A Nearly Quadratic Improvement for Memory Reallocation

Martin Farach-Colton*
New York University
New York, NY, USA
martin.farach-colton@nyu.edu

William Kuszmaul†
Harvard University
Cambridge, MA, USA
william.kuszmaul@gmail.com

Nathan Sheffield
Massachusetts Institute of Technology
Cambridge, MA, USA
shefna@mit.edu

Alek Westover
Massachusetts Institute of Technology
Cambridge, MA, USA
alekw@mit.edu

ABSTRACT

In the Memory Reallocation Problem a set of items of various sizes must be dynamically assigned to non-overlapping contiguous chunks of memory. It is guaranteed that the sum of the sizes of all items present at any time is at most a $(1-\varepsilon)$-fraction of the total size of memory (i.e., the load-factor is at most $1-\varepsilon$). The allocator receives insert and delete requests online, and can re-arrange existing items to handle the requests, but at a *reallocation cost* defined to be the sum of the sizes of items moved divided by the size of the item being inserted/deleted.

The folklore algorithm for Memory Reallocation achieves a cost of $O(\varepsilon^{-1})$ per update. In recent work at FOCS'23, Kuszmaul showed that, in the special case where each item is promised to be smaller than an ε^4-fraction of memory, it is possible to achieve expected update cost $O(\log \varepsilon^{-1})$. Kuszmaul conjectures, however, that for larger items the folklore algorithm is optimal.

In this work we disprove Kuszmaul's conjecture, giving an allocator that achieves expected update cost $O(\varepsilon^{-1/2}\operatorname{polylog}\varepsilon^{-1})$ on any input sequence. We also give the first non-trivial lower bound for the Memory Reallocation Problem: we demonstrate an input sequence on which any resizable allocator (even *offline*) must incur amortized update cost at least $\Omega(\log \varepsilon^{-1})$.

Finally, we analyze the Memory Reallocation Problem on a stochastic sequence of inserts and deletes, with random sizes in $[\delta, 2\delta]$ for some δ. We show that, in this simplified setting, it is possible to achieve $O(\log \varepsilon^{-1})$ expected update cost, even in the "large item" parameter regime ($\delta > \varepsilon^4$).

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**.

KEYWORDS

Memory Reallocation

ACM Reference Format:
Martin Farach-Colton, William Kuszmaul, Nathan Sheffield, and Alek Westover. 2024. A Nearly Quadratic Improvement for Memory Reallocation. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3626183.3659965

1 INTRODUCTION

In the ***Memory Reallocation Problem*** an ***allocator*** must assign a dynamic set of items to non-overlapping contiguous chunks of memory. Given an set of items with sizes $x_1, x_2, \ldots, x_n$, and given a memory represented by the real interval $[0, 1]$, a ***valid allocation*** of these items to memory locations is a set of locations $y_1, \ldots, y_n \in [0, 1]$ so that the intervals $(y_i, y_i + x_i) \subset [0, 1]$ are all disjoint. As objects are inserted/deleted over time, the job of the allocator is rearrange items in memory so that, at any given moment, there is a valid allocation. The allocator is judged by two metrics: the maximum ***load factor*** that it can support; and the ***reallocation overhead*** that it induces. The allocator is said to support ***load factor*** $1-\varepsilon$ if it can handle an arbitrary sequence of item insertions/deletions, where the only constraint is that the sum of the sizes of the items present, at any given moment, is never more than $1-\varepsilon$; and the allocator is said to achieve ***overhead*** (or ***cost***) c on a given insertion/deletion, if the sum of the sizes of the items that are rearranged is at most a c-factor larger than the size of the item that is inserted/deleted. We remark that all of the allocators in this work will be ***resizable***, meaning that if $L \leq 1-\varepsilon$ is the total size of items present at any time then, then all the items are placed in the interval $[0, L+\varepsilon]$.

The Memory Reallocation Problem, and its variations, have been studied in a variety of different settings, ranging from history independent data structures [5, 9], to storage allocation in databases [4], to allocating time intervals to a dynamically changing set of parallel jobs [2, 3, 6]. The version considered here [3, 5, 9] is notable

*This work was supported in part by NSF grants CNS-2118620 and CCF-2106999.
†William Kuszmaul is funded by the Rabin Postdoctoral Fellowship in Theoretical Computer Science at Harvard University. Large parts of this research were completed while William was a PhD student at MIT, where he was funded by a Fannie and John Hertz Fellowship and an NSF GRFP Fellowship. William Kuszmaul was also 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.

SPAA '24, June, 17-21, 2024, Nantes, France

ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/3626183.3659965

for its choice of cost function: if we model the *time* needed to allocation/deallocate/move an object of size s as $O(s)$, then an overhead of $O(c)$ implies that the total time spent moving objects around is at most an $O(c)$-factor larger than the time spent simply allocating/deallocating objects. The problem of minimizing movement overhead is especially important in systems with many parallel readers, since objects may need to be locked while they are being moved.

Past Work. Most early work on memory allocation focused on the setting in which items *cannot* be moved after being allocated (i.e., the 0-cost case) [7, 10, 11]. However, it is known that such allocators necessarily perform very poorly on their space usage – they cannot, in general, achieve a load factor better than $\widetilde{O}(1/\log n))$ [7, 10, 11]. The main goal in studying memory ***re***allocation [4, 5] is therefore to determine *how much* item movement is necessary to achieve a load factor of $1-\varepsilon$.

The ***folklore algorithm*** [4, 5] for the Memory Reallocation Problem is based on the observation that whenever an item of size k must be inserted we can, by the pigeon-hole principle, find an interval of size $O(k\varepsilon^{-1})$ which has k free space. Thus it is possible to handle inserts at cost $O(\varepsilon^{-1})$ and handle deletes for free.

In recent work at FOCS'23 [5], Kuszmaul shows how to handle the case where all items have size smaller than ε^4 with expected update cost $O(\log \varepsilon^{-1})$. However, Kuszmaul conjectures that, in general, the $O(\varepsilon^{-1})$ folklore bound should be optimal. He proposes, in particular, that the special case in which objects have sizes in the range $(\varepsilon, 2\varepsilon)$ should require $\Omega(\varepsilon^{-1})$ overhead per insertion/deletion.

This Paper: Beating the Folklore Bound. In this work we disprove Kuszmaul's conjecture. In fact, we prove a stronger result: that it is possible to beat the folklore $O(\varepsilon^{-1})$ bound without any constraints on object sizes.

We begin by considering the specialized setting in which items have sizes in the range $(\varepsilon, 2\varepsilon)$—this, in particular, was the setting that Kuszmaul conjectured to be hard. We give in Section 3 a relatively simple allocator that achieves $O(\varepsilon^{-2/3})$ amortized update cost in the case where all items have sizes in $(\varepsilon, 2\varepsilon)$. Although this allocator does not solve the full problem that we care about, it does introduce an important algorithmic idea that will be useful throughout the paper: the idea of having a special small set of items stored as a suffix of memory which are each "responsible" for a large number of items in the main portion of memory. Whenever an item from the main portion of memory is deleted, it gets "replaced" with an item that was responsible for it from the small suffix of memory. By using this notion of responsibility in the right way, we can imbue enough combinatorial structure into our allocation algorithm that it is able to beat the folklore $O(\varepsilon^{-1})$ bound.

The construction of Section 3 is a good start, but does not immediately generalize to handle arbitrary item sizes. In Section 4 we give several new ideas to handle the case of items with sizes in $[\varepsilon^5, 1]$. Then, we show how to combine this allocator with Kuszmaul's allocator from [5] to achieve:

Corollary 4.10. *There is a resizable allocator for arbitrary items with expected update cost* $\tilde{O}(\varepsilon^{-1/2}) = O(\varepsilon^{-1/2}\operatorname{polylog}\varepsilon^{-1})$.

At a high level, the algorithm in Corollary 4.10 takes the basic idea from Section 3 (a small suffix of items that take responsibility for items in the main array), and applies it in a nested structure. This nested "responsibility" structure is not simply a recursive application of the technique—rather, it is carefully constructed so that items of a given size can only appear some levels of the nest. This ends up being what enables us to beat the folklore bound with an arbitrary combination of item sizes.

We conclude the paper with two additional results that are of independent interest. The first is a lower bound, showing that $O(1)$ update cost is not, in general, possible. And the second is an upper bound for a special case where the input sequence is generated by a simple stochastic process.

Until now, the only non-trivial lower bounds for the Memory Reallocation Problem have been for very restricted sets of allocation algorithms [5]. In Section 5, we give a lower bound that applies to any (even offline) allocator. In fact, the update sequence which we use to establish the lower bound is remarkably simple, involving just two item sizes.

Theorem 5.1. *There exist sizes* $s_1, s_2 \in \Theta(\varepsilon^{1/2})$ *and an update sequence* S *consisting solely of items of sizes* s_1, s_2 *such that any resizable allocator (even one that knows* S*) must have amortized update cost at least* $\Omega(\log \varepsilon^{-1})$ *on* S.

Finally, in Section 6, we consider a setting where item arrivals and departures follow a simple stochastic assumption. Define a ***δ-random-item sequence*** as one where memory is first filled with items of sizes chosen randomly from $[\delta, 2\delta]$, and then the allocator receives alternating deletes of random items and inserts of items with sizes chosen randomly from $[\delta, 2\delta]$. In this setting we are able to achieve $O(\log \varepsilon^{-1})$ overhead:

Theorem 6.1. *For any* $\delta = \text{poly}(\varepsilon)$*, there is a resizable allocator that handles* δ*-random-item sequences with worst-case expected update cost* $O(\log \varepsilon^{-1})$.

We note that the algorithm for Theorem 6.1 uses very different techniques from the other algorithms proposed in the paper. In fact, because of this, the algorithm in Theorem 6.1 ends up being quite nontrivial to implement time-efficiently. We give an implementation that decides which items to move in worst-case expected time $O(\varepsilon^{-1/2})$ per update. The time bound is due to a technically interesting lemma about subset sums of random sets.

2 PRELIMINARIES AND CONVENTIONS

We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. For set X and value y we define $y + X = \{y + x \mid x \in X\}$ and $y \cdot X = \{yx \mid x \in X\}$. We use log to denote $\log_2$. We use $|I|$ to denote the size of an item I. The ***total size*** of a set of items is defined to be the sum of their sizes. We will refer to memory as going from left to right, i.e., the start of memory is on the left and the end of memory is on the right.

In the ***Memory Reallocation Problem*** with free-space parameter ε, an ***allocator*** maintains a set of items in memory, which is represented by the interval $[0, 1]$. Memory starts empty, and items are inserted and deleted over time by an oblivious adversary, where the only constraint on the update sequence is that the items present at any time must have total size at most $1-\varepsilon$. The job of an allocator is to maintain a dynamic allocation of items to memory, that is, to assign each item to a disjoint interval whose size equals the item's

size. If the allocator moves L total size of items on an update of size k we say the update is handled at ***cost*** L/k.

We construct allocators that give an extra guarantee: If $L \in [0, 1-\varepsilon]$ is the total size of items present at any time, then a ***resizable allocator*** guarantees that all the items are placed in the interval $[0, L+\varepsilon] \subseteq [0, 1]$.

Our analysis is asymptotic as a function of ε^{-1}. Thus, we may freely assume that ε^{-1} is at least a sufficiently large constant. We use the notation $\widetilde{O}$ to hide polylog(ε^{-1}) factors, and the notation poly(n) to denote $n^{\Theta(1)}$.

3 AN ALLOCATOR FOR LARGE ITEMS

In this section we describe a simple allocator for a special case of the Memory Reallocation Problem, disproving a conjecture of Kuszmaul [5]. We remark that the folklore bound only gives performance $O(\varepsilon^{-1})$ in the regime of Theorem 3.1, i.e., gives no non-trivial bound.

