Tiny Pointers

MICHAEL A. BENDER, Stony Brook University, USA
ALEX CONWAY, Cornell Tech, USA
MARTÍN FARACH-COLTON, New York University, USA
WILLIAM KUSZMAUL, Carnegie Mellon University, USA
GUIDO TAGLIAVINI, Rutgers University, USA

This paper introduces a new data-structural object that we call the tiny pointer. In many applications, traditional $\log n$-bit pointers can be replaced with $o(\log n)$-bit tiny pointers at the cost of only a constant-factor time overhead and a small probability of failure. We develop a comprehensive theory of tiny pointers, and give optimal constructions for both fixed-size tiny pointers (i.e., settings in which all of the tiny pointers must be the same size) and variable-size tiny pointers (i.e., settings in which the average tiny-pointer size must be small, but some tiny pointers can be larger). If a tiny pointer references an item in an array filled to load factor $1 - \delta$, then the optimal tiny-pointer size is $\Theta(\log \log \log n + \log \delta^{-1})$ bits in the fixed-size case, and $\Theta(\log \delta^{-1})$ expected bits in the variable-size case.

Our tiny-pointer constructions also require us to revisit several classic problems having to do with balls and bins; these results may be of independent interest.

Using tiny pointers, we apply tiny pointers to five classic data-structure problems. We show that:

- A data structure storing n v-bit values for n keys with constant-factor time modifications/queries can be implemented to take space $nv + O(n \log^{(r)} n)$ bits, for any constant $r > 0$, as long as the user stores a tiny pointer of expected size $O(1)$ with each key—here, $\log^{(r)} n$ is the r-th iterated logarithm.
- Any binary search tree can be made succinct, meaning that it achieves $(1 + o(1))$ times the optimal space, with constant-factor time overhead, and can even be made to be within $O(n)$ bits of optimal if we allow for $O(\log^* n)$-time modifications—this holds even for rotation-based trees such as the splay tree and the red-black tree.
- Any fixed-capacity key-value dictionary can be made stable (i.e., items do not move once inserted) with constant-factor time overhead and $(1 + o(1))$-factor space overhead.
- Any key-value dictionary that requires uniform-size values can be made to support arbitrary-size values with constant-factor time overhead and with an additional space consumption of $\log^{(r)} n + O(\log j)$ bits per j-bit value for an arbitrary constant $r > 0$ of our choice.
- Given an external-memory array A of size $(1 + \varepsilon)n$ containing a dynamic set of up to n key-value pairs, it is possible to maintain an internal-memory stash of size $O(n \log \varepsilon^{-1})$ bits so that the location of any key-value pair in A can be computed in constant time (and with no IOs).

In each case tiny pointers allow for us to take a natural space-inefficient solution that uses pointers and make it space-efficient for free.

Additional Key Words and Phrases: pointers, space-efficient, balanced allocation, balls and bins, hashing, load balancing, randomized algorithms, retrieval

Authors' addresses: Michael A. Bender, bender@cs.stonybrook.edu, Stony Brook University, USA; Alex Conway, me@ajhconway.com, Cornell Tech, USA; Martín Farach-Colton, martin@farach-colton.com, New York University, USA; William Kuszmaul, wkuszmau@andrew.cmu.edu, Carnegie Mellon University, USA; Guido Tagliavini, guido.tag@rutgers.edu, Rutgers University, USA.

ACM 1549-6333/2024/10-ART
https://doi.org/10.1145/3700594

1 INTRODUCTION

How many bits does it take to store a pointer? If we know nothing about the pointer except that it references an item in an array of size n, then there is a lower bound of $\log n$ bits.

For many (and perhaps even most) uses of pointers, however, this information-theoretic lower bound does not apply. As we shall see in this paper, even a small amount of prior information about a pointer (e.g., a node's predecessor in a linked list) can be used to defeat the $\log n$ lower bound.

This paper introduces a general-purpose tool, which we call the ***tiny pointer***, for compressing pointers. In settings where pointers are used, tiny pointers can often be used instead to eliminate almost all of the space overhead of pointers.

What is a tiny pointer? Suppose n or more users (i.e., Alice, Bob, etc.) are sharing an array A of size n. A user can request a location in A via a function Allocate(), which returns a pointer p to a location that is now reserved exclusively for that user, if there is an available location; the user can later relinquish the memory location by calling a function Free(p). Each user promises only to allocate at most one memory location at a time.[1] For example, if Alice calls Allocate() to get a pointer p, she must call Free(p) before calling Allocate() again.

How large do the pointers p need to be? The natural answer is that each pointer uses $\log n$ bits. However, the fact that each pointer has a distinct owner makes it possible to compress the pointers to $o(\log n)$ bits. A critical insight is that the same pointer p can mean different things to different users, via the following scheme in which Allocate, Dereference, Free are given the user's ID as an additional argument. A user k can call Allocate(k) in order to get a tiny pointer p; they can dereference the tiny pointer p by computing a function Dereference(k, p) whose value depends only on k, p, and random bits; and they can free a tiny pointer p by calling a function Free(k, p).

The reason that tiny pointers are not constrained by the information-theoretic lower bound of $\log n$ bits is that k and p *together* encode the allocated location, rather than p alone. Thus this scheme provides a mechanism for how to use information already available about a pointer (namely, who "owns" the pointer) to compress the pointer to size $o(\log n)$ bits.

We refer to the algorithms for the functions Allocate(k)/Dereference(k, p)/Free(k, p), along with the array A and any associated metadata M, as a ***dereference table***. We will often refer to the users (i.e., the owners of tiny pointers) as ***keys*** and to the data stored at the allocated locations pointed at by the tiny pointers as ***values***. In practice, the "users" will often be components of a data structure that have some ownership relationship to the allocation being performed. A dereference table that stores q-bit values in an array of nq bits (and using $O(n)$ bits of metadata) is said to support ***load factor*** $1 - \delta$ if the table is capable of storing $(1 - \delta)n$ values at a time.

An ideal dereference table would simultaneously support a load factor with $\delta = o(1)$, tiny-pointer sizes of $o(\log n)$, and constant-time operations with high probability. As we shall discuss shortly, we prove a tradeoff curve between the best achievable load factor $1 - \delta$ and the best achievable tiny-pointer size s. Constructing optimal dereference tables on this tradeoff curve is one of the central questions of this paper.

Using tiny pointers to get tiny data structures. In addition to constructing dereference tables with tiny pointers, we show that such dereference tables can be used to obtain improved solutions for a number of classic problems:

- A data structure storing n v-bit values for n keys with constant-time modifications and queries can be implemented to take space $nv + O(n \log^{(r)} n)$ bits, for any constant $r > 0$, as long as the user stores a tiny pointer of expected size $O(1)$ with each key—here, $\log^{(r)} n$ is the r-th iterated logarithm.[2]

[1]A user k can request more than one location by creating a unique label ℓ for each of their allocations. In this case, we simply treat the "user" for the allocation as the concatenation $k \circ \ell$, so the user k can have multiple allocations without violating the uniqueness requirement.
[2]That is, $\log^{(1)} n := \log n$ and $\log^{(i+1)} n := \log \log^{(i)} n$.

- Any binary search tree storing n sortable keys in n nodes can be made ***succinct***, meaning that it achieves $(1+o(1))$ times the optimal space[3], with constant-factor time overhead, with constant time overhead, and can be made within $O(n)$ bits of optimal with $O(\log^* n)$-time modifications. This holds even for rotation-based trees such as the splay tree, which is conjectured to be dynamically optimal.
- Any fixed-capacity, e.g. non-resizable, key-value dictionary storing v-bit values can be made stable (i.e., items do not move once inserted) with constant time overhead an additive $O(\log v)$-bit space overhead per value.
- Any key-value dictionary that requires uniform-size values can be made to support arbitrary-size values with constant time overhead and with an additional space consumption of $\log^{(r)} n + O(\log j)$ bits per j-bit value, where $r > 0$ is an arbitrary constant.
- Given an external-memory array A of size $(1+\varepsilon)n$ containing a dynamic set of up to n key-value pairs, it is possible to maintain an internal-memory stash of size $O(n \log \varepsilon^{-1})$ bits so that the location of any key-value pair in A can be computed in constant time (and with no IOs).

What unifies these problems is that each is easy to solve space-inefficiently with pointers, and the difficulty in solving them space-efficiently stems from the challenge of eliminating the pointer overhead.

A theme throughout our uses of tiny pointers is the importance of having access to the full tradeoff curve of optimal tiny-pointer constructions. This is because of the need to balance two types of space overheads: that of storing the tiny pointers themselves, and that of storing the dereference table. The former is determined by tiny-pointer size and the latter is determined by load factor.

Relationship to dynamic perfect hashing. To understand what makes the tiny-pointer abstraction powerful, consider the following alternative approach to removing pointer overhead in the setting where each value has a unique owner: construct a dynamic perfect hash function that maps keys to slots in a densely packed array, and replace pointer dereferences with queries to this hash function. Such an approach has a certain elegance because it removes the pointers entirely. However, it also hits a fundamental bottleneck: any dynamic perfect hash function mapping n $(1+\Theta(1))\log n$-bit keys to $(1+\delta)n$ slots must use $\Theta(n \log\log n + n \log \delta^{-1})$ bits of metadata [3, 26].

The $n \log\log n$-bit term means that dynamic perfect hashing cannot be used to simulate pointers of size any smaller than $\log\log n$ bits. What makes our results on tiny pointers surprising is that, by *reducing* the lengths of pointers (rather than attempting to eliminate them entirely), one can blast through the $n \log\log n$ lower bound, enabling both our bounds on tiny pointers and the data-structural applications that we present in this paper.

This paper. In this paper, we first develop a comprehensive theory of tiny pointers. We consider both ***fixed-size tiny pointers*** (where all of the tiny pointers have the same size in bits) and ***variable-size tiny pointers*** (where the tiny pointers have sizes that are bounded in expectation, but different tiny pointers may have different sizes). For both types of tiny pointers we determine the optimal tradeoff curve between load factor and tiny-pointer size in dereference tables. We then go on to present the five applications of tiny pointers outlined above. As an ancillary result, we also reinterpret our tiny-pointer constructions as balls-and-bins results. In doing so, we improve on the known bounds for dynamic load balancing in some important parameter regimes.

1.1 Results: Constructing Optimal Tiny Pointers

In Sections 4, 5, and 6 we develop tight asymptotic bounds for the best achievable tradeoff curve between tiny-pointer size s and the dereference-table load factor $1-\delta$.

Optimal tradeoffs for fixed and variable-size tiny pointers. For fixed-size tiny pointers, we show that for any load factor $1-\delta \in \Omega(1)$, there is a lower bound of $\Omega(\log\log\log n)$ on the tiny-pointer size s. On the other hand,

[3] In this context, the optimal space to store a binary search tree is the space needed to (1) store the n nodes, each of which consists of a key and possibly a value; and (2) store the binary-tree structure, which comprises $2n + o(n)$ bits of information. The reason that traditional pointer-based search trees are not succinct is because they use $\Theta(n \log n)$ bits to store the binary-tree structure.

parameterizing by δ, we show that it is possible to achieve a fixed tiny-pointer size $s = O(\log\log\log n + \log \delta^{-1})$, and we give a lower bound showing that this tradeoff curve is tight.

We show that the $\log\log\log n$ barrier can be eliminated by instead using variable-size tiny pointers. We prove that for any load factor $1 - \delta$, it is possible to achieve average tiny-pointer size $s = O(1 + \log \delta^{-1})$, and again we prove that this tradeoff curve is tight for all δ.

For variable-size tiny pointers, our construction offers a remarkably strong concentration bound on each tiny pointer's size: if the expected size is k, then the probability of any given allocation returning a tiny pointer of size greater than $k + j$ for any $j > 0$ is *doubly exponentially small* in j.

All of our dereference-table constructions guarantee constant-time operations with high probability, that is, with probability $1 - 1/\text{poly } n$. Thus, tiny pointers can be integrated into data structures while incurring only a constant-factor time overhead.

Relationship to balls-and-bins games. In Section 8, we reinterpret our tiny-pointer results as balls-and-bins results. Notably, we are able to apply our techniques to the dynamic load-balancing problem, a.k.a. balanced allocations [8], where there are n bins and up to $m = nh$ balls present at a time: for $h \geq 1$, we give a balls-and-bins scheme with $d + 1$ hash functions that achieves maximum load $h + O(\sqrt{h \log(hd)}) + \frac{\log\log n}{d \log \phi_d}$, which significantly improves the state of the art [59, 60] when $hd = o(\log n)$.

To understand the relationship between dereference tables and balls-and-bins schemes, think of keys as balls that must be assigned to distinct bins. Each ball x has some probe sequence $h_1(x), h_2(x), \ldots \in [n]$ of bins where it can be placed. Supporting tiny pointers of size $O(s)$ is equivalent to maintaining a dynamic balls-to-bins assignment such that each ball x is in some bin $h_i(x)$ satisfying $i \leq 2^{O(s)}$.

What makes this balls-and-bins problem interesting is that the same ball can be inserted, removed, and subsequently reinserted over time. The first time that a ball is inserted, its probe sequence $h_1(x), h_2(x), \ldots$ is independent of the dereference table's state. But if the ball is removed and then later reinserted, then this is no longer the case: the state of the dereference table has now been affected by (and is partially a function of) the probe sequence. The result is that, in this fully dynamic setting, even the behavior of very simple balls-and-bins schemes (e.g., random probing [43] or linear probing [42, 56]) have resisted theoretical analysis.[4]

A key insight in constructing small tiny pointers is that, by designing the probe sequence of each "ball" to have a certain careful structure, we can achieve small probe complexity (and thus small tiny pointers) for an arbitrary sequence of ball insertions and removals. The same techniques are also what allow us to revisit other related problems such as dynamic load-balancing in bins with unbounded capacities.

Relationship to succinct hashing. At a technical level, our constructions for tiny pointers share an interesting relationship to the constructions used in past works on succinct hashing [6, 10, 12, 26], where a common theme is the use of a *backyard* to store a small fraction of items less space-efficiently than their peers. Our tiny-pointers construction (specifically, in the variable-size case) reveals an alternative way to use backyards: now the items in the backyard have less space-efficient *tiny pointers* than their peers, and rather than there being a single backyard, there is a cascading hierarchy of backyards that must interact cleanly with one another. A key technical insight of this paper is that such a hierarchy of backyards can be implemented (i) with constant-time operations, (ii) while supporting arbitrary sequences of both insertions and deletions, (iii) without moving items once they are inserted, (iv), without having to store the keys corresponding to items in the data structure, and (v) with sufficiently high-probability guarantees that each layer in the cascade behaves in a predictable way.

[4]Work in this setting typically treats linear probing and random probing as techniques for building an open-addressed hash table. In the setting where balls cannot be moved after being placed (or equivalently, where hash-table deletions are implemented with tombstones), the only known bound on either random probing or linear probing is due to Larson [43], who analyzed random probing with random insertions/deletions.

1.2 Results: Five Applications to Data Structures

We now describe our five applications of tiny pointers in more detail. The first application is to the classic data-structural problem of storing a dynamic set of values associated with keys. The next three applications are each black-box transformations in which we show how to remove space inefficiency from large classes of data structures. And the final application is a new data structure for a classic problem in external-memory storage.

Overcoming the $\Omega(\log \log n)$-bit lower bound for the cost of data retrieval. Our first application revisits the classic ***data-retrieval problem*** [3, 26, 27, 29], in which a data structure must store a v-bit value for each of the k-bit keys in some set S, and must answer queries that retrieve the value associated with a given key.[5] In the static case, where the keys/values are given up front, it is possible to solve the retrieval problem with $O(1)$-time queries using $nv + O(\log n)$ bits of space [27, 29]; but in the dynamic case where keys/values are inserted/deleted over time, and there are up to n keys/value pairs present at a time (with keys taken from some large polynomial-size universe), it is known that any solution to the retrieval problem must use a lower bound of $nv + \Omega(n \log \log n)$ bits of space, even if super-constant-time operations are allowed [3, 26]. This means that the number of metadata bits per value is $\Omega(\log \log n)$ on average, even if the values are of size $v = o(\log \log n)$.

We show that, by just slightly modifying the specification of the retrieval problem, we can completely dissolve the $\Omega(\log \log n)$-wasted-bits-per-item lower bound. Suppose, in particular, that whenever the user inserts a key/value pair (x, y), they are given back a small *hint* h that they are responsible for storing. (We will guarantee that the hint has constant expected size.) In the future, when the user wishes to recover the value y for x, they present both the key x and the hint h to the retrieval data structure. We call this the ***relaxed retrieval problem*** and we refer to the hints as ***tiny retrievers***.

The relaxed retrieval problem can also be viewed as a relaxation of the tiny-pointer problem: the tiny retriever h is analogous to a tiny pointer, except that the pair (x, h) does not have to fully encode the position of y—instead, the relaxed-retrieval data structure can make use of not just x and h, but can also make use of a small auxiliary data structure whose purpose is to help recover y.

Given that we have already stated tight bounds for tiny pointers, it is tempting to assume that the same bounds should hold for tiny retrievers. We find that this is not so. We show how to construct tiny retrievers of expected size $O(1)$, while supporting queries in constant time (with high probability), and allowing for the following tradeoff curve: using time $\Theta(r)$ for insertions/deletions, the size of the data structure becomes $nv + O(n(1 + \log^{(r)} n))$ bits. So, with constant-time operations, we can achieve size, say, $nv + O(n \log \log \log \log \log n)$, and with $O(\log^* n)$-time operations, we can achieve size $nv + O(n)$. Moreover, in the special case where the value length v is sub-logarithmic, satisfying $v \le \frac{\log n}{\log^{(r)} n}$, the space consumption reduces to $nv + O(n)$ bits, even for constant r.

Remarkably, our construction for tiny retrievers is *itself a direct application of tiny pointers*—in fact, tiny retrievers are simply variable-length tiny pointers of $O(1)$ expected size. This is because the ability to construct $O(1)$-length tiny pointers into an array with $\Theta(n)$ entries ends up allowing for us to reduce the relaxed retrieval problem to the dictionary problem, for which highly space-efficient solutions are known [12].

We remark that the distinction between tiny pointers and tiny retrievers ends up being significant in several of our applications below. In some cases, tiny retrievers offer a path to remarkable (and unexpected) space efficiency, while in other cases, the smooth tradeoff curve and pointer-like behavior offered by tiny pointers makes them a better fit. The advantage of tiny retrievers is that they offer a steep tradeoff between time and space; the advantage of tiny pointers is that they offer indirection-less reference to items, as well as a flexible tradeoff between different *types* of space consumption (pointer size and load factor).

[5] Note that queries are required to be for a key $x \in S$—the data structure is allowed to return an arbitrary value if $x \notin S$.

