ (Theorem 14).

Applying our result to the insertion-only setting, the array can be filled from empty to full (i.e., $n = m$) in total expected time $O(n\log^{2.5} n)$ (Corollary 16). This improves over the previous state of the art of $O(n\log^3 n)$, which was known to be optimal for deterministic algorithms [20]—again we have a separation between what can be achieved with deterministic and randomized algorithms.

A surprising aspect of our results is how they contrast with the polynomial regime $m = n^{1+\Theta(1)}$, where randomized and deterministic algorithms are asymptotically equivalent [10], [39], [40]. Our final upper-bound result considers a continuum between these regimes, where $m = \omega(n) \cap n^{o(1)}$. In this *sparse regime* there is a folklore bound [9], [24], [38] of $O\left(\frac{\log^2 n}{\log(m/n)}\right)$, which continuously deforms between $O(\log^2 n)$ for the linear regime and $O(\log n)$ for the polynomial regime. Using our techniques , we achieve[2] expected cost

$$O\left(\frac{\log^{3/2} n}{\sqrt{\log(m/n)}}\right).$$

Thus we achieve asymptotic improvements for list labeling for all $m = n^{1+o(1)}$.

B. An unexpected tool: history independence.

One research area that our algorithms build directly upon is the study of history-independent data structures: a data structure is said to be *history independent* [43], [44] if its current state reveals nothing about the history of the past operations beyond the current set of elements that are present.

History independence is typically viewed as a security guarantee, with the intent being to minimize the risk incurred by a security breach. Research on history-independent data structures [43]–[50] (as well as on history-independent list labeling [21] specifically) has focused on history independence as an *end goal*, with the question being whether history independence can be achieved without any increase in running time.

We find that, in the context of list labeling, history independence is actually a valuable *algorithmic* tool for building faster randomized data structures. History independence allows for us to have a data structure with vulnerabilities (i.e., certain spots where an insertion would be expensive) while (1) keeping those vulnerabilities hidden from the adversary; and (2) preventing the adversary from having any control over where those vulnerabilities appear. This simple paradigm plays an important role in allowing our randomized data structures to bypass the $\log^2 n$ barrier.

C. A matching lower bound for history-independent data structures.

Finally, we show that our bounds in the dense regime are asymptotically optimal for any history-independent data structure: there exists a positive constant c such that, for all $1/n^{1/3} \le \varepsilon \le 1/c$, the expected insertion/deletion cost when $m = (1+\varepsilon)n$ is necessarily at least $\Omega(\varepsilon^{-1}\log^{3/2} n)$ for any history-independent data structure.[3]

This means that, if there exists a randomized data structure that achieves better bounds than those in this paper, then the data structure must fundamentally be *adaptive* in how it responds to the history of the operations being performed. Of course, by being adaptive, such a data structure would also implicitly surrender the structural anonymity that history independence offers, revealing information about where the "hotspots" are within the data structure. Our results suggest that $\log^{3/2} n$ is a potentially fundamental barrier—whether or not the bounds achieved in this paper are optimal in general remains an enticing open problem.

D. Paper outline.

The rest of this paper proceeds as follows. Section II gives preliminaries. Section III gives an intuitive overview of the technical content of the paper. Sections IV and V present our upper bounds for the linear/dense regime. A key technical idea is to control the local density of the array via a random process that we call a Zeno random walk—we describe and analyze this random walk in Section IV. Section V then gives our (history-independent) list-labeling data structure and uses the bounds on Zeno random walks to analyze it. We refer readers to the full version[4] of the paper for our lower bound for history-independent data structures, upper bound for sparse list-labeling, and related work.

II. Preliminaries

In this section, we formally define the list-labeling problem and history independence—we then outline the classical $O(\log^2 n)$ solution [1] and a more recent history-independent variation on that solution [21].

A. The list-labeling problem.

A list-labeling data structure stores a dynamically changing set of size $n \le m$ in an array of m slots. It supports two operations:

- Insert(r), $r \in \{1, 2, \ldots, n+1\}$: This operation adds an element whose rank is r. This increments n and also increments the ranks of each of the elements whose ranks were formerly in $\{r, \ldots, n\}$.
- Delete(r), $r \in \{1, 2, \ldots, n\}$: This operation removes the element whose rank is r. This decrements n and also

[2] See Section 6 in the full version of the paper.

[3] See Section 5 in the full version of the paper.

[4] https://arxiv.org/abs/2203.00763

decrements the ranks of each of the elements whose ranks were formally in $\{r+1, \ldots, n\}$.

The list-labeling algorithm must maintain the invariant that the elements appear in sorted order (by rank) within the array. The *cost* of an insertion/deletion is the number of elements that are moved within the array during the insertion/deletion (including the element being inserted/deleted). In the case where $n = \Omega(m)$, we will further guarantee (for our upper bounds) that the maximum gap between any two consecutive elements in the array is at most $O(1)$ positions—this extra guarantee is often required for applications of list labeling in which algorithms perform range queries within the array, e.g., [14], [30], [32], [33], [35].

We will typically use an additional parameter ε such that either $n \leq (1-\varepsilon)m$ or $m \geq (1+\varepsilon)n$ (the specific convention that we follow will differ from section to section to optimize for simplifying the algebraic manipulation in each section).

From the perspective of the list-labeling data structure, the elements that it stores are black boxes—the only information that the data structure knows about its elements is their sorted order. This allows for list labeling to be used in applications where the elements are from arbitrary universes.

Finally, it is important to emphasize that the insertions/deletions are performed by an *oblivious adversary*, who does not get to see the random decisions made by the list-labeling data structure. If the adversary were to be adaptive, then, trivially, no randomized list-labeling data structure could incur expected cost any better than the worst-case cost of the best deterministic list-labeling data structure.

B. History Independence.

A data structure is said to be *history independent* [21], [43]–[50] if, given access to the current state of the data structure, the only information that an adversary can deduce is the current set of elements; that is, the adversary gains no information about the history of operations performed. In the list-labeling data structure the current set of elements is specified only by their relative ranks, so the only information that an adversary can deduce is the *number* of elements.

History independence plays an important supporting role throughout this paper. Indeed, although history independence does not on its own improve the asymptotics of list labeling, it does create a natural abstraction for how to separate the behavior of the data structure that we are designing from the actions of the user.

There are several basic mathematical properties of history independence that will be useful in both our upper and lower bounds. Define the *array configuration* of a list-labeling data structure to be the boolean vector in $\{0,1\}^m$ indicating which n positions of the array contain elements. We have the following properties of a history-independent data structure for list-labeling:

Property 1.

1) *Whenever the array contains n elements, its array configuration A satisfies $A \sim \mathcal{C}_{n,m}$, where $\mathcal{C}_{n,m}$ is some probability distribution over array configurations.*
2) *Whenever an insertion is performed at rank $r \in \{1, 2, \ldots, n+1\}$ in an array with n elements, the array configurations A_0 and A_1 before and after the insertion satisfy $(A_0, A_1) \sim \mathcal{I}_{n,m,r}$, where $\mathcal{I}_{n,m,r}$ is a joint distribution between $\mathcal{C}_{n,m}$ and $\mathcal{C}_{n+1,m}$.*[3]
3) *Whenever a deletion is performed at rank $r \in \{1, 2, \ldots, n+1\}$ in an array with $n+1$ elements, the array configurations A_1 and A_0 before and after the deletion satisfy $(A_0, A_1) \sim \mathcal{D}_{n,m,r}$, where $\mathcal{D}_{n,m,r}$ is a joint distribution between $\mathcal{C}_{n,m}$ and $\mathcal{C}_{n+1,m}$.*

These properties imply that the (probability distribution on the) behavior of the algorithm on any given operation is fully determined by n, m, the operation (insertion or deletion), and the rank r of the element being inserted/deleted. In our upper bounds, we will further have that $\mathcal{D}_{n,m,r} = \mathcal{I}_{n,m,r}$; we call any list-labeling data structure with this property *insertion/deletion symmetric*.

