Linear Probing Revisited: Tombstones Mark the Death of Primary Clustering

Michael A. Bender*
Stony Brook

Bradley C. Kuszmaul
Google Inc.

William Kuszmaul†
MIT

Abstract

First introduced in 1954, the linear-probing hash table is among the oldest data structures in computer science, and thanks to its unrivaled data locality, linear probing continues to be one of the fastest hash tables in practice. It is widely believed and taught, however, that linear probing should never be used at high load factors; this is because of an effect known as primary clustering which causes insertions at a load factor of $1 - 1/x$ to take expected time $\Theta(x^2)$ (rather than the intuitive running time of $\Theta(x)$). The dangers of primary clustering, first discovered by Knuth in 1963, have now been taught to generations of computer scientists, and have influenced the design of some of the most widely used hash tables in production.

We show that primary clustering is not the foregone conclusion that it is reputed to be. We demonstrate that seemingly small design decisions in how deletions are implemented have dramatic effects on the asymptotic performance of insertions: if these design decisions are made correctly, then even if a hash table operates continuously at a load factor of $1 - \Theta(1/x)$, the expected amortized cost per insertion/deletion is $\tilde{O}(x)$. This is because the tombstones left behind by deletions can actually cause an *anti-clustering* effect that combats primary clustering. Interestingly, these design decisions, despite their remarkable effects, have historically been viewed as simply implementation-level engineering choices.

We also present a new variant of linear probing (which we call graveyard hashing) that completely eliminates primary clustering on *any* sequence of operations: if, when an operation is performed, the current load factor is $1 - 1/x$ for some x, then the expected cost of the operation is $O(x)$. Thus we can achieve the data locality of traditional linear probing without any of the disadvantages of primary clustering. One corollary is that, in the external-memory model with a data blocks of size B, graveyard hashing offers the following remarkably strong guarantee: at any load factor $1 - 1/x$ satisfying $x = o(B)$, graveyard hashing achieves $1 + o(1)$ expected block transfers per operation. In contrast, past external-memory hash tables have only been able to offer a $1 + o(1)$ guarantee when the block size B is at least $\Omega(x^2)$.

Our results come with actionable lessons for both theoreticians and practitioners, in particular, that well-designed use of tombstones can completely change the asymptotic landscape of how the linear probing behaves (and even in workloads without deletions).

*Supported in part by NSF grants CCF-2106827, CCF-1725543, CSR-1763680, CCF-1716252, and CNS-1938709.

†Supported in part by an NSF GRFP fellowship and a Fannie and John Hertz Fellowship. Research was partially sponsored by the United States Air Force Research Laboratory 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.

1 Introduction

The linear probing hash table [33, 45, 58, 77, 80, 87, 90, 127, 128, 130, 136, 141] is among the most fundamental data structures to computer science. The hash table takes the form of an array of some size n, where each ***slot*** of the array either contains an element or is empty (i.e., is a ***free slot***). To insert a new element u, the data structure computes a hash $h(u) \in [n]$, and places u into the first free slot out of the sequence $h(u),\ h(u)+1,\ h(u)+2,\ \ldots$ (modulo n). Likewise, a query for u simply scans through the slots, terminating when it finds either u or a free slot.

There are two ways to implement deletions: immediate compactions and tombstones. An immediate compaction rearranges the neighborhood of elements around a deletion to make it as though the element was never there [68, 75]. Tombstones, the approach we default to in this paper, replaces the deleted element with a ***tombstone*** [75, 119]. Tombstones interact asymmetrically with queries and insertions: queries treat a tombstone as being a value that does not match the query, whereas insertions treat the tombstone as a free slot. In order to prevent slowdown from an accumulation of tombstones, the table is occasionally rebuilt in order to clear them out.

The appeal of linear probing. Linear probing was discovered in 1954 by IBM researchers Gene Amdahl, Elaine McGraw, and Arthur Samuel, who developed the hash table design while writing an assembly program for the IBM 701 (see discussion in [73, 75]). The discovery came just a year after fellow IBM researcher Hans Peter Luhn introduced chained hashing (also while working on the IBM 701) [75].[1] Linear probing shared the relative simplicity of chaining, while offering improved space utilization by avoiding the use of pointers.

The key property that makes linear probing appealing, however, is its data locality: each operation only needs to access one localized region of memory. In 1954, this meant that queries could often be completed at the cost of accessing only a single drum track [73, 119]. In modern systems, it means that queries can often be completed in just a single cache miss [124].[2] The result is that, over the decades, even as computer architectures have changed and as the study of hash tables has evolved into one of the richest and most studied areas of algorithms (see related work in Section 9), linear probing has persisted as one of the best performing hash tables in practice [124].

The drawback: primary clustering. Unfortunately, the data locality of linear probing comes with a major drawback known as ***primary clustering*** [73, 76]. Consider the setting in which one fills a linear-probing hash table up to a ***load factor*** of $1 - 1/x$ (i.e., there are $(1 - 1/x)n$ elements) and then performs one more insertion. Intuitively, one might expect the insertion to take time $\Theta(x)$, since one in every x slots are free. As Knuth [73] discovered in 1963[3], however, the insertion actually runs much slower, taking expected time $\Theta(x^2)$.

The reason for these slow insertions is that elements in the hash table have a tendency to cluster together into long runs; this is known as primary clustering. Primary clustering is often described as a "winner keeps winning" phenomenon, in which the longer a run gets, the more likely it is to accrue additional elements; see e.g., [33, 45, 77, 80, 97, 129, 136, 146]. As Knuth discusses in [75], however, winner-keeps-winning is not the main cause of primary clustering.[4] The true culprit is the globbing together of runs: a single insertion can connect together two already long runs into a new run that is substantially longer.

[1] Interestingly, Luhn's discussion of chaining may be the first known use of linked lists [74, 75]. Linked lists are often misattributed [145] as having been invented by Newell, Shaw, and Simon [102, 103] during the RAND Corporation's development of the IPL-2 programming language in 1956 (see discussion in [74]).

[2] Moreover, even when multiple cache misses occur, since those misses are on adjacent cache lines, hardware prefetching can mitigate the cost of the subsequent misses. Both the Masstree [88] and the Abseil B-tree [1] treat the effective cache-line size as 256 bytes even though the hardware cache lines are 64 bytes.

[3] Knuth's result [73] was never formally published, and was subsequently rediscovered by Konheim and Weiss in 1966 [76].

[4] This can be easily seen by the following thought experiment. Consider the length ℓ of the run at a given position in the hash table, and then consider a sequence of $(1 - 1/x)n$ insertions where we model each insertion as having probability $(1 + \ell)/n$ of increasing the length of the run by 1 (here ℓ is the length of the run at the time of the insertion). The expected length ℓ of the run at the end of the insertions won't be $\Theta(x^2)$. In fact, rather than being an asymptotic function of x, it will simply be $\Theta(1)$.

Interestingly, primary clustering is an asymmetric phenomenon, affecting insertions but not queries. Knuth showed that if a query is performed on a random element in the table, then the expected time taken is $\Theta(x)$ [73]. This can be made true for all queries (including negative queries) by implementing a simple additional optimization: rather than placing elements at the end of a run on each insertion, order the elements within each run by their hashes. This technique, known as ***ordered linear probing*** [4, 29], allows a query to terminate as soon as it reaches an element whose hash is sufficiently large.[5] Whereas insertions must traverse the entire run, queries need not.

Knuth's result predates the widespread use of asymptotic notation in algorithm analysis. Thus Knuth derives not only the asymptotic formula of $\Theta(x^2)$ for expected insertion time, but also a formula of

$$\frac{1}{2}\left(1+\frac{1}{x^2}\right), \tag{1}$$

which is exact up to low order terms. There has also been a great deal of follow-up work determining detailed tail bounds and other generalizations of Knuth's result; see e.g., [51, 66, 76, 91, 138–140].

The practical and cultural effects of primary clustering. Primary clustering is taught extensively in both theoretical and practical courses [10, 35, 37, 38, 47, 50, 59, 71, 92, 98, 126, 131]. Many textbooks not only teach primary clustering [33, 45, 58, 77, 80, 87, 90, 127, 128, 130, 136, 141], but also teach the full formula (1) for insertion time [45, 77, 80, 87, 127, 128, 130, 136, 141].[6] (For a more detailed summary of how courses and textbooks teach linear probing and primary clustering, see Appendix A.)

Because (1) is exact, it is viewed as representing the full picture of how linear probing behaves at high load factors. One consequence is that there has been little empirical work on analyzing the asymptotics of real-world linear probing at high load factors. (And for good reason, what would be the point of verifying what is already known?) As Sedgewick observed in 1990 [128], it is not even clear that the classic formula has been empirically verified at high load factors.

A common recommendation [10, 33, 35, 45, 50, 59, 71, 131, 141, 144] is that, in order to avoid primary clustering, one should use ***quadratic probing*** [62, 89, 147] instead. Whereas linear probing places each element x in the first available position out of the sequence $h(x), h(x)+1, h(x)+2, h(x)+3, \ldots$, quadratic probing uses the first available position out of a quadratic sequence such as $h(x), h(x)+1, h(x)+4, h(x)+9, \ldots, h(x)+k^2, \ldots$ or $h(x), h(x)+1, h(x)+3, \ldots, h(x)+k(k-1)/2, \ldots$. By using a more spread out sequence of probes, quadratic probing seems to eliminate primary clustering in practice. In doing so, quadratic probing also compromises the most attractive trait of linear probing, its data locality. This tradeoff is typically viewed as unfortunate but necessary.

The dangers of primary clustering (and the advice of using quadratic probing as a solution) have been taught to generations of computer scientists over roughly six decades. The folklore advice has shaped some of the most widely used hash tables in production, including the high-performance hash tables authored by both Google [1] and Facebook [25].[7] The consequence is that primary clustering—along with the design compromises made to avoid it—has a first-order impact on the performance of hash tables used by millions of users every day.

This paper: why primary clustering isn't a foregone conclusion. In this paper, we reveal that primary clustering is not the fixed and universal phenomenon that it is reputed to be. When implementing linear probing, there is a small set of design decisions that are typically treated as implementation-level engineering choices. We show that these decisions actually have a remarkable effect on performance: even if a workload operates continuously at a load factor of $1-\Theta(1/x)$, if the design decisions are made correctly, then the expected amortized cost per insertion

[5] Contemporary works sometimes also refer to this as ***Robin hood hashing***, but as noted in [29], Robin hood hashing is actually a generalization of ordered linear probing to other open-addressing schemes such as double hashing, uniform probing, etc.

[6] Some books also go through worked examples or tables of (1) to give intuition for how badly linear probing scales, e.g., [80,128,130,136].

[7] Both hash tables are implemented using variants of quadratic probing.

can be decreased all the way to $\tilde{O}(x)$. Our results come with actionable lessons for practitioners, indicating that implementation-level decisions can have unintuitive asymptotic consequences on performance.[8]

We also present a new variant of linear probing, which we call ***graveyard hashing***, that completely eliminates primary clustering on any sequence of operations: if, when an operation is performed, the current load factor is $1 - 1/x$ for some x, then the expected cost of the operation is $O(x)$.

Thus we achieve the data locality of traditional linear probing without any of the disadvantages of primary clustering. One corollary is that, in the external-memory model with a data blocks of size B, graveyard hashing offers the following remarkably strong guarantee: at any load factor $1 - 1/x$ satisfying $x = o(B)$, graveyard hashing achieves $1 + o(1)$ expected block transfers per operation. In contrast, past external-memory hash tables have only been able to offer a $1 + o(1)$ guarantee when the block size B is at least $\Omega(x^2)$ [67].

What the classical analysis misses. Classically, the analysis of linear probing considers the costs of insertions in an insertion-only workload. Of course, the fact that the final insertion takes expected time $\Theta(x^2)$ doesn't mean that all of the insertions do; most of the insertions are performed at much lower load factors, and the average cost is only $\Theta(x)$.

The more pressing concern is what happens for workloads that operate continuously at high load factors, for example, the workload in which a user first fills the table to a load factor of $1 - 1/x$, and then alternates between insertions and deletions indefinitely. Now almost all of the insertions are performed at a high load factor. Conventional wisdom has it that these insertions must therefore all incur the wrath of primary clustering.

This conventional wisdom misses an important point, however, which is that the tombstones created by deletions actually substantially change the combinatorial structure of the hash table. Whereas insertions add elements at the ends of runs, deletions tend to place tombstones in the middles of runs. If implemented correctly, then the anti-clustering effects of deletions actually outpace the clustering effects of insertions.

We call this new phenomenon ***primary anti-clustering***. The effect is so powerful that, as we shall see, it is even worthwhile to simulate deletions in insertion-only workloads by prophylactically adding tombstones.

Our results flip the narrative surrounding deletions in hash tables: whereas past work on analyzing tombstones [7,68] has focused on showing that tombstones do not *degrade* performance in various open-addressing-based hash tables, we argue that tombstones actually *help* performance. By harnessing the power of tombstones in the right way, we can rewrite the asymptotic landscape of linear probing.

1.1 Results

We begin by giving nearly tight bounds on the amortized performance of ordered linear probing.

There are two design decisions that affect performance: (1) the use of tombstones (which we assume by default) and (2) the frequency with which the hash table is rebuilt and the tombstones are cleared out. Thus, in addition to the size parameter n and the load-factor parameter x, our analysis uses a ***rebuild window size*** parameter R, which is the number of insertions that occur between rebuilds.

The parameter R is classically set to be $n/(2x)$, since this means that the number of tombstone cannot affect the asymptotic load factor. Each rebuild can be implemented in time $\Theta(n)$, so the rebuilds only contribute amortized $\Theta(x)$ time per insertion, which is a low-order term.

A subquadratic analysis of linear-probing insertions. Our first result considers the classical setting $R = n/(2x)$ and analyzes a ***hovering workload***, i.e., an alternating sequence of inserts/deletes at a load factor of $1 - 1/x$.

We prove that the expected amortized cost of each insertion is $\tilde{O}(x^{1.5})$. This is tight, which we establish with a lower bound of $\Omega(x^{1.5}\sqrt{\lg \lg x})$. A surprising takeaway is that the answer is both $\tilde{O}(x^{1.5})$ and $\omega(x^{1.5})$.

[8] The right thing to do may even be the *opposite* of what the literature recommends [7,68].

This first result is already substantially faster than the classical $\Theta(x^2)$ bound, but it still has several weaknesses. The first (and most obvious) weakness is that we are still not achieving the ideal bound of $\tilde{O}(x)$. The second weakness is that, although the result applies to hovering workloads, it doesn't generalize to *arbitrary* workloads, as can be seen with the following pathological example: consider a workload in which every rebuild window consists of $R-1$ insertions followed by R deletions followed by 1 insertion. The first $R-1$ insertions in each rebuild window cannot benefit at all from tombstones and thus necessarily incur $\Theta(x^2)$ expected time each.

It turns out that both of these weaknesses can be removed if we simply use a larger rebuild window size R. Intuitively, the larger the R, the more time there is for tombstones to accumulate and the better the insertions perform. On the other hand, tombstone accumulation is *precisely* the reason that R is classically set to be small, since it breaks the classical analysis and potentially tanks the performance of queries.

We show that the sweet spot is to set $R = n/\operatorname{polylog}(x)$. Here, the expected amortized cost per insertion drops all the way to $\tilde{O}(x)$, while queries continue to take expected time $O(x)$. Once again, and somewhat surprisingly, the low-order factors are an artifact of reality rather than merely the analysis: no matter the value of R used, either the average insertion cost or the average query cost must be $\omega(x)$.

The bound of $\tilde{\Theta}(x)$ holds not only for hovering workloads, but also for *any workload* that maintains a load factor of at most $1-1/x$. Note that here we are analyzing a table in which the capacity n is fixed, and the load factor is permitted to vary over time. It's interesting to see about how the pathological case described above is avoided here. Because the rebuild window is so large, the only way there can be a long series of insertions without deletions is if most of them are performed at low load factors, meaning that they are not slow after all.

A surprising lesson: linear probing is already faster than we thought. The core lesson of our results is that linear probing is far less affected by primary clustering than the classical analysis would seem to suggest. Although the classic $\Theta(x^2)$ bound is mathematically correct, it does not accurately represent the amortized cost of insertions at high load factors. This suggests that conclusions that are taught in courses and textbooks, namely that linear probing scales poorly to high load factors, and that alternatives with less data locality such as quadratic probing should be used in its place, stem in part from an incomplete understanding of linear probing and warrant revisiting.

The second lesson is that small implementation decisions can substantially change performance. From a software engineering perspective, our results suggest two simple optimizations (the use of tombstones and the use of large rebuild windows) that should be considered in any implementation of linear probing.

Interestingly, tombstones (and even relatively large rebuild windows) are already present in some hash tables. Thus, one interpretation of primary anti-clustering is as a phenomenon that, to some degree, already occurs around us, but that until now has gone apparently unnoticed.

Graveyard hashing: a variant of linear probing with ideal performance. Our final result is a new version of linear probing, which we call ***graveyard hashing***, that fully eliminates primary clustering on any sequence of operations. The key insight is that, by artificially inserting extra tombstones (that are not created by deletions), we can ensure that every insertion has good expected behavior.

Insertions, deletions, and queries are performed in exactly the same way as for standard ordered linear probing. The difference is in how we implement rebuilds. In addition to cleaning out tombstones, rebuilds are now also responsible for inserting $\Theta(n/x)$ *new tombstones* evenly spread across the key space.

Graveyard hashing adapts dynamically to the current load factor of the table, performing a rebuild every time that x changes by a constant factor. The running time of each operation is a function of whatever the load factor is at the time of the operation. If a query/insert/delete occurs at a load factor of $1-1/x$, then it takes expected time $O(x)$ (even in insertion-only workloads). Graveyard hashing can also be implemented to resize dynamically, so that it is always at some target load factor of $1-\Theta(1/x)$.

Coming full circle: improved external-memory hashing. As we mentioned at the outset, one of the big advan-

tages of linear probing is its data locality (e.g., good cache or I/O performance).

Data locality is formalized via the external-memory model, first introduced by Aggarwal and Vitter in 1988 [2]: a two-level memory hierarchy is comprised of a small, fast ***internal memory*** and a large, slow ***external memory***; blocks can only be read and modified when they are in internal memory, and the act of copying a block from external memory into internal memory is referred to as a ***block transfer***. The model has two parameters, the number B of of records that fit into each block and the number M of records that fit in internal memory. Performance is measured by the number of block transfers that a given algorithm or data structure incurs. This model can be used to capture an algorithm's I/O performance (internal memory is RAM, external memory is disk, and block transfers are I/Os) or cache performance (internal memory is cache, external memory is RAM, and block transfers are cache misses).

Ideally, a hash table incurs only amortized expected $1+o(1)$ block transfers per operation, even when supporting a high load factor $1-1/x$.[9] We call the problem of achieving these guarantees the ***space-efficient external-memory hashing problem***. Standard linear probing is a solution when either x is a (small) constant, or the block size B is very large ($B = \omega(x^2)$), but otherwise, due to primary clustering, it is not [112].

For $B \neq \omega(x^2)$, the state of the art for space-efficient external-memory hashing is due to Jensen and Pagh [67]. They give an elegant construction showing that, if the block size B is $\Theta(x^2)$, then it is possible to achieve amortized expected $1 + O(1/x)$ block transfers per operation, while maintaining a load factor of $1 - O(1/x)$. (In contrast, standard linear probing requires $B = \Omega(x^3)$ to achieve the same $1 + O(1/x)$ result.) However, if $B = o(x^2)$, then no solutions to the problem are known.

