Modern Hashing Made Simple

Michael A. Bender* Martín Farach-Colton† John Kuszmaul‡ William Kuszmaul§

Abstract

Modern work on hashing has led to hash tables with extraordinary guarantees. However, these data structures are too complex to be taught in (even an advanced) data structures course. In this paper, we show that this need not be the case: using standard machinery that we already teach, one can construct a simple hash table that offers guarantees much stronger than what are classically taught:

- Operations are $O(1)$-time with high probability;
- The hash table stores n k-bit items in $nk + O(n \log\log n)$ bits of space;
- The hash table is dynamically resized, so the space bound holds with respect to the *current* size n at each time step.

1 Introduction

Since hash tables were introduced in 1953, there has been a vast literature on the question of how to design space- and time-efficient hash tables [1,2,6–8,12,14,14,16,20,21,24,25,28,32,34–36,38]. Most hash tables that students are taught offer the following basic type of guarantee: to store n keys, each of k bits, the hash table uses space $(1+\varepsilon)kn$ bits and supports insertions/deletions/queries in $\text{poly}(\varepsilon)^{-1}$ expected time. Even in advanced data structures courses, the most sophisticated hash tables that are taught (typically versions of either FKS hashing [15,16,21,22] or Cuckoo hashing [33,34]) improve on this guarantee only in terms of query time, which becomes worst-case rather than expected.[1]

However, state-of-the-art hash tables [2,6–8] offer much stronger of guarantees:

- **Time efficiency:** Queries/insertions/deletions run in constant time with high probability,
- **Space efficiency:** The space that the hash table uses to store n items, each of k bits, is $(1+o(1))nk$ bits.[2]
- **Dynamic Resizing:** The time/space guarantees are maintained even as n changes dynamically over time.

It is worth appreciating how strong these guarantees are. Even though the hash table is using space $(1+\varepsilon)nk$ bits for some $\varepsilon = o(1)$, the operations are still *constant* time. In fact, they are constant time not just in expectation, but with high probability $1 - 1/\text{poly}(n)$. As for resizing, the standard approach to keeping a hash table at space $(1-\varepsilon)nk$ bits, as n changes over time, is to perform $\Omega(n)$-time rebuilds every $\Theta(\varepsilon n)$ operations. So not only is the hash table avoiding the costs typically associated with insertions/deletions/queries in a fixed-capacity hash table, it is even avoiding the costs typically associated with resizing.

So why aren't students taught about these incredible data structures? The issue is that they are too complicated. The only hash tables in the literature to achieve the full set of guarantees [2,6–8] are viewed as technical powerhouses. And even the hash tables that come part of the way (e.g., [12,14,28,36]) can be quite involved.

*Stony Brook University, bender@cs.stonybrook.edu
†Rutgers University, martin@farach-colton.com
‡Yale University, john.kuszmaul@gmail.com
§Harvard University, william.kuszmaul@gmail.com

[1] Moreover, at least as they are typically taught, these data structures also give up on ε, settling instead for space $O(nk)$ bits. Cuckoo hashing can be optimized to support an ε, by either using $d \gg 2$ hash functions [20] or by using bins of some size $h \gg 1$ [18], and then once again the time per operation becomes a function of ε^{-1}.

[2] In fact, an even stronger guarantee can be achieved: there are constant-time hash tables with space within a $1 + o(1)$ factor of the *information-theoretic optimum* [2,6–8,28,36]. Indeed, it is now even known what the optimal time/space tradeoff curve is for the $o(1)$ term [7,27].

In this paper, we show how to construct a modern hash table using nothing but the standard machinery that is often already taught in an advanced data structures course. The resulting data structure, which we call *partition hashing*, achieves the full guarantee described above: each operation is constant time, with high probability, and the space usage is $nk + O(n \log \log n)$ bits, even as n changes over time. The construction is modular so that each technique can be introduced as its own miniature lesson. And, critically, we believe that the construction is *fun* (who would have thought that a *B-tree* would show up in a constant-time hash table?). The goal is to have a modern hash table that can easily be integrated into a modern data-structures course.

We emphasize that this paper *does not introduce any new techniques*. Rather, we extract techniques that have already appeared as components of much more sophisticated data structures [1, 2, 7, 14, 28], and show how to simplify/combine them in order to get a single, simple hash table with strong guarantees.

In the rest of the paper, we introduce partition hashing, and prove the following theorem:

THEOREM 1.1. *Let w be the machine word size, and consider hash tables storing w-bit keys, where the number n of keys stored satisfies $w = \Theta(\log n)$. There is such a hash table that uses $nw + O(n \log \log n)$ bits of space, even as n changes over time, while supporting queries in $O(1)$ worst-case time and insertions/deletions in $O(1)$ time with probability $1 - 1/\text{poly}(n)$.*

We remark that, for simplicity, we assume fully random hash functions throughout. However, there is also a rich literature [13, 17, 19, 30, 31, 37] on how to simulate fully random hash functions in hash tables that we will not be covering here.

The rest of the paper is structured as followed. Section 2 reviews the background that will be necessary in our data structure. Sections 3, 4, 5, and 6 then present the partition hash table, starting with a very weak set of guarantees and adding stronger ones one at a time. Finally, Section 7 provides historical context for the techniques used in the paper.