Theorem 3.1. *There is a resizable allocator for items of with sizes in $[\varepsilon, 2\varepsilon)$ that achieves amortized update cost $O(\varepsilon^{-2/3})$.*

Theorem 3.1 offers an *amortized bound*, although, as we shall see in Section 4, it is also possible to obtain a non-amortized expected bound. We remark that there are two notions of amortized cost that one could reasonably consider – if L_i denotes the total-size of items moved to handle the i-th update and k_i is the size of the i-th update, then either of $\frac{1}{n}\sum_{i=1}^{n} L_i/k_i$ or $\sum_{i=1}^{n} L_i / \sum_{i=1}^{n} k_i$ would be a reasonable objective function. Fortunately, in this section, because the k_i's are all equal up to a factor of two, the two objective functions are the same up to constant factors. In later sections where object sizes differ by larger factors, we will go with the convention that guarantees should be worst-case expected rather than amortized.

Proof. We call our allocator SIMPLE. We partition the sizes $[\varepsilon, 2\varepsilon)$ into $\lceil \varepsilon^{-1/3} \rceil$ ***size classes***, where the i-th size class consists of items with size in the range

$$[\varepsilon + (i-1)\varepsilon^{4/3}, \varepsilon + i\varepsilon^{4/3}).$$

Now we describe the operation of SIMPLE; we also provide pseudocode for SIMPLE in Algorithm 1.

Rebuilds. Every $\lfloor \varepsilon^{-1/3} \rfloor$ updates (starting from the first update) SIMPLE performs a ***rebuild***. Let x_i be the number of items of size class i at the time of this rebuild. In a rebuild operation SIMPLE takes the $\min(x_i, \lfloor \varepsilon^{-1/3} \rfloor)$ smallest items from size class i for each $i \in [\lceil \varepsilon^{-1/3} \rceil]$ and groups them into a ***covering set***. SIMPLE arranges memory so that the items are contiguous, left-aligned (i.e., starting at 0), and so that the covering set is a suffix of the present items.

Handling inserts. When an item is **inserted** SIMPLE adds the item to the covering set and places it directly after the final element currently in memory.

Handling deletes. Suppose an item I of size class i is **deleted**. If I is not part of the covering set SIMPLE finds an item I' in the covering set which is also of size class i but with $|I'| \le |I|$. SIMPLE places I' at the location where I used to start and ***logically inflates*** item I' to be of size $|I|$. That is, SIMPLE will consider item I' to be of size $|I|$ until I' is inflated even further or until the next rebuild.

Algorithm 1 SIMPLE Allocator

SIMPLE maintains a suffix of the items called the ***covering set***.
if it has been $\lfloor \varepsilon^{-1/3} \rfloor$ updates since the last rebuild (or it is the first update) **then**
  Perform a rebuild as follows:
  Logically restore items to their original size (i.e., revert any logical inflation of sizes).
  For each $i \in [\lceil \varepsilon^{-1/3} \rceil]$ let x_i be the number of items of the i-th size class.
  Let S be the union over $i \in [\lceil \varepsilon^{-1/3} \rceil]$ of the smallest $\min(x_i, \lfloor \varepsilon^{-1/3} \rfloor)$ items in the i-th size class.
  Arrange items to be contiguous and left-aligned, with items S occurring after the other items.
  Update the covering set to be S.
if an item I is inserted **then**
  Place I immediately after the final item of the covering set and add I to the covering set.
else if an item I is deleted **then**
  if I is not part of the covering set **then**
    Let I' be an item from the covering set of the same size class as I with $|I'| \le |I|$.
    Place I' where I used to start.
    Logically inflate the size of I' to $|I|$.
  Remove I from memory.
  Compact the covering set, arranging its items to be contiguous and flush with the non-covering set.

On each rebuild all items are reverted to their actual size. We say this ***swap*** operation introduces ***waste*** $|I| - |I'| \le \varepsilon^{4/3}$ into memory. Finally, regardless of whether I was in the covering set, SIMPLE ends the delete by removing I from memory and ***compacting*** the covering set, i.e., arranging the covering set items to be contiguous, and left-aligned against the end of the non-covering-set.

Lemma 3.2. *SIMPLE is correct and well-defined.*

Proof. To verify correctness we must show that SIMPLE places items within the allowed space. SIMPLE essentially stores the items contiguously, except for the waste introduced on deletes. Each delete creates waste at most $\varepsilon^{4/3}$: the maximum possible size difference between two items of the same size class. SIMPLE performs a rebuild every $\lfloor \varepsilon^{-1/3} \rfloor$ updates. Thus, the total waste in memory will never exceed

$$\lfloor \varepsilon^{-1/3} \rfloor \cdot \varepsilon^{4/3} \le \varepsilon.$$

Thus, if the total size of items present is L, SIMPLE stores all items in the memory region $[0, L+\varepsilon]$.

To verify that SIMPLE is well-defined we must argue that on every delete of an item outside of the covering set SIMPLE can find a suitable item in the covering set to swap with the deleted item; all other parts of SIMPLE's instructions clearly succeed. Fix a size class i. We consider two (exhaustive) cases for how many items of size class i were placed in the covering set on the previous rebuild, and argue that in either case whenever an item I of size-class i outside the covering set is deleted SIMPLE can find an appropriate item I' in the covering set to swap with I.

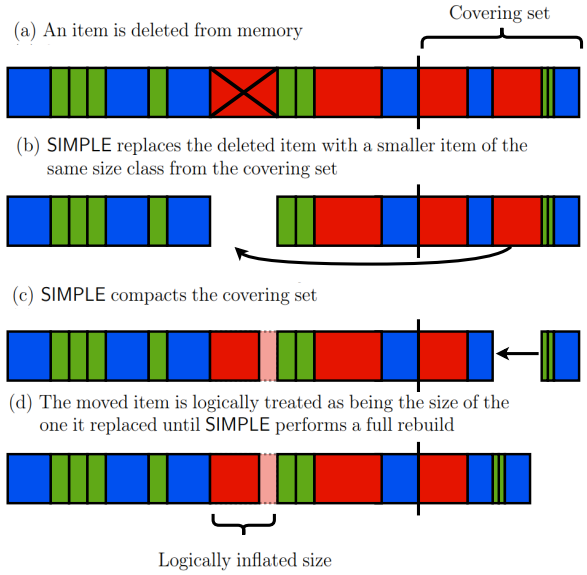

Figure 1: A depiction of SIMPLE handling a delete of an item I outside of the covering set by replacing I with an item I' from the covering set, inflating I' to size $|I|$, and compacting the covering set.

Case 1: The $\lfloor \varepsilon^{-1/3} \rfloor$ smallest items of size class i were placed in the covering set on the previous rebuild; call this set of items S_i. Then, because SIMPLE performs rebuilds every $\lfloor \varepsilon^{-1/3} \rfloor$ updates and because SIMPLE swaps at most one of the items from S_i out of the covering set on each delete we have that on any delete before the next rebuild there is always an element of S_i contained in the covering set. The items in S_i were chosen to be the smallest items of size class i at the time of the previous rebuild. Recall that inserted items are added to the covering set. Thus, we maintain the invariant that all items I of size class i outside of the covering set have (logical) size at least the size of any element in S_i. Thus, there is always an appropriate covering set item to swap with any deleted item of size class i outside of the covering set.

Case 2: If we are not in Case 1, then during the previous rebuild there were fewer than $\lfloor \varepsilon^{-1/3} \rfloor$ total items of size class i, and SIMPLE placed *all of these items* in the covering set. This property, that all items of size class i are contained in the covering set, is maintained until the next rebuild because inserted items are added to the covering set. Thus, until the next rebuild there is *never* a delete of an item of size class i outside of the covering set: no such items exist. So the condition we desire to hold on such deletes is vacuously true.

□

Lemma 3.3. *SIMPLE has amortized update cost $O(\varepsilon^{-2/3})$.*

Proof. The covering set has size at most $2\varepsilon \cdot \lceil \varepsilon^{-1/3} \rceil \cdot 2\lfloor \varepsilon^{-1/3} \rfloor \leq O(\varepsilon^{1/3})$. This is because all items have size at most 2ε, the number of size classes is $\lceil \varepsilon^{-1/3} \rceil$, and the number of items of each size class in the covering set starts as at most $\lfloor \varepsilon^{-1/3} \rfloor$ and then increases by at most one per update during the $\lfloor \varepsilon^{-1/3} \rfloor$ updates between rebuilds, and hence the number of items of each size class in the covering set never exceeds $2\lfloor \varepsilon^{-1/3} \rfloor$. We compact the covering set on each update and so incur cost $O(\varepsilon^{1/3}/\varepsilon) \leq O(\varepsilon^{-2/3})$ per update. Rebuilds incur cost at most $1/\varepsilon$, and occur every $\lfloor \varepsilon^{-1/3} \rfloor$ steps. Thus, their amortized cost is at most $\varepsilon^{-1}/\lfloor \varepsilon^{-1/3} \rfloor \leq O(\varepsilon^{-2/3})$. Overall, SIMPLE's amortized update cost is $O(\varepsilon^{-2/3})$. □

□

4 AN ALLOCATOR FOR ARBITRARY ITEMS

Theorem 3.1 gives a surprising and simple demonstration that the folklore bound is not tight in the large items regime. In this section we will show how to outperform the folklore algorithm for arbitrary items, which is substantially more difficult than Theorem 3.1. In [5] Kuszmaul has already shown how to outperform the folklore algorithm in the regime where items are very small. In Section 4.2 we show that Kuszmaul's allocator can be combined with any resizable allocator fairly easily, to even get a resizable allocator. Thus, the main difficulty we address in this section is extending Theorem 3.1's allocator SIMPLE to work on items with sizes in the interval $[\varepsilon^5, 1]$. There are two major obstacles not present in SIMPLE that arise when handling items with sizes that can differ by factor of $\text{poly}(\varepsilon)$.

The first challenge is that SIMPLE compacts the entire covering set on every delete. The covering set needs to be large enough to contain a substantial quantity of items of each size class. Large items, e.g., of size close to $\varepsilon^{1/2}$ can potentially afford to compact the covering set each time they are the subject of an update. However, it would be catastrophic if updates of smaller items, e.g., items of size ε^3 caused the entire covering set to be compacted each time. In fact, the situation is even more troublesome: we hope to improve SIMPLE's update cost of $O(\varepsilon^{-2/3})$ to $\widetilde{O}(\varepsilon^{-1/2})$. Thus, even items of size $\Theta(\varepsilon)$ cannot afford to compact the entire covering set on each update if the covering set is large. And, in order to make rebuilds infrequent it seems like we must make the covering set quite large.

The second challenge is that SIMPLE breaks items into size classes, which are groups of items whose sizes differ by at most $\varepsilon^{4/3}$. The small multiplicative range of item sizes that we assume in Theorem 3.1 ensures that the number of size classes will be small. However, we cannot use the same style of size classes once the item sizes can vary by a factor of ε^5: there would be far too many size classes. In order to support a larger range of item sizes, we modify our size classes to be ***geometric***. That is, we define size classes of the form $[\delta(1+\alpha)^{i-1}, \delta(1+\alpha)^i]$ instead of $[\delta + \alpha(i-1), \delta + \alpha i]$ for some $\alpha = \text{poly}(\varepsilon)$. However, geometric size classes cause a major complication absent in the fixed-stride size class approach of SIMPLE. Namely, with geometric size classes large items waste more space than small items per delete. Thus, a naive approach of rebuilding whenever the wasted space exceeds ε would be susceptible to the following vulnerability: a few deletes of large items could waste a lot of space, but then the rebuild could be triggered by a small item. But the rebuild is very expensive when triggered by a small item.

We now introduce a construction to address these issues.

4.1 Handling Items with Sizes in $[\varepsilon^5, 1]$

Theorem 4.1. *There is a randomized resizable allocator for items of size at least ε^5 that achieves worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.*

Proof. We call our allocator GEO. GEO labels an item as ***huge*** if it has size at least $\varepsilon^{1/2}/100$. Whenever a huge item I is inserted or deleted GEO rearranges all of memory so that all huge items are compacted together at the start of memory. The cost of each such operation is $O(\varepsilon^{-1/2})$. Thus, we may assume without loss of generality that there are no huge items. Assume that ε^{-1} is a power of 4. This is without loss of generality up to decreasing ε by at most a factor of 4.