Graveyard hashing enables linear probing to be used directly as a solution to the space-efficient external-memory hashing problem, matching Jensen and Pagh's bound for $B = \Theta(x^2)$, and offering an analogous guarantee for arbitrary block sizes $B > \omega(x)$. If $B = \Theta(xk)$ for some $k > 1$, then the amortized expected cost of each operation is $1 + O(1/k)$ block transfers. This means that, even if the block size B is only *slightly* larger than the load factor parameter x, we still get $1 + o(1)$ block transfers per operation.

Additionally, graveyard hashing is cache oblivious [54, 112], meaning that the block size B need not be known by the data structure. Consequently, if a system has a multi-level hierarchy of caches, each of which may have a different set of parameters B and M, then the guarantee above applies to every level of cache hierarchy.

2 Notation and Conventions

We say that an event occurs with probability $1 - 1/\operatorname{poly}(j)$ for some parameter j if, for any positive constant c, the event occurs with probability $1 - O(1/j^c)$.[10] Throughout the paper, we use standard interval notation, where $[m]$ means $\{1, 2, \ldots, m\}$, $[i, j]$ means $\{i, i+1, \ldots, j\}$, $(i, j]$ means $\{i+1, i+2, \ldots, j\}$, etc.

When discussing an ordered linear probing hash table, we use n to denote the number of ***slots*** (i.e., ***positions***). We use $1 - 1/x$ to refer to the ***load factor*** (i.e., the fraction of slots that are taken by elements), and R to refer to the ***rebuild-window size*** (i.e., the number of insertions that must occur before a rebuild is performed). We use S to denote the sequence of operations being performed, and we refer S as the ***workload***.

The operations on the hash table make use of a ***hash function*** $h : U \to [n]$, where U is the universe of possible ***keys*** (also known as ***records*** or ***elements***). We shall assume that h is uniform and fully independent, but as we discuss in later sections, our results also hold for natural families of hash functions such as tabulation hashing (and, for the analysis of graveyard hash tables, also 5-independent hashing). We can also refer to the ***hash*** $h(u)$ of either a tombstone u (i.e., the hash of the element whose deletion created u) or of an operation u (i.e., the hash of the element on which the operation takes place).

[9] Many hash tables that are otherwise very appealing perform poorly on this front. For example, if one uses cuckoo hashing, then negative queries require ≥ 2 block transfers, and insertions at high load factor require $\omega(1)$ block transfers [43, 52].

[10] The constants used to define the event may depend on c.

Each slot in the hash table can either contain a key, be empty (i.e., a free slot), or contain a tombstone. Any maximal contiguous sequence of non-empty slots forms a ***run***. With ordered linear probing, the keys/tombstones in each run are always stored in order of their hash.

Our analysis will often discuss ***sub-intervals*** $I = [i, j] \subseteq [n]$ of the slots in the hash table. We say that an element u ***hashes to*** I if $h(u) \in I$ (but this does not necessarily mean that u resides in one of the slots I). We say that an interval I is ***saturated*** if it is a subset of a run.

Formally, operations in an ordered linear probing hash table are implemented as follows. A query for a key u examines positions $h(u), h(u)+1, \ldots$ until it either finds u (in which case the query returns true), finds an element with hash greater than $h(u)$ (in which case the query returns false), or finds a free slot (in which case the query also returns false). A deletion of a key u simply performs a query to find the key, and then replaces it with a tombstone. Finally, an insertion of a key u examines positions $h(u), h(u)+1, \ldots$ until it finds the position j where u belongs in the run; it then inserts u into that position, and shifts the elements in positions $j, j+1, j+2, \ldots$ each to the right by one until finding either a tombstone or a free slot. We say that the insertion ***makes use*** of that tombstone/free slot. Finally, rebuilds are performed every R insertions, and a rebuild simply restructures the table to remove all tombstones. Throughout the paper, n is fixed and does not get changed during rebuilds, although when we describe graveyard hashing (Section 8), we also give a version that dynamically resizes n.

There are two ways to handle overflow off the end of the hash table. One option is to wrap around, meaning that we treat 1 as being the position that comes after n; the other is to extend the table by $o(n)$ (i.e., it ends in slot $n + o(n)$), so that operations never fall off the end of the table. Both solutions are compatible with all of our results. For concreteness, we assume that the wrap-around solution is used, but to simplify discussion, we treat the slots that we are analyzing as being sufficiently towards the middle of the table that we can use the $<$-operator to compare slots.

3 Technical Overview

In this section, we give a technical overview of our analysis of ordered linear probing; we defer our analysis graveyard hashing to Section 8 in the main body of the paper. We begin by describing the intuition behind Knuth's classic $\Theta(x^2)$ bound. We then turn our attention to sequences of operations that contain deletions, and show that the tombstones left behind by those deletions have a *primary-anti-clustering effect*, that is, they have a tendency to speed up future insertions. One of the interesting components of the analysis is that we perform a series of problem transformations, taking us from the question of how to analyze ordered linear probing to a seemingly very different question involving the combinatorics of monotone paths on a grid. By applying geometric arguments to the latter, we end up being able to achieve nearly tight bounds for the former.

3.1 Understanding the classic bounds: a tale of standard deviations

Suppose we fill an ordered linear-probing hash table from empty up to a load factor of $1-1/x$. Knuth [73] famously showed that the final insertion in this procedure takes expected time $\Theta(x^2)$. As discussed in the introduction, the fact that the insertion takes time $\omega(x)$ can be attributed to primary clustering.

But why does the running time end up being $\Theta(x^2)$ specifically? This turns out to be a result of how standard deviations work. Consider an interval I of x^2 slots in the hash table. The expected number of items that hash into I is $(1-1/x)\,x^2 = |I| - x$. On the other hand, the standard deviation for the number of such items is $\Theta(x)$. It follows that, with probability $\Omega(1)$, the number of items that hash into I is $\Omega(x)$ greater than $|I|$. If we then consider the interval I' consisting of the x^2 slots that follow I, then this interval I' must handle not only the items that hash into it, but also the overflow elements from I. The result is that, with probability $\Omega(1)$, the interval I' is fully saturated and forms a run of length x^2.

The above argument stops working if we consider intervals of size $\omega(x^2)$, because the standard deviation on the number of items that hash into I stops being large enough to overflow the interval. The result is that runs of length $\Theta(x^2)$ are relatively common, but that longer runs are not. This is why the expected running time of the insertion performed at load factor $1 - 1/x$ is $\Theta(x^2)$.

Now suppose that, after reaching a load factor of $1 - 1/x$, we perform a query in our ordered linear-probing hash table. Unlike an insertion, which takes expected time $\Theta(x^2)$, the query takes expected time $\Theta(x)$. We can again see this by looking at standard deviations.

If the query hashes to some position j and takes time t, then there must be at least t elements u that have hashes $h(u) \leq j$ but that reside in positions j or larger. Hence, there is some sub-interval $I = [j_0, j]$ of the hash table (ending in position j) that has overflowed by at least $t - 1$ elements, that is, the number of elements that hash to I is at least $|I| + t - 1$. As before, the interval that matters most ends up being the one of size $\Theta(x^2)$, and the amount by which it overflows in expectation is proportional to the standard deviation $\Theta(x)$ of the number of items that hash into the interval.

Thus, the running times of both insertions and queries are consequences of the same two facts: that (a) for any interval I of size x^2, there is probability $\Omega(1)$ that the interval overflows by $\Theta(x)$ elements; and that (b) a given interval of size $\omega(x^2)$ most likely doesn't overflow at all. The only difference is that the running time of an insertion is proportional to the *size* of the interval that overflows (i.e., $\Theta(x^2)$), but the running time of a query is proportional to the *amount* by which the interval overflows (i.e., $\Theta(x)$).

3.2 Analyzing primary anti-clustering with small rebuild windows

In this subsection, we consider a hovering workload, that is a sequence of operations that alternates between insertions and deletions on a table with load factor $1 - 1/x$, and we set the size of the rebuild window to be $R = n/(2x)$ (i.e., the value that it is classically set to). Our task is to consider a sequence of $R = n/(2x)$ insertion/deletion pairs between two rebuilds, and to analyze the amortized running times.

Analyzing displacement instead of running time. Define the ***peak*** p_u of an insertion u to be either the hash of the tombstone that the insertion uses (if the insertion makes use of a tombstone) or the position of the free slot that the insertion uses (if the insertion makes use of a free slot). Define the ***displacement*** d_u of an insertion to be $d_u = p_u - h(u)$.

One of the subtleties of how displacement is defined is that, if an insertion u uses a tombstone v, then the displacement measures the difference between $h(u)$ and the *hash* $h(v)$, rather than the difference between $h(u)$ and the *position* of v. This ends up being important for how displacement is used in the analysis[11], but it also means that the displacement of an insertion can potentially be substantially smaller than the running time. For example, if the insertion hashes to position 7 and makes use of a tombstone with hash 13 that resides in position 54, then the displacement is only $13 - 7 = 6$ but the running time is proportional to $54 - 7 = 47$.

Although we skip the proof for now (see Lemma 16), it turns out that one can bound the expected difference between displacement and running time by $O(x)$. Thus, even though displacement is not always the same as running time, any bound on average displacement also results in a bound on average running time.

Relating displacement to crossing number. Rather than analyzing the displacement of each *individual* insertion, we bound the average displacement over all R insertions in the rebuild window by relating the displacements to another set of quantities that we call the ***crossing numbers*** $\{c_j\}_{j \in [n]}$. The crossing number c_j counts the number

[11] The reason for this is actually very simple. Whenever a deletion v is performed, the value of $h(v)$ is fixed (it depends only on the hash function) but the position of v is not (it depends on the other elements in the table, and will change over time). Thus, it is cleaner to measure the deletion's effect on future insertions in terms of $h(v)$ (the thing that is fixed) rather than v's position (the thing that is not).

of insertions u in the rebuild window that have a hash $h(u) < j$ but that have a peak $p_u \geq j$ (we consider even the insertions that are subsequently deleted). Each insertion u increments d_u different crossing numbers c_j. Thus

$$\sum_u d_u = \sum_{j \in [n]} c_j.$$

Because we are analyzing the insertions in a rebuild window of size R, the summation on the left side has $R = \Theta(n/x)$ terms, while the summation on the right side has n terms. Thus, if we consider a random insertion u and a random position j, then

$$\mathbb{E}[d_u] = \Theta(x\mathbb{E}[c_j]).$$

If our goal is to establish that the average insertion takes time $o(x^2)$, then it suffices instead to show that the average crossing number c_j is $o(x)$.

Notice that the ratio between $\mathbb{E}[d_u]$ and $\mathbb{E}[c_j]$ is a function of the rebuild window size R; this will come into play later when we consider larger rebuild windows.

Capturing the dependencies between past insertions/deletions. What makes the analysis of a given crossing number c_j interesting is the way in which insertions and deletions interact over time. If an insertion u has hash $h(u) < j$, and there is a tombstone v with hash $h(v) \in [h(u), j)$, then u can make use of the tombstone and avoid contributing to c_j. But, in order to determine whether a given tombstone v is present during the insertion u, we must know whether any past insertions have already used v. That, in turn, depends on which tombstones were present during past insertions, resulting in a chain of dependencies between operations over time.

One of the insights in this paper is that the interactions between insertions and deletions over time can be reinterpreted as an elegant combinatorial problem about paths on a two-dimensional grid. We now give the transformation.

The geometry of crossing numbers. For the sake of analysis, define Z to be the state that our table would be in if we performed only the insertions in the rebuild window and not the deletions.

Since $R = n/(2x)$, the load factor of Z is at most $1 - 1/x + R/n = 1 - 1/(2x)$, which by the classic analysis of linear probing means that the expected distance from any position to the next (and previous) free slot is $O(x^2)$.

Now consider some crossing number c_j. Let j' be the position of the closest free slot to the left of j in Z, that is, the largest $j' < j$ such that position j' is a free slot in Z. Then the only insertions/deletions that we need to consider when analyzing c_j are those that hash into the interval $I = [j' + 1, j)$.

We know from our analysis of Z that $\mathbb{E}[|I|] = O(x^2)$. Although $|I|$ is a random variable with mean $\Theta(x^2)$, to simplify our discussion in this section, we shall treat $|I|$ as simply deterministically equaling x^2. We also treat I as containing no free slots (even at the beginning of the time window being considered). In particular, we know from the classical analysis of linear probing that any interval I of size x^2 has probability $\Omega(1)$ of containing no free slots, so there is no point in trying to make use of potential free slots in I for our analysis.

We can visualize the insertions and deletions that hash into I by plotting them in a two-dimensional grid, as in Figure 1a. The vertical axis represents time flowing up from 1 to $2R$, and the horizontal axis represents the hash locations in the interval I. We draw a blue dot in position (i, t) if the t-th operation is an insertion with hash $i \in I$, and we draw a red dot in position (i, t) if the t-th operation is a deletion with hash $i \in I$. (Note that most operations do not hash to I and thus do not result in any dot.)

In order for a given insertion u to be able to make use of a given deletion v's tombstone, it must be that (a) $h(u) \leq h(v)$;[12] that (b) u occurs temporally after v; and that (c) v's tombstone is not used by any other insertion

[12]Recall that ordered linear probing only ever moves elements to the right over time, meaning that a given insertion u will only use a tombstone if that tombstone has hash at least $h(u)$.

temporally before u. The first two criteria (a) and (b) tell us that for a given insertion (i.e., blue dot) in the grid, the insertion can only make use of tombstones from deletions (i.e., red dots) that are below it and to its right.

Define the set of ***monotone paths*** through the grid to be the set of paths that go from the bottom left to the top right of the grid, and that never travel downward or leftward. Define the ***blue-red deviation*** of such a path to be the number of blue dots below the path minus the number of red dots below the path (see Figure 1a for an example).

What do these monotone paths have to do with the crossing number c_j? Monotone paths with large blue-red deviations serve as witnesses for the crossing number c_j also being large. Suppose that there is a monotone path γ with blue-red deviation $r > 0$. Since we assume that I is initially saturated, each of the insertions (i.e., blue dots) below γ must either make use of the tombstone for a deletion (i.e., red dot) that is also below γ or contribute 1 to c_j. Since there are r more blue dots than red dots below γ, it follows that $c_j \geq r$.

In fact, this relationship goes in both directions (although the other direction requires a bit more work; see Lemma 9). If the crossing number c_j takes some value r, then there must also exist some monotone path with blue-red deviation at least r. The result is that, if we wish to prove either upper or lower bounds on $\mathbb{E}[c_j]$, it suffices to instead prove bounds on the largest blue-red deviation of any monotone path through the grid.

Formalizing the blue-red deviation problem. Let us take a moment to digest the combinatorial problem that we have reached, since on the face of things it is quite different from the problem that we started at.

The expected number of blue dots (and also of red dots) in our grid is $R \cdot |I|/n = \Theta(x)$. If we break the grid into $\sqrt{x}$ rows and $\sqrt{x}$ columns (see Figure 1b for an example), then each cell of the broken-down grid expects to contain $\Theta(1)$ blue and red dots. To simplify our discussion here, think of each cell as independently containing a Poisson random variable $\text{Pois}(1)$ number of blue dots and a Poisson random variable $\text{Pois}(1)$ number of red dots.[13]

Furthermore, rather than considering all monotone paths through the grid, we can restrict ourselves exclusively to the paths that stay on the row and column lines that we have drawn (for an example, see Figure 1b). With high probability in x, this restriction changes the maximum blue-red deviation of any path by at most $\tilde{O}(\sqrt{x})$.

In summary, we have a $\sqrt{x} \times \sqrt{x}$ grid where each cell of the grid contains a Poisson random variable $\text{Pois}(1)$ number of blue points (resp. red points). Whereas our original grid was much taller than it was wide (its height was $2R$ and its width was x^2), our new grid is a $\sqrt{x} \times \sqrt{x}$ square. There are $\binom{2\sqrt{x}}{\sqrt{x}} = \exp(\Omega(x))$ monotone paths γ through the grid, and we wish to prove bounds on the maximum blue-red deviation D achieved by any such path.

Gaining intuition: how blue-red deviations behave. To gain intuition, let us start by considering the trivial path γ that contains the entire grid beneath it. The expected number of blue dots (as well as red dots) beneath γ is $\Theta(x)$, and the standard deviation on the number of blue dots (as well as red dots) is thus $\Theta(\sqrt{x})$. With probability $\Omega(1)$, there are $\Omega(\sqrt{x})$ more blue dots in the grid than red dots, which results in a blue-red deviation of $\Omega(\sqrt{x})$ for γ.

Of course, that's just the blue-red deviation of a *single fixed path*. What should we expect the *maximum* blue-red deviation D over all paths to be? On one hand, there are exponentially many paths that we must consider, but on the other hand, the blue-red deviations of the paths are closely correlated to one another. The result, it turns out, is that D ends up being an $x^{o(1)}$ factor larger than $\sqrt{x}$.

We will show that, with probability $1 - o(1)$, the maximum blue-red deviation D is between $\Omega(\sqrt{x \lg \lg x})$ and $\tilde{O}(\sqrt{x})$. If we backtrack to our original problem (i.e., we relate the blue-red deviations to the crossing numbers, the crossing numbers to the displacements, and the displacements to the running times), we get that the amortized cost of insertions is between $\Omega(x^{1.5}\sqrt{\lg \lg x})$ and $\tilde{O}(x^{1.5})$.

An upper bound of $\tilde{O}(\sqrt{x})$ on the maximum blue-red deviation D. The first step in bounding D is to prove

[13] This allows for us to ignore two minor issues in our discussion here: (1) the fact that a single key can potentially be inserted, deleted, and reinserted, resulting in blue and red dots whose horizontal coordinates are deterministically equal; and (2) the fact that the numbers of blue/red dots in each cell are actually very slightly negatively correlated.

a general result about decompositions of monotone paths. Let $k = 4\sqrt{x}$ be the perimeter of the grid. We claim that for any monotone path γ through the grid, it is always possible to decompose the area under γ into disjoint rectangles $R_1, R_2, \ldots$ such that the sum of the perimeters of the rectangles is at most $k \lg k$.

Such a decomposition can be constructed recursively as follows: (1) find the point q halfway along the path, and drop a rectangle from q to the bottom-right-most point in the grid; (2) then recursively construct a rectangular decomposition for the portion of the path prior to q, and recursively construct a rectangular decomposition for the portion of the path after q. (See Figure 1c for an example.)

The recursive decomposition is designed so that, in the i-th level of recursion, each recursive subproblem takes place on a grid with perimeter exactly $k/2^i$. Since there are at most 2^i subproblems in each level of recursion, each of which contributes a rectangle with perimeter at most $k/2^i$, the sum of all the rectangle perimeters over all levels of recursion is at most $k \lg k$.

The next step in the analysis is to consider the maximum amount that any given rectangle can contribute to the blue-red deviation of a path. If a given rectangle has area a, then the expected number of blue/red dots in the rectangle is $\Theta(a)$, and with high probability in x, the rectangle as a whole has blue-red deviation $O(\sqrt{a} \lg x)$. This, in turn, means that if a rectangle has perimeter p, then with high probability in x, it has blue-red deviation at most $O(p \lg x)$. Finally, since there are only $O(x^2)$ possible rectangles in the entire grid, and this property holds for each of them with probability $1 - 1/\operatorname{poly}(x)$, the property also holds simultaneously for all of them with probability $1 - 1/\operatorname{poly}(x)$.