In our experience, partition hashing can be taught in 1–3 lectures, depending on what subset one wishes to cover. Sections 3 and 4 can be covered in a single lecture and already give a hash table with much stronger guarantees than what is typically taught. From there, each of Sections 5 and 6 can be added optionally (as the techniques are independent).

2 Preliminaries

We present some basic machinery that will be used in our construction. This is machinery that is often already taught in an advanced data structures course: it includes basic balls-and-bins analysis [23, 26], exhaustive subtabulation (a.k.a., the method of four Russians) [3], and two basic types of search trees [4, 5, 10, 11].

2.1 Notation

A ***hash table*** is a data structure that maintains a set $\mathcal{S}$ under operations INSERT$(\mathcal{S}, x)$, which adds x to $\mathcal{S}$, DELETE$(\mathcal{S}, x)$, which removes x from $\mathcal{S}$, and QUERY$(\mathcal{S}, x)$, which answers TRUE if $x \in \mathcal{S}$ and otherwise answers FALSE.

When discussing *fixed-capacity* hash table, we will use n to denote the maximum capacity. When discussing *resizable* hash tables, it will sometimes be helpful to distinguish between the table's (current) capacity versus the number of elements that it contains—in this case, we will typically use $\bar{n}$ for current capacity and n for current number of elements.

We assume that $\mathcal{S} \subseteq \mathcal{U} = [2^w]$, where $w = (1 + \Theta(1)) \log n$ is the ***word size of the machine***, meaning that operations on the bits of a word of size w take constant time. We call a hash table ***space efficient*** if it uses $(1 + o(1))nw$ bits of space.

We define ***with high probability*** (w.h.p.) to mean with probability at least $1 - 1/\text{poly}(n)$. In some cases, the probability guarantees of a data structure may depend on the constants used in the construction. Formally, a guarantee holds w.h.p if, for every positive constant c, it is possible to instantiate the constants in the construction so that the guarantee holds with probability at least $1 - O(1/n^c)$.

Our ultimate goal will be hash tables that support constant-time operations w.h.p. but along the way it will be helpful to have several intermediate notions of constant time-ness for insertions/deletions. A data structure is said to support insertions/deletion in constant ***amortized expected time*** if, for any t, the first t insertions/deletions take total expected time $O(t)$. A data structure is said to support insertions/deletions in constant ***expected time*** if, for any t, the t-th operation takes expected time $O(1)$. Of the three constant-time guarantees (amortized

expected, expected, and w.h.p.), the weakest guarantee is amortized expected, the next weakest is expected, and the strongest is w.h.p.

All of the data structures in this paper can be implemented either as key dictionaries (i.e., store a set of keys) or as key/value dictionaries (i.e., store a set of key/value pairs, so that queries recover the value associated with a given key). For simplicity, we will typically focus on the key-dictionary version of the problem, but in some cases (for example, our application of B-trees) we will also use the fact that the data structures can be augmented to support values as well.

Finally, as noted earlier, we will assume that the hash tables we are constructing have access to fully random hash functions. This means that the hash table can pick a domain U and a range R, and have a function h that maps each element from U to an independent uniformly random (but fixed) element of R. One can think of the hash table as having constant-time oracle access to h, so it can invoke h whenever it wishes, but it does not pay any space to store h.

2.2 Basic Machinery

Chernoff Bounds. The partition hash table will make use of the following basic Chernoff bound in several places.

LEMMA 2.1. *Consider a sum of independent $[0,1]$-valued random variables $X = \sum_{i=1}^{n} X_i$ with mean $\mu = \mathbb{E}[X] \geq \log n$. Then $X \leq \mu + O(\sqrt{\mu \log n})$ with high probability in n.*

COROLLARY 2.1. *Consider throwing n balls randomly into m bins, where $h = n/m \geq \log n$. Then the fullest bin has at most $h + O(\sqrt{(\log n)h})$ balls with high probability in n.*

Exhaustive subtabulation. Here, we give a brief introduction to exhasutive subtabulation (a.k.a. the Method of Four Russians) [3], which can be used to reduce the space or time of certain data structures. The basic idea is that, if there is a function f that has a relatively small range of inputs—say, the input is specified by $(\log n)/2$ bits—then we can pre-compute f on all $2^{(\log n)/2} = \sqrt{n}$ possible inputs, we can store those computations in a lookup table, and then in the future we can use the lookup table to evaluate f in constant-time. We will use this technique to implement edits to data structures where the portion that we are editing is $O(\log n)$ bits.

B-Trees. A B-tree [4,5,10] on n μ-bit keys is a balanced search where each node has up to $B+1$ children, and where the depth of the tree is $O(\log_{B+1} n)$. (So, when $B = 1$, the B-tree becomes a balanced binary search tree.) The B-tree is designed for block-transfer memory, where a transfer of a block of size $\Theta(B\mu)$ bits takes constant time. Note that, so long as the tree consists of $2^{O(\mu)}$ nodes (so that pointers between the nodes require $O(\mu)$ bits), then each node in the tree can be stored in 1 block. There are a variety of schemes to keep such a tree balanced—in short, insertions, deletions, and queries can be implemented in time $O(\log_{B+1} n)$ (in the block-transfer model).

k-Tries. For any power-of-two k, a k-trie [11] on n w-bit keys is a k-ary search tree where the pivots are evenly spaced in the universe of possible keys. For any key $x \in [2^w]$ in the tree, the root-to-leaf path for x can be computed by: using the first (i.e., highest-order) $\log k$ bits to navigate the root node, using the next $\log k$ bits to navigate the level-2 node, using the next $\log k$ bits to navigate the level-3 node, and so on, for a total of $w/\log k$ levels. The depth of a k-trie is therefore $w/\log k$.