C. The Classical Solution and its History-Independent Analogue

1) List labeling with weight-balanced trees.: The original solution to list labeling [1], due to Itai et al. [1] in 1981, can be described in terms of weight-balanced trees [9], [51], [52]. For brevity, we will describe the solution here for the linear regime, where $m = (1+\Theta(1))n$, but the same solution directly generalizes to all regimes from dense ($n = (1-\varepsilon)m$) to polynomial ($m = n^{1+\Theta(1)}$).

Consider an array of size m, and impose a tree structure on it, where the root node represents the entire array, the nodes in the i-th level of the tree represent disjoint sub-arrays of size $m/2^{i-1}$, and the leaf nodes represent sub-arrays of size $\Theta(\log n)$. We keep the tree tightly weight balanced, meaning that, for any pair of sibling nodes x and y, their densities are always within a $1 \pm O(1/\log n)$ factor of each other. In particular, whenever an insertion or deletion breaks this invariant for some pair of siblings x and y, we take the elements in the sub-array $x \cup y$ and rearrange them to be distributed evenly across that sub-array.[6]

This tight weight balancing ensures that *all* of the nodes in the tree have densities that are within a factor of $(1 + O(1/\log n))^{O(\log n)} = O(1)$ of each other. By selecting the constants in the algorithm appropriately, one can ensure that every leaf has more slots than it has elements, which guarantees the correctness of the data structure. On the other hand, in order to maintain such tight weight balancing, one must rebuild nodes a factor of $O(\log n)$ more often than in a standard weight-balanced binary search tree [9], [51], [52], leading to an amortized cost of $O(\log^2 n)$.

Intuitively, the above data structure would seem to be the asymptotically optimal approach to maintaining tightly-balanced densities within an array—the known lower bounds for list labeling [15]–[17], [20] confirm that this is the case

[3] A probability distribution $\mathcal{X}$ is a joint distribution between distributions $\mathcal{A}$ and $\mathcal{B}$ if $(A, B) \sim \mathcal{X} \implies A \sim \mathcal{A}$, $B \sim \mathcal{B}$.

[6] This approach is both deterministic and smooth, and thus consistent with the assumptions made by lower bounds [15]–[17], [20].

for both deterministic and smooth data structures. The upper bounds in this paper reveal that, perhaps surprisingly, it is not the case for randomized data structures. Randomization fundamentally reduces the cost to maintain a tightly weight-balanced tree.

2) History-Independent list labeling: To understand how history independence can be achieved in the context of list labeling, it is helpful to first understand it in the context of balanced binary search trees. The classic example of a balanced binary tree with a history-independent topology is the *randomized binary search tree* [53], [54] (or, similarly, the treap [53], [54]), which maintains as an invariant that, at any given moment, the structure of the tree is random (i.e., that within each subtree, the root of that subtree is a random element). This can be achieved with reservoir sampling [21], [53]–[56]—in particular, whenever a new item is added to a subtree of (former) size r, the element becomes the new root with probability $1/(r+1)$ (in which case the subtree is rebuilt from scratch). This simple approach yields an expected time of $O(\log n)$ per operation.

As shown by Bender et al. [21], the same basic approach can be used to achieve history-independent list labeling. Now, the tree is random across all *tightly balanced trees*—that is, within each subtree T containing elements $x_1 < x_2 < \cdots < x_k$, the root is a random element x_i of those satisfying $|i - k/2| \in O(k/\log n)$. As before, this structure can be maintained using reservoir sampling. However, the restriction that the tree must be tightly balanced increases the frequency with which subtrees are rebuilt, so that the expected cost per operation becomes $O(\log^2 n)$, just as for the standard solution to list labeling.

III. Technical Overview

In this section, we present an intuitive overview of our upper bound and proof techniques. Comprehensive technical details can be found in Sections IV and V. For simplicity, we shall assume in this section that $m = 2n$.

Intuitively, our starting point is the history-independent list labeling solution by Bender, et al. [21]. As described in Section II, in [21], the root of any subtree of size k is a random element of the middle $O(k/\log n)$ elements of the subtree. We call this middle set of elements the *candidate set*.

A natural idea for decreasing the cost of this algorithm is to increase the size of the candidate set to δk for some $\delta = \omega(1/\log n)$. This way, the root would be resampled less often, resulting in fewer total rebalances. However, there is a problem with this approach: the subarrays representing the nodes in the i-th level of the tree have densities bounded between $\frac{1}{2}(1-\delta)^i$ and $\frac{1}{2}(1+\delta)^i$, but this means that nodes in the $\Theta(\log n)$-th level can overflow with a density of $\frac{1}{2}(1+\delta)^{\Theta(\log n)} = \omega(1)$. Thus, having $\delta = \omega(1/\log n)$ violates the correctness of the algorithm.

Notice, however, that *most* nodes in the i-th level of the tree avoid a density of the form $\frac{1}{2}(1+\delta)^i$. Indeed, if we were to perform a random walk down the tree, then the node that we encountered on our i-th step would likely have a density bounded above by $\frac{1}{2}(1+\delta)^{O(\sqrt{i})}$. This means that, if we only wanted *most nodes* to behave well, then we could set δ close to $\frac{1}{\sqrt{\log n}}$.

In order to obtain the benefits of $\delta \approx 1/\sqrt{\log n}$ while maintaining the correctness of $\delta \approx 1/\log n$, we smoothly adjust the candidate set size for each subtree as a function of the subtree's density. We show that almost all subtrees are sparse enough to support a "large" candidate set ($\delta \approx 1/\sqrt{\log n}$), while only a small fraction of subtrees require "smaller" candidate sets (with δ closer to $1/\log n$). This means that most parts of the array support fast insertions/deletions, while only a small portion of the array is slow to insert/delete to.

While we have made progress by ensuring that most of the array can support fast updates, this is not sufficient to prove the final bound. Specifically, if the adversary *knows* which parts of the array are slow to update, they could simply focus all of their insertions/deletions on these slow parts of the array, causing the total cost to be large. Instead, we would like to *hide* the slow parts of the array from the adversary. More precisely, we are concerned about two distinct problems: the adversary could *create* dense regions through their insertion sequence (e.g., by concentrating insertions in one location), or, the adversary could *detect* dense regions created by the algorithm (e.g., through prior knowledge of the algorithm's distribution of states.)

History independence comes into play in guarding against these problems. By definition, the first problem cannot happen with a history-independent algorithm, since the configuration of the array does not depend on the adversary's specific sequence of insertions. For the second problem, we add an additional layer of randomness called a *random shift*. At the start of the algorithm, we insert random number $k \in [m]$ of dummy elements at the front of the array, and $m-k$ at the end. This converts a potentially adversarial insertion at rank j to a uniformly random insertion of rank between j and $j+m$. Together with history independence, the random shift ensures that the adversary cannot target specific regions of the array.