Let $\beta = 1 + \varepsilon^{1/2}$. GEO classifies the non-huge items into $C \le O(\varepsilon^{-1/2} \log \varepsilon^{-1})$ size classes. Specifically, an item is classified as part of the i-th size class if it has size in the interval $[\varepsilon^5 \beta^{i-1}, \varepsilon^5 \beta^i)$. GEO builds a sequence of $\ell = 4.5 \log \varepsilon^{-1}$ ***covering levels*** [1] – nested suffixes of memory with geometrically decreasing sizes. In particular, if an item I is in level j, we also consider I to be in each level $j' < j$. For each $j \in [\ell]$ the ***mass limit*** for each size class in level j is defined to be

$$m_j = 2^{\ell - j + 1} \varepsilon^5.$$

We will ensure the ***level size invariant***: for all $j \in [\ell], i \in [C]$ the total size of items of size class i in level j is at most $2m_j$. In particular, this will mean that the total size of level j is at most $2Cm_j$. Note that $m_\ell = 2\varepsilon^5$: the deepest level can fit only $O(1)$ of even the smallest items. Also note that

$$m_1 = 2^\ell \varepsilon^5 = 2^{4.5 \log \varepsilon^{-1}} \varepsilon^5 = \varepsilon^{1/2},$$

so level 1 can fit at least $\Omega(1)$ of even the largest items. For convenience we will also define the 0-th level to mean all of memory with $m_0 = 1$.

For each $i \in [C]$, let s_i denote the total number of items of size class i; this number will change as items are inserted and deleted. Let $b_i = \varepsilon^5 \beta^i$: all items of size class i have size smaller than b_i. For each $i \in [C], j \in [\ell]$ the number of items of size class i in level j will always be at most twice the quantity

$$c_{i,j} = \lfloor m_j / b_i \rfloor.$$

For convenience we also define $c_{i,0} = \infty$ for each $i \in [C]$.

We now describe GEO.

Level rebuilds. For each $i \in [C]$, define j_i^* to be the largest level $j \in [\ell]$ such that $c_{i,j} \ge 1$; j_i^* is the deepest level that could feasibly contain an item of size class i. In fact we will have $c_{i,j_i^*} = 1$, because the mass limit for any levels $j, j+1$ differ by a factor of 2, and because the mass limit in level ℓ is such that level ℓ fits at most 1 of any size class. For each $i \in [C], j \in [j_i^*]$ GEO keeps ***insert/delete level rebuild thresholds*** $r_{i,j}, r'_{i,j} \in [\lceil c_{i,j}/4 \rceil, \lceil c_{i,j}/3 \rceil] \cap \mathbb{N}$. GEO initializes the level rebuild thresholds uniformly randomly from this range.

Updates will sometimes cause ***level rebuilds***. To simplify the description of our allocator it is also useful to have a concept of a ***free rebuild*** (a type of rebuild). A free rebuild is a "sentinel value": it is only a logical operation and has zero cost. At the very start GEO performs a free rebuild of each level j by each size class i. We describe a level rebuild caused by an insert; level rebuilds caused by deletes are completely symmetric. Suppose an item of size class i_0 is inserted. For each $j \in [\ell]$, let t_j denote the number of inserts since the previous time that level j has been rebuilt (including free rebuilds) by a size class i_0 item. Let j_0 be the smallest $j \in [j_{i_0}^*]$ such that $t_j \ge r_{i_0,j}$ (in fact, we will have $t_{j_0} = r_{i_0,j_0}$).

GEO then ***rebuilds*** level j_0. For each $j \in [\ell], i \in [C]$ define $\mathcal{I}_j^{(i)}$ to be the $\min(s_i, c_{i,j})$ smallest items of size class i, and define

$$\mathcal{I}_j = \bigcup_{i \in [C]} \mathcal{I}_j^{(i)}.$$

Define $\overline{\mathcal{I}_j}$ to be all items except for items $\mathcal{I}_j$. To rebuild, GEO rearranges level $j_0 - 1$ to ensure that for all $j \ge j_0$ the items $\mathcal{I}_j$ appear to the right of items $\overline{\mathcal{I}_j}$. This arrangement is well-defined since for each j we have $\mathcal{I}_{j+1} \subseteq \mathcal{I}_j$. We justify in Lemma 4.2 why GEO can always find any such $\mathcal{I}_j$ as a subset of level $j_0 - 1$, and so achieve this arrangement by rearranging only level $j_0 - 1$. GEO labels the items $\mathcal{I}_j$ as level j for all $j \ge j_0$.

Let J be the set of all levels $j \in [j_i^*]$ such that $t_j \ge r_{i_0,j}$. To finish the rebuild of level j_0 GEO resamples $r_{i_0,j}$ randomly from $[\lceil c_{i,j}/4 \rceil, \lceil c_{i,j}/3 \rceil] \cap \mathbb{N}$ for each $j \in J$. GEO considers this a free rebuild for levels $j \in J \setminus \{j_0\}$ by the size class i_0 item.

Algorithm 2 Rebuild on an insert of item I

1: Let i_0 denote the size class of item I.
2: For each $j \in [\ell]$, let t_j denote the number of inserts since the previous time that level j has been rebuilt (including free rebuilds) by a size class i_0 item.
3: Let j_0 be the smallest $j \in [j_{i_0}^*]$ such that $t_j \ge r_{i_0,j}$.
4: For each $j \in [\ell], i \in [C]$ define $\mathcal{I}_j^{(i)}$ to be the $\min(s_i, c_{i,j})$ smallest items of size class i.
5: For $j \in [\ell]$ define $\mathcal{I}_j = \bigcup_{i \in [C]} I_j^{(i)}$ for all $j \in [\ell]$.
6: **Assert:** items $\mathcal{I}_j$ are present in level $j - 1$
7: **for** $j \leftarrow j_0, j_0 + 1, \ldots, \ell$ **do**
8: Arrange level $j - 1$ so that items $\mathcal{I}_j$ are on the right, and other items are on the left.
9: Label items $\mathcal{I}_j$ as level j.
10: Let J be the set of all levels $j \in [j_i^*]$ such that $t_j \ge r_{i_0,j}$.
11: Resample $r_{i_0,j}$ randomly from $[\lceil c_{i,j}/4 \rceil, \lceil c_{i,j}/3 \rceil] \cap \mathbb{N}$ for each $j \in J$.
12: GEO considers this a free rebuild for levels $j \in J \setminus \{j_0\}$ by the size class i_0 item.

Handling Inserts. GEO handles inserts as follows: Place inserted items directly after the current final item in memory. When an item of size class i is inserted we add it to level ℓ. As discussed earlier, inserts trigger level rebuilds when level rebuild thresholds are reached.

Algorithm 3 Inserts

1: Place I immediately after the final item of level ℓ.
2: Perform necessary level rebuilds.

[1] Note that $\ell \in \mathbb{N}$ by our assumption that ε^{-1} is a power of 4.

Handling deletes. Suppose an item I of size class i is deleted. If item I is not in level j_i^* GEO finds the item I' of size class i in level j_i^* which will have $|I'| \leq |I|$ and ***swaps*** I, I'; in Lemma 4.2 we argue that there is some such item I'. To swap items I and I' GEO places item I' where item I used to be. Next, GEO ***inflates*** the size of item I' to $|I|$. That is, GEO will logically consider item I' to have size $|I|$ until the next ***waste recovery*** step at some later time (or until I' is further inflated). We describe the waste recovery procedure after finishing the description of how GEO handles deletes.

After swapping item I (if necessary) and removing I from memory GEO ***compacts*** level j_i^*, i.e., arranges the items of level j_i^* to be contiguous and left-aligned with the final element that is not part of level j_i^* (or left-aligned with 0 if all elements are part of the level). As discussed earlier a delete triggers level rebuilds when level rebuild thresholds are reached.

Algorithm 4 Delete item I

Input: `waste` so far and waste recovery threshold T.

1: Remove item I from memory.
2: Let i be the size class of item I.
3: Let j_i^* be the largest j such that $c_{i,j} \geq 1$, i.e., so that I fits in level j.
4: **Assert:** the smallest item of size class i is guaranteed to be in level j_i^*.
5: **if** item I is not in level j_i^* **then**
6: Find the size class i item I' in level j_i^*.
7: Place item I' where item I used to be.
8: Logically inflate the size of I' to be $|I|$.
9: Let b_i be the maximum size of an item of size class i.
10: `waste` $\leftarrow$ `waste` $+ \varepsilon^{1/2} b_i$.
11: Compact level j_i^*.
12: Perform necessary level rebuilds.
13: **if** `waste` $> T$ **then**
14: Perform a waste recovery step.

Implementing waste recovery. When handling deletes GEO performs swaps which cause ***waste***. Suppose GEO swaps items I, I' both of size class i, and let b_i be the maximum size of an item in size class i. Define $w_i = \varepsilon^{1/2} b_i$. Then, $|I| - |I'| \leq b_i - b_i/\beta \leq w_i$. We say that the swap causes waste w_i. GEO's waste recovery steps will ensure that the total waste in memory never exceeds ε. This will ensure that the total size of gaps introduced by swaps never exceeds ε.

We consider GEO to have performed a free waste recovery step at the beginning (this is a logical operation incurring zero cost, useful as a sentinel value). At every waste recovery step (and at the beginning) GEO samples threshold $T \leftarrow (\varepsilon/2, \varepsilon)$ uniformly to determine how much waste to allow before triggering the next waste recovery step. More precisely, (excluding the free waste recovery step at the beginning) if the waste recovery threshold was T and the most recent delete would cause the waste introduced since the previous waste recovery step to be $W \geq T$ then GEO performs a waste recovery step. We consider the waste at the start of this waste recovery step to be $W - T$: that is, waste from the final delete which caused the waste recovery step ***overflows*** to count towards

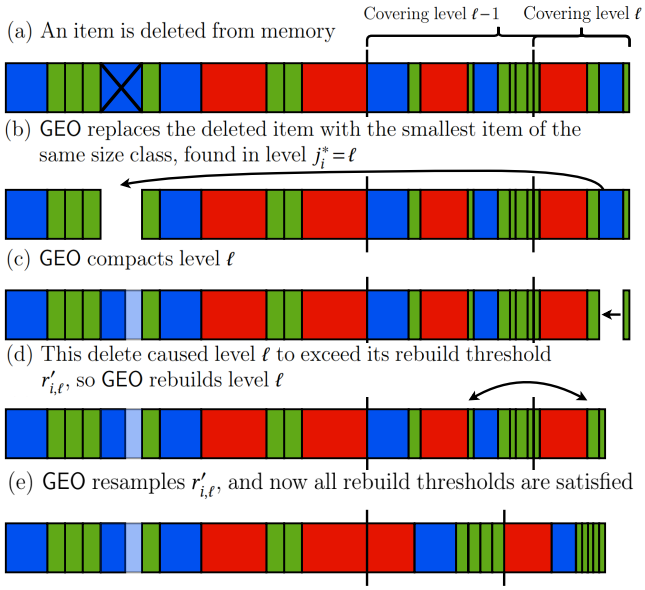

Figure 2: GEO handling a delete.

the next waste recovery step. To perform the waste recovery step GEO logically reverts all items to their original sizes, arranges the items to be contiguous and left-aligned with 0, and then rebuilds level 1.

Algorithm 5 Waste Recovery Step

1: Revert all logical changes to item sizes.
2: Compact all items to be contiguous and left-aligned.
3: Rebuild level 1.
4: Let W be the waste since the previous waste recovery step.
5: `waste` $\leftarrow W - T$.
6: Resample waste recovery threshold $T \leftarrow (\varepsilon/2, \varepsilon)$.

GEO is depicted in Figure 2. Now we analyze GEO.

Lemma 4.2. *GEO is well-defined and correct (i.e., allocates items within the allowed space).*

Proof. First we show that the level size invariant is maintained. This follows from the following stronger property: for all $i \in [C], j \in [\ell]$ there are at most $2c_{i,j}$ items of size class i in level j. First note that this is sufficient to prove the level size invariant because $2c_{i,j}$ items of size class i take up at most $2m_j$ space. Now we argue that the rebuild procedure maintains this stronger property. For all $i \in [C], j > j_i^*$, whenever an item of size class i is inserted some level $j \in [j_i^*]$ is rebuilt, and so no items of size class i can remain in level j because $c_{i,j} = 0$. For $i \in [C], j \in [j_i^*]$, the insert level rebuild threshold $r_{i,j}$ satisfies $r_{i,j} \leq c_{i,j}$. That is, level j will be rebuilt before there are more than $c_{i,j}$ inserts of size class i items, and thus level j can never have more than $2c_{i,j}$ size class i items.