LEMMA 2.2. *A k-trie on n w-bit keys takes time $O(w/\log k)$ per operation and takes space $O(nkw^2/\log k)$ bits. Here, we assume that the arrays used to allocate nodes are already pre-allocated and initialized to zero.*

Proof. Operations in each node are constant time, so insertions, deletions and queries take time proportional to the depth, which is $O(w/\log k)$. We can overestimate the size of the k-trie by charging each of the n items the full cost of its root-to-leaf path. Each node is $O(k)$ machine words (i.e., $O(kw)$ bits), and the depth is $w/\log k$, so the total space is $O(nkw^2/\log k)$ bits. ∎

Resizable Arrays. Finally, it will be helpful to know the following folklore trick for how to implement space-efficient resizable arrays:

LEMMA 2.3. *Consider an array A that grows and shrinks over time, supporting both an extension operation (grow the array by 1) and a shrink operation (shrink the array by 1). If $\bar{n}$ is the maximum size that the array is allowed to be, and n is the current size at any given moment, then the array A can be implemented to use $n + O(\sqrt{\bar{n}})$ machine words while supporting constant-time operations.*

Proof. We break A into slabs $A_1, \ldots, A_{\lceil n/\sqrt{n} \rceil}$ of size $\sqrt{n}$ each. These slabs are dynamically allocated and freed so that there are only $\lceil n/\sqrt{n} \rceil$ at a time. We also keep a smaller array B of size $\sqrt{n}$ whose i-th entry stores a pointer to slab A_i for each $i \leq \lceil n/\sqrt{n} \rceil$.

The only wasted space is the space consumed by B plus the unused space in slab $A_{\lceil n/\sqrt{n} \rceil}$. Thus the total space usage is $n + O(\sqrt{n})$ machine words, as desired. Array operations are trivially implemented in constant time, since each slab can be accessed through B, and since each growth/shrinkage of the array requires at most one slab to be allocated/freed. ∎

3 Slow Partition Hash Tables

We begin by constructing a very simple fixed-capacity hash table that has small space, that supports constant-time insertions (w.h.p.), but that incurs polylogarithmic time per query/deletion.

Define a ***bucket*** to be an array of length $\log^3 n + c \log^2 n$ slots, for a constant $c > 0$. Define a ***bucket array*** to be an array of $n/\log^3 n$ buckets. The total size of a bucket array is thus $n + cn/\log n$ slots.

Our first attempt at a hash table is simply the following: when an item arrives, it is hashed to a random bucket in the bucket array, and it is placed in an arbitrary empty slot in that bucket. If the bucket is full on insertion, the entire hash table is rebuilt. When a query is made, it is hashed to a bucket and every slot in the bucket is examined.

When an insertion is hashed to a bucket, all it needs to do is to find an empty slot. To make this efficient, we keep the items in a bucket ***left justified***, i.e., if there are k items in a bucket, they are kept in the first k slots of the bucket. We can achieve this invariant by maintaining an $O(\log \log n)$-bit counter in each bucket that keeps track of k. Insertions place the item in position $k + 1$ and increment the counter. Deletions must also keep the bucket left justified—when an item is deleted from an arbitrary location in a bucket other than k, the rightmost item is moved to that location, and then the counter k is decremented.

The slow partition hash table can be analyzed with the following lemma.

LEMMA 3.1. *The slow partition hash table takes $O(1)$ time for insertions, with high probability, and $O(\log^3 n)$ worst-case time for deletions and queries. It takes space $nw + O(n)$ bits.*

Proof. Recall that $w = \Theta(\log n)$, so the $n + O(n/\log n)$ slots in the bucket array take $nw + O(n)$ bits.

Queries/deletions are always $O(\log^3 n)$ time since they look at a single bucket. Insertions take $O(1)$ time *unless* a rebuild is performed. Thus it remains to analyze the probability of rebuilding during a given insertion.

Since there are $n/\log^3 n$ buckets and (up to) n elements, the expected number of elements per bucket is at most $\log^3 n$. Corollary 2.1 bounds the probability that any bucket has more than $\log^3 n + c \log^2 n$ keys to be $1/\text{poly}(n)$. The probability of a bucket overflowing (causing a rebuild) is therefore also $1/\text{poly}(n)$. ∎

Of course, the only reason that slow partition hash tables are able to support constant-time insertions w.h.p. is because they are completely neglecting queries/deletions. Next, we will see how to improve the hash table so that insertions take constant expected time and deletions/queries take worst-case constant time.

4 Constant-Time Queries/Deletions and Constant Expected-Time Insertions

The ***indexed partition hash table*** is an extension of the slow partition hash table that supports insertions in constant expected time and queries/deletions in worst-case constant time. The indexed partition hash table will be very space efficient, using $(1 + o(1))nw$ bits, so already we will be achieving a set of guarantees that are not typically covered in an advanced data structures course. With that said, we will see in later sections how to improve on such hash tables (adding high-probability guarantees for insertions and dynamic resizing of n).

