# Tight Bounds for Classical Open Addressing

Michael A. Bender*
Stony Brook University and RelationalAI

William Kuszmaul†
CMU

Renfei Zhou‡
CMU

**Abstract**

We introduce a classical open-addressed hash table, called *rainbow hashing*, that supports a load factor of up to $1-\varepsilon$, while also supporting $O(1)$ expected-time queries, and $O(\log\log\varepsilon^{-1})$ expected-time insertions and deletions. We further prove that this tradeoff curve is optimal: any classical open-addressed hash table that supports load factor $1-\varepsilon$ must incur $\Omega(\log\log\varepsilon^{-1})$ expected time per operation.

Finally, we extend rainbow hashing to the setting where the hash table is *dynamically resized* over time. Surprisingly, the addition of dynamic resizing does not come at any time cost—even while maintaining a load factor of $\geq 1-\varepsilon$ at all times, we can support $O(1)$ queries and $O(\log\log\varepsilon^{-1})$ updates.

Prior to our work, achieving any time bounds of the form $o(\varepsilon^{-1})$ for all of insertions, deletions, and queries simultaneously remained an open question.

## 1 Introduction

Open-addressing is a simple framework for hash-table design that captures many of the most widely-used hash tables in practice (e.g., linear probing, quadratic probing, double hashing, cuckoo hashing, graveyard hashing, Robin-Hood hashing, etc). What these hash tables share in common, and indeed, what makes them examples of open addressing, is that in each case:

1. The data structure itself is just an array of some size $N$ containing elements, free slots, and (in some hash-table designs) tombstones.[1]
2. Each element $x$ has a ***probe sequence*** $h_1(x), h_2(x), \ldots \in [N]$ that is fully determined by $x$, $N$, and random bits (i.e., hash functions).
3. And the procedure for querying $x$ is to simply examine the array positions $h_1(x), h_2(x), \ldots$ either until $x$ is found or until the query is able to conclude, based on what it has seen, that $x$ is not present.

In the decades since open addressing was first introduced, there have been dozens (or possibly even hundreds) of hash-table designs proposed within the open-addressing model. However, the most basic question that one could have remained open: *What is the best space-time tradeoff that any open-addressed hash table can achieve?*

*Partially supported by NSF grants CCF 2247577 and CCF 2106827 and John L. Hennessy Chaired Professorship.
bender@cs.stonybrook.edu.
†Partially supported by a Harvard Rabin Postdoctoral Fellowship and by a Harvard FODSI fellowship under NSF grant DMS-2023528.
kuszmaul@cmu.edu.

‡renfeiz@andrew.cmu.edu.

[1] In some hash tables, when an element is deleted, it is replaced with a ***tombstone***. Then, once there are sufficiently many tombstones in the hash table, they are all removed at once in a single batch, and the hash table is rebuilt. The constructions in this paper will not make use of tombstones, but we include hash tables that do in our discussions of prior work.

**The space vs. time tradeoff.** A hash table is said to support a ***load factor*** of $1-\varepsilon$ if it supports sequences of insertions/deletions/queries with up to $\lceil(1-\varepsilon)N\rceil$ elements present at a time. As $\varepsilon$ decreases, the space efficiency improves, but the time per operation gets worse. The ***space-time tradeoff*** for a hash table is the relationship between $\varepsilon$ and the expected time for each type of operation (insertions, deletions, and membership queries).

In some hash-table constructions, the hash table supports ***dynamic resizing***, meaning that the hash table changes the size $N$ of the array that it uses, over time, in order to maintain a load factor of at least $1-\varepsilon$ at all times. Hash tables that do not do this (i.e., that use a fixed $N$) are referred to as having a ***fixed-capacity***. We will be interested in both fixed-capacity and dynamically-resized hash tables in this paper.

**The historical barrier: Achieving $o(\varepsilon^{-1})$-time operations.** It is relatively straightforward to construct an open-addressed hash table that achieves $O(\varepsilon^{-1})$ expected-time operations (e.g., by using uniform probing with tombstones [25]). With more sophisticated techniques [9, 18, 31, 16, 13], one can achieve expected query time $o(\varepsilon^{-1})$ while supporting insertions and deletions in $f(\varepsilon) = \Omega(\varepsilon^{-1})$ time for some $f$. In fact, if all that one cares about are *positive* queries (and if one disregards insertion/deletion time entirely), then the expected query time can even be reduced to $O(1)$ [9, 18, 31].

It has remained an open question whether one might be able to achieve an expected time bound of $o(\varepsilon^{-1})$ for all operations simultaneously. This is not to say that there is no hope. It is *conjectured*, for example, that bucketed cuckoo hashing [13] with buckets of size $\Theta(\sqrt{\varepsilon^{-1}})$ (and using random-walk insertions) can support $\widetilde{O}(\sqrt{\varepsilon^{-1}})$-time operations—but even bounding the insertion time by $f(\varepsilon^{-1})$ for any function $f$ remains an open question [13, 17]. If one further brings down the target time to $o(\sqrt{\varepsilon^{-1}})$ per operation, then there are not even any conjectured solutions. All known solutions that achieve $O(t)$-time queries for some $t \leq \varepsilon^{-1}$ require insertions to spend at least $\Omega(\varepsilon^{-1}/t)$ time even just deciding which free slot to consume [9, 18, 31, 16, 13].

It is worth remarking that there is an additional bottleneck if one wishes to support dynamic resizing. The standard approach to maintaining a dynamic load factor of $\geq 1-\varepsilon$ is to rebuild the hash table every time that $\Theta(\varepsilon n)$ insertions or deletions occur. These rebuilds require $\Omega(n)$ time (even just to read through the entire data structure), thereby contributing at least $\varepsilon^{-1}$ amortized expected cost per insertion/deletion. Even if one could achieve $o(\varepsilon^{-1})$-time operations for a fixed-capacity table, it is not clear whether such a guarantee could be extended to the dynamic-resizing case.

**This paper: Optimal open-addressing.** In this paper, we introduce ***rainbow hashing***, an open-addressed hash table that achieves expected query time $O(1)$ and expected insertion/deletion time $O(\log\log\varepsilon^{-1})$. The name of the data structure refers to the way in which it assigns colors to elements in order to decide the layout of the hash table.

Our second result is an extension of rainbow hashing that supports dynamic resizing without changing the time bounds. That is, even while maintaining the invariant the $N \geq (1-\varepsilon)n$ at all times, the hash table is able to support $O(1)$ expected-time queries and $O(\log\log\varepsilon^{-1})$ expected-time insertions/deletions.

An interesting consequence is what happens at a load factor of 1. Here, our construction yields an open-addressed hash table with constant expected-time queries, with $O(\log\log n)$ expected-time insertions/deletions, and where the hash table is fully compacted at all times—that is, the number of slots $N$ that the hash table uses is always the *same* as the number of elements $n$ that it contains.

Finally, we conclude the paper by proving a matching lower bound: we show that, in any open-

addressed hash table that supports positive queries in $O(\log\log\varepsilon^{-1})$ expected time, the amortized expected time per insertion/deletion must be at least $\Omega(\log\log\varepsilon^{-1})$. At a technical level, our lower bound can be viewed as a (highly nontrivial) extension of the potential-function techniques previously developed in [6] for analyzing the average log-probe-complexity in a hash table.

Interestingly, the lower bound applies not just to insertion/deletion *time* but also to the *number of items* that the insertion/deletion rearranges. Thus the result applies even to hash tables that go beyond 'pure' open addressing, and that store arbitrary amounts of additional metadata to help with the operation of the data structure.

Combined, our results fully resolve the optimal time complexity of open addressing.

**Other related work.** It is not possible to describe the entire body of work on open addressing, so we will focus instead on summarizing the high-level trajectory of the area. Early work [24, 35, 25, 2, 10, 26], spanning the late 1950s through the early 1970s, focused largely on evaluating variations of linear probing [24, 35, 26, 26], quadratic probing [32], and uniform probing [35, 25]. In a significant 1973 breakthrough [9], Brent showed that one could support $O(1)$-expected-time positive queries regardless of $\varepsilon$. Brent's result prompted a large body of work on query-optimized hash tables. This included both lower bounds [42, 1] on restricted classes of hash tables, alternative techniques for obtaining fast queries [18, 31], and in the past few decades, the emergence of *cuckoo hashing* [16, 13, 34]. Cuckoo hashing, when implemented with the appropriate parameters [16, 13], can be used to support $O(\log\varepsilon^{-1})$-time queries, not just in expectation, but even in the *worst case*. Although this time bound is worse than the $O(1)$-bound achieved by Brent and others [9, 18, 31], it is notable for applying to both positive *and* negative queries.

A common feature among many open-addressing schemes is that, even in cases where the hash table's behavior is intuitively easy to understand, it is often quite difficult to analyze. The variants of cuckoo hashing that achieve $O(\log\varepsilon^{-1})$-time queries, for example, are conjectured to also achieve $O(\varepsilon^{-1})$-time insertions [13, 16, 17, 4], but even bounds of the form $\text{poly}(\varepsilon^{-1})$ remain open. Another example is quadratic probing [32, 21], which despite widespread use since the early 1970s [41], has resisted time bounds of the form $f(\varepsilon^{-1})$ for *any* $f$. Other examples, still, include double hashing, the analysis of which remained open for decades [20, 30], and robin-hood hashing [10] (a.k.a. ordered linear probing [2]), which was only very recently revealed to yield much better time bounds than were previously thought to hold [7].

In recent decades, there has also been a great deal of work on hash tables that go beyond the classical open-addressing model. This includes work on succinct data structures [6, 28, 29, 5, 8, 36, 3], hash tables with high-probability time guarantees [19, 27, 5], and dictionaries in the external-memory model [22, 11, 23, 38]. In the area of succinct data structures, in particular, there have been a series of recent breakthroughs [6, 28, 29] that, together, fully characterize the optimal time-space tradeoff curve of any unordered dictionary. Interestingly, many of the data structures introduced in this line of work share a subtle connection to open addressing: they can be viewed as classical open-addressed hash tables in which the probe position of each item $x$ is cached in a secondary data structure. Roughly speaking, this allows one to take an open-addressed hash table in which a key $x$ would have required $t$ time to find, and to instead store $\log t$-bit number $t$ explicitly (in the secondary data structure), so that $x$ can be found in constant time. This connection led Bender et al. [6] to study a variation of open addressing in which the goal is to minimize the average *log* query time over all elements. Not surprisingly, this leads to a very different tradeoff curve than the one in this paper—for example, $O(1)$ log query time corresponds to $O(\log^* n)$-time insertions. Nonetheless, as we show in Section 7, the lower-bound techniques from [6] can actually be extended to our setting, albeit, with several significant changes.

## 2 Preliminaries

**Open addressing.** At a high level, an ***open-addressed hash table*** with ***capacity*** $N$ is a hash table that stores its keys (elements of some universe $U$) in an array of size $N$, and that uses a ***probe-sequence function*** $h(x) = \langle h_1(x), h_2(x), \ldots \rangle \in [N]^\infty$ to perform queries.

The ***state*** of the hash table is an array $A$ of $N$ slots, where each slot either stores a key or is a free slot. Specifically, if $S \subseteq U$ is the current set of keys, then each key $x \in S$ appears exactly once in $A$, and the remaining $N - |S|$ slots are left empty.

In addition to its state $A$, the hash table gets to store the capacity $N$ of the array and the current number $n = |S|$ of elements. The hash table can also invoke a fully random hash function which it is not responsible for storing. These are the only things that the hash table has access to.

Queries $\textsc{Query}(x)$ for a key $x$ are implemented by scanning a probe sequence $h(x) = \langle h_1(x), h_2(x), \ldots \rangle$ of positions in the array, until either $x$ is found or until some stopping condition is met. The probe sequence for $x$ must depend on only $x$, $N$, and random bits (i.e., hash functions). Insertions and deletions are permitted to rearrange the elements in the hash table however they wish, so long as they preserve the correctness of queries.

The ***load factor*** of the hash table is defined to be $n/N$, and is typically denoted by $1 - \varepsilon$. The goal is to design insertion, deletion, and query algorithms that allow for time-efficient operations as a function of $\varepsilon^{-1}$.

**Fixed capacity vs. dynamic resizing.** A classical open-addressing hash table is said to have ***fixed capacity*** if $N$ remains the same over time, and is said to be ***dynamically resized*** if $N$ changes over time (so that, at any given moment, the hash table resides in the first $N$ slots of an infinite array). We will be interested in proving upper bounds for both the fixed-capacity and dynamically-resized cases. Our lower bounds will be for the fixed-capacity case but using inputs in which the total number of elements changes by only $\pm 1$ over time.

When discussing fixed-capacity hash tables, we will aim to support a large maximum load factor. When discussing dynamically-resized hash tables, we will aim to support a continual load factor—that is, as $n$ changes, $N$ will also change to preserve the load factor.

**Other variations.** The flavor of open addressing studied in this paper is the most classical version of the problem—the one typically referred to simply as *open addressing* [33, 25]. However, there are also many other variations that have been studied, some of which have also been referred to as open addressing: notable examples include *greedy* open addressing, where each insertion must use the first free slot it finds [37, 42]; and *non-oblivious* open addressing, where the querier can probe the hash table adaptively rather than using a fixed probe sequence [15, 14]. Thus the term *open addressing* is not always unambiguous in the literature, and to emphasize the fact that we are focusing on standard open addressing (rather than the greedy or non-oblivious variations), we will refer to the class of hash tables that we are studying as ***classical open addressing*** for the rest of the paper.

## 3 The Rainbow Cell

In this section, we describe a simple hash table called the ***rainbow cell***. The rainbow cell operates continuously at a load factor of 1. That is, the hash table is initialized to contain $n$ elements in $n$ slots, and then each update to the hash table both deletes some element and inserts some new

element, so that every slot is still occupied. One could also extend the rainbow cell to support load factors less than 1, but as we shall see, this will not actually be necessary for our applications of it.

What makes the rainbow cell a bit unusual, as a hash table, is that it prioritizes update time over query time. Indeed, whereas updates will take expected time $O(1)$, queries will be permitted to take expected time $O(n^{3/4})$. This may seem counterproductive, given that our final data structure, the rainbow hash table (Section 4), will support constant-time *queries* and (slightly) super-constant-time insertions/deletions. Nonetheless, the rainbow cell will end up serving as a critical building block in constructing the full rainbow hash table.

Finally, for this section, we will assume that deletions *already know the position of the element being deleted*. In many hash tables, including those constructed in later sections, this assumption is without loss of generality, because the expected time to query an element is less than the intended expected deletion time. The assumption is *not* without loss of generality for the rainbow cell, because queries are slower than deletions, but the assumption will turn out to be true (by design) in our applications of rainbow cells later on.

In the rest of the section, we will define how a rainbow cell works and prove the following proposition:

**Proposition 3.1.** *The rainbow cell is a classical open-addressed hash table that operates continuously at load factor* 1*, supports updates in expected time* $O(1)$*, and supports queries in expected time* $O(n^{3/4})$*. Updates consist of a deletion and insertion pair, and assume that the element being deleted is in a position already known (i.e., that we do not need to perform a query to find the element).*

**The data structure.** The rainbow-cell partitions an array into $n^{1/4}$ ***buckets***, each of which has $n^{3/4}$ slots. The final $n^{1/2}$ slots in each bucket will play a special role, and are referred to as ***sky slots***. Across all $n^{1/4}$ buckets, there are $n^{3/4}$ total sky slots.

Each element $x$ will be assigned a status as either ***heavy*** or ***light***. The status is given to $x$ by a ***status hash function*** $s(x)$ that sets $s(x) = \texttt{heavy}$ with probability $1-1/(2n^{1/4})$ and $s(x) = \texttt{light}$ with probability $1/(2n^{1/4})$. If an element $x$ is heavy, then it is also assigned a random bucket $h(x)$.

Whenever possible, the state of the data structure will be as follows: each heavy element $x$ is stored in its assigned bucket $h(x)$, and each light element $y$ is stored in some sky slot of some bucket (any bucket will do). If the elements are stored in this type of configuration, we say the hash table is in a ***common-case configuration***. An example is given in Figure 1.

Assuming a common-case configuration, queries are straightforward to implement: Heavy elements $x$ are queried by scanning the single bucket $h(x)$, and light elements can be queried by scanning the sky slots of all buckets. Both cases result in $O(n^{3/4})$-time queries.

Update operations will keep the data structure in a common-case configuration whenever it is possible to do so. If it is not possible, because some bucket has either fewer than $n^{3/4} - n^{1/2}$ or greater than $n^{3/4}$ (heavy) elements that hash to it, then the hash table is said to have incurred a ***full failure***. Whether or not the hash table is incurring a full failure is indicated to queries by the relative order of the final two elements in each bucket (i.e., whether the final two elements $a$ and $b$ satisfy $a < b$ or $b < a$). If a query observes that a full failure has occurred, then after it has finished the $n^{3/4}$ probes that it would normally perform, it performs $n$ additional probes to check the entire hash table for the element being queried. Thus the query time is $O(n^{3/4})$ whenever a full failure has *not* occurred, and is $O(n)$ whenever a full failure has occurred.

Finally, we must describe how to implement updates so that (1) the hash table is in a common-case configuration whenever possible; (2) the update detects whenever a full failure occurs; and (3) the expected time for the update is $O(1)$.