To show that GEO is correct, we need to verify that after every update for every $j_0 \in [\ell], j \geq j_0$, items $\mathcal{I}_j$ are contained in level $j_0 - 1$. This is necessary for GEO's rebuild operation to be well-defined. Because for each j we have $\mathcal{I}_{j+1} \subseteq \mathcal{I}_j$ it suffices to show that for

each $j \in [\ell]$ the items $\mathcal{I}_j$ are contained in level $j-1$. Recalling the definition of $\mathcal{I}_j$ our goal is to show that for all $i \in [C]$, $j \in [\ell]$ the $\min(s_i, c_{i,j})$ smallest items of size class i are contained in level $j-1$. Fix some size class i. First, observe that for $j > j_i^*$ we have $c_{i,j} = 0$, so the claim is vacuously true. We prove the claim for $j \in [j_i^*]$ by induction on j. The claim is clearly true for $j = 1$: level 0 is all of memory, so in particular contains the s_i smallest items of size class i. Assume the claim for $j \in [j_i^* - 1]$, we prove the claim for $j+1$.

Because the claim is true for j we have that whenever level j has just been rebuilt it will contain the $\min(s_i, c_{i,j})$ smallest elements of size class i, because these items were present in level $j-1$. We consider two cases.

Case 1: $c_{i,j} \le 3$. Then $r'_{i,j} = 1$, i.e., level j will be rebuilt every time a size class i item is updated. By our inductive hypothesis rebuilding level j results in the $\min(s_i, c_{i,j}) \ge \min(s_i, c_{i,j+1})$ smallest size class i items being in level j, so the claim holds here.

Case 2: $c_{i,j} > 3$. Then

$$c_{i,j} - \lceil c_{i,j}/3 \rceil \ge \lceil c_{i,j}/2 \rceil \ge c_{i,j+1}.$$

Thus, if the smallest $c_{i,j}$ items of size class i were placed in level j on the previous level j rebuild the smallest $c_{i,j+1}$ items of size class i will still be in level j at all times until the next rebuild. On the other hand, if the smallest s_i items of size class i were placed in level j on the previous level j rebuild then no items of size class i can exit level j until the next level j rebuild: there are no size class i items outside of level j to trigger a swap. Inserts are added to level j so they do not break the invariant. This proves the claim for $j+1$, so by induction the claim is true for all j.

In order for deletions to be well-defined, we must also show that after every update for every size class i with $s_i > 0$, the smallest element of size class i is in level j_i^*. This holds because we always have $r'_{i,j_i^*} = 1$, so every time level j_i^* loses the smallest item of size class i it will be rebuilt, and when it is rebuilt it must have the smallest item because level $j_i^* - 1$ always contains items $\mathcal{I}_{j_i^*}$. All inserted items are inserted to level j_i^*, so again insertions cannot break the invariant.

Now, we argue that GEO always places items within the memory bounds. If we consider items at their inflated (i.e., logical) sizes then the items are contiguous. Recall that the total size of gaps introduced into the array by inflation is bounded by the waste recovery threshold $T < \varepsilon$. Hence, if there is L total size of items present at some time GEO allocates all items in the memory region $[0, L + \varepsilon]$. That is, GEO is resizable.

For completeness we check the fact claimed when defining the size classes, that $C \le \widetilde{O}(\varepsilon^{-1/2})$. Indeed,

$$C \le O(\log_\beta \varepsilon^{-4.5}) \le O\left(\frac{\log \varepsilon^{-1}}{\log(1+\varepsilon^{1/2})}\right) \le O(\varepsilon^{-1/2} \log \varepsilon^{-1}).$$

□

Before analyzing GEO's expected update cost we need two simple lemmas. The proofs are deferred to Appendix A.

Lemma 4.3. *Fix $a, b, W \in \mathbb{R}$ with $0 \le a < b$, and $W > 0$. Let $x_1, x_2, \ldots$ be uniformly and independently sampled from $(W/2, W)$. The probability that there exists j with $\sum_{i \le j} x_i \in [a, b]$ is at most $4(b-a)/W$.*

Lemma 4.4. *Fix integers $y, N \in \mathbb{N}$. Let $x_1, x_2, \ldots$ be uniformly and independently sampled from $[\lceil N/4 \rceil, \lceil N/3 \rceil] \cap \mathbb{N}$. The probability that there exists j with $\sum_{i \le j} x_i = y$ is at most $100/N$.*

Now we analyze the worst-case expected cost of an update. For the remainder of the proof we fix an arbitrary update index $u \in \mathbb{N}$; our goal is to show that the expected cost on update u is small. We break the cost of this update into $\Gamma_W + \Gamma_S + \Gamma_R$, where Γ_W is the cost of waste recovery, Γ_S is the cost of swapping elements and compacting to handle deletes, and Γ_R is the cost of rebuilding levels. We will show $\mathbb{E}[\Gamma_W + \Gamma_S + \Gamma_R] \le \widetilde{O}(\varepsilon^{-1/2})$.

Lemma 4.5. *The expected cost due to waste recovery on update u satisfies $\mathbb{E}[\Gamma_W] \le \widetilde{O}(\varepsilon^{-1/2})$.*

Proof. If update u is an insert then GEO never performs a waste recovery step on update u. Thus, for the purpose of analyzing the cost of waste recovery it suffices to consider the case that update u is a delete. Let update u be the u'-th delete, and let the corresponding deleted item be of size class i. Let $x_1, x_2, \ldots,$ be the sequence of sizes of items that will be deleted. For each k, let w_k be the space wasted by delete k, i.e., the maximum size difference between items in the size class of the k-th deleted item; we have $w_k \le O(\varepsilon^{1/2} x_k)$. GEO repeatedly samples waste recovery thresholds $T_1, T_2, \ldots$ independently from $(\varepsilon/2, \varepsilon)$. A waste recovery step occurs on update u if there exists $M \in \mathbb{N}$ such that update u causes the total waste to cross the M-th waste recovery threshold, i.e., so that

$$\sum_{t=1}^{M} T_t \in \left[\sum_{k=1}^{u'-1} w_k, \sum_{k=1}^{u'} w_k\right].$$

Here we have used the fact that waste *overflows* between waste recovery steps. By Lemma 4.3 the probability that such an M exists is at most $4w_{u'}/\varepsilon$. If u must perform waste recovery the cost is at most $1/x_{u'}$. Thus, the expected cost of waste recovery on delete u' is at most

$$\frac{4w_{u'}}{\varepsilon}\frac{1}{x_{u'}} \le O(\varepsilon^{-1/2}).$$

□

Lemma 4.6. *The cost due to swapping and compacting on update u satisfies $\Gamma_S \le \widetilde{O}(\varepsilon^{-1/2})$.*

Proof. When an item I of size class i is deleted GEO potentially moves an item I' also of size class i to replace item I. This costs $O(1)$. After removing item I from memory GEO must compact level j_i^*. The cost of this compaction is bounded by the maximum possible size of level j_i^* divided by $|I|$. The size of level j_i^* is at most $2Cm_{j_i^*}$ by the level size invariant. We claim $|I| \ge m_{i,j_i^*}/4$. If $j_i^* < \ell$ but $|I| \le m_{i,j_i^*}/4$ then I's size class can fit on a deeper level than j_i^*, contradicting the definition of j_i^*. If $j_i^* = \ell$ then the inequality is true because $m_{i,\ell}/4$ is smaller than the minimum item size. Thus, the cost of compacting level j_i^* is at most

$$\frac{2Cm_{j_i^*}}{|I|} \le O(C) \le \widetilde{O}(\varepsilon^{-1/2}).$$

Note that there is zero cost here on an insert. □

Lemma 4.7. *The expected cost due to rebuilding levels on update u satisfies $\mathbb{E}[\Gamma_R] \le \widetilde{O}(\varepsilon^{-1/2})$.*

Proof. There are only $\ell = \Theta(\log \varepsilon^{-1})$ levels. Thus, it suffices to fix a level $j \in [\ell]$ and show that the expected cost due to rebuilding level j on update u is at most $O(C) \leq \widetilde{O}(\varepsilon^{-1/2})$. Fix $j \in [\ell]$ and let update u be an item I of size class $i \in [C]$. First, note that if $j > j_i^*$ level j is never rebuilt by an item of size class i. So, we may assume $j \in [j_i^*]$. We claim the probability that update u triggers a rebuild of level j is at most $100/c_{i,j}$. Suppose that update u is an insert; the case of deletes is symmetric. Let u be the u'-th insert of a size class i item. Let the sequence of insert rebuild thresholds $r_{i,j}$ for level j on items of size class i chosen by GEO be $x_1, x_2, \ldots$. Recall that these are sampled from $[\lceil c_{i,j}/4 \rceil, \lceil c_{i,j}/3 \rceil] \cap \mathbb{N}$. Then, the probability of update u triggering a rebuild of level j is precisely the chance that there is some k^* such that $\sum_{k \leq k^*} x_k = u'$. This is exactly the situation described in Lemma 4.4. Thus, the probability that u triggers a rebuild of level j is at most $100/c_{i,j}$ in this case.

If update u triggers a rebuild of level j the cost is at most $2Cm_j/|I|$ (and may even be 0 in the case that it was a free rebuild, i.e., covered by a larger level's rebuild). Thus, the expected cost of rebuilding level j on update u is at most

$$\frac{200Cm_j}{c_{i,j}|I|}. \tag{1}$$

Recall the definition of $c_{i,j}$: if b_i denotes the maximum possible size in size class i then $c_{i,j} = \lfloor m_j/b_i \rfloor$. Thus, because $c_{i,j} \geq 1$ we have

$$c_{i,j} \cdot |I| \geq c_{i,j} b_i/\beta = \lfloor m_j/b_i \rfloor b_i/\beta \geq m_j/(2\beta) \geq m_j/4.$$

This shows that (1) is bounded by $O(C)$. □

Thus, the expected cost of update u is at most

$$\mathbb{E}[\Gamma_S + \Gamma_W + \Gamma_R] \leq \widetilde{O}(\varepsilon^{-1/2}).$$

□

4.2 Combining GEO with Kuszmaul's Allocator

Throughout the subsection we say that an item is ***large*** if it has size larger than ε^4, and ***tiny*** otherwise. In Theorem 4.1 we described the GEO allocator which can handle large items. In [5] Kuszmaul constructed an allocator based on min-hashing, which we call TINYHASH, that can handle tiny items with worst-case expected update cost $O(\log \varepsilon^{-1})$. Kuszmaul's TINYHASH is even a resizable allocator, like GEO. Combining GEO and TINYHASH immediately yields:

Corollary 4.8. *There is an allocator for arbitrary items with worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.*

Proof. Instantiate GEO with $\varepsilon/3$ free space starting at the beginning of memory and instantiate TINYHASH with $\varepsilon/3$ free space, but starting at the end of memory and growing backwards. When we get an update of a tiny item we send the update to TINYHASH, and when we get an update of a large item we send the update to GEO. The correctness of this approach follows from the fact that TINYHASH and GEO are resizable. In particular, if at some time there is L_1 total size of tiny items present and L_2 total size of large items present then GEO only places items in the memory region $[0, L_1 + \varepsilon/3]$, and TINYHASH only places items in the memory region $[1 - L_2 - \varepsilon/3, 1]$. Because $L_1 + L_2 \leq 1 - \varepsilon$ these intervals are disjoint.