We will describe the indexed partition hash table in three pieces: first, we describe how to assign all of the keys x within a given bucket to distinct $\Theta(\log \log n)$-bit fingerprints $f(x)$; then we describe a simple per-bucket data structure, called the ***query mapper***, that maps each fingerprint $f(x)$ to the position k where the key resides in the bucket; and finally describe how the query mapper is used to implement operations within the bucket.

Our first step is to map the keys within each bucket to distinct fingerprints. We do this with a hash function f mapping keys to random $\Theta(\log \log n)$-bit integers. We will prove that, with probability at least $1 - 1/\text{polylog}(n)$, all fingerprints within a bucket are different. In the event that the fingerprints in a bucket are *not* all different, we reconstruct the fingerprints from scratch using a different hash function (and so on, until we get distinct fingerprints).

LEMMA 4.1. *Consider a set B of $O(\log^3 n)$ keys, and consider a random hash function $f : B \to [\log^9 n]$. With probability at least $1 - O(1/\log^3 n)$, the hashes $f(x)$ are distinct across $x \in B$.*

Proof. For a given pair of items $x, y \in B$, the probability of a collision $f(x) = f(y)$ is $1/\log^9 n$. By a union bound the probability of any collisions occurring is at most

$$\sum_{x \neq y \in S} \Pr[f(x) = f(y)] \leq \frac{|S|^2}{\log^9 n} \leq O\left(\frac{1}{\log^3 n}\right).$$

∎

Having mapped the keys within each bucket to distinct fingerprints, our next step is to add what we call a ***query mapper*** to each bucket. The query mapper in each bucket is a data structure that maps each fingerprint $f(x)$ to the corresponding position $i \in [O(\log^3 n)]$ where the item x is stored in the bucket. Note that both the fingerprint $f(x)$ and the position i are $\Theta(\log \log n)$ bits each.

The query mapper might seem like a difficult data structure to implement. After all, the query mapper, itself, is a dynamic data structure mapping keys to values (isn't this the problem we are trying to solve in the first place?). However, the fact that the keys and values are *small* turns out to be important, allowing us to make use of a somewhat unexpected tool: the B-tree!

LEMMA 4.2. *One can implement a query mapper, mapping $\Theta(\log \log n)$-bit keys to $\Theta(\log \log n)$-bit values, so that it supports queries/insertions/deletions in worst-case time $O(1)$ and uses space $O(m \log \log n)$ bits to store $m = \text{polylog}(n)$ key/value pairs.*

Proof. We implement the query mapper as a B-tree with $B = \sqrt{\log n}$. This B-tree is quite unusual: each node stores up to $\sqrt{\log n}$ keys/values of size $\Theta(\log \log n)$ bits each, along with up to $\sqrt{\log n} + 1$ pointers to children nodes. Notice that the pointers are also $O(\log \log n)$ bits each, since the total number of nodes in the tree is $\text{polylog}(n)$.

Thus each node has total size only $O(\sqrt{\log n} \log \log n)$ bits. The fact that each node fits in $o(\log n)$ bits means that we can use exhaustive subtabulation in order to implement node-level operations (queries, modifications, insertions, deletions) in $O(1)$ time.

Moreover, since the tree contains at most $O(\log^5 n)$ leaves and has fanout $\Theta(\sqrt{\log n})$, the depth of the tree is guaranteed to be $O(1)$. Thus, in this context, the B-tree is actually an $O(1)$-time data structure. ∎

Having constructed the query mapper within each bucket, we can now describe how the query mapper interacts with operations in the bucket: To insert an item x, a fingerprint $f(x)$ is computed, and then that fingerprint is queried in the B-tree query-mapper; if the fingerprint is already present, then the index is rebuilt from scratch using a new fingerprint function; otherwise, the key is placed in the first empty position k and the pair $(f(x), k)$ is added to the query mapper. To query an item x, we simply use the query mapper to obtain a position i. If the query mapper fails to find $f(x)$, or if a different item x' is in position i, then x is not in the hash table. To delete an item x, we first perform a query for it, then we remove the item and possibly move one other item x' in order to keep the bucket left aligned; finally, we update the query mapper to remove x, and to update the position stored for x'.

Notice that, out of the three operations, the only one that is at risk of taking super-constant time is the insertion, which may need to rebuild the query-mapper in the event of a fingerprint collision. Thus, we can complete the analysis of the indexed partition hash table as follows:

LEMMA 4.3. *An indexed partition hash table supports $O(1)$-time queries and deletions and $O(1)$ expected-time insertions. It takes $nw + O(n \log \log n)$ bits of space.*