Succinct rotation-based binary search trees. To describe our second application, we first take a digression into the world of succinct binary trees. Since there are at most 4^n ordered binary trees on n nodes, the pointer structure of a binary tree can be encoded in $O(n)$ bits. This observation has led to a great deal of work on optimal (and near-optimal) encodings of binary trees [24, 25, 33, 35, 48, 49, 52, 55]. Apart from navigation, state-of-the-art trees also support a wide variety of query operations (e.g., subtree size [24, 33, 48, 49, 55], depth [24, 49], lowest-common ancestor [24, 49], level ancestor [24, 49], etc.), while also supporting basic dynamism (e.g., inserting/removing leaves [24, 33, 48, 49, 55], inserting a node in the middle of an edge [24, 33, 48, 49, 55], compacting a path of length two [24, 33, 48, 49, 55], etc.).

One natural form of dynamic operation has proven elusive, however: the known succinct binary trees do not efficiently support rotations. The lack of support for rotations is especially important for binary *search trees*, which store a set of n sortable keys in n nodes. Almost all dynamic balanced binary search trees (e.g., AVL trees [2], red-black trees [38], splay trees [58], treaps [57], etc.) rely on rotations when modifying the tree. None of these tree structures can be encoded with the known succinct-tree techniques.

We give a randomized black-box approach for transforming dynamic binary search trees into succinct data structures. If there are n keys in the succinct search tree, each of which is k bits long, then the size of the succinct search tree will be $nk + O(n \log^{(r)} n)$ bits. The transformation induces only a constant-factor time overhead on query operations, and only an $O(r)$-factor time overhead on tree modifications. So, for example, if we set $r = O(\log^* n)$, then edge traversals take time $O(1)$, edge insertions/deletions take time $O(\log^* n)$, and the tree structure is encoded using $O(n)$ bits. In contrast, the previous state of the art [49] for implementing rotations in space-efficient binary search trees also encoded the tree structure in $O(n)$ bits (actually, $2n + o(n)$ bits) but required $\tilde{\Omega}(\log n)$ time to implement a single rotation. It is worth noting that [49] is deterministic, while the new result succeeds with high probability.

When r is set to be $O(1)$, the fact that running times are preserved means that other properties, such as dynamic optimality, are as well. For example, if the splay tree [58] is dynamically optimal (as the widely believed Dynamic-Optimality Conjecture [58] posits), then so is the succinct splay tree.

Space-efficient stable dictionaries. Our third application is a black-box approach for transforming any key-value dictionary that stores its values in a fixed-capacity, e.g. non-resizeable, array into a stable dictionary with the same operation set and with only a constant-factor time overhead. If the original dictionary stores v-bit values, then the new stable dictionary also stores v-bit values, and uses space equal to the space of the orginal data structure plus $O(\log v)$ bits per value.

Formally, a key-value dictionary (e.g., a binary search tree, hash table, etc.) is ***stable*** if whenever a key-value pair is inserted, the position in which the value is stored never changes. (This property is sometimes also referred to as referential integrity [56] or value stability [10].) Stability ensures that users can maintain pointers into a data structure without those pointers becoming invalidated by changes to the data structure [39, 56]. Stability is a strict requirement in many library data structures [16–23] (and it is a core reason why high-performance languages such as C++ use chained hashing [18, 21], which is stable, instead of more space-efficient alternatives, such as linear probing [41, 53] or cuckoo hashing [30, 34, 51]).

Empirical research on achieving stability in space-efficient hash tables dates back to the 1980s [39, 56] (see also the discussion in Knuth's Volume 3 [42, Chapter 6.4]) and the resulting techniques have been built into widely-used hash tables released by Google [1] and Facebook [32]. On the theoretical side, if a data structure is storing k-bit keys and v-bit values, where $k, v = O(\log n)$, it is known how to achieve stability at the cost of an extra $\Theta(\log \log n)$ bits of space per value [26], but it is not known whether $\Omega(\log \log n)$ bits per value are *necessary*.[6] Our result shows that it is not—stability can be achieved with $O(\log v)$ extra bits per value. This is especially noteworthy in cases

[6]Interestingly, there are several specific approaches for which $\Omega(\log \log n)$ bits per value are known to be necessary, for example if stability is achieved via perfect hashing (see Theorem 2 of [26]).

where the value-size v is small[7]. Our result applies to arbitrary fixed-capacity dictionaries, including, for example, the succinct splay tree constructed above.

Space-efficient dictionaries with variable-size values. Our fourth application is a black-box approach for transforming any key-value dictionary (designed to store fixed-size values) into a dictionary that can store different-sized values for different keys. The resulting data structure incurs a constant-factor time overhead and offers the following guarantee on space efficiency. Let $\log^{(r)} n$ be the r-th iterated logarithm and set r to be a positive constant of our choice. The new data structure incurs an additive space overhead of only $O(\log^{(r)} n + \log |x|)$ bits for each value x, where $|x|$ is the bit-length of the string x. (Interestingly, the iterated logarithm $\log^{(r)} n$ in this application comes from an entirely different source than in our previous applications.)

The ability to store variable-length values also yields a simple solution to the ***multi-set problem***, which is the problem of how to design a space-efficient constant-time hash table that stores multi-sets of keys (rather than just sets). The multi-set problem was first posed as an open question by Arbitman et al. [6], who gave a succinct constant-time hash table capable of storing sets but not multi-sets. A series of subsequent works gave solutions to the multi-set problem, first in the case of random multi-sets [15], and then very recently for arbitrary multi-sets [14]. The known solutions come with a drawback, however: the bound on space is the same for duplicate keys as it is for non-duplicate keys. So, if there are m_i copies of some key, then they are permitted to take m_i times as much space as a single copy would, even though, in principle, $m_i - 1$ of the copies could be encoded using an $\log m_i$-bit counter. Our transformation gives a simple alternative solution that avoids this drawback and that can even be applied directly to the original hash table of Arbitman et al. [5]: by storing the multiplicity of each key as a (variable-length) value, one can support arbitrary multisets at an additional space cost of only $\log^{(r)} n + \log m_i + O(\log \log m_i)$ bits per key, where m_i is the multiplicity of the key and r is a positive constant of our choice; this is remarkably space efficient considering the fact that $\log m_i$ bits are needed just to store the multiplicity. A nice feature of our solution is that it also applies directly to other dictionaries such as, for example, the succinct splay tree discussed earlier in the section.

An optimal internal-memory stash. Our final application of tiny pointers revisits one of the oldest problems in external-memory data structures: the problem of maintaining a small internal-memory stash that allows one to directly locate where items reside in a large external-memory array.

In more detail, the problem can be described as follows [36]. We are given an (initially blank) external-memory array with $(1 + \varepsilon)n$ slots, for some parameters ε, n. We must maintain a dynamically changing set S of key-value pairs (where keys are distinct) in the array, such that each time a key-value pair (x, y) is inserted into S, the pair (x, y) is assigned some permanent position where it resides in the external-memory array. We must then also maintain a small internal-memory data structure X, known as a ***stash***, that can be used to recover, for each key x, precisely where its key-value pair (x, y) is stored in the external-memory array. A stash enables queries to be performed in a *single* access to external memory.

Work on designing space-efficient and time-efficient stashes dates back to the late 1980s [36, 44, 45], and is also closely related to the problem of designing space-efficient page tables in operating systems [4, 9, 40]. The best-known theoretical results are due to Gonnet and Larson [36], who give a stash that uses only $O(n \log \varepsilon^{-1})$ bits of space. A consequence is that, if $\varepsilon = \Theta(1)$, the stash uses only $O(n)$ bits.

Gonnet and Larson's result comes with several drawbacks, however [36]. First, the stash only offers provable guarantees in the setting where insertions/deletions to S are random; in the case where S is modified by an arbitrary sequence of insertions/deletions/queries, the problem of designing a space-efficient stash remains open. Second,

[7]One especially remarkable consequence is the following: if we wish to store $O(1)$ control bits associated with each key in a data structure, and we wish for the positions of those bits to be stable so that a third party who does not have access to the data structure can still access/modify the control bits, then we can accomplish this with only $O(1)$ extra bits of space per item.

the internal-memory operations on the stash of [36] are not constant-time in the RAM model (or even constant expected time, when $\varepsilon = o(1)$).

By combining tiny pointers with modern techniques for constructing space-efficient filters, we show that it is possible to construct a stash of size $O(n \log \varepsilon^{-1})$ bits that supports constant-time operations in the RAM model (not just in expectation, but even with high probability) and that supports *arbitrary* sequences of insertions/deletions/queries.

2 PRELIMINARIES

In this section, we give some preliminary definitions and notation.

Operations. A dereference table with q-bit-values is a data structure that supports the following operations:

- $\textsc{Create}(n, q, \delta)$: The procedure creates a new dereference table, and returns a pointer to an array with n slots, each of size q bits. We call this array the ***store***. The dereference table will be capable of supporting up to $(1-\delta)n$ concurrent allocations at at time. We require that $\delta = O(1/q)$.
- $\textsc{Allocate}(x)$: Given a key x, the procedure allocates a slot in the store to x, and returns a bit string p, which we call a ***tiny pointer***.
- $\textsc{Dereference}(x, p)$: Given a key x and a tiny pointer p, the procedure returns the index of the slot allocated to x in the store. If p is not a valid tiny pointer for x (i.e., p was not returned by a call to $\textsc{Allocate}(x)$), then the procedure may return an arbitrary index in the store.
- $\textsc{Free}(x, p)$: Given a key x and a tiny pointer p, the procedure deallocates slot $\textsc{Dereference}(x, p)$ from x. The user is only permitted to call this function on pairs (x, p) where p is a valid tiny pointer for x (i.e., p was returned by the most recent call to $\textsc{Allocate}(x)$).

We say a key x is ***present*** or ***allocated*** if it has been allocated more recently than it has been freed; in this case the tiny pointer p returned by the most recent call to $\textsc{Allocate}(x)$ is said to be x's tiny pointer. The user is only permitted to allocate at most one tiny pointer p to each key x. That is, each time that $\textsc{Allocate}(x)$ is called to obtain some tiny pointer p, the function $\textsc{Free}(x, p)$ must be called before $\textsc{Allocate}(x)$ can be called again.

We say that slot i in the store is ***occupied*** if there is a present key x with tiny pointer p such that $\textsc{Dereference}(x, p) = i$, and otherwise we say it is ***free***. We call occupied slots ***items***. We typically refer to the parameter n (i.e., the number of slots in the store) as the table's ***size*** or ***capacity***.

Guarantees. Dereference tables provide the following guarantees:

- For any two present keys $x_1 \neq x_2$ with tiny pointers p_1 and p_2, respectively, $\textsc{Dereference}(x_1, p_1) \neq \textsc{Dereference}(x_2, p_2)$.
- $\textsc{Dereference}(x, p)$ only depends on x, p, random bits, and the parameter n. One consequence is that, once a key is allocated a slot in the store, the position of that slot cannot change until the key is subsequently freed and reallocated.

The second property ensures that the act of dereferencing a tiny pointer is similar to the act of dereferencing a standard pointer; in both cases, one does not need to access the data structure being pointed into in order to perform the dereference. This ends up being important for several of our applications later. In particular, it ensures that in external-memory applications, each dereference incurs only a single I/O; and it ensures that in data-structure applications, the locations pointed at by tiny pointers are stable (i.e., once a tiny pointer p is allocated to a key x, the location that is being pointed at does not change).

Space. The dereference table can support up to $(1-\delta)n$ allocations at a time—the quantity $1-\delta$ is referred to as the table's ***load factor***. If the ALLOCATE function is called when there are already $(1-\delta)n$ allocations performed that have not been freed, then the dereference table is permitted to fail the allocation.[8]

The dereference table may store metadata in order to perform updates (allocations and frees) efficiently. Metadata can either be stored as part of the store (in slots that are not allocated), or in an auxiliary data structure that is permitted to consume up to $O(n)$ bits. In other words, the dereference table is allowed to use $O(n)$ bits (i.e., $O(1)$ bits of overhead per slot) of metadata for "free", without that counting towards the space consumption of the store, but any additional metadata must count towards the space consumption of the store. Note that the dereference table is not allowed to store metadata in any slot of the store that is currently allocated.

We can now see why it is natural to require that $\delta \leq O(1/q)$. Since dereference tables can use up to $O(n)$ space for metadata, the total amount of space consumed by a dereference table may be as large as $nq + O(n) = (1-\delta)nq + \delta nq + O(n)$. The first term $(1-\delta)nq$ is space that allocations can make use of, the second term δnq is space that is allocated but not used, and the third term $O(n)$ is metadata. The second and third terms δnq and $O(n)$ cumulatively represent the total amount of space not used by allocated objects. There is no point in the user specifying a value of δ, q that results in $\delta nq = o(n)$, because this does not reduce the total amount of extra space below $O(n)$. Thus, we can assume without loss of generality that the user is constrained to $\delta \leq O(1/q)$.

Failure probability. We will permit allocations to have a small failure probability. That is, each allocation is permitted to fail with probability $1/\text{poly}(n)$,[9] in which case the allocation simply returns a failure message rather than a tiny pointer. In general, if a random event occurs with probability $1 - 1/\text{poly}(n)$, we say that it occurs ***with high probability (w.h.p.)***. Note that here, and throughout the paper, we use $\text{poly}(n)$ to mean n^c for some large positive constant c of our choice.

We remark that, when analyzing dereference tables, we shall always assume that the sequence of allocations, frees, and dereferences are determined by an oblivious adversary (i.e., the sequence is determined ahead of time, rather than adapting to the behavior of the dereference table). One consequence of this is that, if a given allocation fails, the only effect on the operation sequence is that the corresponding call to FREE is removed.

Hashing and independence. Our dereference-table constructions will all make use of hash functions. For simplicity, we shall treat hash functions in this paper as being uniform and fully independent. This assumption is without loss of generality since there are already known families of hash functions [31, 50] that simulate n-independence with constant-time evaluation and linear space, and there are already well understood techniques [6, 28, 46] for applying these families to data structures that require $n^{O(1)}$-independence, while using space only $\tilde{O}(n^{\varepsilon})$ bits to store the hash function.[10] These known techniques can easily be applied directly to all of our data structures; the only caveat is that the families of hash functions being used [31, 50] introduce their own additional $1/\text{poly}(n)$ failure probability to the data structure. So, even if a data structure offers sub-polynomial failure probability under the assumption of fully random hash functions, if we wish to use an explicit family of hash functions, then we must allow for a $1/\text{poly}(n)$ failure probability.

3 WARMUP: A SIMPLE CONSTRUCTION AND A SIMPLE APPLICATION

To ease the reader into the notion of a tiny pointer, we we begin in this section with two simple but illustrative warmups.

[8]Note that, even though a dereference table only guarantees the ability to store up to $(1-\delta)n$ allocations at a time, we still use the terms "size" and "capacity" of a dereference table to refer to n, rather than $(1-\delta)n$, since n represents the total number of q-bit entries in the store.
[9]Specifically, this means that the dereference table depends on some constant $c > 0$ and fails with probability at most $1/n^c$.
[10]The basic idea is to replace the data structure of capacity n with $n^{1-\varepsilon}$ data structures of capacity n^{ε}. Each item x in the full data structure gets hashed at random to one of the $n^{1-\varepsilon}$ data structures (using an $O(1)$-independent hash function), each of which only requires $(n^{\varepsilon})^{O(1)} = o(n)$ independence.

A simple dereference table. Our first warmup is a tiny-pointer construction that supports $q \geq \log n$ and $\delta = 1/\log n$. This construction will not be sufficient for any of our applications in Section 7, but it does illustrate some of the basic principles for how to design a dereference table. Additionally, it serves as a simple demonstration of how, once we have the *abstraction* of a tiny pointer, it is actually relatively simple to get from there to a nontrivial result.

Theorem 3.1 (Warmup Construction). *Let $q \geq \log n$ and $\delta = 1/\log n$. There is a dereference table for q-bit values that (i) succeeds on each allocation w.h.p., (ii) has load factor $1 - \delta$, (iii) has constant-time operations, and (iv) produces tiny pointers of size $O(\log \log n)$ bits.*

Our construction will make use of the following basic fact:

Claim 3.2. *Suppose we throw $(1 - \delta)n$ balls into n/b bins, where the throws are independent and uniformly random. If $\delta = 1/\log n$ and $b = \log^4 n$, then we have w.h.p. in n that every bin contains fewer than b balls.*

Proof. It suffices to show that, w.h.p., each individual bin contains fewer than b balls. For a given bin, the number of balls that go to that bin is a sum X of i.i.d. indicator random variables with mean $\mu = b \cdot (1 - \delta)$. By a Chernoff bound, we have for any $k \in O(\sqrt{\mu})$ that

$$\Pr[X \geq \mu + k\sqrt{\mu}] \leq 2^{-\Omega(k^2)}.$$

Plugging in $k = \log n$ we can conclude that

$$\Pr[X \geq \mu + \log n \cdot \sqrt{mu}] \leq 2^{-\omega(\log n)} \leq 1/\text{poly}(n).$$

Since $\mu + \log n\sqrt{\mu} = (1 - \delta)b + \log n \cdot \sqrt{(1 - \delta)b} \leq (1 - \delta + \log n/\sqrt{b}) \cdot b = b$, we can conclude that

$$\Pr[X \geq b] \leq 1/\text{poly}(n),$$

as desired. □

We can make use of Claim 3.2 to prove Theorem 3.1 as follows.

Proof of Theorem 3.1. We partition the store into n/b buckets, each of which has $b = \log^4 n$ slots. Each key x hashes to a random bucket $h(x) \in [n/b]$. Whenever a key x is allocated, it is allocated one of the b slots in bucket $h(x)$. If the key is allocated the p-th slot in the bucket, then the number p is returned as the tiny pointer for x. Not only does this result in tiny pointers of length $\log b = O(\log \log n)$ bits, but it also makes the dereference function trivial to implement: The function Dereference(x, p) simply returns a pointer to the p-th slot of bin $h(x)$.

It's important to emphasize the role that Claim 3.2 plays here. The claim tells us that, whenever a key x is allocated, we have w.h.p. that bin $h(x)$ contains at least one free slot that x can use. In the low-probability even that there is no such free slot, we can afford to declare failure.

Finally, there is one nontrivial algorithmic question that we must tackle: How can the allocation function efficiently find a free slot in bucket $h(x)$? Within each bucket, we store a ***free list***, which is a linked list keeping track of the free slots in the bucket. The internal nodes of the linked lists can be kept in the store, with the node that represents a given free slot s being stored *in that slot.* Additionally, each bucket must store the head pointer for its linked list as external metadata, but this takes very little space, coming out to $O(\log n) \cdot n/b = o(n)$ bits. With a free list for each bin, it becomes straightforward to implement allocations/deallocations in $O(1)$ time.

In summary, whenever a key x is allocated, it is assigned to a free slot within bin $h(x)$. If it is given the p-th slot of the bin, then the number p acts as its tiny pointer. If the same key x is later dereferenced, then the function Dereference(x, p) simply returns a pointer to the p-th slot of bin $h(x)$. We know by Claim 3.2 that each allocation will succeed w.h.p., and we can use a free list within each bin to implement both allocations and deallocations in time $O(1)$. □

Although the above construction is quite a bit simpler than most of the constructions that will appear later on, it is nonetheless a good starting place for how to think about constructing tiny pointers. It demonstrates the important role that hash functions serve, and the relationship between tiny pointers and balls-and-bins games.