This allocator inherits the max of the worst-case expected update costs in GEO, TINYHASH as its expected update cost. □

In fact, by exploiting the modular structure of TINYHASH, rather than simply using TINYHASH as a black box we can strengthen Corollary 4.10 to obtain the same (asymptotically) update cost, but with a resizable allocator. Now, the layout of memory will be space $[0, L_1 + \varepsilon/2]$ allocated to GEO, where L_1 is the total size of large items and then space $[L_1 + \varepsilon/2, L_1 + L_2 + \varepsilon]$ allocated to TINYHASH where L_2 is the total size of tiny items present. As before, GEO handles large items and TINYHASH handles tiny items. The difference now is that TINYHASH doesn't have a fixed start location: as the region of memory managed by GEO changes size we have to ensure that the region of memory managed by TINYHASH starts right after GEO's memory region ends. That is, in addition to the usual ***internal updates***, we have to modify TINYHASH to support ***external updates***, which are requests of the form "rearrange all of memory to start at a location k ahead or k behind its current start point". Such an external update is considered an operation of "size" k, and a resizable allocator capable of handling external updates is called ***relocatable***. The cost of an external update is the total size L of items moved to handle the external update divided by the size k of the external update. We now show:

Lemma 4.9. *If all internal updates are tiny and all external updates are large, there is a relocatable allocator achieving worst-case expected internal update cost $O(\log \varepsilon^{-1})$, and worst-case expected external update cost $O(1)$.*

Proof. TINYHASH operates by breaking memory into ***slabs***, which are contiguous chunks of memory. Let M denote the largest possible size of a slab. TINYHASH satisfies $M \leq O(\varepsilon^3)$. Slabs have specific sizes and allowed locations. In particular, slabs are of size $M/2^i$ for some $i \in \mathbb{Z}_{\geq 0}$. The start locations of slabs must obey the following ***alignment property***: a slab of size L must be placed at a location $i \cdot L$ for some integer $i \in \mathbb{Z}_{\geq 0}$. In particular, the smaller slabs nest perfectly within the larger slabs. We refer to intervals of the form $[M \cdot i, M \cdot (i+1)]$ for $i \in \mathbb{Z}_{\geq 0}$ as ***memory units***. Because TINYHASH never places items spanning across memory units, rearranging memory units doesn't break TINYHASH's correctness.

We exploit this modular structure of TINYHASH to make a relocatable version of TINYHASH which we call FLEXHASH. FLEXHASH uses $\varepsilon/2$ free space to create a ***buffer***. FLEXHASH partitions the external update sizes $(\varepsilon^4, 1]$ geometrically into $C \leq O(\log \varepsilon^{-1})$ ***update-types***, with the i-th update-type consisting of updates with size in the interval $(2^{i-1}\varepsilon^4, 2^i\varepsilon^4]$. FLEXHASH will use the buffer to "hide" the external updates from TINYHASH, which it will run as a subroutine. FLEXHASH reserves the remaining $\varepsilon/2$ free space for the normal execution of TINYHASH.

FLEXHASH splits the buffer into C parts, one for each update-type. We use the term ***central memory*** to refer to the region of memory in which TINYHASH will operate. For each $i \in [C]$ define variable B_i. B_i stores how much of update-type i's portion of the buffer has been used. FLEXHASH will maintain as an invariant that $B_i \in [0, 16M]$ for all $i \in [C]$ at all times. Furthermore, FLEXHASH will guarantee that the distance between the start of central memory and the actual start of all of memory is at most $\sum_{i \in [C]} B_i$. Let s denote the number of memory units that exist at some time. Let the

i-th memory unit denote the memory unit that TINYHASH places at location iM. Let Δ denote the starting location of the memory region assigned to FLEXHASH. Then, there is some permutation $\pi : [s] \to [s]$ such that FLEXHASH places the i-th memory unit starting at location $\Delta + \sum_{j \in [C]} B_j + \pi_i \cdot M$. The contents of memory unit i are identical between reality and the simulation of TINYHASH.

Any action that TINYHASH takes which happens purely within memory units can easily be simulated by FLEXHASH. To complete our description of FLEXHASH we must describe how FLEXHASH handles when TINYHASH creates and deletes memory units, and how FLEXHASH handles external updates.

Handling Resize operations. TINYHASH occasionally must perform ***resize operations*** which delete or create memory units. When TINYHASH creates a new memory unit FLEXHASH also creates a new memory unit, and places it directly after the physically final memory unit currently in its memory.

When a resize operation destroys a memory unit for TINYHASH, it is destroying the final memory unit for TINYHASH, and so does not create a hole in TINYHASH's memory. However, TINYHASH's final memory unit may not correspond to the the physically final memory unit of FLEXHASH. So, destroying this memory unit might cause FLEXHASH to have a large hole in its memory. FLEXHASH fills this hole by swapping its physically final memory unit into the location of the deleted memory unit.

Note that swapping memory units is a quite expensive operation. Fortunately it is at most a constant-factor more expensive than Kuszmaul's resize operations already were. Thus, the memory unit swaps only increase TINYHASH's expected update cost by a constant-factor.

Handling External Updates. Whenever an external update occurs it "pushes" memory to the right or left by its size. When an external update of update-type i and size x occurs we update B_i in the appropriate direction based on if the update pushed memory to the right or left.

First we describe how to handle external updates of items with size at least $M/100$. When an external update of this size occurs, if it does not break the invariant $B_i \in [0, 16M]$ we do nothing. If it does break the invariant FLEXHASH must restore the invariant. FLEXHASH is allowed to increase or decrease B_i by ***rotating*** a memory unit in the appropriate direction, i.e., taking the physically final memory unit and placing it right before the physically first memory unit. Whenever the invariant is violated FLEXHASH rotates memory units until B_i is restored to being within M of $8M$. The cost of doing this on such an external update is $O(1)$.

FLEXHASH handles the smaller external updates by performing ***buffer rebuilds*** whenever there is a sufficient number of external updates of some update-type. Let C' be the largest i such that update-type i consists of updates of size at most $M/100$. In the beginning FLEXHASH chooses random ***rebuild thresholds*** $R_i, R'_i \leftarrow (2M, 4M)$ for each update-type $i \in [C']$. FLEXHASH also initializes counters P_i, P'_i to 0; these store the total amount that memory has been "pushed" in either direction by external updates of update-type i. When an external update of update-type i and size x pushes memory to the right or left FLEXHASH increases P_i or P'_i (respectively) by x. If this causes $P_i > R_i$ or $P'_i > R'_i$ FLEXHASH then performs a ***buffer-i rebuild***. Suppose the buffer-i rebuild was triggered by an external update of size x that pushed memory to the right; the other case (left push) is symmetric. To perform the buffer-i rebuild FLEXHASH rotates memory blocks (as described in the analysis of handling large external updates) to make $B_i \in [7M, 9M]$. Then, we set $P_i \leftarrow P_i - R_i$ (i.e., we *overflow* the unused update size to count towards the next rebuild). Then, we randomly select $R_i \leftarrow (2M, 4M)$. Because we always set $R_i, R'_i < 4M$ and we restore $B_i \in [7M, 9M]$ on each buffer-i rebuild we clearly maintain the invariant $B_i \in [0, 16M]$. It remains to analyze the expected cost per update. Fix some update u of update-type $i \in [C']$ and size x. Applying Lemma 4.3 we find that the probability of update u causing a buffer-i rebuild is at most $O(x/M)$. Hence, the expected external update cost is at most

$$\frac{O(M)}{x} O(x/M) \leq O(1).$$

□

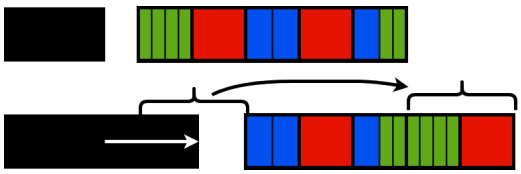

Figure 3: Because TINYHASH decomposes into interchangeable memory-units, we can make TINYHASH relocatable by rotating memory-units to handle external updates.

Using the relocatable allocator FLEXHASH from Lemma 4.9 it is easy to show:

Corollary 4.10. *There is a **resizeable** allocator for arbitrary items with worst-case expected update cost $\widetilde{O}(\varepsilon^{-1/2})$.*

Proof. We instantiate GEO with $\varepsilon/2$ free space starting from 0. We also instantiate the relocatable FLEXHASH from Lemma 4.9 with $\varepsilon/2$ free space, and we maintain the property that FLEXHASH starts after GEO's memory region ends. If there are L_1 total size of large items present and L_2 total size of tiny items present then GEO's memory region is $[0, L_1 + \varepsilon/2]$ and FLEXHASH's memory region is $[L_1+\varepsilon/2, L_1+L_2+\varepsilon]$. We handle tiny items with FLEXHASH and large items with GEO.

Whenever the portion of memory managed by GEO changes size by k (due to an update of size k), we issue an external update of size k to FLEXHASH in the appropriate direction. The cost of an external update is defined precisely so that if FLEXHASH handles this external update at cost x then the total size of items moved by FLEXHASH is $O(kx)$. Thus, the actual cost of this update is $O(x)$ as well. Hence, on any update the expected cost due to handling external updates is $O(1)$. The expected cost due to updates handled by GEO is at most $\widetilde{O}(\varepsilon^{-1/2})$, and the expected cost of internal updates for FLEXHASH is $O(\log \varepsilon^{-1})$. Thus, our allocator's expected cost is $\widetilde{O}(\varepsilon^{-1/2})$. □

5 A LOWER BOUND

In this section we give the first non-trivial lower bound for the reallocation problem using a surprisingly simple update sequence.

Theorem 5.1. *There exist sizes $s_1, s_2 \in \Theta(\varepsilon^{1/2})$ and an update sequence S consisting solely of items of sizes s_1, s_2 such that any resizable allocator (even one that knows S) must have amortized update cost at least $\Omega(\log \varepsilon^{-1})$ on S.*

Proof. Without loss of generality assume $\varepsilon^{-1/2} \in 4\mathbb{N}$, and let $n = (\varepsilon^{-1/2})/4$. We call the items of size s_1 ***A's*** and the items of size s_2 ***B's***. Set $s_1 = \varepsilon^{1/2} + 2\varepsilon$ and $s_2 = \varepsilon^{1/2}$. The sequence S is as follows: First, insert n A's. Then, for n iterations, delete an A and insert a B.

Consider an allocator operating on S. We will think of the allocator's experience as follows. Every ***step*** the allocator must rearrange memory such that it ends with an A. Then, that A is turned into a B. This is without loss of generality because a resizable allocator cannot afford to leave a gap of size s_1 in memory after an A is deleted. Let the "i-th item" denote the i-th item counting from the end of memory. For $i \in [n]$ let B_i denote the number of B's among the final i items of memory. Define potential function (which we only measure when there are n items in memory, i.e., at the start of each step):

$$\Phi = \sum_{i=1}^{n} \frac{B_i}{i}.$$

Whenever an A at the end of memory is turned into a B, each B_i increases by 1, so Φ increases by $\sum_{i=1}^{n} 1/i \geq \Omega(\log n)$.

Now we analyze how much the allocator can change Φ by performing x work. We claim that the allocator's rearrangement can be ***decomposed*** into "***full permutations***", operations of the form: pick $i, j \in [n]$ and for each $k \in [i, j] \cap \mathbb{N}$ assign item k a new location. Clearly the cost of such an operation is $\Omega(j - i)$. Intuitively this decomposition is possible because s_1, s_2 were constructed to have no additive structure: for any $\lambda_1, \lambda_2 \in [0, n] \cap \mathbb{Z}$ not both 0 we have $|\lambda_1 s_1 - \lambda_2 s_2| \geq 2\varepsilon$. Now we show how to decompose the allocator's rearrangement into full permutations. Fix $i, j \in [n]$ such that the allocator moves item k for each $k \in [i, j]$, but does not move item k' for $k' \in \{i - 1, j + 1\} \cap [n]$. Let x_1 be the location where item $j + 1$ ends (set $x_1 = 0$ if $j = n$) and let x_2 be the location where item $i - 1$ starts (set $x_2 = 1$ if $i = 1$). Suppose that there are a A's and b B's in the memory region $[x_1, x_2]$ to start, and a' A's and b' B's in this memory region after the the rearrangement. Note that there are no items only partially in $[x_1, x_2]$ before or after the re-arrangement due to the assumption that the items immediately on either side of the interval (or the endpoints of memory if no such items exist) do not move. As argued above, if $(a, b) \neq (a', b')$ then $|(a - a')s_1 + (b - b')s_2| \geq 2\varepsilon$. A resizable allocator is not allowed to have more than an ε gap anywhere in memory, so this would be an invalid rearrangement. Hence we must have $(a, b) = (a', b')$. And then the allocator can simply rearrange the items within items $[i, j]$ rather than taking items from outside of $[i, j]$. Thus, we can decompose any set of rearrangements into full permutations.