Proof. To establish the time guarantees, it suffices to show that the expected time per insertion spent rebuilding the query mapper is $O(1)$. We know from Lemma 4.1 that the probability of a fingerprint collision (and thus a rebuild) occurring is at most $O(1/\log^3 n)$. If a rebuild occurs, it will take $O(\log^3 n)$ time to reconstruct the bucket using new fingerprints. These fingerprints could also have a collision (with probability at most $O(1/\log^3 n)$),

causing yet another rebuild, and so on. For all $i > 0$, the probability of an i-th rebuild being performed is at most $O(\log^3 n)^{-i}$, and the time to perform the rebuild is $O(\log^3 n)$. So the expected time spent on rebuilds is

$$\sum_{i \geq 1} \frac{O(\log^3 n)}{O(\log^3 n)^i} = O(1).$$

Finally, notice that the space consumed by the query mappers is collectively $O(n \log \log n)$ bits, since each item uses $O(\log \log n)$ bits in some query mapper. This establishes the desired space guarantee. □

5 Constant-Time Queries and Consant-Time Insertions/Deletions w.h.p.

So far, we have constructed a hash table that supports constant-time queries/deletions and constant expected-time insertions, with $nw + O(n \log \log n)$ bits of space. We now show how to improve the insertion time to $O(1)$ with high probability. In fact, the technique used here, which can be viewed as a simplification of an approach used in past works [1,2,14], can actually be extended to apply to *any* hash table (see Appendix A), a result that to the best of our knowledge has not previously appeared in the literature.

Our new data structure will make use of two $\sqrt{n}$-tries, each storing up to $n^{1/4}$ w-bit keys. Since $w = \Theta(\log n)$, the depth of such a $\sqrt{n}$-trie is $O(w/\log \sqrt{n}) = O(1)$; and the space for such a $\sqrt{n}$-trie is $O(n^{1/4} \cdot \sqrt{n} \cdot w^2/\log \sqrt{n}) = O(n^{0.75} \log n)$ bits.

The two tries have different roles. The ***growing*** trie receives new insertion/deletion operations. The ***shrinking*** trie has items removed as they are implemented. For every insertion/deletion operation that is added to the growing tree, a constant $c > 1$ amount of time is spent implementing operations from the shrinking tree in the actual hash table. We will guarantee that, whenever the growing tree reaches size $n^{1/4}$, the shrinking tree will have shrunk to size 0—at this point, the two tries swap roles.

The key technical claim that we must prove is that: Any set of $n^{1/4}$ insertion/deletions can be implemented in the hash table in $O(n^{1/4})$ total time, with high probability. If we can prove this, then our approach of performing $O(1)$ work per operation on the shrinking trie will be sufficient to empty the trie out within $n^{1/4}$ operations, as desired.

LEMMA 5.1. *Any batch Q of $n^{1/4}$ insertions/deletions can be implemented in total time $O(n^{1/4})$ with high probability.*

Proof. We assume without loss of generality that the deletions in Q are on different keys than the insertions, since otherwise, they cancel each other.

Define $A_1, \ldots, A_{n/\log^3 n}$ so that A_i is the number of insertions/deletions from Q that get mapped to the i-th bucket. With high probability, every A_i satisfies $A_i \leq O(\log^3 n)$ (as this is true even if we consider all of the items in the hash table, rather than just the ones in Q). Condition on any fixed outcome of the A_is satisfying $A_i \leq O(\log^3 n)$ for each i, and define random variables $X_1, \ldots, X_{n^{0.9}}$ so that X_i is the amount of time needed to implement the operations from A_i on the i-th bucket.

We begin by bounding the worst-case value that X_i could take (with high probability). With high probability, none of the operations in A_i overflow the bucket, so they do not trigger any rebuilds of the entire data structure. On the other hand, some insertions may take super-constant time, as they may cause fingerprint collisions that require the bucket to be rebuilt. Observe, however, that each rebuild takes $O(\log^3 n)$ time, and that the number of rebuilds triggered by a given insertion is a geometric random variable (each attempt fails with probability $\ll 1/2$). It follows that, with high probability, there will be $O(\log n)$ rebuild attempts per insertion, and thus that each insertion takes $O(\log^4 n)$ time. Summing over the $O(\log^3 n)$ insertions in A_i, we can guarantee that $X_i \leq O(\log^7 n)$ with high probability.

Next, we consider the expected value of X_i. This is easy, since each of the $|A_i|$ operations takes $O(1)$ expected time, so by linearity of expectation we have $\mathbb{E}[X_i] \leq O(|A_i|)$.

Finally, we can complete the proof with a simple Chernoff bound. The X_is are independent since X_i's randomness comes from the i-th sub-hash-table, and since we are already conditioning on fixed outcomes for the A_is, so they are not providing any randomness. Each X_i has expected value $\mathbb{E}[X_i] = O(A_i)$ and worst-case value at most $O(\log^7 n)$. Thus $X = \sum_i X_i$ is the sum of independent random variables with total expected value $O(\sum_i A_i) = O(n^{1/4})$, and where each X_i is at most $O(\log^7 n)$. This means that the quantity $X/\Theta(\log^7 n)$ is a

sum of independent $[0,1]$ random variables with total mean $O(n^{1/4}/\log^7 n)$, which by a Chernoff bound (Lemma 2.1) implies that $X/\Theta(\log^7 n)$ exceeds its mean $\mu = n^{1/4}/\log^7 n$ by at most $O(\mu\sqrt{\log n})$ with high probability. Unfolding the bound, we get that X exceeds its mean by at most $O(n^{1/8}\log^6 n) = o(n^{1/4})$. Thus $X \leq O(n^{1/4})$, as desired. ∎

6 Resizing Without Hurting Space or Time

Finally, we will show how to add dynamic resizing so that the space-bound becomes $nw + O(n \log\log n)$ bits, where n is the current number of items.

