

Dynamic Dictionary with Subconstant Wasted Bits per Key

Tianxiao Li* Jingxun Liang† Huacheng Yu‡ Renfei Zhou§

Abstract

Dictionaries have been one of the central questions in data structures. A dictionary data structure maintains a set of key-value pairs under insertions and deletions such that given a query key, the data structure efficiently returns its value. The state-of-the-art dictionaries [BFK⁺22] store n key-value pairs with only $O(n \log^{(k)} n)$ bits of redundancy, and support all operations in $O(k)$ time, for $k \leq \log^* n$. It was recently shown to be optimal [LLYZ23b].

In this paper, we study the regime where the number of redundant bits is $R = o(n)$, and show that when R is at least $n/\text{poly} \log n$, all operations can be supported in $O(\log^* n + \log(n/R))$ time, matching the lower bound in this regime [LLYZ23b]. We present two data structures based on which range R is in. The data structure for $R < n/\log^{0.1} n$ utilizes a generalization of *adapters* studied in [BKP⁺22, LLYZ23a]. The data structure for $R \geq n/\log^{0.1} n$ is based on recursively hashing into buckets with logarithmic sizes.

*Institute for Interdisciplinary Information Sciences, Tsinghua University. litx20@mails.tsinghua.edu.cn.

†Institute for Interdisciplinary Information Sciences, Tsinghua University. liangjx20@mails.tsinghua.edu.cn.

‡Department of Computer Science, Princeton University. yuhch123@gmail.com.

§Institute for Interdisciplinary Information Sciences, Tsinghua University. zhourf20@mails.tsinghua.edu.cn.

1 Introduction

Dictionaries are among the most basic data structures. A dictionary maintains a set of key-value pairs, where the keys are distinct numbers from the key space $[U]$, and values are from the value space $[V]$. The data structure supports key-value pair insertions and deletions such that given a key $x \in [U]$, it is able to quickly return the value associated to x , or report that x does not exist in the current set.

Hash tables are textbook implementations of dictionaries. Universal hashing together with chaining gives linear space and constant expected time per operation. Extensive research has been conducted on dictionaries and hash tables [Knu73, ANS09, ANS10, BCF⁺23, DMPP06, DKM⁺94, DM90, FPSS03, FKS84, Knu63, LYY20, Pag01, PR04, Pät08, RR03, Yu20, BFK⁺22]. Modern hash tables occupy space close to the information theoretical limit,

$$\mathcal{B}(U, V, n) = \log \binom{U}{n} + n \log V,$$

to store n key-value pairs. A dictionary data structure that uses $\mathcal{B}(U, V, n) + rn$ bits of space is said to incur r *wasted bits per key*, and $R = rn$ is called the *redundancy*. The state-of-the-art hash tables by Bender, Farach-Colton, Kuzmaul, Kuzmaul and Liu [BFK⁺22] achieve $O(\log^{(k)} n)$ wasted bits per key¹ while supporting all operations in $O(k)$ time, for any $k \leq \log^* n$.² Interestingly, this was later shown to be optimal [LLYZ23b] in the cell-probe model. In particular, when a constant number of wasted bits per key is allowed, the optimal operational time is $\Theta(\log^* n)$.

On the other hand, the design of [BFK⁺22] “wastes” a constant number of bits per key in several components, making such a linear redundancy inherent to their hash tables. It was not known how to construct a nontrivial dictionary with $o(n)$ bits of redundancy and fast operational time. In this paper, we study the space-time tradeoff when only a $o(1)$ bits are wasted per key: can we achieve a low operational time when $r = o(1)$, ideally close to $O(\log^* n)$, or do the dictionaries intrinsically need to slow down significantly when r is sub-constant? The lower bound of [LLYZ23b] shows that when the redundancy R is $o(n)$, the operational time is at least $\Omega(\log^* n + \log(n/R))$, which does not rule out $O(\log^* n)$ -time when the redundancy is slightly sublinear.

The main result of this paper shows that this tradeoff is tight when R is not too small.

Theorem 1.1. *For $R > n/\log^{O(1)} n$, there is a dictionary data structure with redundancy R that supports all operations in $O(\log^* n + \log(n/R))$ expected time, in word RAM with word-size $w = \Omega(\log UV)$. Moreover, queries take constant expected time.*

In fact, we use two different constructions to cover the range of R between $n/\text{poly } \log n$ and n . The first data structure has $O(\log \log n)$ operational time when $R > n/\log^{O(1)} n$. It is based on the recently introduced *adapters* [LLYZ23a] and dynamic aB-trees (daB-trees). Adapters connect multiple variable-length data structures efficiently with *zero* extra redundancy, overcoming an essential obstacle in designing succinct dynamic data structures. Li, Liang, Yu and Zhou [LLYZ23a] presented an adapter connecting two variable-length data structures (two-way adapters). In this paper, we extend adapters to B -way for $B = O(\sqrt{\log n})$, and apply it recursively to obtain daB-trees with large branching factors for dictionary, showing another application of the adapters. The second data structure matches the lower bound when $R > n/\log^{0.1} n$. It is based on recursively hashing

¹ $\log^{(k)} n$ is the iterated logarithm.

²Moreover, the hash table of [BFK⁺22] has the claimed operational time in worst case with high probability (high-probability), and occupies space in terms of the *current* n (dynamic-resizing). We will not focus on these stronger properties in this paper.

into buckets that are logarithmically smaller in size, hence, constructing a tree of depth $O(\log^* n)$. For leaves (the base cases), we introduce *minimaps*. They are “mini-dictionaries” maintaining a small set of keys, which can be encoded in one word, together with their (potentially large) values, and achieving almost no redundancy and all operations in logarithmic time. Interestingly, B -way adapters and minimaps share similar constructions, while they play completely different roles in the data structures for the two regimes respectively.

We remark that our dictionary for $R > n/\log^{O(1)} n$ with $O(\log \log n)$ operational time is *strongly history-independent* (a.k.a. *uniquely representable*) if carefully implemented, which means that the memory state of the dictionary only depends on the set of stored key-value pairs as well as some random bits that the algorithm might use. This matches a lower bound in [LLYZ23b]: any strongly history-independent dictionary must have operational time $\Omega(\log \frac{n \log U}{R})$, for any $R \leq O(n \log U)$. Recently, Kuszmaul [Kus23] shows an optimal strongly history-independent dictionary with $\Theta(\log \frac{n \log U}{R})$ operational time for $R \in [n \log \log n, n \log n]$, which together with our result concludes the following theorem.

Theorem 1.2. *For $U = n^{1+\Theta(1)}$ and $n/\log^{O(1)} n \leq R \leq n \log n$, there is a strongly history-independent dynamic dictionary that stores n elements from universe $[U]$ with R bits of redundancy and $O(\log \frac{n \log U}{R})$ expected time per operation, which is optimal in the cell-probe (or word RAM) model with word-size $w = \Theta(\log U)$. The dictionary also achieves expected constant-time queries.*

1.1 Organization

In Section 2, we give an overview of the two data structures. In Section 3, we formalize the setup and present the constructions of B -way adapters and minimaps. Then the data structure via B -way adapter trees is presented in Section 4, and the data structure via multiple levels of hashing is presented in Section 5.

2 Overview

In this section, we focus on the case where U, V, n are bounded by polynomials of each other, and the word-size $w = \Theta(\log n)$. We start by giving a short summary of the hash table by Bender et al. [BFK⁺22]. The first step, similar to several prior works, is to hash into buckets that contain expected $K - K^{2/3}$ keys for some $K = \text{poly log } n$. Then by concentration, all buckets have at most K keys hashed to it with high probability. By allocating space that fits K key-value pairs to each bucket and handling them separately, this effectively reduces the number of keys to K , and have incurred wasted bits of roughly $1/K^{1/3}$ per key (from allocating space conservatively to each bucket).

Now within each bucket, we start to recursively hash the keys into smaller buckets, forming a tree structure where the buckets are the nodes. For a bucket of size S (i.e., that can store a total of S keys), its children in the tree will have size $\text{poly log } S$ each, i.e., the degree of this node is $S/\text{poly log } S$. The entire tree has k levels for some parameter k , hence, every leaf in the tree has size roughly $\log^{(k)} n$.

All keys (and their values) are only directly stored in the leaves. The entire tree rooted at a top-level bucket of size K has exactly K total slots to store the keys. If the hash function were able to completely evenly split the key set, then a key and its value can simply be stored in the leaf it hashes to, and can be located by looking up this leaf. However, a node u may hash more keys to some child v than others, resulting in v having more keys than its capacity, i.e., v overflows. In this case, some of the keys will be stored in the empty slots in v 's siblings. Provided that u does not

overflow, we can always find slots for all keys hashed to u . An extra pointer will be stored to locate a key that is stored in a different child, incurring a redundancy of $\log S$ bits when u has size S . A simple calculation shows that by hashing S keys into $S/\text{poly log } S$ buckets, the total number of keys that are “overflowed” to a different bucket is $S/\text{poly log } S$. Hence, the pointers incur $1/\text{poly log } S$ wasted bits per key on average. Finally, the leaf nodes are sufficiently small, and data structures with $O(\log^{(k+1)} n)$ wasted bits per key are used, e.g., by storing a pointer for each key. It turns out that the keys can be inserted or deleted by walking down the tree, taking $O(k)$ time.

2.1 Dictionaries for $R > n/\log^{0.1} n$

For $R > n/\log^{0.1} n$, one contribution of this work is to handle the leaves efficiently while incurring only $o(1)$ wasted bits per key. The leaves contribute a majority of the redundancy in [BFK⁺22]. Moreover, their data structure always needs to store a “query router” in each leaf, which points each key to a slot. The query router costs an average of $\Omega(1)$ wasted bits per key to distinguish all keys in addition to the pointers, even when k is close to $\log^* n$. This is unaffordable when only $o(1)$ wasted bits per key is allowed.