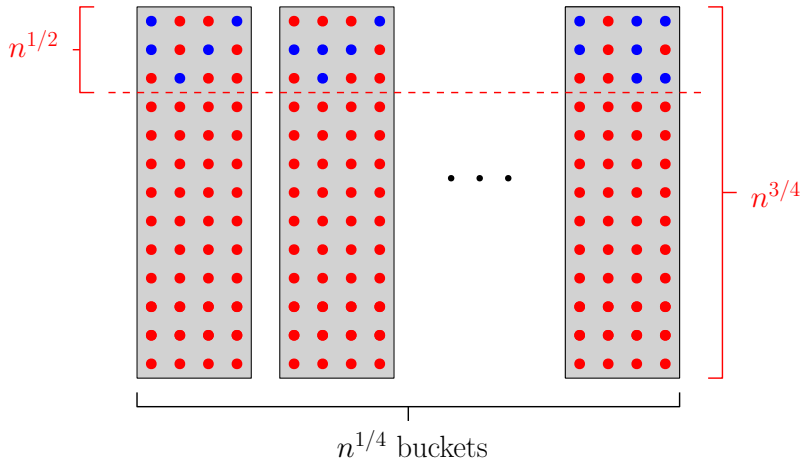


Figure 1: A rainbow cell in a common-case configuration. The red elements are heavy elements, each of which is in their assigned bucket; the blue elements are light elements, each of which is in a sky slot of some bucket.

To describe the update operation, it is helpful to first assume what we call a ***update-friendly input***: such an input is one in which every bucket $j \in [1, n^{1/4}]$ has between $n^{3/4} - \frac{2}{3}\sqrt{n}$ and $n^{3/4} - \frac{1}{3}\sqrt{n}$ heavy items that hash to it. What is nice about update-friendly inputs is that the sky slots in each bucket are are guaranteed to be at least 1/3 heavy items and at least 1/3 light items. This means that if we wish to find a light (resp. heavy) item within the sky slots of some bin, we can do so in $O(1)$ expected time by simply sampling random sky slots in the bin until we find a light (resp. heavy) item. We call this the ***sampling trick***.

Assuming an update-friendly input, we can implement updates as follows. Suppose we wish to delete some item $x$, currently in some bucket $j$, and insert some item $y$. The update can be performed as follows:

- We begin by removing $x$, which creates a free slot $s$ in bucket $j$.
- If $s$ is not a sky slot, then we find some heavy element $x'$ that *is* in a sky slot of bucket $j$ (we can do this in $O(1)$ expected time using the sampling trick). We then move $x'$ to slot $s$, freeing up some sky slot $s'$. For notational convenience, if $s$ was already a sky slot, then we simply define $s' = s$. Thus, at the end of this step, we have created a free sky slot $s'$ in bin $j$.
- If $y$ is a light element, then we complete the update by placing it in slot $s'$. If $y$ is a heavy element, then we need to create a free slot $s''$ in bin $h(y)$ and place $y$ there. To do this, we find a light element $z$ in some sky slot $s''$ of bin $h(y)$ (again, this step takes $O(1)$ expected time using the sampling trick). We then move $z$ to slot $s'$, and place $y$ in slot $s''$. This completes the update, while keeping the hash table in a common-case configuration.

Whenever the hash table is in a common-case configuration, each update will attempt to use the above protocol. If the protocol succeeds, then the hash table will continue to be in a common-case configuration, as desired. The protocol is said to ***fail*** if it runs for time $n$ without completing. In this case, the update swaps to a more naive method: it scans the entire hash table, determines whether we are experiencing a full failure, and rebuilds the entire hash table appropriately. This failure mode guarantees that the update takes worst-case $O(n)$ time.

Finally, whenever the hash table is *not* in a common-case configuration (i.e., it is already experiencing a full failure), the updates default to the $O(n)$-time failure mode: they scan the entire

hash table and rebuild it in whatever way is appropriate based on whether the hash table is still experiencing a full failure or not.

**Analyzing the rainbow cell.** Having described how the rainbow cell works, the analysis follows from a straightforward application of Chernoff bounds.

**Lemma 3.2.** *Each input is update-friendly with probability* $1 - n^{-\omega(1)}$.

*Proof.* The expected number of heavy elements that hash to a given bucket is $n^{3/4} - \frac{1}{2}n^{1/2}$. It follows by a Chernoff bound that, with probability $1 - n^{-\omega(1)}$, the number of such elements will be between $n^{3/4} - \frac{2}{3}\sqrt{n}$ and $n^{3/4} - \frac{1}{3}\sqrt{n}$. Moreover, by a union bound, the probability of this bound failing for any bucket $j$ is at most $n^{3/4} \cdot n^{-\omega(1)} = n^{-\omega(1)}$. □

**Lemma 3.3.** *The expected time to perform an update is* $O(1)$.

*Proof.* If the input is update-friendly, then the sampling trick allows us to complete the update in $O(1)$ expected time. If the input is not update-friendly, then the update may take time as much as $O(n)$. Thus, by Lemma 3.2, the expected update time is at most

$$O(1) + \Pr[\text{non-update-friendly input}] \cdot O(n) = O(1) + n^{-\omega(1)} \cdot O(n) = O(1).$$ □

**Lemma 3.4.** *The expected time to perform a query is* $O(n^{3/4})$.

*Proof.* If the hash table is in a common-case configuration, then the query takes time $O(n^{3/4})$. If the hash table is in a non-common-case configuration, then it is experiencing a full failure, and the query may take time as much as $O(n)$. The probability of a full failure occurring is at most the probability that the input is non-update-friendly. Thus, by Lemma 3.2, the expected update time is at most

$$O(n^{3/4}) + \Pr[\text{non-update-friendly input}] \cdot O(n) = O(n^{3/4}) + n^{-\omega(1)} \cdot O(n) = O(n^{3/4}).$$ □

Since the rainbow cell is, by design, a classical open-addressed hash table that operates at load factor 1, we can combine Lemmas 3.3 and 3.4 to obtain a proof of Proposition 3.1.

# 4 The Rainbow Hash Table

In this section, we describe the most basic version of ***rainbow hashing***. For now, we will confine ourselves to the setting where the hash table operates continually at a load factor of 1 (we will support other load factors later on in Section 6), and where the hash table does not resize (we will add dynamic resizing in Section 5). Subject to these restrictions, we will achieve $O(1)$ expected-time queries and $O(\log \log n)$ expected-time updates.

**Proposition 4.1.** *Basic rainbow hashing is a classical open-addressed hash table that operates continually at load factor* 1*, achieves* $O(1)$ *expected-time queries, and achieves* $O(\log \log n)$ *expected-time updates.*

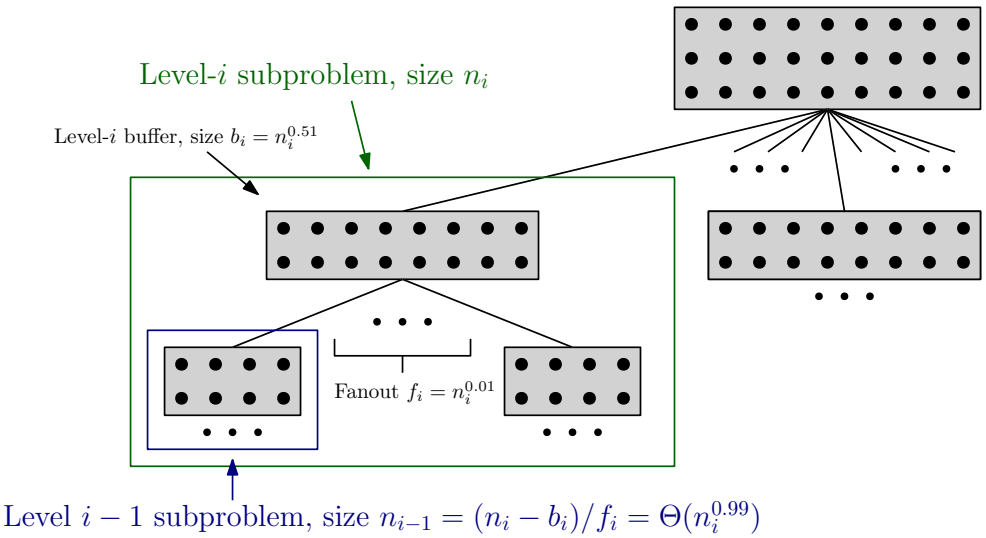


Figure 2: The recursive structure of a rainbow hash table.

**Defining a recursive structure.** To describe basic rainbow hashing, we will think of the array as being broken into the recursive tree structure shown in Figure 2. The root of the tree, which occurs at some level $\overline{\ell} = \Theta(\log \log n)$, is a single subproblem that contains the entire array—its size is denoted by $n_{\overline{\ell}} := n$. The leaves of the tree, which occur at level 1, are subproblems that consist of a buffer of $b_1 = n_1 = O(1)$ elements. For every level $i \in (1, \overline{\ell}]$, the subproblems in level $i$ have the following structure: each level-$i$ subproblem has size $n_i$, and it consists of a ***level-$i$ buffer*** of size $b_i = n_i^{0.51}$ as well as $f_i = n_i^{0.01}$ children that are each level-$(i-1)$-nodes of sizes

$$n_{i-1} := (n_i - b_i)/f_i = \Theta(n_i^{0.99}).$$

Note that the recursive relationship between $n_i$ and $n_{i-1}$ determines all of the values $\{n_i\}, \{b_i\}, \{f_i\}$ in the tree. Finally, it will be helpful to also define $m_1, m_2, \ldots, m_{\overline{\ell}}$ so that $m_i = \Theta(n/n_i)$ is the number of level-$i$ subproblems.

Each item $x$ will hash to a random ***color*** $C(x) \in \{2, 3, \ldots, \overline{\ell}\}$ (we will specify the distribution on $C(x)$ in a moment), and to a uniformly random subproblem $h(x)$ out of the $m_{C(x)}$ subproblems with level $C(x)$. The probability distribution for $C(x)$ is given by $\Pr[C(x) = i] = p_i$ for a set of values $p_2, p_3, \ldots, p_{\overline{\ell}}$ satisfying $\sum_i p_i = 1$ and such that, for $1 < i < \overline{\ell}$,

$$(p_2 + \cdots + p_i)n \in \left(\sum_{j=1}^{i} m_j \cdot b_j\right) - [0.4, 0.6] \cdot m_i \cdot b_i. \tag{1}$$

It may be helpful to think about this inequality at a per-subproblem level: It is equivalent to say that, for a subproblem $s$ in level $1 < i < \overline{\ell}$, if we define $t_s$ to be the sum of the sizes of the buffers of $s$ and all of $s$'s descendants, and we define $q_s$ to be the total number of elements that hash to $s$ and $s$'s descendants, then

$$\mathbb{E}[q_s] = t_s - [0.4, 0.6] \cdot b_i. \tag{2}$$

The use of the range $[0.4, 0.6]$ in (1) and (2) will not be important in this section (we could just as well use the concrete value 0.5), but it will be important later on in Section 5 where, in order to support resizing, we will allow $m_i$ to change over time.

We conclude the definition of the $\{p_i\}$'s by noting their asymptotic relationship to $n_i, b_i$:

**Lemma 4.2.** *For $i > 1$, each $p_i$ satisfies $p_i = \Theta(b_{i-1}/n_{i-1})$.*

*Proof.* For $i = 2$ (or $i = O(1)$), the lemma is trivial. By (1), we have for $2 < i < \overline{\ell}$ that

$$p_i n \in m_i b_i - [0.4, 0.6] \cdot m_i b_i + [0.4, 0.6] \cdot m_{i-1} b_{i-1},$$

and that for $i = \overline{\ell}$,

$$p_i n \in m_i b_i + [0.4, 0.6] \cdot m_{i-1} b_{i-1}.$$

Either way,

$$\begin{aligned} p_i n &= \Theta(m_i b_i) + \Theta(m_{i-1} b_{i-1}) \\ &= \Theta(m_{i-1} b_{i-1}), \end{aligned}$$

where the final step uses the fact that the buffers in level $i-1$ have a larger cumulative size than those in level $i$. It follows that

$$p_i = \Theta\left(\frac{b_{i-1}}{n/m_{i-1}}\right) = \Theta(b_{i-1}/n_{i-1}). \qquad \square$$

**The role of rainbow cells.** For each subproblem $s$ in the tree, we implement $s$'s buffer as a rainbow cell. This means that updates to the buffer can be implemented in $O(1)$ expected time and that queries can be implemented in $O(b_i^{3/4})$ expected time. The fact that queries to $s$'s buffer take time polynomially smaller than $b_i$ will end up being critical to the data structure.

**Separating insertions and deletions.** So that we can discuss insertions and deletions separately, we will allow the hash table to take intermediate states containing a free slot in some (known) position. Deletions create such a free slot, and guarantee that the free slot appears in the buffer of the root subproblem; and insertions consume such a free slot, assuming that it was initially in the buffer of the root subproblem. Thus, several of the definitions that follow (namely the definitions of common-case configurations and of the Common-Case Invariant) should be viewed as applying both in the context where there is a free slot somewhere in the hash table and in the context where there is not.

**High-level overview: the common-case behavior of the data structure.** To motivate the data structure, let us take a moment to describe how the data structure will behave when certain rare (at a per-subproblem level) failure events do not occur. Precluding such failure events, the data structure will look as in Figure 3: for each level-$i$ subproblem $s$, a constant fraction of the elements in $s$'s buffer will be color-$i$ elements that hash to $s$ (unless $s$ is a leaf), and the remaining will be color-$(i+1)$ elements that hash to $s$'s parent (unless $s$ is the root). Notably, all of the elements in the table that hash to $s$ will be stored in either $s$'s buffer or in $s$'s children's buffers.

It follows that, to query an element $x$ that hashes to $h(x) = s$, we can simply query $s$'s buffer and $s$'s children's buffers. Since each of these buffers is implemented as rainbow cells, this takes expected time

$$O\left(b_i^{3/4} + f_i \cdot b_{i-1}^{3/4}\right).$$

This may seem large, but remember that most elements $x$ hash to very low-level nodes $s$. By Lemma 4.2, for $i > 1$, the probability that $x$ hashes to a level-$i$ node is $p_i = \Theta(b_{i-1}/n_{i-1})$ , so the expected time to query an element is

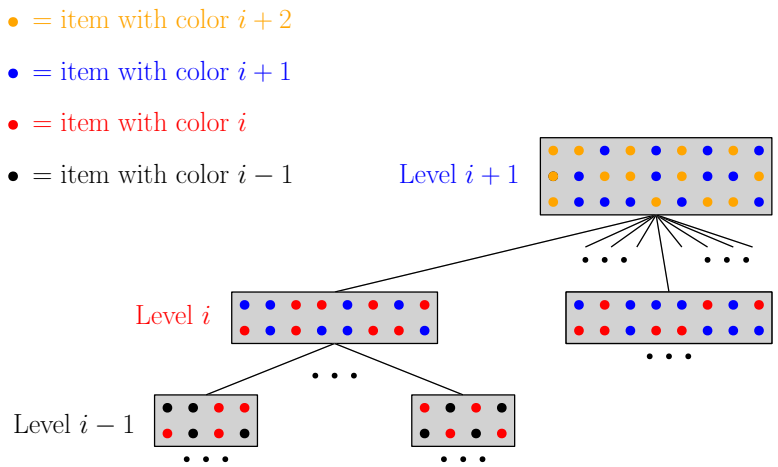


Figure 3: For $i < \overline{\ell}$, the common-case state of a level-$i$ node $s$ will be that a constant fraction of the elements in its buffer have color $i$ (and hash to $s$) and the rest of have color $i+1$ (and hash to $s$'s parent).

$$
\begin{aligned}
&O\left(1 + \sum_{i>1} \frac{b_{i-1}}{n_{i-1}} \cdot \left(b_i^{3/4} + f_i \cdot b_{i-1}^{3/4}\right)\right) \\
&= O\left(1 + \sum_{i>1} \frac{n_i^{0.99 \cdot 0.51}}{n_i^{0.99}} \cdot \left((n_i^{0.51})^{3/4} + n_i^{0.01} \cdot ((n_i^{0.99})^{0.51})^{3/4}\right)\right) \\
&= O\left(1 + \sum_{i>1} \frac{n_i^{0.99 \cdot 0.51}}{n_i} \cdot \left(n_i^{0.01} \cdot (n_i^{0.51})^{3/4}\right)\right) \\
&= O\left(1 + \sum_{i>1} n_i^{0.99 \cdot 0.51 - 1 + 0.01 + 0.51 \cdot 3/4}\right) \\
&= O\left(1 + \sum_{i>1} n_i^{-0.1}\right) \\
&= O(1).
\end{aligned}
$$

Again, this is all assuming that certain rare failure events do not occur, but we will come back to that later.

To implement insertions and deletions, an important insight is that we can make use of a "sampling trick" similar to the one we used in rainbow cells. Suppose that we are looking at a subproblem $s$ (that is not a leaf) and that we wish to find an element $y$ in $s$'s buffer that hashes specifically to $s$. Since a constant fraction of the elements in $s$'s buffer hash to $s$, we can use random sampling to find such an element in $O(1)$ expected time. Likewise, if $s$ is not the root, and if we wish to find an element in $s$ that hashes to $s$'s parent, we can also do this in $O(1)$ expected time using random sampling.