A simple application: binary search trees. As a simple application, let us consider the task of compressing a rotation-based binary search tree. We will present this application in more detail (and with much better bounds) in Section 7, but for now, we will use it a simple example of how to apply tiny pointers.

In a standard search tree, each node stores three things: a key k, a left-child pointer p_1, and a right-child pointer p_2. We can replace p_1 and p_2 with tiny pointers that are each dereferenced using key k. So, for example, to determine where the node's left child is stored, we simply calculate $\textsc{Dereference}(k, p_1)$.

Part of what is nice about this approach is that it allows for straightforward edits to the search tree. If we want to perform a rotation, we just need to update $O(1)$ tiny pointers, which, in turn, corresponds to performing $O(1)$ allocations/deallocations.

With a bit of care, this approach can be used to reduce the space used per pointer to $O(\log \log n)$ bits per node. This bound is far from optimal (and can also be achieved with already-known data-retrieval techniques, see, e.g., [26]). Nonetheless, it is a good example of how to use tiny pointers, and it is a demonstration of how even our warmup construction (Theorem 3.1) can be used to get results that, *a priori*, are nontrivial.

4 UPPER BOUND FOR FIXED-SIZE POINTERS

In this section, we give optimal constructions for fixed-size tiny pointers. We prove the following theorem:

Theorem 4.1. *Let $\delta \in (0, 1)$ be a parameter. There is a dereference table for q-bit values, for any q, that (i) succeeds on each allocation w.h.p., (ii) has load factor at least $1 - \delta$, (iii) has constant-time dereferences and has constant-time updates w.h.p., and (iv) has tiny pointers of size $O(\log \log \log n + \log \delta^{-1})$.*

In particular, for $\delta = 1/\log \log n$, we get tiny pointers of size $O(\log \log \log n)$. Thus, we can doubly-exponentially beat raw $\log n$-bit pointers, while still supporting a load factor of $1 - o(1)$.

The proof is the simplest of our tiny-pointer constructions, and makes use of two algorithmic building blocks.

The first building block: load-balancing tables. A load-balancing table is a simple type of dereference table that has a very specific internal representation, and that, unlike normal dereference tables, is permitted to fail on calls to $\textsc{Allocate}$ with a probability larger than $1/\text{poly}(n)$. Roughly speaking, if a load-balancing table has load factor $1 - \delta$, then the load-balancing table is permitted to fail on a δ-fraction of allocations.

Load-balancing tables are implemented as follows. If the store is of some size m, then we partition it into m/b buckets of size $b = \Theta(\delta^{-2} \log \delta^{-1})$, where the constant in the Θ notation is selected to be sufficiently large. To allocate a slot for key x, we hash x into one of the buckets, using a hash function h. If bucket $h(x)$ contains a free slot, then we allocate any free slot $i \in [b]$ within that bucket, and we return i as the tiny pointer. Otherwise, all b slots in the bucket are occupied, and the allocation fails. The function $\textsc{Dereference}(x, i)$ can then be implemented to simply return the i-th slot in bin $h(x)$.

Load-balancing tables will serve as a building block in the dereference tables that we construct. The basic idea is that we can use a load-balancing table to handle all but a δ-fraction of allocations, and the remaining allocations can be handled via some other mechanism. Thus, we will need the following lemma which bounds the total number of failed allocations that are alive at any given moment (where we consider each allocation to be ***alive*** up until the time at which the corresponding free occurs, even if the allocation fails).

Lemma 4.2. *Consider a load-balancing table with size m and load factor $1 - \delta$, where $\delta \leq 1/2$. Consider a sequence of allocations and frees, where at most $(1 - \delta)m$ allocations are alive at a time. Then, at any moment, the number of allocations that have failed and are still alive is δm with probability at least $1 - \exp(-\delta^{O(1)} m)$.*

We remark that in all of our applications of Lemma 4.2, we will have w.l.o.g. that $\log \delta^{-1} = o(\log m)$ (since, otherwise, we would have $\log \delta^{-1} = \Omega(\log m)$ and so could just use standard $O(\log m)$-bit pointers instead of dereference tables). Thus the probability bound offered by the lemma will always be at least $1 - \exp(-m^{1-o(1)}) \geq 1 - 1/\text{poly}(m)$.

We defer the proof of Lemma 4.2 to Section 8.1, which establishes a more general version of the lemma. Although the proof is nontrivial, due to interdependencies that form from the same key potentially being allocated/freed/re-allocated many times, we do not view it as one of the main technical contributions of this paper. This is because Lemma 4.2 follows easily from a lemma established in our recent paper on space-efficient hash tables [10]. Still, we present an alternative proof in Section 8.1 both for completeness, as well as because the proof takes a somewhat different (and more elegant) approach than in our past work, and in order to cover a larger parameter regime.

To conclude our discussion of load-balancing tables, we must describe how to implement allocations and frees in constant time. Here, there are two cases, depending on how b compares to the size n of the dereference table that the load-balancing table is being used within.

If $b \leq \log n$, then we can store a b-bit bitmap for each bucket indicating which slots in the bucket are free; and we can use standard bit-manipulation on the bitmap to implement the allocation and free functions in constant time.

We take a different approach if $b \geq \log n$. In this case, we claim that without loss of generality, $q = \omega(\log b)$, where q is the size in bits of each of the items being stored (we will prove this claim in a moment). This claim means that we can keep track of which slots are free in each bucket of a load-balancing table as follows: we simply store a ***free list*** in each bucket, that is, a linked list consisting of all the free slots, where each free slot contains a pointer to the next free slot in the list. This is possible since each free slot is q bits and each pointer in the linked list needs only $\log b = o(q)$ bits. The $\log b$-bit base pointers of the m/b linked lists can be stored in an auxiliary metadata array of size $O((m/b) \cdot \log b) \leq O(m)$, where m is the size of the load-balancing table. The free lists makes it possible to implement the allocation and free functions in constant time.[11]

To prove that this free-list approach works, it remains to show that $q = \omega(\log b)$ without loss of generality. Let $1 - \delta$ be the load factor of the full dereference table (that the load-balancing table is part of) and let $1 - \gamma$ be the load factor of the load-balancing table. Since $b \geq \log n$, we must have $\gamma^{-1} = \tilde{\Omega}(\sqrt{\log n})$. In all of our constructions of dereference tables (i.e., the constructions in both this section and in Section 5), if we use a load-balancing table with load factor $1-\gamma$ satisfying $\gamma^{-1} = \tilde{\Omega}(\sqrt{\log n})$ (or even $\gamma^{-1} = \omega(\log \log n)$), we will always have $\log \delta^{-1} \geq \Omega(\log \gamma^{-1})$. Recall that, if a dereference table has load factor $1 - \delta$, then it is assumed that the dereference table is storing objects of size $q \geq \Omega(\delta^{-1})$ bits. Thus, we have that $q = \omega(\log \delta^{-1}) = \omega(\log \gamma^{-1}) = \omega(\log b)$, as desired.

The second building block: a power-of-two-choices dereference table. To compensate for the higher than desired failure probability of load-balancing tables, we develop our second building block: a simple dereference table that supports $O(\log \log \log n)$-bit tiny pointers and has a lower failure probability than a load-balancing table. The downside of this second building block is that it only supports a very small load factor.

Lemma 4.3. *There exists a δ satisfying $1 - \delta = \Theta(1/\log \log n)$, such that there is a dereference table that (i) succeeds on each allocation w.h.p., (ii) has load factor at least $1 - \delta$, (iii) has constant-time updates w.h.p., and (iv) has tiny pointers of size $O(\log \log \log n)$.*

The proof of Lemma 4.3 will make use of a celebrated balls-and-bins result [59, 60]—for more background on this result, see also Section 8.

Proof. We partition the store into buckets of size $b = \Theta(\log \log n)$. When $\text{ALLOCATE}(x)$ is called, the key x is hashed to two buckets $h_1(x), h_2(x) \in [1, n/b]$. The key x is allocated a slot in whichever of the two buckets contains

[11]It is tempting to try to store the metadata for the free list in the slots that are themselves free—however, a dereference table must be able to support even the case where q is very small, meaning that the metadata per free slot could actually exceed the size of the slot.

the most free slots. The tiny pointer p is $1 + \log b = O(\log\log\log n)$ bits long , and indicates which slot in the two buckets was allocated.

We can think of the allocations as balls that are inserted into bins using the power-of-two-choices rule [59, 60], with the same ball possibly being inserted/deleted/reinserted over time. Since the load factor is $\Theta(1/\log\log n)$, the expected number of balls in each bin is $O(1)$. In this setting, it is known that, w.h.p., the number of balls in the fullest bin is $O(\log\log n)$ [59, 60]. Thus allocations succeed w.h.p.

Finally, to implement allocations and frees in constant time, we can just use a bitmap to keep track of which slots in each bucket are free; since each bucket is only $O(\log\log n)$ slots, the bitmaps are each only $O(\log\log n)$ bits, and thus each bitmap fits into a machine word. Using standard bit manipulation, the bitmaps can be used to keep track of which slots are free in constant time per allocation/free (and to find a free slot for a given allocation also in constant time). The bitmaps consume a total of $O(n)$ bits of space. □

Putting the pieces together. Of course, power-of-two-choices dereference tables are not very useful on their own, because they only support $o(1)$ load factors, whereas load-balancing tables have too high a probability of failure on allocation. We now show how to combine the two data structures in order to prove Theorem 4.1.

Proof of Theorem 4.1. Since we are willing to have tiny pointers of size $\Theta(\log\log\log n + \log\delta^{-1})$, we can assume without loss of generality that $\delta = o\left(\frac{1}{\log\log n}\right)$.

We store a $1-\delta^2$ fraction of the allocations in a load-balancing table of size $m = (1-\delta/2)n$ slots that supports load factor $1-\delta^2/c$ for some sufficiently large positive constant c; we call this the ***primary table***. Allocations that fail in the primary table are handled in a secondary table implemented with Lemma 4.3 to have size $n' := \delta n/2$ slots and support load factor $1-\delta' := \Theta(1/\log\log n')$. If an allocation fails in the secondary table, or if the load factor of the secondary table ever exceeds $\Theta(1/\log\log n')$, then the allocation fails in the full dereference table as well. Note that the total size (in terms of slots) of the primary and secondary tables is n. See Figure 1 for a picture of the layouts of the two tables.

Since both the primary and secondary tables have constant-time operations, so does the full dereference table. Additionally, each allocation can return a tiny pointer that is either in the primary table or in the secondary table (plus 1 bit of information indicating which table it is being pointed into). Since the primary and secondary tables both have tiny pointers of size $O(\log\log\log n + \log\delta^{-1})$, the claim about tiny-pointer size is also proven.

Our final task is to bound the probability of a given allocation failing. Lemma 4.2 tells us that the number of keys allocated in the secondary table will be a most $\delta^2 n$ at any given moment w.h.p. Since the secondary table has $n' = \Theta(\delta n/2)$ slots, and since $\delta = o\left(\frac{1}{\log\log n}\right)$, it follows that the number of allocations in the secondary table at any given moment is $o(n'/\log\log n) = o(n'/\log\log n')$ with high probability. We therefore get from Lemma 4.3 that the allocations in the secondary table each succeed with high probability in n'. Without loss of generality, $n' \geq \sqrt{n}$ (since otherwise $\delta \leq O(1/\sqrt{n})$, and we can just use standard $\log n$-bit pointers). Thus the allocations in the secondary table each succeed with high probability in n. □

5 UPPER BOUNDS FOR VARIABLE-SIZE POINTERS

In this section, we give optimal constructions for variable-size tiny pointers. We prove the following theorem:

Theorem 5.1. *Let $\delta \in (0,1)$ be a parameter. There exists a dereference table that (i) succeeds on each allocation w.h.p., (ii) has load factor at least $1-\delta$, (iii) has constant-time updates w.h.p., and (iv) has tiny pointer size $O(P + \log\delta^{-1})$, where P is a random variable such that $\Pr[P \geq i] \leq 2^{-2^{\Omega(i)}}$ for all i. In particular, the tiny pointer size is $O(1+\log\delta^{-1})$ in expectation.*

We can assume without loss of generality that $\delta < \alpha$ for some sufficiently small positive constant α of our choice (if $\delta > \alpha$, we can reset $\delta = \alpha = \Theta(1)$ without changing the guarantee of the theorem).

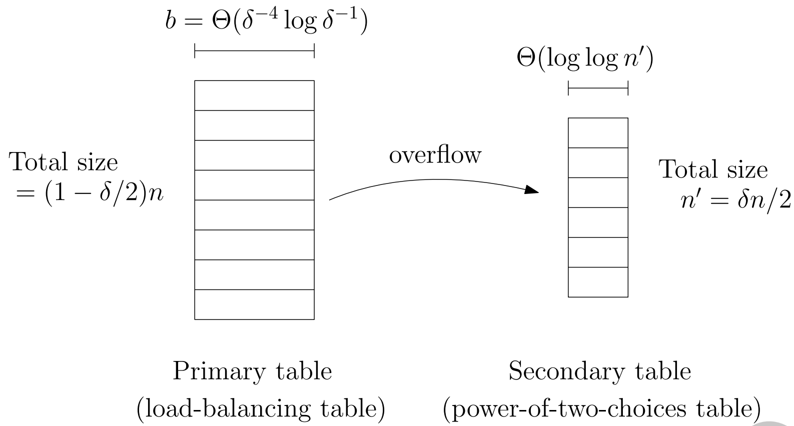

Fig. 1. A pictoral representation of the layouts of the primary and secondary tables. The primary table is implemented to support load factor $1 - \Theta(\delta^2)$, so that only $\delta^2 n$ allocations overflow to the secondary table at a time. The secondary table is implemented to have size $n' = \delta n/2$ and to support a (much sparser) load factor of $\Theta(1/\log \log n') = \omega(\delta)$, so that it can successfully store all of the overflowed allocations from the primary table.

Observe that, using the same primary/secondary-table construction as in the proof of Theorem 4.1, we can immediately reduce to the case where the load factor is a positive constant of our choice. Indeed, suppose that we could implement a dereference table T with load factor α for some positive constant $\alpha > 0$ and average tiny pointer size $O(1)$. Then we can use T as the secondary table in the construction: if the entire dereference table supports load factor $1 - \delta$, then the requirement for the secondary table is that it must be able to support $\delta^2 n$ items using $\delta n/2$ slots. So as long as $\delta < \alpha/2$ (which can be assumed without loss of generality), then T suffices.

Thus our task of proving Theorem 5.1 reduces to the task of proving the following proposition.

Proposition 5.2. *There exists a dereference table that (i) succeeds w.h.p. on each allocation, (ii) has load factor $\Omega(1)$, (iii) has constant-time updates w.h.p. in n, and (iv) has tiny pointer size P, where P is a random variable satisfying* $\Pr[P \geq i] \leq 2^{-2^{\Omega(i)}}$ *for all i.*

Let $\hat{n}$ be the maximum number of keys that can be allocated in the dereference table. We will construct a dereference table with $n = O(\hat{n})$ slots. Because $\hat{n}$ and n are only a constant factor away from one another, we may use $O(\hat{n})$ bits of metadata, and allow failures with probability $1/\text{poly}(\hat{n})$.

Constructing the dereference table. We now describe our construction for the dereference table that we use to prove Proposition 5.2. The dereference table hashes every allocated key into one of $\hat{n}/\log \hat{n}$ ***containers***, so that, at all times, each container has $\log \hat{n}$ items in expectation. We deterministically limit the number of items in each container to $s = c \log \hat{n}$, for some large enough constant $c > 1$ to be determined later. When a key is hashed into a container that already has $c \log \hat{n}$ items, the allocation fails.

Each container is managed independently, and its allocations/frees are performed using a scheme with $\log_2 s$ levels, as follows. For every $0 \leq i < \log_2 s$, the ith level is a load-balancing table with $s_i := s/2^i$ buckets, each with b slots, for some large enough constant $b \geq 2$ to be determined.

The basic idea is that, when an allocation in level i fails due to a bucket being full, we recursively attempt the allocation in the next level $i + 1$ (which uses a different hash function than does level i). Intuitively, as long as b is a sufficiently large constant, then each level should succeed on at least $1/2$ of its allocations, which is why we can afford to let the next level $i + 1$ have half the size of the previous one.

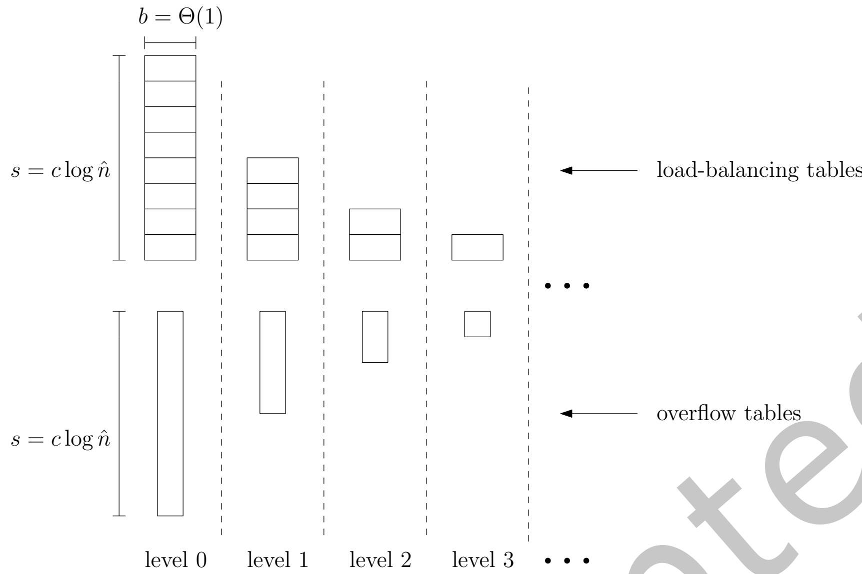

Fig. 2. A pictoral representation of the layout used to implement each container of size $\Theta(\log \hat{n})$. When an allocation fails in the i-th load-balancing table, it either proceeds to the $(i+1)$-th load-balancing table (if $L_{i+1} < s_{i+1}$) or it proceeds to the i-th overflow table (which is deterministically guaranteed to have a free slot).

The problem with this basic construction is that if even just a few consecutive levels behave badly, resulting in $\omega(s_i)$ items being sent to some level i, then there may not be room for those items in all of the levels $i, \ldots, \log_2 s$ combined. On the other hand, our construction must be able to handle such bad scenarios, because most of the levels are so small that we cannot offer high-probability guarantees on their behavior. Thus, we must modify the construction so that, when a level behaves badly, the effects of that bad behavior are isolated.

To do this, we add a fallback structure to each level, which we call ***overflow array***, to prevent excessive occupancy. The overflow array in each level i has s_i slots (the same number of slots as the load-balancing table at that level). Let L_i be the random variable denoting the number of values currently stored in levels i or larger, including their overflow arrays. Whenever an allocation at some level i fails (because the bucket is full), we recursively allocate in the next level only if $L_{i+1} < s_{i+1}$, otherwise we place the value in any available slot in the overflow array of level i. The result of this is that we deterministically guarantee $L_i \leq s_i$ for every level i (including level 0, for which this is trivial, since $s_0 = s$).