Putting the pieces together, we know that every path γ has a rectangular decomposition such that the sum of the rectangle perimeters is $O(\sqrt{x} \lg x)$. We further know that, if a rectangle in the decomposition has perimeter p, then it contributes at most $O(p \lg x)$ to the blue-red deviation of γ. It follows that the total blue-red deviation of any path γ is at most $O(\sqrt{x} \lg^2 x)$.

A lower bound of $\Omega(\sqrt{x \lg \lg x})$ on the maximum blue-red deviation D. Now we turn our attention to proving a lower bound on D. We wish to find a monotone path γ whose blue-red deviation is $\Omega(x^{1.5}\sqrt{\lg \lg x})$.

Call a $j \times j$ square within the grid ***high-value*** if it has blue-red deviation at least $\Omega(j\sqrt{\lg \lg x})$. The definition is designed so that every square has probability at least $1/\sqrt{\lg x}$ of being high-value.

We construct a monotone path γ recursively as follows. First break the grid into quadrants, and check whether the top left quadrant is a high-value square. If so, then return the trivial path that contains the entire grid below it. Otherwise, recursively construct a path through the bottom-left quadrant, recursively construct a path through the top-right quadrant, and set γ to be the concatenation of the two paths.[14] (For an example, see Figure 1d.)

There are two types of base cases in the recursion. The first type is when a subproblem finds a high-value square; we call this a ***successful base case***. The second type is when a subproblem terminates because it is on a 1×1 grid; we call this a ***failed base case***.

If a successful base case takes place on a sub-grid of width w, then the high-value square that it discovers contributes $\Omega(w\sqrt{\lg \lg x})$ to the total blue-red deviation of γ. In order to establish a lower bound of $\Omega(\sqrt{x \lg \lg x})$ on the blue-red deviation, it therefore suffices to show that the sum of the widths of the successful base cases is $\Omega(\sqrt{x})$.[15]

By construction, the sum of the widths of both the successful base cases and the failed base cases is exactly $\sqrt{x}$. Moreover, each failed base case has width exactly 1. Thus our task reduces to bounding the number of failed base cases by $o(\sqrt{x})$ with probability $1 - o(1)$.

In order for a given failed base case to occur, there is a recursion path of $\Theta(\lg x)$ sub-problems that must all fail to find a high-value square. Each of these failures occurs with probability at most $1 - 1/\sqrt{\lg x}$, so the probability

[14] When we recurse, we do not change the threshold $\Omega(j\sqrt{\lg \lg x})$ that dictates whether a given $j \times j$ square is special; that is, x acts as a global variable setting this threshold.

[15] There is also a large portion of the area underneath γ that is not contained in any of the high-value squares of the subproblems. Technically, we must also ensure that the blue/red dots in this unaccounted-for area do not substantially change the blue-red deviation, but this follows from a straightforward Chernoff bound.

of all of them occurring is

$$\left(1 - 1/\sqrt{\lg x}\right)^{\Theta(\lg x)} = o(1).$$

Since the probability of any given failed base case occurring is $o(1)$, the expected number of failed base cases that occur is $o(\sqrt{x})$. By Markov's inequality, the number of failed base cases is $o(\sqrt{x})$ with probability $1 - o(1)$, as desired.

3.3 Stronger primary anti-clustering with larger rebuild windows

In this section, we consider what happens if a larger rebuild window size $R = n/\operatorname{polylog}(x)$ is used. As discussed in the introduction, this allows for us to improve our amortized insertion time from $\tilde{\Theta}(x^{1.5})$ to $\tilde{\Theta}(x)$, while still achieving average query time $\Theta(x)$.

Remarkably, these bounds hold not just for hovering workloads, but also for arbitrary workloads that stay below a load factor of $1 - 1/x$. To simplify discussion in this section, however, we continue to focus on the hovering case.

There are two main technical challenges that our analysis must overcome. The first challenge is obvious: we must quantify the degree to which tombstones left behind by deletions improve the performance of subsequent insertions. The second challenge is a bit more subtle: in order to support large rebuild-window sizes R, our analysis must be robust to the fact that tombstones can accumulate over time, increasing the effective load factor of the hash table. This latter challenge is further exacerbated by the fact that the choice of which tombstones are in the table at any given moment is a function not only of the sequence of operations being performed, but also of the randomness in the hash table. This means that, even if the cumulative load factor from the elements and tombstones can be bounded (e.g., by $1 - \Theta(1/x)$), we still cannot analyze the tombstones as though they were normal elements; a consequence of this is that we cannot even apply the classic analysis to deduce an $O(x^2)$-time bound for insertions or an $O(x)$-time bound for queries.

Using crossing numbers to rescue the queries. Let us consider the time that it takes to query an element whose hash is j. The time is proportional to the number of elements and tombstones that have hashes smaller than j but that reside in positions j or larger.[16] Call the slots containing these elements ***j-crossed***. Right after a rebuild is performed, the number s of j-crossed slots has expected value $O(x)$. Over the course of the time window between consecutive rebuilds, however, the quantity s gradually increases.

Fortunately, the amount by which s increases is *precisely* the crossing number c_j. Indeed, c_j gets incremented exactly whenever a formerly non-j-crossed slot becomes j-crossed.

Thus, if we can bound c_j to be small then we hit two birds with one stone: we are able to bound the running times of both queries and insertions.

But how do we rescue the crossing numbers? Large rebuild windows also break the analysis of crossing numbers, however. In the original analysis, we argued that there is most likely some position $j' < j$ satisfying $j - j' = O(x^2)$ such that position j' remains an empty slot throughout the entire rebuild window. This meant that, when considering c_j, we only had to analyze insertions and deletions that hash into the interval $I = [j' + 1, j)$ of size $\Theta(x^2)$.

We can no longer argue that such a j' necessarily exists, however, since the accumulation of tombstones over time might eliminate all of the free slots near position j. Thus we must extend our analysis to consider intervals I of size $\omega(x^2)$.

Fortunately, if we consider any interval I of size at least $x^2 \operatorname{polylog} x$, then we can argue that the interval most likely *initially* contains $\Omega(|I|/x)$ free slots. Define the ***insertion surplus*** of an interval I to be the maximum blue-red deviation of any monotone path through the grid representing I, *minus* the number of free slots initially in I. We prove that the crossing number c_j is exactly equal to the maximum insertion surplus of any interval I of

[16] Technically, we must also consider elements that hash to exactly j but the expected number of such elements is $O(1)$.

the form $[j'+1, j)$. Since large intervals I have a $\Theta(1/x)$-fraction of their slots initially empty, it is very unlikely that they end up determining the crossing number c_j. The result is that we can again focus primarily on intervals of size $O(x^2)$, and perform the analysis of c_j as before.

Putting the pieces together for $R = n/\operatorname{polylog}(x)$. We now analyze the case of $R = n/\operatorname{polylog}(x)$. As before, we use the displacement d_u for an insertion as a proxy for insertion time (although bounding the difference between the two requires a more nuanced argument than before, see Lemmas 13 and 14). We can then relate the displacements to the crossing numbers by

$$\sum_u d_u = \sum_{j \in [n]} c_j.$$

Now, however, both sums consist of between $n/\operatorname{polylog} x$ and n terms. This means that for a random insertion u and a random $j \in [n]$,

$$\mathbb{E}[d_u] \leq \mathbb{E}[c_j \operatorname{polylog}(x)].$$

As before, we can transform the problem of bounding c_j into the problem of bounding the maximum blue-red deviation of any monotone path in a certain grid. The expected number of blue/red dots in the grid is now $x^2/\operatorname{polylog}(x)$ (rather than $\Theta(x)$). Thus our bound on blue-red deviation comes out as $\tilde{O}(x/\operatorname{polylog}(x)) = O(x)$ (rather than $\tilde{O}(\sqrt{x})$). This means that $\mathbb{E}[c_j] = O(x)$, which implies that the expected time taken by any query is also $O(x)$ and that the average time taken by each insertion is $\tilde{O}(x)$.

We can now also see why $n/\operatorname{polylog}(x)$ is the right rebuild window size to use. In particular, if we make R smaller than $n/\operatorname{poly}(x)$, then insertion times suffer, but if we let R get too close to n (or exceed n) then the crossing numbers c_j become $\omega(x)$ (thanks to our lower bound construction on blue-red deviations), and thus queries take time $\omega(x)$. Thus it is impossible to select a value for R that achieves expected time $O(x)$ for all operations, and if we want $O(x)$-time queries, we must make $R = o(n)$.

Analyzing arbitrary workloads. Finally, we generalize these results to *any* sequence of operations that stays below a load factor of $1 - 1/x$. We argue that, within any rebuild window, it is possible to re-organize the operations in such a way that (a) none of the crossing numbers decrease, and (b) the operations consist of a series of insertions, followed by a series of alternating insertions/deletions, followed by a series of deletions (see Proposition 7). The alternating insertions/deletions can be analyzed as above, and because R is large, the crossing-number cost of the initial insertions can be amortized away. The fact that the re-organization of operations does not decrease any crossing numbers ends up being easy to prove using our characterization of crossing numbers in terms of insertion surpluses of intervals.

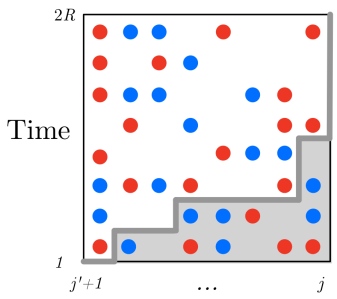
3.4 Graveyard hashing

Can we achieve a bound of $O(x)$ for both query and insertion times? We have already seen that standard ordered linear probing cannot, regardless of the choice of R.

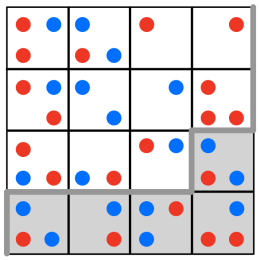
The guarantee can be achieved, however, by a slightly modified version of ordered linear probing that we call graveyard hashing. The basic idea of graveyard hashing is to insert extra tombstones artificially during each rebuild. This extra injection of tombstones ensures that, during the next rebuild window, tombstones for insertions to make use of remaining plentiful at all times. Thus, we are able to put all insertions in a "best-case scenario", guaranteeing that each operation has expected time $O(x)$. For details, see Section 8.

4 Some Basic Balls-and-Bins Lemmas

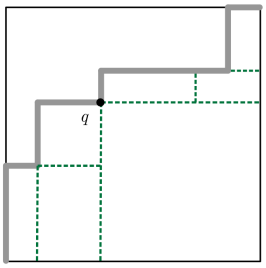
We begin by proving several basic lemmas having to do with balls and bins. What makes these lemmas different from standard balls-and-bins bounds is that they consider all prefixes of a sequence of bins, bounding the probability



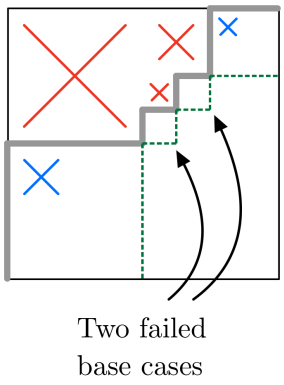
(a) An example of representing insertions/deletions in $I = [j' + 1, j)$ graphically over time. Blue dots are insertions and red dots are deletions. (To simplify this picture, we plot each point in one of eight y-coordinates, but in reality, each dot would have a distinct y-coordinate.) An example monotone path is also given, and the blue-red deviation is $6 - 4 = 2$.



(b) To simplify the problem, we draw a $\sqrt{x} \times \sqrt{x}$ grid, and consider only paths that go along the drawn grid lines. Here $x = 16$.



(c) An example of a recursively constructed rectangular decomposition for a path. The point q used in the top level of recursion is also labeled.



(d) An example of the construction for a path with high blue-red deviation. For each recursive subproblem whose top-left quadrant is not high-value, we place a red X through the quadrant; for each recursive subproblem whose top-left quadrant is high-value, we place a blue X through the quadrant; the only subproblems not to have an X drawn in them are the failed base cases, each of which takes place on a 1×1 grid. There are four base cases, two of which are failed.

Figure 1

of any prefix behaving abnormally. In order to demonstrate how these lemmas relate to linear probing, we will also use them to reprove the classic bounds on the performance of insertions and queries for ordered linear probing.

Throughout the rest of the section, consider the setting in which we place $m = \Theta(n)$ balls randomly into n bins. Let $\mu = n/m$ be the expected number of balls in each bin.

Lemma 1. *Let $x > 1$ and $k \geq 1$. With probability $1 - 2^{-\Omega(k)}$, for every $i \geq x^2 k$, the number of balls in the first i bins is between $(1 - 1/x)i\mu$ and $(1 + 1/x)i\mu$.*

Proof. Let $\ell = x^2 k$. We wish to show that, with probability at least $1 - 2^{-\Omega(k)}$, there is no $i \geq \ell$ such that the first i bins contain either fewer than $(1 - 1/x)i\mu$ balls or more than $(1 + 1/x)i\mu$ balls. We will focus on the more-than-$(1 + 1/x)i\mu$ case, since the fewer-than-$(1 - 1/x)i\mu$ case follows by a symmetric argument.

Let X_i be the indicator random variable for the event that the first i bins contain at least $(1 + 1/x)i\mu$ balls. By a Chernoff bound,

$$\Pr[X_\ell] \leq 2^{-\Omega(k)}.$$

It is tempting to apply a similar Chernoff bound to every $i \geq \ell$, and then to take a union bound; but this would yield a bound of $\Pr[X_i \text{ for some } i \geq \ell] \leq \ell 2^{-\Omega(k)}$ rather than the desired bound of $\Pr[X_i \text{ for some } i \geq \ell] \leq 2^{-\Omega(k)}$. Thus a slightly more delicate approach is needed.

To complete the proof, we prove directly that

$$\Pr[X_i \text{ for some } i \geq \ell] \leq O(\Pr[X_\ell]), \tag{2}$$

which we have already shown to be $2^{-\Omega(k)}$.

Suppose that X_i holds for some $i \geq \ell$ and let j be the largest such i. The number B of balls in the first j bins must satisfy $B \geq (1 + 1/x)j\mu$. Each of these B balls independently has probability ℓ/j of being in one of the first ℓ bins. Conditioning on a given value of B, the number of balls in the first ℓ bins is a binomial random variable with expected value at least $(1 + 1/x)\ell\mu$. With probability $\Omega(1)$ (and using the fact that $(1 + 1/x)\ell\mu = \Omega(1)$), this binomial random variable takes a value greater than or equal to its mean $(1 + 1/x)\ell\mu$, which implies that X_ℓ occurs. Having established (2), the proof is complete. □

Corollary 1. *Consider the largest i such that the first i bins contain at least $(1+1/x)i\mu$ balls. Then $\mathbb{E}[i] \leq O(x^2)$.*

Lemma 2. *Let $x > 1$ and $k \geq 1$. With probability $1 - 2^{-\Omega(k)}$, there does not exist any i such that the first i bins contain at least $(1 + 1/x)i\mu + kx$ balls.*

Proof. Let Y_i be the indicator random variable for the event that the first i bins contain at least $(1 + 1/x)i\mu + kx$ balls. Let $r = x^2 k$. By Lemma 1,

$$\Pr[Y_i \text{ for some } i \geq r] \leq 2^{-\Omega(k)}.$$

Thus it suffices to argue that

$$\Pr[Y_i \text{ for some } i \leq r] \leq 2^{-\Omega(k)}.$$

As in the previous proof, taking a Chernoff bound and then summing over i will not give the result that we are aiming for. Thus a more refined approach is again needed.

Suppose that Y_i occurs for some $i \leq r$, and let j be the largest such i. Let B satisfying $B \geq j\mu + kx$ be the number of balls in the first j bins, and let 2^q be the largest power of two satisfying $2^q \leq j$. Each of the B balls in the first j bins has probability $2^q/j$ of being in the first 2^q bins. Conditioning on a given value of B, we therefore have that the number of balls that land in the first 2^q bins is a binomial random variable with expected value at least $2^q\mu + kx/2$. With probability $\Omega(1)$, this random variable takes a value at least as large as its mean. That is, if we condition on Y_i occurring for some $i \leq r$, then with constant probability there is some power of two $2^q \leq r$ such that at least $2^q\mu + kx/2$ balls land in the first 2^q bins.

For $q \in \{0, 1, \ldots, \lg r\}$, let Z_q denote the indicator random variable for the event that at least $2^q\mu + kx/2$ balls land in the first 2^q bins. So far, we have shown that

$$\Pr[Y_i \text{ for some } i \leq r] = O(\Pr[Z_q \text{ for some } q \leq \lg r]).$$

By a Chernoff bound, if we consider a given Z_q, and we define s such that $2^{q+s} = kx$, then

$$\Pr[Z_q] \leq \begin{cases} 2^{-\Omega(skx)} & \text{if } s > 0, \\ 2^{-\Omega\left(k^2x^2/2^q\right)} & \text{if } s \leq 0. \end{cases}$$

Recalling that $r = x^2k$, it follows that

$$\Pr[Z_q \text{ for some } q \leq \lg r] \leq \sum_{s>0} 2^{-\Omega(skx)} + \sum_{q=\lg(kx)}^{\lg(kx^2)} 2^{-\Omega\left(k^2x^2/2^q\right)}.$$

The first sum is a geometric series summing to $2^{-\Omega(kx)}$. The second sum is dominated by its final term which is $2^{-\Omega(k)}$. Thus the lemma is proven. □

To demonstrate how these results relate to linear probing, we now use them to re-create the classic upper bounds on insertion performance (Proposition 1) and query performance (Proposition 2) for ordered linear probing.

Proposition 1. *Starting with an empty linear-probing hash table of n slots, suppose that we perform $(1 - 1/x)n$ insertions. The running time T of the final insertion satisfies*

$$\Pr[T \geq kx^2] \leq 2^{-\Omega(k)}.$$

Proof. If the final insertion hashes to some position p, and takes time $T \geq kx^2$, then the insertion must have been inserted into a run of elements going from position $p - i$ to position $p + T - 1 \geq P + kx^2 - 1$ for some i. Consequently, the number of elements u that satisfy $h(u) \in [p-i, p+kx^2-1]$ must be at least $i+kx^2$, even though the expected number of such elements is $(1-1/x)(i+kx^2)$. By treating the positions $p+kx^2-1, p+kx^2-2, \ldots$ as bins, we can apply Lemma 1 to deduce that the probability of such an i existing is at most $2^{-\Omega(k)}$. □

Proposition 2. *Consider an ordered-linear-probing hash table with n slots that contains $(1 - 1/x)n$ elements and no tombstones, and suppose we perform a query. The running time T of the query satisfies*

$$\Pr[T \geq kx] \leq 2^{-\Omega(k)}.$$

Proof. Notice that we do not have to distinguish between positive and negative queries, since even for a negative query we only need to perform a linear scan until we find a key with a larger hash than the one we are querying.

If the key that we are querying hashes to some position p, and takes time $T \geq kx$, then all of the elements in positions $p, \ldots, p+T-2 \geq p+kx-2$ must have hashes at most p. Consider the largest i such that all of positions $p-i, \ldots, p+kx-2$ contain elements. All of the elements in positions $p-i, \ldots, p+kx-2$ must have hashes between $p-i$ and p. Thus the total number of elements that hash to positions $[p-i, p]$ must be at least $i+kx-1$, even though the expected number of elements that hash to those positions is $(1-1/x)(i+1)$. By treating the positions $p, p-1, p-2, \ldots$ as bins, we can apply Lemma 2 to deduce that the probability of any such i existing is at most $2^{-\Omega(k)}$. □

The proof of Proposition 2 comes with a corollary that will be useful to reference later.