To analyze our algorithm, we introduce the notion of a *Zeno random walk*, which is a special type of bounded random walk where the step size decreases as the distance to a boundary decreases. The Zeno walk captures the way in which the densities of subproblems evolve if we perform a random walk down our tree. Our analysis of this random walk (Proposition 4) allows us to bound the cost of a random insertion (Lemma 12). Finally, we extend this analysis for a *random* insertion to an *arbitrary* insertion using the ideas outlined above of history independence and a random shift, achieving an expected $O(\log^{3/2} n)$ cost for any insertion/deletion.

IV. Zeno's Random Walk

This section describes and analyzes a simple but somewhat unusual type of random walk that we will refer to as a *Zeno walk*—this random walk will play an important algorithmic role in later sections.

Let $\delta \in (0, 1/2]$. A Zeno walk $Z_0, Z_1, Z_2, \ldots$ starts at $Z_0 = 0$ and deterministically satisfies $Z_t \in (-1, 1)$ for all t. We

define $\alpha_t = 1 - |Z_t|$ to be the distance between Z_t and the nearest boundary 1 or -1. We determine Z_{t+1} from Z_t as follows:

- An adaptive adversary selects a quantity $\delta_t \leq \delta$, possibly as a function of $Z_0, Z_1, \ldots, Z_t$.
- Z_{t+1} is then set to be one of $Z_t + \alpha_t\delta_t$ or $Z_t - \alpha_t\delta_t$, each with equal probability.

What makes the Zeno walk unusual is that, the closer it gets to -1 or 1, the smaller its steps become (since the t-th step has its size multiplied by α_t). The result is that (as in Zeno's paradox), the walk can get arbitrarily close to ± 1 but can never reach ± 1.

We will be interested in Zeno walks $Z_1, \ldots, Z_\ell$ where the relationship between δ and the length ℓ of the walk is $\delta = O(1/\sqrt{\ell})$. To gain some intuition here, consider the case where $\delta_t = \delta = 1/\sqrt{\ell}$ for all t, and let us compare the Zeno walk $Z_1, \ldots, Z_\ell$ to a standard unbiased random walk $X_1, \ldots, X_\ell$ that changes by $\pm 1/\sqrt{\ell}$ on each step. After ℓ steps, the random walk $X_1, \ldots, X_\ell$ deviates from the origin by $O(1)$ in *expectation* (but could deviate by much more) and has the property that each step is *deterministically* the same size. The Zeno walk does the complement of this: it deviates from the origin by at most 1 *deterministically*, but to do this it decreases the size of the t-th step by a factor of $1/\alpha_t$. The key property that we will prove (Proposition 3) is that, although the multiplier $1/\alpha_t$ can potentially be large, the *expected value* satisfies $O(1/\alpha_t) = O(1)$ for $t \in [\ell]$. With this intuition in mind, we can now begin the analysis.

Define $Y_t := \ln(1/(1 - Z_t))$. Rather than analyze the Z_t's directly, we will instead analyze the Y_t's. We will see that the sequence $Y_1, Y_2, \ldots$ behaves similarly to the standard random walk $X_1, X_2, \ldots$ that we described in the previous paragraph (except that (1) Y_t is slightly biased and (2) Y_t can never go below $\ln(0.5)$. To make this more precise, the next lemma shows that the random walk $Y_1, Y_2, \ldots$ takes steps of size at most $O(\delta)$ and has bias at most $O(\delta^2)$ per step.

Lemma 2. *For $t \geq 0$, we have that*

$$|Y_{t+1} - Y_t| = O(\delta) \tag{1}$$

deterministically, and that

$$\left|\mathbf{E}[Y_{t+1} - Y_t \mid Y_1, \ldots, Y_t, \delta_t]\right| = O(\delta^2). \tag{2}$$

Proof. Define

$$\gamma_t = \frac{\alpha_t\delta_t}{1 - Z_t}.$$

Note that, if $Z_t \geq 0$, then $\gamma_t = \delta_t$, and otherwise $\gamma_t < \delta_t$. Since $Z_{t+1} = Z_t \pm (1 - Z_t)\gamma_t$, we have that

$$\begin{aligned} Y_{t+1} &= \ln\left(\frac{1}{1 - Z_t \pm (1 - Z_t)\gamma_t}\right) \\ &= \ln\left(\frac{1}{1 - Z_t} \cdot \frac{1}{1 \pm \gamma_t}\right) \\ &= \ln\left(\frac{1}{1 - Z_t}\right) + \ln\left(\frac{1}{1 \pm \gamma_t}\right) \\ &= Y_t + \ln\left(\frac{1}{1 \pm \gamma_t}\right). \end{aligned}$$

By a Taylor approximation, we know that $\ln\left(\frac{1}{1 \pm \gamma_t}\right)$ is within $O(\gamma_t^2)$ of $\pm\gamma_t$. That is, Y_{t+1} can be computed from Y_t by first adding $\pm\gamma_t$ at random to Y_t, and then adding/subtracting an additional $O(\gamma_t^2)$. We therefore have that

$$|Y_{t+1} - Y_t| \leq \gamma_t + O(\gamma_t^2) \leq \delta_t + O(\delta_t^2) \leq O(\delta)$$

and that

$$\left|\mathbf{E}[Y_{t+1} - Y_t \mid Y_1, \ldots, Y_t, \gamma_t]\right| \leq O(\gamma_t^2) \leq O(\delta_t^2) \leq O(\delta^2).$$

□

Using Lemma 2, we can now bound $\mathbf{E}[1/\alpha_t]$ for the $t = O(1/\delta^2)$-th step of a Zeno walk:

Proposition 3. *For $t = O(1/\delta^2)$, we have $\mathbf{E}[1/\alpha_t] = O(1)$.*

Proof. By symmetry, it suffices to show that

$$\mathbf{E}[1/\alpha_t \cdot \mathrm{I}_{Z_t \geq 0}] = O(1),$$

where $\mathrm{I}_{Z_t \geq 0}$ is 0-1 indicator random variable for the event $Z_t \geq 0$. Note that

$$\begin{aligned} \mathbf{E}[1/\alpha_t \cdot \mathrm{I}_{Z_t \geq 0}] &= E[1/(1 - Z_t) \cdot \mathrm{I}_{Z_t \geq 0}] \\ &\leq E[1/(1 - Z_t)], \end{aligned}$$

so we can complete the proof by showing that

$$\mathbf{E}[1/(1 - Z_t)] = O(1). \tag{3}$$

Let c be a sufficiently large positive constant and define the sequence $X_1, X_2, \ldots$, where

$$X_t = Y_t - t \cdot c\delta^2.$$

This means that $X_{t+1} - X_t = Y_{t+1} - Y_t - c\delta^2$, so we can think of the X_t's as being a modification of the Y_t's that eliminates any upward bias that the Y_t's might have (recall by Lemma 2 that the Y_t's have bias at most $O(\delta^2)$).

Formally, one can apply Lemma 2 to deduce that the X_t's are a supermartingale with bounded differences of $O(\delta)$. That is, by (2) we have $\mathbf{E}[X_{t+1} \mid X_1, \ldots, X_t] \leq X_t$ (so the X_t's form a supermartingale) and by (1) we have $|X_{t+1} - X_t| \leq O(\delta)$ (so the martingale has bounded differences of $O(\delta)$).