In this work, we introduce *minimaps*, and use them to store the key-value pairs in the leaves. Roughly speaking, a minimap maintains n key-value pairs with keys from $[U]$ and values from $[V]$, such that *the set of keys fits in one word*, i.e., $\log \binom{U}{n} \leq w$ (the values of U, V, n here may be different from those in the full dictionary data structure). Note that this task is nontrivial since U and n may be close but both super constant, and V can still be very large, e.g., one could think $n = \sqrt{w}$, $U = w^{O(1)}$ and $V = 2^w$. Hence, the main challenge here is to map the keys to n slots to store their *values*, while supporting efficient key insertions and deletions.³

To rephrase the task, the set of keys is cheap enough to encode in one word, we want to store their values in n slots with (almost) no redundancy such that given a key, we know where its value is stored by reading the entire key set, and when updating the key set by inserting or deleting one key, not many values need to be relocated. A trivial solution is to store the values in the increasing order of their keys. Since the entire key set can be achieved in constant time, given a key, we can compute its rank within the set, and hence, recover the index of the slot containing its value. However, this solution may take $O(n)$ time to update the key set, since we may need to shift all values by one slot when inserting or deleting a key with a small rank. Another solution is to keep the values in an arbitrary order while storing a pointer for each key to its value. This achieves constant-time update and query, but uses $O(n \log n)$ redundancy bits in total (this is what the prior work uses). Our (randomized) minimap achieves $O(\log n)$ expected time for insertion and deletion of a key-value pair, as well as constant lookup time. Moreover, it does *not* introduce *any* redundancy.

The design of a minimap is inspired by the consistent hashing [KLL⁺97], and has a very similar structure to a solution to *memoryless worker-task assigning* [BKP⁺22], a problem studied in distributed computing. Roughly speaking, we first map n keys and n slots to the unit circle randomly, and if a key x is right after a slot a , we store the value of x in a . Then we repeat this process on the remaining keys and slots using independent randomness. Each round matches a constant fraction of the keys and slots, hence, all keys are matched to slots after $O(\log n)$ rounds with high probability. One can show that after one update, at most one key in each round needs to be relocated to a different slot. Therefore, the update time is $O(\log n)$. Besides the random functions, which can be shared between all minimaps, this structure has no redundancy, since the locations of all values are entirely determined by the key set (and the hash functions).

³The values can be stored using the *spillover representation* [Pät08] to avoid the problem of rounding V to the next power of two.

Using minimaps, we construct a dictionary with sublinear redundancy. The main structure is similar to [BFK⁺22], which recursively hashes keys to logarithmically smaller buckets. Our construction also forms the same tree where each node corresponds to a bucket. We note that there are other technical changes in order to keep the redundancy $o(n)$. One such challenge comes from storing the keys: the fact that a key is hashed to a node carries information about the key, i.e., naively encoding the key would implicitly waste space. Hence, when encoding a key, we must *exploit* the fact that it has been hashed to this bucket (i.e., this fact shrinks the set of possible keys). A common solution to this is the technical of *quotienting* [Knu73, Pag01]. Extra care needs to be taken when there is multiple levels of hashing as in our structure.

To avoid issues that may arise with variable-size data structures, we always allocate a fixed amount of space to each node. Now not all keys are stored in the leaves, each internal node is also allocated a fixed number of slots. We calculate the *minimum* number of keys that will be hashed to a leaf with high probability, allocate and store exactly this number of keys in this leaf, and send the excessive keys to its parent node. For each internal node u in the tree, we also calculate the minimum number $m(u)$ of keys that will be hashed to u , and allocate a total of exactly $m(u)$ keys to u and all its descendants, i.e., the number of keys stored directly in u is determined by subtracting the total number of slots allocated to its children. When a child of u violates this minimum number, we will rehash the *subtree* rooted at u , which only happens with low probability (in terms of the size of u). Note that this strategy incurs another implicit redundancy by allowing a key to be stored in one of the many levels, i.e., for a given key, whether it is in the leaf or is stored directly in an ancestor is some information not determined by the input but implicitly determined by the data structure. However, a careful calculation shows that this redundancy is also negligible. By following this construction to the level such that each leaf roughly stores $O(n/R)$ keys, we achieve the claimed performance.

2.2 Dictionaries for $R < n/\log^{0.1} n$

The above strategy fails when the redundancy is $O(n/\log^{10} n)$. The primary reason is that this redundancy would require each leaf node to store (a large-)poly $\log n$ keys while incurring only a constant bits of redundancy. However, the minimap requires that the key set fits in one word, which does not hold when there are much more than $w = \log n$ keys.

After the first step of hashing into buckets of size $K = \text{poly log } n$, in this regime, we construct directly a data structure for each bucket with $O(1)$ bits of redundancy and supporting updates and queries in $O(\log \log n)$ time, based on the recently introduced *adapters* [LLYZ23a].

An adapter stores jointly B arrays D_1, \dots, D_B using exactly $\sum_{i=1}^B |D_i|$ entries (with no redundancy), such that one can access one entry of an array D_i , or efficiently resize an array by incrementing or decrementing its size (and hence also the total number of entries in all arrays) by one. The work of [LLYZ23a] designed a deterministic adapter for $B = 2$ (two-way adapters) such that the entries can be accessed in constant time, and each resizing operation can be performed in $O(\log(|D_1| + |D_2|))$ time. Adapters are useful to combine multiple components of a data structure, especially when each component has a variable length under updates. Each array D_i corresponds to the memory words of a data structure, and without incurring any redundancy, an adapter “connects” multiple data structures whose sizes may change.

In this paper, we generalize it to B -way adapters for larger B using the essentially same algorithm as *minimaps* described in the previous subsection, supporting accesses in constant time and resizing in $O(\log \sum |D_i|)$ time *in expectation* (now the construction is randomized). Roughly speaking, an adapter is to map the entries $\{(i, j) : i \in [B], j \in [|D_i|]\}$ to the indices $[\sum_{i=1}^B |D_i|]$ with no redundancy, i.e., the mapping only depends on the array sizes, such that when some $|D_i|$

changes by one, only very few entries are mapped to a different location in expectation.

The seminal work of Pătraşcu [Păt08] showed that augmented B -trees (aB-trees), a tree data structure with branching factor B where each node stores some auxiliary information on its label that only depends the labels of its children, can be succinctly encoded with only $O(1)$ bits of redundancy, and can be accessed with no overhead in time (for B not too large). In particular, it can be used to build a *static* dictionary for *each bucket* of size $K = \text{poly log } n$ with $O(1)$ redundancy and lookup time $O(\log_B K) = O(\log_B \log n)$. Two-way adapters were used to dynamize the augmented B -trees of Pătraşcu [Păt08] for $B = 2$ in [LLYZ23a]. Roughly speaking, the succinct representation of a static aB-tree is constructed bottom-up: Having constructed the data structures for the children of a node u , the data structure for u is obtained by first concatenating the data structures for the children of u , then encoding their labels succinctly together with the concatenation. The adapters are used to store them succinctly with no space overhead, and allow for fast resizing. It turns out that dynamizing aB-trees [LLYZ23a] results in another polynomial slowdown in the depth of the tree (in addition to the cost of adapters). For each bucket of size $\text{poly log } n$, a tree with branching factor $B = 2$ has depth $\log \log n$, and would be slowed down by a $\text{poly log log } n$ factor.

Now we have efficient B -way adapters, and they give the potential to dynamize augmented B -trees for larger B . In fact, we can set B to $\sqrt{w} = O(\sqrt{\log n})$ (the largest B that even the static aB-trees allow), and have a tree with constant depth. It turns out that B -way adapters alone are not enough to dynamize general aB-trees for large B . Since adapters only allow for incrementing or decrementing each data structure by one complete word (exactly w bits), and the last less-than- w bits of every component (an incomplete word) need to be handled separately also with no extra redundancy, it is not clear how to dynamically maintain B arrays that have incomplete words of arbitrary number of bits in $[0, w)$ that may resize.

One important fact that we crucially exploit in this work is that the aB-tree implementing dictionary *does not resize arbitrarily*, as the insertion and deletion of a key will always resize the data structure by $\log(U/n) \pm O(1)$ bits. By setting the entries size in the B -way adapter to be $\log(U/n)$, we always only need to resize by $O(1)$ bits in addition to one complete word. This allows us to maintain the incomplete words efficiently using the same strategy as the adapters.

The question of dynamizing general aB-trees for large B , where the incomplete words may resize arbitrarily, remains open. Resolving this question may improve the running times of several dynamic succinct data structures studied in [LLYZ23a].

3 Adapters

It is a fundamental problem to manage multiple variable-size data structures within a contiguous piece of memory. The main purpose of this section is to introduce *adapters* to help resolve this problem. The most important subroutine of adapters is to maintain an *address mapping* between multiple small pieces of memory and a large one, which is further abstracted as the *dynamic matching problem* which we will start with. Dynamic matching has various applications, including but not limited to the adapters. We note that the solution here uses the same idea as a memoryless worker-task assigning algorithm [BKP⁺22].

3.1 Dynamic Matching

Problem 3.1 (Dynamic matching). *The dynamic matching problem asks us to dynamically maintain an injection (a matching) σ from a set of balls $A \subseteq U$ to a set of bins $B \subseteq V$, under the insertions and deletions of balls and bins (the operations). It is promised that $|A| \leq |B|$ at any*

time. Furthermore, the matching σ is required to only depend on A, B (and possibly the random bits fixed in advance), written as $\sigma = \sigma_{A,B}$.

The cost of an operation is defined as the number of ball relocations. Suppose that after a single operation, the new ball set and bin set become A' and B' respectively, the cost is

$$\Delta(\sigma_{A,B}, \sigma_{A',B'}) := \sum_{a \in A \cup A'} \mathbb{1}[\sigma_{A,B}(a) \neq \sigma_{A',B'}(a)].$$

A (deterministic) solution to this problem is described by a collection of matchings $\{\sigma_{A,B}\}_{A \subseteq U, B \subseteq V, |A| \leq |B|}$, i.e., by assigning a matching to every possible pairs of ball set A and bin set B . We call it a *matching scheme*. A randomized matching scheme is a distribution over the deterministic ones.

The main goal of this subsection is to introduce a randomized matching scheme with expected cost $O(\log |A|)$, requiring only a small number of balls to be moved during each operation. We will sample the whole matching scheme $\{\sigma_{A,B}\}_{A,B}$ from a fixed distribution \mathcal{D} ; after it is sampled and fixed, the maintained matching is determined solely by A and B , i.e., the behavior of ball-relocations during each operation is deterministic.