Importantly, no overflow array can ever run out of space: since $L_i \leq s_i$ (deterministically), the total number of items in the overflow array for level i is also a guaranteed to be a most s_i, which is precisely the capacity of the overflow array.

We are now ready to describe the full allocation algorithm. See Figure 2 for a picture of the layout used to implement each container.

Allocate(x):
1. Hash x into one of the $\hat{n}/\log \hat{n}$ containers.
2. If the selected container is already at full capacity s, fail.
3. Else, allocate x in the selected container:
   (a) For each $i = 0, 1, \ldots, \log_2(s) - 1$:

(i) Increment L_i.
(ii) Try to allocate x in the ith load-balancing table.
(iii) If the allocation succeeds:
- Let j be the chosen slot within the chosen bucket.
- Return `(level` i`, load-balancing table, bucket slot` j`)`.

(iv) If $L_{i+1} \geq s_{i+1}$:
- Pick any free slot in the i-th overflow array.
- Let j be the chosen slot in the array.
- Return `(level from the back` $\log_2(s) - 1 - i$`, overflow array, slot` j`)`.

Notice that if an allocation ends up using a slot j in some bucket in the i-th level's load-balancing table, then the tiny pointer encodes: the quantity i, which is $O(\log i)$ bits; the fact that the allocation used the load-balancing table rather than the overflow array, which is $O(1)$ bits; and the quantity j, which is $O(\log b) = O(1)$ bits. The total length of the tiny pointer is $O(\log i)$ in this case.[12]

On the other hand, if an allocation ends up using the j-th slot in the i-th level's overflow array, then the tiny pointer encodes: the quantity $\log_2(s) - 1 - i$, which is $O(\log(\log_2(s) - 1 - i))$ bits; the fact that the allocation used the overflow array rather than the load-balancing table, which is $O(1)$ bits; and the quantity j, which is $O(\log s_i)$ bits. Importantly, in this case, we elect to encode $\log_2(s) - 1 - i$, rather than the equivalent quantity i. This allows us to bound the total size of the tiny pointer by

$$O(\log(\log_2(s) - 1 - i)) + O(1) + O(\log s_i) = O(\log\log(s/2^i) + \log s_i) = O(\log\log s_i + \log s_i) = O(\log s_i).$$

Thus, when an allocation uses the overflow array in level i, we can bound the tiny-pointer size by $O(\log s_i)$.

Implementing operations in constant time. The information in the tiny pointers enables dereferences to easily be performed in time $O(1)$. Performing allocations and frees in time $O(1)$ is slightly more difficult, however.

Let us start by considering the naïve approach to implementing allocations and see why this is too slow. We must first identify which container to use (this just requires us to evaluate a hash function, taking constant time). We must then determine which level we will be using; if we end up using level i, then this takes time $\Theta(i)$, which is too slow when $i = \omega(1)$.

We solve this problem as follows. Let d to be some sufficiently large positive constant. We will implement levels $0, 1, \ldots, d-1$ using the naive approach, and then we will implement the levels $d, \ldots, \log_2 s$ using the Method of Four Russians (i.e., the "lookup-table approach"). Notice that, since d is at least a sufficiently large positive constant, the total number of slots in the levels $d, \ldots, \log_2 s$ is at most $4s_d/2^d \leq (\log \hat{n})/10$. Thus the entire state of which slots are free in those levels can be encoded in $(\log \hat{n})/10$ bits; we store this quantity as metadata for each container, totaling to $O(\hat{n})$ bits of metadata across all $\hat{n}/\log \hat{n}$ containers. Moreover, the hash values $h_1(x), h_2(x), \ldots, h_{\log_2 s}(x)$ that are used to select a bucket in each level together represent only $O((\log\log \hat{n})^2)$ bits (and can be implemented to just be the first $O((\log\log \hat{n})^2)$ bits of a single hash function). Thus, the entire state of levels $d, \ldots, \log_2 s$, plus all of the information about the hash values $h_1(x), h_2(x), \ldots, h_{\log_2 s}(x)$, can be encoded in an integer ϕ of $(\log \hat{n})/2$ bits that can be constructed in time $O(1)$. This means that we can pre-construct a lookup table of size $2^{(\log \hat{n})/2} = \sqrt{\hat{n}}$ that we can use to determine, for any given value of ϕ, which level the allocation should use. The lookup table takes a negligible amount of metadata space, allows for allocations to be performed in time $O(1)$, and can be constructed in time $\tilde{O}(\sqrt{\hat{n}})$ during the dereference table's creation.

[12]We follow the convention that $\log i = \Omega(1)$ for all i, so $\log 0$ and $\log 1$ are set to 1.

Now that we have specified how to implement allocations, frees are simple to implement, since they just update the metadata to reflect that the slot has been freed (this just flips a single bit in the metadata).

We have now fully specified the construction and implementation of our dereference table. It remains to analyze its properties, namely the probability of failure, the load factor, and the tiny-pointer sizes.

Probability of failure. The only way that an allocation can fail is if there is no room in the container that it hashes to, i.e., the container has $c \log \hat{n}$ items already. Otherwise, if the container has fewer than $c \log \hat{n}$ items, then the allocation is guaranteed to succeed (but, of course, it is not guaranteed to result in a small tiny pointer).

On average, $\log \hat{n}$ keys hash to any particular container, so by a Chernoff bound the maximum size across all containers is at most $c \log \hat{n}$ w.h.p. in $\hat{n}$ for some positive constant c. By the union bound, this holds for all of the $\hat{n}/\log \hat{n}$ containers simultaneously, w.h.p. in $\hat{n}$. Thus, if we pick $s = c \log \hat{n}$ for some large enough constant c, at any point in time, all containers will be below capacity w.h.p. in $\hat{n}$.

Load factor. Next, we verify that the total number of slots is $O(\hat{n})$. The dereference table for each container uses space $O(\sum_i s_i) = O(s_0) = O(s) = O(\log \hat{n})$ slots, and there are $\hat{n}/\log \hat{n}$ containers. Hence, the total space is $O(\hat{n})$, so the load factor is $\Omega(1)$, as desired.

Tiny pointer size. To conclude the proof of Proposition 5.2, we analyze the tiny pointer size of a given allocation, conditioned on the event that the allocation doesn't fail. The size of the tiny pointer depends on which level the key ends up allocated in. Specifically, as we have seen above:

- $O(\log i)$ if the key is allocated in the ith load-balancing table;
- $O(\log s_i)$ if the key is allocated in the ith overflow array.

Fix an arbitrary container to be the one where the allocation takes place, and consider the following events:

- $\mathcal{B}_i$: the key is allocated in the ith load-balancing table;
- $\mathcal{O}_i$: the key is allocated in the ith overflow array;
- $\mathcal{L}_i$: $L_i < s_i$.

We will condition on two events: (i) that the item picks the container we fixed, and (ii) that the container contains fewer than $c \log \hat{n}$ items (i.e., the allocation doesn't fail). We will drop the conditioning notation for clarity. Let P be the size of the output tiny pointer. Then, by the law of total expectation,

$$\mathbb{E}\left[P\right] \leq \sum_i \Pr\left[\mathcal{B}_i\right] \cdot O(\log i) + \sum_i \Pr\left[\mathcal{O}_i\right] \cdot O(\log s_i). \tag{1}$$

We bound each term separately. On the one hand,

$$\begin{aligned}\Pr\left[\mathcal{B}_i\right] &\leq \Pr\left[\overline{\mathcal{B}_0}, \mathcal{L}_1, \overline{\mathcal{B}_1}, \ldots, \mathcal{L}_{i-1}, \overline{\mathcal{B}_{i-1}}\right] \\ &\leq \Pr\left[\overline{\mathcal{B}_0}\right] \cdot \Pr\left[\overline{\mathcal{B}_1} \mid \overline{\mathcal{B}_0}, \mathcal{L}_1\right] \cdots \Pr\left[\overline{\mathcal{B}_{i-1}} \mid \overline{\mathcal{B}_0}, \mathcal{L}_1, \ldots, \overline{\mathcal{B}_{i-2}}, \mathcal{L}_{i-1}\right].\end{aligned} \tag{2}$$

In the last product, for every $j \in \{1, \ldots, i-1\}$, the load factor of the load-balancing table in level j is at most $1/b$, because there are $L_j < s_j$ items, s_j buckets, and each bucket has capacity b. This means that at most $1/b$ of the bins are full, deterministically, so the probability that a full bucket is chosen is Firat most $1/b$. Hence, every term in Equation (2) is bounded by $1/b$, and

$$\Pr\left[\mathcal{B}_i\right] \leq 1/b^i \leq 1/2^i.$$

On the other hand,

$$\Pr[\mathcal{O}_i] \leq \Pr[\overline{\mathcal{L}_{i+1}}].$$

We can bound the latter probability using Lemma 4.2. By construction, the load-balancing table in level i always has at most s_i allocations made to it (including the failed ones, since $L_i \leq s_i$ and L_i counts both the items in level i and the items in levels $i+1, i+2, \ldots$); moreover, the allocations and frees performed on the load-balancing table (which may differ from those performed on the overall dereference table) are independent of the randomness used in the load-balancing table. Assuming that the bucket size b is a sufficiently large constant, it follows that we can apply Lemma 4.2 (where the value of m being used in the lemma is s_i and the value of δ being used in the lemma is a small positive constant) to deduce that, with probability at least $1 - \exp(-\Omega(s_i))$, the number of failed allocations at level i at any given moment is less than $s_i/2 = s_{i+1}$. This, in turn, implies that $\mathcal{L}_{i+1}$ holds. Thus we can conclude that

$$\Pr[O_i] \leq 1/2^{\Omega(s_i)}.$$

Putting the pieces together,

$$\mathbb{E}[P] = \sum_i \frac{O(\log i)}{2^i} + \sum_i \frac{O(\log s_i)}{2^{\Omega(s_i)}} = O(1).$$

Notice that these calculations show that a tiny pointer of size $O(\log \ell)$ has probability $2^{-\Omega(\ell)}$, or, equivalently, a tiny pointer of size $O(\ell)$ has probability $2^{-2^{\Omega(\ell)}}$. This suggests that the tiny pointer size decays at a doubly-exponential rate. We prove this next. For any ℓ, in order for $P \geq \ell$ to occur, there must exist a positive constant γ such that at least one of $\log i$ or $\log s_i$ is at least $\gamma\ell$. It follows that

$$\begin{aligned}\Pr[P \geq \ell] &\leq \sum_{i \geq 2^{\gamma\ell}} \Pr[\mathcal{B}_i] + \sum_{s_i \geq 2^{\gamma\ell}} \Pr[O_i] \\ &= \sum_{i \geq 2^{\gamma\ell}} \frac{1}{2^i} + \sum_{s_i \geq 2^{\gamma\ell}} \frac{1}{2^{\Omega(s_i)}}.\end{aligned}$$

Both sums are dominated by their first terms, and are thus $1/2^{2^{\Omega(\ell)}}$. Therefore,

$$\Pr[P \geq \ell] \leq \frac{1}{2^{2^{\Omega(\ell)}}},$$

which completes the proof of Proposition 5.2. As discussed earlier, Proposition 5.2, in turn, implies Theorem 5.1.

Bounding sums of tiny-pointer sizes. In our applications of tiny pointers, a common way of using variable-size pointers will be to pack $\Theta\left(\frac{\log n}{\log \delta^{-1}}\right)$ of them into $\Theta(\log n)$ bits. Therefore, we conclude this section by proving a bound on the total number of bits consumed by a set S of $O(\log n / \log \delta^{-1})$ tiny pointers.

Proposition 5.3. *Using the construction in Theorem 5.1, for any set S of $O\left(\frac{\log n}{\log \delta^{-1}}\right)$ tiny pointers, the sum of their sizes will be $O(\log n)$ bits w.h.p.*

Proof. With high probability, all of the allocations for S succeed. This means that we can ignore the case where allocations fail, so when an allocation fails, we shall treat it as contributing a tiny pointer of size 0.

Let K be the set of keys corresponding to the tiny pointers in S. The easy case is if every key $x \in K$ hashes to a different container; in this case, we can analyze each container separately to conclude that each tiny pointer Allocate(x) independently has length $O(\log \delta^{-1} + P_x)$ bits, where $\Pr[P_x > \ell] \leq 2^{-2^{\Omega(\ell)}}$. Applying a Chernoff bound for sums of independent geometric random variables, we can conclude that $\sum_{x \in K} P_x \leq O(\log n)$ w.h.p., and thus that the total number of bits consumed by S is $O(\log n)$.

What if some of the keys $x \in K$ hash to the same container as other keys from K? Then we can no longer analyze the lengths of the resulting tiny pointers independently. Let X denote the set of such keys x. Since each tiny pointer is deterministically at most $O(\log n)$ bits, we can complete the proof by establishing that, with w.h.p., $|X| = O(1)$.

Let $x_1, x_2, \ldots$ denote the keys in K, and let X_i be the indicator random variable for the event that x_i hashes to the same container as one of $x_1, x_2, \ldots, x_{i-1}$. Then $|X| \leq 2\sum_i X_i$. On the other hand, each X_i satisfies $\mathbb{E}[X_i] \leq$

$(i-1)/n \le |S|/n \le O((\log n)/n)$. Thus $\sum_i X_i$ is a sum of independent indicator random variables with total mean $O((\log^2 n)/n)$. Applying a Chernoff bound, we will conclude that $\sum_i X_i = O(1)$ w.h.p., which completes the proof. Specifically, if we set $\mu = \mathbb{E}[X] = O(\log^2 n/n)$ and $\delta = \gamma\mu^{-1}$ for some large positive constant γ, then

$$\Pr[X = \omega(1)] \le \Pr[|X - \mathbb{E}[X] \ge (1+\delta)\mu] \le \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \le \frac{e^{\delta\mu}}{(1+\delta)^{(1+\delta)\mu}} \le \frac{O(1)}{(1+\delta)^{\gamma}} = 1/\text{poly}(n).$$

□

6 LOWER BOUNDS

In this section we prove that the bounds in Theorems 4.1 and 5.1 are tight. We begin by proving a lower bound for variable-size tiny pointers, since it is then used as part of the proof for the fixed size case.

What makes the lower bound for variably sized tiny pointer tricky is that any single tiny pointer might be very small. For example, the dereference table could have a single special slot that corresponds to the tiny pointer 0 (for every key), and then if the dereference table ever wanted to make a single tiny pointer small, it could allocate the special slot. Thus, our proof treats different types of slots differently: for each slot j, we define a potential function $\phi(j)$ indicating how "useful" that slot is to a random insertion. The idea is that insertions that use slots j with small potentials $\phi(j)$ must, on average, have relatively large tiny pointers; but insertions that use slots j with large potentials $\phi(j)$ must be rare, since only a relatively small fraction of the slots can have large potentials, and the number of insertions into them can be bounded by the number of deletions out of them.

Theorem 6.1. *Consider a universe $\mathcal{U}$ of keys, where $\mathcal{U}$ is assumed to have a sufficiently large polynomial size. If a dereference table supports variable-size tiny pointers of expected size s and load factor $1-\delta = \Omega(1)$, then $s = \Omega(\log \delta^{-1})$.*

Proof. Let $\mathcal{U}$ be a universe of size n^c where c is a sufficiently large constant. Let $\delta < 1/4$. Let T be a dereference table with n slots and load factor $1-\delta$ (i.e., it is capable of allocating up to $(1-\delta)n$ slots to keys from $\mathcal{U}$ at a time). Moreover, suppose that T guarantees an expected tiny-pointer length of at most μ. Then we wish to show that

$$\mu \ge \Omega(\log \delta^{-1}).$$

To simplify our discussion, we shall think of a key $x \in \mathcal{U}$ as residing in the location that is allocated to it. Thus allocations correspond to insertions, and frees correspond to deletions.

Consider a workload in which the table is initialized to contain $(1-\delta)n$ arbitrary items, and then we alternate between insertions and deletions for $n^{c/2}$ steps. Each insertion selects a random item of $\mathcal{U}$ (with high probability in n, we never insert an item that is already present), and each deletion selects a random item out of those present.

We treat tiny pointers as taking values in $\mathbb{N}$. If the tiny pointer takes value i, then it uses $\Omega(\log i)$ bits. For each item $x \in \mathcal{U}$, let $h_i(x)$ denote the position where x would reside in T if x had a tiny pointer with value i. Set $\ell = \delta^{-1}/32$. For each position $j \in [n]$ in the table, define the ***potential*** $\phi(j)$ to be

$$\phi(j) = \frac{|\{(x,i) \mid x \in \mathcal{U}, i \in [\ell], h_i(u) = j\}|}{|\mathcal{U}|}.$$

Call an insertion ***safe*** if the item x that is inserted is inserted into one of positions $h_1(x), \ldots, h_\ell(x)$. Call an insertion ***resource efficient*** if the item x that is inserted is inserted into a position j satisfying $\phi(j) \le \frac{4\ell}{n}$.

We begin by bounding the probability that a given insertion is both safe and resource efficient. Consider some point in the operation sequence, with some set X of keys present in the table. Now consider the insertion of some key $x \notin X$. The probability that the insertion is both safe and resource efficient is at most

$$\begin{aligned}
&\sum_{\substack{\text{empty position } j\in[n]\\ \phi(j)\le \frac{4\ell}{n}}} \sum_{i=1}^{\ell} \Pr_{x\in\mathcal{U}}[h_i(x)=j] \\
&= \sum_{\substack{\text{empty position } j\in[n]\\ \phi(j)\le \frac{4\ell}{n}}} \sum_{i=1}^{\ell} \frac{|\{x | x\in\mathcal{U}\setminus X \wedge h_i(x)=j\}|}{|\mathcal{U}\setminus X|} \\
&\le 2 \sum_{\substack{\text{empty position } j\in[n]\\ \phi(j)\le \frac{4\ell}{n}}} \sum_{i=1}^{\ell} \frac{|\{x | x\in\mathcal{U} \wedge h_i(x)=j\}|}{|\mathcal{U}|} \\
&= 2 \sum_{\substack{\text{empty position } j\in[n]\\ \phi(j)\le \frac{4\ell}{n}}} \phi(j) \\
&\le 2 \sum_{\text{empty position } j\in[n]} \frac{4\ell}{n} \\
&= 2\delta n \frac{4\ell}{n} \\
&= \frac{1}{4}.
\end{aligned}$$

It follows that the expected number of insertions that are safe and resource efficient is at most $n^{c/2}/4$.