Corollary 2. *Consider an ordered-linear-probing hash table with n slots that contains $(1 - 1/x)n$ elements and no tombstones. Consider a position i, and let T be the number of elements u that reside in positions i or greater, but that have hashes $h(u) < i$. Then*

$$\Pr[T \geq kx] \leq 2^{-\Omega(k)}.$$

5 Bounds on Insertion Surplus

In this section, we introduce two core technical propositions that will be used in subsequent sections for our analysis of ordered linear probing.

Consider a sequence S of operations which alternate between insertions and deletions. (Think of S as the operations between two rebuilds.) Let n be the number of slots in the hash table. Let P be a sub-interval of $[n]$ and let S_P denote the subset $\{u \in S \mid h(u) \in P\}$.

We say that a subset $S' \subseteq S_P$ is ***downward-closed*** (with respect to S_P) if it satisfies the following property: for every insertion or deletion $u \in S'$, every $v \in S_P$ that occurs temporally before u and satisfies $h(v) \geq h(u)$ is also in S'. Define the ***insertion surplus*** of a downward-closed set S' to be the number of insertions in S' minus the number of deletions in S', if there are more insertions than deletions in S', and 0 otherwise.

The purpose of this section is to prove upper and lower bounds on the maximum insertion surplus of any downward-closed subset $S' \subseteq S_P$. We will parameterize these bounds by $\mu := \mathbb{E}[|S_P|]/2$, which is the expected number of insertions (and also the expected number of deletions) in S_P.

Notice that S_P itself is downward-closed. If we assume that all of the operations in S_P are on distinct keys, then the expected insertion surplus of S_P will be $\Omega(\sqrt{\mu})$ (since the number of insertions and the number of deletions in S_P both have standard deviation $\Theta(\sqrt{\mu})$). A natural question is whether there exists any downward-closed subset $S' \subseteq S_P$ with a significantly larger insertion surplus.

This section proves two propositions.

Proposition 3. *Suppose that $|P| \geq \sqrt{\mu}$. With probability $1 - 1/\operatorname{poly}(\mu)$, every downward-closed subset $S' \subseteq S_P$ has insertion surplus $\tilde{O}(\sqrt{\mu})$.*

Proposition 4. *Suppose that the insertions/deletions in S_P are all on different keys. Suppose that $|S| \leq n$, that n is sufficiently large as a function of μ and $|P|$, and that $|P| \geq \sqrt{\mu}$. Then with probability $1 - o(1)$, there exists a downward-closed subset $S' \subseteq S_P$ with insertion surplus $\Omega(\sqrt{\mu \lg \lg \mu})$.*

Whereas Proposition 3 tells us that no downward-closed subset S' has significantly larger insertion surplus then the expected insertion surplus of S_P, Proposition 4 tells us that there most likely is some downward-closed subset S' whose insertion surplus is (at least slightly) asymptotically larger than the expected insertion surplus of S_P.

Reinterpreting insertion surplus in terms of paths on a grid. Before proving the propositions, we first reinterpret the propositions in terms of paths on a grid. Consider a grid G with height $|S|$ and width $|P|$. Define r so that $r + 1$ is the beginning of interval P. Color the cell (i, j) in the grid G blue if the i-th operation in S is an insertion whose hash is $r + j$; and color the cell (i, j) red if the i-th operation in S is a deletion whose hash is $r + j$. (Many cells will be neither red nor blue.) The blue and red cells in the grid G correspond to the insertions and deletions in S_P.

Now consider the set of ***monotone paths through G***, that is, the set of paths that begin in the bottom left corner, and then walk along grid lines to the top right corner, only ever moving up or to the right. Define the ***blue-red differential*** of a path to be the number of blue cells that reside below the path minus the number of red cells that reside below the path (we say that the path ***covers*** these cells).

A subset $S' \subseteq S$ is downward-closed if and only if there is a monotone path γ through G such that the blue and red cells covered by γ in G are precisely the insertions and deletions in S'. Thus, rather than considering downward-closed subsets S' of S_P, we can consider monotone paths γ through G, and rather than considering the insertion surplus of each subset S', we can consider the blue-red differential of each monotone path γ. Although this distinction may at first seem superficial, we shall see later that the geometry of monotone paths makes them amenable to clean combinatorial analysis.

Considering a more coarse-grained grid G'. One aspect of G that makes it potentially unwieldy is that it will likely be sparse, meaning that the vast majority of cells are neither red nor blue. Thus, in our proofs, it will also be useful to define a more coarse-grained grid G' that is laid on top of G. The grid G' has width and height $\sqrt{\mu}$, meaning that each cell in G' corresponds to a sub-grid of G with width $|P|/\sqrt{\mu}$ and height $|S|/\sqrt{\mu}$. To avoid confusion, we will refer to the blue/red cells in G as blue/red dots in G'.

Note that, since G' is a $\sqrt{\mu} \times \sqrt{\mu}$ grid, we are implicitly assuming that $\sqrt{\mu}$ is a positive integer; this assumption is w.l.o.g.. To simplify our discussion (so that we can treat all of the cells of G' as having uniform widths and heights), we will further assume that $|P|$ and $|S|$ are divisible by $\sqrt{\mu}$. These assumptions can easily be removed by rounding all of the quantities to powers of four, and performing the analysis using the rounded quantities.[17]

The grid G' is parameterized so that the expected number of blue dots (resp. red dots) in each cell is exactly 1. As terminology, we say that for each cell in G', the insertions and deletions that ***pertain*** that cell are the ones that have blue/red dots in that cell; and the keys that ***pertain*** to the cell are the keys that have at least one insertion/deletion pertaining to the cell. The expected number of distinct keys that pertain to a given cell in G' is at most 1. Moreover, by a Chernoff bound, and with probability $1 - 1/\operatorname{poly}(\mu)$, each cell has at most $O(\lg \mu)$ keys that pertain to it.

5.1 Proof of Proposition 3

We wish to show that, with probability $1 - 1/\operatorname{poly}(\mu)$, every monotone path γ through G has blue-red differential at most $\tilde{O}(\sqrt{\mu})$. The next lemma establishes that, rather than considering monotone paths in G, it suffices to instead consider monotone paths in G'.

Lemma 3. *With probability $1 - 1/\operatorname{poly}(\mu)$, the following holds. For every monotone path α with blue-red differential a in G, there is a monotone path β with blue-red differential b in G' such that $a - b = O(\sqrt{\mu} \lg \mu)$.*

Proof. Define β to be the same as α, except that the path is rounded to the grid lines of G' as follows: for any cell in G' that the path α goes through the interior of, we round the path so that β does not cover any of the points from that cell.

Every cell in G' that is entirely covered by α is also entirely covered by β; similarly, every cell in G' that is entirely not covered by α is also entirely not covered by β. Thus the only difference between α and β is that there are $O(\sqrt{\mu})$ cells in G' that are partially covered by α but that are not covered by β.

For each cell in G', define the ***risk potential*** of that cell to be the maximum blue-red differential of any monotone path in G from the bottom left corner of that cell to the top right corner of that cell. To complete the proof, it suffices to show that every cell in G' has risk potential at most $O(\lg \mu)$. Notice, however, that the risk potential of each cell is at most as large as the number of distinct keys that pertain to the cell. Thus the risk potentials of the cells are all $O(\lg \mu)$ with probability $1 - 1/\operatorname{poly}(\mu)$. □

To complete the proof of Proposition 3, it suffices to show that, with probability $1 - 1/\operatorname{poly}(\mu)$, every monotone path through G' has blue-red differential at most $\tilde{O}(\sqrt{\mu})$.

The rest of the proof is completed in two pieces. The first piece is to show that for every monotone path γ, it is possible to decompose the area under the path into rectangles where the sum of the perimeters of the rectangles is $O(\mu \lg \mu)$. The second piece is to show that, for each rectangle in G', the blue-red differential of that rectangle is at most the perimeter of the rectangle times $O(\lg \mu)$. Combining these two facts, we get a bound on the blue-red differential of any monotone path γ.

Lemma 4. *Consider any monotone path γ through an $a \times b$ grid, where $\ell = a + b$ is a power of two. The area under the path can be decomposed into disjoint rectangles such that the sum of the perimeters of the rectangles is $O(\ell \lg \ell)$.*

[17] Importantly, both propositions assume $|P| \geq \sqrt{\mu}$, so if we round both $|P|$ and μ to powers of 4, then the rounded value for $|P|$ will be a multiple of the rounded value for μ (and μ will be a square number).

Proof. We construct the rectangular decomposition through the following recursive process. Break the path γ into two pieces of length $\ell/2$ which we call γ_1 and γ_2, and let q be the point at which the two pieces meet. Let R be the rectangle whose top left corner is q, and whose bottom right corner is the bottom right point of the grid. Then we define our rectangular decomposition to consist of the rectangle R, along with a recursively constructed rectangular decomposition for the path γ_1, and a recursively constructed rectangular decomposition for the path γ_2. The two recursive decompositions take place in the grids containing γ_1 and γ_2, and the base case of the recursion is when we get to path that is either entirely vertical or entirely horizontal (meaning that there is no area to decompose into rectangles).

By design, the i-th level of recursion takes place on a grid with perimeter $O(\ell/2^i)$. It follows that the rectangles added in the i-th level of recursion each have perimeters $O(\ell/2^i)$. On the other hand, the i-th level of recursion has at most 2^i recursive subproblems, so the total perimeter of the rectangles added in those subproblems is at most $O(\ell)$. There are at most $O(\lg \ell)$ levels of recursion, so the sum of the perimeters of all of the rectangles in the decomposition is at most $O(\ell \lg \ell)$. □

Lemma 5. *Consider any rectangle X in grid G'. If X has perimeter k, then with probability $1 - 1/\operatorname{poly}(\mu)$, the number of blue dots minus the number of red dots in X is $O(k \lg \mu)$.*

Proof. Let T be the time window that X covers on its vertical axis. The total number of insertions that occur in T is the same as the number of deletions that occur in T, up to ± 1. Call an insertion in T ***serious*** if the key being inserted has not previously been deleted in the same time window, and call a deletion in T ***serious*** if the key being deleted is not subsequently reinserted in the same time window. Notice that the number of blue dots from non-serious insertions in X is the same as the number of red dots from non-serious deletions in X, so we can ignore both. Since the number of non-serious insertions equals the number of non-serious deletions, the number of serious insertions in T is the same (up to ± 1) as the number of serious deletions in T.

Since X has perimeter k, it can contain at most k^2 cells in G'. Thus the expected number of distinct keys that pertain to the cells in X is $O(k^2)$. It follows that the expected number of serious insertions (and similarly, the expected number of serious deletions) of keys that pertain to X is $O(k^2)$. In order for the number of blue dots minus the number of red dots in X to exceed $D = \Omega(k \lg \mu)$, we would need that either (a) the number of serious insertions pertaining to X is at least $D/2$ greater than its mean; or (b) the number of serious deletions pertaining to X is at least $D/2$ smaller than its mean. By a Chernoff bound, the probability of either (a) or (b) occurring is that most $1/\operatorname{poly}(\mu)$.[18] □

Proof of Proposition 3. By Lemma 3, it suffices to show with probability $1 - 1/\operatorname{poly}(\mu)$ that every monotone path γ through G' has blue-red differential $\tilde{O}(\sqrt{\mu})$. By Lemma 4, every such path has a rectangular decomposition where the sum of the perimeters of the rectangles is $\tilde{O}(\sqrt{\mu})$. By Lemma 5, with probability $1-1/\operatorname{poly}(\mu)$, the contribution of each of the rectangles to the blue-red differential of γ is that most $O(\lg \mu)$ times the perimeter of the rectangle. Thus, with probability $1-1/\operatorname{poly}(\mu)$, every monotone path γ through G' has blue-red differential $O(\sqrt{\mu} \lg^2 \mu)$. □

5.2 Proof of Proposition 4

We wish to construct a monotone path through G' that has blue-red deviation $\Omega(\sqrt{\mu \lg \lg \mu})$.

Let $a_{i,j}$ be the number of insertions that pertain to cell (i, j) and let $b_{i,j}$ the number of deletions that pertain to cell (i, j) in G'. Each $a_{i,j}$ and each $b_{i,j}$ is a binomial random variable with mean 1. However, the random variables are not completely independent (they are slightly negatively correlated), which makes them a bit unwieldy to work with. To handle this, the following lemma Poissonizes the random variables in order to show that they are $o(1)$-close to being independent.

[18] Note that all of the serious insertions (resp. serious deletions) are on distinct keys, meaning that their hashes are independent, hence the Chernoff bound.

Lemma 6. *Let A be the random variable $\langle a_{i,j}, b_{i,j} \mid i,j \in [\sqrt{\mu}]\rangle$ and let $\overline{A}$ be a random variable $\langle \overline{a}_{i,j}, \overline{b}_{i,j} \mid i,j \in [\sqrt{\mu}]\rangle$, where each $\overline{a}_{i,j}$ and each $\overline{b}_{i,j}$ is an independent Poisson random variable with mean 1. Then it is possible to couple the random variables A and $\overline{A}$ such that they are equal with probability $1 - o(1)$.*

Proof. Recall that, by assumption in Proposition 6, all of the operations in S are on different keys. We can think of each insertion/deletion as being performed on a random hash in $[n]$ (rather than being performed on any specific key). As part of our construction of $\overline{A}$, we will end up modifying S (i.e., adding and removing some operations) to get a new operation sequence $\overline{S}$. When adding new operations to S, we will need not bother associating the new operations with actual keys, and will instead think of the new operations is simply each being associated with a random hash.

We now describe our construction of $\overline{A}$. Let $T_1, T_2, \ldots, T_{\sqrt{\mu}}$ be the time windows that correspond to the rows of G'. Let $y_1, y_2, \ldots, y_{\sqrt{\mu}}, z_1, z_2, \ldots, z_{\sqrt{\mu}}$ be independent Poisson random variables with mean $|S|/(2\sqrt{\mu})$. Define $\overline{S}$ to be S, except that the operations in each time window T_i are modified so that the number of insertions is y_i and the number of deletions is z_i; note that this may require us to either add or remove operations to the time window. Then define $\overline{A}$ in exactly the same way as A, except using $\overline{S}$ in place of S. That is, we let $\overline{a}_{i,j}$ be the number of insertions in $\overline{S}$ that pertain to cell (i,j) in G' and we let $\overline{b}_{i,j}$ the number of deletions in $\overline{S}$ that pertain to cell (i,j) in G'.

To understand the distribution of $\overline{A}$, we make use of an important fact about Poisson random variables: if J balls are placed at random into K bins, and J is a Poisson random variable with mean ϕ, then for each bin the number of balls in the bin is an independent Poisson random variable with mean ϕ/K (see Chapter 5.4 of [95]). This implies that the $\overline{a}_{i,j}$'s and $\overline{b}_{i,j}$'s are independent Poisson random variables each of which has mean 1.

To complete the proof, we must establish that $\Pr[A \neq \overline{A}] = o(1)$. For each time window T_i, the expected number of operations that are in one of S or $\overline{S}$ but not the other is

$$O(\sqrt{|T_i|}) = O\left(\sqrt{|S|/\sqrt{\mu}}\right) = O\left(\sqrt{n/\sqrt{\mu}}\right) = O(\sqrt{n}).$$

In total over all $\sqrt{\mu}$ time windows T_i, the expected number of operations that are in one of S or $\overline{S}$ but not the other is therefore $O(\sqrt{n\mu})$. The probability that any of these operations are on keys that hash to P is

$$O\left(\frac{|P|}{n} \cdot \sqrt{n\mu}\right) = O(|P|\sqrt{\mu/n}),$$

which by the assumptions of Proposition 6 is $o(1)$. □

Throughout the rest of the proof, we will treat the $a_{i,j}$s and $b_{i,j}$s as independent Poisson random variables each of which has mean 1. Our proof will make use of the fact that for any Poisson random variable J with mean $\phi \geq 1$,

$$\Pr[J > \phi + \sqrt{\phi}t] \geq \exp(-\Omega(t^2)). \tag{3}$$

For a derivation of (3), see Theorem 1.2 of [118]. Set $D = \sqrt{\lg\lg\mu}/c$, where c is a sufficiently large constant. We will be making use of (3) in the case where $t = 2D$, that is,

$$\begin{aligned}\Pr[J > \phi + 2\sqrt{\phi}D] &\geq \exp(-\Omega((\sqrt{\lg\lg\mu}/c)^2)) \\ &= \exp(-\Omega(\lg\lg\mu/c^2)) \\ &\geq 1/\sqrt{\lg\mu}. \end{aligned} \tag{4}$$

We now construct a monotone path γ through the grid G', and show that γ has blue-red deviation $\Omega(\sqrt{\mu\lg\lg\mu})$ with probability $1 - o(1)$. The construction of γ is recursive, with different levels of recursion operating on squares

grids of different sizes. To avoid ambiguity, we will use k to refer to the height (or width) of the grid in the current recursive sub-problem, and we will use $\sqrt{\mu}$ to refer to the height (or width) of the grid in the top-level sub-problem.

The construction of γ in a $k \times k$ subproblem is performed as follows. Break the $k \times k$ grid into four $k/2 \times k/2$ quadrants. If the top left quadrant has blue-red deviation at least kD (that is, it contains at least kD more blue dots than red dots) then we say that the subproblem ***successfully terminates***, and we set γ to be the path that consists of k vertical steps followed by k horizontal steps. Otherwise, we construct γ by concatenating together a path recursively constructed through the bottom left quadrant and a path recursively constructed through the top right quadrant. If a recursive subproblem is on a 1×1 grid, then we return the path consisting of a vertical step followed by a horizontal step, and we call the subproblem a ***failed leaf***.

Lemma 7. *Each subproblem with $k > 1$ has probability at least $\Omega(1/\sqrt{\lg \mu})$ of successfully terminating.*

Proof. The number of blue dots and the number of red dots in the top left quadrant of the subproblem are both Poisson random variables with means $k^2/4$. It follows by (4) that the number of blue dots has probability at least $1/\sqrt{\lg \mu}$ of exceeding its mean by kD. Since the number of red dots has probability at least $\Omega(1)$ of being less than or equal to its mean, it follows that the blue-red deviation of the quadrant is at least kD with probability at least $\Omega(1/\sqrt{\lg \mu})$. □

Lemma 8. *With probability $1 - o(1)$, the number of failed leaves is less than $\sqrt{\mu}/2$.*

Proof. We will prove that the expected number of failed leaves is $o(\sqrt{\mu})$, after which the lemma follows by Markov's inequality.

There are $\sqrt{\mu}$ potential failed leaves in the recursion tree, so it suffices to show that each of them has $o(1)$ probability of occurring. In order for a given failed leaf to occur, the recursion path of depth of $\lg \sqrt{\mu}$ that must occur. By Lemma 7, each of the subproblems in the recursion path (except for the leaf) independently has at least a $\Omega(1/\sqrt{\lg \mu})$ probability of successfully terminating. Thus each potential failed leaf in the recursion tree has probability at most

$$\left(1 - \Omega(1/\sqrt{\lg \mu})\right)^{\lg \sqrt{\mu} - 1} \leq o(1)$$

of occurring. □

We can now analyze the blue-red deviation of γ to prove Proposition 4.

Proof of Proposition 4. By Lemma 8, we may assume that the number of failed leaves is less than $\sqrt{\mu}/2$.