As we perform the insertion/deletion, we need to preserve the invariant that, for each subproblem $s$, all of the elements that hash to $s$ are stored in either $s$'s buffer or in $s$'s children's buffers (again, this ignores certain rare failure events). How can we implement insertions and deletions while preserving this invariant?

Suppose we wish to insert an element $x$ that hashes to some subproblem $h(x)$. We want to place $x$ in the buffer of $h(x)$, but we currently have a free slot in the buffer of the root subproblem $r$. Let $s_1 = h(x), s_2, s_3, \ldots, s_j = r$ be the path from subproblem $h(x)$ to the root subproblem $r$. We can use the sampling trick to find elements $y_1, y_2, \ldots, y_{j-1}$ in the buffers of $s_1, s_2, \ldots, s_{j-1}$ with the properties that $h(y_i) = s_{i+1}$. We can then place $y_{j-1}$ in the free slot in the root's buffer, place $y_{j-2}$ in $y_{j-1}$'s former position, place $y_{j-3}$ in $y_{j-2}$'s former position, and so on, ultimately freeing up the position where $y_1$ resided. Finally, we can put $x$ in this position. With this augmenting-path approach, we can complete the insertion while preserving the invariant that every item appears in the buffers of either the subproblem it hashes to or one of that subproblem's children.

Likewise, suppose we wish to delete an element $x$ that is currently in the buffer of some subproblem $s$. By removing $x$, we create a free slot in $s$'s buffer that we need to move to the root. Let $s_1 = s, s_2, s_3, \ldots, s_j = r$ be the path from subproblem $h(x)$ to the root subproblem $r$. We can use the sampling trick to find elements $y_2, y_3, \ldots, y_j$ in the buffers of $s_2, s_2, \ldots, s_j$ with the properties that $h(y_i) = s_i$. We can then place $y_2$ in the free slot in $s$'s buffer, place $y_3$ in $y_2$'s former position, place $y_4$ in $y_3$'s former position, and so on, ultimately freeing up a slot in $r$'s buffer. Critically, we have once again preserved the invariant that every item appears in the buffers of either the subproblem it hashes to or one of that subproblem's children.

This completes the description of how the data structure would behave if certain rare (at a per-subproblem level) failure events never occurred. The expected time per query would be $O(1)$ and the expected time per update would be $O(\log \log n)$. In general, however, we must be able to handle failure events that break our desired invariants (i.e., that cause the population in the buffer of a node to not simply be a constant fraction of elements that hash to the node and a constant fraction of elements that hash to the parent). Most of the effort in formalizing the data structure will be in designing and maintaining an invariant (that we will call the Common-Case Invariant) that lets us handle these failure events cleanly. We now continue in the rest of the section by presenting and analyzing the full data structure.

**Storing a boolean in each buffer.** We shall assume for the sake of discussion that, for each subproblem $s$, we can store a boolean (called the ***failure indicator***) indicating whether $s$ is in a certain type of failure mode (to be specified later). This boolean will not affect the probe sequence that queries perform (since, after all, queries must be oblivious), but will help queries determine when they can terminate. We emphasize that the failure indicator can easily be encoded implicitly in the relative order of, say, the first two elements of $s$. Thus it is only for ease of discussion that we treat the boolean as being stored explicitly.

**The Common-Case Invariant.** A subproblem $s$ is said to be in a ***weakly common-case configuration*** if:

- For each child $c$ of $s$, all of the elements that hash to $c$ and its descendants are stored in the buffers of $c$ and its descendants.
- The only elements in $s$'s descendants' buffers that do not hash to $s$'s descendants are elements that hash to $s$ and are in $s$'s children's buffers.

The subproblem $s$ is further said to be in a ***strongly common-case configuration*** if both $s$ and $s$'s ancestors are all in weakly common-case configurations. Critically, this implies that all of the elements that hash to $s$ are in the buffers of $s$ and $s$'s children, and that all of the elements in $s$'s buffer that do not hash to $s$ are elements that hash to $s$'s parent.

A subproblem $s$ is said to be ***weakly feasible*** if it is possible to arrange the elements in the tree so that $s$ is in a weakly common-case configuration, *and* so that any free slot in the tree (if we are in an intermediate state between a deletion and an insertion) is not contained in the buffers of $s$'s descendants. The subproblem $s$ is said to be ***strongly feasible*** if all of $s$ and its ancestors are weakly feasible.

Our algorithm will maintain what we call the ***Common-Case Invariant:***

- For each subproblem $s$, the failure indicator correctly identifies whether or not $s$ is strongly feasible.
- If a subproblem $s$ is strongly feasible, then $s$ is in a strongly common-case configuration.

Before continuing, it is worth establishing three lemmas about the Common-Case Invariant:

**Lemma 4.3.** *Suppose that the Common-Case Invariant holds. For a given level-$i$ subproblem $s$, we have with probability $1 - n_i^{-\omega(1)}$ that $s$ is strongly feasible, that $s$ is in a strongly common-case configuration, and that:*

(i) *all of the elements in $s$'s buffer have colors $i$ and $i+1$;*

(ii) *if $s$ is neither the root nor a leaf, then the fraction of elements in $s$'s buffer that have color $i$ (as opposed to color $i+1$) is in the range $[0.3, 0.7]$.*

*Moreover, all of this is true even if we pick $s$ based in part on the hash of some specific element $x$ (this part will be straightforward, since knowing $h(x)$ can only change how many items hash to a given subtree by $\pm 1$).*

We remark that, in Lemma 4.3, we are using $n_i^{-\omega(1)}$ to denote $n_i^{-f(n_i)}$ for some $f \in \omega(1)$, meaning that the $\omega$-notation is in terms of $n_i$ rather than, say, $n$.

*Proof.* For a given level $i$ and level-$i$ subproblem $s$, define $t_s$ to be the sum of the sizes of the buffers of $s$ and all of $s$'s descendants; and define $q_s$ to be the total number of elements that hash to $s$ and $s$'s descendants. By construction, $\mathbb{E}[q_s] \in t_s - b_i \cdot [0.4, 0.6] \pm 1$ (the $\pm 1$ comes from the role of $x$ in the last sentence of the lemma statement). If $1 < i < \overline{\ell}$, then by a Chernoff bound, we know that with probability $1 - n_i^{-\omega(1)}$, we have $|q_s - \mathbb{E}[q_s]| \leq \widetilde{O}(\sqrt{n_i}) < 0.1 b_i$. Thus, if $i < \overline{\ell}$, then we have with probability $1 - n_i^{-\omega(1)}$ that

$$q_x = \begin{cases} 0 & \text{if } i = 1 \\ t_s - b_i \cdot [0.3, 0.7] & \text{otherwise.} \end{cases} \tag{3}$$

We can apply a union bound to conclude that, for any $1 \leq i \leq \overline{\ell}$ and any level-$i$ subproblem $s$, we have with probability $1 - f_i \cdot n_{i-1}^{-\omega(1)} = 1 - n_i^{-\omega(1)}$ that (3) is true for all of $s$'s children (note that level-1 subproblems have $f_1 = 0$ children). Applying another union bound, we can conclude with probability

$$1 - \sum_{j \geq i} n_j^{-\omega(1)} = 1 - n_i^{-\omega(1)}$$

that (3) is true not just for all of $s$'s children but also for all of $s$'s parents' children, all of $s$'s grandparents' children, etc. We will assume this for the rest of the proof.

The fact that (3) holds for all of $s$'s children and $s$'s ancestors' children implies that all of $s$ and its ancestors are strongly feasible. By the Common Case Invariant, it follows that all of $s$ and its ancestors are in strongly common-case configurations.

The fact that both $s$ and its parent (if the parent exists) are in (even weakly) common-case configurations implies condition (i). Finally, if $s$ is neither the root nor a leaf, then condition (ii) follows from the second point along with (3). □

**Lemma 4.4.** *Let $T$ be the entire recursive tree and let $T'$ be a subtree. Suppose that every subproblem $s \in T \setminus T'$ that is strongly feasible is in a strongly common-case configuration, but that this is not necessarily the case for every $s \in T'$. Then, in $|T'| \log\log n$ time, one can rearrange the elements in $T'$ (in-place) so that the Common-Case Invariant holds; and so that, if there is a free slot in $T'$, it appears in the buffer of the root subproblem.*

*Proof.* Let $s$ be the root subproblem of $T'$. We can check in time $O(|T'| + \log\log n)$ if $s$ is strongly feasible. If $s$ is not strongly feasible, then neither will any of its descendants be, and we can complete the lemma by (1) moving any free slot in $T'$ to be in $s$'s buffer and (2) setting all of the failure indicator bits of $s$ and its descendants to be true. If $s$ is strongly feasible, then $s$'s parent (if it has one) is in a strongly common-case configuration, which implies that all of the elements that hash to $s$ or $s$'s descendants are already in $T'$. Since $s$ is weakly feasible, it follows that we can rearrange the elements in $T'$ alone so that $s$ is in a weakly common-case configuration and so that any free slot in $T'$ appears in $s$'s buffer—this can be done in place in time $O(|T'|)$. At this point, because all of $s$ and its ancestors are in weakly common-case configurations, we can conclude that $s$ is, in fact, in a strongly common-case configuration (and we can update $s$'s failure indicator appropriately). Having placed $s$ in a strongly common-case configuration (and placed any free slot in $T'$ in $s$'s buffer), we can recurse on $s$'s children's subtrees in order to complete the lemma. It is straightforward to see that each of the $O(\log\log |T'|)$ layers of recursion takes time at most $O(|T'| + \log\log n)$ time, making for a total of $O(|T'| \log\log |T'| + (\log\log |T'|) \cdot \log\log n) = O(|T'| \log\log n)$ time. □

**Lemma 4.5.** *Call a subproblem $s$* **well supplied** *if every (strict) ancestor $q$ of $s$ contains at least one element that hashes to $q$ in its buffer. If the Common-Case Invariant holds, then the following modifications to the data structure cannot violate it:*

(i) *For a subproblem $s$ that is strongly feasible and contains a free slot in its buffer, insert an element $x$ satisfying $h(x) = s$ into that free slot.*

(ii) *For a subproblem $s$ that is strongly feasible and well-supplied, delete an element $x$ satisfying $h(x) \in \{s, \texttt{parent}(s)\}$ from $s$'s buffer.*

(iii) *Move an item $x$ between the buffers of the subproblem $h(x)$ that it hashes to and one of $h(x)$'s children.*

*Proof.* Critically, the types of modifications described in all three items have the properties that: they do not change for any subproblem in the tree whether the subproblem is in a weakly common-case configuration.

Thus, for subproblems that were already strongly feasible (and thus already in a strongly common-case configuration), those subproblems continue to be in strongly common-case configurations. We will complete the proof by showing that the modifications described in each bullet point do not change which subproblems in the tree are weakly feasible. This means that, if the Common-Case Invariant held before the modification, then it continues to hold after.

The insertion in (i) risks changing the weak feasibility of one of $s$'s ancestors. However, we know that each of $s$'s ancestors $q$ are already in strongly common-case configurations prior to the insertion, and we know that the insertion does not change this, so $s$'s ancestors are still in strongly (and thus weakly) common-case configurations. Since, after the insertion, there is no free slot in

table, this implies that $s$'s ancestors are all still weakly feasible—their feasibility statuses have not changed after all.

The deletion in (ii) also risks changing the weak feasibility of one of $s$'s ancestors. Once again, we know that $s$'s ancestors are already in strongly common-case configurations prior to the deletion, and we know that the deletion does not change this, so $s$'s ancestors are still in strongly (and thus weakly) common-case configurations. Furthermore, since $s$ is well-supplied, each of $s$'s ancestors $q$ contains at least one element in its buffer that hashes to $q$. So, via moves of the third type, we could move the free slot that is currently in $s$ (due to our deletion) to be in the root's buffer without changing which subproblems are in weakly common-case configurations. Thus it is possible for $s$ and its ancestors all to be in weakly common-case configurations while having the free slot appear only in the root's buffer. This, in turn, implies that $s$ and its ancestors are all still weakly feasible, so, once again, the feasibility statuses have not changed for any subproblems.

Finally, the modification in (iii) does not change the set of elements present overall. Thus it also cannot change for any subproblem whether that subproblem is weakly feasible. □

**Implementing queries.** We can now describe how to implement queries. Suppose we wish to query an element $x$, and let $s = h(x)$ be the level-$C(x)$ subproblem that $x$ hashes to.

To query $x$, we begin by querying $s$'s buffer and the buffers of $s$'s children. Since these buffers are each implemented as rainbow cells, this takes expected time

$$O\Big(b_{C(x)}^{3/4} + f_{C(x)} \cdot b_{C(x)-1}^{3/4}\Big).$$

If any of these queries finds $x$, then we are done. Otherwise, we continue with the following logic.

Having examined $s$'s failure indicator, we know at this point whether or not $s$ is strongly feasible. If $s$ is strongly feasible, then by the Common-Case Invariant, $s$ must be in a strongly common-case configuration. This means that the only way $x$ could be in the hash table would be for it to appear in one of $s$'s or $s$'s children's buffers—since this is not the case, we can conclude that $x$ is not present. On the other hand, if $s$ is not strongly feasible, then we complete the query with the following ***failure-mode probe sequence***.

In the failure mode, the query scans the entirety of $s$ (including all of the buffers of all of $s$'s descendants). It then checks whether $s$'s parent $s^{(1)}$ is strongly feasible. If so, then the query is complete; otherwise, the query scans the entirety of $s^{(1)}$ (including all of the buffers of all of $s^{(1)}$'s descendants). It then checks whether $s^{(1)}$'s parent $s^{(2)}$ is strongly feasible. If so, then the query is complete; otherwise, the query scans the entirety of $s^{(2)}$, etc. The query continues like this until either it finds $x$ or it encounters an ancestor of $s$ that *is* strongly feasible, at which point the query can conclude by the Common-Case Invariant that $x$ is not present.

Before continuing with our description of rainbow hashing, it is worth taking a moment to verify the correctness and running time of queries.

**Lemma 4.6.** *Supposing the Common-Case Invariant, rainbow-hashing queries will be correct.*

*Proof.* If $s = h(x)$ is strongly feasible, and thus is in a strongly common-case configuration, then every element that hashes to $s$ is guaranteed to be in the buffer of either $s$ or one of $s$'s children. Thus, by querying these buffers, the query will succeed.

Suppose $s \in h(x)$ is not strongly feasible. Let $y$ be the lowest ancestor of $s$ that is strongly feasible. (If $y$ does not exist, then the query scans the entire hash table and is thus necessarily correct.) Let $z$ be the child of $y$ whose subtree contains $s$. Then the query scans the entirety of $z$'s subtree, so it suffices to show that all $x$ satisfying $h(x) = s$ are contained in the subtree. The fact that $y$ is strongly feasible implies by the Common-Case Invariant that $y$ is in a strongly common-case

configuration, which implies that every element $x$ that hashes to $z$ or $z$'s descendants is contained in the buffers of $z$ and its descendants. Therefore, if $x$ is in the hash table, it is contained in $z$ and its descendants, so the query will correctly ascertain whether $x$ is present. □

**Lemma 4.7.** *Supposing the Common-Case Invariant, the expected time per query, overall, will be* $O(1)$.

*Proof.* If we condition on $C(x)$, then since $s$ and $s$'s children are implemented as rainbow cells, the expected time to query $s = h(x)$'s and $s$'s children's buffers is

$$O\left(n_{C(x)}^{3/4} + f_{C(x)} \cdot n_{C(x)-1}^{3/4}\right).$$

Recalling that, for $i > 1$, Lemma 4.2 tells us that $\Pr[C(x) = i] = p_i = \Theta(b_{i-1}/n_{i-1})$, the expected time to query $s$'s and $s$'s children's buffers is

$$\begin{aligned}
&O\left(1 + \sum_{i>1} \frac{b_{i-1}}{n_{i-1}} \cdot \left(b_i^{3/4} + f_i \cdot b_{i-1}^{3/4}\right)\right) \\
&= O\left(1 + \sum_{i>1} \frac{n_i^{0.99 \cdot 0.51}}{n_i^{0.99}} \cdot \left((n_i^{0.51})^{3/4} + n_i^{0.01} \cdot ((n_i^{0.99})^{0.51})^{3/4}\right)\right) \\
&= O\left(1 + 1\sum_{i>1} \frac{n_i^{0.99 \cdot 0.51}}{n_i} \cdot \left(n_i^{0.01} \cdot (n_i^{0.51})^{3/4}\right)\right) \\
&= O\left(1 + \sum_{i>1} n_i^{0.99 \cdot 0.51 - 1 + 0.01 + 0.51 \cdot 3/4}\right) \\
&= O\left(1 + \sum_{i>1} n_i^{-0.1}\right) \\
&= O(1).
\end{aligned}$$

Additionally, to handle the case where $s$ may not be strongly feasible, the query will spend additional time $O(n_j)$, where $j$ is the largest $j$ such that the level-$j$ subproblem containing $s$ is not strongly feasible. Define $X_j$ to be the indicator random variable that the level-$j$ subproblem containing $s$ is not strongly feasible. Then our time contribution from cases where $s$ is not strongly feasible is at most

$$O\left(\sum_j X_j \cdot n_j\right),$$

which by Lemma 4.3 has expectation

$$O\left(\sum_{j \geq i} n_j^{-\omega(1)} \cdot n_j\right) = O(1).$$

Thus the overall expected query time is $O(1)$. □

**A helper method for updates: the "sampling trick".** Before we describe how to implement updates, let us first describe a sub-task that will prove useful. Given a level-$i$ subproblem $S$, and a color $\ell \in \{i, i+1\}$, the SAMPLE$(s, c)$ protocol either returns an element in $s$'s buffer with color $\ell$ or declares that no such element exists. (If $i = \overline{\ell}$, then the only valid value for $\ell$ is $i$, and if $i = 1$ then the only valid value is 2.)