Lemma 3.2. *There is a randomized matching scheme $\{\sigma_{A,B}\}_{A,B} \sim \mathcal{D}$ such that the expected cost of any single operation on (A, B) is $O(\log |A|)$.*

Matching Scheme. To prove Lemma 3.2, we first introduce a matching algorithm that takes (A, B) as input and computes the matching $\sigma_{A,B}$. Following that, we will analyze its expected cost on a single operation.

Our algorithm is inspired by the *consistent hashing* [KLL⁺97], but with multiple rounds. In each round, we use a hash function h_i to map all remaining balls and bins to random points on the unit circle (i.e., the interval $[0, 1]$ with its two endpoints connected to form a loop). Suppose $a \in A$ is a ball whose next point in the clockwise direction on the circle is a bin $b \in B$, then we create a matched pair (a, b) , i.e., we put ball a into bin b . Intuitively, we expect half of the balls to be matched in such a round, so the whole algorithm ends after $O(\log |A|)$ rounds with high probability. The formal description of this algorithm is shown in Algorithm 1.

Algorithm 1: Dynamic Matching Algorithm

```

1 Hash functions  $h_1, \dots, h_{|U|}$  are predetermined, where  $h_i$  maps  $U \cup V$  to the unit circle
2 Let  $A_i, B_i$  denote the sets of unmatched balls and bins before the  $i$ -th round, respectively
3  $A_1 \leftarrow A, B_1 \leftarrow B, i \leftarrow 1$ 
4 while  $A_i \neq \emptyset$  do
5    $A_{i+1} \leftarrow A_i, B_{i+1} \leftarrow B_i$ 
6   Hash all elements in  $A_i$  and  $B_i$  onto the circle according to  $h_i$ 
7   for  $a \in A_i$  do
8     if the element next to  $a$  in the clockwise direction on the circle is a bin  $b \in B_i$  then
9       Set  $\sigma_{A,B}(a) = b$ 
10      Remove  $a, b$  from  $A_{i+1}, B_{i+1}$ 
11    $i \leftarrow i + 1$ 

```

The hash functions $\{h_i\}_{i \geq 1}$ are predetermined and stored in the memory in advance, and remain fixed throughout all operations. Therefore, when we insert or delete an element in A or B , the images of other elements under h_i remain unchanged. For any fixed pair (A, B) , over the randomness of h_i , the order of $h_i(A \cup B)$ on the circle is uniformly at random.

The algorithm consists of multiple rounds, and denote by T the number of rounds. A worst-case upper bound is $T \leq |A|$, because in each round, at least one ball $a \in A$ will be matched. This also shows why using $|U|$ independent hash functions is always enough. Furthermore, we can prove that the expected number of rounds is $O(\log |A|)$.

Claim 3.3. *For any $A \subseteq U$, $B \subseteq V$ with $|A| \leq |B|$, if $\{h_i\}$ are uniformly random permutations, we have $\mathbb{E}[T] = O(\log |A|)$.*

Proof. Consider the number of pairs that are matched in the i -th round. We first list all $|A_i| + |B_i|$ points on the unit circle in clockwise order, starting from some globally fixed point. Denote by $x_1, \dots, x_{|A_i|+|B_i|}$ the elements on the list. Due to the randomness of the hash function h_i , the list follows uniform distribution over all $(|A_i| + |B_i|)!$ permutations of $A_i \cup B_i$. If for some $j \in [1, |A_i + B_i|]$, x_j is a ball while x_{j+1} is a bin, then we found a matched pair ($x_{|A_i|+|B_i|+1} := x_1$ for simplicity). For each of these events, the probability of it occurring is

$$\Pr[x_j \text{ is a ball} \wedge x_{j+1} \text{ is a bin}] = \frac{|A_i|}{|A_i| + |B_i|} \cdot \frac{|B_i|}{|A_i| + |B_i| - 1} > \frac{|A_i| \cdot |B_i|}{(|A_i| + |B_i|)^2}.$$

Multiplying by the number of events, the expected number of matched pairs we find in the i -th round is at least $\frac{|A_i| \cdot |B_i|}{(|A_i| + |B_i|)^2} \cdot (|A_i| + |B_i|) \geq \frac{|A_i|}{2}$. This implies $\mathbb{E}[|A_{i+1}|] \leq \frac{|A_i|}{2}$. Applying Markov's inequality on $|A_{i+1}|$, we know that $\Pr[|A_{i+1}| \leq \frac{3}{4}|A_i|] \geq \frac{1}{3}$. As the number of balls decreases by a constant factor $3/4$ with constant probability $1/3$ in each round, the entire process ends within $O(\log |A|)$ rounds in expectation. \square

Algorithm 1 has induced a matching scheme $\{\sigma_{A,B}\}_{A,B}$. We also know that the expected rounds of Algorithm 1 is $\mathbb{E}[T] = O(\log |A|)$. To prove Lemma 3.2, it remains to show that the cost of each operation on (A, B) is bounded by $O(T)$.

We use the matching scheme induced by Algorithm 1. Assume (A, B) is a pair of ball set and bin set, and (A', B') differs from (A, B) by a single element, i.e., (A', B') is obtained by inserting or deleting a single element in A or B . Let T and T' be the number of rounds when running Algorithm 1 on (A, B) and (A', B') , respectively. From Claim 3.3, we know that $\mathbb{E}[T] = O(\log |A|)$.

We still use A_i, B_i to denote the sets of remaining balls and bins just before the i -th round when running Algorithm 1 on (A, B) . Similarly, let A'_i, B'_i denote the corresponding sets when running on (A', B') . We will compare (A_i, B_i) with (A'_i, B'_i) , and show the following claim:

Claim 3.4. *For all $1 \leq i \leq \max(T, T')$, (A'_i, B'_i) differs from (A_i, B_i) by a single element. Moreover, only $O(T)$ elements have different matched counterparts when the matching algorithm is executed on (A', B') compared to (A, B) .*

Proof. When $i = 1$, it is true because $(A'_1, B'_1) = (A', B')$ is obtained by a single operation from $(A, B) = (A_1, B_1)$. Below, we assume the claim holds for a certain i , and prove it for $i + 1$.

(A'_i, B'_i) is obtained from (A_i, B_i) by inserting or removing a ball or bin. Due to the symmetry between insertions and deletions, as well as between balls and bins, let us only consider inserting a bin to obtain $B'_i = B_i \cup \{b^*\}$ for now. Consider how the matched pairs in the i -th round will change when we add a new bin b^* . There are three cases depending on the previous element of b^* on the circle, denoted by x . Specifically, x is the element adjacent to b^* in the *counterclockwise* direction.

1. x is a bin $x \in B_i$, so b^* stays unmatched and $(A'_{i+1}, B'_{i+1}) = (A_{i+1}, B_{i+1} \cup \{b^*\})$.
2. x is a ball $x \in A_i$ that is unmatched in round i before adding b^* . Now the algorithm will match x with b^* , which implies $(A'_{i+1}, B'_{i+1}) = (A_{i+1} \setminus \{x\}, B_{i+1})$.

3. x is a ball $x \in A_i$ that is matched to some other $b \neq b^*$ in this round before inserting b^* . Now x will be matched to b^* while b becomes unmatched. In this case, $(A'_{i+1}, B'_{i+1}) = (A_{i+1}, B_{i+1} \cup \{b\})$.

In all three cases, only $O(1)$ elements changed their matching in the i -th round, so the total number of such elements throughout all rounds is $O(\max(T, T'))$. Finally, observe that $T' \leq T + 1$, because (A'_{T+1}, B'_{T+1}) differs from (A_{T+1}, B_{T+1}) by a single element and $A_{T+1} = \emptyset$, thus $|A'_{T+1}| \leq 1$; since each round matches at least one ball, we know the algorithm must end within $T + 1$ rounds when running on (A', B') . \square

Finally, combining Claims 3.3 and 3.4 proves Lemma 3.2.

Remark 3.5. *In our application of dynamic matching in the later sections, we always have $|A| = |B|$, and will only insert or delete a ball and a bin simultaneously.*

3.2 Virtual Memory Model

Same as [LLYZ23a], we use a storage model called the *virtual memory model* to capture the essential memory-accessing behavior of variable-size data structures. Similar to the word RAM model, the memory consists of an infinite sequence of w -bit words, which are labeled with positive integers $\{1, 2, \dots\}$ as their *addresses*. One should view the memory as a tape that starts from word 1 and extends to infinity.

At any given time, the variable-size data structure is allowed to use a prefix of the memory string. Formally, there is a positive integer M indicating the number of available memory bits, and let $L := \lfloor M/w \rfloor$. Like in a word RAM with M bits, the data structure can read or write one of the first L words on the tape given its address $i \in [L]$, as well as the first $M - Lw = (M \bmod w)$ bits within the $(L + 1)$ -th word. The latter part is viewed as an *incomplete word*, which we also refer to as the *tail*; in contrast, the former part is called the *complete words*. We may use “accessing a word” to refer to reading or writing it. Accessing an arbitrary word (including the tail) counts as one *word-access*.

We also allow the data structure to change the memory size M via *allocations* and *releases*. Due to technical reasons, we only allow two types of allocations (releases): increasing (decreasing) M by 1 or w bits, but not between. The memory size M is stored by some outside entity and is always given to the algorithm when performing any operation.

The available part on the tape (i.e., the first L words plus $(M \bmod w)$ bits) is called a *virtual memory* (VM for short), which is responsible of storing variable-size data structures. Usually, we think allocations and releases are more expensive than word-accesses; similarly, word-accesses to VM are more expensive than arithmetic instructions, because there might be multi-level address translations between the VM and the physical memory. So we observe the following quantities to measure the time performance of a variable-size data structure stored in a VM:

1. the number of arithmetic instructions and lookup table queries it performs;
2. the number of word-accesses to the VM;
3. the number of allocations and releases (including both 1-bit and w -bit types).