Say that the ***width*** of a subproblem is width of the grid in which it takes place. The sum of the widths of the leaf subproblems is $\sqrt{\mu}$. Each failed leaf has width 1, so if the number of failed leaves is less than $\sqrt{\mu}/2$, then the sum of the widths of the leaves that successfully terminate must be at least $\sqrt{\mu}/2$.

For each leaf subproblem with width k that successfully terminates, its top left quadrant contributes $\Omega(k\sqrt{\lg \lg \mu})$ to the blue-red deviation of γ. Summing over the leaf subproblems that successfully terminate, the top left quadrants of all of them contribute at least $\Omega(\sqrt{\mu \lg \lg \mu})$ to the blue-red deviation.

Additionally, we must consider the effect of the blue and red dots below γ that are not contained in any of these aforementioned top-left quadrants. Once the path γ is determined, the number X of such blue dots and the number Y of such red dots are independent Poisson random variables satisfying $\mathbb{E}[X] = \mathbb{E}[Y] \leq \mu$. By a Chernoff bound, we have that with probability $1 - o(1)$,

$$Y - X \leq \sqrt{\mu \lg \lg \lg \mu}.$$

Thus, with probability $1 - o(1)$, the blue-red deviation of γ is at least

$$\Omega(\sqrt{\mu \lg \lg \mu}) - \sqrt{\mu \lg \lg \lg \mu} = \Omega(\sqrt{\mu \lg \lg \mu}).$$

□

6 Relating Insertion Surplus to Crossing Numbers

In this section we use our insertion-surplus bounds from Section 5 to obtain bounds on a different set of quantities $\{c_j\}_{j\in[n]}$ that we call the crossing numbers; later, in Section 7, our bounds on crossing numbers will allow for us to analyze the amortized costs of insertions/deletions/queries in ordered linear probing.

Consider an ordered linear probing hash table with n slots, and suppose that the hash table is initialized to have load factor $1 - 1/x$ or smaller. Consider a sequence of insertion and deletion operations S such that the load factor never exceeds $1 - 1/x$. (Note that, unlike in Section 5, the lemmas in this section will not all require that S alternates between insertions and deletions.)

Based on the initial state of the hash table and on the sequence S of operations, define the ***crossing numbers*** $c_1, c_2, \ldots, c_n$ so that c_i is the number of times that an insertion with a hash smaller than i either (a) uses a tombstone left by a key that had hash at least i; or (b) uses a free slot in a position greater than or equal to i.

The purpose of this section is to prove nearly tight bounds on $\mathbb{E}[c_i]$. Subsequent sections will then show how to use these bounds in order to analyze the performance of linear probing.

We will need the following additional definitions. Define the ***insertion surplus*** of a subinterval $P \subseteq [n]$ to be the maximum insertion surplus of any downward-closed subset of $\{u \in S \mid h(u) \in P\}$, minus the number of free slots that are initially in the range P. Define the ***peak*** p_u of an insertion u to be the hash of the tombstone that the insertion uses (if it uses a tombstone) or the position of the free slot that the insertion uses (if it uses a free slot).

The following lemmas characterize the crossing numbers in terms of the insertion surpluses of intervals.

Lemma 9. *For each $s \in [n]$, there exists an interval $P = [r, s-1]$ whose insertion surplus is at least c_s.*

Proof. Call an insertion u with hash smaller than s ***special*** if it satisfies the following recursive property: either (a) $p_u \geq s$; or (b) there is another special insertion v that occurs temporally after u such that $p_u \geq h(v)$. Call a deletion v ***special*** if $h(v) < s$ and there exists a special insertion u that occurs temporally after v and satisfies $h(u) \leq h(v)$.

Let r be the smallest hash of any special insertion/deletion, and define $P = [r, s-1]$. We will prove that the insertion surplus of P is at least c_s. Towards this end, define A to be the number of special insertions, define B to be the number of special deletions, and define C to be the number of free slots initially in P. The set of special operations is downward-closed by design, and its insertion surplus is $A - B - C$. Thus we wish to show that

$$A - B - C \geq c_s.$$

In order for an insertion u to contribute to the crossing number c_s, the insertion must have peak $p_u \geq s$ and thus must also be special. To complete the proof, we will show that there are at least $B + C$ special insertions with peaks $p_u < s$ (and thus at most $A - B - C$ special insertions have $p_u \geq s$). That is, we will show that every tombstone created by a special deletion and every free slot initially in P is used by some special insertion.

Consider a tombstone that is created by a special deletion v. Since v is special, there must exist a special insertion u that occurs after v and satisfies $h(u) \leq h(v)$. Let u be the last such insertion. We must have that $p_u > h(v)$, since otherwise, we could arrive at a contradiction as follows: in order so that u could be special, there would have to be a subsequent special insertion z with $h(z) \leq p_u$; but this would imply that $h(z) \leq h(v)$, which would contradict the fact that u is the final special insertion satisfying $h(u) \leq h(v)$. Since $p_u > h(v)$, it must be that the tombstone created by the deletion v has already been used by the time that insertion u is performed. The insertion w that used the tombstone must have occurred before the insertion u and must have had peak $p_w = h(v) \geq h(u)$. This means that w is itself a special insertion. Thus the tombstone created by v is used by a special insertion, as desired.

Now consider a free slot j that is initially present in P. By the definition of P, there must exist a special insertion u that satisfies $h(u) \leq j$. Let u be the last such insertion. We must have that $p_u > j$, since otherwise, we could arrive at a contradiction as follows: in order so that u could be special, there would have to be a subsequent

special insertion z with $h(z) \leq p_u$; but this would imply that $h(z) \leq j$, which would contradict the fact that u is the final special insertion satisfying $h(u) \leq j$. Since $p_u > j$, it must be that the free slot j has already been used by the time that insertion u is performed. The insertion w that used slot j must have occurred before the insertion u and must have had peak $p_w = j \geq h(u)$. This means that w is itself a special insertion. Thus the free slot is used by a special insertion, as desired. □

The converse of the previous lemma is also true.

Lemma 10. *If there exists an interval $P = [r, s-1]$ with insertion surplus q, then $c_s \geq q$.*

Proof. Let S' be the downward-closed subset of $\{u \in S \mid h(u) \in P\}$ with the largest insertion surplus. Every insertion in S' must either (a) use a tombstone created by a deletion in S', (b) use a free slot initially present in P, or (c) have peak at least s. It follows that if A is the number of insertions in S', B is the number of deletions in S', and C is the number of free slots initially in P, then we must have that

$$c_s \geq A - B - C.$$

Since the quantity on the right side is exactly the insertion surplus of P, the proof is complete. □

The previous lemmas tells us that, in order to understand the crossing numbers c_s, it suffices to understand the insertion surplus of each interval P. This insertion surplus, in turn, depends on two quantities: the maximum insertion surplus of any downward-closed subset of $\{u \in S \mid h(u) \in P\}$; and the number of free slots initially in P. We have already achieved a good understanding of the first quantity in the previous section. The next two lemmas analyze the second quantity.

Lemma 11. *Suppose that the hash table initially has load factor $1-\varepsilon$. Consider any interval $P \subseteq [n]$ of size at least $c\,\varepsilon^{-2} \lg \varepsilon^{-1}$, where c is taken to be a sufficiently large constant. With probability $1 - 1/\operatorname{poly}|P|$, the interval P initially contains at least $\Omega(\varepsilon|P|)$ free slots.*

Proof. The expected number of elements that hash into P initially is $(1-\varepsilon)|P|$, which since $|P| \geq c\,\varepsilon^{-2} \lg \varepsilon^{-1}$, is at most $|P| - \sqrt{|P| c \lg |P|}$. It follows by a Chernoff bound that, with probability $1 - 1/\operatorname{poly}(|P|)$, the number of elements that initially hash into P is at most $|P| - \frac{1}{2}\sqrt{|P| c \lg |P|}$. On the other hand, by Corollary 2, and with probability $1 - 1/\operatorname{poly}(|P|)$, the number of elements that reside in P but hash to a position prior to P is at most $O(\varepsilon^{-1} \lg P) \leq \frac{1}{4}\sqrt{|P| c \lg |P|}$. The total number of elements that reside in P is therefore at most $|P| - \frac{1}{4}\sqrt{|P| c \lg |P|}$, which completes the proof. □

Lemma 12. *Suppose that the hash table initially has load factor $1-\varepsilon$. Consider any interval $P = [a, b] \subseteq [n]$ of size ε^{-2}/c, where c is taken to be a sufficiently large constant. With probability $\Omega(1)$, there are initially no free slots in P.*

Proof. Consider the state of the hash table initially, and let r be the length of the run of non-free slots beginning at position a. Knuth in [73] established that $\mathbb{E}[r] = \Theta(\varepsilon^{-2})$. On the other hand, Proposition 1 tells us that $\Pr[r > k\varepsilon^{-2}] \leq 1/\operatorname{poly}(k)$ for all k. The only way that these two facts can be consistent is if $r = \Omega(\varepsilon^{-2})$ with probability $\Omega(1)$. Thus the lemma is established. □

We are now in a position to upper bound the crossing number c_j.

Proposition 5. *Suppose that the hash table initially has load factor $1 - 1/x$, suppose that $|S| = \Omega(n/x)$ and $|S| \leq n$, and suppose that S alternates between insertions and deletions.*

There exists a positive constant d such that for any $j \in [n]$ and any

$$r \geq \sqrt{\frac{|S|}{n}x^2} \lg^d x,$$

we have $q_j < r$ with probability $1 - 1/\operatorname{poly}(r)$.

Proof. Define

$$\mu_i = |S| \cdot i/n$$

to be the expected number of operations in S that hash into the $[j-i, j-1]$. Define

$$\lambda_i = \sqrt{\mu_i} \operatorname{polylog} \mu_i$$

where the polylogarithmic factor is selected so that Proposition 3 offers the following guarantee: with probability $1 - 1/\operatorname{poly}(\mu_i)$, every downward-closed subset of $\{u \in S \mid h(u) \in [j-i, j-1]\}$ has insertion surplus less than λ_i. Let

$$K = x^2 \lg^c x$$

for some sufficiently large positive constant c. This results in

$$\begin{aligned}\lambda_K &= \sqrt{\frac{|S|}{n}x^2 \lg^c x} \operatorname{polylog}\left(\frac{|S|}{n}x^2 \lg^c x\right) \\ &= \sqrt{\frac{|S|}{n}x^2} \operatorname{polylog} x.\end{aligned}$$

Thus, if we select the constant d in the proposition statement appropriately, then the requirement that $r \geq \sqrt{\frac{|S|}{n}x^2} \lg^d x$ implies that $r \geq \lambda_K$, and thus that $r = \lambda_R$ for some $R \geq K$. To prove the proposition, it suffices to establish that for every $R \geq K$, we have

$$\Pr[c_j \geq \lambda_R] \leq 1/\operatorname{poly}(\lambda_R). \tag{5}$$

Note that in the parameter regime $R \geq K$, we have that $\operatorname{poly}(\lambda_R) = \operatorname{poly}(\mu_R) = \operatorname{poly}(R)$ (here we are using that $\Omega(n/x) \leq |S| \leq O(n)$), so we will treat the three as interchangeable throughout the rest of the proof.

Define $P_i = [j-i, j-1]$ and define $s(P_i)$ to be the insertion surplus of P_i. By Lemma 9,

$$\begin{aligned}\Pr[c_j \geq \lambda_R] &\leq \Pr[s(P_i) \geq \lambda_R \text{ for some } i] \\ &\leq \Pr[s(P_i) \geq \lambda_R \text{ for some } i < R] + \sum_{i \geq R} \Pr[s(P_i) > 0].\end{aligned}$$

To prove (5), we begin by bounding $\Pr[s(P_i) \geq \lambda_R \text{ for some } i < R]$. If $s(P_i) \geq \lambda_R$ for some $i < R$, then there must be a downward-closed subset S' of $\{u \in S \mid h(u) \in [j-(R-1), j-1]\}$ such that the insertion surplus of S' is at least λ_R. But by Proposition 3 and by the definition of λ_{R-1}, we know that with probability $1 - 1/\operatorname{poly}(\mu_{R-1})$ (and thus also in λ_{R-1}), every such S' has insertion surplus at most $\lambda_{R-1} < \lambda_R$. Thus the probability that $s(P_i) \geq \lambda_R$ for any $i < R$ is at most $1/\operatorname{poly}(\lambda_R)$.

To complete the proof, it remains to show that

$$\sum_{i \geq R} \Pr[s(P_i) > 0] \leq \frac{1}{\operatorname{poly}(\lambda_R)}.$$

We will establish a stronger statement, namely that for every $i \geq K$,

$$\Pr[s(P_i) > 0] \leq \frac{1}{\operatorname{poly}(i)}.$$

Since $i > K$, we can apply Lemma 11 to deduce that, with probability $1 - 1/\operatorname{poly}(i)$, the interval P_i initially contains at least $\Omega(i/x)$ free slots. We further have that, by Proposition 3, and with probability $1 - 1/\operatorname{poly}(\mu_i) = 1 - 1/\operatorname{poly}(i)$, every downward-closed subset of $\{u \in S \mid h(u) \in [j-i, j-1]\}$ has insertion surplus less than λ_i. It follows that $s(P_i)$ is that most

$$\max(0, \lambda_i - \Omega(i/x)).$$

In order to establish that $s(P_i)$ is zero, it suffices to show that

$$\lambda_i = o(i/x).$$

This is simply a matter of calculation:

$$\begin{aligned}
\lambda_i &= \sqrt{\mu_i} \operatorname{polylog} \mu_i && \text{(by definition of } \lambda_i\text{)} \\
&= \sqrt{\frac{|S|i}{n}} \operatorname{polylog}\left(\frac{|S|i}{n}\right) && \text{(by definition of } \mu_i\text{)} \\
&\leq \sqrt{i} \operatorname{polylog} i && \text{(since } |S| \leq n\text{)} \\
&\leq O\left(\frac{i \operatorname{polylog} i}{\sqrt{K \lg^c(i/K)}}\right) && \text{(since } i > K\text{)} \\
&= O\left(\frac{i \operatorname{polylog} i}{\sqrt{x^2 \lg^c x \lg^c(i/K)}}\right) && \text{(since } K = x^2 \lg^c x\text{)} \\
&= O\left(\frac{i \operatorname{polylog} i}{\sqrt{x^2 \lg^c K \lg^c(i/K)}}\right) && \text{(since } K = x^2 \lg^c x\text{)} \\
&= O\left(\frac{i \operatorname{polylog} i}{\sqrt{x^2 \lg^c i}}\right) && \text{(since } \lg a \lg b \geq \Omega(\lg(ab))\text{)} \\
&= \frac{i/x \operatorname{polylog} i}{\sqrt{\lg^c i}} \\
&= o(i/x) && \text{(since } c \text{ is a sufficiently large constant).}
\end{aligned}$$

□

Corollary 3. *Suppose that the hash table initially has load factor $1 - 1/x$, suppose that $|S| = \Omega(n/x)$ and $|S| \leq n$, and suppose that S alternates between insertions and deletions.*

For each $j \in [n]$,

$$\mathbb{E}[c_j] \leq \sqrt{\frac{|S|}{n} x^2} \operatorname{polylog} x.$$

We can also obtain a nearly matching lower bound for $\mathbb{E}[c_j]$.

Proposition 6. *Suppose that the hash table initially has load factor $1 - 1/x$, suppose that $|S| = \Omega(n/x)$ and $|S| \leq n$, and suppose that S alternates between insertions and deletions.*

Further suppose that each operation in S applies to a different key. Then for each $j \in [n]$,

$$\mathbb{E}[c_j] = \Omega\left(\sqrt{\frac{|S|}{n}x^2 \lg\lg x}\right).$$

Proof. By Lemma 10, it suffices to show that there is some interval $P_i = [j - i, j - 1]$ with insertion surplus

$$\Omega\left(\sqrt{\frac{|S|}{n}x^2 \lg\lg x}\right).$$

By Lemma 12, there exists a positive constant c such that, with probability $\Omega(1)$, the interval P_{cx^2} in the hash table initially (i.e., at the beginning of the operations S) contains no free slots. Furthermore, Proposition 4 tells us that with probability $1 - o(1)$, there exists a downward-closed subset of $\{u \in S \mid h(u) \in P_{cx^2}\}$ with insertion surplus at least

$$\Omega\left(\sqrt{\frac{|S|}{n}x^2 \lg\lg\left(\frac{|S|}{n}x^2\right)}\right)$$
$$= \Omega\left(\sqrt{\frac{|S|}{n}x^2 \lg\lg x}\right).$$

With probability $\Omega(1)$, both of the preceding events occur simultaneously. Thus the proposition is proven. □

The previous two propositions both focus on the case in which the workload S alternates between insertions and deletions. We conclude this section by considering the case where S is allowed to perform an arbitrary sequence of insertions and deletions, subject only to the constraint that the load factor never exceeds $1 - 1/x$.

Proposition 7. *Suppose that the hash table begins at a load factor of at most $1-1/x$, and that the sequence of operations S keeps the load factor at or below $1-1/x$. Finally, suppose that $|S| \leq n/\operatorname{polylog}(x)$. Then for each $j \in [n]$,*

$$\mathbb{E}[c_j] = O(x).$$

Proof. We begin by constructing an alternative sequence of insertions/deletions $\overline{S}$ such that the crossing numbers with respect to $\overline{S}$ are guaranteed to be at least as large as the crossing numbers with respect to S. We will then complete the proof by analyzing the crossing numbers of $\overline{S}$.

Suppose that the hash table initially contains $(1 - 1/x)n - w$ elements for some w. We construct $\overline{S}$ through two steps:

- Call an insertion ***novel*** if the key being inserted has not been inserted in the past and was not originally present in the hash table. The first step is to take each of the first w novel insertions, and to move them to the front of the operation sequence.[19]

[19] We can assume without loss of generality that there are at least w such novel insertions, since if there are not, we can artificially add additional insertions to the end of S and then perform the rest of the proof without modification.

- Call a triple of three consecutive operations ***unbalanced*** if the triple consists of two deletions followed by an insertion. At least one of the two deletions in such a triple must act on a different key than the insertion acts on. We can ***balance*** the triple by changing the order of operations so that the aforementioned deletion occurs last. The second step in constructing $\overline{S}$ is to repeatedly find and balance any unbalanced triples until there are no such triples left.

Observe that the sequence $\overline{S}$ is a valid sequence of operations, since the transformation from S to $\overline{S}$ never swaps the order of any two operations that act on the same key.

We claim that the crossing numbers with respect to $\overline{S}$ are at least as large as the crossing numbers with respect to S. The transformation from S to $\overline{S}$ moves certain insertions to occur earlier than they would have otherwise, and certain deletions to occur later than they would have otherwise. Importantly, these types of moves cannot decrease the insertion surplus of any interval P in the hash table. By Lemmas 9 and 10, the crossing numbers are completely determined by the insertion surpluses of the intervals $P \subseteq [n]$. Since the insertion surpluses for $\overline{S}$ are at least as large as those for S, it follows that the crossing numbers for $\overline{S}$ are also at least as large as those for S.

Our next claim is that $\overline{S}$ never causes the load factor to exceed $1 - 1/x$. This can be seen by analyzing each of the two steps of the construction separately. The first step modifies only the window of time in which the first w novel insertions are performed; no rearrangement of the operations in this window of time can possibly cause the load factor to exceed $1 - 1/x$. The second step repeatedly performs balancing operations on unbalanced triples; such a balancing operation does not change the maximum load factor that is achieved during the triple, however, since that load factor is achieved prior to the first operation of the triple. Combining the analyses of the two steps, we see that the load factor never exceeds $1 - 1/x$.