Next we bound the expected number of insertions A that are safe but not resource efficient. Rather than bound A directly, we instead examine the number of *deletions* B where the deleted item is deleted from a position j satisfying $\phi(j) > \frac{4\ell}{n}$. With the exception of the up to n insertions that have not yet been deleted, every insertion counted by A is counted by B, so

$$A \le B + n.$$

By the definition of $\phi(j)$, we have that $\sum_{j=1}^{n} \phi(j) = \ell$. It follows that $|\{j \in [n] \mid \phi(j) > \frac{4\ell}{n}\}| \le n/4$. Each random deletion therefore has probability at most $\frac{n/4}{(1-\delta)n} \le 1/2$ of removing an item in a position j satisfying $\phi(j) > \frac{4\ell}{n}$. Thus $\mathbb{E}[B] \le n^{c/2}/2$ which means that

$$\mathbb{E}[A] \le n^{c/2}/2 + n \le (1+o(1))n^{c/2}/2.$$

The expected number of safe insertions is therefore at most $(1+o(1))\frac{3}{4}n^{c/2}$, which implies that the expected number of unsafe insertions is at least $\Omega(n^{c/2})$. Each unsafe insertion results in a tiny pointer of length at least $\Omega(\log \ell) = \Omega(\log \delta^{-1})$ bits. Since a constant fraction of the insertions are expected to result in a tiny pointer of length at least $\Omega(\log \delta^{-1})$, we must have $\mu \ge \Omega(\log \delta^{-1})$. □

Next we prove a lower bound for fixed-sized tiny pointers, which shows that the bound in Theorem 4.1 is tight.

Theorem 6.2. *Consider a universe $\mathcal{U}$ of keys, where $\mathcal{U}$ is assumed to have a sufficiently large polynomial size. If a dereference table supports fixed-sized tiny pointers of size s and load factor $1 - \delta = \Omega(1)$, then $s = \Omega(\log\log\log n + \log \delta^{-1})$.*

It suffices to prove that $s = \Omega(\log\log\log n)$, since we have already shown that $s = \Omega(\log \delta^{-1})$.

The proof re-purposes a classic balls-and-bins lower bound. Say that a ball-placement rule is ***sequential*** if balls are placed sequentially, without knowledge of future ball arrivals, and if balls are never moved after being placed.

Theorem 6.3 (Theorem 2 in [59]). *Suppose that m balls are placed sequentially into m bins using an arbitrary sequential ball placement rule, where each ball chooses between d bins that are selected independently at random according to an arbitrary probability distribution on $[m]^d$. Then the number of balls in the fullest bin is* $(\log\log m)/d + O(1)$ *w.h.p.*

We now prove Theorem 6.2.

Proof of Theorem 6.2. Assume for contradiction that there exists a dereference table with load factor $1-\delta = \Omega(1)$ that supports fixed-size tiny pointers of size $s = o(\log\log\log n)$ bits. Let n be the number of slots in the dereference table, and let $m = (1-\delta)n$ be the maximum number of allocations that the dereference table can support at a time; assume without loss of generality that $1/(1-\delta) \in \mathbb{N}$, so n is a multiple of m. Finally, let $S = 2^s$, and observe that, by assumption, $S = o(\log\log n)$—and since $m = \Theta(n)$, $S = o(\log\log m)$.

Recall that $\mathcal{U}$ is the universe from which the keys are taken. For each key $x \in \mathcal{U}$, define the sequence $h_1(x), h_2(x), \ldots, h_S(x) \in [m]$ so that $h_i(x) = \lfloor \frac{m}{n}\textsc{Dereference}(x, i)\rfloor$. Note that, by the definition of the $\textsc{Dereference}$ function, the sequence $h_1(x), h_2(x), \ldots, h_S(x)$ is a function of only x, i, n, and the random bits of the dereference table—therefore, the sequence is predetermined by the coin flips, and is independent of the sequence of allocations/deallocations that are performed. Let $R \in [m]^S$ be a random variable obtained by selecting $x \in \mathcal{U}$ at random and setting $R = \langle h_1(x), h_2(x), \ldots, h_S(x)\rangle$; and let $\mathcal{R}$ be the probability distribution for R.

We will now construct a sequential ball-placement rule for mapping m balls to m bins. Our rule independently assigns each ball a random bin sequence $\langle h_1, h_2, \ldots, h_S\rangle \sim \mathcal{R}$ of S bins. Equivalently, we can think of the m balls as being m keys $x_1, x_2, \ldots, x_m$, where each x_i is selected uniformly and independently at random from $\mathcal{U}$, and each x_i has a bin sequence of $\langle h_1(x), h_2(x), \ldots, h_S(x)\rangle \in [m]^S$.

Since $|\mathcal{U}|$ is at least a sufficiently large polynomial in n, we have that with high probability in n, the x_i's are distinct. Our ball placement rule runs the dereference table on the side and uses the tiny pointers that it produces to decide where to place balls. To place ball x_i into a bin, we compute $p_i = \textsc{Allocate}(x_i)$, and we place x_i into the p_i-th bin in x_i's bin sequence, which is given by bin

$$h_{p_i}(x_i) = \left\lfloor \frac{m}{n}\textsc{Dereference}(x_i, p_i)\right\rfloor \in [m].$$

In summary, we have constructed a sequential ball placement rule that places m balls sequentially into m bins and that chooses a set of $d = S$ bins for each ball according to a probability distribution $\mathcal{R}$ over $[m]^d$. By Theorem 6.3, we can deduce that the fullest bin contains at least

$$\Omega\left((\log\log m)/d\right) = \Omega\left((\log\log m)/S\right) = \omega(1)$$

balls with high probability in m.

On the other hand, the dereference table guarantees that $\textsc{Dereference}(x_i, p_i) \in [n]$ is unique for each $i \in [m]$. The number of balls x_i satisfying

$$\left\lfloor \frac{m}{n}\textsc{Dereference}(x_i, p_i)\right\rfloor = j$$

for a given j is therefore at most $\frac{n}{m} = O(1)$. This means that the number of balls in any given bin is also $O(1)$. Since the dereference table succeeds with high probability in n, we can deduce that there are $O(1)$ balls in the fullest bin with high probability in n. This contradicts the fact that the number of balls in the fullest bin is $\omega(1)$, thereby completing the proof by contradiction. □

7 APPLYING TINY POINTERS TO FIVE PROBLEMS IN DATA STRUCTURES

In this section we present several applications of tiny pointers to classical problems in data structures:

- Relaxed Retrieval: we show that a slight modification to the classic retrieval problem eliminates the classical lower bound of $\Omega(\log \log n)$ wasted bits per item (Section 7.2).
- Succinct binary search trees: we give an approach for transforming arbitrary dynamic binary search trees into succinct data structures (Section 7.3).
- Space-efficient stable dictionaries: we transform any fixed-capacity key-value dictionary into a key-value stable dictionary (Section 7.4).
- Space-efficient dictionaries: we transform any dictionary with fixed-size values into one which can space-efficiently store variably sized values (Section 7.5).
- An optimal internal-memory stash: we construct a constant-time stash that space-efficiently stores the locations of items residing in a large external-memory data structure (Section 7.6).

7.1 Some General-Purpose Techniques for Using Tiny Pointers

Before diving into specific applications, we discuss several preliminary definitions and techniques that will be useful in several of the applications.

Key-value dictionaries. Several of our applications will perform black-box transformations in order to add new features (namely, stability and variable-size values) to key-value dictionaries. Formally, a ***key-value dictionary*** (often just called a ***dictionary***) is any data structure that stores key-value pairs (e.g., a hash table or a tree), where each key appears at most once. Typically, a key-value dictionary supports insertions, deletions, and queries, where queries, in particular, return the value associated to some key. Depending on the data structure, additional operations may also be supported, for example successor queries, which return the successor to some key.

We say that a key-value dictionary uses a ***value array*** if it designates some contiguous chunk of memory (that can be extended or shrunk over time) whose purpose is to store the values corresponding to keys. When performing a query on a key, the dictionary uses the key to determine where in the value array the corresponding value is currently stored. Thus, even dictionary implementations that don't seem to use value arrays — e.g. red-black trees — can be directly modified to use them. If values are k bits long, then the value array can be viewed as a array of k-bit objects.

In our applications, we will restrict ourselves to dictionaries that store their values in value arrays. For simplicity, we will assume that the dictionary uses a single value array, although all of our results can also easily be applied to a dictionary that makes use of many separately-allocated value arrays (as long as each individual value array is at least $\Omega(\log n)$ bits). The reason that we assume a single value array is because, to the best of our knowledge, all of the known space-efficient key-value dictionaries can easily be implemented in this format, so we choose to avoid introducing unnecessary complication to the results.

How to store value arrays of tiny pointers. A theme in several of our applications will be to modify a value array so that, rather than storing values directly, we instead store tiny pointers of some size k. Recall, however, that tiny pointers of size $k = o(\log \log \log n)$ bits are not fixed-size, meaning that some tiny pointers may require more than k bits. Nonetheless, if we are willing to use a value-array that is a constant-factor larger, then there is a simple trick, which we call ***chunked pointer storage***, that lets us interact with these variable-length tiny pointers in the same way that we would interact with fixed-length tiny pointers.

Break the value array into contiguous chunks of $O((\log n)/k)$ tiny pointers. By Proposition 5.3, the total number of bits used by the tiny pointers in each chunk is $O(\log n)$ with high probability in n. Thus each chunk can be stored in $O(\log n)$ bits, meaning that the entire value array can be stored in $O(nk)$ bits.

There is, however, the remaining issue of how to efficiently access and modify the j-th tiny pointer in a given chunk. For each chunk, we can store an additional $O(\log n)$-bit bitmap where the bits that are set to 1 indicate the positions in the chunk where tiny pointers begin and end. To efficiently find the j-th tiny pointer, it suffices to find the j-th and $j+1$-th 1s in the bitmap. (The tiny pointer can then be extracted, modified, and reinserted, in constant time using standard bit manipulation on the bitmap and the chunk.) The problem of finding the j-th 1 in a $O(\log n)$-bit bitmap is easily solved with the method of four Russians [7]: simply store an auxiliary lookup table of size $\sqrt{n}$ that allows for such queries to be answered in a $(\log n)/2$-bit bitmap in a single lookup, and then perform $O(1)$ lookups to perform such a query in an $O(\log n)$-bit bitmap.

How to dynamically resize a data structure using tiny pointers. Several of our applications will also encounter the problem of using tiny pointers in a data structure whose size dynamically changes over time. Of course, this means that we must also dynamically resize dereference tables. Our applications will take the following approach, which we call ***zone-aggregated resizing***.

Consider a value array storing tiny pointers to k-bit items in a dereference table (and assume k bits fit in $O(1)$ machine words). Suppose that we wish to maintain the dereference table at a load factor of $1-\Theta(1/k)$, that way the number of bits wasted per item stored is $O(1)$; note that this means that the tiny pointers in the value array are $\Theta(\log k)$ bits on average. Further suppose, however, that the value array dynamically changes size over time (meaning that items must be added and removed from the dereference table). For our discussion here, we will assume that the value array itself is dynamically resized to always be at a load factor of at least $\Omega(1)$.

How can we update the dereference table to maintain a load factor of $1-\Theta(1/k)$ while the number of items changes over time? Rather than just using a single dereference table, we use k dereference tables, and add $\Theta(\log k)$ bits to each tiny pointer in order to indicate which dereference table is being pointed into (this doesn't change the asymptotic size of the tiny pointers). We can grow and shrink the capacity (i.e., number of slots) of the dereference tables by either (a) rebuilding the smallest dereference table to double its size, or (b) rebuilding the largest dereference table to halve its size. If we assume for the moment that rebuilding a dereference table takes time proportional to the table's size, then the rebuilds can be de-amortized to take time $O(1)$ per operation (i.e., per modification to the dereference tables), while maintaining the desired load factor of $1-\Theta(1/k)$.

The problem with rebuilding a dereference table is that all of the tiny pointers into that dereference table become invalidated. The actual construction of the new dereference table can easily be performed in linear time, but how do we update the tiny pointers in the value array? If the value array has size n, then the dereference table being rebuilt consists of only $\Theta(n/k)$ items. We want to identify where the tiny pointers to those items are in the value array in time $\Theta(n/k)$ rather than time $\Theta(n)$.

The solution to this issue is very simple: break the value array into contiguous ***zones*** each of which consists of k values. Within each zone, maintain k linked lists, where the i-th linked list contains the tiny pointers that point into the i-th dereference table. Importantly, because these linked lists are within a zone of size k, the pointers *within* each linked list only require $\Theta(\log k)$ bits each; thus the linked lists do not asymptotically increase the size of the value array. On the other hand, in order to find all of the tiny pointers for a given dereference table, one can simply look at one linked list in each of the $\Theta(n/k)$ zones, allowing for all $\Theta(n/k)$ of the tiny pointers to be identified in time $\Theta(n/k)$.

For reasons that we shall see later, one of our applications will also require us to use larger zones of size $\text{poly}(k)$ rather than just of size k. For now, we simply remark that using larger zones of size $\text{poly}(k)$ still allows for the linked-list overhead of each tiny pointer to be bounded by $\Theta(\log k)$ bits, and that the time needed to identify the tiny pointers to a dereference table of size j is only

$$O(j + n/\text{poly}(k)), \tag{3}$$

since the number of linked lists that must be examined is only $O(n/\text{poly}(k))$.

7.2 Overcoming the $\Omega(\log\log n)$-Bit Lower Bound for Data Retrieval

Our first application revisits the classic retrieval problem [3, 26, 27, 29], in which a data structure must store a v-bit value for each of the k-bit keys in some set S, and must answer queries that retrieve the value associated with a given key. Here, we address the dynamic version of the problem, where the data structure must support the functions $\textsc{Insert}(x, y)$ (which inserts a new $x \in [2^k]$ into S and associates it with value $y \in [2^v]$), $\textsc{Delete}(x)$ (which removes some $x \in S$ from S), and $\textsc{Query}(x)$ (which returns the value y corresponding to x for some $x \in S$, or returns an arbitrary value of $x \notin S$), allowing for the set S to grow up to some maximum size n. Note that in the retrieval problem it is the *user's responsibility* to ensure that every invocation of $\textsc{Insert}$ is on a key $x \notin S$ and every invocation of $\textsc{Delete}$ is on a key $x \in S$.

It is known that, if $k = (1 + \Omega(1)) \log n$ bits, then any solution to the dynamic retrieval problem must use at least $nv + \Omega(n \log\log n)$ bits of space [3], regardless of the time complexity, and even if $v = 1$. It is further known that, if $k = \Theta(\log n)$ and $v = O(\log n)$, then the $nv + \Theta(n \log\log n)$ space bound can be accomplished by a randomized constant-time data structure [26].

We will now show that, by slightly relaxing the retrieval problem, we can use tiny pointers to obtain significantly better space bounds. In the ***relaxed retrieval problem***, the insertion/deletion/query operations are modified to work as follows. The operation $\textsc{Insert}(x, y)$ now returns a ***tiny retriever*** r which the user must remember. In the future, if the user wishes to query x (and they have not yet deleted x), they call $\textsc{Query}(x, r)$ to obtain the value y. Finally, if the user ever wishes to remove x from the set S, then they call $\textsc{Delete}(x, r)$.

The role of the tiny retriever is similar to that of a tiny pointer—it acts as a hint to assist the data structure. Unlike for tiny pointers, however, the pair (x, r) does not have to fully encode the position of y; instead, query operations $\textsc{Query}(x, r)$ can use auxiliary metadata, beyond just x and r, to determine the value y. We shall now see that this distinction is very important, allowing for us to do better than *both* the lower bound for the retrieval problem [3] and our lower bound for the tiny-pointer problem (Theorem 6.1). At the same time (almost paradoxically), it is our construction for variable-size tiny pointers that allows for us to get around both of these lower bounds. In the following, let $\log^{(r)} n = \log\log\cdots\log n$ denote the r-th iterated logarithm of n.

Theorem 7.1. *Consider the relaxed retrieval problem with k-bit keys, v-bit values, and a maximum capacity of n key/value pairs. Let $r \in [\log^* n]$ be a parameter. There is a solution to the relaxed retrieval problem that uses tiny retrievers of expected size $O(1)$, and that with high probability in n: takes constant time per query, takes $O(r)$ time per insertion/deletion, and uses total space $nv + O(n \log^{(r)} n)$ bits.*

Furthermore, if $\log^{(r)} n = \omega(1)$ and $v \leq \frac{\log n}{\log^{(r)} n}$, then the space consumption becomes $nv + O(n)$ bits.

The above theorem comes with an interesting tradeoff curve: constant-time insertions/deletions can achieve a space consumption of, for example, $nv + O(n \log\log\log\log\log n)$ bits, and $O(\log^* n)$-time insertion/deletions can achieve space consumption $nv + O(n)$ bits. Moreover, if v is slightly sub-logarithmic, then one can even achieve constant-time insertions/deletions with only $nv + O(n)$ bits of space.

We remark that the tiny retrievers in Theorem 7.1 are, in fact, variable-size tiny pointers as constructed in Theorem 5.1. They therefore satisfy the doubly-exponential tail inequality given by Theorem 5.1, as well as the concentration inequality given by Proposition 5.3.

Proof. We shall make use of Theorem 5.1 to construct a dereference table T with $2n$ slots. What makes our application of Theorem 5.1 unusual, however, is that we will not store anything in the store (if fact, we need not even allocate space for it). Instead, we will take advantage of the fact that $\textsc{Dereference}(x, p)$ is a $(1 + \log n)$-bit number that has been uniquely allocated to x.

To implement the operation $\textsc{Insert}(x, y)$, we call $\textsc{Allocate}(x)$ to obtain a tiny pointer p of expected size $O(1)$ (note that p will also be our tiny retriever). Define $s_x = \textsc{Dereference}(x, p)$ to be the slot number in $[2n]$ allocated to x. The main property that we will exploit is that $s_x \neq s_{x'}$ for all other $x' \in S$. To complete the $\textsc{Insert}$ operation, we insert the key/value pair (s_x, y) into a succinct hash table H (whose specifications we will describe later). Queries and deletes are then implemented as follows: Query(x, p) returns $H[$Dereference$(x, p)]$; and Delete(x, p) deletes key Dereference(x, p) from H and calls Free(x, p) on the dereference table T.

The correctness of the data structure follows from the fact that, for each $x \in S$ with tiny retriever p, Dereference(x, p) is unique. The dereference table uses space only $O(n)$ bits and supports constant-time operations (with high probability). Thus, to prove the theorem, it remains to analyze the hash table H.

We construct H using the most space-efficient known construction for a hash table [12]. If H is storing up to n keys from a universe U and values are v bits, then it supports the following guarantees with high probability: queries are constant-time, insertions/deletions take time $O(r)$, and the total space consumption is

$$\log \binom{|U|}{n} + nv + O(n \log^{(r)} n)$$

bits. If, in addition, $\log^{(r)} n = \omega(1)$ and $v \leq \frac{\log n}{\log^{(r)} n}$, then the space becomes $\log \binom{|U|}{n} + nv + O(n)$ bits.