Variable-size data structures in spillover representation. The *spillover representation*, first introduced in [Pät08], aims to avoid the 1-bit redundancy from rounding up the data structure to

an integer number of bits. It represents the data structure with a pair $(k, m) \in [K] \times \{0, 1\}^M$, where k is called the *spillover* and K is called the *spill universe*.

This representation can be naturally combined with the VM model when the represented data structure is dynamic and has variable size: We directly store the M memory bits into a virtual memory, supporting word-accesses, allocations and releases. Then, any update to the data structure (including changing its size) is mainly expressed as a sequence of VM operations. The spillover k , as well as K, M , is stored by some external entity, so we do not count the time taken to it when we analyze the stored data structure itself.

3.3 Adapters

Recall that we want to store B variable-size data structures D_1, \dots, D_B in a contiguous piece of memory. For now, let us focus on the simplest case: each D_i is encoded into $M_i = \ell_i w$ memory bits, which are stored in a VM, without tail or spillover. In simpler terms, we need to “concatenate” B smaller VMs (sub-VMs) with ℓ_i complete words and no tail, then store them into a larger VM (super-VM) with $L := \ell_1 + \dots + \ell_B$ words. In this subsection, we introduce *B-way adapters* to solve this simple case. Concatenating tails and spillovers involve more technical details and will be introduced later in Section 4.

A B -way adapter maintains an address mapping between B sub-VMs and the super-VM. Formally, we use (i, j) to indicate the j -th word in the i -th sub-VM, while the words in the super-VM are labeled with $\{1, 2, \dots, L\}$. Our adapter maintains a dynamic matching σ between $\{(i, j) : 1 \leq i \leq B, 1 \leq j \leq \ell_i\}$ and $[L]$, which only depends on $\ell := (\ell_1, \dots, \ell_B)$, written as $\sigma = \sigma_\ell$. Suppose $\sigma(i, j) = t$, then we store the content of word (i, j) into the t -th word in the super-VM. The metadata ℓ is stored outside and we assume free access to it. Using this design, storing the complete-word parts of B data structures incurs no redundancy.

The matching scheme we use is given in Section 3.1: By viewing the words in sub-VMs (i, j) as balls and the words in the super-VM as bins, it is clear that it is an instance of the dynamic matching Problem 3.1. We maintain the matching via lookup tables, which requires that $|\text{supp } \ell|$ is small enough. When we want to access the j -th word in the i -th sub-VM, we first query the lookup table to get $\sigma_\ell(i, j) = t$ in constant time, then access the t -th word in the super-VM.

When any of the sub-VM requests an allocation (or release), the super-VM should also perform the same request. Afterward, some of the words should be moved as the matching changes from σ_ℓ to $\sigma_{\ell'}$. The number of moved words equals

$$\Delta(\sigma_\ell, \sigma_{\ell'}) = \sum_{i,j} \mathbb{1}[\sigma_\ell(i, j) \neq \sigma_{\ell'}(i, j)].$$

By Lemma 3.2, its expectation is bounded by $O(\log L)$. That is, an allocation (or release) of a sub-VM is transformed to an allocation (or release) of the super-VM with $O(\log L)$ additional word-accesses in the super-VM in expectation.

We summarize the above discussion with the following lemma:

Lemma 3.6. *Assuming $\ell := (\ell_1, \dots, \ell_B)$ is stored outside and allows free access, B sub-VMs with ℓ_1, \dots, ℓ_B words respectively and no tails can be maintained within a single super-VM with $L := \sum_{i=1}^B \ell_i$ words, while*

- *a word-access on any sub-VM is simulated by a word-access on the super-VM;*
- *an allocation [resp. release] on a sub-VM is simulated by an allocation [resp. release] and $O(\log L)$ word-accesses in expectation on the super-VM.*

Moreover, if $L \leq L_{\max}$ always holds, we can precompute lookup tables occupying $O(BL_{\max}^{B+1})$ words in total, and complete the simulation within $O(1)$ computation cost for each access, and $O(\log L)$ expected computation cost for each allocation or release. The time of computing these lookup tables is $O((B + L_{\max})L_{\max}^{B+1})$.

Proof. Based on the discussion above, it remains to calculate the size of lookup tables. We precompute the following two lookup tables:

- Given ℓ and an address (i, j) in sub-VM, output the corresponding address $\sigma_{\ell}(i, j)$ in the super-VM.
- Given an allocation/release changing ℓ to ℓ' , find out the list of all word-moves in the super-VM during this operation, where each word-move is represented by a pair (t, t') , meaning that the content of word t should be moved to word t' .

The number of vectors ℓ is bounded by L_{\max}^B ; for every ℓ , there are $O(B)$ possible ℓ' that can be reached by a single operation. Each entry of the second table is a list of at most L_{\max} pairs, so the total size is bounded by BL_{\max}^{B+1} . It is also clear that the first table occupies only $O(L_{\max}^{B+1})$ words of memory.

These two tables can also be computed efficiently. To compute the first one, we only need to run Algorithm 1 on every possible ℓ , which takes $O(L_{\max}^{B+2})$ time. To compute the second one, we only need to compare $\sigma_{\ell}(\cdot, \cdot)$ with $\sigma_{\ell'}(\cdot, \cdot)$ for every adjacent pair (ℓ, ℓ') , which takes $O(BL_{\max}^{B+1})$ time. \square

3.4 Minimaps for Dictionary Problem

Another application of dynamic matching is maintaining key-value pairs with almost zero redundancy. Suppose we need to maintain n key-value pairs, supporting insertions and deletions of key-value pairs, and querying the associated value of a given key. Differently from the general dictionary problem, here we additionally require that $\log \binom{U}{n} = O(w)$, i.e., the whole key set can be encoded into $O(1)$ words. Note that the problem is non-trivial, as the value-universe V can still be large. We introduce variable-size data structures called *minimaps* for this regime, stated as follows.

Lemma 3.7 (Minimap). *Suppose $r > 0$ is an integer and the word-length is $w = \Omega(n \log(Ur) + \log V)$. There is a data structure that stores n key-value pairs from key-universe $[U]$ and value-universe $[V]$, which is encoded using M memory bits (stored in a VM) and a spillover with universe $K \leq 2r$, has at most $4/r$ bits of redundancy, $O(1)$ query time, and $O(\log n)$ expected insertion/deletion time. The data structure is allowed to access a lookup table of $O(Un \binom{U}{n} \cdot (2rn)^n)$ words that can be shared between multiple instances of minimaps with the same parameters U, V, n, r .*

Besides the dynamic matching technique, we will also use the following lemma from [Pät08], which formalizes the fundamental functionality of the spillover representation.

Lemma 3.8 ([Pät08, Lemma 3]). *Given an arbitrary set \mathcal{X} . Fixing a parameter $r > 0$, we can represent any element in \mathcal{X} by a pair $(k, m) \in [K] \times \{0, 1\}^M$ with $K \leq 2r$, incurring at most $2/r$ bits of redundancy. If $w = \Omega(\log |\mathcal{X}|)$, encoding and decoding only take $O(1)$ arithmetic operations.⁴*

The main idea of our data structure is to establish n “slots” in the memory, each storing a value in $[V]$. The order we store these values is maintained as a dynamic matching, i.e., once key x is matched with slot i , we store the value associated with x into slot i . During an insertion or deletion, Lemma 3.2 guarantees that only few values need to be relocated.

⁴The original lemma in [Pät08] required $r \leq |\mathcal{X}|$; however, when $r > |\mathcal{X}|$, it is easy to encode an element in \mathcal{X} with a spillover $k \in [|\mathcal{X}|]$ and no memory bits, which incurs no redundancy, so the lemma holds as well.

Proof of Lemma 3.7. First, we apply Lemma 3.8 on the value-universe $[V]$ with parameter $r' = rn$, obtaining a spillover representation $[V] \rightarrow [K^*] \times \{0, 1\}^{M^*}$ with $K^* \leq 2rn$. Then, every value in $[V]$ is represented by a pair $(k_i^*, m_i^*) \in [K^*] \times \{0, 1\}^{M^*}$. This step induces $\leq 2n/(rn) = 2/r$ bits of redundancy.

In the second step, we combine the key set and all n spillovers k_1^*, \dots, k_n^* . Formally, let $\mathbf{x} = (x_1, \dots, x_n, k_1^*, \dots, k_n^*)$, where $x_1 < x_2 < \dots < x_n$ are the n distinct keys to be stored, and $k_i^* \in [K^*]$ is the spillover of x_i 's value. Let \mathcal{X} be the set of all possible \mathbf{x} 's, then $|\mathcal{X}| \leq \binom{U}{n} \cdot (2rn)^n$. We apply Lemma 3.8 again on \mathcal{X} with parameter r , obtaining a spillover representation with spill universe $K \leq 2r$, incurring at most $2/r$ bits of redundancy. Also note that $\log|\mathcal{X}| \leq O(w)$, the memory bits from this step only occupy $O(1)$ words.

The third step is to partition $M^* \cdot n$ consecutive memory bits into n slots of M^* bits each, labeled from 1 to n . We maintain a dynamic matching between the key set $\{x_1, \dots, x_n\} \subset [U]$ and slots $\{1, \dots, n\}$. Once some key x_i is mapped to slot j , we store the memory bits m_i^* of x_i 's value in the j -th slot.

Finally, we *append* the memory bits from the second step ($O(1)$ words) to the $M^* \cdot n$ memory bits from the third step. The result is stored into a VM and forms the final memory bits of our data structure (minimap). The spillover of the minimap $k \in [K]$ is the spillover from the second step. Hence, the total redundancy does not exceed $4/r$ bits.

During an insertion or deletion, according to Lemma 3.2, only $O(\log n)$ slots need to relocate their contents, which takes $O(\log n)$ word-accesses. The outcome of the second step can be fully re-encoded as it takes only $O(1)$ word-accesses.

It remains to calculate the sizes of the lookup tables we need in this process:

- Given \mathbf{x} and a key x_i , decode the spillover k_i^* of its value, and find the address of the slot where x_i 's associated value is stored. It occupies $O(\binom{U}{n} \cdot (2rn)^n \cdot U)$ words.
- Given \mathbf{x} and a key x to be inserted or deleted, find out the updated \mathbf{x} and a sequence of slot-moves (which slot's content should be moved to which location) during this operation. It requires $O(n)$ words for every possible operation, thus occupies $O(\binom{U}{n} \cdot (2rn)^n \cdot U \cdot n)$ words.