The protocol is implemented by simply performing (up to) $b_i$ random samples from the buffer (returning if it ever finds an element with color $\ell$), and then, if none of those samples succeed, scanning the buffer in $O(b_i)$ additional time.

The following basic lemma will allow us to reason about the behavior of SAMPLE.

**Lemma 4.8.** *Let $x$ be an element, let $s$ be the level-$i$ ancestor of $h(x)$ for some $i$, and let $\ell \in \{\max(i, 2), \min(i+1, \overline{\ell})\}$. If the Common-Case Invariant holds, then* SAMPLE$(s, \ell)$ *takes $O(1)$ expected time.*

*Proof.* By Lemma 4.3, we have with probability $1 - n_i^{-\omega(1)}$ that at least a constant-fraction of the elements in $s$'s buffer have color $\ell$. If this is the case, then the expected number of random samples needed to find such an element is $O(1)$. If this is not the case, which happens with probability $n_i^{-\omega(1)}$, then the SAMPLE procedure may take time as much as $O(b_i)$. Thus the overall expected time is

$$O(1) + O(n_i^{-\omega(1)} b_i) = O(1). \qquad \square$$

**Implementing insertions.** Suppose we wish to insert an element $x$. If the root $r$ is not strongly feasible, then we perform the insertion by rebuilding the entire table from scratch. Otherwise, we invoke a recursive function INSERT$(r, x)$ that we will now define. In general, the function INSERT$(s, x)$ takes two inputs:

- a strongly feasible subproblem $s$ that currently contains the hash table's only free slot;
- and an element $x$ that hashes to either $s$ or one of $s$'s descendants.

It then implements the insertion of $x$ while preserving the Common-Case Invariant. The protocol for INSERT$(s, x)$ is:

1. If $x$ hashes to $s$, then insert $x$ into $s$'s free slot, and return. By Lemma 4.5, this preserves the Common-Case Invariant.

   Otherwise, let $c$ be the child of $s$ on the path from $s$ to $h(x)$. Compute $y =$ SAMPLE$(c, j)$, where $j$ is the level of $s$.

2. If either $c$ is not strongly feasible or $y =$ `null`, then rebuild $s$ from scratch to perform the insertion and preserve the Common-Case Invariant. This is possible by Lemma 4.4.

3. Otherwise, move $y$ into the free slot in $s$, creating a free slot in $c$. (By Lemma 4.5, this preserves the Common-Case Invariant.) Now $c$ is a strongly feasible subproblem that contains the hash table's only free slot, so we can complete the insertion by calling INSERT$(c, x)$.

**Implementing deletions.** Now suppose we wish to delete an element $x$ (and create a free slot in the root subproblem). We will assume that we already know where the element is in the hash table, since this can be determined with an $O(1)$-expected-time query. If the root $r$ is not strongly feasible, then we perform the deletion by rebuilding the entire table from scratch. Otherwise, we invoke a recursive function DELETE$(r, x)$ that we will now define. In general, the function DELETE$(s, x)$ takes two inputs:

- a strongly feasible subproblem $s$ that is well-supplied, as defined in Lemma 4.5;
- and an element $x$ that hashes to either $s$ or one of $s$'s descendants.

It then implements the deletion of $x$ and creates a free slot in $s$'s buffer, all while preserving the Common-Case Invariant. The protocol for $\textsc{Delete}(s, x)$ is:

1. If $x$ is in $s$, then delete $x$ and return. By Lemma 4.5, this preserves the Common-Case Invariant.

   Otherwise, let $c$ be the child of $s$ on the path from $s$ to the subproblem whose buffer contains $x$. Compute $y = \textsc{Sample}(s, j)$, where $j$ is the level of $s$.

2. If either $c$ is not strongly feasible or $y = \texttt{null}$, then rebuild $s$ from scratch to delete $x$, place a free slot in $s$'s buffer, and preserve the Common-Case Invariant. This is possible by Lemma 4.4.

3. Otherwise, since $s$ is well-supplied and $y$ exists, we can conclude that $c$ is well-supplied. Since, furthermore, $c$ is strongly feasible, we can legally invoke $\textsc{Delete}(c, x)$. Doing so creates a free slot in some position $p$ of $c$'s buffer and (by induction) preserves the Common-Case Invariant. Finally, we move $y$ from $s$'s buffer to position $p$ of $c$'s buffer. This move creates a free slot in $s$'s buffer, as desired, and preserves the Common-Case Invariant by Lemma 4.5.

**Analyzing insertions and deletions.** Because the Insert and Delete protocols are so similar, we combine their analyses into a single lemma:

**Lemma 4.9.** *The insertion and deletion protocols each take $O(\log \log n)$ expected time and preserve the Common-Case Invariant.*

*Proof.* The preservation of the Common-Case Invariant has already been established via Lemmas 4.4 and 4.5. So it suffices to prove the time bounds.

The slow case for insertions/deletions is if the operation is forced to rebuild an entire subtree. This can happen if either the root $r$ is not strongly feasible, or if a call to either $\textsc{Insert}(\cdot, \cdot)$ or $\textsc{Delete}(\cdot, \cdot)$ triggers a rebuild of a subproblem in Step 2 of either protocol.

Let $T_1$ denote the time spent on rebuilding entire subtrees and $T_2$ denote the other time spent on the insertion/deletion. The second quantity $T_2$ is dominated by making $O(\log \log n)$ calls to the Sample function, each of which we know by Lemma 4.8 takes $O(1)$ expected time. So $\mathbb{E}[T_2] \leq O(\log \log n)$. To complete the proof, we must also show that $\mathbb{E}[T_1] \leq O(\log \log n)$.

To bound $\mathbb{E}[T_1]$, we must first prove the following claim.

**Claim 4.10.** *In the $\textsc{Insert}(s, x)$ protocol, if Step 2 performs a rebuild because $\textsc{Sample}(c, j) = \texttt{null}$, then $s$ is not strongly feasible after the insertion.*

*Similarly, in the $\textsc{Delete}(s, x)$ protocol, if Step 2 performs a rebuild because $\textsc{Sample}(s, j) = \texttt{null}$, then $s$ is not strongly feasible after the deletion.*

*Proof.* We begin by proving the claim for insertions. If $\textsc{Sample}(c, j) = \texttt{null}$, then (prior to the insertion) there are no color-$j$ elements in $c$'s buffer. Since (prior to the insertion) $s$ is in a strongly common-case configuration, we know that all of the non-color-$j$ elements in the buffers of $c$ and its descendants hash to $c$ and its descendants. It follows that, after the insertion, the total number of items that hash to $c$ and its descendants will be $n_{j-1} + 1$. It is therefore not possible for $s$ to be in a strongly (or even weakly) common-case configuration after the insertion, so $s$ is no longer strongly (or weakly) feasible.

Next, we prove the claim for deletions. If $\textsc{Sample}(s, j) = \texttt{null}$, then (prior to the deletion) there are no color-$j$ elements in $s$'s buffer (and therefore no elements that hash to $c$ or its descendants). Since (prior to the deletion) $s$ is in a strongly common-case configuration, all of the elements that hash to $s$ and $s$'s descendants are contained in the buffers of $s$ and $s$'s descendants. Since there are no such elements in $s$'s buffer, it follows that the total number of elements that hash to $s$ and $s$'s descendants is at most $n_i - b_i$. After the deletion, the number of such items will be at most $n_j - b_i - 1$, which means that the buffers of $s$'s descendants *cannot* be occupied only by these items. It follows that (after the deletion) $s$ is no longer strongly (or weakly) feasible. □

From the preceding claim, we can conclude that, if either $\textsc{Insert}(s, x)$ or $\textsc{Delete}(s, x)$ performs a rebuild in Step 2, then at least one of $s$ or its child $c$ must be non-strongly-feasible either before or after the insertion or deletion. We know from Lemma 4.3 that the probability of this happening for $s$ in a given level $j$ is at most $n_{j-1}^{-\omega(1)} = n_j^{-\omega(1)}$. If a rebuild is performed, then it takes at most $\text{poly}(n_j)$ time, so the expected contribution of each level $j$ to $T_1$ is at most

$$n_j^{-\omega(1)} \cdot \text{poly}(n_j) \leq O(1/2^j).$$

This allows us to bound

$$\mathbb{E}[T_1] \leq \sum_j O(1/2^j) = O(1).$$ □

Putting the pieces together, we have proven Proposition 4.1.

# 5 Dynamic Resizing Without Increasing Update Time

In this section, we will extend Basic Rainbow Hashing to support dynamic resizing. As in the previous section, we shall continue to focus on hash tables that operate at load factor 1. In this context, what dynamic resizing means is that, as the total number $n$ of elements changes over time, the hash table automatically reconfigures itself to use exactly the first $n$ slots in memory. Perhaps surprisingly, we shall see that this seemingly stringent resizing property can be achieved without changing the timing characteristics of the hash table. In particular, we will prove the following proposition:

**Proposition 5.1.** *The basic rainbow hash table can be extended to support dynamic resizing with a continual load factor of* 1*, while preserving an expected query time of* $O(1)$ *and an expected update time of* $O(\log \log n)$*, where* $n$ *denotes the current size of the hash table.*

The main challenge in resizing is to allow $n$ to change by a constant factor, that is to support $n$ changing within a range of the form, say, $[N, 1.1 \cdot N]$ for some $N$. This will be our focus for most of the section.

**Embedding the recursion tree into an array with bit-reversed ordering.** In order to describe our resizing approach, it will be helpful to adopt a specific layout for how we embed the buffers in the recursion tree into an array of size $n$. We will refer to this layout as the ***bit-reversed layout*** for reasons that will become clear shortly.

Without loss of generality, we can choose $b_i, m_i$ to all be powers of two for all $i > 1$. We can also defer any rounding errors to the leaf subproblems. That is, we can ensure that every level-$i$ subproblem, $i > 1$, has buffer size *exactly* $b_i$ and fanout *exactly* $f_i$, while allowing some leaf subproblems to have sizes that differ by $\pm 1$ from each other. Finally, just to simplify our discussion

of the layout, we will think of every *slot* in the bottom layer of the tree as representing its own subproblem (notice that this doesn't change the behavior of the data structure at all), and we will think of the fanout of any level-2 subproblem as being simply the sum of the sizes of the level-1 subproblems that it contains. (Note that, in doing this, we are implicitly redefining $f_2$ to be what was formerly $f_1 \cdot n_1$, we are redefining $n_1 = b_1 = 1$, and we are allowing the fanouts of level-2 subproblems to be within 1 of $f_2$.)

With these WLOG assumptions in mind, place the buffers in levels $\overline{\ell}, \overline{\ell} - 1, \ldots$ in sub-arrays $A_{\overline{\ell}}, A_{\overline{\ell}-1}, \ldots$ that appear one after another from left to right. Each $A_i$ with $i > 1$ will have a power-of-two size, but $A_1$ may not.

We label the subproblems in level $i$ using integers $0, 1, \ldots, m_i - 1$, and refer to them as "the $j$-th subproblem in $A_i$" for $0 \leq j < m_i$. For $i < \overline{\ell}$, it is tempting to declare the parent of the $j$-th subproblem in $A_i$ to be the $\lfloor j/f_{i+1} \rfloor$-th subproblem in $A_{i+1}$. Rather than using the standard layout, however, we will use a ***bit-reversed layout***: for $i < \overline{\ell}$, the parent of the $j$-th subproblem in $A_i$ is the $k$-th subproblem in $A_{i+1}$, where $k$ is given by the *low-order* $\log m_{i+1}$ bits of $j$. Conversely, the $k$-th subproblem in $A_{i+1}$ has as children the subproblems with indices of the form $r \cdot m_{i+1} + k$ in level $i$. But important subtlety here is the relationship between levels 1 and 2. If level 1 has size $m_1$, then the children of subproblem $k$ in level 2 are the level-1 subproblems (i.e., slots in $A_1$) with indices

$$\{j = r \cdot m_2 + k \mid 0 \leq j < m_1\}.$$

Conveniently, the restriction $j < m_1$ automatically handles rounding errors—it guarantees that the total number of subproblems/slots in level 1 is exactly $m_1$, and dictates the assignment of those subproblems to level-2 parents. We can confirm that the layout handles rounding errors correctly, giving each level-$i$ subproblem the same total number of leaf slots up to $\pm 1$, with the following lemma.

**Lemma 5.2.** *Using a bit-reversed layout, every level-$i$ subproblem has the same total number of leaf slots up to $\pm 1$.*

*Proof.* For the $k$-th level-$i$ subproblem in $A_i$, the leaf slots in $k$'s subtree are the slots in $A_1$ whose low-order bits are given by $k$, that is, the indices of the form

$$\{j = r \cdot m_i + k \mid 0 \leq j < m_1\}.$$

The number of such indices is exactly

$$1 + \lfloor (m_1 - k)/m_i \rfloor.$$

Since $k \in \{0, 1, \ldots, m_i - 1\}$, this value varies by at most $\pm 1$ for different subproblems $k$ in $A_j$. □

What is nice about this layout is not just that it handles rounding errors cleanly (a fact that should be viewed as a minor detail), but rather that, as we will now see, it enables a surprisingly simple resizing approach.

**Resizing by changing $m_1$ only.** Suppose we wish to allow $n$ to change within the range $[N, 1.1 \cdot N]$ for some $N$. We will achieve this by simply changing the value of $m_1$ (i.e., the number of slots in $A_1$). Before an insertion, we first increment $n$ (and thus $m_1$). This creates a free slot in some leaf, which we can then migrate to the buffer of the root subproblem using the same protocol that we used to migrate free slots up the recursion tree for deletions. Similarly, after deletion, we decrement $n$ (and thus $m_1$). This removes a slot from some leaf—the element that gets evicted from that slot

can be re-inserted using the standard insertion procedure (notice, in particular, that because we have just performed a deletion, there is a free slot in the root buffer of the tree).

With these modifications in mind, the only point that we must be careful about is that, as $n$ changes within the range $[N, (1+0.1)N]$, the values of $p_i$ do *not* change. For each subproblem $s$, let $t_s$ denote the size of the subproblem. Note that when we change $n$ and thus $m_1$, this also implicitly changes some values of $t_s$.

The only parts of the probabilistic analysis in Section 4 that use the relationship between the $p_i$'s and the other parameters are the proofs of Lemmas 4.2 and 4.3. The fact that Lemma 4.2 holds for $n = N$ directly implies that it holds for $n \in [N, 1.1 \cdot N]$. Lemma 4.3 requires a bit more care, however, as it needs the $p_i$'s to satisfy (2), which for a level-$i$ subproblem $s$, with $1 < i < \overline{\ell}$, expands to

$$(p_2 + \cdots + p_i)n/m_i \in t_s - [0.4, 0.6] \cdot b_i. \tag{4}$$

To recover the proof of Lemma 4.3, it suffices to show that, as $n$ changes within the range $[N, 1.1 \cdot N]$, even though the values $\{p_i\}$ do not change, (4) continues to hold.

**Lemma 5.3.** *Let $1 < i < \overline{\ell}$ and let $s$ be a level-$i$ subproblem. Let $T_s$ be the value of $t_s$ when $n = N$, and suppose that*

$$(p_2 + \cdots + p_i)N/m_i = T_s - 0.5 \cdot b_i \pm 1.$$

*Then, we claim that for all $n \in [N, 1.1 \cdot N]$, if we change $m_1$ so that the total number of slots is $n$ (and calculate the values of $t_s$ based on $n$), then we have*

$$(p_2 + \cdots + p_i)n/m_i \in t_s - [0.4, 0.6] \cdot b_i.$$

*Proof.* Observe that

$$\begin{aligned}
&(p_2 + \cdots + p_i)n/m_i \\
&= \frac{n}{N}(p_2 + \cdots + p_i)N/m_i \\
&= \frac{n}{N} \cdot T_s - \frac{n}{N} \cdot 0.5 \cdot b_i \pm O(1) \\
&\in \frac{n}{N} \cdot T_s - [0.5, 0.55] \cdot b_i \pm O(1).
\end{aligned}$$

To complete the proof, we will show that

$$\frac{n}{N} \cdot T_s = t_s \pm O(1).$$

Since, by Lemma 5.2, $t_s = n/m_i \pm 1$ and $T_s = N/m_i \pm 1$, it suffices to show that

$$\frac{n}{N} \cdot \frac{N}{m_i} = \frac{n}{m_i} \pm O(1),$$

which holds trivially. □

Having recovered Lemma 4.3, the rest of the analysis from Section 4 holds without modification. This gives us the following proposition:

**Proposition 5.4.** *Given a parameter $N$, the basic rainbow hash table can be extended to support dynamic resizing with a continual load factor of 1 and with $n$ varying in the range $[N, 1.1 \cdot N]$. Furthermore, this preserves an expected query time of $O(1)$ and an expected update time of $O(\log \log n)$, where $n$ denotes the current size of the hash table.*

**Allowing $n$ to change by more than a factor of 1.1.** Finally, we can use standard rebuilding techniques to allow for $n$ to change by more than a factor of 1.1. Let $r \in (0.99, 1)$ be uniformly random. Let $N_i := \lfloor 1.09^i \cdot r \rfloor$. Whenever $n$ crosses from $\leq N_i$ to $> N_i$ for some $i$, we will rebuild the entire hash table to use $N = N_i$. Such a rebuild can be performed in place and in $O(n \log \log n)$ expected time using Lemma 4.4 (with $T'$ equal to the entire tree). Since each value $n$ has probability $\Theta(1/n)$ of being within $\pm 1$ of some $N_i$, the probability of a given update triggering a rebuild is $\Theta(1/n)$. The expected time spent per update on these rebuilds is therefore

$$\Theta(1/n) \cdot O(n \log \log n) = O(\log \log n).$$

Thus, we can extend Proposition 5.4 to allow $n$ to change arbitrarily over time, while only adding $O(\log \log n)$ additional expected time per update, as desired. This completes the proof of Proposition 5.1.