Now, consider the potential change caused by a full permutation that moves x items. This operation only changes the B_i values for x items. Thus, because $B_i/i \leq 1$ for all i, the operation decreases Φ by at most x. This operation requires at least $x/2$ work. In summary, the allocator requires at least $x/2$ work to decrease Φ by x.

We have shown a sequence of $3n$ updates such that, over the course of the whole update sequence, Φ must increase by $\Omega(n \log n)$. Since the potential starts at 0 and is always at most n, the allocator must have amortized cost at least

$$\frac{1}{2} \frac{\Omega(n \log n) - n}{3n} \geq \Omega(\log n) \geq \Omega(\log \varepsilon^{-1}).$$

□

6 AN ALLOCATOR FOR ITEMS WITH RANDOM SIZES IN $[\delta, 2\delta]$

In this section we consider allocators for ***random items***, i.e., items with uniformly random sizes in some range $[\delta, 2\delta]$. In this setting we are able to create an allocator with substantially better performance than the allocators of Section 4.

Fix $\delta = \text{poly}(\varepsilon)$. A δ-***random-item sequence*** is the following sequence of updates: The first $\lfloor \delta^{-1}/4 \rfloor$ updates are inserts of items with sizes chosen randomly from $[\delta, 2\delta]$. Then, the sequence alternates between a deletion of a random item and an insertion of an item with size chosen randomly from $[\delta, 2\delta]$. Note that there will always be (within 1 of) $\lfloor \delta^{-1}/4 \rfloor$ items present. Our main result of this section is:

Theorem 6.1. *There is a randomized resizable allocator that handles δ-random-item sequences with worst-case expected update cost $O(\log \varepsilon^{-1})$. Furthermore, the set of items that our allocator moves to handle an update can be computed in expected time $O(\varepsilon^{-1/2})$.*

Note that in this stochastic setting where the total size of items present is variable the resizable guarantee of our allocator is the most natural property to hope for. To prove Theorem 6.1, the following property of δ-random-item sequences is quite useful: After $d \geq \lfloor \delta^{-1}/4 \rfloor$ updates the distribution of items sizes present is the same distribution as obtained by sampling $\lfloor \delta^{-1}/4 \rfloor$ (or $\lfloor \delta^{-1}/4 \rfloor + 1$ depending on the parity of d) values independently from $[\delta, 2\delta]$.

Our allocator for random items is based on the observation that random independent values can make many subset sums. The subset sums of random sets have been studied before (see, e.g., [8]). However, to the best of our knowledge previous work has only given an asymptotic version of the result we need, namely Theorem 6.2. Our self-contained analysis explicitly determines the constant-factor for how large a random set has to be in order to contain a subset of a desired sum with constant probability. This is important for our application because the constant-factor appears as an exponent in the running time of our algorithm.

In what follows our analysis is asymptotic in a parameter $n \in \mathbb{N}$ (rather than in ε^{-1} like in all other places in the paper). First we need a standard fact about sums of random variables. We show in Appendix A how to derive this fact from a theorem in [12].

Fact 1. *Fix constants $a, b > 0$. Let $x_1, \ldots, x_n \leftarrow [0, 1]$ be chosen uniformly randomly and independently. Then*

$$\Pr\left[\sum_{i=1}^{n} x_i \in [n/2 - a, n/2 + b]\right] = \Theta(1/\sqrt{n}).$$

We will also need the following asymptotic expression for binomial coefficients (see, e.g., [13]):

Fact 2. *Define the binary entropy function H as $H(x) = -x \log x - (1 - x) \log(1 - x)$. For any constant $\alpha \in (0, 1)$,*

$$\binom{n}{\lceil \alpha n \rceil} = \Theta\left(2^{nH(\alpha)}/\sqrt{n}\right).$$

We establish the following theorem:

Theorem 6.2. *Let $m = 2\lceil(\log n)/2\rceil$. Fix arbitrary $y \in (3/4)m + [-1, 1]$. Let $x_1, \ldots, x_m \leftarrow [1, 2]$ be uniformly random and independent values. Then, with probability $\Omega(1)$ there exists an $(m/2)$-element subset of $x_1, \ldots, x_m$ with sum in $[y - \frac{\log n}{n}, y]$.*

Proof. Let $\mathcal{I}_y = [y - \frac{\log n}{n}, y]$. Let random variable S denote the number of $(m/2)$-element subsets of $x_1, \ldots, x_m$ with sum in $\mathcal{I}_y$.

Lemma 6.3. $\mathbb{E}[S] \geq \Omega(1)$.

Proof. Let $z_1, z_2, \ldots, z_{m/2}$ be sampled uniformly from $[1, 2]$. Define random variable $Z = \sum_{i=1}^{m/2-1} z_i$. Let Z^* denote the event $Z \in [y - 2, y - 1 - \frac{\log n}{n}]$. Then,

$$\Pr\left[Z + z_{m/2} \in \mathcal{I}_y\right] \geq \Pr[Z^*] \cdot \Pr[Z + z_{m/2} \in \mathcal{I}_y \mid Z^*].$$

Bounding the probability in this manner is productive because Z^* is very likely, and conditional on Z^* the event $Z + z_{m/2} \in \mathcal{I}_y$ is easy to analyze. In particular,

$$\mathbb{E}[Z] = (m/2 - 1) \cdot (3/2) \in [y - 3, y + 3].$$

Thus Fact 1 implies that $\Pr[Z^*] = \Theta(1/\sqrt{m})$. If Z^* occurs, making the $(m/2 - 1)$-th partial sum very close to the desired value, then with probability $\frac{\log n}{n}$ the value of $z_{m/2}$ makes $Z + z_{m/2} \in \mathcal{I}_y$. So we have found

$$\Pr[Z + z_{m/2} \in \mathcal{I}_y] \geq \Omega\left(\frac{\log n}{n\sqrt{m}}\right) \geq \Omega\left(\frac{\sqrt{\log n}}{n}\right). \tag{2}$$

Now we use (2) to show $\mathbb{E}[S]$ is large. Using linearity of expectation over all $\binom{m}{m/2}$ possible $(m/2)$-element subsets of the x_i's we conclude:

$$\mathbb{E}[S] \geq \Omega\left(\frac{\sqrt{\log n}}{n}\right) \cdot \binom{m}{m/2} \geq \Omega\left(\frac{\sqrt{\log n}}{n} \cdot \frac{2^{\log n}}{\sqrt{\log n}}\right) \geq \Omega(1).$$

□

Let A_1 denote a uniformly random value from $[1, 2]$ and for each $i \in \mathbb{N}$ let A_{i+1} denote A_i plus another random independent value drawn from $[1, 2]$.

Lemma 6.4. *For any constant $\lambda \in (0, 1)$, any $i \in \mathbb{N}$ with $i \leq \lambda m/2$ and any $a \in \mathbb{R}$ we have*

$$\Pr[A_{m/2} \in \mathcal{I}_y \mid A_i = a] \leq O\left(\frac{\sqrt{\log n}}{n}\right).$$

Proof. By Fact 1 we have that for any value of A_i there is at most a $O(1/\sqrt{m/2 - i}) \leq O(1/\sqrt{m})$ chance that $A_{m/2-1}$ sums to within 2 of y. Conditional on $A_{m/2-1}$ being this close to y there is at most a $\frac{\log n}{n}$ chance that the value added to $A_{m/2-1}$ to make $A_{m/2}$ makes the sum $A_{m/2}$ precisely lie in the interval $\mathcal{I}_y$. Multiplying these probabilities yields the desired bound. □

We now proceed with the proof of the theorem. We will use the second moment method ([1]) to show that $\Pr[S > 0] \geq \Omega(1)$.

Let $\mathcal{X}$ denote the set of all size-$m/2$ subsets of $[m]$. For $A \in \mathcal{X}$ let indicator variable $S_A \in \{0, 1\}$ indicate the event that $\sum_{i \in A} x_i \in \mathcal{I}_y$. Of course $S = \sum_{A \in \mathcal{X}} S_A$. Let $\lambda = 4/5$. We decompose $\mathbb{E}[S^2]$ as:

$$\mathbb{E}[S^2] = \sum_{\substack{A,B \in \mathcal{X}^2 \\ A=B}} \Pr[S_A \wedge S_B] + \sum_{\substack{A,B \in \mathcal{X}^2 \\ |A \cap B| < \lambda m/2}} \Pr[S_A \wedge S_B] + \sum_{\substack{A,B \in \mathcal{X}^2 \\ \lambda m/2 \leq |A \cap B| < m/2}} \Pr[S_A \wedge S_B]. \tag{3}$$

Let T_1, T_2, T_3 denote the three terms in (3) in the order they appear. T_1 is simply $\mathbb{E}[S]$. Recall that Lemma 6.3 says $\mathbb{E}[S] \geq \Omega(1)$. Thus,

$$T_1 = \mathbb{E}[S] \leq O(\mathbb{E}[S]^2). \tag{4}$$

We can bound the probability in the sum defining T_2 using Lemma 6.4. In particular, observe that $A \cap B$ is a sufficiently small set, so if we condition on $\sum_{i \in A \cap B} x_i$ the conditional probabilities of S_A, S_B are at most $O((\sqrt{\log n})/n)$. The number of terms in the sum defining T_2 is trivially at most $|\mathcal{X}|^2$. Thus, we have

$$T_2 \leq O\left(\frac{\log n}{n^2}\right) \cdot \binom{m}{m/2}^2 \leq O(1). \tag{5}$$

The probabilities of S_A, S_B in the sum defining T_3 might be highly correlated so we cannot use the strong bound that we used when bounding T_2. Fortunately, for $A \neq B$ in order for both events S_A, S_B to occur we need two distinct random values to land in specific intervals of size $\frac{\log n}{n}$. Specifically if $A \neq B$ then we can find $i_B \in B \setminus A$ and $i_A \in B \setminus A$. Then, after conditioning on the value of x_i for each $i \in [m] \setminus \{a, b\}$ the probability that S_A and S_B both occur is at most $\frac{\log^2 n}{n^2}$. Fortunately the number of terms in the sum defining T_3 is not too large: it is at most

$$\binom{m}{\lceil m\lambda/2 \rceil}\binom{m}{m/2 - \lceil m\lambda/2 \rceil}^2,$$

because we can first chose $A \cap B$ and then chose $A \setminus B, B \setminus A$. Thus,

$$T_3 \leq \frac{\log^2 n}{n^2}\binom{m}{\lceil \lambda m/2 \rceil}\binom{m}{m/2 - \lceil m\lambda/2 \rceil}^2.$$

Now we show $T_3 < o(1)$. Using Fact 2 we have

$$T_3 \leq O\left(\frac{\log^2 n}{n^2} 2^{mH(\lambda/2)} 2^{2mH((1-\lambda)/2)} \frac{1}{\left(\sqrt{\log n}\right)^3}\right).$$

Thus

$$\log T_3 \leq O(1) + \log\log n + \left(H(\lambda/2) + 2H\left(\frac{1-\lambda}{2}\right) - 2\right) \cdot \log n.$$

Evaluating the expression with $\lambda = 4/5$ we find $\log T_3 < -\Omega(\log n)$. Thus $T_3 < o(1)$ as desired.

Now we combine our bounds on T_1, T_2, T_3 to obtain, via the second moment method (see chapter 4 of [1]), the bound

$$\Pr[S > 0] \geq \frac{\mathbb{E}[S]^2}{\mathbb{E}[S^2]} = \frac{\mathbb{E}[S]^2}{T_1 + T_2 + T_3} \geq \Omega(1).$$

□

We are now equipped to prove Theorem 6.1.

Proof of Theorem 6.1. We call our allocator RSUM. We start by giving a construction that works if $\delta \leq \varepsilon/4$. At the end of the proof we show how to modify this construction to work in the case $\delta > \varepsilon/4$ as well. RSUM reserves $\varepsilon/2$ free space for use as a ***buffer***, which will separate the ***main-body*** of memory and the ***trash can***: a suffix of the used portion of memory. The trash can, buffer and main-body all start empty. It is important that the buffer is at least the size of the largest item, i.e., $\varepsilon/2 \geq 2\delta$. If this is not the case we will need a more involved construction for the buffer; we discuss this at the end of the proof. RSUM reserves the remaining $\varepsilon/2$ free space to enable RSUM to create waste by introducing up to $\delta^{-1}/(2\log\varepsilon^{-1})$ gaps of size up to $g = \varepsilon\delta\log\varepsilon^{-1}$ in memory.