We can apply Azuma's inequality for supermartingales with bounded differences to deduce the following tail bound. For $k \geq 1$, we have

$$\Pr[X_t \geq \delta k\sqrt{t}] \leq e^{-\Omega(k^2)}.$$

Unrolling the definition of X_ℓ, we get that

$$\Pr[\ln(1/(1-Z_\ell)) \geq \delta k\sqrt{i} + i\alpha\delta^2] \leq e^{-\Omega(k^2)}.$$

Plugging in $i = \ell = O(1/\delta^2)$, we conclude that

$$\Pr[\ln(1/(1-Z_\ell)) \geq \Omega(k)] \leq e^{-\Omega(k^2)}.$$

This further simplifies to

$$\Pr\left[1/(1-Z_\ell) \geq e^{\Omega(k)}\right] \leq e^{-\Omega(k^2)},$$

which implies (3), and completes the proof. □

We conclude the section by generalizing Zeno walks to take place in an arbitrary interval $(\lambda-\epsilon, \lambda+\epsilon)$. This works exactly as before, except that now the Zeno walk begins at $Z_0 = \lambda$; it deterministically stays in the interval $(\lambda-\epsilon, \lambda+\epsilon)$; it sets $\alpha_t = \epsilon - |Z_t - \lambda|$ to be the distance from Z_t to the nearest boundary $\lambda-\epsilon$ or $\lambda+\epsilon$; and then $Z_{t+1} = Z_t \pm \alpha_t\delta_t$ where $\delta_t \leq \delta$ is selected by an adversary. Equivalently, a sequence $\{Z_t\}$ is a Zeno walk in the interval $(\lambda-\epsilon, \lambda+\epsilon)$ if $\{(Z_t-\lambda)/\epsilon\}$ is a Zeno walk in $(-1,1)$ (and the two Zeno walks have the same parameter δ as each other). Thus we get the following generalization of Proposition 3.

Proposition 4. *Consider a Zeno walk in $(\lambda-\epsilon, \lambda+\epsilon)$. For $\ell = O(1/\delta^2)$, we have $\mathbf{E}[1/\alpha_\ell] \leq O(\epsilon^{-1})$.*

V. The Zeno Embedding: A Data Structure for $m \geq (1+\epsilon)n$

In this section, we give a list-labeling solution for $m \geq (1+\epsilon)n$ that achieves expected cost $O(\epsilon^{-1}\log^{3/2} n)$ per insertion and deletion. We will treat $m \in \mathbb{N}$ and $\epsilon \in (0,1)$ as being fixed, and we will allow the number n of elements to vary subject to the constraint that $m \geq (1+\epsilon)n$. We will also assume without loss of generality that n is at least a sufficiently large positive constant.

We construct and analyze the data structure in three phases. First, we describe a certain type of static construction, which we call the *Zeno embedding*, for how to embed n elements into m slots. Then we show how to dynamize the Zeno embedding in order to efficiently implement *random* insertions/deletions. Finally, we present one last modification to the Zeno embedding in order to implement arbitrary insertions/deletions efficiently.

A. The Static Zeno Embedding

The Zeno embedding treats the array as having a simple recursive structure: the *level-0 subproblem* consists of the entire array; and the *level-i subproblems* each consist of either $\lfloor m/2^i \rfloor$ or $\lceil m/2^i \rceil$ contiguous slots in the array.

Each level-i subproblem S is either a *base case* (meaning it does not have child subproblems) or has two recursive children. If S has $q \in \{\lfloor m/2^i \rfloor, \lceil m/2^i \rceil\}$ slots, then the children of S have $\lfloor q/2 \rfloor$ and $\lceil q/2 \rceil$ slots, respectively. Here we are taking advantage of the basic mathematical fact that

$$\{\lfloor \lfloor m/2^i \rfloor/2 \rfloor, \lfloor \lceil m/2^i \rceil/2 \rfloor, \lceil \lfloor m/2^i \rfloor/2 \rceil, \lceil \lceil m/2^i \rceil/2 \rceil\}$$
$$\subseteq \{\lfloor m/2^{i+1} \rfloor, \lceil m/2^{i+1} \rceil\}.$$

For each level-i subproblem S, define $|S|$ to be the number of elements stored in that subproblem, and define the *density* μ_S of the subproblem to be

$$\mu_S = \frac{|S|}{n/2^i}.$$

Note that in the definition of μ_S, the denominator is the average number of elements per level-i subproblem, which means that μ_S can be greater than 1. In fact, we will guarantee deterministically that $\mu_S \in [1-\epsilon/2, 1+\epsilon/2]$. The upper bound will ensure correctness (i.e., that no subproblem overflows), and the lower bound will ensure that every pair of consecutive elements are within $O(1)$ slots of each other.

We can now describe how to implement a given level-i subproblem S. Define

$$\alpha_S = \epsilon/2 - |1-\mu_s|$$

to be the distance between μ_S and the nearest boundary $\{1-\epsilon/2, 1+\epsilon/2\}$. Let $x_1, \ldots, x_{|S|}$ denote the elements of S in sorted order. Define the *pivot candidate set* for S to be

$$C_S = \left\{x_i \;\middle|\; \frac{|S|}{2} - \frac{n}{2^i}\cdot\frac{\alpha_S}{\sqrt{\log n}} \leq i \leq \frac{|S|}{2} + \frac{n}{2^i}\cdot\frac{\alpha_S}{\sqrt{\log n}}\right\}.$$

Roughly speaking, C_S consists of the elements representing the middle $\Theta(\alpha_S/\sqrt{\log n})$-fraction of the subproblem.

If $|C_S| \leq 4$, we declare S to be a *base case*, and we spread the elements of S evenly across its slots. Otherwise, we define the *pivot* p_S for S to be an element of C_S chosen uniformly at random. The elements $x_i \leq p_S$ are recursively placed in S's left child, and the elements $x_i > p_S$ are recursively placed in S's right child.

Later on, when we discuss the *dynamic* Zeno embedding, we will see several ways that one can implement the random choice of p_S. For concreteness, we will mention one natural approach here: define $h_0, h_1, h_2, \ldots, h_{O(\log n)}$ to be an independent sequence of hash functions[7] where each h_i maps each element to a uniformly random real number in $[0,1]$, and set

$$p_S = \operatorname{argmin}_{x \in C_S} h_i(x).$$

The key property of the Zeno embedding is that if we perform a random walk down the recursive tree, then the densities μ_S that we encounter form an $O(\log n)$-step Zeno walk in the interval $[1-\epsilon/2, 1+\epsilon/2]$:

Lemma 5. *Fix any outcomes for the hash functions $h_0, h_1, h_2, \ldots$. Consider a random walk $S_0, S_1, S_2, \ldots, S_t$ down the recursion tree, where each S_{i+1} is a random child of S_i, and S_t is a base-case subproblem. Then the sequence $\{\mu_{S_i}\}_{i=1}^t$ is a Zeno walk on $[1-\epsilon/2, 1+\epsilon/2]$ with $\delta = O(1/\sqrt{\log n})$.*

[7] Technically, our data structure does not necessarily have access to the internal values of elements, so it cannot compute a hash $h_i(x)$ of any given element. However, we can simulate a hash function h_i by assigning each element x a random value $h_i(x)$ when the element is inserted.

Proof. Recall that a Zeno walk on $[1-\epsilon/2, 1+\epsilon/2]$ is any walk $Z_0, Z_1, \ldots$ that starts at 1 and takes the following form: each step $Z_{t+1} - Z_t$ is randomly $\pm\alpha_t\delta_t$ for some $\delta_t \le \delta$ (that may be chosen by an adversary) and where $\alpha_t = \epsilon/2 - |1 - Z_t|$. Or, equivalently, each step $Z_{t+1} - Z_t$ is randomly $\pm\beta_t$ for some $\beta_t \le \delta\,(\epsilon/2 - |1 - Z_t|)$.