# 6 Supporting Load Factor $1 - \varepsilon$

In this section, we give a black-box transformation that takes the resizable rainbow hash table construction from Proposition 5.1 (which operates at load factor 1) and uses it to construct a dynamically-resized hash table that operates at load factor $1 - \varepsilon$, supports $O(1)$ expected-time queries, and supports $O(\log \log \varepsilon^{-1})$ expected-time updates.

Our main result will be the following theorem:

**Theorem 6.1.** *There exists a classical open-addressed hash table that is dynamically resized to maintain a load factor of $\geq 1 - \varepsilon$ while supporting queries in $O(1)$ expected time and updates in $O(\log \log \varepsilon^{-1})$ time.*

At the end of the section, we will also prove the analogous result for fixed-capacity hash tables.

Throughout the section, we will assume that $\varepsilon = n^{-o(1)}$, since when $\varepsilon = n^{-\Omega(1)}$, we are okay with $O(\log \log n)$-time updates, so we can keep the hash table at load factor 1. Furthermore, as in Section 5, it suffices to handle $n \in [N, 1.1 \cdot N]$ for some known parameter $N$, since we can use the random-threshold rebuild technique from Section 5 to handle larger changes in $n$. Finally, we will satisfy ourselves with a load factor of the form $1 - \Theta(\varepsilon)$, since this is equivalent to $1 - \varepsilon$ up to constant-factor changes in $\varepsilon$.

**The basic setup: rainbow hash tables with overflow handling.** Our basic construction will be as follows. Let $k = \operatorname{poly} \varepsilon^{-1}$. We will maintain $N/k$ dynamically resized rainbow hash tables $H_1, H_2, \ldots, H_{N/k}$ that are each allocated space $(1 - \varepsilon)\frac{n}{N} \cdot k \pm 1$ at any given moment. (The slots of the hash tables can be interleaved so that, if we wish to increase the space in each hash table by 1, we just need to extend the size of our array overall by $N/k$.) As a convention, we will set $k' = \frac{n}{N}k$ (so $k'$ changes over time).

Each element $x$ uses a random hash $g(x) \in [N/k]$ to select which hash table $H_{g(x)}$ it belongs in. If the hash table $H_{g(x)}$ is full when $x$ is inserted (which will be the common case), then $x$ is placed in a ***overflow buffer*** $O_{g(x)}$ (whose implementation we will specify later). If an element in some $H_i$ is deleted, and the overflow buffer $O_i$ is non-empty, then an element in $O_i$ is moved to $H_i$. Similarly, if the amount of space allocated to $H_i$ is incremented, and $|O_{g(x)}| > 0$, then an item from $O_{g(x)}$ is moved to $H_i$ (so that the increase in the size of $H_i$ corresponds to an insertion from $H_i$'s perspective); and if the amount of space allocated to $H_i$ is decremented (and $H_i$ was full beforehand), then a random item will be deleted from $H_i$ (so that the decrease in size of $H_i$ corresponds to a deletion from $H_i$'s perspective) and that item will be placed in $O_i$.

The overall rule will be that, if $O_i$ is non-empty, then $H_i$ is full. Thus, if we use $z_i$ to denote the capacity of $H_i$ at any given moment, and $r_i$ to denote the number of elements $x$ satisfying $g(x) = i$ at any given moment, then $|O_i| = \max(0,\, r_i - z_i)$.

**Lemma 6.2.** *With probability* $1 - k^{-\omega(1)}$*, we have*

$$|O_i| \in [0.5 \cdot \varepsilon k',\, 1.5 \cdot \varepsilon k'].$$

*Furthermore,* $\mathbb{E}[\max(0,\, |O_i| - 2\varepsilon k)] = o(1)$.

*Proof.* Let $o_i = X_i - |H_i|$, and note that $|O_i| = \max(0, o_i)$. By design, $o_i = X_i - |H_i| = X_i - (1 - \varepsilon)k' \pm O(1)$ where $X_i$ is a binomial random variable with mean $k'$. The amount that $o_i$ deviates from its mean of $\varepsilon k' \pm 1$ is at most the amount that $X_i$ deviates from its mean of $k'$. By a Chernoff bound, the probability of $X_i$ deviating from its mean by $\Omega(\varepsilon k') > \Omega((k')^{3/4})$ is at most $(k')^{-\omega(1)} = k^{-\omega(1)}$; it follows that, with probability $1 - k^{-\omega(1)}$, we have $o_i \in [0.5 \cdot \varepsilon k',\, 1.5 \cdot \varepsilon k']$ and therefore also $|O_i| \in [0.5 \cdot \varepsilon k',\, 1.5 \cdot \varepsilon k']$.

To obtain the final claim in the lemma, we can again apply a Chernoff bound to $X_i$ to obtain that the expected value of $\max(0,\, X_i - \mathbb{E}[X_i] - \Omega(\varepsilon k))$ is $o(1)$. Finally, since

$$o_i - 2\varepsilon k = X_i - (1 - \varepsilon)k' - 2\varepsilon k \pm 1 = X_i - \mathbb{E}[X_i] - 2\varepsilon k + \varepsilon k' = X_i - \mathbb{E}[X_i] - \Omega(\varepsilon k),$$

it follows that $\mathbb{E}[\max(0,\, |O_i| - 2\varepsilon k)] = \mathbb{E}[\max(0,\, o_i - 2\varepsilon k)] = o(1)$. □

**Handling under-filled $H_i$'s.** If $H_i$ is full, at any given moment, it can be implemented directly as a resizable rainbow hash table. If $H_i$ is not full (which by Lemma 6.2 happens with probability $k^{-\omega(1)}$), then $H_i$ is said to incur an ***under-fill error***. In this case, slot 1 of $H_i$ is left empty so that queries and updates can examine it and determine that an under-fill error has occurred. When an under-fill error has occurred, queries to elements $x$ satisfying $g(x) = i$ read all of $H_i$; and updates to $H_i$ spend $O(\text{poly}\,|H_i|)$ time checking if $H_i$ is still experiencing an under-fill error, and rebuilding $H_i$ appropriately based on whether it is or is not still experiencing an under-fill error.

For any given element $x$, the probability that $H_{g(x)}$ is incurring an under-fill error at any given moment is $k^{-\omega(1)}$ by Lemma 6.2. So the increase in expected query and update times due to under-fill errors is $o(1)$.

**Implementing the overflow buffers.** The only tricky part of the data structure is implementing the overflow buffers. We can afford to use $O(\varepsilon N)$ total space to implement the $O_i$'s, and we wish to implement them in such a way that we can support the following operations in $O(1)$ expected time each:

- $\textsc{Sample}(O_i)$: samples a random element from $O_i$, if $|O_i| > 0$, and returns `null` if $|O_i| = 0$.
- $\textsc{Query}(O_i, y)$: determines if $y \in O_i$.
- $\textsc{Insert}(O_i, y)$: inserts $y$ into $O_i$.
- $\textsc{Delete}(O_i, y)$: deletes $y$ from $O_i$.

Each $O_i$ is allocated a $2\varepsilon k$-size array $A_i$ that it uses to store its elements (unless $|O_i| > 2\varepsilon k$). The elements stored in $A_i$ treat $A_i$ as a linear-probing hash table. By Lemma 6.2, the load factor of $A_i$ is between 0.1 and 0.9 with probability $1 - k^{-\omega(1)}$. Thus, the expected time to perform queries/inserts/deletes in $A_i$ is $O(1)$. Additionally, if we wish to sample a random element from

$A_i$, we can just randomly sample slots until we find one that is occupied; since, with probability $1 - k^{-\omega(1)}$ the number of occupied slots in $A_i$ is $\geq 0.1 \cdot |A_i|$, the expected time of this sampling procedure is $O(1)$.

The only question is what we should do when $A_i$ itself overflows, that is, when $|O_i| \geq |A_i| = 2\varepsilon k$. Note that, since this occurs with only a small probability ($k^{-\omega(1)}$), we are okay with having relatively expensive (say, poly($k$)-time) operations in this case.

To handle overflowed $A_i$'s, we allocate an array $B$ of size $\varepsilon N$. If $A_i$ overflows, its overflow elements $x$ treat $B$ as a linear-probing hash table, where the hash of $x$ within $B$ is calculated by a "hash function"

$$\overline{h}(i) := \varepsilon \cdot k \cdot i.$$

Note that, since $A_i$ is indexed by $i$ ranging over $i \in [N/k]$, the quantity $\overline{h}(i)$ is a valid index in $[|B|] = [\varepsilon N]$.

Since all of the overflow elements $x$ from $A_i$ have the same hash $\overline{h}$ as each other, it suffices to show that, even if we condition on $A_i$ overflowing, the expected length of the run in $B$ that contains the overflow elements is at most poly($k$). This will allow us to implement operations on $O_i$ in poly($k$) expected time when $A_i$ overflows, as desired.

Thus, to complete our discussion of how to implement the $O_i$'s, it suffices to prove the following lemma:

**Lemma 6.3.** *Conditioned on $A_i$ overflowing, the expected length of the run in $B$ that contains the overflow elements from $A_i$ is at most* poly($k$).

*Proof.* It suffices to show that, conditioned on $A_i$ overflowing, we have for each run-length $t >$ poly($k$) (where poly($k$) is a large polynomial of our choice) that: the probability that the overflow elements from $A_i$ are in a run in $B$ of length $t$ is at most

$$e^{-\omega(\log t)}. \tag{5}$$

In order for the elements to be in a run of length $t$, there must be some contiguous interval $I \ni \overline{h}(i)$ of $t$ slot indices in $B$ such that the number of elements in $B$ that hash into $I$ is exactly $|I| = t$. As there are only $t$ options for $I$ satisfying $|I| = t$ and $\overline{h}(i) \in I$, it suffices to show that each individual option has probability at most $e^{-\omega(\log t)}$ of occurring. Since $t >$ poly($k$) for a polynomial of our choice, this, in turn, reduces to bounding the probability of a given interval $I$ occurring by, say,

$$e^{-\Omega(\sqrt{t}/\operatorname{poly}(k))}.$$

For the rest of the proof, fixing some interval $I$ of $t$ slots in $B$ containing $\overline{h}(i)$, we wish to bound the probability that, conditioned on $A_i$ overflowing, there are at least $t$ elements that overflow from $A_j$'s satisfying $\overline{h}(j) \in I$.

Define $X_j$ as the number of items that hash to $H_j$. (So, if we were to not condition on anything, then $X_j$ would be a binomial random variable with mean $k'$.) Define $Y_j$ as the number of items that overflow from $A_j$, that is,

$$Y_j = \max(0,\, X_j - |A_j| - |H_j|).$$

Recall that we are conditioning on $Y_i \geq 1$.

There are $O(\varepsilon t/k)$ values of $j$ such that $\overline{h}(j) \in I$. Let $J$ be the set of such values $j$, excluding $i$. Then, in order for $I$ to have $t$ elements in it, we would need

$$Y_i + \sum_{j \in J} Y_j \geq t,$$

which implies either that

$$Y_i \geq t/2 \tag{6}$$

or that

$$\sum_{j \in J} Y_j \geq t/2. \tag{7}$$

Again, we are interested in the probabilities of these events conditioned on $Y_i \geq 1$.

To bound the probability of (6), observe that the random variable $(Y_i - 1 \mid Y_i \geq 1)$ is dominated by $X_i$ (not conditioned on anything).[2] Therefore,

$$\Pr[Y_i \geq t/2 \mid Y_i \geq 1] \leq \Pr[X_i \geq t/2 - 1].$$

Since $X_i$ is a binomial random variable with mean $\Theta(k)$, this latter probability is, by a Chernoff bound, at most $e^{-\Omega(t/k)}$.

To bound the probability of (7), observe that

$$\Pr\left[\sum_{j \in J} Y_j \geq t/2 \;\middle|\; Y_i \geq 1\right] < \Pr\left[\sum_{j \in J} Y_j \geq t/2\right],$$

where the latter probability does not condition on anything.

The values $\{Y_j \mid j \in J\}$ are negatively associated random variables that each has expected value $o(1)$ (by Lemma 6.2) and that are each bounded above by a geometric random variable with mean $O(k)$ (since even $X_j$ is, by even a very weak Chernoff bound, bounded above by such a geometric random variable). By a Chernoff bound for sums of negatively associated geometric random variables [40], the probability that $\sum_{j \in J} Y_j$ exceeds its mean by more than $r \cdot \sqrt{|J|}k$, for a given $r > 0$, is at most $e^{-\Omega(r)}$. Since the mean of this sum is $|J| \cdot o(1) \leq |J|$, it follows that

$$\Pr\left[\sum_{j \in J} Y_j > |J| + r \cdot \sqrt{|J|}k\right] \leq e^{-\Omega(r)}.$$

Since $|J| = O(\varepsilon t/k) = o(t)$, this lets us bound

$$\begin{aligned}
\Pr\left[\sum_{j \in J} Y_j \geq t/2\right] &< \Pr\left[\sum_{j \in J} Y_j \geq |J| + t/4\right] \\
&= e^{-\Omega(t/(k\sqrt{|J|}))} \\
&= e^{-\Omega(t/(k\sqrt{\varepsilon t/k}))} \\
&= e^{-\Omega(\sqrt{t}/\operatorname{poly}(k))},
\end{aligned}$$

as desired. □

In the case where $B$ overflows, which is an extremely rare event, we say the ***global failure*** has occurred. When this happens, we keep the entire hash table in an arbitrary state (i.e., placing all keys in arbitrary slots), and for each insertion/deletion/query, we spend $\Theta(n)$ time scanning through the table and checking if it can return to the normal case. A special boolean is stored to

[2] This is an application of the more general fact that, for a binomial random variable $X$, $\Pr[X \geq r \mid X \geq r - 1]$ is a monotonically decreasing function in $r$.

indicate whether the global failure is occurring, which is again encoded by the relative order of two special elements. According to (5), at any given moment, the probability that $B$ overflows is at most $e^{-\omega(\log(\varepsilon N))} = N^{-\omega(1)}$, so the global failure contributes a negligible amount to the expected insertion/deletion/query time.

**Putting the pieces together.** We can now summarize the full procedure for implementing insertions/deletions/queries.

To query an element $x$, we must query $O_{g(x)}$ and $H_{g(x)}$. The expected time needed to query $H_{g(x)}$ is $O(\log\log k) = O(\log\log \varepsilon^{-1})$; and the expected time needed to query $O_{g(x)}$ is $O(1)$. To keep the query within the model of classical open addressing, we can simply interleave the two probe sequences and stop when both queries have completed. Overall, the expected query time is $O(1)$.

To insert an element $x$, we first check if $H_{g(x)}$ is under-filled. If so, we follow the protocol described earlier to handle under-fill errors. Otherwise, we insert $x$ into $O_{g(x)}$. Each of the above steps takes $O(\log\log k) = O(\log\log \varepsilon^{-1})$ expected time.

To delete an element $x$, we first determine where it is. If $x \in O_{g(x)}$, we delete it in $O(1)$ expected time. Otherwise, if $x \in H_{g(x)}$, we delete it from $H_{g(x)}$; we use $\textsc{Sample}(O_{g(x)})$ to find an element $y$ that we can move from $O_{g(x)}$ into $H_{g(x)}$; and we insert $y$ into $H_{g(x)}$ in order to keep $H_{g(x)}$ at load factor 1. If $y$ does not exist, then the free slot in $H_{g(x)}$ is placed in its slot 1 to indicate that it is experiencing an under-fill error. Each of the above steps takes $O(\log\log k) = O(\log\log \varepsilon^{-1})$ expected time.

Finally, as insertions and deletions are performed, we must not only implement those operations, but also add/remove slots to the $H_i$'s in order to maintain the invariant that each $H_i$ is given $(1-\varepsilon)\frac{n}{N} \cdot k \pm 1$ slots at any given moment.