We begin by focusing on the case where n stays in the range $[\bar{n}/2, \bar{n}]$ for some parameter $\bar{n}$. In this regime, there is a *very simple* trick that we can use for resizing.

The trick is that, rather than trying to resize the hash table as a single unit, we focus on resizing each of the buckets. The only modification that we need to make is to allow the size of each bucket to grow/shrink over time as n grows/shrinks. We can do this by directly applying Lemma 2.3 to each bucket in the hash table: the bucket, whose capacity is $O(\log^3 \bar{n})$, can store k items for any $k \leq O(\log^3 \bar{n})$ using only $k + O(\log^{1.5} \bar{n})$ machine words. Collectively over all $\Theta(\bar{n}/\log^3 \bar{n})$ buckets, the buckets waste less than $O(\bar{n}/\log^{1.5} \bar{n})$ machine words of space, which sums to $O(\bar{n}/\log^{0.5} \bar{n}) \leq O(\bar{n})$ bits.

This brings us to the following lemma.

LEMMA 6.1. *If there are only $n \leq \bar{n}$ items stored in a resizable partition hash table, the space used is $nw + O(\bar{n} \log\log \bar{n})$ bits. If $n \in [\bar{n}/2, \bar{n}]$, then this becomes $nw + O(n \log\log n)$ bits.*

Proof. As was the case for non-resizable partition hash tables, the per-bucket query mappers combine to use space $O(\bar{n} \log\log \bar{n})$ bits. The buckets themselves then use $nw + O(\bar{n})$ bits of space, as argued above. ∎

Finally, we extend our construction to be space efficient even if n changes dramatically. Now, the only constraint on n will be the machine-word size requirement: that the machine word size should be logarithmic in n, that is, $\log n = \Theta(\log w)$. This constraint is necessary so that (1) the definition of *w.h.p. in n* stays the same over time; and (2) so that the lookup tables used in exhaustive tabulation take a negligible amount of space.

Our approach for allowing n to change dramatically over time will be the standard one: whenever n changes by a constant factor, we rebuild the hash table according to Lemma 6.1 from scratch using a new $\bar{n} = \Theta(n)$. We spread the rebuild over $O(n)$ operations so that each operation spends only $O(1)$ time on it.

We must be careful about space efficiency, however. As we are migrating elements from the old version of the hash table to the new one, we must consider the total space consumed by both hash tables. This is where Lemma 6.1 comes to the rescue:

COROLLARY 6.1. (COROLLARY OF LEMMA 6.1) *Consider hash tables H_1 and H_2 that are each implemented with Lemma 6.1 using capacities $\bar{n}_1$ and $\bar{n}_2$. Suppose that the hash tables contain n_1 and n_2 elements, respectively. Then, collectively, they use space*

$$(n_1 + n_2)w + O(\bar{n}_1 \log\log \bar{n}_1 + \bar{n}_2 \log\log \bar{n}_2)$$

bits.

As we are performing a given resize (spread over $O(n)$ operations), the elements are spread across two tables (a new and old one, with capacities $\bar{n}_1$ and $\bar{n}_2$ that are both $\Theta(n)$ but that differ by constant factors). Collectively, the hash tables contain all n elements, but each element is in at most one of the hash tables. Thus, by Corollary 6.1, the space usage at any given moment is $nw + O(\bar{n}_1 \log\log \bar{n}_1) + O(\bar{n}_2 \log\log \bar{n}_2) = nw + O(n \log\log n)$ bits.

To summarize, by rebuilding the hash table every time that n changes by a constant factor, and by spreading the time spent for the rebuild across $\Theta(n)$ operations, we arrive at the following:

LEMMA 6.2. *A partition hash can be resized dynamically so that if it has $n \in [2^{\varepsilon w}, 2^w]$ items, where ε is a constant in $(0,1)$, it uses space $nw + O(n \log\log n)$ bits. Its time complexity remains constant time per query and constant time w.h.p. for insertions and deletions.*

Putting the pieces together, we have arrived at our main theorem, restated below:

THEOREM 6.1. *Let w be the machine word size, and consider hash tables storing w-bit keys, where the number n of keys stored satisfies $w = \Theta(\log n)$. There is such a hash table that uses $nw + O(n \log \log n)$ bits of space, even as n changes over time, while supporting queries in $O(1)$ worst-case time and insertions/deletions in $O(1)$ time with probability $1 - 1/\text{poly}(n)$.*

7 Historical Notes

The basic idea of hashing keys to buckets, and then using an indexing data structure to perform queries, appears in almost all modern work on space-efficient hashing [2,6,7,29,36]. Historically, a major obstacle here was how to design a space- and time-efficient query-mapping structure [2,6,29,36]. The idea of using exhaustive tabulation to help implement the query mapper was due to Blelloch and Golovin in their work on history-independent hash tables [9]. The insight that the query mapper can be implemented using a B-tree [4,5,10] is much more recent, due to Bender et al. [7], although their query mappers are necessarily far more complex than the ones here as they seek to optimize even the low-order terms in the space usage.