RSUM operates somewhat similarly to the GEO allocator of Section 4 in that RSUM handles deletes by performing ***swaps*** that introduce small amounts of waste in memory, and periodically ***rebuilds*** memory to eliminate this waste. The main difference between RSUM and GEO is that RSUM swaps *sets* of items rather than single items. This gives RSUM much greater flexibility, resulting in its substantially lower cost.

RSUM groups the items in the main-body into ***blocks*** of $m = 2\lceil(\log\varepsilon^{-1})/2\rceil$ items; the items in the trash can are not part of blocks. Blocks will be the basic units that facilitate RSUM's swap operations. Blocks are marked as either ***valid*** or ***invalid***.

Handling Deletes. Suppose an item I is deleted. RSUM forms a set Y containing I and roughly $m/2 - 1$ other nearby items, with total size $y \in \frac{3}{4}m\delta + [-\delta, \delta]$. In particular, if I is in the main-body RSUM arbitrarily adds items contiguous with I from the same block to Y until the total size of Y lies in $\frac{3}{4}m\delta + [-\delta, \delta]$. If I is in the trash can RSUM simply adds arbitrary trash can items contiguous with I to Y until the total size is appropriate. Constructing Y is possible because items have size at most 2δ.

RSUM then attempts to find a block B near the end of the main-body with a subset of elements S whose sum z is in the interval $[y - g, y]$. We say that such a block is ***compatible*** with Y. To find a compatible block RSUM ***checks*** whether the final valid block in the main-body is compatible with Y. If it is not RSUM invalidates this block and keeps trying valid blocks. If the number of valid blocks ever becomes too small RSUM will abandon its search for a compatible block and instead handle the delete via a *rebuild operation*, which will be described later. So we may assume RSUM finds a valid block B with corresponding subset S of sum $z \in [y - g, y]$.

RSUM now swaps S, Y. To swap S, Y, RSUM first takes items S and arranges them contiguously in the region of memory where items Y used to be, leaving a gap of size at most g. RSUM then takes items $Y \setminus \{I\}$ and items $B \setminus S$ and arranges them contiguously in the region of memory that was occupied by block B. We remark that if I is part of block B the above steps do nothing. RSUM then removes I from memory. Once a block of items has been used for a swap RSUM marks the block as invalidated. In particular, both the block B used to repair the delete and the block where the delete occurred (if I was in the main body) are invalidated.

To finish the swap RSUM ***pushes*** some blocks into the trash can. Recall that the trash can is a suffix of the utilized portion of memory, separated from the main-body by a small buffer. Once an item I_0's block has been invalidated RSUM may place I_0 in the trash can. However, invalidated blocks need not be immediately placed in the trash can. When a swap happens, taking items S from block B to repair a delete, RSUM takes block B and all blocks to its right in the main-body and moves them to be contiguous with the start of the trash can, and compacts them against the start of the trash can. At this point RSUM no-longer considers items from these pushed blocks to be part of any blocks.

When RSUM pushes blocks into the trash can it will potentially increase the size of the buffer (i.e., the distance between the trash can and the main body) due to the empty space created by removing item I from memory. If the buffer size now exceeds $\varepsilon/2$ RSUM takes items from the end of the trash can and rotates them to be flush with the beginning until the buffer size is again at most $\varepsilon/2$.

Handling Inserts. RSUM handles inserts by placing the inserted item after the final item currently in memory and adding the inserted item to the trash can.

Performing Rebuilds. In addition to responding to deletes and inserts as described above RSUM occasionally must perform expensive ***rebuild operations*** that ensure necessary guarantees on the layout of items in memory.

In the beginning RSUM uniformly randomly samples a ***rebuild threshold*** $r \leftarrow (\delta^{-1}/(8m), \delta^{-1}/(6m)) \cap \mathbb{N}$. This counts as a "free rebuild" (as a sentinel value). If an update would cause the number of valid blocks to drop below r, instead of handling it normally RSUM randomly permutes all items, places them contiguously into memory to eliminate all waste, and then logically partitions the items into blocks of m contiguous items, starting from the right of memory. RSUM then resamples r.

We give pseudocode for RSUM in Algorithm 6. Now we verify that RSUM is well-defined and analyze RSUM's performance.

Lemma 6.5. *RSUM places items in valid locations.*

Proof. Note that the items present are always kept contiguous except for small gaps introduced by swaps and the buffer between the main-body and the trash can. Each swap creates wasted space at most g and invalidates at least 1 block. The total number of blocks is $\lfloor\lceil\delta^{-1}/4\rceil/m\rfloor$. RSUM certainly rebuilds before all blocks are invalidated. Hence, the wasted space never exceeds $\lfloor\lceil\delta^{-1}/4\rceil/m\rfloor g \leq \varepsilon/2$. RSUM regulates the size of the buffer to be at most $\varepsilon/2$, so this ensures that if there is L total size of items present at some point in time then the items fit in the space $[0, L + \varepsilon]$. □

Lemma 6.6. *RSUM's worst-case expected update cost is $O(\log\varepsilon^{-1})$. The set of items to move at each update by RSUM can be computed in expected time $O(\varepsilon^{-1/2})$.*

Proof. RSUM clearly has cost $O(1)$ per insert. Before analyzing the expected cost of deletes, we analyze the rate at which blocks are invalidated: this will dictate the cost of rebuilds.

We will show using Theorem 6.2 that in expectation only $O(1)$ valid blocks must be checked before finding a compatible valid block to handle each delete. Fix some delete. Let $y \in \frac{3}{4}m\delta + [-\delta, \delta]$ be the size of the set of items Y contiguous with the deleted item which we aim to swap. Suppose we are given a set X of m items with sizes chosen uniformly randomly and independently from $[\delta, 2\delta]$. We claim that with constant probability there is a subset

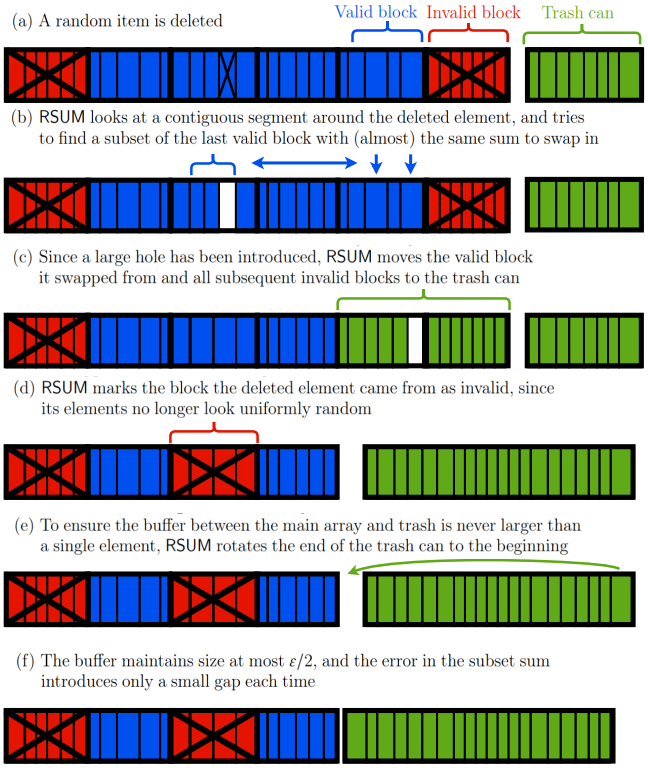

Figure 4: Operation of RSUM. Since valid blocks have never been modified, their elements will look uniformly random – by our analysis, we know they're large enough that one of the last couple valid blocks has a subset sum with size very close to the neighbourhood of the deleted element. So, RSUM swaps that subset in and then pushes a suffix of memory into the trash can to eliminate the hole.

$X' \subset X$ such that $\sum_{x \in X'} x \in [y - g, y]$. This follows immediately from Theorem 6.2, with all sizes scaled down by a factor of δ.

Intuitively this means that the expected number of valid blocks RSUM looks at on each delete should be $O(1)$. Now we formalize this intuition. Define a ***phase*** to be the set of updates between rebuild steps. Note that the set of items present is highly correlated between phases, so great care is needed. However, we will argue that RSUM's periodic rebuild operations, where RSUM randomly permutes all present items, guarantee the following property: Let C_i denote the event that the i-th check of a valid block's compatibility during a fixed phase succeeds. Then for all distinct i, j the events C_i, C_j are independent and occur each with probability $\Omega(1)$. We call this property the "***purity of valid blocks***".

We now argue why the purity of valid blocks property holds. If a block is valid, it means that RSUM has not touched or even looked at the items in the block during the phase so far. Since the set of items sizes present at the start of the phase is equivalently distributed to randomly sampled items, the sizes of the items in each valid block is equivalently distributed to randomly sampled items, as their randomness has not been spoiled. Thus, the events C_i are indeed independent random variables, and occur with probability $\Omega(1)$ by the argument above (i.e., applying Theorem 6.2).

Algorithm 6 Random-Item Sequence: RSUM
Assume $\delta < \varepsilon/4$

Choose random rebuild threshold $r \in \left(\frac{\delta^{-1}}{8m}, \frac{\delta^{-1}}{6m}\right) \cap \mathbb{N}$, initialize the trash can, buffer and main-body to be empty and consider this a "free rebuild".
if *at any time* there are fewer than r remaining valid blocks **then**
    Perform a rebuild:
    Stop the current operation.
    Compact all the items, eliminating all gaps.
    Randomly permute the items.
    Logically partition the items into ***blocks*** of m contiguous items, starting from the end of memory.
    Mark all blocks as ***valid***.
    Set the trash can to be empty.
    Resample $r \in \left(\frac{\delta^{-1}}{8m}, \frac{\delta^{-1}}{6m}\right) \cap \mathbb{N}$
if an item I is inserted **then**
    Place I immediately after the currently final item and add I to the trash can.
else if an item I is deleted **then**
    Let Y be a set of items contiguous with I, including I, whose total size y satisfies $y \in \frac{3}{4}m\delta + [-\delta, \delta]$; this is possible because the maximum item size is 2δ. If I is not in the trash can choose these items to be in the same block as I.
    while there is no subset of the final valid block with total size in $[y - g, y]$ **do**
        Invalidate the final valid block.
    Let B be the final valid block and let S be a subset of B with total size in $[y - g, y]$.
    Take items S and arrange them contiguously in the region of memory where items Y used to be, leaving a gap of size at most g.
    Take items $(Y \setminus \{I\}) \cup (B \setminus S)$ and arrange them contiguously in the region of memory that was occupied by block B
    Invalidate all blocks involved in the swap.
    Remove item I from memory, introducing a gap in block X.
    Take the block X and all blocks to its right not yet in the trash can and move all of these items into the trash can, compressing them to eliminate gaps between them.
    if the buffer has now grown too large **then**
        Take an item from the end of the trash can and swap it to the beginning of the trash can.

Thus, the expected number of blocks that RSUM invalidates on each delete is the expectation of a geometric random variable with probability $\Omega(1)$ of occurring and hence is $O(1)$. In particular this implies that the expected number of steps before there are fewer than r valid blocks is $\Omega(\delta^{-1}/m)$.

Now we analyze the expected cost of update u. There are four costs that we must analyze: the costs due to (1) rebuilding, (2) swapping items to handle deletes, (3) pushing items into the trash can, and (4) rotating items to make the buffer sufficiently small.

Intuitively, because each phase has expected length $\Omega(\delta^{-1}/m)$ and because the rebuild threshold r is random, the expected cost of rebuilding per update is $O(\log \varepsilon^{-1})$; we formalize this Lemma 6.7 after discussing the other costs.