Adding a slot to some $H_i$ is implemented as follows. A call to $\textsc{Sample}(O_i)$ is made to either get a random element of $O_i$ or determine that $|O_i| = 0$. If $|O_i| = 0$, then the addition of a slot to $H_i$ will cause it to be under-filled. In this case, $H_i$ is rebuilt from scratch (but remember that, since under-filled $H_i$'s are very rare, this contributes negligibly to our expected time bound). In the more common case where $|O_i| > 0$, we move an item from $O_i$ into $H_i$—this allows $H_i$ to view its increase in size as being due to an insertion, which in turn takes $O(\log\log k) = O(\log\log \varepsilon^{-1})$ expected time.

Deleting a slot from some $H_i$, on the other hand, is implemented as follows. If $H_i$ is under-filled, then $H_i$ is simply rebuilt (again, since under-filled $H_i$s are very rare, this contributes negligibly to our expected time bound). Otherwise, a random element is removed from $H_i$ and placed into $O_i$. This allows $H_i$ to view its decrease in size as being caused by a deletion, which in turn takes $O(\log\log k) = O(\log\log \varepsilon^{-1})$ expected time.

Putting the time bounds together, we have proven Theorem 6.1.

**Supporting all load factors $< 1-\varepsilon$ for the fixed-capacity case.** Finally, we conclude the section by describing how to handle arbitrary load factors in a fixed-capacity hash table.

**Theorem 6.4.** *There exists a classical open-addressed hash table that has a fixed capacity $N$, and that allows for load factors of up to $1-\varepsilon$ while supporting queries in $O(1)$ expected time and updates in $O(\log\log \varepsilon^{-1})$ time.*

*Proof.* Let us begin by assuming that the size is guaranteed to stay in $[0.95N, (1-\varepsilon)N]$. Then, we will implement our fixed-capacity data structure, which we will call $D$, by actually using the dynamically resized data structure, which we will call $D'$, already described earlier in the section. The data structure $D'$ lives in a prefix of the memory allocated to $D$, and is parameterized to preserve a load factor of $1-\varepsilon$ as the number of elements varies within the range $[N', (1-\varepsilon)N < 1.1N']$

for $N' = 0.95N$. As we have already established, $D'$ supports $O(1)$ expected-time queries and $O(\log\log\varepsilon^{-1})$ expected-time updates.

There is one point that we must be very careful about, however. One cannot, *in general*, use a dynamically-resized classical-open-addressed hash table to implement a fixed-capacity classical open-addressed hash table. This is because, in the dynamic-resizing case, the probe sequence for an item $x$ is permitted to depend on both $x$ and the *current* value of $N$; but in the fixed-capacity case, the probe sequence must depend on only $x$ and a *fixed* $N$. Fortunately, in our construction of $D'$, the probe sequences that we use for $n \in [N', 1.1 \cdot N']$ are subsequences of the probe sequences that we use for $n = 1.1 \cdot N$ (and the difference for each probe sequence is just the addition/removal of $O(1)$ probes in the first $O(1)$ entries of the sequence). Therefore, in our case, we *can* use $D'$ within $D$.

Finally, we must also handle cases where our size drops below $0.95N$. Let $T$ be a random threshold in $[0.9N, 0.95N]$. Whenever the number of elements in the hash table crosses below $T$, we rebuild the hash table to use standard linear probing. Whenever the number of elements crosses above $T$, we rebuild the hash table to use $D'$ as described above. Each rebuild takes $O(N\log\log\varepsilon^{-1})$ time, and the probability of a given insert/deletion crossing the threshold $T$ is $O(1/N)$, so the contribution of rebuilds to the expected update time is $O(\log\log\varepsilon^{-1})$. □

# 7 The Lower Bound

In this section, we prove an $\Omega(\log\log\varepsilon^{-1})$ lower bound on the amortized expected time per insertion/deletion in any classical open-addressing hash table that supports (even moderately) efficient queries.

**Theorem 7.1.** *Suppose the universe size $U = \operatorname{poly} n$ is a large polynomial of $n$. If a classical open-addressing hash table stores $n$ keys with load factor $1-\varepsilon$, then the expected amortized time per operation is at least $\Omega(\log\log\varepsilon^{-1})$. Moreover, as long as the expected query time is $O\big(2^{\sqrt{\log\varepsilon^{-1}}}\big)$, the expected amortized time per insertion/deletion is at least $\Omega(\log\log\varepsilon^{-1})$.*

The hard distribution used for proving Theorem 7.1 will simply be a sequence of $n^2$ random insertions and deletions (Distribution 1). Under this operation sequence, we will show that, if a classical open-addressing hash table has low average probe complexity, it must relocate a large number of keys to other slots during the operations (hence giving it a large insertion/deletion time). By Yao's minimax principle, we also assume the hash table is deterministic.

**Distribution 1:** Hard distribution

1 Initialize the hash table with $n$ random keys from the universe $[U]$
2 **Repeat** $M = n^2$ times**:**
3 | Delete a random element from the current key set $S$
4 | Insert a random element in $[U] \setminus S$

Recall that $n$ is the maximum number of keys the hash table can store, and assume the load factor is $1-\varepsilon$ for some $1/n \le \varepsilon \le \Theta(1)$, so the number of slots is $N = (1+\Theta(\varepsilon))\cdot n$. There is a *deterministic* function $h$ that maps each key $x \in [U]$ to a probe sequence $\{h_i(x)\}_{i\ge 1}$. For technical reasons, we allow each entry $h_i(x)$ to be either a slot $s \in [N]$ or "`null`". Without loss of generality, we assume that there is a *special slot*, say slot $N$: When we insert any key $x$, key $x$ is first put into the special slot, then the algorithm will arrange the keys to move $x$ to a normal slot.

Recall that a key $x$ storing in slot $s = h_i(x)$ is said to have ***probe complexity*** $i$ (assuming $h_i(x)$ is the first occurrence of $s$ in the probe sequence). When a key is stored in the special slot, we say the key has probe complexity $N$. Any key with probe complexity $i$ will cost the query algorithm $O(i)$ time. Thus, the *average probe complexity* over the current key set measures the query time of the hash table. During insertions and deletions, the hash table may move some keys to other slots, and we define the ***switching cost*** of this operation to be the number of moved keys. The switching cost is a lower bound of the time spent on the operation.

For any probe-sequence function $h$, integer $i \geq 1$, and slot $s \in [N]$, we define

$$q(h, i, s) \coloneqq n \Pr_{x \in [U]}[\text{probe-complexity}(x, s) \leq i] = n \Pr_{x \in [U]}[h_k(x) = s \text{ for some } k \leq i].$$

We say $h$ is ***nearly uniform*** if $q(h, i, s) \leq O(i^{10})$ for all $i$ and $s$. We first assume the function $h$ is nearly uniform and prove the following lower bound. At the end of this section, we will remove this assumption.

**Theorem 7.2.** *Suppose the universe size $U = \operatorname{poly} n$ is a large polynomial of $n$. Assume there is a classical open-addressing hash table, which stores $n$ keys with load factor $1 - \varepsilon$, uses a nearly uniform probe-sequence function $h$, and has average probe complexity $O\big(2^{\sqrt{\log \varepsilon^{-1}}}\big)$ in expectation at any given moment, then the expected amortized switching cost during each insertion or deletion must be $\Omega(\log \log \varepsilon^{-1})$.*

To prove this theorem, we let the hash table take Distribution 1 as input, and show a lower bound on the average switching cost per insertion and deletion. We start by setting up main concepts in our analysis.

**Levels.** We define $L \coloneqq \lceil (\log \log \varepsilon^{-1})/2 \rceil$. Suppose a key $x$ is stored in slot $s = h_k(x)$ with probe complexity $k$, i.e., $k = \min\{k' \in \mathbb{N}_+ \mid h_{k'}(x) = s\}$. We define the ***level*** of key $x$ stored in slot $s$, written $\ell(x, s)$, as follows:

- If $k \leq 2^{2^L}$, the level is $\ell(x, s) = 0$.
- If $k > 2^{2^{2L}}$, the level is $L$. As a special case, any key stored in the special slot has level $L$.
- Otherwise, if $k \in (2^{2^{L+i-1}}, 2^{2^{L+i}}]$ for some $i \in \mathbb{N}_+$, the level is $i$.

Moreover, we define the level of a slot $s$ in a given state to be the level of the key stored in slot $s$, if slot $s$ is not empty; the level of an empty slot is defined as $L$. It is clear that the level of the special slot is always $L$.

When the algorithm moves a key $x$ from one slot to another, it may change the level of the key. We define the ***impact*** of the move to be how much the level of $x$ *decreases* (negative impact means the level increases). We have the following lemma from [6] with almost the same proof.

**Lemma 7.3** ([6, Lemma 5])**.** *Let $\Psi$ be the sum of impacts of all the moves that the algorithm performs during the $M$ insertions and deletions. Then*

$$\mathbb{E}[\Psi] = \Theta(ML).$$

*Proof Sketch.* Let $J$ be the sum of levels of all keys stored in the current hash table. When we insert a key $x$, it is first placed in the special slot with level $L$, thus increasing $J$ by $L$. When we delete a key $x$, the expected level of $x$ is $O(1)$ since the expected average probe complexity at any moment is small, so removing $x$ from the slot containing it will decrease $J$ by $O(1)$ in expectation. Therefore, the $M$ insertions and deletions will increase $J$ by $\Theta(ML)$, implying that the hash table should rearrange the keys to decrease the sum of levels by $\Theta(ML)$. The lemma follows. □

**Potential function.** To prove Theorem 7.2, we will construct a potential function $\Phi$ of any given state of the hash table, satisfying three properties:

- **Property 1.** Each insertion or deletion increases $\Phi$ by $O(1)$ in expectation.
- **Property 2.** If the algorithm performs a move with impact $r$ (i.e., the level of the moved key is decreased by $r$), $\Phi$ will be decreased by $r \pm O(1)$.
- **Property 3.** $0 \le \Phi \le O(nL)$ always holds.

Once we have a potential function $\Phi$ satisfying all three properties, the following proposition from [6] will imply Theorem 7.2.

**Proposition 7.4** (Implicit in [6])**.** *If Lemma 7.3 holds and there is a potential function $\Phi$ satisfying Properties 1 to 3 above, then Theorem 7.2 holds.*

*Proof Sketch.* At a given moment, let $\Psi$ be the total impact the algorithm has made so far, and let $\Phi$ be the potential function on the current state. By Lemma 7.3 and Property 3 of $\Phi$, the value $\Psi + \Phi$ is increased by $\Theta(ML)$ during all $M$ insertions and deletions, but each operation and each move can only increase it by $O(1)$. Therefore, the number of moves during all operations is at least $\Omega(ML)$. □

The only remaining step to prove Theorem 7.2 is to construct such a potential function.

## 7.1 Constructing the potential function

In this subsection, we construct the potential function $\Phi$ based on the concept of *stanzas*.

**Definition 7.5** (Stanzas)**.** Fix a state of the hash table. Let $10 \le i \le L$ and $j \ge 2$ be integers. We say a sequence of slots $s_1, s_2, \ldots, s_j$ is an $i$-***stanza***, if the following conditions hold:

- $s_1$ and $s_j$ have level $\ge i$, while $s_2, \ldots, s_{j-1}$ are *distinct* slots with level $\le i - 10$ (we allow $s_1 = s_j$).
- Slots $s_1, \ldots, s_{j-1}$ are non-empty. Assuming keys $x_1, \ldots, x_{j-1}$ are stored in slots $s_1, \ldots, s_{j-1}$ respectively, then $\ell(x_k, s_{k+1}) \le i - 10$ for every $k \in [j-1]$.

In such a stanza, we call $s_1$ the ***starting slot***, $s_2, \ldots, s_{j-1}$ the ***internal slots***, and $s_j$ the ***final slot***. We say a collection of stanzas is ***disjoint*** if each slot with level $\ge i$ is used at most once as the starting slot and at most once as the final slot, whereas each slot with level $\le i - 10$ is used at most once as the internal slot.

**Definition 7.6** (Potential of stanzas)**.** For an $(i+10)$-stanza $s_1, \ldots, s_j$, assuming $x_k$ is the key stored in slot $s_k$ for $1 \le k \le j-1$, we define its ***potential*** to be

$$\phi(s_1, \ldots, s_j) \coloneqq 1 - \sum_{k=2}^{j-1} \sqrt{2}^{\ell(x_k, s_k) - i} - \sum_{k=1}^{j-1} \sqrt{2}^{\ell(x_k, s_{k+1}) - i}.$$

**Definition 7.7** (Potential function)**.** We define the potential of a collection of disjoint stanzas to be the sum of potentials of the stanzas in the collection. For $10 \le i \le L$, let $\Phi_i$ denote the maximum potential of a collection of disjoint $i$-stanzas in the current state. We define the potential function

$$\Phi \coloneqq \sum_{i=10}^{L} \Phi_i.$$

To analyze the properties of $\Phi$, we define the ***key-slot graph*** $G$ as follows.

**Definition 7.8** (Key-slot graph)**.** At any given moment, we use $A_1$ to denote the set of $n$ keys in the current key set; use $A_2$ to denote $n$ random keys in $[U] \setminus A_1$, in which we will insert a random one in the next insertion. The ***key-slot graph*** of the given state is a bipartite graph $G$ with $A \coloneqq A_1 \cup A_2$ being the left-vertices and the set $[N]$ of all slots being the right-vertices. For each key $a \in A$ and each slot $s \in [N]$, if $\ell(a, s) < L$, there is an edge in $G$ connecting them with level $\ell(a, s)$.

**Definition 7.9.** For each vertex $x \in G$, we define its ***level-$i$ degree*** as the number of edges with level *at most* $i$ associated with $x$, written $\deg_i(x)$. We say vertex $x$ is ***low-degree***, if for all $0 \le i < L$, there is $\deg_i(x) \le 2^{2^{L+i+5}}$. Otherwise, we say $x$ is ***high-degree***.

Note that the level-$i$ degree of a *left-vertex* $a \in A$ is at most $2^{2^{L+i}}$ deterministically, so all left-vertices are low-degree. For right-vertices, the expected level-$i$ degree is also bounded by $\Theta\big((2^{2^{L+i}})^{10}\big)$ since the probe-sequence function is nearly uniform.

**Lemma 7.10.** *Let $S$ be the set of high-degree (right-)vertices in $G$. With probability $1 - 1/\operatorname{poly} L$, the total number of neighbors of vertices in $S$ is*

$$\sum_{s \in S} \deg_{L-1}(s) \le \frac{n}{2^{\Omega(2^{2^L})}}.$$

*Proof.* We fix a right-vertex $s \in [N]$ and analyze its level-$i$ degree. Since $U$ is a large polynomial of $n$, without loss of generality, the left-vertices can be viewed as $2n$ independently random keys from $[U]$, written $\{a_1, \ldots, a_{2n}\}$. Let $X_k \coloneqq \mathbb{1}[\ell(a_k, s) \le i]$, then

$$\mathbb{E}[X_k] = \Pr_{a_k \in [U]}\big[\text{probe-complexity}(a_k, s) \le 2^{2^{L+i}}\big] \le \Theta\left(\frac{(2^{2^{L+i}})^{10}}{n}\right) \le \frac{2^{2^{L+i+4}}}{n}$$

by the near-uniformity of the probe-sequence function. Then, $X \coloneqq \sum_{k=1}^{2n} X_k$ is the level-$i$ degree of $s$, whose expectation does not exceed $O(2^{2^{L+i+4}})$. By a Chernoff bound, for each $D \ge \Omega(2^{2^{L+i+4}})$,

$$\Pr[X > D] \le 2^{-\Omega(D)}. \tag{8}$$

Substituting $D = \Theta(2^{2^{L+i+5}})$ into the inequality, we know

$$\Pr[X > 2^{2^{L+i+5}}] \le 2^{-\Omega(2^{2^{L+i+5}})} \le 2^{-\Omega(2^{2^L})}.$$

Taking a union bound over $0 \le i < L$, we know that

$$\Pr[s \text{ is high-degree}] \le 2^{-\Omega(2^{2^L})} \cdot L = 2^{-\Omega(2^{2^L})}. \tag{9}$$

Moreover, taking summation of (8) over all integers $D > 2^{2^{L+i+5}}$, we know

$$\mathbb{E}\Big[\deg_i(s) \cdot \mathbb{1}\big[\deg_i(s) > 2^{2^{L+i+5}}\big]\Big] \le 2^{-\Omega(2^{2^L})}. \tag{10}$$

To bound the number of neighbors of $S$ (the set of high-degree nodes), we divide $S$ into two parts: Let $S_1$ be the set of vertices $x$ satisfying $\deg_{L-1}(x) > 2^{2^{2L+4}}$, i.e., they are high-degree because their level-$(L-1)$ degrees are too large; let $S_2 = S \setminus S_1$. We bound the number of neighbors of the two parts separately.