Summing up, the lookup table occupies $O(Un \binom{U}{n} \cdot (2rn)^n)$ words, which concludes the proof. \square

4 Dictionary via Adapter Tree

Recall that a dictionary data structure maintains a set of N distinct keys from $[U]$ under key insertions and deletions, such that given a query $x \in [U]$, it returns if x is in this set. In some scenarios, the dictionaries may also need to store a $O(w)$ -bit string called the *value* for each key. For most algorithms including ours, storing values is not the main challenge, so we will focus on storing keys. Now fix an integer $100 \leq h \leq \log^{0.3} N$ and let $R = N/\log^h N$. In this section, we will design a dictionary with redundancy $O(R)$, which supports queries in $O(h^2)$ time, and updates in $O(h^3 \log \log N)$ time. When $h = \Theta(1)$, this result covers the parameter regime $R \leq N/\log^{0.1} N$ of Theorem 1.1.

Additionally, we assume $U = N^{1+\alpha}$ for some constant $\alpha > 0$; also assume $w = \Omega(\log N)$. Without loss of generality, we assume the dictionary always contains N or $N - 1$ keys, but not fewer.⁵

⁵This assumption is without loss of generality because we may insert N special elements e_1, \dots, e_N to the universe U , each indicating that the i -th key in the dictionary does not exist. If the number of keys is smaller than $N - 1$, we include a suffix of special elements in the dictionary to make the number of keys equal $N - 1$ or N at any time. The entropy increased by enlarging the universe is negligible: $1/\text{poly}(N)$.

Algorithm Framework. Let $B = O(\sqrt{\log N})$ be a parameter, $n_1 = O(B^{8h})$, and $n_{8h} = n_1/B^{8h-1} = O(B)$. Consider the following two-stage hashing: We first hash keys into N/n_1 buckets, which we call the *level-1 nodes* or *root nodes*, with the expected number of keys in each bucket being n_1 . For all keys that mapped to a level-1 node u , we further hash them into n_1/n_{8h} buckets, which we call *level-8h nodes* or *leaf nodes*, with the expected number of keys in each bucket being n_{8h} .

Next, we build a full B -ary tree between every root and its leaves. Formally, for $1 \leq \ell < 8h$, every level- ℓ node has exactly B children at level $\ell + 1$. We now get a forest where each key is finally hashed into a leaf node (through two-stage hashing). For a level- ℓ node u , we say a key x is hashed into u if x is hashed into a descendant leaf of u ; the expected number of keys hashed into u is denoted by $n_\ell := B^{8h-\ell}n_1$.

The design of our data structure is recursive: For a level- ℓ node u , we construct a variable-size data structure maintaining all keys hashed into u , which is stored in a *virtual memory* (VM). This is mainly done by “concatenating” the sub-VMs from its B children on the next level. The leaf nodes are directly maintained by *mini-maps* as the number of keys hashed into every leaf node is very small ($O(\sqrt{\log N})$). We call this recursive data structure *adapter trees*.

Tree Parameters. We specify some parameters here. For technical reason, we choose B to be the unique power of two in $[\sqrt{\log N}, 2\sqrt{\log N})$, and $n_1 \in [B^{8h}, 2B^{8h})$ be the unique number such that N/n_1 is a power of two (note that n_1 is *not* necessarily an integer). Correspondingly, $n_\ell = n_1/B^{\ell-1}$ is also not necessarily an integer.

Let $\bar{n}_1 = \lceil n_1 + n_1^{2/3} \rceil$ and $\underline{n}_1 = \lfloor n_1 - n_1^{2/3} \rfloor$. By carefully choosing the hash functions (as we will describe in Appendix C), we can ensure that the number of keys $n(u)$ in each level-1 node u is between $[\underline{n}_1, \bar{n}_1]$ with high probability. When designing our data structure, we always assume this to be true. Once some root u violates this condition, we say u *overflows* ($n(u) > \bar{n}_1$) or *underflows* ($n(u) < \underline{n}_1$), and reconstruct the whole dictionary immediately (using different hash functions). Since the probability is sufficiently small, the expected cost of reconstruction is negligible (this will be analyzed in Appendix C and Section 5.5). Similarly, we define $\bar{n}_{8h} = \lceil n_{8h} + n_{8h}^{2/3} \rceil$ and $\underline{n}_{8h} = \lfloor n_{8h} - n_{8h}^{2/3} \rfloor$, and always assume that the number of keys in each level-8h node is between $[\underline{n}_{8h}, \bar{n}_{8h}]$.

When N is sufficiently large, we know $\underline{n}_{8h} \geq n_{8h}/2$ and $\bar{n}_{8h} \leq 2 \cdot n_{8h}$. Thus, for any given level- ℓ node u , we know $n(u) \in [n_\ell/2, 2n_\ell]$, because the keys hashed to u is a union of those of u 's descendent leaves.

4.1 Hashing and ID-Quotient Separation

Imagine that we have n keys that are independent random bit strings. With probability $\geq 1 - 1/n^2$, the first $\lceil 4 \log n \rceil$ bits are all distinct, thus sufficient to distinguish the keys from each other. We call the first $\lceil 4 \log n \rceil$ bits the *identifier* (“ID” for short) of each key, and call the remaining bits the *quotient*. An original key can now be regarded as a *key-value pair*: the ID is enough to identify a key while the quotient is considered as the value associated with the ID.

A merit of separating IDs and quotients is that the IDs can be much shorter than the original keys, which enables us to use efficient data structures for them. Typically, storing associated values in a dictionary is much easier than storing the keys. By putting most of the bits in the quotient, we actually reduce the key universe a lot. For $n_{8h} = O(\log^{0.5} N)$ keys, it is possible to encode the whole ID set into a single word while supporting constant time operations based on lookup tables, so we can maintain the ID-quotient pairs using a *minimap*.

In our algorithm, the keys are represented in ID-quotient pairs since they were hashed into level-1 nodes. Formally, a level-1 node u is responsible for storing a subset of its universe $[2^{I_1}] \times [Q]$, where $I_1 := \lceil 4 \log n_1 \rceil$ is the length of the ID part, and Q is the range of the quotient part.⁶ The hash function for this hashing step is a *bijection* $h^{(1)} : [U] \rightarrow [N/n_1] \times [2^{I_1}] \times [Q]$, which maps a key $x \in [U]$ into a triple $(u, x_{\text{id}}, x_{\text{quot}})$, with u being the index of a level-1 node, x_{id} and x_{quot} being the ID and quotient of key x in the root node u , respectively. (We round up U to a multiple of $(N/n_1) \cdot 2^{I_1}$ so that $h^{(1)}$ can be a bijection, which only increases the entropy $\log \binom{U}{N}$ by a negligible value.)

The second hashing step further hashes each key from a level-1 node to level- $8h$ nodes. Benefiting from the fact that the keys are already represented as ID-quotient pairs, the second hashing only takes the ID x_{id} as input. Specifically, this hash function maps x_{id} (the ID of x in the root node u) into a pair (v, y_{id}) , where $v \in [B^{8h-1}]$ indicates a descendant leaf of the root u , and y_{id} is the ID of x in the leaf v . Formally, this hash function is a bijection $h^{(2)} : [2^{I_1}] \rightarrow [B^{8h-1}] \times [2^{I_{8h}}]$, where $I_{8h} = I_1 - (8h - 1) \log B$. The quotient remains the same, i.e., the ID-quotient pairs for x in the leaf node is $(y_{\text{id}}, x_{\text{quot}})$.

The hash function $h^{(1)}$ should satisfy two properties: (1) it should evenly distribute all keys to the target buckets, preventing overflow or underflow at any bucket; (2) for all keys that are hashed to the same bucket, they must be assigned different IDs in it. $h^{(2)}$ also needs to satisfy the first property. We defer the formal design of hash function families to Appendix C.

After two hashing steps, each leaf is responsible for storing a set of ID-quotient pairs from $[2^{I_{8h}}] \times [Q]$. We use $U_{8h} := 2^{I_{8h}} \cdot Q = U \cdot n_{8h}/N$ to denote the universe size of leaf nodes. Accordingly, we define $U_\ell := U_{8h} \cdot B^{8h-\ell}$, representing that any level- ℓ node u is responsible for storing $n(u)$ keys from a universe of U_ℓ elements. We assume $U_\ell \geq 4n_\ell$ for all $1 \leq \ell \leq 8h$ as N is sufficiently large, which also implies $U_\ell \geq 2n(u)$ for any level- ℓ node u .

4.2 Weak Virtual Memory

Recall that on each non-leaf node u of the tree, we plan to inductively maintain a variable-size data structure that stores the set of keys hashed to u , mainly by concatenating the sub-VMs from u 's children. The *adapters*, introduced in Section 3.3, can work efficiently when the sub-VMs have no *tails* (i.e., their lengths are integer numbers of words), allowing us to allocate or release w bits at a time. Unfortunately, we do not know any general method that can concatenate the tails under the same performance.

Toy method of concatenating tails. To develop intuition for *weak virtual memories* (introduced later), we start with a simple idea of concatenating the tails: treat them as bits and use adapters. Assume there are B sub-VMs, namely $m^{[1]}, \dots, m^{[B]}$, which we want to concatenate. Every $m^{[i]}$ can be regarded as a sequence of complete words plus a tail. We further treat the tail as a sequence of less than w bits. The simple method is to concatenate the complete words using one adapter while concatenating the *tail bits* using another adapter. The “outputs” of the two adapters are further organized into a super-VM.

This toy method has two issues. First, it only allows allocating one bit or w bits at a time, but not between. For instance, if we want to allocate $w/2$ bits, we need to repeatedly allocate 1 bit for $w/2$ times, which is unaffordably slow. Second, the tail of any given sub-VM $m^{[i]}$ is not mapped to consecutive bits in the super-VM, so accessing the tail of a sub-VM could require accessing up to $\Theta(w)$ different locations in the super-VM.

⁶We allow the quotient range not to be a power of two.

Before we resolve these two issues, we first introduce the *weak virtual memory* model, which appears naturally in the toy method.