Consider a non-base-case subproblem S_t, and let A and B be the child subproblems of S_t. By construction,

$$\big||A| - |B|\big| = O\left(\frac{n}{2^t} \cdot \frac{\alpha_S}{\sqrt{\log n}}\right).$$

Since $|A| + |B| = |S_t|$, we have that $\mu_A + \mu_B = 2\mu_{S_t}$ and

$$|\mu_A - \mu_B| = \frac{\big||A| - |B|\big|}{n/2^{t+1}} = O\left(\frac{\alpha_{S_t}}{\sqrt{\log n}}\right).$$

Thus, since S_{t+1} is randomly one of A or B, we have that $\mu_{S_{t+1}}$ is randomly one of

$$\mu_{S_t} + \beta_t \text{ or } \mu_{S_t} - \beta_t,$$

where

$$\beta_t = |\mu_A - \mu_B|/2 = O\left(\frac{\alpha_{S_t}}{\sqrt{\log n}}\right) = O\left(\delta\left(\epsilon/2 - |1 - \mu_t|\right)\right).$$

Thus the sequence $\{\mu_t\}$ is a Zeno walk on $[1-\epsilon/2, 1+\epsilon/2]$ with $\delta = O(1/\sqrt{\log n})$.

For clarity, we remark that the definition of the Zeno walk includes an adaptive adversary who chooses $\delta_t < \delta$. The adversary for the Zeno walk in this lemma simply chooses a pivot uniformly at random from the pivot candidate set, which determines δ_t. □

The reason that Lemma 5 is important is that it allows for us to bound the quantities α_S^{-1}. Indeed, we use Proposition 4 to prove the following inequality.

Lemma 6. *Let $\mathcal{S}_t$ be the set of level-t subproblems. Then*

$$\frac{1}{2^t}\sum_{S \in \mathcal{S}_t} \alpha_S^{-1} = O(\epsilon^{-1}).$$

Proof. Fix any outcomes for the hash functions $h_0, h_1, h_2, \ldots$. Consider a random walk $S_0, S_1, S_2, \ldots, S_\ell$ down the recursion tree, where each S_{t+1} is a random child of S_t, and S_ℓ is a base-case subproblem. Lemma 5 tells us that $\{\mu_{S_t}\}_{t=1}^{\ell}$ is a Zeno walk on $[1-\epsilon/2, 1+\epsilon/2]$ with $\delta = O(1/\sqrt{\log n})$ (and, moreover, α_{S_t} corresponds to α_t in the Zeno walk).

For $i \in [0, \log n]$, define $\overline{\alpha}_i$ to be α_{S_i} if S_i exists and 0 otherwise (i.e., if $i > \ell$). Proposition 4 tells us that, for each $i \in [0, \log n]$,

$$\mathbf{E}[1/\overline{\alpha}_i] = O(\epsilon^{-1}). \tag{4}$$

On the other hand, each level-i subproblem has probability exactly $1/2^i$ of being S_i. Thus

$$\mathbf{E}[1/\overline{\alpha}_i] = \frac{1}{2^i}\sum_{S \in \mathcal{S}_i} \alpha_S^{-1}. \tag{5}$$

Combined, (4) and (5) imply the lemma. □

It is interesting to note that, whereas Lemma 5 is a statement about random walks, Lemma 6 is a *deterministic* bound on the α_S^{-1}s, even though it uses a probabilistic argument to derive the bound.

Lastly, we also need to explicitly show that no subproblem ever overflows:

Lemma 7. *Each level-i subproblem S satisfies $|S| \le \lfloor n/2^i \rfloor$.*

This lemma is a technicality that is essentially immediate from the fact that each subproblem S has density $\mu_S \le 1+\epsilon/2$. The only difficulty in the proof comes from the necessity to carefully handle floors/ceilings. We defer the proof to Appendix in the full version of the paper.

B. Dynamizing the Zeno Embedding

We now describe a dynamic version of the Zeno embedding; we will treat n and ϵ as fixed, and allow n to vary subject to the constraint that $n \ge (1+\epsilon)^n$.

We note that, in this section we will focus on analyzing *random* insertions/deletions, that is, an insertion/deletion that is performed at a random rank (in an array with arbitrary contents). Our solution will be history independent, and we will see in the next subsection that this allows the random-rank assumption to be removed.

a) Implementing insertions and deletions: To implement an insertion/deletion in the Zeno embedding, we simply update the embedding to account for the element being added/removed. More concretely, we can implement an insertion/deletion of an element x as follows. We will describe the process recursively, focusing on how to insert/delete x into a given level-i recursive subproblem S. The insertion/deletion of x may change the values of μ_S, α_S, C_S, and p_S. Note that the values of C_S and p_S can change regardless of whether the insertion/deletion of x takes place in the candidate set. If it changes the pivot p_S, or if S is a base-case, then we implement the insertion/deletion by rebuilding the entire subproblem from scratch, incurring a cost of $O(n/2^i)$. Otherwise, we recursively insert/delete x into either the left child (if $x \le p_S$) or the right child (if $x > p_S$). Once the insertion/deletion is complete, the Zeno embedding will be the same as if it were constructed from scratch on the current set of elements.

As described in the static Zeno embedding, there are multiple ways to implement randomly choosing a pivot. One way is to use the hash functions h_t described in the previous subsection. This means that a level-i subproblem S being inserted/deleted into gets rebuilt if $\operatorname{argmin}_t\{h_t(x) \mid x \in C_S\}$ is changed by the insertion/deletion. We note that, in this construction, the hash functions are fixed at the very beginning and are never resampled (even when subproblems are rebuilt).

Another way to implement the random choice of pivot is to use reservoir sampling [21], [53]–[56]. This means that, when a subproblem is first built (or rebuilt), it picks a random $x \in C_S$ to be the pivot; whenever an element x is added to C_S, it has probability $1/|C_S \cup \{x\}|$ of becoming the pivot; and whenever an element x is removed from C_S, if x was the

pivot, then a random element in $C_S \setminus \{x\}$ is chosen as the new pivot. Like the hashing method, reservoir sampling maintains as an invariant that each candidate in C_S is equally likely to be the pivot.

Each of the two methods (hashing and reservoir sampling) have their own benefits: reservoir sampling can be used to immediately obtain an algorithm in the RAM-model that has the same asymptotic running time as its list-labeling cost, while hashing, on the other hand, ensures that the embedding is deterministic after fixing the hash functions. In our formal arguments, we use the hash function method, but this can easily be replaced with reservoir sampling.

b) Analyzing a random insertion/deletion.: To begin analyzing the dynamic Zeno embedding, we observe that, by construction, the dynamic Zeno embedding is insertion/deletion symmetric and history independent.

Observation 8. *The dynamic Zeno embedding is insertion/deletion symmetric and history independent.*

Due to the insertion/deletion symmetry, the expected cost of a random insertion on an array with n elements is the same as the expected cost of a random deletion on an array with $n+1$ elements. Thus we need only analyze the expected cost of a random deletion.

We will analyze the probability that the deletion of an element x causes the rebuild of a subproblem. More precisely, we say that a subproblem S is *rebuilt* if the pivot of S changes, while the pivots of all of the ancestors of S do not change.

Next, we will prove that, if we delete an element x, and S is the level-i subproblem that contains x, then the probability that S is rebuilt is $O(|C_S|^{-1})$.

Lemma 9. *If an element x is deleted from a subproblem S, then S is rebuilt with probability*

$$O\left(|C_S|^{-1}\right).$$

Proof. If S is a base-case subproblem, either before or after the deletion, then $|C_S| = O(1)$, and the lemma is trivial. Now, suppose S is not a base-case subproblem.