For $S_1$, we take summation of (10) over all right-vertices with $i = L - 1$, obtaining

$$\mathbb{E}\left[\sum_{s\in S_1} \deg_{L-1}(s)\right] \le n \cdot 2^{-\Omega(2^{2^L})}. \tag{11}$$

For $S_2$, we take summation of (9) to obtain

$$\mathbb{E}\left[\sum_{s\in S_2} \deg_{L-1}(s)\right] \le \mathbb{E}[|S_2|] \cdot 2^{2^{2L+4}} \le n \cdot 2^{-\Omega(2^{2^L})} \cdot 2^{2^{2L+4}} = n \cdot 2^{-\Omega(2^{2^L})}. \tag{12}$$

Adding (11) and (12) together and applying a Markov inequality, the lemma follows. □

***i*-short paths.** Assume $s_1, \ldots, s_j$ is an $(i + 10)$-stanza with positive potential, and $x_k$ is stored in slot $s_k$ for $k \in [j - 1]$. Then, there must be a path in $G$ from $x_1$ to $s_j$ consisting of:

- no edges with level $> i$;
- at most 1 edge with level $i$;
- at most $\sqrt{2}$ edges with level $i - 1$;
- at most $\sqrt{2}^2$ edges with level $i - 2$;
- $\cdots$

Formally, for every integer $0 \le \delta \le i$, the number of edges with level $i - \delta$ in the path cannot exceed $\sqrt{2}^{\delta}$. We say a path is ***i*-short** if it satisfies this condition.

Next, we show that a random key $a_1 \in A_1$ or $a_2 \in A_2$ is unlikely to reach any right-vertex (slot) $s \in [N]$ with level $\ge i + 10$ via an $i$-short path.

**Lemma 7.11.** *Let $a_1 \in A_1$ and $a_2 \in A_2$ be random keys, and $0 \le i \le L - 10$ be an integer. With probability $1 - 1/\operatorname{poly} L$, neither $a_1$ nor $a_2$ can reach a right-vertex (slot) in the key-slot graph $G$ with level $\ge i + 10$ via an $i$-short path.*

*Proof.* We first build an auxiliary graph $G'$ from $G$ as follows:

- for every high-degree vertex $s$, we color all its neighbors in black;
- for each right-vertex $s \in [N]$ with level $\ge i + 10$, we color it in black;
- we delete all high-degree vertices.

After these modifications, we call the obtained graph $G'$, which contains a subset of vertices colored black. There are two properties of $G'$:

1. every node in $G'$ is low-degree;
2. if a node $a \in A$ can reach a slot $s \in [N]$ in $G$ with level $\ge i + 10$ via an $i$-short path, then it can reach a black node in $G'$ via an $i$-short path (including the case where $a$ itself was colored black).

The second property is because we have colored all neighbors of the deleted nodes in black, thus any valid $i$-short path in $G$ will still lead to a black node in $G'$.

Next, we count the number of vertices that can be reached from a black node via an $i$-short path. Since every node in $G'$ is low-degree, fixing a black node as the starting node, an $i$-short path of length $p > 0$ can be encoded with the following parameters:

$$\ell_1, \dots, \ell_p \in [0, L-10] \cap \mathbb{Z}, \qquad t_k \in \left[2^{2^{L+\ell_k+5}}\right] \text{ for } k \in [p],$$
$$\text{s.t.} \qquad \{k \in [p] \mid \ell_k = i - \delta\} \le \sqrt{2}^{\delta} \text{ for all } 0 \le \delta \le i.$$

Given these parameters, the $i$-short path is determined edge-by-edge: $\ell_k$ denotes the level of the $k$-th edge on the path; $t_k$ indicates one of the associated level-$\ell_k$ edges of the $k$-th node on the path. We upper bound the number $C_1$ of different configurations for $(\ell_1, \dots, \ell_p)$ by enumerating $q_\delta := |\{k \in [p] \mid \ell_k = i - \delta\}| \in \left[0, \sqrt{2}^{\delta}\right] \cap \mathbb{Z}$:

$$\begin{aligned}
C_1 &\le \sum_{q_0=0}^{\lfloor\sqrt{2}^0\rfloor} \sum_{q_1=0}^{\lfloor\sqrt{2}^1\rfloor} \sum_{q_2=0}^{\lfloor\sqrt{2}^2\rfloor} \cdots \sum_{q_i=0}^{\lfloor\sqrt{2}^i\rfloor} \binom{q_0 + q_1 + \cdots + q_i}{q_0,\, q_1,\, \dots,\, q_i} \\
&\le \prod_{\delta=0}^{i} \left(\sqrt{2}^{\delta} + 1\right) \cdot \binom{\lfloor\sqrt{2}^0\rfloor + \lfloor\sqrt{2}^1\rfloor + \cdots + \lfloor\sqrt{2}^i\rfloor}{\lfloor\sqrt{2}^0\rfloor, \lfloor\sqrt{2}^1\rfloor, \dots, \lfloor\sqrt{2}^i\rfloor} \\
&\le 2^{O(L^2)} \cdot \prod_{j=1}^{i} \binom{\lfloor\sqrt{2}^0\rfloor + \cdots + \lfloor\sqrt{2}^j\rfloor}{\lfloor\sqrt{2}^j\rfloor} \\
&\le 2^{O(L^2)} \cdot \prod_{j=1}^{i} 2^{O(\sqrt{2}^j)} \\
&\le 2^{O(L^2)} \cdot 2^{O(2^{L/2})} = 2^{o(2^L)}.
\end{aligned}$$

Given any configuration of $(\ell_1, \dots, \ell_p)$, the number of configurations for $(t_1, \dots, t_p)$ is bounded by

$$C_2 \le \prod_{\delta=0}^{i} \left(2^{2^{L+i-\delta+5}}\right)^{\sqrt{2}^{\delta}} = 2^{(2^5 \cdot \sum_{\delta=0}^{i} 2^{L+i-\delta} \cdot 2^{\delta/2})} = 2^{(32 \cdot \sum_{\delta=0}^{i} 2^{L+i-\delta/2})} \le 2^{(32 \cdot \frac{1}{1-1/\sqrt{2}})(2^{L+i})} \ll 2^{110 \cdot 2^{L+i}}.$$

Therefore, the number of reachable nodes from any given black node is at most

$$C_1 \cdot C_2 \le 2^{(110+o(1)) \cdot 2^{L+i}} \ll 2^{111 \cdot 2^{L+i}}. \tag{13}$$

Lastly, we bound the number of black nodes, which consist of three parts:

1. The neighbors of high-degree nodes. According to Lemma 7.10, with probability $1 - 1/\operatorname{poly} L$, the number of these black nodes does not exceed $n/2^{\Omega(2^{2^L})}$.

2. The non-empty slots with level $\ge i + 10$. Since the expected average probe complexity over all non-empty slots is at most $O\left(2^{\sqrt{\log \varepsilon^{-1}}}\right) = O(2^{2^L})$, with probability $1 - 1/\operatorname{poly} L$, the total probe complexity is at most $O\left(n \cdot 2^{2^L} \cdot \operatorname{poly} L\right)$; every non-empty slot with level $\ge i + 10$ has probe complexity at least $2^{2^{L+i+9}}$, thus the number of such slots does not exceed

$$\frac{O\left(n \cdot 2^{2^L} \cdot \operatorname{poly} L\right)}{2^{2^{L+i+9}}} \ll \frac{n}{2^{510 \cdot 2^{L+i}}}.$$

3. The empty slots (with level $L$). There are at most

$$O(\varepsilon n) \le O(n/2^{2^{2L-1}}) \le O(n/2^{2^{L+i+9}}) \ll \frac{n}{2^{511\cdot 2^{L+i}}}$$

empty slots as the load factor equals $1-\varepsilon$.

Adding them together, we know that the number of black nodes is at most $O(n/2^{510\cdot 2^{L+i}})$ with probability $1-1/\operatorname{poly} L$. Multiplying with (13), we conclude that with probability $1-1/\operatorname{poly} L$, only $n/2^{399\cdot 2^{L+i}} = n/2^{\Omega(2^{L+i})}$ nodes can be reached via an $i$-short path from a black node. Finally, since $a_1 \in A_1$ and $a_2 \in A_2$ are sampled randomly, they do not belong to the reachable nodes with probability $1-1/\operatorname{poly} L$, thus the lemma holds. □

Based on Lemma 7.11, we can now analyze the desired properties of $\Phi$.

**Lemma 7.12** (Property 1 of $\Phi$)**.** *Any insertion or deletion increases $\Phi$ by at most $1/\operatorname{poly} L$ in expectation.*

*Proof.* Suppose we insert a random key $a \in A_2$ to the current hash table. For each $0 \le i \le L-10$, $\Phi_{i+10}$ is increased by at most 1. A necessary condition for $\Phi_{i+10}$ to be increased is that there exists an $(i+10)$-stanza, starting with the special slot $s_1$ containing the inserted key $a$, with positive potential. Say the stanza consists of slots $s_1, \ldots, s_j$ with $x_1, \ldots, x_{j-1}$ residing in slots $s_1, \ldots, s_{j-1}$ respectively, where $s_1$ is the special slot and $x_1 = a$. In the key-slot graph $G$ of the state before the insertion, this implies an $i$-short path from $a = x_1$ to $s_j$: $x_1 - s_2 - x_2 - \cdots - s_{j-1} - x_{j-1} - s_j$. The last vertex on the path is a slot with level $\ge i+10$. Due to Lemma 7.11, the probability of existing such a path is at most $1/\operatorname{poly} L$, i.e., $\Phi_i$ increases by $1/\operatorname{poly} L$ in expectation. Taking summation over all $i$, we know that $\Phi$ increases by $1/\operatorname{poly} L$ in expectation as well.

Similarly, suppose we delete a random key $a \in A_1$ from the current hash table. For each $0 \le i \le L-10$, in order to increase $\Phi_{i+10}$ by at most 1, there must be an $(i+10)$-stanza $s_1, \ldots, s_j$, with $x_k$ stored in $s_k$ for $k \in [j-1]$, where $s_j$ is the slot containing key $a$ before the deletion ($s_j$ becomes empty after the deletion and thus can serve as the final slot). In the key-slot graph $G$ of the state before the deletion, the stanza corresponds to an $i$-short path $s_1 - x_2 - s_2 - x_3 - \cdots - x_{j-1} - s_j - a$. It is connecting the key $a$ to delete with a slot $s_1$ with level $\ge i+10$, thus by Lemma 7.11, such a path exists with probability at most $1/\operatorname{poly} L$. Taking summation over $0 \le i \le L-10$, we conclude that each deletion increases $\Phi$ by at most $1/\operatorname{poly} L$, and the lemma follows. □

**Lemma 7.13** (Property 2 of $\Phi$)**.** *If the algorithm performs a move with impact $r$, $\Phi$ will be decreased by $r \pm O(1)$.*

*Proof.* Without loss of generality, we assume the algorithm can only (a) move a key from a non-special slot to the special slot, and (b) move a key from the special slot to a non-special slot. This is because any move between non-special slots can be replaced by two moves of types (a) and (b) respectively. By symmetry, it suffices to analyze the moves of type (b).

Suppose the algorithm moves a key $x$ from the special slot $s_1$ to a non-special slot $s_2$, where $\ell(x, s_2) = j$. The move has impact $r = L - j$. For the sake of discussion, we denote by $\Sigma$ the state of the hash table before the move, and $\Sigma'$ the state after the move. Let $\Phi$ (resp. $\Phi'$) be the potential of $\Sigma$ (resp. $\Sigma'$), and let $\Phi_i$ (resp. $\Phi'_i$) be the $i$-th summation term in $\Phi$ (resp. $\Phi'$).

We will show that for each $10 \le i \le L$:

1. if $i \le j$, then $\Phi'_i = \Phi_i$;
2. if $j < i < j+10$, then $\Phi'_i - \Phi_i \in [-2, 0]$;

3. if $i \geq j + 10$, then $\Phi'_i - \Phi_i \in \left[-1 - O(\sqrt{2}^{j-i}), -1 + O(\sqrt{2}^{j-i})\right]$.

**Case 1.** If we represent every stanza $(\widetilde{s}_1, \widetilde{s}_2, \ldots, \widetilde{s}_j)$ using the tuple $(x_1, \widetilde{s}_2, \widetilde{s}_3, \ldots, \widetilde{s}_j)$, where $x_1$ is the key residing in $\widetilde{s}_1$, then the set of representations of valid $i$-stanzas in $\Sigma$ and $\Sigma'$ are the same, also with the same potentials. Therefore, $\Phi'_i = \Phi_i$ holds.

**Case 2.** The only changes from $\Sigma$ to $\Sigma'$ are:

- $s_1$ is no longer a valid starting slot;
- $s_2$ is no longer a valid final slot.

Every valid $i$-stanza in $\Sigma'$ is also valid in $\Sigma$ with the same potential, so $\Phi'_i \leq \Phi_i$; each of the above two changes will invalidate at most 1 stanza from the collection of disjoint $i$-stanzas in $\Sigma$, each decreasing the potential by at most 1, so $\Phi'_i \geq \Phi_i - 2$. So the statement also holds.

**Case 3.** Let $C$ be a collection of disjoint $i$-stanzas in $\Sigma$ with the maximum (sum of) potential. Let $s_1 \circ c_1$ be the stanza in $C$ that starts with the special slot $s_1$, where $c_1$ is a sequence of slots, if such a stanza exists; let $c_2 \circ s_2$ be the stanza in $C$ that ends with $s_2$, where $c_2$ is a sequence of slots, if such a stanza exists.

We first adjust $C$ to make sure that both stanzas $s_1 \circ c_1$ and $c_2 \circ s_2$ exist while decreasing the potential of $C$ by at most $O(\sqrt{2}^{j-i})$. If $s_1 \circ c_1$ does not exist, we remove any stanza ending with $s_2$, decreasing the potential by at most 1; then add the stanza $(s_1, s_2)$ with potential $1 - \sqrt{2}^{j-i+10}$. Similarly, if $c_2 \circ s_2$ does not exist, we remove any stanza starting with $s_1$ and add the stanza $(s_1, s_2)$. Further, if $s_1 \circ c_1$ and $c_2 \circ s_2$ are the same stanza, then we replace it with stanza $(s_1, s_2)$, which can only decrease the potential by at most $\sqrt{2}^{j-i+10}$ as well. Below, we assume $C$ is the adjusted collection of disjoint stanzas with potential at least $\Phi_i - O(\sqrt{2}^{j-i})$.

Then, we show $\Phi'_i \geq \Phi_i - 1 - O(\sqrt{2}^{j-i})$ by constructing a set $C'$ of disjoint stanzas in $\Sigma'$. If $(s_1, s_2) \in C$, $C' = C \setminus \{(s_1, s_2)\}$ has potential at least $\Phi_i - 1$. Otherwise, $C$ must contain two different stanzas $s_1 \circ c_1$ and $c_2 \circ s_2$. We let $C' = C \setminus \{s_1 \circ c_1, c_2 \circ s_2\} \cup \{c_2 \circ s_2 \circ c_1\}$ and show the following facts:

- The new stanza $c_2 \circ s_2 \circ c_1$ is a valid $i$-stanza in $\Sigma'$, because:
  - The first slot in $c_2$ and the last slot in $c_1$ have levels $\geq i$, as they serve as starting and final slots of stanzas in $\Sigma$, and their accommodated keys are unchanged in $\Sigma'$.
  - The internal slots of the new stanza have levels at most $i - 10$. For $s_2$, this is because $\ell(x, s_2) = j \leq i - 10$; for other slots, it is because they are internal slots of the two original stanzas.
  - Every key in the new stanza (except the key residing in the last slot of $c_1$, if there is one) can be stored in the subsequent slot with level at most $i - 10$. For key $x$ in slot $s_2$, this is because the stanza $s_1 \circ c_1$ requires that $\ell(x, \text{first slot of } c_1) \leq i - 10$; for other keys, it is a requirement of stanzas $s_1 \circ c_1$ and $c_2 \circ s_2$ in $\Sigma'$.
- The potential of the new stanza satisfies $\phi(c_2 \circ s_2 \circ c_1) = \phi(s_1 \circ c_1) + \phi(c_2 \circ s_2) - 1 - O(\sqrt{2}^{j-i})$, because if we compare the summations in Definition 7.6 for the three stanzas, there will be only one extra term for $c_2 \circ s_2 \circ c_1$: $\sqrt{2}^{\ell(x,s_2)-i+10} = \sqrt{2}^{j-i+10} = O(\sqrt{2}^{j-i})$.

These facts imply that $C'$ is a valid set of disjoint stanzas in $\Sigma'$ with potential at least $\Phi_i - 1 - O(\sqrt{2}^{j-i})$.

Finally, we show that $\Phi_i \geq \Phi'_i + 1 - O(\sqrt{2}^{j-i})$. Similar to above, we let $\widetilde{C}'$ be the collection of $i$-stanzas in $\Sigma'$ with the maximum sum of potential $\Phi'$. If $\widetilde{C}'$ does not contain a stanza using slot $s_2$, we simply set $\widetilde{C} = \widetilde{C}' \cup \{(s_1, s_2)\}$ to be a collection of disjoint $i$-stanzas in $\Sigma$, with potential $\Phi'_i + 1 - O(\sqrt{2}^{j-i})$. Otherwise, we suppose there is a stanza $c_2 \circ s_2 \circ c_1$ using $s_2$ in $\widetilde{C}'$, as $s_2$ can only serve as an internal slot. We let $\widetilde{C} = \widetilde{C}' \setminus \{c_2 \circ s_2 \circ c_1\} \cup \{s_1 \circ c_1, c_2 \circ s_2\}$, and by a similar reasoning as above, we know the potential of $\widetilde{C}$ is at least $\Phi'_i + 1 - O(\sqrt{2}^{j-i})$.