Weak virtual memory. A *weak virtual memory* (WVM for short) consists of two tapes: the *word tape* contains a sequence of complete words of w bits each, whereas the *bit tape* contains a sequence of memory bits. Similar to VM, each word (resp. bit) in the word tape (resp. bit tape) is labeled by a positive integer in $[1, L_{\text{word}}]$ (resp. $[1, L_{\text{bit}}]$), which we call the *address* of the word (resp. bit), where L_{word} (resp. L_{bit}) is the number of words (resp. bits) in the word tape (resp. bit tape). The bits in the bit tape are called *extra bits*, and we stipulate that the number of extra bits $L_{\text{bit}} < 2w$.

In an *access* operation in WVM, one is allowed to either access a word in the word tape, or access an extra bit in the bit tape, given their addresses as inputs. In an *allocation* operation, one is allowed to either allocate a word in the word tape, or allocate an extra bit in the bit tape. The *release* operation is defined in the similar way.

Note that WVM is a weaker model than VM: each WVM can be simulated by a VM by storing the bit tape using the tail and possibly the last complete word. Any access of the WVM can be transformed into one access of the VM, whereas any allocation of the WVM can be transformed into one allocation plus $O(1)$ accesses of the VM.

Based on this model, the above toy method of concatenating tails can be described in the following way: Assume we want to concatenate B VMs $m^{[1]}, \dots, m^{[B]}$. We treat each of them as a WVM $\tilde{m}^{[i]}$ by storing $m^{[i]}$'s tail into the bit tape of $\tilde{m}^{[i]}$ (the word tape of $\tilde{m}^{[i]}$ still stores all complete words in the VM). Then, we concatenate all word tapes using one adapter, and concatenate all bit tapes using another adapter. The outcome is stored into a super-VM.

The two issues of the toy method are closely related to WVMs: The first issue is because WVM only supports allocating or releasing a single bit or a single word at a time, but not between. The second issue arises as we treat VMs as WVMs: An access of the tail in a VM becomes $O(w)$ accesses in the WVM, which is way more expensive than accessing a complete word in the VM (the latter is transformed to still one access in the WVM).

Feature on the VM size. Under the specific framework of our algorithm, we actually do not need allocations and releases of arbitrary length: supporting w -bit and 1-bit allocations is enough for us, which bypasses the first issue. Recall that a level- ℓ node u stores $n(u)$ keys from a universe U_ℓ , so the data structure at node u should use roughly $\log \binom{U_\ell}{n(u)}$ bits of space. When adding a new key to the subtree of u , the number of bits to allocate is around $\log \binom{U_\ell}{n(u)+1} - \log \binom{U_\ell}{n(u)}$. By choosing a proper word-length w , this number of bits to allocate is always $w + O(1)$, which allows us to allocate efficiently. This feature is formalized by the following claim:

Claim 4.1. *For each ℓ and $n_\ell/2 \leq n(u) \leq 2n_\ell$, we have*

$$\log \frac{U_{sh}}{n_{sh}} - 3 \leq \log \binom{U_\ell}{n(u)+1} - \log \binom{U_\ell}{n(u)} \leq \log \frac{U_{sh}}{n_{sh}} + 1; \quad (1)$$

$$\left(\log \frac{U_{sh}}{n_{sh}} - 1 \right) n(u) \leq \log \binom{U_\ell}{n(u)} \leq \left(\log \frac{U_{sh}}{n_{sh}} + 3 \right) n(u). \quad (2)$$

Proof. One can compute

$$\log \binom{U_\ell}{n(u)+1} - \log \binom{U_\ell}{n(u)} = \log \frac{U_\ell - n(u)}{n(u) + 1}. \quad (3)$$

Note that $n(u) \geq n_\ell/2$, $U_\ell - n(u) \geq U_\ell/2$, and $n(u) + 1 \leq 2n_\ell + 1 \leq 4n_\ell$, we have

$$\log \frac{U_\ell}{8n_\ell} \leq \log \frac{U_\ell - n(u)}{n(u) + 1} \leq \log \frac{2U_\ell}{n_\ell}.$$

Combining with (3) and $U_\ell/n_\ell = U_{8h}/n_{8h}$ gives (1).

To show (2), we observe that

$$n(u) \log \frac{U_\ell}{n(u)} \leq \log \binom{U_\ell}{n(u)} \leq n(u) \log \frac{U_\ell}{n(u)} + (\log e)n(u),$$

and

$$n(u) \log \frac{U_\ell}{n(u)} = n(u) \log \frac{U_\ell}{n_\ell} + n(u) \log \frac{n_\ell}{n(u)},$$

where $U_\ell/n_\ell = U_{8h}/n_{8h}$ and $\log \frac{n_\ell}{n(u)} \in [-1, 1]$ because $\frac{1}{2}n_\ell \leq n(u) \leq 2n_\ell$. Combining these two formulas together, we get

$$\log \binom{U_\ell}{n(u)} - n(u) \log \frac{U_{8h}}{n_{8h}} \in \left[-n(u), (1 + \log e)n(u) \right] \subset \left[-n(u), 3n(u) \right],$$

which proves (2). □

We choose $w := \lfloor \log \frac{U_{8h}}{n_{8h}} \rfloor - 5$ as the word-length⁷, then:

- (1) inserting (deleting) a key to u will cause an allocation (release) of one word plus $O(1)$ bits;
- (2) the space usage of u 's data structure roughly equals $n(u)$ words plus $O(n(u))$ bits.

Here, Condition (1) addresses the first issue of the toy method; Condition (2) will lead to the following property and benefit resolving the second issue.

Definition 4.2 (Word-dominant). *A VM is word-dominant if at any given time, there is a positive integer n , such that the current number of bits in the VM lies in $[nw + 3n, nw + 10n]$. For a word-dominant VM, we forbid allocations and releases that will result in a violation of this word-dominant condition.*

Random swap. As mentioned in the second issue, when we treat a VM as a WVM by storing the tail in the bit tape, accessing the tail of VM becomes very slow, as it would require $O(w)$ bit-accesses to the WVM. However, accessing any given complete word of VM is still efficient (transformed into only one word-access to the WVM). To resolve the second issue, we amortize the cost of tail-access by swapping the tail with a random complete word. Specifically, we choose a word $i \in [L_{\text{word}}]$ almost uniformly at random, and swap the tail with the prefix of word i of the same length L_{bit} before we store the “new tail” into the bit tape of WVM. As result, accessing the original tail of VM becomes efficient, since we only need to access one word in the WVM; accessing the original word i requires us to access the new tail, which costs $O(w)$ accesses to the WVM. Since i is randomly chosen, the expected cost of any given access operation is low. This idea is formalized as the following lemma.

⁷ w mainly determines how we partition memory bits into words in a VM. If the “physical” word-length is different, one can simulate the algorithm with word-length w at no additional cost since $w \leq \log U_{8h} = O(\log N)$.

Lemma 4.3 (Random swap). *We can store a word-dominant VM containing M bits ($2w < M < Nw$) into a WVM with the same total number of bits $wL_{\text{word}} + L_{\text{bit}} = M$, while any given operation to the VM can be transformed into several operations to the WVM within constant expected additional computation time. Moreover:*

- (a) *An access to the VM (either a complete word or the tail) is transformed into $1+O(\min(1, w^2/M))$ accesses to the WVM in expectation.*
- (b) *Allocating/releasing a word or a bit in the VM can be transformed into $O(1)$ allocation/release operations followed by $O(1)$ accesses to the WVM (both in expectation).*
- (c) *The transformation from VM to WVM does not introduce additional space usage, except that we need to access an $O(\log^2 N)$ -bit random seed which can be shared between multiple instances. L_{word} and L_{bit} are fully determined by M and the seed.*

This lemma shows that word-dominant VMs can be “simulated” by WVMs without much loss of efficiency. Its proof is deferred to Appendix A.

4.3 Inductive Construction of Adapter Tree

In this subsection, we will construct the core part of our dictionary – an adapter tree storing the set of keys in a level-1 bucket – by inductively aggregating the sub-VMs of children. Formally, we will prove the following statement inductively:

Statement 1. *Fix a level- ℓ node u and denote the number of keys hashed to u by $n(u)$. Fix parameter $r = n_1$. Assuming free access to $n(u)$, there is a data structure maintain the set of keys hashed to u with spill universe $[K(U_\ell, n(u))]$ and $M(U_\ell, n(u))$ memory bits, such that*

- $r < K(U_\ell, n(u)) \leq 2r$.
- $\log \binom{U_\ell}{n(u)} - 1 < M(U_\ell, n(u)) + \log K(U_\ell, n(u)) \leq \log \binom{U_\ell}{n(u)} + \frac{n_\ell - 1}{r}$.

Moreover, the $M(U_\ell, n(u))$ memory bits are stored in a VM.

We assume without loss of generality that the lower bounds $M(U_\ell, n(u)) + \log K(U_\ell, n(u)) > \log \binom{U_\ell}{n(u)} - 1$ and $K(U_\ell, n(u)) > r$ hold, because otherwise we can pad zeros to the end of the memory until $M(U_\ell, n(u)) + \log K(U_\ell, n(u)) > \log \binom{U_\ell}{n(u)} - 1$, or repeatedly include the last memory bit into the spillover⁸ until $r < K(U_\ell, n(u)) \leq 2r$. Hence, we will not prove these two inequalities in the following inductive proof.

Proof of Statement 1. We prove this by induction. The proof is similar to [LLYZ23a] and [Pät08], except that the WVMs and the random swap lemma will involve when we aggregate sub-VMs.