Now let us reason about the structure of $\overline{S}$. By design, $\overline{S}$ begins with w insertions, bringing the load factor up to exactly $1 - 1/x$. Since the load factor never exceeds $1 - 1/x$, and since there are never two deletions in a row followed by an insertion, it must be that the remaining insertions in $\overline{S}$ are each preceded by exactly one deletion. In other words, $\overline{S}$ must consist of three parts $\overline{S}_1, \overline{S}_2, \overline{S}_3$ where $\overline{S}_1$ consists only of insertions, $\overline{S}_2$ alternates between deletions and insertions, and $\overline{S}_3$ consists only of deletions.

Since $\overline{S}_3$ consists only of deletions, it does not contribute anything to the crossing numbers. By Corollary 3, the expected contribution of the operations in $\overline{S}_2$ to each crossing number c_j is at most $O(x)$.

It remains to bound the contribution of $\overline{S}_1$ to the crossing numbers. If an insertion takes time t, then it can contribute at most t to the sum $\sum_j c_j$. Knuth showed in [73] that the total time needed to fill an empty table up to a load factor of $1 - 1/x$ is $O(nx)$ in expectation. Thus the expected contribution of $\overline{S}_1$ to $\sum_j c_j$ is $O(nx)$, completing the proof. □

7 Relating Crossing Numbers to Running Times

In this section, we give nearly tight bounds on the performance of ordered linear probing. Notably, we find that, if the size R of each rebuild window is chosen correctly, then the amortized time per insertion is guaranteed to be $\tilde{O}(x)$. The key technical component to the section will be a series of arguments transforming our bounds on crossing numbers (in Section 6) into bounds on running times.

Consider an ordered linear probing hash table that uses tombstones for deletions. Recall that there are three parameters: the number n of slots in the table, the number R of insertions in each time window between rebuilds, and the maximum load factor $1 - 1/x$ that the hash table ever achieves. Based on these parameters, we wish to analyze the average running time of the operations being performed on the hash table.

We will be focusing exclusively on the regime in which $R = \Omega(n/x)$ and $R \leq n$. Since each rebuild can be implemented in linear time $O(n)$, the average time spent performing rebuilds per operation is $O(n/R) = O(x)$

(which for our purposes will be negligible). Thus the focus of our analysis will be on analyzing the costs of the operations that occur between consecutive rebuilds.

Before diving into the details, we remark that there are two main technical challenges that our analysis must overcome. The first challenge is obvious: we must quantify the degree to which tombstones left behind by deletions improve the performance of subsequent insertions. The second challenge is a bit more subtle: in order to support large rebuild-window sizes R, our analysis must be robust to the fact that tombstones can accumulate over time, increasing the effective load factor of the hash table. This latter challenge is exacerbated by the fact that the choice of which tombstones are in the table at any given moment is a function not only of the sequence of operations being performed, but also of the randomness in the hash table. This means that, even if the cumulative load factor from the elements and tombstones can be bounded (e.g., by $1 - 1/(2x)$), we still cannot apply the classic analysis at that load factor in order to bound the expected time of queries.

One of the interesting features of our analysis is that we completely circumvent the issue of how fast tombstones accumulate over time. Rather than focusing on what the effective load factor of the hash table is at each moment in time, the analysis instead analyzes the state of the hash table at the beginning of the rebuild window, and then analyzes the dynamics of how the local structure of the hash table changes over time.

In the following lemmas, we will focus on a single window W of time between two rebuilds. We begin by defining three quantities that we will be able to express the running times of operations in terms of.

For a given position $i \in [n]$, define the ***positional offset*** o_i to be the quantity $j - i$ where $j \geq i$ is the largest position such that, at the end of the time window W, all of the positions $[i, j-1]$ contain elements and tombstones whose hashes are smaller than i. Note that, although the positional offset is defined at the end of the time window W, if we were to define the same quantity at any other point during the time window, it would be at most o_i (that is, the positional offset only increases over time).

For a given position $i \in [n]$, define the ***spillover*** s_i to be the largest $k \geq 0$ such that if we consider all keys that are either initially present or inserted at some point during W, at least $4k$ of them have hashes in the range $[i, i+k)$.

For each insertion u, define the ***displacement*** d_u of the insertion to be $p_u - h(u)$, where p_u is the peak of the insertion as defined in Section 6.

Lemma 13. *If an insertion u hashes to a position i, then the insertion takes time at most*

$$O(o_i + s_i + d_u + 1).$$

Similarly, if a query/deletion u hashes to a position i, then the operation takes time at most

$$O(o_i + s_i + 1).$$

Proof. Consider an insertion u that hashes to a position i. If u uses a free slot in some position $i + r$, then the time to perform the insertion is $r = d_u + 1$. Suppose, on the other hand, that u uses a tombstone with some hash $i + t$, and the tombstone is in some position $i + r$. Then the running time of the insertion is r, and we wish to show that

$$r = O(o_i + s_i + t + 1). \tag{6}$$

If $r - o_i = O(t)$, then (6) trivially holds and we are done. Otherwise, we may assume that $r - o_i \geq 4(t+1)$. By the definition of the positional offset o_i, all of the elements/tombstones in positions $[i + o_i, i + r]$ must have hashes in the range $[i, i + t]$. Combining this with the fact that $r - o_i \geq 4(t+1)$, it follows that the spillover s_i satisfies $s_i \geq \lfloor (r - o_i)/4 \rfloor$, hence (6).

Next consider a query/deletion u that hashes to a position i. The operation takes time at most $O(o_i + T)$ where T is the total number of elements with hash i that are either present at the beginning of W or inserted at some point during W. By the definition of s_i, we have that $T \leq 4s_i + O(1)$. Thus the operation takes time $O(o_i + s_i + 1)$. □

Our next lemma relates o_i, s_i, and d_u to the crossing numbers c_i defined in the previous section.

Lemma 14. *For each $i \in [n]$, the positional offset o_i satisfies $o_i \geq c_i$ and*

$$\mathbb{E}[o_i] = O(x) + \mathbb{E}[c_i], \tag{7}$$

and the spillover s_i satisfies

$$\mathbb{E}[s_i] = O(1). \tag{8}$$

Finally, if we consider a random insertion u in the time window W, then

$$\mathbb{E}[d_u] = \frac{n}{R}\mathbb{E}[c_i]. \tag{9}$$

Proof. Although the positional offset o_i is only defined at the end of the time window W, let us slightly abuse notation and think about how the quantity evolves over time (that is, what would happen if we defined the quantity at each point in time in the time window). By Corollary 2, the initial positional offset (at the beginning of W) has expected value $O(x)$. During the time window, the positional offset increases by one each time that an insertion whose hash is less than i has a peak that is at least i. The number of such insertions is precisely c_i. Since these insertions are the only operations that can change the positional offset, it follows that $o_i \geq c_i$ and $\mathbb{E}[o_i] = O(x) + \mathbb{E}[c_i]$.

Next we consider the spillover s_i. Let V be the set of all elements that are present at some point during W. Then $|V| \leq 2n$, and the expected number of elements in V that hash to a given position j is at most 2. It follows by Lemma 1 that $\Pr[s_i > k] \leq \exp(-\Omega(k))$ for all k. This implies (8).

Finally we establish (9). Observe that, if an insertion u has displacement d_u, then the insertion contributes to exactly d_u crossing numbers c_i. It follows that

$$\sum_u d_u = \sum_{i \in [n]} c_i.$$

If we select a random insertion u out of the R insertions that occur in W, then

$$R \cdot \mathbb{E}[d_u] = \sum_{i \in [n]} c_i.$$

Since the c_i's all of the same expected values, it follows that for a given i,

$$R \cdot \mathbb{E}[d_u] = n \cdot \mathbb{E}[c_i].$$

This implies (9). □

We are now prepared to prove the main results of the section. We begin by considering a hovering workload, that is a workload in which queries can be performed at arbitrary times, but insertions and deletions must alternate.

Theorem 1. *Consider an ordered linear probing hash table that uses tombstones for deletions, and that performs rebuilds every R insertions. Suppose that the table is initialized to have capacity n and load factor $1 - 1/x$, where $R = \Omega(n/x)$ and $R \leq n$. Finally, consider a sequence S of operations that alternates between insertions and deletions (and contains arbitrarily many queries).*

Then the expected amortized time I spent per insertion satisfies

$$I \leq \tilde{O}\left(x\sqrt{\frac{n}{R}}\right) \tag{10}$$

and, if all insertions/deletions in each rebuild window are on distinct keys, then

$$I \geq \Omega\left(x\sqrt{\frac{n}{R}\lg\lg x}\right). \tag{11}$$

Moreover, the expected time Q of a given query/deletion satisfies

$$Q \leq O(x) + \tilde{O}\left(x\sqrt{\frac{R}{n}}\right) \tag{12}$$

and, if all operations in each rebuild window are on distinct keys, then for any negative query at the end of a rebuild window, we have

$$Q \geq \Omega\left(x + x\sqrt{\frac{R}{n}\lg\lg x}\right). \tag{13}$$

Proof. Consider a random insertion u with some hash i. By Lemma 13, we have that u takes time at most $O(o_i + s_i + d_u + 1)$. By Lemma 14, this has expectation at most

$$O\left(x + \mathbb{E}[c_i] + \frac{n}{R}\mathbb{E}[c_i] + 1\right) = O\left(x + \frac{n}{R}\mathbb{E}[c_i]\right).$$

By Corollary 3, this is that most[20]

$$\tilde{O}\left(x + \frac{n}{R}\sqrt{\frac{R}{n}x^2}\right) = \tilde{O}\left(x\sqrt{\frac{n}{R}}\right).$$

On the other hand, the insertion time is necessarily at least d_u, which by Lemma 14, has expectation

$$\frac{n}{R-1}\mathbb{E}[c_j]$$

for each $j \in [n]$. If we assume that every insertion/deletion in the rebuild window is on a different key, then we can further apply Proposition 6 to conclude that the insertion time has expected value at least

$$\Omega\left(\frac{n}{R}\sqrt{\frac{R}{n}x^2\lg\lg x}\right) = \Omega\left(x\sqrt{\frac{n}{R}\lg\lg x}\right).$$

[20]There is one technicality that we must be slightly careful about here: the hash $i = h(u)$ is independent of where every key $v \neq u$ hashes to, but is (trivially) not independent of where key u hashes to. However, since u is only one key, it can easily be factored out of the analysis so that we can treat i as being a random slot (independent of the hash function h).

Now instead consider a query/deletion u that hashes to some position i. By Lemma 13, we have that u takes time at most $O(o_i + s_i + 1)$. By Lemma 14, the expected time that u takes is therefore at most

$$O\left(x + \mathbb{E}[c_i] + 1\right).$$

By Corollary 3, this is that most

$$\tilde{O}\left(x + \sqrt{\frac{R}{n}x^2}\right) = \tilde{O}\left(x + x\sqrt{\frac{R}{n}}\right).$$

If we assume that the query is a negative query, performed at the end of a rebuild window whose insert/delete operations are all on different keys, then the query time is necessarily at least o_i, which by Lemma 14 is at least c_i. It follows by Proposition 6 that the expected query time is at least

$$\Omega\left(x\sqrt{\frac{R}{n}\lg\lg x}\right).$$

The expected query time is also $\Omega(x)$ trivially, by the standard analysis of linear probing [73].

□

Theorem 1 has several important corollaries. Our first corollary considers the setting in which rebuilds are performed every $\Theta(n/x)$ insertions.

Corollary 4. *Suppose $R = \Theta(n/x)$. Then*

$$I \leq \tilde{O}(x^{1.5}),$$

and, if all insertions/deletions in each rebuild window are on distinct keys, then

$$I \geq \Omega(x^{1.5}\sqrt{\lg\lg x}).$$

Moreover, $Q = \Theta(x)$.

Our next corollary considers the setting in which rebuilds are performed every $n/\operatorname{polylog}(x)$ insertions. In this case, the hash table achieves nearly optimal scaling.

Corollary 5. *If $R = n/\lg^c x$ for a sufficiently large positive constant c, then*

$$I = \tilde{\Theta}(x).$$

Moreover, $Q = \Theta(x)$.

Our final corollary considers the question of whether it is possible to select a value of R that allows for both I and Q to be $O(x)$. The corollary establishes that no such R exists.

Corollary 6. *For every choice of R, there exists S such that either $I = \omega(x)$ or $Q = \omega(x)$.*

Proof. If $n/R = \omega(1)$, then (11) tells us that there exists a sequence of operations for which I is $\omega(x)$. On the other hand, if $n/R = o(\lg\lg x)$, then (13) tells us that there exists a sequence of operations for which Q is $\omega(x)$. □

Up until now, we have been focusing on a hovering workload. Our final result considers an arbitrary workload of operations, where the only constraint is that the load factor never exceeds $1-1/x$. Notice that if R is small (i.e., $R = \Theta(n/x)$), then ordered linear probing can potentially perform very poorly in the setting, since an entire rebuild window can potentially consist of only insertions, none of which are able to make use of tombstones, but all of which are performed at a load factor of $1 - \Theta(1/x)$. Our next theorem establishes, however, that if R is selected appropriately, then the amortized performance of ordered linear probing is near the optimal $O(x)$ that one could hope to achieve.

Theorem 2. *Let c be a sufficiently large positive constant. Consider an ordered linear probing hash table that uses tombstones for deletions, and that performs rebuilds every $R = n/\lg^c x$ insertions. Finally, consider a sequence of operations S that never brings the load factor above $1 - 1/x$.*

Then the expected amortized cost of each insertion is $\tilde{O}(x)$ and the expected cost of each query/deletion is $O(x)$.

Proof. This follows from the same proof as Theorem 1, except using Proposition 7 instead of Corollary 3. □

Remark. The proofs of Theorems 1 and 2 assume a fully random hash function, but it turns out this assumption is not needed. In particular, one can instead use tabulation hashing, and modify the proofs in the preceding sections as follows: analogues of Lemmas 1 and 2 follow directly from Theorem 8 of [116], and then all of the Chernoff bounds throughout the paper can be re-created using Theorem 1 of [116]. Note that each application of Theorem 8 and Theorem 1 of [116] introduces a $1/\operatorname{poly}(n)$ failure probability, but this is easily absorbed into the analysis.

8 Graveyard Hashing

In this section, we describe and analyze a new variant of linear probing, which we call ***graveyard hashing***. Graveyard hashing takes advantage of the key insight in this paper, which is that tombstones have the ability to significantly improve insertion performance.

Description of graveyard hashing. Graveyard hashing uses different rebuild window sizes, depending on the load factor. If a rebuild is performed at a load factor of $1 - 1/x$, then the next rebuild will be performed $n/(4x)$ operations later.[21]

Whenever the hash table is rebuilt, Graveyard hashing first removes all of the tombstones that are currently present. It then spreads *new tombstones* uniformly across the table. If the current load factor is $1 - 1/x$, then $n/(2x)$ tombstones are created, with one tombstone assigned to each of the hashes $\{2ix \mid i \in [n/(2x)]\}$. The purpose of these tombstones is to help all of the up to $n/(4x)$ insertions that occur between the current rebuild and the next rebuild.

The insertion of tombstones during rebuilds is the *only difference* between graveyard hashing and standard ordered linear probing. Thus insertions, queries, and deletions are implemented exactly as in a traditional ordered linear probing hash table.

If desired, one can implement graveyard hashing so that each rebuild also dynamically resizes the table, ensuring that the load factor is always $1 - \Theta(1/x)$ for some fixed parameter x. Note that, when resizing the table, the elements of the table will need to be assigned to new hashes, and thus will need to be permuted. In the RAM model, this can easily be done in linear time (and in place) using an in-place radix sort. In the external-memory model, resizing can be implemented in $O(n/B)$ block transfers (where B is the external memory block size) using Larson's block-transfer efficient scheme for performing partial expansions/contractions on a hash table [78] (this technique has also been used in past work on external-memory hashing, see [67, 112]).

Analysis of graveyard hashing. To perform the analysis, we will need one last balls-and-bins lemma:

[21] Note that graveyard hashing counts both insertions and deletions as part of the length of a rebuild window.

Lemma 15. *Suppose that μn balls are placed into n bins at random. Let $x > 1$, $k \geq 1$, and $j \in [n]$. With probability $1 - 2^{-\Omega(k)}$, for every interval $I \subseteq [n]$ that contains j, the number of balls in the bins I is at most $(1 + 1/x)|I|\mu + xk$.*

Proof. Suppose there is some interval I satisfying $i \in I$ such that the number of balls in the interval I is greater than $(1 + 1/x)|I|\mu + xk$. If $I = [j_0, j_1]$ for some j_0, j_1, then we can break I into two sub-intervals $I_1 = [j_0, j]$ and $I_2 = (j, j_1]$. Since $(1 + 1/x)|I|\mu + xk$, at least one of the two subintervals $I_k \in \{I_1, I_2\}$ must contain at least

$$(1 + 1/x)|I_k|\mu + xk/2$$

balls. However, by Lemma 2, the probability of any such subinterval I_k existing is at most $2^{-\Omega(k)}$. □

Corollary 7. *Suppose that μn balls are placed into n bins at random. Let $x > 1$, $k \geq 1$, and $j \in [n]$. With probability $1 - 2^{-\Omega(k)}$, for every interval $I \subseteq [n]$ that contains j and has size $|I| \geq x^2 k$, the number of balls in the bins I is at most $(1 + 1/x)|I|\mu$.*

Proof. This follows by applying Lemma 15 with $x' = x/2$ and $k' = kx/2$. □

We now turn our attention to analyzing graveyard hashing. As in the previous sections, it will be easier to analyze the displacement of an insertion rather than directly analyzing the running time of each insertion. Recall that the displacement d_u of an insertion u is $i - h(u)$ where i is the hash of the tombstone that u uses, if u uses a tombstone, and i the position of the free slot that u uses, if u uses a free slot.

The next lemma bounds the difference between displacement and running time. The fact that the rebuild windows for graveyard hashing are so small (only $n/(4x)$ operations) ends up allowing for an especially simple argument.

Lemma 16. *Consider the insertion of an element u. Let d denote the displacement of the insertion, and t denote the running time. Then, for any $r \geq 1$,*

$$\Pr[t - d - 1 \geq rx] \leq \exp(-\Omega(r)).$$

Proof. We can assume without loss of generality that u makes use of some tombstone v (rather than a free slot), since otherwise the lemma is trivial. The displacement of u is therefore given by $d = h(v) - h(u)$ and the running time of u is given by $t = k - h(u) + 1$, where k is the position in which v resides. Thus

$$\Pr[t - d - 1 \geq rx] = \Pr[k - h(v) \geq rx],$$

which means that we want to show that

$$\Pr[k - h(v) \geq rx] \leq \exp(-\Omega(r)).$$

For each element/tombstone v' in the run containing position $h(u)$, define the ***placement-error*** $e_{v'}$ to be $k' - h(v')$, where k' is the position in which v' resides (at the moment of time prior to the insertion u).

We wish to show that $e_v < rx$, but we must be careful about the fact that v is partially a function of the randomness of the hash table. In order to bound e_v, we assume that v is selected adversarially, and instead bound the quantity

$$\beta = \max\{e_{v'} \mid v' \text{ is in the same run as } u, \text{ and } h(v') \geq h(u)\}.$$

We cannot afford to simply union bound over the different options for v' here; instead we must make use of the fact that the values of $e_{v'}$ are closely correlated for different keys v' in the same run.