The basic idea of achieving w.h.p. bounds for insertions/deletions in a hash table by (a) queuing insertions/deletions in an auxiliary buffer, and (b) spending constant work per operation on emptying that buffer, can be attributed to a 1990 paper by Dietzfelbinger and Meyer auf der Heide [14]. Variations on the idea have also been introduced and analyzed in the context of cuckoo hashing [1,2]. The idea ends up being especially simple to describe in the context of partition hash tables because the hash table naturally decomposes into buckets that can be analyzed individually. Another simplification that we make here is to implement the buffer directly as a trie—this means that, unlike in past work, the buffer itself does not require any novel randomized data structures. Finally, the observation that the technique can be generalized to apply to arbitrary hash tables appears to be a new contribution to the literature.

The question of how to perform space-efficient resizing was left open [2] until relatively recently [6,7,28]. The local-bin-resizing approach that we take here can be viewed as a variation on the approach that was introduced by Liu et al. [28] in the context of incremental (i.e., insertion-only) hash tables. Practically speaking, one downside of this approach is that it adds an extra layer of indirection to the hash table. For an indirection-free approach to resizing, see *waterfall addressing* [6].

All of the techniques used in this paper can be viewed as descendants of techniques that have appeared in past work as components of far more sophisticated data structures. The contribution of this paper is therefore not to invent new techniques (although we do simplify them), but rather to present a single, simple data structure, that allows for each of the techniques to be presented with as little extraneous baggage as possible.

8 Acknowledgements

The authors are grateful to Keith Schwarz for his extensive feedback on earlier versions of this manuscript. His suggestions significantly improved the final paper.

The authors would like to thank the NSF for support through grants CCF 2106999, CCF 2247576, CCF 2247577, and CCF 2106827.

Finally, 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.

References

[1] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5555 of *Lecture Notes in Computer Science*, pages 107–118, 2009.

[2] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 787–796. IEEE, 2010.

[3] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. On economical construction of the transitive closure of an oriented graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1970. URL: `http://cr.yp.to/bib/entries.html#1970/arlazarov`.

[4] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. In Edgar F. Codd, editor, *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141, 1970.

[5] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[6] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. All-purpose hashing. *arXiv preprint arXiv:2109.04548*, 2021.

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

[8] Ioana O Bercea and Guy Even. Dynamic dictionaries for multisets and counting filters with constant time operations. *Algorithmica*, 85(6):1786–1804, 2023.

[9] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007)*, pages 272–282, Providence, Rhode Island, USA, 21–23 October 2007.

[10] Douglas Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[11] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, Western Joint Computer Conference*, pages 295–298, 1959.

[12] Erik D Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Latin American Symposium on Theoretical Informatics*, pages 349–361. Springer, 2006.

[13] Martin Dietzfelbinger. Design strategies for minimal perfect hash functions. In *International Symposium on Stochastic Algorithms*, pages 2–17. Springer, 2007.

[14] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *International Colloquium on Automata, Languages, and Programming*, pages 6–19. Springer, 1990.

[15] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[16] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proceedings 29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 524–531, October 1988.

[17] Martin Dietzfelbinger and Michael Rink. Applications of a splitting trick. In *International Colloquium on Automata, Languages, and Programming*, pages 354–365. Springer, 2009.

[18] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, 2007.

[19] Martin Dietzfelbinger and Philipp Woelfel. Almost random graphs with simple hash functions. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 629–638, 2003.

[20] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G Spirakis. Space efficient hash tables with worst case constant access time. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STOC)*, pages 271–282, 2003.

[21] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS'82)*, pages 165–169, 3–5 November 1982.

[22] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[23] Norman Lloyd Johnson and Samuel Kotz. *Urn models and their application: An approach to modern discrete probability theory*. Wiley, New York, NY, 1977.

[24] Don Knuth. Notes on "open" addressing, 1963.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[26] Valentin F. Kolchin. *Random Allocations*. Scripta Series in Mathematics. VH Winston, 1978.

[27] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. Tight cell-probe lower bounds for dynamic succinct dictionaries. In *FOCS*, 2023.

[28] Mingmou Liu, Yitong Yin, and Huacheng Yu. Succinct filters for sets of unknown sizes. In *47th International Colloquium on Automata, Languages, and Programming (ICALP)*. Schloss Dagstuhl-Leibniz-Zentrum für

Informatik, 2020.
[29] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 327–336, 2011.
[30] Anna Ostlin and Rasmus Pagh. Uniform hashing in constant time and linear space. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 622–628, 2003.
[31] Anna Pagh and Rasmus Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, 2008.
[32] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.
[33] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133, University of Aaarhus, Denmark, 28–31 August 2001. Springer.
[34] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.
[35] Mihai Patrascu. Succincter. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.
[36] Rajeev Raman and Satti Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 357–368, 2003.
[37] A Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 20–25, 1989.
[38] Huacheng Yu. Nearly optimal static Las Vegas succinct dictionary. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1389–1401, 2020.

A Deamortizing Insertions in Arbitrary Hash Tables

In this section, we show how to deamortize insertions/deletions in any hash table, so that if they formerly required $O(1)$ expected time, they now require $O(1)$ worst-case time with high probability.
We begin by considering ***non-blocking hash tables***, defined to be a hash table with the property that queries can be performed even when insertions or deletions are in the process of executing.