Let C_S denote the pivot candidate set prior to the deletion of x, and let $\overline{C}_S$ denote the pivot candidate set after the deletion. Each time that we add/remove an element to/from C_S, the probability that $p_S = \operatorname{argmin}_{x \in C_S} h(x)$ changes is $\Theta(1/|C_S|)$. It therefore suffices to show that C_S and $\overline{C}_S$ have a symmetric difference of at most $O(1)$ elements.

We can think of the transformation of C_S into $\overline{C}_S$ as taking place in three steps. First we update

$$\alpha_S = \epsilon/2 - \left|1 - \frac{|S|}{n/2^i}\right|$$

to become

$$\alpha_S = \epsilon/2 - \left|1 - \frac{|S|-1}{n/2^i}\right|.$$

This changes α_S by at most $\pm\frac{1}{n/2^i}$, which changes the set

$$C_S = \left\{x_i \;\middle|\; \frac{|S|}{2} - \frac{n}{2^i} \cdot \frac{\alpha_S}{\sqrt{\log n}} \le i \le \frac{|S|}{2} + \frac{n}{2^i} \cdot \frac{\alpha_S}{\sqrt{\log n}}\right\} \tag{6}$$

by at most $O(1)$ elements. Second, we replace $|S|$ in (6) with $|S|-1$. This again changes the set C_S by at most $O(1)$ elements. Third, we remove the element x; if $x = x_j$ for some j, then the removal of x has the effect of decrementing the index of each x_i with $i \ge j$. This again changes C_S by at most $O(1)$ elements.

Combined, the three steps complete the transformation of C_S into $\overline{C}_S$, meaning that $\overline{C}_S$ and C_S have a symmetric difference of $O(1)$ elements, as desired. □

Lemma 9 immediately implies a bound on the expected cost incurred from rebuilding S.

Lemma 10. *If an element x is deleted from a level-i subproblem S, the expected cost incurred from possibly rebuilding S is*

$$O\left(\frac{n/2^i}{|C_S|}\right).$$

Proof. A rebuild of S costs $\Theta(n/2^i)$. Thus the lemma follows from Lemma 9. □

Observe that, by design,

$$\frac{n/2^i}{|C_S|} = O(\alpha_S^{-1}\sqrt{\log n}).$$

This is where Lemma 6 comes into play: it tells us that even though $\frac{n/2^i}{|C_S|}$ may be large for some subproblems S, it cannot be consistently large across all subproblems. Using this, we can analyze the expected cost to delete a random element.

Lemma 11. *The expected cost to delete a random element x from the Zeno embedding is $O(\epsilon^{-1}\log^{3/2} n)$.*

Proof. Let $\mathcal{S}_i$ denote the set of level-i subproblems (prior to the deletion). Each $S \in \mathcal{S}_i$ contains $\Theta(n/2^i)$ elements, so

$$\Pr[x \in S] = \Theta\left(\frac{1}{2^i}\right).$$

If $x \in S$, then we have by Lemma 10 that S incurs expected rebuild cost

$$O\left(\frac{n/2^i}{|C_S|}\right) = O(\alpha_S^{-1}\sqrt{\log n}).$$

The expected cost from rebuilds in the i-th level of recursion is therefore at most

$$O\left(\sum_{S \in \mathcal{S}_i} \frac{1}{2^i} \cdot \alpha_S^{-1}\sqrt{\log n}\right),$$

which by Lemma 6 is at most

$$O\left(\epsilon^{-1}\sqrt{\log n}\right).$$

Summing over the $O(\log n)$ levels of recursion, the total expected cost of the deletion is $O(\epsilon^{-1}\log^{3/2} n)$. □

Due to the previously described symmetry between insertions and deletions, the same lemma is true for insertions.

Lemma 11. *The expected cost to insert an element x with a random rank in $\{1, 2, \ldots, n+1\}$ into the Zeno embedding is $O(\epsilon^{-1}\log^{3/2} n)$.*

C. Achieving a Bound on Arbitrary Insertions/Deletions.

So far, we have only analyzed random insertions/deletions. At first glance, this may seem like an insignificant accomplishment. (Indeed, it is already known that random insertions/deletions can be supported in $O(\epsilon^{-1})$ amortized time per operation [57].)

What makes the Zeno embedding special is that it is history independent. We will now show how to reduce the list-labeling problem (with *arbitrary* insertions/deletions) to the problem of constructing an insertion/deletion-symmetric history-independent embedding that supports efficient *random* insertions/deletions.

Within any history-independent data structure, the expected cost to perform a deletion at rank r on an array of size m containing n elements can be expressed by a *cost function* $T(m, n, r)$ only dependent on m, n and r. Moreover, if the data structure is insertion/deletion symmetric, then the same cost function T expresses the expected cost for an insertion; specifically, the expected cost to perform an insertion at rank r on an array of size m containing n elements is $T(m, n-1, r)$.

To reduce from the arbitrary insertion/deletion case to the random insertion/deletion case, we will show that given any (insertion/deletion-symmetric) history-independent algorithm $\mathcal{A}$ with cost function $T(m, n, r)$, we can construct a history-independent algorithm $\mathcal{B}$ with cost function $T'(m, n, r)$ such that for each individual rank r, the cost $T'(m, n, r)$ is upper bounded by the *average* of the costs $T(m, n, r)$ across all ranks (up to constant factors).

Lemma 13. *Suppose there is an insertion/deletion-symmetric history-independent algorithm $\mathcal{A}$ whose cost is determined by a function $T(m, n, r)$. Then we can construct a new insertion/deletion-symmetric history-independent algorithm $\mathcal{B}$ with cost function $T'(m, n, r)$ satisfying*

$$T'(m, n, r) = O\left(\frac{1}{m+1}\sum_{j=1}^{2m} T(2m, m+n, j)\right)$$

for all r.

Proof. Fix a history-independent algorithm $\mathcal{A}$. We will construct a history-independent algorithm $\mathcal{B}$. We will describe the behavior of the algorithm $\mathcal{B}$ on an array of size m with an arbitrary sequence $\mathcal{S}$ of insertions/deletions.

To do so, we will construct from $\mathcal{S}$ an input to $\mathcal{A}$. The input to $\mathcal{A}$ is an array of size $2m$ with the following insertion/deletion sequence. First we insert m dummy elements as follows. Let q be a uniformly random integer in $[0, m]$. Insert q dummy elements that are treated as taking infinitely small values (i.e., $-\infty$), and insert $m-q$ dummy elements that are treated as taking infinitely large values (i.e., ∞). Now, execute the sequence $\mathcal{S}$.

Now, define $\mathcal{B}$ as the algorithm that behaves identically to $\mathcal{A}$ on $\mathcal{A}$'s subarray $[q, q+m)$ (that is, $\mathcal{A}$'s subarray from the q^{th} slot to the $q+m^{\text{th}}$ slot), ignoring the dummy elements. That is, for all i, after the i^{th} insertion from $\mathcal{S}$, the subarray $[q, q+m)$ of $\mathcal{A}$'s array with the dummy elements removed, is identical to $\mathcal{B}$'s array.

We note that $\mathcal{B}$ is well defined in the sense that all elements of $\mathcal{S}$ always appear in $\mathcal{A}$'s subarray $[q, q+m)$. This is simply due to the existence of the dummy elements in $\mathcal{A}$'s array.