Let v' be the element for which $\beta = e_{v'}$. Let p be the position of the left-most element in u's run. All of the elements/tombstones that reside in positions $p, \ldots, h(v') + e_{v'}$ must have hashes in $[p, h(v')]$. The number of elements/tombstones (at the time prior to u's insertion) that hash to the interval $I = [p, h(v')]$ must therefore be at least $|I| + e_{v'}$. In contrast, the expected number of elements/tombstones that hash into the interval I is that most $|I|$. Thus there are at least $e_{v'}$ more element/tombstones that hash into I then are expected. By Lemma 15, it follows that $\Pr[e_{v'} \geq rx] \leq \exp(-\Omega(r))$. □

Graveyard hashing is designed so that there are always copious tombstones for insertions to make use of. Note, in particular, that each rebuild window begins with $n/(2x)$ tombstones but only contains at most $n/(4x)$ insertions. This allows for the following bound on displacement.

Lemma 17. *Consider an insertion u. The displacement d of u satisfies*

$$\Pr[d \geq rx] \leq \exp(-\Omega(r))$$

for all $r > 1$.

Proof. Call a tombstone ***primitive*** if it was inserted during the rebuild prior to the current rebuild window. Let T be the set of primitive tombstones present when u is inserted. Let

$$j_0 = \max\{h(v) \mid v \in T, h(v) < h(u)\}$$

and

$$j_1 = \min\{h(v) \mid v \in T, h(v) \geq h(u)\}.$$

The displacement d is at most $j_1 - h(u) \leq j_1 - j_0$. To complete the proof, we will bound the probability that $j_1 - j_0 \geq rx$.

At the beginning of the rebuild window, there were $\frac{j_1 - j_0}{2x} - 1$ primitive tombstones with hashes in the range $I = (j_0, j_1)$; denote the set of these tombstones by L. By the time u is inserted, all of the tombstones L have been used by insertions (this is by the definition of j_0 and j_1). Since there is still a primitive tombstone with hash j_0, the insertions that used up L must have all had hashes at least $j_0 + 1$. Thus, during the current rebuild window, there have been at least $|L| = \frac{j_1 - j_0}{2x} - 1$ insertions that hashed into the interval $I = (j_0, j_1)$.

Recall, however, that each rebuild window consists of only $n/(4x)$ operations. The expected number of insertions that hash into I is therefore a most $\frac{j_1 - j_0}{4x}$.

In summary, the only way have $j_1 - j_0 \geq rx$ is if (1) there is an interval I containing $h(u)$ that satisfies $|I| \geq rx - 2$, and (2) the number q of insertions (during the current rebuild window) that hash into I is a constant factor larger than the expected number $\mathbb{E}[q]$ of such insertions. To bound the probability of such an I existing, we partition the slots of the hash table into "bins" of size x, and treat keys inserted during the rebuild window as balls that each hash to a bin. In order for I to exist, there must be a contiguous subsequence of $\Theta(r)$ bins such that the interval of hashes covered by the bins contains $h(u)$, and such that the bins contain a constant factor more balls than expected. By Corollary 7, the probability of such a subsequence of bins existing is that most $\exp(-\Omega(r))$. □

Combining the previous lemmas, we can analyze the running time of graveyard hashing.

Theorem 3. *Consider a graveyard hash table. For each insertion/query/deletion, if the operation is performed at some load factor of $1 - 1/x$ then the operation takes expected time $O(x)$, and incurs amortized time $O(x)$ for rebuilds.*

Proof. Since graveyard hashing uses small rebuild windows (i.e., of size $n/(4x)$), we can analyze queries by ignoring the deletions in the rebuild window, and applying the classic $O(x)$ bound for query time in an insertion-only table (Proposition 2). Deletions of keys u take the same amount of time that a query for that key would have, so deletions also take expected time $O(x)$. To analyze insertions, we can apply Lemmas 16 and 17, which together bound the expected time by $O(x)$.

Finally, we analyze the amortized cost of rebuilds per operation. If a rebuild window starts at a load factor of $1 - 1/x$, then the next rebuild is performed after $\Theta(n/x)$ operations, and all of those operations are performed at load factors $1 - \Theta(1/x)$. The rebuild can be performed in time $\Theta(n)$ and thus the amortized cost per operation is $\Theta(x)$. □

Remark. The proof of Theorem 3 assumes a fully random hash function, but this assumption is not necessary. The theorem continues to hold if we use either tabulation hashing or 5-independent hashing. In particular, one can use Equation (23) from [116] as a substitute for Lemma 15, and then re-create all of the proofs above without modification.

We conclude the section by analyzing graveyard hashing in the external-memory model [2].

Theorem 4. *Consider the external memory model with $B = rx$ for some $r \geq 1$ and $M = \Omega(B)$. Graveyard hashing can be implemented to offer the following guarantee on any sequence of operations. The load factor of the table is maintained to be $1 - 1/\Theta(x)$ at all times, and each operation incurs*

$$1 + O(1/r)$$

block transfers in expectation. Furthermore, the amortized block-transfer cost (per operation) of rebuilds is $O(1/r)$.

Proof. By Theorem 3, the expected time taken by a given operation is $O(x)$. It follows that the expected number of block boundaries that are crossed by the operation is $O(x/B) = O(1/r)$. Thus the expected number of block transfers incurred is $O(1/r)$.

Next we analyze the cost of rebuilds. Each rebuild window contains $\Theta(n/x)$ operations at a load factor of $1 - \Theta(1/x)$, and, as discussed earlier, the rebuild at the end of the window can be implemented with $O(n/B)$ block transfers. This implies the desired bound of $O(1/r)$ on amortized rebuild cost. □

Corollary 8. *If $x = o(B)$, then the amortized expected number of block transfers per operation is $1 + o(1)$.*

9 Related Work

Alternative probing methods. Beginning in the late 1960s, there was a flurry of work on alteratives to linear probing. A central question has been whether one can have probe sequences that benefit from the data locality of traditional linear probing while also eliminating primary clustering.

At one extreme is uniform probing, where each probe is to a random location, thus sacrificing locality in the probe sequence [119]. There has been intensive work in analyzing variations on uniform probing, including in the presence of deletions [28, 68, 70, 79, 93, 94, 123, 137].

Double hashing [21, 60, 75, 85, 86, 93, 143] is a classic alternative to uniform probing, in which a primary hash function determines the first probe and a secondary hash function determines jump size between indices in the probe sequence. Double hashing has been shown to have short probe sequences similar to that of uniform hashing, but like uniform probing, these short probe sequences come with a corresponding loss in locality.

In 1968, Maurer [62, 89] introduced quadratic probing, which remains a widely used solution today (see, e.g., [1, 25]). Although Maurer's original scheme used a probing sequence that cycled after $n/2$ iterations, subsequent work [62] has shown how to construct quadratic probing sequences that hit every position in the table.

Quadratic probing can be viewed as a hybrid of linear probing and double hashing, maintaining some of the spatial locality of the former, while empirically obtaining probe complexity similar to the latter [89].

Cuckoo hashing. In addition to probing and chaining, another form of hash table that has become widely used (see, e.g., [81,120,132]) is cuckoo hashing, which was introduced in 2004 by Pagh and Rodler [109,110]. Cuckoo hashing guarantees that every record u is in one of two positions $h_1(u)$ or $h_2(u)$ in the hash table. The result is that, even in the worst case, queries take time $O(1)$. Although originally cuckoo hashing only supported load factors smaller than $1/2$, subsequent work has shown how to support higher load factors by either (1) using $d > 2$ hash functions [52], or (2) hashing records u to bins $h_1(u)$ and $h_2(u)$ (rather than to slots) that each have some capacity $c > 1$ [43].

A drawback of cuckoo hashing is that it is less I/O efficient than linear probing. Negative queries always require $d \geq 2$ I/Os, and insertions at high load factors require $\omega(1)$ I/Os [43, 52]. In contrast, we show that linear probing can offer $1 + o(1)$ I/Os per operation, as long as the memory block size is $\omega(x)$.

Other work on hashing. Broadly speaking, work on hashing can be categorized into three major categories. The first, discussed above, has been to understand and try to improve the behavior of three core hash-table designs: probing, cuckoo hashing, and, to a lesser extent, chaining. The second direction has been to study what types of additional features are possible to achieve in a hash table (or, more generally, a dictionary). And the third direction has been to construct explicit families of hash functions that can be used in place of full independence.

As an example of a major result in the second category, Dietzfelbinger and Meyer auf der Heide [40] showed that it is possible to achieve *worst case* constant time operations (with high probability), building on prior work by Fredman et al. [53] and by Dietzfelbinger et al. [41]. Subsequent work has pushed this guarantee even further, showing that it is also possible to have a $1 - o(1)$ load factor [5, 39, 82], as well as a sub-polynomial failure probability [56, 57]. There has also been a series of work on upper and lower bounds for deterministic dictionaries [61, 108, 117, 125, 133] and external-memory hashing [32, 63, 67, 111, 112], as well as work on achieving security guarantees such as history independence [11, 18, 100, 101] and protection against an adversary that can see where in memory is being accessed [30, 31].

In the third category, that is, the study of explicit hash function families, there are now some quite universal results known, including families of hash functions [44, 69, 84, 99, 104] that can be made compatible with essentially any hash table (see, for example, the usage in [6, 82]).

Additionally, there has been quite a bit of work on families of hash functions for specific hash tables, especially linear probing [96, 106, 107, 115, 116, 134] and cuckoo hashing [8, 9, 42, 116, 135]. Linear probing, in particular, has been shown to be compatible with several especially simple families: Pagh et al. [106, 107] showed that *any* 5-independent family suffices, and Pătraşcu And Thorup [116] showed that the family of simple tabulation hash functions also suffices. (In Section 8, we describe how these results can be applied to graveyard hashing as well.) Although linear probing is, in general, *not* compatible with all 4-independent families of hash functions [115], one of the most surprising results in the area is that of Mitzenmacher and Vadhan [96], which establishes that, in any workload with sufficient entropy, even 2-independence suffices.

Relationship with filters. One of the most widely used applications of hash tables at high load factors is for the construction of ***filters***, which are compact dictionaries that supports some false-positive rate ε. The classic filter is the Bloom filter [19], which has inspired numerous variants [3, 20, 24, 27, 36, 49, 83, 121, 122]. Whereas the Bloom filter supports only a limited set of operations (no deletions) and no resizing, modern filters have shown how to adapt space-efficient hash tables in order to construct practical space-efficient filters that support a richer set of operations. Quotient filters and variants (e.g., counting and vector) [15, 15, 46, 55, 113, 114] are built on the idea of storing small fingerprints via ordered linear probing [4]; and cuckoo [48] and Morton filters [22] are based on the idea of storing these fingerprints via cuckoo hashing [109, 110]. There has also been an effort to push forward the theoretical frontiers of what guarantees a filter can offer (see, e.g., Pagh's single-hash filter [105] as well as more recent results [14, 82]).

The fact that many filters are implemented on top of hash-table designs means that improvements to hash table performance directly results in improvements to filter performance. For example, our techniques for improving linear probing can immediately be applied to linear-probing based filters [15,46,55,113,114].

Relationship with other data structures. One of the interesting features of our results in that it reveals an unexpected connection between the linear probing and several other problems in data structures (e.g., list labeling [26], file maintenance [17,64,65,148–150], cache-oblivious and locality-preserving B-trees [12,13,23], and even sorting [16]). Solutions to these problems all share a commonality, which is that they strategically leave extra space between elements of a data structure in order to enable fast modifications. One interesting aspect of linear probing is that this "extra space" (tombstones and free slots) *already* naturally occurs spread throughout the table, but that different forms of extra space (i.e., tombstones versus free slots) end up interacting very differently with operations of the hash table.

References

[1] Abseil, 2017. Accessed: 2020-11-06. URL: https://abseil.io/.

[2] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. doi:10.1145/48529.48535.

[3] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable Bloom filters. *Journal of Information Processing Letters*, 101(6):255–261, March 2007. doi:10.1016/j.ipl.2006.10.007.

[4] Ole Amble and Donald Ervin Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, January 1974. doi:10.1093/comjnl/17.2.135.

[5] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP 2009)*, volume 5555 of *Lecture Notes in Computer Science*, pages 107–118, 2009. doi:10.1007/978-3-642-02927-1_11.

[6] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proceedings of the 51st Annual Symposium on Foundations of Computer Science (FOCS 2010)*, pages 787–796, Las Vegas, Nevada, USA, 23–26 October 2010. doi:10.1109/FOCS.2010.80.

[7] Attractive Chaos Blog. Deletion from hash tables without tombstones, December 2019. Accessed 22-May-2021. URL: https://attractivechaos.wordpress.com/2019/12/28/deletion-from-hash-tables-without-tombstones/.

[8] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and efficient hash families suffice for cuckoo hashing with a stash. *Algorithmica*, 70(3):428–456, November 2014. doi:10.1007/s00453-013-9840-x.

[9] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. A simple hash class with strong randomness properties in graphs and hypergraphs. arXiv:1611.00029, 31 October 2016.

[10] Daniel Bauer. Columbia COMS W3134: Data structures in Java — Lecture 12: Introduction to hashing, October 2015. URL: http://www.cs.columbia.edu/~bauer/cs3134-f15/slides/w3134-1-lecture12.pdf.

[11] Michael A. Bender, Jonathan W. Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'16)*, pages 289–302, San Fransisco, California, USA, 26 June–1 July 2016. doi:10.1145/2902251.2902276.

[12] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 399–409, Redondo Beach, California, 12–14 November 2000. doi:10.1109/SFCS.2000.892128.

[13] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 53(2):115–136, November 2004. doi:10.1016/j.jalgor.2004.04.014.

[14] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *Proceedings of the 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2018)*, pages 182–193, Paris, France, October 2018. doi:10.1109/FOCS.2018.00026.

[15] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kaner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012. doi:10.14778/2350229.2350275.

[16] Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is $O(n \lg n)$. *Theory of Computing Systems*, 39(3):391–397, June 2006. Special Issue on *FUN '04*. doi:10.1007/s00224-005-1237-z.

[17] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. File maintenance: When in doubt, change the layout! In *Procedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 1503–1522, Barcelona, Spain, 16–19 January 2017. doi:10.1137/1.9781611974782.98.

[18] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007)*, pages 272–282, Providence, Rhode Island, USA, 21–23 October 2007. doi:10.1109/FOCS.2007.36.

[19] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, June 1970. doi:10.1145/362686.362692.

[20] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting Bloom filters. In *Proceedings of the 14th European Symposium on Algorithms (ESA 2006)*, pages 684–695, Zurich, Switzerland, 11–13 September 2006. doi:10.1007/11841036_61.

[21] Richard P. Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, February 1973. doi:10.1145/361952.361964.

[22] Alex D Breslow and Nuwan S Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018. URL: https://www.vldb.org/pvldb/vol11/p1041-breslow.pdf.

[23] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 39–48, San Francisco, California, USA, 6–8 January 2002. doi:doi/10.5555/545381.545386.

[24] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 31 December 2003. doi:10.1080/15427951.2004.10129096.

[25] Nathan Bronson and Xiao Shi. Open-sourcing F14 for faster, more memory-efficient hash tables, 25 April 2019. Accessed: 2020-11-06. URL: https://engineering.fb.com/2019/04/25/developer-tools/f14/.

[26] Jan Bulánek, Michal Koucký, and Michael Saks. Tight lower bounds for the online labeling problem. In *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC'12)*, pages 1185–1198, New York, New York, USA, 19–22 May 2012. doi:10.1145/2213977.2214083.

[27] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Christian A Lang, and Kenneth A Ross. Buffered Bloom filters on solid state storage. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS2010) (Part of VLDB)*, pages 1–8, Singapore, 13 September 2010. URL: http://www.vldb.org/archives/workshop/2010/proceedings/files/vldb_2010_workshop/ADMS_2010/adms10-canim.pdf.

[28] Pedro Celis and John V. Franco. The analysis of hashing with lazy deletions. *Information Sciences*, 62(1-2):13–26, January 1992. doi:10.1016/0020-0255(92)90022-Z.

[29] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin Hood hashing (preliminary report). In *26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 281–288, Portland, Oregon, USA, 21–23 October 1985. doi:10.1109/SFCS.1985.48.

[30] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *Advances in Cryptology — ASIACRYPT 2017 — 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, volume 10624 of *Lecture Notes in Computer Science*, pages 660–690, 3–7 December 2017. doi:10.1007/978-3-319-70694-8_23.

[31] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *Revised Papers from the 5th International Workshop on Information Hiding (IH'02)*, pages 400–414, Noordwijkerhout, The Netherlands, 7–9 October 2002. doi:10.5555/647598.732027.

[32] Alexander Conway, Martín Farach-Colton, and Philip Shilane. Optimal hashing in external memory. In *Proceedings 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:14, Prague, Czech Republic, 9–13 July 2018. doi:10.4230/LIPIcs.ICALP.2018.39.

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, USA, 3rd edition, 2009.

[34] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Education, 2006.

[35] Lilian de Greef. UW CSE 373: Data structures and algorithims — Lecture 7: Hash table collisions, Summer 2017. URL: https://courses.cs.washington.edu/courses/cse373/17su/lectures/Lecture%2007%20-%20Hash%20Table%20Collisions.pdf.

[36] Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. BloomFlash: Bloom filter on flash-based storage. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 635–644, Minneapolis, Minnesota, USA, 20–24 June 2011. doi:10.1109/ICDCS.2011.44.

[37] Erik Demaine. MIT 6.897: Advanced data structures – Lecture 10, Spring 2012. URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/calendar-and-notes/MIT6_851S12_L10.pdf.

[38] Erik D. Demaine and Charles E. Leiserson. MIT 6.046J/18.401J: Introduction to algorithms – Lecture 7: Hashing I, October 2005. URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-7-hashing-hash-functions/lec7.pdf.

[39] Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. De dictionariis dynamicis pauco spatio utentibus (*lat.* on dynamic dictionaries using little space). In *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN 2006)*, volume 3887 of *Lecture Notes in Computer Science*, pages 349–361, Valdiva, Chile, 20–24 March 2006. doi:10.1007/11682462_34.

[40] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the Automata, Languages and Programming, 17th International Colloquium (ICALP 1990)*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19, Warwick, United Kingdom, 16–20 July 1990. doi:10.1007/BFb0032018.

[41] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proceedings 29th Annual Symposium on Foundations of Computer Science (FOCS'88)*, pages 524–531, White Plains, New York, USA, 24–26 October 1988. doi:10.1109/SFCS.1988.21968.

[42] Martin Dietzfelbinger and Ulf Schellbach. On risks of using cuckoo hashing with simple universal hash classes. In *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA 2009)*, pages 795–804, New York, New York, USA, 4–6 January 2009. doi:10.5555/1496770.1496857.

[43] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, June 2007. doi:10.1016/j.tcs.2007.02.054.

[44] Martin Dietzfelbinger and Philipp Woelfel. Almost random graphs with simple hash functions. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC'03)*, pages 629–638, San Diego, California, USA, 9–11 June 2003. doi:10.1145/780542.780634.

[45] Adam Drozdek and Donald L. Simon. *Data Structures in C*. PWS, Boston, Massachusetts, USA, 1995.

[46] Gil Einziger and Roy Friedman. Counting with TinyTable: Every bit counts! In *Proceedings of the 17th International Conference on Distributed Computing and Networking (ICDCN '16)*, pages 27:1–27:10, Singapore, Singapore, 4–7 January 2016. doi:10.1145/2833312.2833449.

[47] Jeff Erickson. UIUC CS473: Algorithms — Lecture 5: Hash tables, 2017. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf.