Base case. When $\ell = 8h$, for each level- $8h$ node u , there are $n(u)$ keys hashed to it, each is represented as an ID-quotient pair $(y_{\text{id}}, x_{\text{quot}})$. By regarding the ID as “key” and quotient as “value”, we use a minimap to maintain it: The set of IDs can be encoded into

$$n(u) \cdot I_{8h} \leq \bar{n}_{8h} \cdot I_1 = \Theta(B \cdot \log n_1) = \Theta(\sqrt{\log N} \cdot h \cdot \log \log N) \ll O(w)$$

bits. (Recall that $I_1 = \lceil 4 \log n_1 \rceil$ and $n_1 \leq 2B^{8h}$; the last inequality holds because $h \leq \log^{0.3} N$.) Applying Lemma 3.7 with key-universe $[2^{I_{8h}}]$, value-universe $[Q]$ and the same r as here, we get

⁸We would never run out of memory bits as $\log 2r \ll \log \binom{U_\ell}{n(u)} \approx M(U_\ell, n(u)) + \log K(U_\ell, n(u))$.

a minimap which maintains the set of keys hashed to u , with query time $O(1)$, insertion/deletion time $O(\log n(u)) = O(\log \log N)$ (here $n(u) \leq \bar{n}_{sh} \leq 2B$), spill universe $K \leq 2r$, and redundancy at most $\frac{4}{r} \leq \frac{n_{\ell}-1}{r}$ bits.

Induction step. Assume the induction hypothesis holds for level $\ell + 1$ and we are going to prove it for level ℓ . For any level- ℓ node u , denote its B children by v_1, v_2, \dots, v_B ; denote by $n(v_i)$ the number of keys hashed to v_i . By the induction hypothesis, these $n(v_i)$ keys can be represented by $M(U_{\ell+1}, n(v_i))$ memory bits and a spillover $k_i \in [K(U_{\ell+1}, n(v_i))]$, with these $M(U_{\ell+1}, n(v_i))$ memory bits stored in a sub-VM from v_i . Below, we will introduce our encoding procedure step by step.

Step 1: Random swap. Guided by the intuition introduced in Section 4.2, the first step is to transform the sub-VM from each v_i into a WVM. To apply Lemma 4.3, we only need to check that these sub-VMs are word-dominant. Recalling that $w + 5 \leq \log \frac{U_{sh}}{n_{sh}} < w + 6$, by Claim 4.1, we have

$$n(v_i)w + 4n(v_i) \leq \log \binom{U_{\ell+1}}{n(v_i)} \leq n(v_i)w + 9n(v_i).$$

Hence by the induction hypothesis, we have

$$\begin{aligned} M(U_{\ell+1}, n(v_i)) &\leq M(U_{\ell+1}, n(v_i)) + \log K(U_{\ell+1}, n(v_i)) \\ &\leq \log \binom{U_{\ell+1}}{n(v_i)} + \frac{n_{\ell+1} - 1}{r} \leq n(v_i)w + 10n(v_i) \end{aligned}$$

(the last inequality holds because $n_{\ell+1} - 1 < n_1 = r < n(v_i)r$). On the other hand, by induction hypothesis, we have $M(U_{\ell+1}, n(v_i)) + \log K(U_{\ell+1}, n(v_i)) \geq \log \binom{U_{\ell+1}}{n(v_i)} - 1$. Hence,

$$\begin{aligned} M(U_{\ell+1}, n(v_i)) &\geq \log \binom{U_{\ell+1}}{n(v_i)} - \log K(U_{\ell+1}, n(v_i)) - 1 \\ &\geq n(v_i)w + 4n(v_i) - \log(2r) - 1 \geq n(v_i)w + 3n(v_i) \end{aligned}$$

(the last inequality holds because $\log 2r + 1 = \Theta(h \log \log N) \ll B/2 \leq \underline{n}_{sh} \leq n(v_i)$, as $h \leq \log^{0.3} N$ and $B \in [\sqrt{\log N}, 2\sqrt{\log N}]$). Therefore, the sub-VM for each v_i is word-dominant (see Definition 4.2), and by Lemma 4.3, it can be stored into a WVM with little performance loss.

Step 2: Adapters. The WVM from each v_i , according to its definition, is composed of a tape of $L_{\text{word}}^{[i]}$ words (denoted by $m_{\text{word}}^{[i]}$), and a tape of $L_{\text{bit}}^{[i]}$ extra bits (denoted by $m_{\text{bit}}^{[i]}$). We use two adapters to aggregate all of them together.

The first adapter aggregates all the extra bits $\{m_{\text{bit}}^{[i]} : 1 \leq i \leq B\}$. Although in Section 3.3 we only considered how to aggregate sequences of complete words, by treating every bit as “a word with word-length 1”, we can easily construct a *bit-wise* adapter that aggregates sequences of bits. The output of this adapter is a sequence of $\sum_{i=1}^B L_{\text{bit}}^{[i]}$ bits, which we store into a VM called the *bit VM*. We denote the bit VM’s complete-word part and tail part as $m_{\text{word}}^{[0]}$ and $m_{\text{bit}}^{[0]}$, respectively.

The second adapter aggregates all the complete words $\{m_{\text{word}}^{[i]} : 0 \leq i \leq B\}$, including the complete words produced by the first adapter. This results in a VM with $\sum_{i=1}^B L_{\text{word}}^{[i]} + \lfloor (\sum_{i=1}^B L_{\text{bit}}^{[i]})/w \rfloor$ words, which we call the *word VM*.

Finally, we directly concatenate the word VM with the tail part of the bit VM, getting a VM with totally $M_{\text{cat}} := \sum_{i=1}^B (wL_{\text{word}}^{[i]} + L_{\text{bit}}^{[i]}) = \sum_{i=1}^B M(U_{\ell+1}, n(v_i))$ bits, which we call the *concatenated memory*, namely, m_{cat} .

Step 3: Cut the memory. We cut the concatenated memory m_{cat} into two parts m_{fix} and m_{rem} , such that the first part has M_{fix} bits which only depends on $n(u)$ but *not* $n(v_i)$ for each i ; the second part has at most $O(w)$ bits. Formally, we define

$$M_{\text{max}} := \max_{n(v_1)+n(v_2)+\dots+n(v_B)=n(u)} \sum_{i=1}^B M(U_{\ell+1}, n(v_i)), \quad M_{\text{fix}} := M_{\text{max}} - w.$$

(There is always $M_{\text{fix}} \geq 0$ because $M_{\text{max}} > M(U_{\ell+1}, n(v_i)) = \Theta(\log \binom{U_{\ell+1}}{n(v_i)}) - \Theta(\log r) \geq \Theta(Bw) \gg w$.) We divide m_{cat} into the leftmost M_{fix} bits and the remaining $M_{\text{rem}} := M_{\text{cat}} - M_{\text{fix}}$ bits. It is possible that $M_{\text{cat}} < M_{\text{fix}}$ for the current number of keys $n(v_i)$, in which case m_{fix} is formed by padding zeros to the end of m_{cat} until it has M_{fix} bits; m_{rem} is left empty. In all cases, the second part contains at most w bits.

After cutting the memory into two parts, m_{fix} directly appears as the leftmost bits in our final encoding, while m_{rem} is further compressed with other information in the next step.

Step 4: Compress the labels and spillovers. Next, we compress the children's numbers of keys $\mathbf{n} := (n(v_1), n(v_2), \dots, n(v_B))$, their spillovers (k_1, k_2, \dots, k_B) , and the remaining part m_{rem} from the last step together, using the following lemma from [Pät08].

Lemma 4.4 ([Pät08]). *Assume we have to represent a variable $x \in \mathcal{X}$, and a pair $(y_M, y_K) \in \{0, 1\}^{M(x)} \times [K(x)]$. Let $p(x)$ be a probability density function on \mathcal{X} , and $K(\cdot)$, $M(\cdot)$ be non-negative functions on \mathcal{X} satisfying:*

$$\forall x \in \mathcal{X} : \quad \log \frac{1}{p(x)} + M(x) + \log K(x) \leq H. \quad (4)$$

We further assume the word size $w = \Omega(\log |\mathcal{X}| + \log r + \log \max K(x))$, then we can design a spillover representation for x , y_M , and y_K , denoted by $(m^, k^*) \in \{0, 1\}^{M^*} \times [K^*]$, with the following parameters:*

- *The spill universe is K^* with $K^* \leq 2r$; the memory size is M^* bits.*
- *The redundancy is at most $4/r$ bits, i.e., $M^* + \log K^* \leq H + 4/r$.*
- *Given a precomputed table of $O(|\mathcal{X}| \cdot r \cdot \max K(x))$ words that only depends on the input functions K, M , and p , and assuming $H \leq O(w)$, both decoding (x, y_M, y_K) from (m^*, k^*) and encoding (x, y_M, y_K) to (m^*, k^*) takes $O(1)$ time on a word RAM. The table can be precomputed in linear time.*

Let $k_{\text{cat}} \in [\prod_{i=1}^B K(U_{\ell+1}, n(v_i))]$ be the combination of the children's spillovers (k_1, k_2, \dots, k_B) . We are going to apply the above lemma on $\mathcal{X} = \{\mathbf{n} : \sum_{i=1}^B n(v_i) = n(u)\}$ and $(y_M, y_K) = (m_{\text{rem}}, k_{\text{cat}})$.

To construct the probability distribution, we first define

$$p(\mathbf{n}) := \frac{\prod_{i=1}^B \binom{U_{\ell+1}}{n(v_i)}}{\binom{U_{\ell}}{n(u)}},$$

that is, the induced marginal distribution on \mathbf{n} when the set of keys hashed to u follows the uniform distribution over all $n(u)$ -element subsets of the universe. Same as [Pät08], we slightly perturb the distribution for stronger properties.

Claim 4.5 ([Pät08]). *For any probability distribution $p(\cdot)$ over set \mathcal{X} and any parameter $r > 0$, we can perturb $p(\cdot)$ to another probability distribution $p'(\cdot)$, such that for any $x \in \mathcal{X}$,*

- $p'(x) \geq \frac{1}{2r|\mathcal{X}|}$.
- $\log \frac{1}{p'(x)} \leq \log \frac{1}{p(x)} + \frac{2}{r}$.