Now let us bound the expected cost $T'(m, n, r)$ for $\mathcal{B}$ to perform a deletion at rank r. This corresponds to a deletion at rank $r+q$ in $\mathcal{A}$, which has cost $T(2m, m+n, r+q)$. Notice, however, that $r+q$ is a random element in $\{r, r+1, \ldots, r+m\}$. Thus,

$$T'(m, n, r) = \frac{1}{m+1}\sum_{j=r}^{r+m} T(2m, m+n, j),$$

which in turn is at most

$$O\left(\frac{1}{m+1}\sum_{j=1}^{2m} T(2m, m+n, j)\right).$$

□

In the case where $\mathcal{A}$ is the Zeno embedding, we refer to $\mathcal{B}$ as the **shifted Zeno embedding**. Now, we are ready to put everything together and prove our main theorem, that the shifted Zeno embedding incurs expected cost $O(\epsilon^{-1}\log^{3/2} n)$ per insertion/deletion.

Theorem 14. *Let $\epsilon \in (0, 1)$, and suppose $m \geq (1+\epsilon)n$, where m is a static value while n changes dynamically. The shifted Zeno embedding on an array of size m with n elements incurs expected cost $O(\epsilon^{-1}\log^{3/2} n)$ per insertion/deletion.*

Proof. Let $T(m, n, r)$ be the cost function associated with the Zeno embedding, and let $T'(m, n, r)$ be the cost function associated with the shifted Zeno embedding. From Lemma 13, we know that

$$T'(m, n, r) = O\left(\frac{1}{m+1}\sum_{j=1}^{2m} T(2m, m+n, j)\right). \quad (7)$$

The right side of Equation V-C is within a constant factor of the average value of $T(2m, m+n, j)$ over all ranks j. Thus, it is within a constant factor of the expected value of $T(2m, m+n, j)$ where j is chosen uniformly at random over all ranks, which we know from Lemmas 11 and 12, is $O(\epsilon^{-1}\log^{3/2} n)$. Thus, $T'(m, n, r) = O(\epsilon^{-1}\log^{3/2} n)$, as desired. □

The following corollary follows immediately by applying Theorem 14 to an $n(1+\epsilon)$ sized subarray of a linearly sized array for any $\epsilon < 1$.

Corollary 15. *There exists a list-labeling algorithm for an array of size $m = n(1+\Theta(1))$ with expected cost $O(\log^{3/2} n)$ per insertion/deletion.*

We can also use the theorem to bound the total cost to insert into every slot in an array.

Corollary 16. *There exists a list-labeling algorithm to fill an array of size m from empty to full with expected total cost $O(m \log^{3.5} m)$.*

Proof. We will apply a shifted Zeno embedding in $\Theta(\log m)$ phases, using an e_i, defined below, for phase i and rebuilding the array between phases. The first phase consists of the first $m/2$ insertions, and each phase inserts half as many elements as the preceding phase. This continues until $n > m - \log m$, at which point the final phase consists of inserting the remaining at most $\log m$ elements.

More precisely, let $k = \lceil \log m - \log\log m \rceil$, and define

$$s_i = \frac{m(2^i - 1)}{2^i} \quad \text{for } i = 0, 1, \ldots, k,$$

and $s_{k+1} = m$.

Items are inserted by ranks, specified by $r_1, \ldots, r_m$, so that for example, since the first insertion is into an empty array, $r_1 = 1$. Phase P_i is defined by the insertions r_j with $j \in (s_{i-1}, s_i]$. We define $e_i = (2^i - 1)^{-1}$ for $i > 1$, and $e_1 = \frac{m-1}{m}$.

Let $C(P_i)$ denote the expected total cost of the insertions in phase P_i. For $i > 1$ and for all $j \in (s_{i-1}, s_i]$,

$$(1+e_i)j \le (1+e_i)s_i = \left(1 + \frac{1}{2^i - 1}\right)\left(\frac{m(2^i-1)}{2^i}\right) = m.$$

Similarly, in phase P_1, we have

$$(1+e_i)j \le \left(1 + \frac{m-1}{m}\right) \cdot \frac{m}{2} \le m.$$

Therefore, we can apply Theorem 14 to say that for all i, an insertion during phase P_i incurs expected cost

$$O(e_i^{-1} \log^{3/2} s_i) = O(2^i \log^{3/2} m),$$

and

$$C(P_i) = O\left(2^i \cdot \frac{m}{2^i} \cdot \log^{3/2} m\right) = O(m \log^{3/2} m).$$

Summing over the first $k = O(\log m)$ phases, this gives expected total insertion cost $O(m \log^{3.5} m)$.

By construction, the final phase has at most $\log m$ insertions, and thus has total expected cost $O(m \log m)$. Finally, since the total number of elements in the array is bounded by m, the rebuilds between phases incur total cost $O(m \log m)$, completing the proof.

□

We conclude the section with a remark.

Remark 17. *Many applications of list labeling require that, if $m = \Theta(n)$, then the number of empty slots between any two consecutive elements is at most $O(1)$. The Zeno embedding satisfies this property by design, since each subproblem has density at least $1 - \varepsilon/2$. The shifted Zeno embedding therefore also satisfies the same property.*

VI. Acknowledgements

This research was partially sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

This work was also supported by NSF grants CCF-2106999, CCF-2118620, CNS-1938180, CCF-2118832, CCF-2106827, CSR-1763680, CCF-1716252, CNS-1938709. Nicole Wein is supported by a grant to DIMACS from the Simons Foundation (820931). Finally, William Kuszmaul is partially supported by a Hertz Fellowship and an NSF GRFP Fellowship.

References

[1] A. Itai, A. G. Konheim, and M. Rodeh, "A sparse table implementation of priority queues," in *Proc. 8th International Colloquium on Automata, Languages, and Programming (ICALP)*, ser. Lecture Notes in Computer Science, vol. 115, 1981, pp. 417–431.

[2] D. E. Willard, "Maintaining dense sequential files in a dynamic environment (extended abstract)," in *Proc. 20th Annual Symposium on Theory of Computing (STOC)*, 1982, pp. 114–121.

[3] ——, "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)*, 1986, pp. 251–260.

[4] ——, "A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time," *Information and Computation*, vol. 97, no. 2, pp. 150–204, 1992.

[5] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito, "Two simplified algorithms for maintaining order in a list," in *Proc. 10th European Symposium on Algorithms (ESA)*. Springer, 2002, pp. 152–164.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, T. Kopelowitz, and P. Montes, "File maintenance: When in doubt, change the layout!" in *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2017, pp. 1503–1522.

[7] A. Bil and I. Knisel, "Canonical density control," *Inf. Process. Lett.*, vol. 104, no. 6, pp. 203–208, 2007.

[8] A. Andersson and T. W. Lai, "Fast updating of well-balanced trees," in *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, ser. Lecture Notes in Computer Science, J. R. Gilbert and R. G. Karlsson, Eds., vol. 447, July 1990, pp. 111–121. [Online]. Available: https://doi.org/10.1007/3-540-52846-6_82

[9] I. Galperin and R. L. Rivest, "Scapegoat trees," in *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. ACM/SIAM, 1993, pp. 165–174.

[10] M. Babka, J. Bulánek, V. Čunát, M. Koucký, and M. E. Saks, "On online labeling with large label set," *SIAM J. Discret. Math.*, vol. 33, no. 3, pp. 1175–1195, 2019.

[11] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Trans. Database Syst.*, vol. 32, no. 4, pp. 26–26, Nov. 2007.

[12] I. Knisel, "Implicit data structures based on local complexities," Master's thesis, Technion – Israel Inst. of Tech., Haifa, May 2002.