The swap operation has cost $O(m) \leq O(\log \varepsilon^{-1})$ because there are $O(m)$ items amongst the two blocks involved in the swap. Repairing the buffer has cost $O(1)$: it requires moving at most $O(1)$ items. Now we analyze the cost of pushing items into the trash can. Using the purity of valid blocks property we have that every delete decreases the number of valid blocks by at most $O(1)$ in expectation. Since RSUM always rebuilds before the number of valid blocks drops below $\delta^{-1}/(8m)$ at most $1/2$ of the blocks are invalid at any point. Since the delete locations are uniformly random, the subset of blocks that are invalid is uniformly distributed in the main-body, conditional on its size. Thus, in expectation the number of blocks that RSUM must push to the trash can on update u is at most twice the number of blocks it invalidates. As RSUM invalidates $O(1)$ expected blocks, it only pushes $O(1)$ expected blocks to the trash can in total, for which it incurs cost $O(\log \varepsilon^{-1})$.

Now we formally analyze the expected cost due to rebuilding.

Lemma 6.7. *The expected cost of rebuilding on update u in RSUM is $O(\log \varepsilon^{-1})$.*

Proof. Fix some update u. Let $L = \delta^{-1}/(8 \log \varepsilon^{-1})$ be the maximum number of blocks that can be invalidated during a phase. Note that the minimum number of blocks that must be invalidated in each phase is $\delta^{-1}/(12 \log \varepsilon^{-1}) = 2L/3$. Recall that by the purity of valid blocks property the random variables C_i, indicating whether the i-th check for compatibility succeeds within some fixed phase are independent and each occur with probability $p \geq \Omega(1)$. Thus, by a Chernoff Bound, the ***length*** of each phase, i.e., the number of updates that are handled during that phase, is at least $Lp/50$ with probability $1 - e^{-\Omega(-L)}$. Thus, with exponentially good probability there are at most $\lceil 50/p \rceil$ rebuilds during the interval $[u - L, u]$. For each $i \in [\lceil 50/p \rceil]$ the chance that u is responsible for the ith rebuild in $[u - L, u]$ is at most $O(1/L)$. Applying a union bound we see that update u is responsible for a rebuild with probability at most $O(1/L)$. Note that it is impossible for u to be responsible for multiple rebuilds. Thus, the expected cost of performing a rebuild on update u is at most

$$O(\delta^{-1}/L) \leq O(\log \varepsilon^{-1}).$$

□

Now we analyze the expected running time required to compute RSUM's strategy. The running time is dominated by the expected $O(1)$ times that RSUM must check if a valid block is compatible to handle the delete. Each such check can be performed by computing all subset sums of the m item sizes in the valid block that it is checking. This requires $O(\varepsilon^{-1/2})$ time by using the meet-in-the-middle algorithm for finding subset sums. □

In the above analysis we have assumed $\delta \leq \varepsilon/4$ for simplicity of exposition. The only place we used this assumption is in constructing the buffer that separates the trash can from the main-body: a simple buffer requires an items-worth of slack. We now show how to handle the regime $\delta > \varepsilon/4$ as well.

Lemma 6.8. *RSUM can be modified to work for $\delta > \varepsilon/4$.*

Proof. We now describe a more complicated buffer management strategy that allows RSUM to handle $\delta > \varepsilon/4$. Fix some delete. After pushing items into the trash can on this delete RSUM ***stashes*** the final valid block from the main-body: that is, RSUM temporarily considers this block to not be contained in memory. Then RSUM rotates items from the back of the trash can to the front until the distance between the main-body and the start of the trash can is some value $y \in (3/4)m\delta + [-\delta, \delta]$. Then, RSUM attempts to find a subset S of the stashed block summing to a value in the range $[y - \varepsilon/2, y]$. RSUM will succeed with constant probability due to Theorem 6.2, which applies because $\varepsilon/2 > g$ due to $\delta = \text{poly}(\varepsilon)$. If RSUM fails, it invalidates the stashed valid block, pushes it and all blocks to its right into the trash can, and redoes the the stashing and cycling steps from above using the next valid-block. By the same analysis as in Lemma 6.6 in expectation it takes at most $O(1)$ tries before RSUM successfully finds a valid block with a subset S summing to a value in the range $[y - \varepsilon/2, y]$. Suppose RSUM finds a block with items B that has a subset S with the desired sum. RSUM then places all items B in the trash can. However, RSUM places items S at the front of the trash can and items $B \setminus S$ at the end of the trash can. Then, the gap between the main-body and the trash can is of size at most $\varepsilon/2$, as desired.

Clearly this more complex buffer management scheme increases the cost of RSUM's updates and the time required to compute RSUM's strategy by at most constant factors.

□

□

7 CONCLUSION

Our main contribution in this paper is an allocator for the memory reallocation problem achieving expected update cost $\widetilde{O}(\varepsilon^{-1/2})$. However, there are several indications that it should be possible to construct an allocator with much lower expected update cost.

Kuszmaul has already established that if all items are smaller than ε^4 then there is an allocator with expected update cost $O(\log \varepsilon^{-1})$. Using similar techniques to the covering sets introduced in this paper one can see that there are efficient allocators for sets of items with few distinct sizes and where and all sizes are fairly similar. Combined with the standard technique of discretizing item sizes this approach becomes even more powerful. Thus, "structured" sets of items can be handled efficiently. On the other hand, we gave an allocator that achieves update cost $O(\log \varepsilon^{-1})$ for large stochastic items. Thus, it seems plausible that there is a "structure versus randomness" dichotomy that can be exploited to achieve better allocators for arbitrary items. We leave constructing an allocator with expected update cost $o(\varepsilon^{-1/2})$, or strengthening our lower bound, as open problems.

REFERENCES

[1] Noga Alon and Joel H Spencer. 2016. *The probabilistic method.* John Wiley & Sons.

[2] Michael A Bender, Martin Farach-Colton, Sándor Fekete, Jeremy T Fineman, and Seth Gilbert. 2013. Reallocation problems in scheduling. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures.* 271–279.

[3] Michael A Bender, Martín Farach-Colton, Sándor P Fekete, Jeremy T Fineman, and Seth Gilbert. 2015. Cost-oblivious reallocation for scheduling and planning. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. 143–154.
[4] Michael A Bender, Martin Farach-Colton, Sándor P Fekete, Jeremy T Fineman, and Seth Gilbert. 2017. Cost-oblivious storage reallocation. *ACM Transactions on Algorithms (TALG)* 13, 3 (2017), 1–20.
[5] William Kuszmaul. 2023. Strongly History Independent Storage Allocation: New Upper and Lower bounds. *FOCS* (2023).
[6] Wei Quan Lim, Seth Gilbert, and Wei Zhong Lim. 2015. Dynamic Reallocation Problems in Scheduling. *arXiv preprint arXiv:1507.01981* (2015).
[7] Michael G Luby, Joseph Naor, and Ariel Orda. 1996. Tight bounds for dynamic storage allocation. *SIAM Journal on Discrete Mathematics* 9, 1 (1996), 155–166.
[8] George S Lueker. 1998. Exponentially small bounds on the expected optimum of the partition and subset sum problems. *Random Structures & Algorithms* 12, 1 (1998), 51–62.
[9] Moni Naor and Vanessa Teague. 2001. Anti-persistence: History independent data structures. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*. 492–501.
[10] John Michael Robson. 1971. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM (JACM)* 18, 3 (1971), 416–423.
[11] John Michael Robson. 1974. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM (JACM)* 21, 3 (1974), 491–499.
[12] J. V. Uspensky. 1937. *Introduction to Mathematical Probability*. McGraw-Hill, New York. 305 pages.
[13] Yufei Zhao. 2023. *Graph Theory and Additive Combinatorics: Exploring Structure and Randomness*. Cambridge University Press.

A OMITTED LEMMAS

In this section we prove several lemmas omitted from the main paper.

Fact 1. *Fix constants $a, b > 0$. Let $x_1, \ldots, x_n \leftarrow [0,1]$ be chosen uniformly randomly and independently. Then*

$$\Pr\left[\sum_{i=1}^{n} x_i \in [n/2 - a, n/2 + b]\right] = \Theta(1/\sqrt{n}).$$

Proof. According to [12] there exists an absolute constant $C > 0$ such that the following holds. Suppose $y_1, y_2, \ldots, y_n \leftarrow [-1/2, 1/2]$ are chosen uniformly randomly and independently. Then for any $t > 0$ we have:

$$\left|\Pr\left[\left|\sum_{i=1}^{n} y_i\right| \le t\sqrt{n}\right] - C\int_{-t}^{t} e^{-u^2/2}du\right| \le O(1/n). \tag{6}$$

Let $c \in \{a, b\}$ and take $t = c/\sqrt{n}$. Then

$$\int_{-t}^{t} e^{-u^2/2}du = \Theta(1/\sqrt{n}).$$

Using this in (6) we find

$$\Pr\left[\left|\sum_{i=1}^{n} y_i\right| \le c\right] = \Theta(1/\sqrt{n}).$$

By symmetry we then have

$$\Pr\left[\sum_{i=1}^{n} y_i \in [0, c]\right] = \Pr\left[\sum_{i=1}^{n} y_i \in [-c, 0]\right] = \Theta(1/\sqrt{n}).$$

Summing the probability of the sum landing in either of $[0, b]$, $[-a, 0]$ and translating the y_i's by $+1/2$ and gives the desired bound. □

Lemma 4.3. *Fix $a, b, W \in \mathbb{R}$ with $0 \le a < b$, and $W > 0$. Let $x_1, x_2, \ldots$ be uniformly and independently sampled from $(W/2, W)$. The probability that there exists j with $\sum_{i \le j} x_i \in [a, b]$ is at most $4(b-a)/W$.*

Proof. If $b - a \ge W/4$ the statement is vacuously true. So, we may assume $b - a < W/4$.

For each $z \in \mathbb{R}$ let $\mathsf{H}(z)$ denote the event that there exists j with $\sum_{i \le j} x_i = z$. Let $\mathsf{H}(a, b)$ denote the event that there exists j with $\sum_{i \le j} x_i \in [a, b]$. Observe that $\mathsf{H}(a, b) = \int_a^b \mathsf{H}(z)dz$. If $b \le W$ then $\Pr[\mathsf{H}(a, b)] \le 2(b - a)/W$. Suppose $b > W$.

In order for $\mathsf{H}(a, b)$ to happen the following must occur: there must be $j \in \mathbb{N}$ and $z \in (a - W, b - W/2)$ such that $\sum_{i \le j} x_i = z$, and then x_{j+1} must satisfy $z + x_{j+1} \in [a, b]$. Observe that events $\mathsf{H}(z), \mathsf{H}(z')$ are disjoint if $|z - z'| \le W/2$. Thus we have:

$$\Pr[\mathsf{H}(a, b)] \le \int_{a-W}^{b-W/2} \frac{2(b-a)}{W}\mathsf{H}(z)dz \le \frac{4(b-a)}{W}.$$

□

Lemma 4.4. *Fix integers $y, N \in \mathbb{N}$. Let $x_1, x_2, \ldots$ be uniformly and independently sampled from $[\lceil N/4\rceil, \lceil N/3\rceil] \cap \mathbb{N}$. The probability that there exists j with $\sum_{i \le j} x_i = y$ is at most $100/N$.*

Proof. For any $z \in \mathbb{Z}$ let $\mathsf{H}(z)$ denote the event that there exists j with $\sum_{i \le j} x_i = z$. If $\mathsf{H}(y)$ then we must either have $y \le \lceil N/3\rceil$ or else $\mathsf{H}(y - i)$ is true for some $i \in [\lceil N/4\rceil, \lceil N/3\rceil]$. In the case that $y \le \lceil N/3\rceil$ we clearly have $\Pr[\mathsf{H}(y)] \le 100/N$. Now, suppose $y > \lceil N/3\rceil$. Observe that $\{\mathsf{H}(y - i) \mid i \in [\lceil N/4\rceil, \lceil N/3\rceil]\}$ are disjoint events. Thus,

$$\sum_{i=\lceil N/4\rceil}^{\lceil N/3\rceil} \Pr[\mathsf{H}(y - i)] \le 1.$$

Thus we have

$$\Pr[\mathsf{H}(y)] \le \sum_{i=\lceil N/4\rceil}^{\lceil N/3\rceil} \frac{\Pr[\mathsf{H}(y-i)]}{\lceil N/3\rceil - \lceil N/4\rceil + 1} \le 100/N.$$

□