Our use of tiny pointers ensures that the keys in H are from the very small universe $U = [2n]$. So

$$\log \binom{|U|}{n} = \log \binom{2n}{n} = O(n)$$

by Stirling's formula. This completes the proof of the theorem. □

A remark on resizing. In Subsection 7.3, we shall see an application of tiny retrievers to the problem of constructing succinct binary search trees. In this application, we will want to have two relaxed-retrieval data structures whose sizes sum to at most n. Here, we can take advantage of the fact that the hash table H used above actually offers a dynamically-resizing guarantee: if, at any given moment, the hash table has size m, then it uses space at most

$$\sqrt{n} + \log \binom{2n}{m} + mv + O(m \log^{(r)} n),$$

with high probability in n. The full retrieval data structure (consisting of the hash table H and the dereference table T) therefore uses space at most

$$\log \binom{2n}{m} + mv + O(n + m \log^{(r)} n).$$

By Stirling's formula, this is at most

$$m \log n - m \log m + mv + O(n + m \log^{(r)} n).$$

Thus, if we have two relaxed-retrieval data structures, one of size $m_1 \leq n$ and one of size $m_2 \leq n$, and $m = m_1 + m_2 = \Theta(n)$, then their total space consumption will be at most

$$\begin{aligned} &(m_1 + m_2) \log n - m_1 \log m_1 - m_2 \log m_2 + (m_1 + m_2)v + O((m_1 + m_2) \log^{(r)} n) \\ = &\, m \log n - m_1 \log m_1 - m_2 \log m_2 + mv + O(m \log^{(r)} n). \end{aligned}$$

By Jensen's inequality, $m_1 \log m_1 + m_2 \log m_2 \geq (m_1 + m_2) \log \frac{m_1+m_2}{2} = m \log \frac{m}{2} = m \log n - O(n)$. Thus the total space is at most

$$\begin{aligned} & m \log n - (m \log n - O(n)) + mv + O(m \log^{(r)} n) \\ & = mv + O(m \log^{(r)} n) \\ & = mv + O(m \log^{(r)} m) \end{aligned}$$

This, of course, is the same bound that we get for a single relaxed-retrieval data structure of size m.

The reason that this matters is that it allows for a simple way to perform dynamic resizing: every time that the size m of a data structure changes by a factor of two, we move all of the items in the current relaxed-retrieval data structure D_1 into a new relaxed-retrieval data structure D_2 (parameterized as having capacity $n = \Theta(m)$ based on the new value of m). As we move items from D_1 to D_2, the total space consumption of D_1 and D_2 will continue to be $mv+O(m \log^{(r)} m)$ bits. Note that, to move an item from D_1 to D_2, we will need to generate a new tiny retriever for that item (since we are deleting the item from D_1 and inserting it into D_2). In our binary-search-tree application, this will be easy to do by simply running through all of the items and relocating them one by one. Furthermore, since the work of constructing D_2 can be spread across $\Theta(n)$ operations, it can be achieved at a cost of $O(r)$ per insertion/deletion.

7.3 Succinct Binary Search Trees

Our second application is a black-box approach for transforming dynamic binary search trees into succinct data structures. If there are n items in the succinct search tree, each of which is k bits long, then the size of the succinct search tree will be at most $nk + O(n + n \log^{(r)} n)$ bits, where $r > 0$ is an arbitrary parameter. Path traversals in the tree incur only a constant-factor overhead, and modifications to the tree incur only an $O(r)$-factor overhead.

An advantage of our approach is that it can be applied to rotation-based search trees. This includes, for example, red-black trees [38], splay trees [58], etc. If the dynamic-optimality conjecture [58] is true, meaning that the splay tree is dynamically optimal, then our succinct splay tree is also dynamically optimal when $r = O(1)$.

Theorem 7.2. *Consider any binary search tree storing a-bit keys and b-bit values, where every node is associated with a distinct key, and where each node has pointers to its children. For any $r > 0$, the tree can be implemented to offer the following guarantees with high probability in the tree size n: the tree takes space $na+nb+O(n+n \log^{(r)} n)$ bits, traversals from parents to children take time $O(1)$, and modifications to the tree (i.e., adding or removing an edge) take time $O(r)$.*

We remark that, information theoretically, the tree uses at least $n(a + b)$ bits of space. And since the keys are distinct, $na = \Omega(n \log n)$. Thus, for any $r > 1$, the search tree above is succinct.

Proof. To avoid ambiguity between different types of 'keys' and 'values' in our discussion, we will sometimes refer to the a-bit keys and b-bit values stored by the user as *user keys* and *user values.*

We will make use of our solution to the relaxed retrieval problem (Theorem 7.1). The retrieval keys/values will be different from the user keys/values. Each retrieval key x will correspond to a user key with an additional bit appended to it (more on this later), and each retrieval value y will store an $(a + b)$1-bit user key/value pair (for that node in the tree), along with two tiny retrievers r_1 and r_2 (which can be used to retrieve the children of that node). Since r_1 and r_2 are themselves variable-length tiny pointers of expected size $O(1)$, this means that the retrieval value is also variable-length. On the other hand, the relaxed-retrieval data structure is designed for *fixed-length* values. Fortunately, we can store the tiny retrievers r_1 and r_2 with the following method. Recall that, in our construction for the relaxed retrieval problem, we create a dereference table with $2n$ slots, but we do not actually store anything in the dereference table's store. We now change this so that the store is a value array with $2n$ slots that stores the tiny retrievers r_1 and r_2 for each item in the dereference table (so, if p is the tiny pointer for x, then r_1, r_2 are in the

Dereference(x, p)-th position of the value array). Using the chunked-pointer-storage technique, we can ensure that the total size of the value array is $O(n)$ bits, even though the pointers that it stores are variable length.

We now describe our encoding of the binary search tree: Each node in the search tree stores a user key-value pair (u, v) corresponding to that node, along with two tiny retrievers r_1 and r_2. The tiny retriever r_1 is for the left child and uses $x \circ 0$ as its retrieval key (so Query$(x \circ 0, r_1)$ returns the left child of x), and the tiny retriever r_2 is for the right child and uses $x \circ 1$ as its retrieval key (so Query$(x \circ 1, r_1)$ returns the right child of x).[13] Note that, if the left child (resp. right child) does not exist, then we simply set r_1 (resp. r_2) to null.

Let us begin by assuming that our binary search tree has a fixed capacity of n user keys/values, so we can use a relaxed-retrieval data structure with capacity n. Then our relaxed-retrieval data structure uses $na + nb + O(n + n \log^{(r)} n)$ bits. Navigating from a node to its child takes time $O(1)$ (since it requires a single query to the relaxed-retrieval data structure) and adding/removing an edge (x, z) from a node x to a child z takes time $O(r)$, with high probability, since it requires only a single insert/delete to the relaxed-retrieval data structure; importantly, if z is the root of some subtree, the act of setting z to be x's child *does not* require any nodes besides z to inserted/deleted in the relaxed-retrieval data structure.

Finally, let us modify our data structure so that it dynamically resizes as a function of the current number n of user key/value pairs. For this, we can simply use the resizing approach outlined in Section 7.2. Every time that n changes by a constant factor, we rebuild the relaxed-retrieval data structure to have capacity $\Theta(n)$ for the new value of n. (Note that this does not require us to rebuild the tree; it just requires us to update the tiny retrievers used in each node.) For each tiny retriever in the binary search tree, we can store an extra bit indicating which of the two relaxed-retrieval data structures it uses—this preserves correctness. As observed in Section 7.2 the act of moving items from the old relaxed-retrieval data structure to the new one does not violate our desired space guarantee: the total number of bits used by our search tree remains $na + nb + O(n + n \log^{(r)} n)$ at all times. And, by spreading the work of rebuilding the relaxed-retrieval data structure across $\Theta(n)$ operations, we maintain the property that each edge insertion/deletion takes time $O(r)$. Thus the theorem is proven. □

7.4 Space-Efficient Stable Dictionaries

Using tiny pointers, we give a black-box approach for transforming any fixed-capacity key-value dictionary into a ***stable*** dictionary, meaning that the position in which a value is stored never changes after the value is inserted. If the original dictionary stored v-bit values, then the new dictionary also stores v-bit values, and uses at most $O(\log v)$ more extra bits of space per value than the original data structure.

Theorem 7.3. *Consider a fixed-capacity key-value dictionary data structure T that stores its values in a value array of some size m. Let v denote the size of each value in bits.*

It is possible to construct a new data structure T' with the same operations and asymptotics (with high probability) as T, but with the additional property that T' is stable. Moreover, the total space consumed by T' is guaranteed (with high probability in m) to be at most $O(m \log v)$ more bits than T.

Proof. To construct T', we simply replace the value array for T with an array of m tiny pointers, each of size $\Theta(\log v)$ bits. (If $\log v < \log \log \log n$, then the chunked-pointer-storage technique can be used to handle the situation where different tiny pointers have different sizes.) The tiny pointers point into a dereference table of size $(1 + 1/v)m$ that stores the m v-bit values. (So the load factor is $1 - \Theta(1/v)$.) If a tiny pointer points at the value y corresponding to a key x, then the tiny pointer uses x as its key. This ensures stability, since even if the location in which the tiny pointer is stored changes, the tiny pointer does not have to change (and the value y does not have to move).

[13] An important distinction here is that we are using the *user key x* to construct the keys for the two tiny retrievers, rather than using the *position in memory* where x is stored—this means that, when we move an item x around in memory, we do not invalidate the tiny retrievers of nodes in the subtree rooted at x.

The array of tiny pointers consumes $O(m \log v)$ space. Whereas the value array in T consumes mv bits, the dereference table in T' consumes $(1 + 1/v)mv$ bits, which is only $O(m)$ more bits than used in T. Thus the claim on space efficiency is proven. Since tiny pointers only add constant time per access/modification of the value, the asymptotics are (with high probability in m) the same for both T and T'. □

By applying our result to the data structure from [13], which is a non-stable hash table with redundancy $O(n \log^{(O(1))} n)$ bits, we obtain the following corollary for hash tables.

Corollary 7.4. *Let r be a large positive constant, let n, v be parameters, let U be a universe of keys, and suppose that the machine word size is at least $\max(\log |U|, v)$. It is possible to construct a stable hash table that stores up to n key-value pairs, where the keys are from U and the values are v bits, and that uses space $\log \binom{|U|}{n} + O(n \log v) + O(n \log^{(r)} n)$ bits.*

7.5 Space-Efficient Dictionaries with Variable-Size Values

Our fourth application is a black-box approach for transforming any key-value dictionary (designed to store fixed-size values) into a dictionary that can store different-sized values for different keys. The resulting data structure offers the following remarkable guarantee on space efficiency. Let r be a positive constant of our choice, and let m be the number of entries in the value array used by the original dictionary (at some given moment). The new dictionary, which allows for values to be arbitrary lengths, replaces the value array for T with a data structure that consumes at most

$$O(m \log^{(r)} m) + \sum_{i=1}^{m} (v_i + O(\log v_i))$$

bits, where $v_1, v_2, \ldots, v_m$ denote the lengths in bits of the values being stored.

Theorem 7.5. *Consider a key-value dictionary data structure T that stores its values in a value array, and that is designed to store fixed-length keys. Let r be a positive constant of our choice. It is possible to construct a new data structure T' with the same operations and asymptotics (with high probability) as T, but with the additional property that T' can store values of arbitrary lengths (up to $O(1)$ machine words).*

At any given moment, if T were using a value array of size m bits, and the machine word size w satisfies $w \le m^{o(1)}$, then the total space consumed by T' to implement the value array is guaranteed (with high probability in m) to be at most

$$O(m \log^{(r)} m) + \sum_{i=1}^{m} (v_i + O(\log v_i)) \tag{4}$$

bits, where $v_1, v_2, \ldots, v_m$ are the lengths of the values.

We remark that the limitation on value size to be $O(1)$ machine words is simply so that each value can be written/read in constant time, since then it is easy to discuss how the asymptotics of T and T' compare. The same techniques work for even larger values without modification, as long as one is willing to spend the necessary time to read/write values that are of super-constant size.

Proof of Theorem 7.5. Values in T' are stored with up to r levels of indirection. If a value is v bits, then it is pointed at by a tiny pointer p_1 of size $O(\log n)$ bits. The tiny pointer p_1 is, in turn, pointed at by a tiny pointer p_2 of size $O(\log \log n)$ bits, and so on, with pointers of size $O(\log \log \log n), O(\log \log \log \log n), \ldots, O(\log^{(r)} n)$. That is, every value is stored at the end of a linked list of length $O(1)$, where the base pointer of the linked list is $O(\log^{(r)} n)$ bits, and each subsequent pointer is exponentially larger than the previous one.

For each tiny pointer of some size j in the data structure, we must also store $O(j)$ extra bits of information indicating (a) whether the tiny pointer is pointing at another tiny pointer or at a final value, and (b) what the size is of the tiny-pointer/value being pointed at. Throughout the rest of the proof, we will count these $O(j)$ extra bits as being part of the size of the tiny pointer.

With this in mind, we can now formally define what we mean by the "levels of indirection" discussed earlier. Recall that, when we dereference a tiny pointer, we obtain a slot in a dereference table. This slot will either contain another tiny pointer (including the auxillary information from he previous paragraph) or a final value (i.e., the value for some key). This is how we can have tiny pointers pointing at tiny pointers, etc., with multiple layers of indirection until we get to the actual value associated with a key.

Since there are both values and tiny pointers of many different sizes, we must use a different dereference table for each size-class of tiny-pointer and the different dereference table for each size-class of values being stored. (Note that the dereference tables storing tiny pointers may need to use the chunked-pointer-storage technique to handle variable-size tiny pointers, so the same dereference table should not be used to store both tiny pointers and values.)

The problem of dynamically resizing all of the dereference tables simultaneously is slightly tricky. Consider a dereference table A (to A could also be the value array) that stores j-bit tiny pointers for some j. There are $K = 2^{\Theta(j)}$ different dereference tables $B_1, B_2, \ldots, B_K$ that these tiny pointers can point into (depending on the size of the object being pointed at, and whether the object is a tiny pointer or a value). Each B_i must individually be dynamically resized. We will maintain what we call the ***dynamic-sizing invariant***, which guarantees that each B_i is either (a) at a load factor $1 - O(1/j')$, where j' is the size of the objects stored in B_i, or (b) is at most a $o(1/(Kj))$-fraction the size (in bits) of A.

To implement the dynamic-sizing invariant, we dynamically resize each B_i using zone-aggregated resizing (recall from Section 7.1 that this means B_i is broken into multiple components, and each component is occasionally rebuilt so that its size either doubles or halves). To allow for components of each B_i to be rebuilt efficiently, we break A into zones of size $\text{poly}(K)$, meaning by (3) from Section 7.1 that a given component (of some B_i) consisting of s entries can be rebuilt in time

$$|A|/\text{poly}(K) + s,$$

where $|A|$ is the number of entries in A. We perform dynamic resizing on B_i differently depending on whether it is very small (its components contain fewer than $|A|/\text{poly}(K)$ items each) or not:

- If the components contain $s = \Omega(|A|/\text{poly}(K))$ items each, then we perform zone-aggregated resizing (exactly as in Section 7.1) to keep B_i at a load factor $1 - O(1/j')$, where j' is the size of the objects stored in B_i. In this case, the time needed to rebuild a component of size s is $\Theta(s)$, so the dynamic resizing of B_i can be deamortized to take $O(1)$ time per operation (on B_i). Note that, here, B_i is in case (a) of the dynamic-resizing invariant.
- If the components contain fewer than $|A|/\text{poly}(K)$ items each, then we perform zone-aggregated resizing to keep each component of B_i at a capacity of $\Theta(|A|/\text{poly}(K))$ (even as $|A|$ changes over time, and *regardless* of whether the number of items per component may be significantly smaller than $|A|/\text{poly}(K)$). Note that, here, B_i is in case (b) of the dynamic-resizing invariant.

  When B_i is in this regime, we cannot amortize the work spent rebuilding B_i to the operations that are performed on B_i. Instead, we spread out the work spent rebuilding components of B_i in the following way: for every $\Theta(K)$ work that is spent on A we also spend $O(1)$ time on resizing B_i. Since B_i is more than a factor of K smaller than A, this is sufficient time to keep B_i in a state where each component has capacity $\Theta(|A|/\text{poly}(K))$.

  From the perspective of A, every time that we spend constant time on insertions/deletions/rebuilding A, we also may spend constant time performing rebuild-work on one of the B_is (which, in turn, may recursively lead us to spend constant time on rebuilding one of the dereference tables pointed at by B_i, etc.). Importantly, since chains of tiny pointers are at most $r \leq O(1)$ long, the time spent on rebuilds only introduces a constant-factor overhead on running time per operation.

The resizing approach described above guarantees the dynamic-sizing invariant while incurring only a constant-factor time overhead per operation. Next we use the invariant to bound the space consumption of T'. The dereference tables B_i in case (a) are implemented space-efficiently enough that the empty slots in them take negligible space compared to the actual objects stored in them (i.e., the empty slots add $O(1)$ bits per object), and the dereference tables B_i in case (b) are small enough that they take negligible space compared to the size of the parent dereference table A (i.e., they cumulatively add $o(1)$ bits per slot in A). It follows that the total space consumed by dereference tables will be at most the sum of the sizes of the objects being stored in the dereference tables, plus $O(1)$ bits per object; this, in turn, means that the space used by T' to store values/tiny pointers is given by (4).

Next, we bound the time-overhead of T' when compared to T. We have already shown that the time-overhead of performing dynamic-resizing on dereference tables is $O(1)$ per operation. Since values are stored with at most $r = O(1)$ levels of indirection, the time needed to access/modify a value is also $O(1)$. Thus T' has the same time asymptotics as T.

Finally, we argue that the dereference tables used by T' succeed at their allocations with high probability.[14] There are several approaches that we could take to doing this; the simplest is to just add one more modification to how we perform dereference-table resizing: whenever a dereference table gets down to size $\Theta(\sqrt{m})$, we do not ever resize it to be any smaller.[15] This means that some dereference tables could be very sparse, containing $\sqrt{m}$ slots, but containing far fewer items. Since there are only $O(w) = m^{o(1)}$ different dereference tables (recall that w is the machine-word size), the net space consumption of the dereference tables of size $\Theta(\sqrt{m})$ is $o(m)$ bits. The fact that every dereference table has size at least $\Omega(\sqrt{m})$ means that all of the dereference tables offer high probability guarantees, as desired. □

By applying our result to the data structure from [13], which is a hash table with redundancy $O(n \log^{(O(1))} n)$ bits, we obtain the following corollary for hash tables.

Corollary 7.6. *Let r be a positive constant of our choice. It is possible to construct a dynamically-resized hash table that stores keys from a universe U and that stores dynamic-sized value (up to $O(1)$ machine words) with the following guarantees.*

At any given moment, if n keys are present, then with high probability in n, the time spent on the next operation is $O(1)$ and the overall space usage is

$$\log \binom{|U|}{n} + O(n \log^{(r)} n) + \sum_{i=1}^{m} (v_i + O(\log v_i)) \tag{5}$$

bits, where $v_1, v_2, \ldots$ are the sizes of the values.

7.6 An Optimal Internal-Memory Stash

Our final application of tiny pointers revisits one of the oldest problems in external-memory data structures: the problem of maintaining a small internal-memory ***stash*** that allows for one to locate where items reside in a large external-memory data structure.