PROPOSITION A.1. *A non-blocking hash table with constant worst-case queries and constant expected-time insertions/deletions can be modified so that it performs queries in worst-case constant time and insertions and deletions in constant time w.h.p. The total space overhead is $o(n)$ bits.*

Our data structure makes use of two $\sqrt{n}$-tries, exactly as in Section 6. As before, what we need to prove is that: Any set of $n^{1/4}$ insertion/deletions can be implemented in the hash table in $O(n^{1/4})$ total time, with high probability.
However, in general, this claim may not be true. To make it so, we need to make one more algorithmic modification to how we use the non-blocking hash table. Rather than storing a single hash table, we keep $n^{0.9}$ sub-hash-tables, and we hash each key to a random sub-hash-table where it resides. By Corollary 2.1 each sub-hash-table will have at most $n^{0.1} + O(\sqrt{n^{0.1}\log n})$ keys at a time, w.h.p.. Since each sub-hash-table can be implemented to have capacity $n^{0.1} + O(\sqrt{n^{0.1}\log n})$, the sum of the capacities is $n + O(n^{0.95}\sqrt{\log n})$, and the excess capacity (beyond n) contributes a space overhead of only $o(n)$ bits.
Now we can analyze the total time needed to implement $n^{1/4}$ insertions/deletions. This, in turn implies Proposition A.1.

LEMMA A.1. *Any batch Q of $n^{1/4}$ insertions/deletions can be implemented in total time $O(n^{1/4})$ with high probability.*

Proof. Our proof will be very similar to that of Lemma 5.1. The difference is that now the sub-hash-tables of size $O(n^{0.1})$ will serve the role that formerly was served by the buckets of size $O(\log^5 n)$.
Without loss of generality, the deletions are on different keys than the insertions, since otherwise they cancel. Define $A_1, \ldots, A_{n^{0.9}}$ so that A_i is the number of insertions/deletions from Q that get mapped to the i-th sub-hash-table. Fix any outcome of the A_is (with $A_i \leq O(n^{0.1})$ for each i), and define random variables $X_1, \ldots, X_{n^{0.9}}$ so that X_i is the amount of time needed to implement the A_i operations on the i-th sub-hash-table. We will show that we can apply a Chernoff bound to the total work $\sum_i X_i$.

A key observation is that no X_i can be more than $O(n^{0.1} \log n)$. This is because, if the operations on the i-th sub-table take more than $O(n^{0.1})$ time, then we can simply rebuild that hash table from scratch (using a new hash function). With high probability, we will not need to do so more than $O(\log n)$ times (since, each time that we do it, we have at least a, say, 50% chance of succeeding in less than $O(n^{0.1})$ time). Thus we have $X_i \leq O(n^{0.1} \log n)$ for each i, w.h.p..

We can now complete our proof with a Chernoff bound (Lemma 2.1). The X_is are independent, since X_i's randomness comes from the i-th sub-hash-table, and since we are already conditioning on fixed outcomes for the A_is, so they are not providing any randomness. Each X_i has expected value $\mathbb{E}[X_i] = A_i$. Thus $X = \sum_i X_i$ is the sum of independent random variables with total expected value $O(\sum_i A_i) = O(n^{1/4})$, and where each X_i is at most $O(n^{0.1} \log n)$. This means that the quantity $X/\Theta(n^{0.1} \log n)$ is a sum of independent $[0, 1]$ random variables with total mean $n^{0.15}$, which by a Chernoff bound implies that $X/\Theta(n^{0.1} \log n)$ exceeds its mean by at most $O(\sqrt{(\log n) n^{0.15}})$ w.h.p.. Unfolding the bound, we get that X exceeds its mean by at most $O(\log^{1.5} n^{0.1+.075}) = o(n^{1/4})$. Thus $X \leq O(n^{1/4})$, as desired. ∎

We now deal with hash tables that can block queries during insertions and deletion. The question that we must answer is: how can we gradually implement the shrinking buffer, while still allowing queries to run?

As an insertion/deletion is being performed, there are modifications that it would like to make to the hash table. To shield the modifications from queries (until the insertion/deletion is complete), we keep a ***diff-buffer*** that records the differences between what the insertion/deletion thinks the hash table looks like and what queries think it looks like—that is, the diff-buffer records edits that the insertion/deletion plans to make. We keep these in a $\sqrt{n}$-trie, just like the insertion buffer. Thus, the queries that happen during the insertion/deletion can read the old hash table, and the insertion/deletion can read bytes by consulting both the old hash table and the diff buffer.

Once the insertion/deletion has recorded all its changes, the hash table enters a new state, in which all queries now consult the hash table and the diff buffer (so they see the insertion/deletion as having been implemented). During this phase, changes are removed incrementally from the diff buffer and applied to the hash table. Once the diff buffer is empty, queries can go back to accessing just the hash table, and we are ready to process another insertion/deletion from the insertion buffer.

At each point in time, queries see a well-formed hash table, and any currently active insertion/deletion sees the hash table as it is being incrementally modified. We conclude that:

THEOREM A.1. *Any hash table that has constant time queries and constant expected time insertions and deletions can be deamortized to yield constant time queries and constant time insertions/deletion w.h.p. by using an extra $o(n)$ bits.*