Putting all three cases together, we take a summation of $\Phi'_i - \Phi_i$ over $10 \leq i \leq L$, which gives $\Phi' - \Phi = -(L - j) \pm O(1) = -r \pm O(1)$ and concludes the proof. □

Lastly, Property 3 of $\Phi$ is true, because each $\Phi_i$ corresponds to a collection of disjoint stanzas, whose size is at most $O(n)$, while each stanza has potential at most 1, thus $\Phi_i \leq O(n)$; taking a summation over $i$ shows $\Phi \leq O(nL)$.

So far, we have proved all 3 desired properties of the potential function $\Phi$. By Proposition 7.4, these properties imply Theorem 7.2.

## 7.2 Non-nearly-uniform probe sequences

The previous subsection proves the lower bound on the expected switching cost per operation based on the assumption of near uniformity. We recall that the probe-sequence function $h$ is ***nearly uniform*** if

$$q(h, i, s) := n \Pr_{x \in [U]}[\text{probe-complexity}(h, x, s) \leq i] \leq O(i^{10}) \tag{14}$$

for all $i$ and $s$. For clarity, we use probe-complexity$(h, x, s)$ to denote the probe complexity of key $x$ storing in slot $s$ under the probe-sequence function $h$. In this subsection, we reduce every probe-sequence function $h$ to a nearly uniform one $h'$, changing the expected average probe complexity by only a constant factor. This will remove the assumption of near uniformity in Theorem 7.2.

Formally, let $A$ be an assignment of $n$ keys to $N$ slots. We denote by $c(A, h)$ the total probe complexity of all $n$ balls under the probe-sequence function $h$.

**Lemma 7.14.** *For every (deterministic) probe-sequence function $h$, we can construct a nearly uniform probe-sequence function $h'$, such that for any assignment $A$ of $n$ keys to $N$ slots, $c(A, h') \leq O(c(A, h) + n)$.*

By definition, the probe sequence $h(x) = (h_1(x), h_2(x), \ldots)$ contains at most one slot $h_i(x)$ on its $i$-th position. However, for the sake of discussion, we introduce ***generalized probe sequences*** $(\widetilde{h}_1(x), \widetilde{h}_2(x), \ldots)$, where on the $i$-th position of the sequence there is a set $\widetilde{h}_i(x)$ of slots. It is required that, among the first $i$ positions on the generalized probe sequence, the number of slots should not exceed $i$.

For a generalized-probe-sequence function $\widetilde{h}$, probe-complexity$(\widetilde{h}, x, s)$ is still defined as the first position $i$ containing slot $s$; $q(\widetilde{h}, i, s)$ and the notion of near uniformity are still defined according to (14). The proof consists of two steps: first, we construct a nearly uniform generalized-probe-sequence function $\widetilde{h}$ that meets the requirements; second, we show that any (nearly uniform) generalized probe sequence can be transformed into a (nearly uniform) probe sequence with little overhead.

**Step 1: Constructing generalized probe sequences.** We construct $\widetilde{h}$ by reassigning some occurrences of slots in the probe sequences to later positions. For each pair $(i, s)$ where $q(h, i, s) > i^5$, which we call a ***bad pair***, and for each key $x$ where $h_i(x) = s$, we move the occurrence of $s$ from $h_i(x)$ to a later position $h_{\lceil\sqrt{q(h,i,s)}\rceil}(x)$, resulting in generalized probe sequences $\widetilde{h}$.

Next, we show that $\widetilde{h}$ meets our requirements.

**Claim 7.15** (Probe complexity of $\widetilde{h}$)**.** *For any assignment $A$ of $n$ keys to $N$ slots, the total probe complexity of $\widetilde{h}$ is at most $c(A,\widetilde{h}) \le c(A,h) + O(n)$.*

*Proof.* For each integer $Q$ that is a power of two, the number of bad pairs $(i,s)$ where $q(h,i,s) \in [Q,2Q)$ is bounded by

$$\begin{aligned}
&\sum_{i=1}^{(2Q)^{1/5}} \left| \left\{ s \in [N] \;\middle|\; \Pr_{x\in[U]}\big[\text{probe-complexity}(h,x,s) \le i\big] = \Theta(Q/n) \right\} \right| \\
&\le \sum_{i=1}^{(2Q)^{1/5}} O(i\cdot n/Q) \\
&= O\big(n/Q^{3/5}\big),
\end{aligned}$$

where the first equality is because $\sum_{s\in[N]} \Pr_{x\in[U]}[\text{probe-complexity}(h,x,s) \le i]$ does not exceed $i$. For each of the bad pairs, moving the occurrence of $s$ to position $\lceil q(h,i,s)^{1/2} \rceil = \Theta(Q^{1/2})$ may increase the probe complexity of at most one key by $O(Q^{1/2})$ (potentially, the key $x$ stored in slot $s$ in the assignment $A$ will get a higher probe complexity). Taking a summation over all $Q$, we upper bound the increment on the total probe complexity by

$$c(A,\widetilde{h}) - c(A,h) \le \sum_{Q\ge 1 \text{ is power of two}} O\big(n/Q^{3/5}\big)\cdot O\big(Q^{1/2}\big) = O(n). \qquad \square$$

**Claim 7.16.** *$\widetilde{h}$ is nearly uniform.*

*Proof.* For each $i \ge 1$ and slot $s$, we have

$$\begin{aligned}
&\Pr_{x\in[U]}\big[\text{probe-complexity}(\widetilde{h},x,s) = i\big] \\
&= \Pr_{x\in[U]}\big[\text{probe-complexity}(h,x,s) = i \text{ and } (i,s) \text{ is not a bad pair}\big] \qquad (15) \\
&+ \Pr_{x\in[U]}\big[\text{probe-complexity}(\widetilde{h},x,s) = i \text{ and } s \in \widetilde{h}_i(x) \text{ is a newly-assigned element in } \widetilde{h}\big]. \qquad (16)
\end{aligned}$$

Eq. (15) is at most $i^5/n$ since $(i,s)$ is not bad. When the event in (16) occurs, there must be a position $i' < i$ where $q(h,i',s) = \Theta(i^2)$, i.e., $\Pr_{x\in[U]}[\text{probe-complexity}(h,x,s) \le i'] = \Theta(i^2/n)$. Thus, we can bound (16) by enumerating $i' < i$:

$$\begin{aligned}
\text{Eq. (16)} &= \sum_{i'<i} \Pr_{x\in[U]}\big[\text{probe-complexity}(h,x,s) = i' \text{ and } s = h_{i'}(x) \text{ is reassigned to } \widetilde{h}_i(x)\big] \\
&\le \sum_{i'<i} O(i^2/n) = O(i^3/n).
\end{aligned}$$

Adding (15) and (16) together, and taking a summation over $i \in [i_0]$, we get

$$\Pr_{x\in[U]}\big[\text{probe-complexity}(\widetilde{h},x,s) \le i_0\big] \le O(i_0^6/n),$$

so $\widetilde{h}$ is nearly uniform. $\square$

**Step 2: Transforming into (normal) probe sequences.** We have already constructed a generalized-probe-sequence function $\widetilde{h}$ that has low probe complexity and is nearly uniform. The last step is to transform it into a probe-sequence function.

**Claim 7.17.** *Let $\widetilde{h}$ be a nearly uniform generalized-probe-sequence function. There exists a nearly uniform probe-sequence function $h'$, such that for any assignment $A$ of $n$ keys to $N$ slots, the total probe complexity $c(A, h') \le 2c(A, \widetilde{h})$.*

*Proof.* The given function $\widetilde{h}$ maps each key $x$ to a generalized probe sequence $(\widetilde{h}_1(x), \widetilde{h}_2(x), \ldots)$, where each position accommodates a set of slots (possibly empty). For each $\widetilde{h}_i(x) = \varnothing$, we insert a single element `null` into it. After that, we write down all elements in the order they appear in the generalized probe sequence (when a position $\widetilde{h}_i(x)$ contains multiple slots, we write in an arbitrary order), forming a probe sequence consisting of $[N] \cup \{\texttt{null}\}$. We denote it by $h'$.

For any slot $s$ that occur in $\widetilde{h}_i(x)$, its position in $h'$ after the above transformation will be between $i$ and $2i$ (both included): it is at least $i$ because we inserted `null`s to make sure every position contains at least one element; it is at most $2i$ because (1) there can only be $i$ elements among the first $i$ positions, before we insert any `null`; (2) the number of `null`s we inserted to the first $i$ positions is at most $i$. This implies that

$$\text{probe-complexity}(\widetilde{h}, x, s) \le \text{probe-complexity}(h', x, s) \le 2 \cdot \text{probe-complexity}(\widetilde{h}, x, s)$$

for every key $x$ and slot $s$. So we conclude that (1) $h'$ is nearly uniform; (2) $c(A, h') \le 2c(A, \widetilde{h})$ for any assignment $A$ of keys. □

Combining Claims 7.15 to 7.17 together, we have proved Lemma 7.14.

### 7.3 Putting pieces together

Combining the results from the previous subsections, we can derive a formal lower bound on classical open-addressing.

Specifically, for any classical open-addressing hash table, we analyze its performance on Distribution 1. By Yao's minimax principle, we only need to consider deterministic hash tables, i.e., the probe-sequence function $h$ is fixed. Then, Lemma 7.14 transforms $h$ to a nearly uniform function $h'$ while only losing a constant factor on the average probe complexity on any state of the hash table. Finally, Theorem 7.2 shows a lower bound on the switching cost per operation, provided that the expected average probe complexity is small enough. We summarize the result as the following theorem.

**Theorem 7.1** (Restated)**.** *Suppose the universe size $U = \operatorname{poly} n$ is a large polynomial of $n$. If a classical open-addressing hash table stores $n$ keys with load factor $1 - \varepsilon$, then the expected amortized time per operation is at least $\Omega(\log\log \varepsilon^{-1})$. Moreover, as long as the expected query time is $O\big(2^{\sqrt{\log \varepsilon^{-1}}}\big)$, the expected amortized time per insertion/deletion is at least $\Omega(\log\log \varepsilon^{-1})$.*

## 8 Open Problems

We conclude the paper with several appealing open questions.

**Non-oblivious open addressing.** The first question concerns ***non-oblivious open addressing*** [15, 14]: this is a generalization of open-addressing in which queries are not constrained to follow any particular probe sequence. Instead, insertions/queries/deletions can be implemented arbitrarily subject to the constraint that the *state* of the data structure, at any given moment, is that of an open-addressed hash table. Formally, this means that, if the hash table is storing $n$ keys, then its state is an array with $n$ non-empty slots, where the non-empty slots contain some permutation of the keys being stored.

All classical (a.k.a. oblivious) open-addressed hash tables are also valid non-oblivious open-addressed hash tables. Thus the upper bounds in this paper also apply to the non-oblivious case. However, the *lower bounds* do not. This raises the following question: can a non-oblivious open-addressed hash table hope to achieve $O(1)$ expected-time queries while also achieving an expected insertion/deletion time of $o(\log\log\varepsilon^{-1})$?

**High-probability worst-case query time bounds.** One major direction in recent decades has been to develop open-addressed hash tables that support high load factors while also offering (high-probability) worst-case query time bounds. Using variations of Cuckoo hashing [13, 4, 16], one can achieve $O(\log\varepsilon^{-1})$ worst-case query time, while also supporting $f(\varepsilon^{-1})$ expected insertion/deletion time for some function $f$. One can also show using coupon-collector-style arguments that this $O(\log\varepsilon^{-1})$ bound is the best (worst-case) bound that one can hope for. What is not clear is whether one might also be able to ask for a very good insertion/deletion time. Can one achieve $O(\log\varepsilon^{-1})$ worst-case queries (w.h.p.) while also supporting $o(\varepsilon^{-1})$ expected insertion/deletion time? Or, more generally, can one hope to achieve (high-probability) worst-case query time $Q$ and expected insertion/deletion time $I$ for some $Q$ and $I$ satisfying $QI = o(\varepsilon^{-1})$? We conjecture that such a bound should not be possible.

**High-probability worst-case time bounds for all operations.** Finally, it is also interesting to consider the task of achieving high-probability worst-case time bounds for *all* operations (insertions, deletions, and queries). For example, if we consider a load factor of 1/2, what are the best (worst-case) bounds that a classical open-addressed hash table can hope to achieve (as a function of $n$)? Using results from the power of two choices [39, 12], one can achieve a bound of $O(\log\log n)$. Is this the best bound possible?

## References


[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. There is no fast single hashing algorithm. *Information Processing Letters*, 7(6):270–273, 1978.

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

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

[4] Tolson Bell and Alan Frieze. $O(1)$ insertion for random walk $d$-ary cuckoo hashing up to the load threshold. In *Proc. 65th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2024.

[5] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. Iceberg hashing: Optimizing many hash-table criteria at once. *Journal of the ACM*, 70(6):1–51, 2023.

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

[7] Michael A. Bender, Bradley C. Kuszmaul, and William Kuszmaul. Linear probing revisited: Tombstones mark the demise of primary clustering. In *Proc. 62nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1171–1182, 2022.

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

[9] Richard P. Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, 1973.

[10] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin Hood hashing (preliminary report). In *Proc. 26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 281–288, 1985.

[11] Alexander Conway, Martín Farach-Colton, and Philip Shilane. Optimal hashing in external memory. In *Proc. 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 39:1–39:14, 2018.

[12] Ketan Dalal, Luc Devroye, and Ebrahim Malalla. Two-way linear probing revisited. *Algorithms*, 16(500), 2023.

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

[14] Amos Fiat and Moni Naor. Implicit $O(1)$ probe search. *SIAM Journal on Computing*, 22(1):1–10, 1993.

[15] Amos Fiat, Moni Naor, Jeanette Schmidt, and Alan Siegel. Non-oblivious hashing. In *Proc. 20th ACM Symposium on Theory of Computing (STOC)*, pages 367–376, 1988.

[16] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

[17] Alan Frieze and Samantha Petti. Balanced allocation through random walk. *Information Processing Letters*, 131:39–43, 2018.

[18] Gaston H. Gonnet and J. Ian Munro. Efficient ordering of hash tables. *SIAM Journal on Computing*, 8(3):463–478, 1979.

[19] Michael T. Goodrich, Daniel S. Hirschberg, Michael Mitzenmacher, and Justin Thaler. Cache-oblivious dictionaries and multimaps with negligible failure probability. In *Proc. 1st Mediterranean Conference on Algorithms (MedAlg)*, pages 203–218, 2012.

[20] Leo J. Guibas and Endre Szemerédi. The analysis of double hashing. *Journal of Computer and System Sciences*, 16(2):226–274, April 1978.

[21] 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.

[22] John Iacono and Mihai Pătraşcu. Using hashing to solve the dictionary problem. In *Proc. 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 570–582, 2012.

[23] Morten Skaarup Jensen and Rasmus Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, November 2008.

[24] Donald E. Knuth. Notes on "open" addressing. Available online at https://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf, 1963.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

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

[27] William Kuszmaul. A hash table without hash functions, and how to get the most out of your random bits. In *Proc. 63rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 991–1001, 2022.

[28] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. Dynamic "succincter". In *Proc. 64th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1715–1733, 2023.

[29] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. Tight cell-probe lower bounds for dynamic succinct dictionaries. In *Proc. 64th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1842–1862, 2023.

[30] George Lueker and Mariko Molodowitch. More analysis of double hashing. In *Proc. 20th ACM Symposium on Theory of Computing (STOC)*, pages 354–359, 1988.

[31] Efrem G. Mallach. Scatter storage techniques: A unifying viewpoint and a method for reducing retrieval times. *The Computer Journal*, 20(2):137–140, 1977.

[32] W. D. Maurer. An improved hash code for scatter storage. *ACM*, 11(1):35–38, 1968.

[33] J. Ian Munro and Pedro Celis. Techniques for collision resolution in hash tables with open addressing. In *Proc. 1986 ACM Fall joint computer conference*, pages 601–610, 1986.

[34] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proc. 9th European Symposium on Algorithms (ESA)*, pages 121–133, 2001.

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

[36] Rajeev Raman and Srinivasa Rao Satti. Succinct dynamic dictionaries and trees. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 357–368, 2003.

[37] Jeffrey D. Ullman. A note on the efficiency of hashing functions. *Journal of the ACM*, 19(3):569–575, 1972.

[38] Elad Verbin and Qin Zhang. The limits of buffering: A tight lower bound for dynamic membership in the external memory model. *SIAM Journal on Computing*, 42(1):212–229, 2013.

[39] Berthold Vöcking. How asymmetry helps load balancing. *Journal of the ACM*, 50(4):568–589, July 2003.

[40] David Wajc. Negative association – definition, properties, and applications. Available online at https://www.cs.cmu.edu/~dwajc/notes/Negative%20Association.pdf, 2017.

[41] Wikipedia contributors. Quadratic probing. Available online at https://en.wikipedia.org/wiki/Quadratic_probing, 2023.

[42] Andrew C. Yao. Uniform hashing is optimal. *Journal of the ACM*, 32(3):687–693, 1985.