The problem can be formalized as follows. We must store a dynamically changing set S of up to n key-value pairs, where each key-value pair can be stored in one machine word, and where each key is unique. We are given an ***external memory*** consisting of $(1 + \varepsilon)n$ machine words, where the key-value pairs S are to be stored. In addition to storing key-value pairs in external memory, we must maintain a small internal-memory data structure X, which we will refer to as the ***stash***, that supports the following operations:

- **Query**(k)**:** Using only information in the stash data structure, returns the position in external memory where the key k and its corresponding value v are stored.
- **Insert**(k, v)**:** Inserts the key-value pair (k, v), placing the pair somewhere in external memory, and updating the stash.
- **Delete**(k, v)**:** Removes the key/value pair (k, v) from the external-memory array, and updates the stash.

[14] There are many different ways that one could handle allocation failures, including, for example, performing batch-rebuilds of the data structure.
[15] However, since m may dynamically change over time, we do need to spend constant time per operation resizing dereference tables of size $\Theta(\sqrt{m})$ so that they stay size $\Theta(\sqrt{m})$ as m changes.

The important feature of a stash is that queries can be completed with a single access to external memory. On the other hand, in order for a stash to be useful, several other objectives must be achieved:

- **Compactness:** The stash X needs to be as small as possible, that way it can fit into an internal memory with limited size.
- **Efficient inserts and deletes:** Although a stash prioritizes queries, insertions and deletions should ideally also require only $O(1)$ accesses/modifications to external memory.
- **RAM efficiency:** Finally, so that computational overhead does not become a bottleneck, the operations on a stash should be as efficient as possible in the RAM model, ideally taking time $O(1)$.

A concrete example of a stash that is used in real-world systems is the ***page table*** [4, 9, 40], which is an operating-system-level dictionary that maps virtual page addresses to where their corresponding physical pages reside in memory. The page table is accessed for every address translation, so it is performance critical and thus highly optimized. Additionally, it is important that the page table be space-efficient, so that it may be effectively cached in the processor cache hierarchy. Note that, although page tables get to select where physical pages reside in memory, they do not get to move physical pages that have already been placed; thus any stash that is used as a page table must also be stable. For this reason, past work [36, 44, 45] has typically included stability as an additional criterion for a stash.

Work on designing space-efficient and time-efficient stashes dates back to the late 1980s [36, 44, 45]. The best-known theoretical results are due to Gonnet and Larson [36], who give a stable stash that uses only $O(n \log \varepsilon^{-1})$ bits. A remarkable consequence of this is that, when $\varepsilon = \Theta(1)$, it is possible to construct a stash using only $O(n)$ bits.

Gonnet and Larson's result comes with several significant drawbacks, however [36], which have proven difficult to fix. First, due to its reliance on stable uniform probing [43] as a mechanism for determining where keys/values should reside, the stash only offers provable guarantees in the setting where insertions/deletions are performed *randomly*. Second, the data structure is not constant-time in the RAM model, instead taking expected time $\Theta(\varepsilon^{-1})$.

Using tiny pointers, we show that modern techniques for constructing filters can easily be adapted in order to construct a stable stash of size $O(n \log \varepsilon^{-1})$ bits that supports constant-time operations in the RAM model (with high probability) and that supports arbitrary sequences of insertions/deletions/queries.

Theorem 7.7. *It is possible to construct a stable stash that supports constant-time operations in the RAM model, that stores up to m keys/values in an external-memory array of size $(1 + \varepsilon)m$, and that uses only $O(m \log \varepsilon^{-1})$ bits of internal-memory space. All of the guarantees for the stash hold with high probability in m.*

Proof. The starting point for our design is the adaptive filter of Bender et al. [11]. Like a stash, their filter is a space-efficient internal-memory data structure that summarizes the state of an external-memory key-value dictionary. Unlike a stash, their filter does not indicate where in external memory each key/value is stored. Instead, the filter answers containment queries with the following guarantee: each positive query is guaranteed to return true, and each negative query is guaranteed to return false with probability at least $1-\varepsilon$ (for some parameter ε). The size of their internal-memory data structure is only $(1+o(1))m \log \varepsilon^{-1} = O(m \log \varepsilon^{-1})$ bits, where m is the capacity of the filter.[16]

The basic idea behind the adaptive filter of [11] is to store a ***fingerprint*** for each key x, where each fingerprint is taken to be some prefix of the hash $h(x)$. Different keys have different-length fingerprints, and the invariant maintained by the filter is that no fingerprint is a prefix of any other fingerprint. To maintain this invariant while also keeping the fingerprints as small as possible, the filter will sometimes change the lengths of $O(1)$ different fingerprints during a given insertion/deletion; to change the length of a fingerprint, the key corresponding to that fingerprint must first be fetched from external memory, that way the hash $h(x)$ of that key can be recomputed.[17]

[16] In fact, their data structures also dynamically resizable, but for our application that will not be necessary.

[17] The original data structure also sometimes updates the lengths of fingerprints during negative queries, but such updates are not needed for the purposes of our data structure.

The fingerprints in the filter are stored as follows. The first $\lg n$ bits of each fingerprint are called the ***quotient***, and these bits are used to assign the key to one of n bins; importantly, the fact that the bin-choice encodes the quotient of each of the keys in the bin means that the data structure does not have to explicitly store the quotients of the fingerprints. The next $\log \varepsilon^{-1}$ bits of each fingerprint are called the ***baseline bits***, and these bits are included for every fingerprint in the data structure. Finally, any subsequent bits in a fingerprint are called the ***adaptivity bits***, and these bits are added/removed in order to maintain the prefix-freeness invariant. A central piece of [11]'s analysis is to show that there are only $O(m)$ adaptivity bits in total, and that these bits can be stored efficiently.

We now describe how to modify the filter to be a stash. In addition to storing a fingerprint for each key, we now also store a tiny pointer with expected size $\Theta(\log \varepsilon^{-1})$. These tiny pointers are easy to store, since the filter has already made room for $\log \varepsilon^{-1}$ baseline bits for each key. Of course, different tiny pointers may have different lengths, but this issue can easily be resolved by either using the chunked-pointer-storage technique described in Section 7.1 (or by adapting the techniques already used in [11] to handle variable-length fingerprints).

One minor difficulty is that the filter assumes access to an external-memory dictionary (rather than just a dereference table) so that it can look up keys in order to modify their fingerprints. In the case of our stash, however, these lookups can easily be performed using the tiny pointers that are already stored, so one does not need a full dictionary in external memory.

The fact that the tiny pointers have size $\Theta(\log \varepsilon^{-1})$ means that external memory can be implemented as a dereference table with load factor $1 - \varepsilon$. The fact that the original adaptive filter supported constant-time operations (with high probability in m) translates to the stash also supporting constant-time operations. And the fact that the original adaptive filter used space $O(m \log \varepsilon^{-1})$ bits in internal memory also translates the same guarantee for the stash. Thus the theorem is proven.

□

8 DYNAMIC BALLS AND BINS

In this section, we reinterpret our tiny-pointer constructions as balls-and-bins schemes in order to improve the state of the art for the classic dynamic load balancing problem.

In the dynamic load-balancing problem, there is a system of n bins and a large universe U of balls. Balls are inserted and deleted (and sometimes reinserted) over time by an oblivious adversary, so that the total number of balls in the system never exceeds $m = hn$ for some parameter h. Whenever a ball x is inserted, it must be placed in one of d bins from among $h_1(x), \dots, h_d(x)$, where $h_i()$ is some hash function from balls to bins. Once a ball is placed in a bin, it cannot be moved until it is deleted. The goal of the ***dynamic load-balancing problem*** is to assign balls to bins in order to achieve the smallest maximum load possible (i.e., to minimize the number of balls in the fullest bins). We refer to the special case where balls can be inserted and deleted but not reinserted as the ***semi-dynamic load-balancing problem.***

There are two classic solutions to the problem. The first is Single balls-to-bins assignment: we set $d = 1$ and just place each x in $h_1(x)$. The second is Left$[d]$ balls-to-bins assignment: divide the bins into d groups so that each h_i is uniform into the i-th group; when inserting x, pick the bin $h_i(x)$ with the smallest load, and break ties by minimizing i.

Single's behavior is history independent, in that the maximum load at any time only depends on which balls are present, and not the history of their arrival. The maximum load is then completely characterized by standard Chernoff bounds [54].

Left$[d]$, on the other hand, is history dependent. The first time that a ball x is inserted, the hashes $h_1(x), \dots, h_d(x)$ are independent of the system state, but if a ball x is ever deleted and then later *reinserted*, then the past insertion of x can have long-term side effects on the system state meaning that the state is not necessarily independent of $h_1(x), \dots, h_d(x)$.

In the insertion-only setting (i.e., balls are not deleted), Left$[d]$ offers a celebrated bound [59] of

$$h + \frac{\log \log n}{d \log \phi_d} + O(1) \tag{6}$$

on maximum load, where ϕ_d is the generalized golden ratio. In the dynamic setting, Left$[d]$ has proven to be significantly more difficult to analyze. The original analysis of Left$[d]$ by Vöcking [59] can be used to achieve a bound of

$$O(h) + \frac{\log \log n}{d \log \phi_d} \tag{7}$$

for the semi-dynamic setting, but as Woelfel observed [60], the same argument does not apply directly to the fully dynamic setting.[18] He shows how to modify Vöcking's proof to achieve a bound of

$$O(d) + \frac{\log \log n}{d \log \phi_d} \tag{8}$$

in the setting where $h = 1$. In general, when $h > 0$, Woelfel's argument yields a bound of

$$O(1 + hd) + \frac{\log \log n}{d \log \phi_d}, \tag{9}$$

which has remained the state of the art.

The bound (9) is most interesting in the case where h is relatively small, that is, $h = o(\log n)$. Here, (9) can be significantly better than the $\Theta(\log n / \log \log n)$ bound that would be achieved by Single. Of course, the question remains as to whether there exists a balls-to-bins scheme that achieves a better bound. We answer this question in the affirmative, by giving a bin-selection rule with $d + 1$ hash functions that achieves maximum load

$$h + \frac{\log \log n}{d \log \phi_d} + O(\sqrt{h \log(hd)}). \tag{10}$$

We remark that, even when h is a constant, this bound improves the dependence on d from $O(d)$ to $O(\sqrt{\log d})$.

Our rule, which we call ***Iceberg***$[d]$ is a hybrid of Single and Left$[d]$. This rule is closely related to the rule that we used in Section 4 for constructing fixed-size tiny pointers. This rule was analyzed in our paper on iceberg hashing [10], hence the name. As noted above, here we present an alternative proof, both for completeness, as well as because the proof takes a somewhat different (and more elegant) approach than in our past work, and in order to cover a larger parameter regime.

The rest of the section proceeds as follows. We begin in Subsection 8.1 by proving a useful technical lemma. In Subsection 8.2, we present and analyze Iceberg$[d]$. Finally, in Subsection 8.3, we reinterpret our variable-size tiny-pointer construction as a result about probe-complexity of balls-and-bins schemes with bins of capacity 1; in particular, we give the first dynamic ball-allocation scheme to offer $\delta^{-O(1)}$ average probe complexity in the setting where there are up to $(1 - \delta)n$ balls present in the system at a time.

[18]The difficulty has to do with the analysis of the leaves in the witness tree, and is easy to describe in the case where $h = 1$. To analyze a leaf ball x, the original analysis uses Markov's inequality to deduce that each of x's d bins has at most a $1/3$ probability of having 3 or more balls, and the analysis concludes that the probability of all d bins containing 3 or more balls is at most $1/3^d$. This same analysis does not apply in the fully dynamic setting since it would need the state of the system of to be independent of x's hash functions $h_1(x), \ldots, h_d(x)$, which is not the case due to subtle history dependencies in the system's state.

8.1 A Useful Lemma

This section proves a generalization of a technical lemma introduced in recent work on space-efficient hash tables [10]. The new lemma extends the original one to a wider parameter regime. We also take a different combinatorial approach, resulting in a simpler proof that reveals an interesting relationship between the lemma and Talagrand's inequality. We remark that, earlier in the paper, we have already made use of the results from this section in order to obtain Lemma 4.2.

Consider a dynamic balls-and-bins game with n bins and at most $m = hn$ balls at all times that are placed with the Single rule. Whenever a ball is thrown into a bin, if the bin contains $h + \tau$ or more balls, then the ball is labeled as ***τ-exposed*** (and the label persists until the ball is next deleted). We will simply say that the ball is ***exposed*** when τ is clear from context.

Lemma 8.1. *Suppose $1 \le \tau \le h$. At any fixed point in time, the number of τ-exposed balls is* $\mathrm{poly}(h) \cdot ne^{-\tau^2/(3h)}$ *with probability* $1 - \exp(-\Omega(me^{-\tau^2/(3h)}))$.

Our proof of the Lemma 8.1 will make use of a variant of Talagrand's inequality [47, Chapter 10]:

Theorem 8.2 (Talagrand's inequality). *Let $X_1, \dots, X_n$ be n independent random variables from an arbitrary domain. Let F be a non-negative function of $X_1, \dots, X_n$, not identically 0. Suppose that for some $c, r > 0$, F is c-Lipschitz and r-certifiable, defined as follows:*

- *F is c-Lipschitz if changing the outcome of any single X_i changes F by at most c.*
- *F is r-certifiable if, for any s, if $F(X_1, \dots, X_n) \ge s$, then there is a certifying set of at most rs X_i's whose outcomes serve as a witness that $F \ge s$, that is, $F \ge s$ no matter the outcome of the other X_j not in the certifying set.*

Then, for any $0 \le t \le \mathbb{E}[F]$,

$$\Pr\left[|F - \mathbb{E}[F]\,| > t + 60c\sqrt{r\,\mathbb{E}[F]}\right] \le 4\exp\left(-\frac{t^2}{8c^2 r\,\mathbb{E}[F]}\right).$$

The proof of the Lemma 8.1 proceeds by bounding the expected number of exposed balls, then using Talagrand's inequality to achieve a concentration bound.

Consider any fixed point in time. In what follows, we refer to the balls that at the end are present at that time as $a_1, \dots, a_k$ and we refer to the remaining balls in the universe as $a_{k+1}, \dots, a_\ell$. We denote by α_i the bin choice for a_i. For $i \in [k]$, we define t_i to be the last time at which a_i is inserted, we define X_i to be the random variable indicating if a_i is an exposed ball at the end of the game, and we define $X = \sum_{i=1}^{k} X_i$ to be the total number of exposed balls.

Claim 8.3. *The expected number of exposed balls satisfies* $\mathbb{E}[X] = O(me^{-\tau^2/(3h)})$.

Proof. Recall that $X = \sum_i X_i$ where X_i indicates whether a_i is exposed. By linearity of expectation, it suffices to show that $\mathbb{E}[X_i] = O(e^{-\tau^2/(3h)})$ for each $i \in [k]$.

Fix $i \in [k]$. Consider the final time t_i at which ball a_i is inserted. The ball a_i is exposed if and only if the number of balls in bin α_i is at least $h + \tau$. If we set Y to be the number of balls in bin α_i, and we set $\varepsilon = \tau/h$, then we can bound the probability of $Y \ge h + \tau$ using a Chernoff bound:

$$\Pr[Y \ge h + \tau] = \Pr[Y \ge (1 + \varepsilon)h] \le e^{-\varepsilon^2 h/3} = e^{-\tau^2/(3h)}.$$

Thus $\mathbb{E}[X_i] = \Pr[X_i] \le e^{-\tau^2/(3h)}$. □

Claim 8.4. *The random variable X is $(h + \tau + 1)$-Lipschitz and $(h + \tau + 1)$-certifiable as a function of $\{\alpha_i\}_{i=1}^{\ell}$.*

Proof. Changing the value of a single α_i to α_i' can only affect the number of exposed balls in bin α_i (which may decrease) and in bin α_i' (which may increase). The number of *unexposed* balls in a bin is deterministically at most $h+\tau$. This means that moving ball a_i out of bin α_i can increase the number of unexposed balls in the bin by at most $h + \tau$, and thus can decrease the number of exposed balls by at most $h + \tau + 1$ (where the +1 accounts for the removal of a_i itself). Similarly, moving ball a_i into bin α'_i can decrease the number of unexposed balls in the bin by at most $h + \tau$, and thus can increase the number of exposed balls by at most $h+\tau+1$. This establishes that X is $(h+\tau+1)$-Lipschitz.

To certify that $X \geq s$, let J with $|J| = s$ be a set of values $j \in [k]$ such that a_j is exposed at the end of the game. For each $j \in J$, let R_j be a selection of $h+\tau$ balls a_i such that ball a_i was present at the last time t_j that a_j was inserted and such that $\alpha_i = \alpha_j$. The set of random variables $\{\alpha_i \mid i \in R_j\} \cup \{\alpha_j\}$ acts as a certificate that a_j is exposed. Thus the set

$$\bigcup_{j \in J} \{\alpha_i \mid i \in R_j\} \cup \{\alpha_j\}$$

acts as a certificate that $X \geq s$. This certificate consists of no more than $s(h + \tau + 1)$ random variables, hence X is $(h + \tau + 1)$-certifiable. □

Proof of Lemma 8.1. Set $Q = m \exp\left(-\tau^2/(3h)\right)$. By Claim 8.3, we know that $\mathbb{E}[X] \leq Q$. By Claim 8.4, we can apply Talagrand's inequality (Theorem 8.2) to X with $c = r = h + \tau + 1 = O(h)$. Applying Talagrand's inequality with $t = \Theta(c\sqrt{r}Q)$, and using Q as an upper bound on $\mathbb{E}[X]$, we can deduce that

$$X = O(c\sqrt{r}Q)$$

with probability at least

$$1 - \exp(-\Omega(Q)).$$

It follows that $X \leq \text{poly}(h) \cdot O(ne^{-\tau^2/(3h)})$ with probability $1 - \exp(-\Omega(me^{-\tau^2/(3h)}))$. □

Finally, we complete the section by using Lemma 8.1 to prove Lemma 4.2 from Section 4.

Proof of Lemma 4.2. We can prove Lemma 4.2 by applying Lemma 8.1 and setting the parameters $h = (1 - \delta)b$ and $\tau = \delta b$. The τ-exposed balls correspond to allocations in Lemma 4.2 that have failed and are still alive.

Note that, in this setting,

$$\tau^2/(3h) = \Theta(\tau^2/b) = \Theta(\delta^2 b) = c' \log \delta^{-1}$$

for some large positive constant c', where the last step uses the fact that $b = c\delta^{-2} \log \delta^{-1}$ for some large positive constant c.