[13] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *Proc. 17th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005, pp. 228–237.

[14] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 341–358, 2005.

[15] P. F. Dietz and J. Zhang, "Lower bounds for monotonic list labeling," in *Scandinavian Workshop on Algorithm Theory*. Springer, 1990, pp. 173–180.

[16] P. F. Dietz, J. I. Seiferas, and J. Zhang, "A tight lower bound for on-line monotonic list labeling," in *Scandinavian Workshop on Algorithm Theory*. Springer, 1994, pp. 131–142.

[17] ——, "A tight lower bound for online monotonic list labeling," *SIAM Journal on Discrete Mathematics*, vol. 18, no. 3, pp. 626–637, 2004.

[18] J. Zhang, "Density control and on-line labeling problems," Ph.D. dissertation, University of Rochester, 1993.

[19] M. Solis, "Online labeling: Algorithms, lower bounds and open questions," in *International Computer Sciences Symposium in Russia (CSR)*, vol. 10846. Springer, 2018, pp. 23–32.

[20] J. Bulánek, M. Koucký, and M. Saks, "Tight lower bounds for the online labeling problem," in *Proc. 44th Annual Symposium on Theory of Computing (STOC)*, 2012, pp. 1185–1198.

[21] M. A. Bender, J. Berry, R. Johnson, T. M. Kroeger, S. McCauley, C. A. Phillips, B. Simon, S. Singh, and D. 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)*, June 2016, pp. 289–302.

[22] W. B. Devanny, J. T. Discenza, M. T. Goodrich, and T. Kopelowitz, "The online bonus numbering problem: Min-max online list labeling," in *Proc. 25th European Symposium on Algorithms (ESA)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[23] P. F. Dietz, "Maintaining order in a linked list," in *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC)*. New York, NY, USA, 1982, pp. 122–127. [Online]. Available: https://doi.org/10.1145/800070.802184

[24] A. Andersson, "Improving partial rebuilding by using simple balance criteria," in *Proc. Workshop on Algorithms and Data Structures (WADS)*, ser. Lecture Notes in Computer Science, vol. 382. Springer, 1989, pp. 393–402.

[25] V. 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)*, 1999, pp. 337–345. [Online]. Available: https://doi.org/10.1145/303976.304009

[26] A. Gál, M. Mahajan, R. Sundaram, and T. Thaler, "Computational complexity of discrete problems (dagstuhl seminar 21121)," in *Dagstuhl Reports*, vol. 11, no. 2. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[27] D. E. Willard, "Inserting and deleting records in blocked sequential files," Bell Labs Tech Reports, Tech. Rep. TM81-45193-5, 1981.

[28] M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton, "Scanning and traversing: Maintaining data for traversals in a memory hierarchy," in *Proc. 10th European Symposium on Algorithms (ESA)*, ser. Lecture Notes in Computer Science, vol. 2461, 2002, pp. 139–151.

[29] G. S. Brodal, R. Fagerberg, and R. Jacob, "Cache oblivious search trees via binary trees of small height," in *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002, pp. 39–48.

[30] M. A. Bender, Z. Duan, J. Iacono, and J. Wu, "A locality-preserving cache-oblivious dynamic dictionary," *Journal of Algorithms*, vol. 53, no. 2, pp. 115–136, 2004.

[31] M. A. Bender, M. Farach-Colton, and H. C. Kuszmaul, "Cache-oblivious string B-trees," in *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. ACM, 2006, pp. 233–242.

[32] B. Wheatman and H. Xu, "A parallel packed memory array to store dynamic graphs," in *Proc. Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2021, pp. 31–45.

[33] ——, "Packed compressed sparse row: A dynamic graph representation," in *HPEC*. IEEE, 2018, pp. 1–7.

[34] B. Wheatman and R. Burns, "Streaming sparse graphs using efficient dynamic sets," in *IEEE BigData*. IEEE, 2021, pp. 224–234.

[35] P. Pandey, B. Wheatman, H. Xu, and A. Buluç, "Terrace: A hierarchical graph container for skewed dynamic graphs," in *Proc. 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2021, pp. 1372–1385.

[36] D. D. Leo and P. A. Boncz, "Teseo and the analysis of structural dynamic graphs," *Proc. VLDB Endowment*, vol. 14, no. 6, pp. 1053–1066, 2021.

[37] ——, "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)*. ACM, 2019, pp. 8:1–8:8.

[38] T. 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)*, 2012, pp. 283–292.

[39] J. Bulánek, M. Koucký, and M. E. Saks, "On randomized online labeling with polynomially many labels," in *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, ser. Lecture Notes in Computer Science, vol. 7965. Springer, 2013, pp. 291–302.

[40] M. Babka, J. Bulánek, V. Cunát, M. Koucký, and M. E. Saks, "On online labeling with polynomially many labels," in *ESA*, ser. Lecture Notes in Computer Science, vol. 7501. Springer, 2012, pp. 121–132.

[41] K. Mehlhorn, P. Sanders, and P. Sanders, *Algorithms and data structures: The basic toolbox*. Springer, 2008, vol. 55.

[42] R. S. Bird and S. Sadnicki, "Minimal on-line labelling," *Inf. Process. Lett.*, vol. 101, no. 1, pp. 41–45, 2007.

[43] D. Micciancio, "Oblivious data structures: applications to cryptography," in *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC)*, 1997, pp. 456–464.

[44] M. Naor and V. Teague, "Anti-persistence: history independent data structures," in *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC)*, 2001, pp. 492–501.

[45] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. C. Rocke, "Characterizing history independent data structures," *Algorithmica*, vol. 42, no. 1, pp. 57–74, 2005.

[46] N. Buchbinder and E. Petrank, "Lower and upper bounds on obtaining history independence," in *Advances in Cryptology*, 2003, pp. 445–462.

[47] G. E. Blelloch and D. Golovin, "Strongly history-independent hashing with applications," in *Proc. 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2007, pp. 272–282.

[48] M. Naor, G. Segev, and U. Wieder, "History-independent cuckoo hashing," in *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 2008, pp. 631–642.

[49] D. Golovin, "B-treaps: A uniquely represented alternative to B-trees," in *Proc. 36th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, 2009, pp. 487–499.

[50] ——, "The B-skip-list: A simpler uniquely represented alternative to B-trees," *arXiv preprint arXiv:1005.0662*, 2010.

[51] J. Nievergelt and E. M. Reingold, "Binary search trees of bounded balance," in *Proc. 4th Annual ACM Symposium on Theory of Computing (STOC)*, 1972, pp. 137–142. [Online]. Available: https://doi.org/10.1145/800152.804906

[52] ——, "Binary search trees of bounded balance," *SIAM Journal on Computing*, vol. 2, no. 1, pp. 33–43, 1973. [Online]. Available: https://doi.org/10.1137/0202005

[53] C. R. Aragon and R. G. Seidel, "Randomized search trees," in *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1989, pp. 540–545.

[54] R. Seidel and C. R. Aragon, "Randomized search trees," *Algorithmica*, vol. 16, no. 4, pp. 464–497, 1996.

[55] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[56] K.-H. Li, "Reservoir-sampling algorithms of time complexity $O(n(1+\log(N/n)))$," *ACM Transactions on Mathematical Software (TOMS)*, vol. 20, no. 4, pp. 481–493, 1994.

[57] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro, "Insertion sort is $O(n \log n)$," *Theory of Computing Systems*, vol. 39, no. 3, pp. 391–397, 2006.