[48] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT'14)*, pages 75–88, Sidney, Australia, 2–5 December 2014. doi:10.1145/2674005.2674994.

[49] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, June 2000.

[50] Gene Fisher. CalPoly CSC103: Fundamentals of computer science – hashing, 2001. URL: http://users.csc.calpoly.edu/~gfisher/classes/103/lectures/week5.2.html.

[51] Philippe Flajolet, Patricio Poblete, and Alfredo Viola. On the analysis of linear probing hashing. *Algorithmica*, 22(4):490–515, December 1998. doi:10.1007/PL00009236.

[52] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, December 2005. doi:10.1007/s00224-004-1195-x.

[53] Michael L. Fredman, Janos Komlos, and Endre Szemeredi. Storing a sparse table with $O(1)$ worst case access time. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS'82)*, pages 165–169, 3–5 November 1982. doi:10.1109/SFCS.1982.39.

[54] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*, pages 285–297, 17–19 October 1999. doi:10.1109/SFFCS.1999.814600.

[55] Afton Geil, Martin Farach-Colton, and John D Owens. Quotient filters: Approximate membership queries on the GPU. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–462, Vancouver, British Columbia, Canada, 21–25 May 2018. doi:10.1109/IPDPS.2018.00055.

[56] Michael T Goodrich, Daniel S Hirschberg, Michael Mitzenmacher, and Justin Thaler. Fully de-amortized cuckoo hashing for cache-oblivious dictionaries and multimaps. arXiv:1107.4378, July 2011.

[57] Michael T Goodrich, Daniel S Hirschberg, Michael Mitzenmacher, and Justin Thaler. Cache-oblivious dictionaries and multimaps with negligible failure probability. In *First Mediterranean Conference on Algorithms (MedAlg 2012)*, volume 7659 of *Lecture Notes in Computer Science*, pages 203–218, Kibbutz Ein Gedi, Israel, 3–5 December 2012. Springer. doi:10.1007/978-3-642-34862-4_15.

[58] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design and Applications*. Wiley, Hoboken, New Jersey, USA, 2015.

[59] David Gries and Doug James. Cornell CS210: Object-oriented programming and data structures — recitation week 8: Hashing, Fall 2014. URL: https://www.cs.cornell.edu/courses/cs2110/2014fa/recitations/recitation08/HashPresentation.pptx.

[60] Leo J. Guibas and Endre Szemerédi. The analysis of double hashing. *Journal of Computer and System Sciences*, 16(2):226–274, April 1978. doi:10.1016/0022-0000(78)90046-6.

[61] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):69–85, October 2001. doi:https://doi.org/10.1006/jagm.2001.1171.

[62] F. R. A. Hopgood and J. Davenport. The quadratic hash method when the table size is a power of 2. *The Computer Journal*, 15(4):314–315, 1972. doi:10.1093/comjnl/15.4.314.

[63] John Iacono and Mihai Pătraşcu. Using hashing to solve the dictionary problem. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 570–582, Kyoto, Japan, 17–19 January 2012.

[64] Alon Itai and Irit Katriel. Canonical density control. *Informatino Processing Letters*, 104(6):200–204, December 2007. doi:10.1016/j.ipl.2007.07.001.

[65] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Proceedings 8th International Colloquium on Automata, Languages, and Programming (ICALP 1981)*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, July 1981. doi:10.5555/646235.682700.

[66] Svante Janson and Alfredo Viola. A unified approach to linear probing hashing with buckets. *Algorithmica*, 75(4):724–781, August 2016. doi:10.1007/s00453-015-0111-x.

[67] Morten Skaarup Jensen and Rasmus Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, November 2008. doi:10.1007/s00453-007-9155-x.

[68] Rosa M. Jiménez and Conrado Martínez. On deletions in open addressing hashing. In *Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 23–31, New Orleans, Louisana, USA, 8–9 January 2018. SIAM. doi:10.1137/1.9781611975062.3.

[69] Eyal Kaplan, Moni Naor, and Omer Reingold. Derandomized constructions of k-wise (almost) independent permutations. *Algorithmica*, 55(1):113–133, September 2009. doi:10.1007/s00453-008-9267-y.

[70] Claire M. Kenyon and Jeffrey Scott Vitter. Maximum queue size and hashing with lazy deletion. *Algorithmica*, 6(4):597–619, June 1991. doi:10.1007/BF01759063.

[71] Gregory Kesden. CMU 15-310: System-level software development — hashing review, 2007. Accessed 31-May-2021. URL: https://www.andrew.cmu.edu/course/15-310/applications/ln/hashing-review.html.

[72] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Addison-Wesley, Boston, Massachusetts, USA, 2006.

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

[74] Donald Ervin Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997. URL: https://www.worldcat.org/oclc/312910844.

[75] Donald Ervin Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998. URL: https://www.worldcat.org/oclc/312994415.

[76] Alan G. Konheim and Benjamin Weiss. An occupancy discipline and applications. *SIAM Journal on Applied Mathematics*, 14(6):1266–1274, November 1966. doi:10.1137/0114101.

[77] Robert L. Kruse. *Data Structures and Program Design*. Prentice-Hall Inc, Englewood Cliffs, New Jersey, USA, 1984.

[78] Per-Åke Larson. Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems*, 7(4):566–687, December 1982. doi:10.1145/319758.319763.

[79] Per-Åke Larson. Analysis of uniform hashing. *Journal of the ACM*, 30(4):805–819, October 1983. doi:10.1145/2157.322407.

[80] Harry R. Lewis and Larry Denenberg. *Data Structures and Their Algorithms*. HarperCollins Publishers, New York, New York, USA, 1991.

[81] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, pages 27:1–27:14, Amsterdam, The Netherlands, 14–16 April 2014. doi:10.1145/2592798.2592820.

[82] Mingmou Liu, Yitong Yin, and Huacheng Yu. Succinct filters for sets of unknown sizes. In *Proceedings 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, volume 168 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 79:1–79:19, Saarbrücken, Germany, 8–11 July 2020. doi:10.4230/LIPIcs.ICALP.2020.79.

[83] Guanlin Lu, Biplob Debnath, and David H.C. Du. A forest-structured Bloom filter with flash memory. In *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, Denver, Colorado, USA, 23–27 May 2011. doi:10.1109/MSST.2011.5937232.

[84] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988. doi:10.1137/0217022.

[85] George S. Lueker and Mariko Molodowitch. More analysis of double hashing. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC'88)*, pages 354–359, Chicago, Illinois, USA, 2–4 May 1988. doi:10.1145/62212.62246.

[86] George S. Lueker and Mariko Molodowitch. More analysis of double hashing. *Combinatorica*, 13(1):83–96, March 1993. doi:10.1007/BF01202791.

[87] Michael Main and Walter Savitch. *Data Structures and Other Objects Using C++*. Addison-Wesley, Boston, Massachusetts, USA, 2001.

[88] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European conference on Computer Systems (EuroSys'12)*, pages 183–196, Bern, Switzerland, 10–13 April 2012. doi:10.1145/2168836.2168855.

[89] W. D. Maurer. An improved hash code for scatter storage. *Communications of the ACM*, 11(1):35–38, 1968. doi:10.1145/362851.362880.

[90] Michael McMillan. *Data Structures and Algorithms with JavaScript*. O'Reilly, Sebastopol, California, USA, 2014.

[91] Haim Mendelson and Uri Yechiali. A new approach to the analysis of linear probing schemes. *Journal of the ACM*, 27(3):474–483, July 1980. doi:10.1145/322203.322209.

[92] Shyamal Mitra. UT CS 313E: Elements of software design — hashing, Spring 2021. URL: https://www.cs.utexas.edu/~mitra/csSpring2021/cs313/lectures/hash.html.

[93] Michael Mitzenmacher. More analysis of double hashing for balanced allocations. In *Proceedings of the Thirteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 1–9, Arlington, Virginia, USA, 2016. doi:10.1137/1.9781611974324.1.

[94] Michael Mitzenmacher. A new approach to analyzing Robin Hood hashing. In *Proceedings of the Thirteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 10–24, Arlington, Virginia, USA, 2016. doi:10.1137/1.9781611974324.2.

[95] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2nd edition, 2017.

[96] Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2008)*, pages 746–755, San Francisco, California, USA, 20–22 January 2008. doi:doi/10.5555/1347082.1347164.

[97] Robert Morris. Scatter storage techniques. *Communications of the ACM*, 11(1):38–44, January 1968. doi:10.1145/362851.362882.

[98] Dave Mount. UMD CMSC 420: Data structures — lecture 11: Hashing — handling collisions, Spring 2019. URL: https://www.cs.umd.edu/class/fall2019/cmsc420-0201/Lects/slides-11-hash-collision.pdf.

[99] Moni Naor and Omer Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. *Journal of Cryptology*, 12(1):29–66, 1999. doi:10.1007/PL00003817.

[100] Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In *Proceedings 35th International Colloquium on Automata, Languages, and Programming, (ICALP 2008)*, volume 5126 of *Lecture Notes in Computer Science*, pages 631–642, Reykjavik, Iceland, 7–11 July 2008. doi:10.1007/978-3-540-70583-3_51.

[101] Moni Naor and Vanessa Teague. Anti-persistence: History independent data structures. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing (STOC'01)*, pages 492–501, Hersonissos, Greece, 6–8 July 2001. doi:10.1145/380752.380844.

[102] A. Newell and J. C. Shaw. Programming the Logic Theory Machine. In *Proceedings of the Western Joint Computer Conference: Techniques for Reliability*, Los Angeles, California, USA, 26–28 February 1957. doi:10.1145/1455567.1455606.

[103] Allen Newell and Herbert A. Simon. The logic theory machine—a complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79, September 1956. doi:10.1109/TIT.1956.1056797.

[104] Anna Pagh and Rasmus Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, 2008. doi:10.1137/060658400.

[105] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal Bloom filter replacement. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pages 823–829, Vancouver, British Columbia, Canada, 23–25 January 2005. doi:10.5555/1070432.1070548.

[106] Anna Pagh, Rasmus Pagh, and Milan Ružić. Linear probing with 5-wise independence. *SIAM Review*, 53(3):547–558, 2011. doi:10.1137/110827831.

[107] Anna Pagh, Rasmus Pagh, and Milan Ruzic. Linear probing with constant independence. In *Proceedings 39th Annual ACM Symposium on Theory of Computing (STOC'07)*, pages 318–327, San Diego, California, USA, 11–13 June 2007. doi:10.1145/1250790.1250839.

[108] Rasmus Pagh. Faster deterministic dictionaries. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 487–493, USA, February 2000. doi:10.5555/338219.338595.

[109] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133, University of Aaarhus, Denmark, 28–31 August 2001. Springer. doi:10.1007/3-540-44676-1_10.

[110] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004. doi:10.1016/j.jalgor.2003.12.002.

[111] Rasmus Pagh, Zhewei Wei, Ke Yi, and Qin Zhang. Cache-oblivious hashing. In *Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'10)*, pages 297–304, Indianpolis, Indiana, USA, 6–11 June 2010. doi:10.1145/1807085.1807124.

[112] Rasmus Pagh, Zhewei Wei, Ke Yi, and Qin Zhang. Cache-oblivious hashing. *Algorithmica*, 69(4):864–883, 2014. doi:10.1007/s00453-013-9763-6.

[113] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD'17)*, pages 775–787, Chicago, Illinois, USA, 14–19 May 2017. doi:10.1145/3035918.3035963.

[114] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD'21)*, 20–25 June 2021. To appear.

[115] Mihai Pătraşcu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. In *Proceedings 37th International Colloquium on Automata, Languages, and Programming (ICALP 2010)*, pages 715–726, Bordeaux, France, 6–10 July 2010. Springer. doi:10.1007/978-3-642-14165-2_60.

[116] Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *Journal of the ACM (JACM)*, 59(3):14:1–14:50, June 2012. doi:10.1145/2220357.2220361.

[117] Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proceedings 55th Annual Symposium on Foundations of Computer Science (FOCS 2014)*, pages 166–175, Philadelphia, Pennsylvania, USA, 18–21 October 2014. doi:10.1109/FOCS.2014.26.

[118] Christos Pelekis. Lower bounds on binomial and Poisson tails: An approach via tail conditional expectations. arXiv:1609.06651, September 2016.

[119] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, April 1957. doi:10.1147/rd.12.0130.

[120] Salvatore Pontarelli and Pedro Reviriego. Cuckoo cache: A technique to improve flow monitoring throughput. *IEEE Internet Computing*, 20(4):46–53, July–August 2016. doi:10.1109/MIC.2016.55.

[121] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash- and space-efficient Bloom filters. In *Sixth International Workshop on Experimental and Efficient Algorithms (WEA 2007)*, volume 4525 of *Lecture Notes in Computer Science*, pages 108–121, Rome, Italy, 6–8 June 2007. Springer.

[122] Yan Qiao, Tao Li, and Shigang Chen. Fast Bloom filters and their generalization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(1):93–103, January 2014. doi:10.1109/TPDS.2013.46.

[123] M. V. Ramakrishna. Analysis of random probing hashing. *Information Processing Letters*, 31(2):83–90, 26 April 1989. doi:10.1016/0020-0190(89)90073-2.

[124] Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB – Porceedings of the VLDB Endowment*, 9(3):96–107, November 2015. The 42nd International Conference on Very Large Data Bases, New Delhi, India. URL: https://vldb.org/pvldb/vol9/p96-richter.pdf.

[125] Milan Ružić. Uniform deterministic dictionaries. *ACM Transactions on Algorithms*, 4(1):1–23, March 2008. doi:10.1145/1328911.1328912.

[126] Keith Schwarz. Stanford CS166: Data structures — linear probing, Spring 2021. URL: http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/12/Small12.pdf.

[127] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, USA, 1983.

[128] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, Massachusetts, USA, 1990.

[129] Peter Smith. *Applied Data Structures with C++*. Jones & Bartlett Learning, 2004.

[130] Thomas A. Standish. *Data Structures, Algorithms, and Software Principles in C*. Addision-Wesley, Reading, Massachusetts, USA, 1995.

[131] David G. Sullivan. Harvard CS S-111: Intensive introduction to computer science using Java — unit 9, part 4: Hash tables, Summer 2021. URL: https://sites.fas.harvard.edu/~libs111/files/lectures/unit9-4.pdf.

[132] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. SmartCuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 553–565, Santa Clara, California, USA, July 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/sun.

[133] Rajamani Sundar. A lower bound for the dictionary problem under a hashing model. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 612–621, San Juan, Puerto Rico, USA, October 1991. doi:10.1109/SFCS.1991.185427.

[134] Mikkel Thorup. String hashing for linear probing. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA 2009)*, pages 655–664, New York, New York, USA, January 2009. doi:10.1137/1.9781611973068.72.

[135] Mikkel Thorup. Fast and powerful hashing using tabulation. *Communications of the ACM*, 60(7):94–101, July 2017. doi:10.1145/3068772.

[136] Jean-Paul Tremblay and Paul G. Sorenson. *An Introduction to Data Structures with Applications*. McGraw-Hill, 1984.

[137] Christopher J. Van Wyk and Jeffrey Scott Vitter. The complexity of hashing with lazy deletion. *Algorithmica*, 1(1–4):17–29, November 1986. doi:10.1007/BF01840434.

[138] Alfredo Viola. Exact distribution of individual displacements in linear probing hashing. *ACM Transactions on Algorithms (TALG)*, 1(2):214–242, October 2005. doi:10.1145/1103963.1103965.

[139] Alfredo Viola. Distributional analysis of the parking problem and Robin Hood linear probing hashing with buckets. *Discrete Mathematics and Theoretical Computer Science (DMTCS)*, 12(2), January 2010. URL: https://dmtcs.episciences.org/519.

[140] Alfredo Viola and Patricio V. Poblete. The analysis of linear probing hashing with buckets (extended abstract). In *Algorithms — ESA '96, Fourth Annual European Symposium*, volume 1136 of *Lecture Notes in Computer Science*, pages 221–233. Springer, 1996. doi:10.1007/3-540-61680-2_58.

[141] Mark Allen Weiss. *Data Structures and Problem Solving using C++*. Addison-Wesley, Reading, Massachusetts, USA, 2000.

[142] Jay Wengrow. *A Common-Sense Guide to Data Structures and Algorithms*. The Pragmatic Programmers, 2017.

[143] Wikipedia contributors. Double hashing, 2021. Accessed 31-May-2021. URL: https://en.wikipedia.org/wiki/Double_hashing.

[144] Wikipedia contributors. Linear probing, 2021. Accessed 31-May-2021. URL: https://en.wikipedia.org/wiki/Lienar_probing.

[145] Wikipedia contributors. Linked list, 2021. Accessed 22-May-2021. URL: https://en.wikipedia.org/wiki/Linked_list.

[146] Wikipedia contributors. Primary clustering, 2021. Accessed 22-May-2021. URL: https://en.wikipedia.org/wiki/Primary_clustering.

[147] Wikipedia contributors. Quadratic probing, 2021. Accessed 31-May-2021. URL: https://en.wikipedia.org/wiki/Quadratic_probing.

[148] Dan E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 114–121, San Francisco, California, USA, May 1982. doi:0.1145/800070.802183.

[149] Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD'86)*, pages 251–260, Washington, DC, USA, May 1986. doi:10.1145/16894.16879.

[150] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, April 1992. doi:10.1016/0890-5401(92)90034-D.

[151] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1976.

[152] Niklaus Wirth. *Algorithms and Data Structures*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1986.

A Linear Probing in Textbooks and Courses

Textbooks' Stances on Linear Probing

Source	Teaches primary clustering	Teaches Knuth's formulae	Recommends (QP=quadratic probing) (DH=double hashing)
Cormen, Leiserson, Rivest, and Stein [33]	yes	no	QP or DH
Dasgupta, Papadimitriou, and Vazirani [34]	no	no	not applicable
Drozdek and Simon [45]	yes	yes	QP or DH
Goodrich and Tamassia [58]	yes	no	$\leq 50\%$ load factor
Kleinberg and Tardos [72]	no	no	not applicable
Kruse [77]	yes	yes	chaining
Lewis and Denenberg [80]	yes	yes	DH with ordered probing
Main and Savitch [87]	yes	yes	DH
McMillan [90]	yes	no	chaining
Sedgewick [127, 128]	yes	yes	chaining or DH
Standish [130]	yes	yes	not prescriptive
Tremblay and Sorenson [136]		yes	DH
Weiss [141]	yes	yes	QP
Wengrow [142]	no	no	$\leq 70\%$ load factor
Wikipedia [144]	yes	yes	QP or DH
Wirth [151, 152]	yes	partly	search trees
Wirth [151, 152]	yes	partly	search trees

Some Course Notes' Stances on Linear Probing

Source	Teaches primary clustering	Teaches Knuth's formulae	Recommends
CMU Systems [71]	yes	no	QP or DH
CalPoly Fundamentals of CS [50]	yes	no	QP or DH
Columbia Data Structs in Java [10]	yes	no	QP or DH
Cornell Prog. and Data Structs [59]	partly	partly	QP
Harvard Intro. to CS [131]	yes	no	QP or DH
MIT Advanced Data Structs [37]	yes	no	low load factor
MIT Intro. to Algorithms [38]	yes	no	DH
Stanford Data Structs [126]	yes	yes	chaining or low load factor
UIUC Algorithms [47]	yes	no	binary probing
UMD Data Structs [98]	yes	yes	DH
UT Software Design [92]	yes	no	$\leq 2/3$ load factor
UW Data Structs and Algorithms [35]	yes	yes	QP or DH