Therefore, Lemma 8.1 bounds the number of such balls to be at most

$$\begin{aligned} \text{poly}(h) \cdot ne^{-\tau^2/(3h)} &\leq \text{poly}(\delta^{-1}) \cdot ne^{-c' \log \delta^{-1}} \\ &\leq \delta n \end{aligned}$$

with probability

$$\begin{aligned} 1 - \exp(-\Omega(me^{-\tau^2/(3h)})) &= 1 - \exp(-\Omega(me^{-c' \log \delta^{-1}})) \\ &= \exp(-m\delta^{O(1)}), \end{aligned}$$

as desired. □

8.2 Iceberg[d]

We now present the Iceberg[d] balls-in-bins rule. Let n be the number of bins, let hn be the maximum number of balls allowed to be present at any given moment, and let $d > 1$ be a parameter. Partition the bins into d equal-size sets $S_1, \ldots, S_d$. Let g be a hash function mapping balls uniformly at random to bins, and let $h_1, \ldots, h_d$ be hash functions such that each h_i maps balls uniformly at random to a random bin in S_i.

We shall have three types of balls: level-one balls, level-two balls, and level-three balls. Each level-one ball x will reside in bin $g(x)$, each level-two ball x will reside in one of bins $h_1(x), \ldots, h_d(x)$, and each level-three ball x will reside in bin 1 (but, at any given moment, the number of level-three balls will be zero w.h.p.).

Set $\tau = c\sqrt{h \log(hd)}$ for some sufficiently large positive constant c. We shall also keep track of a variable q counting the number of level-two balls present at any given moment.

The procedure for inserting a ball x is as follows. If bin $g(x)$ contains less than $h+\tau$ level-one balls, then we place x in bin $g(x)$, and we classify x as a level-one ball. Otherwise, we check whether $q < n/d$. If $q < n/d$, then we examine bins $h_1(x), \ldots, h_d(x)$, and we place x as a level-two ball into whichever bin $h_i(x)$ contains the fewest level-two balls (breaking ties towards the smallest i). Finally, if $q \geq n/d$, then we place x as a level-three ball into bin 1.

Theorem 8.5. *Suppose $1 \leq h \leq n^{o(1)}$ and $1 < d \leq n^{o(1)}$. Suppose balls are inserted/deleted/reinserted into n bins over time (by an oblivious adversary) according to* Iceberg$[d]$ *rule, with no more than hn balls present at a time. Then, w.h.p. in n, at any given moment, the number of balls in the fullest bin is $h + \frac{\log\log n}{d \log \phi_d} + O(\sqrt{h \log(hd)})$.*

Proof. Each bin deterministically contains at most $h + \tau = h + O(\sqrt{h \log(hd)})$ level-one balls. Thus, it suffices to bound the number of level-two and level-three balls in each bin by $\frac{\log\log n}{d \log \phi_d} + O(1)$.

The number q of level-two balls in the entire system is deterministically at most n/d at any given moment. In other words, the level-two balls are placed according to the Left$[d]$ rule with $h'n$ balls, where $h' = 1/d$. Thus we can apply (9) to deduce that, w.h.p., the maximum number of such balls per bin is

$$O(1 + h'd) + \frac{\log\log n}{d \log \phi_d} = \frac{\log\log n}{d \log \phi_d} + O(1),$$

Note that, in this application of (9), we are using Woelfel's analysis [60] of Left$[d]$ in a somewhat unusual parameter regime; that is, the analysis is intended primarily to be used in the regime $h' \geq 1$ (and Woelfel's result was only explicitly stated for $h' = 1$), but we are taking advantage of the fact that the analysis also holds for $h' = o(1)$ without modification.

We complete the proof by showing that, w.h.p. The number of level-three bins is zero. By Lemma 8.1, the number q of level-two balls satisfies $q < n/h$ (at any given moment) with probability at least $1 - \exp(-n/(hd)^{O(1)})$, which by the assumption $h, d \leq n^{o(1)}$ is at least $1 - 1/\text{poly}(n)$. It follows that each individual ball insertion has probability at most $1/\text{poly}(n)$ of being level-three. Taking a union bound over all of the balls in the system, the probability that any of them are level-three is $1/\text{poly}(n)$, as desired. □

8.3 Assigning Balls to Capacity-1 Bins with Low Average Probe Complexity

Our final result of the section considers a dynamic balls-and-bins game in which there are n bins each with capacity 1, and at most $(1-\delta)n$ balls are present at a time. Each ball x has a predetermined (infinite) sequence $h_1(x), h_2(x), \ldots$ of bins where it can reside, and we wish to minimize the ***probe complexity*** of each ball x, which is defined to be the smallest i such that ball x is in bin $h_i(x)$. Since we are in the dynamic setting, the same ball may be inserted, deleted, and reinserted many times.

First note that, in the insertion-only setting, it is easy to achieve probe average complexity $O(\delta^{-1})$ by simply using uniform probing, which sets each $h_i(x)$ to be random, and places each ball x into the first available slot in the sequence $h_1(x), h_2(x), \ldots$. In the dynamic setting, however, there is not yet any known bin-assignment scheme that achieves average probe complexity $\delta^{-O(1)}$ (for example, uniform probing has only successfully been analyzed in the random-deletions setting [43], and the analysis of linear probing without moving items around remains an open problem [56]).

We now construct a bin-assignment scheme that achieves average probe complexity $\delta^{-O(1)}$.

Theorem 8.6. *Suppose $\delta = 1/n^{o(1)}$. There exists a bin-assignment scheme that supports arbitrary ball insertions/deletions/reinsertions, and guarantees an expected probe complexity of $O\delta^{-O(1)}$ for each ball in the system.*

Proof. Consider a variable-size-tiny-pointer dereference table with n slots and load factor $1 - \delta$. For each ball x and each $i \in \mathbb{N}$, define $h_i(x) = \text{Dereference}(x, i)$. To assign a ball x to a bin, we call the function $i = \text{Allocate}(x)$, and place x into bin $h_i(x) = \textsc{Dereference}(x, i)$. To delete a ball x, we call $\textsc{Free}(x, i)$ in order to deallocate the appropriate slot in the dereference table.

Let $c > 0$ be a sufficiently small positive constant. By Theorem 5.1, each ball x gets assigned to a bin $h_i(x)$ where i (which is the tiny pointer returned by $\textsc{Allocate}(x)$) is

$$O(\log \delta^{-1} + P)$$

bits for some random variable P satisfying $\Pr[P \geq j] \leq O\left(2^{-2^{cj}}\right)$. It follows that $i \leq \delta^{-O(1)} k$ with probability at least $1 - O(2^{-2^{c \log k}}) = 1 - O(2^{k^c})$, and hence that the expected probe complexity of each ball x is $\delta^{-O(1)}$. □

9 CONCLUSION

This paper introduces a new data-structural object that we call the tiny pointer. We use tiny pointers to produce several space-efficient data structures.

Our work suggests several open problems. Top among these is: can tiny pointers be used to make data structures space efficient in practice? Our related work [9, 37] on address translation in virtual-memory systems uses ideas closely related to tiny pointers to compress pointers to where a page is located in cache. This can be shown to improve the performance of address translation hardware. The natural question is if tiny pointer techniques can be used elsewhere in practice.

The other open problem relates to pointers in graphs. In the tiny pointer setup, each item is pointed to by a single user. Thus, trees are easily encoded because each node has one parent. It is probably too much to hope to extend these ideas to general graphs. Are there classes of graphs for which tiny pointers can be generalized?

ACKNOWLEDGMENTS

We would like to thank Sepehr Assadi for helpful comments that led to the proof of Lemma 8.1.

This research was supported in part by NSF grants CSR-1938180, CCF-2106999, CCF-2118620, CCF-2118832, CCF-2106827, CCF-1725543, CSR-1763680, CCF-1716252 and CNS-1938709, as well as an NSF GRFP fellowship and a Fannie and John Hertz Fellowship.

This research 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] Abseil. 2024. Google's Abseil C++ Library. https://abseil.io/. Accessed: 2024-07-18.

[2] George Adel'son-Vel'skii and Evgenii Landis. 1962. An algorithm for organization of information. In *Doklady Akademii Nauk*, Vol. 146. Russian Academy of Sciences, 263–266.

[3] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. 2001. Optimal static range reporting in one dimension. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis (Eds.). ACM, 476–482. https://doi.org/10.1145/380752.380842

[4] AMDManual. 2024. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf. Accessed: 2024-07-18.

[5] Yuriy Arbitman, Moni Naor, and Gil Segev. 2009. De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results. In *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 5555)*, Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas (Eds.). Springer, 107–118. https://doi.org/10.1007/978-3-642-02927-1_11

[6] Yuriy Arbitman, Moni Naor, and Gil Segev. 2010. Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. IEEE Computer Society, 787–796. https://doi.org/10.1109/FOCS.2010.80

[7] Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and IgorAleksandrovich Faradzhev. 1970. On economical construction of the transitive closure of an oriented graph. In *Doklady Akademii Nauk*, Vol. 194. Russian Academy of Sciences, 487–488.

[8] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. 1999. Balanced Allocations. *SIAM J. Comput.* 29, 1 (Sept. 1999), 180–200. https://doi.org/10.1137/S0097539795288490

[9] Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martin Farach-Colton, Rob Johnson, Sudarsun Kannan, William Kuszmaul, Nirjhar Mukherjee, Donald E. Porter, Guido Tagliavini, Janet Vorobyeva, and Evan West. 2021. Paging and the Address-Translation Problem. In *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, Kunal Agrawal and Yossi Azar (Eds.). ACM, 105–117. https://doi.org/10.1145/3409964.3461814

[10] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once. *J. ACM* 70, 6 (2023), 40:1–40:51. https://doi.org/10.1145/3625817

[11] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom Filters, Adaptivity, and the Dictionary Problem. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, Mikkel Thorup (Ed.). IEEE Computer Society, 182–193. https://doi.org/10.1109/FOCS.2018.00026

[12] Michael A. Bender, Martin Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. 2022. On the optimal time/space tradeoff for hash tables. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, Stefano Leonardi and Anupam Gupta (Eds.). ACM, 1284–1297. https://doi.org/10.1145/3519935.3519969

[13] Michael A. Bender, Martin Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. 2022. On the optimal time/space tradeoff for hash tables. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, Stefano Leonardi and Anupam Gupta (Eds.). ACM, 1284–1297. https://doi.org/10.1145/3519935.3519969

[14] Ioana Oriana Bercea and Guy Even. 2020. A Space-Efficient Dynamic Dictionary for Multisets with Constant Time Operations. *CoRR* abs/2005.02143 (2020). arXiv:2005.02143 https://arxiv.org/abs/2005.02143

[15] Ioana O. Bercea and Guy Even. 2023. Dynamic Dictionaries for Multisets and Counting Filters with Constant Time Operations. *Algorithmica* 85, 6 (2023), 1786–1804. https://doi.org/10.1007/S00453-022-01057-0

[16] c++ std::map. 2023. cppreference std::map. https://en.cppreference.com/w/cpp/container/map. Accessed: 2024-07-18.

[17] c++ std::set. 2024. cppreference std::set. https://en.cppreference.com/w/cpp/container/set. Accessed: 2024-07-18.

[18] c++ std::unordered_map. 2024. cppreference std::unordered_map. https://en.cppreference.com/w/cpp/container/unordered_map. Accessed: 2024-07-18.

[19] c++ stl_set.h. 2024. gcc-mirror/gcc libstdc++-v3 stl_set.h. https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_set.h. Accessed: 2024-07-18.

[20] c++ sty_map.h. 2024. gcc-mirror/gcc libstdc++-v3 stl_map.h. https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_map.h. Accessed: 2024-07-18.

[21] c++ unordered_map.h. 2024. gcc-mirror/gcc libstdc++-v3 unordered_map.h. https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/unordered_map.h. Accessed: 2024-07-18.

[22] c++ unordered_set. 2024. cppreference std::unordered_set. https://en.cppreference.com/w/cpp/container/unordered_set. Accessed: 2024-07-18.

[23] c++ unordered_set.h. 2024. gcc-mirror/gcc libstdc++-v3 unordered_set.h. https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/unordered_set.h. Accessed: 2024-07-18.

[24] Joshimar Cordova and Gonzalo Navarro. 2016. Simple and efficient fully-functional succinct trees. *Theor. Comput. Sci.* 656 (2016), 135–145. https://doi.org/10.1016/J.TCS.2016.04.031

[25] Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. 2017. On Succinct Representations of Binary Trees. *Math. Comput. Sci.* 11, 2, 177–189. https://doi.org/10.1007/S11786-017-0294-4

[26] Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Puatracscu. 2006. De Dictionariis Dynamicis Pauco Spatio Utentibus (*lat.* On Dynamic Dictionaries Using Little Space). In *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3887)*, José R. Correa, Alejandro Hevia, and Marcos A. Kiwi (Eds.). Springer, 349–361. https://doi.org/10.1007/11682462_34

[27] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Track A: Algorithms, Automata, Complexity, and Games (Lecture Notes in Computer Science, Vol. 5125)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer, 385–396. https://doi.org/10.1007/978-3-540-70575-8_32

[28] Martin Dietzfelbinger and Michael Rink. 2009. Applications of a Splitting Trick. In *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 5555)*,

Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas (Eds.). Springer, 354–365. https://doi.org/10.1007/978-3-642-02927-1_30

[29] Martin Dietzfelbinger and Stefan Walzer. 2019. Constant-Time Retrieval with O(log m) Extra Bits. In *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany (LIPIcs, Vol. 126)*, Rolf Niedermeier and Christophe Paul (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:16. https://doi.org/10.4230/LIPICS.STACS.2019.24

[30] Martin Dietzfelbinger and Christoph Weidling. 2007. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.* 380, 1-2, 47–68. https://doi.org/10.1016/J.TCS.2007.02.054

[31] Martin Dietzfelbinger and Philipp Woelfel. 2003. Almost random graphs with simple hash functions. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, Lawrence L. Larmore and Michel X. Goemans (Eds.). ACM, 629–638. https://doi.org/10.1145/780542.780634

[32] F14 [n. d.]. Facebook's F14 Hash Table. https://engineering.fb.com/2019/04/25/developer-tools/f14/. Accessed: 2024-07-18.

[33] Arash Farzan and J. Ian Munro. 2011. Succinct representation of dynamic trees. *Theor. Comput. Sci.* 412, 24 (2011), 2668–2678. https://doi.org/10.1016/J.TCS.2010.10.030

[34] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. 2005. Space Efficient Hash Tables with Worst Case Constant Access Time. *Theory Comput. Syst.* 38, 2 (2005), 229–248. https://doi.org/10.1007/S00224-004-1195-X

[35] Gianni Franceschini and Roberto Grossi. 2003. Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees. In *Algorithms and Data Structures, 8th International Workshop, WADS 2003, Ottawa, Ontario, Canada, July 30 - August 1, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2748)*, Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Michiel H. M. Smid (Eds.). Springer, 114–126. https://doi.org/10.1007/978-3-540-45078-8_11

[36] Gaston H. Gonnet and Per-Åke Larson. 1988. External hashing with limited internal storage. *J. ACM* 35, 1 (1988), 161–184. https://doi.org/10.1145/42267.42274

[37] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martin Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter. 2023. Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 433–448. https://doi.org/10.1145/3582016.3582021

[38] Leonidas J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. IEEE Computer Society, 8–21. https://doi.org/10.1109/SFCS.1978.3

[39] Takao Gunji and E Goto. 1980. Studies on hashing part-1: A comparison of hashing algorithms with key deletion. *J. Information Processing* 3, 1 (1980), 1–12.

[40] Intel. 2024. Intel®64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, 3C, and 3D: System Programming Guide. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-3a-3b-3c-and-3d-system-programming-guide.html. Accessed: 2024-07-18.

[41] Don E. Knuth. 1963. Notes on "Open" Addressing.

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

[43] Per-Åke Larson. 1983. Analysis of Uniform Hashing. *J. ACM* 30, 4 (1983), 805–819. https://doi.org/10.1145/2157.322407

[44] Per-Åke Larson. 1988. Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval. *ACM Trans. Database Syst.* 13, 3 (1988), 366–388. https://doi.org/10.1145/44498.44500

[45] Per-Åke Larson and Ajay Kajla. 1984. File Organization: Implementation of a Method Guaranteeing Retrieval in One Access. *Commun. ACM* 27, 7 (1984), 670–677. https://doi.org/10.1145/358105.358193

[46] Mingmou Liu, Yitong Yin, and Huacheng Yu. 2020. Succinct Filters for Sets of Unknown Sizes. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference) (LIPIcs, Vol. 168)*, Artur Czumaj, Anuj Dawar, and Emanuela Merelli (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 79:1–79:19. https://doi.org/10.4230/LIPICS.ICALP.2020.79

[47] M. Molloy and B. Reed. 2013. *Graph Colouring and the Probabilistic Method*. Springer Berlin Heidelberg. https://books.google.com/books?id=gU3xCAAAQBAJ

[48] J. Ian Munro, Venkatesh Raman, and Adam J. Storm. 2001. Representing dynamic binary trees succinctly. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*, S. Rao Kosaraju (Ed.). ACM/SIAM, 529–536. http://dl.acm.org/citation.cfm?id=365411.365526

[49] Gonzalo Navarro and Kunihiko Sadakane. 2014. Fully Functional Static and Dynamic Succinct Trees. *ACM Trans. Algorithms* 10, 3 (2014), 16:1–16:39. https://doi.org/10.1145/2601073

[50] Anna Pagh and Rasmus Pagh. 2008. Uniform Hashing in Constant Time and Optimal Space. *SIAM J. Comput.* 38, 1 (2008), 85–96. https://doi.org/10.1137/060658400

[51] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/J.JALGOR.2003.12.002

[52] Mihai Pătraşcu. 2008. Succincter. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*. IEEE Computer Society, 305–313. https://doi.org/10.1109/FOCS.2008.83

[53] W. Wesley Peterson. 1957. Addressing for Random-Access Storage. *IBM J. Res. Dev.* 1, 2 (1957), 130–146. https://doi.org/10.1147/RD.12.0130

[54] Martin Raab and Angelika Steger. 1998. "Balls into Bins" - A Simple and Tight Analysis. In *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1518)*, Michael Luby, José D. P. Rolim, and Maria J. Serna (Eds.). Springer, 159–170. https://doi.org/10.1007/3-540-49543-6_13

[55] Rajeev Raman and S. Srinivasa Rao. 2003. Succinct Dynamic Dictionaries and Trees. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings (Lecture Notes in Computer Science, Vol. 2719)*, Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger (Eds.). Springer, 357–368. https://doi.org/10.1007/3-540-45061-0_30

[56] Peter Sanders. 2018. Hashing with Linear Probing and Referential Integrity. *CoRR* abs/1808.04602 (2018). arXiv:1808.04602 http://arxiv.org/abs/1808.04602

[57] Raimund Seidel and Cecilia R. Aragon. 1996. Randomized Search Trees. *Algorithmica* 16, 4/5 (1996), 464–497. https://doi.org/10.1007/BF01940876

[58] Daniel Sleator and Robert Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686. https://doi.org/10.1145/3828.3835

[59] Berthold Vöcking. 2003. How asymmetry helps load balancing. *J. ACM* 50, 4 (2003), 568–589. https://doi.org/10.1145/792538.792546

[60] Philipp Woelfel. 2006. Asymmetric balanced allocation with simple hash functions. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*. ACM Press, 424–433. http://dl.acm.org/citation.cfm?id=1109557.1109605

Received 23 July 2023; revised 7 August 2024; accepted 23 September 2024