We then apply Lemma 4.4 with the perturbed distribution p' and

$$H := \log \binom{U_\ell}{n(u)} - M_{\text{fix}} + \frac{n_\ell - 5}{r}.$$

We check the condition (4) by discussing two cases:

- Suppose $M_{\text{cat}} \geq M_{\text{fix}}$, i.e., we did not pad zeros to M_{cat} . In this case, $|m_{\text{rem}}| = M_{\text{cat}} - M_{\text{fix}}$. The induction hypothesis implies that

$$M(U_{\ell+1}, n(v_i)) + \log K(U_{\ell+1}, n(v_i)) \leq \log \binom{U_{\ell+1}}{n(v_i)} + \frac{n_{\ell+1} - 1}{r}.$$

Therefore, for any $\mathbf{n} \in \mathcal{X}$, the LHS of (4) is

$$\begin{aligned} & \log \frac{1}{p'(\mathbf{n})} + (M_{\text{cat}}(\mathbf{n}) - M_{\text{fix}}) + \log \left(\prod_{i=1}^B K(U_{\ell+1}, n(v_i)) \right) \\ & \leq \left(\log \frac{1}{p(\mathbf{n})} + \frac{2}{r} \right) + \sum_{i=1}^B M(U_{\ell+1}, n(v_i)) - M_{\text{fix}} + \sum_{i=1}^B \log K(U_{\ell+1}, n(v_i)) \\ & \leq \log \frac{\binom{U_\ell}{n(u)}}{\prod_{i=1}^B \binom{U_{\ell+1}}{n(v_i)}} + \sum_{i=1}^B (M(U_{\ell+1}, n(v_i)) + \log K(U_{\ell+1}, n(v_i))) + \frac{2}{r} - M_{\text{fix}} \\ & \leq \log \binom{U_\ell}{n(u)} + B \cdot \frac{n_{\ell+1} - 1}{r} + \frac{2}{r} - M_{\text{fix}} \\ & = \log \binom{U_\ell}{n(u)} + \frac{n_\ell - B + 2}{r} - M_{\text{fix}} \\ & < H. \end{aligned}$$

- Suppose $M_{\text{cat}} < M_{\text{fix}}$. In this case, m_{rem} is empty, so the LHS of (4) equals

$$\begin{aligned} & \log \frac{1}{p'(\mathbf{n})} + \log \left(\prod_{i=1}^B K(U_{\ell+1}, n(v_i)) \right) \\ & \leq \log(2r|\mathcal{X}|) + \log(2r)^B = \Theta(B \log r) \ll w, \end{aligned}$$

where the first inequality is due to Claim 4.5 and the induction hypothesis $K(U_{\ell+1}, n(v_i)) \leq 2r$; the second inequality holds as $|\mathcal{X}| \leq n(u)^B \leq (2n_1)^B = (2r)^B$; the last inequality holds as $B \log r = \Theta(\sqrt{\log N} \cdot h \log \log N) \ll w$, since $B \leq 2\sqrt{\log N}$ and $h \leq \log^{0.3} N$. On the other side, the RHS of (4) is

$$\begin{aligned} H & \geq \max_{\mathbf{n} \in \mathcal{X}} \log \left(\prod_{i=1}^B \binom{U_{\ell+1}}{n(v_i)} \right) - M_{\text{fix}} + \frac{n_\ell - 5}{r} \\ & \geq \max_{\mathbf{n} \in \mathcal{X}} \left(M_{\text{cat}}(\mathbf{n}) - B \cdot \frac{n_{\ell+1} - 1}{r} \right) - M_{\text{fix}} + \frac{n_\ell - 5}{r} \\ & = M_{\text{max}} - M_{\text{fix}} + \frac{B - 5}{r} > w \geq \text{LHS}. \end{aligned}$$

Here, the second inequality holds due to the redundancy constraint in the induction hypothesis; the third inequality is because $M_{\text{fix}} = M_{\text{max}} - w$. Therefore (4) also holds in this case.

Applying Lemma 4.4 gives us a spillover representation $(m^*, k^*) \in \{0, 1\}^{M^*} \times [K^*]$ of $(\mathbf{n}, k_{\text{cat}}, m_{\text{rem}})$ conditioning on $n(u)$, where $K^* \leq 2r$ and

$$M^* + \log K^* \leq H + \frac{4}{r} = \log \binom{U_\ell}{n(u)} + \frac{n_\ell - 1}{r} - M_{\text{fix}}.$$

With the help of proper lookup tables, both the encoding and decoding procedures can be completed within constant time.

Step 5: Concatenate. The last step involves concatenating m_{fix} with m^* , the outcome memory bits obtained from the previous compression step, to form a bit string of length $M_{\text{fix}} + M^* =: M(U_\ell, n(u))$. This memory string, combined with the spillover $k^* \in [K^*] = [K(U_\ell, n(u))]$, form the encoding for the set of keys hashed to level- ℓ node u . The induction hypothesis holds for ℓ , since $K^* \leq 2r$ and

$$M(U_\ell, n(u)) + \log K(U_\ell, n(u)) = M_{\text{fix}} + M^* + \log K^* \leq \log \binom{U_\ell}{n(u)} + \frac{n_\ell - 1}{r}.$$

Finally, the $M(U_\ell, n(u))$ memory bits are again divided into $\lfloor M(U_\ell, n(u))/w \rfloor$ words and a tail, stored within a VM. \square

Remark 4.6. *Observe that the primary part, m_{fix} , which comes from the adapter, is “aligned” with the resulting VM. This means that a complete word in a child’s sub-VM is still stored as a complete word in the new VM. This alignment benefits our query and update algorithms because each word-access from a child will translate into only one word-access of u ’s virtual memory.*

Now, consider Statement 1 for level-1 node u . We directly encode the final spillover $k \in [K(U_1, n(u))]$ into memory, incurring a 1-bit redundancy due to rounding. Then, we can store the set of keys hashed to a level-1 bucket within a space of

$$M(U_1, n(u)) + \log K(U_1, n(u)) + 1 \leq \log \binom{U_1}{n(u)} + \frac{n_1 - 1}{r} + 1 < \log \binom{U_1}{n(u)} + 2.$$

4.4 Construction of the whole Dictionary

The whole dictionary consists of the following parts:

- A fixed space of $\log \binom{U_1}{\bar{n}_1} + 2$ bits for each level-1 node. As we have seen, the set of keys hashed to a level-1 bucket u can be stored in $\log \binom{U_1}{n(u)} + 2 \leq \log \binom{U_1}{\bar{n}_1} + 2$ bits, assuming free access to the number of keys $n(u)$.
- A space of $\log \bar{n}_1$ bits for each level-1 node to store the number of keys hashed to it.
- Several lookup tables used in the induction processes, which will be analyzed later.
- Random seed used in the random swap lemma (Lemma 4.3), which is shared between nodes on the same level, while the seeds for different levels are chosen independently. They only occupy a very limited space of $O(h \cdot \log^2 N)$ bits.

- The space for storing hash functions, which is $o(R)$ bits. The analysis is deferred to Appendix C.

As mentioned in Section 1, every part of the memory only depends on the current set of keys and some random bits, which means that our dictionary is strongly history-independent. Below, we calculate the space usage of each part.

The first part occupies the most of the space. Recalling that $R := N/\log^h N = \Omega(N/n_1^{1/4}) \gg (N \log N)/n_1^{1/3}$, we have

$$\begin{aligned} \frac{N}{n_1} \log \binom{U_1}{\bar{n}_1} &\leq \log \binom{U}{N \cdot \bar{n}_1/n_1} \leq \log \binom{U}{N \cdot (1 + 2 \cdot n_1^{-1/3})} \leq (1 + 2 \cdot n_1^{-1/3}) \log \binom{U}{N} \\ &= \log \binom{U}{N} + \Theta((N \log U)/n_1^{1/3}) \leq \log \binom{U}{N} + O(R), \end{aligned}$$

where the third inequality comes from the log concavity of $\binom{n}{k}$ as a function of k . Therefore, the total space usage of the first part does not exceed $\frac{N}{n_1} \log \binom{U_1}{\bar{n}_1} + \frac{2N}{n_1} \leq \log \binom{U}{N} + O(R)$ bits.

The space usage of the second and the fourth parts do not exceed $\frac{N}{n_1} \cdot \log N = o(R)$ bits and $O(\log^3 N) \ll o(R)$ bits, respectively. The hash functions (the last part) only occupy $o(R)$ bits as well.

It remains to check the sizes of the lookup tables, which are listed below. All the lookup tables occupy at most $O(\sqrt{N})$ words which is far less than $O(R)$ bits.

- lookup table for minimaps. We used minimaps to maintain at most $n' := \bar{n}_{8h} = \Theta(\sqrt{\log N})$ key-value pairs from a key-universe $U' := 2^{l_{sh}} = \text{poly log } N$. According to Lemma 3.7, the lookup table size is $U' n' \binom{U'}{n'} \cdot (2r n')^{n'} = (\text{poly log } N)^{\Theta(\sqrt{\log N})} \ll O(\sqrt{N})$ words.
- lookup table for adapters. According to Lemma 3.6, the lookup table for adapters occupies $O(B L_{\max}^{B+1})$ words, where L_{\max} is the maximum number of words in the aggregated super-VM. Plugging in $L_{\max} = O(n_1) = (\log N)^{O(\log^{0.3} N)}$ and $B = \Theta(\sqrt{\log N})$, we know the lookup table size is less than $O(\sqrt{N})$ words.
- lookup table for encoding/decoding \mathbf{n} and k_{cat} . At the beginning of the step 4, we need to encode the number of keys in each child $\mathbf{n} = (n(v_1), n(v_2), \dots, n(v_B))$ using its index in the set $\{\mathbf{n} : \sum_{i=1}^B n(v_i) = n(u)\}$. For a fixed ℓ and $n(u)$, it needs a lookup table of $O(n(u)^B)$ words. Hence, the total size is at most $O(h \bar{n}_1^{B+1})$ words.

Moreover, we need to compute vectors $(L_{\text{word}}^{[i]})_{i \in [B]}$ and $(L_{\text{bit}}^{[i]})_{i \in [B]}$, the sizes of the tapes of WVMs from all children, which are required for adapter operations and are fully determined by \mathbf{n} . This is done with a lookup table of the same size.

We also need to encode all the spillovers k_1, k_2, \dots, k_B into $k_{\text{cat}} \in [\prod_{i=1}^B K(U_{\ell+1}, n(v_i))]$. For a fixed ℓ and \mathbf{n} , it needs a lookup table of $O((2r)^B)$ words, so the total size is at most $O(h \bar{n}_1^B (2r)^B) = (\log N)^{O(\log^{0.3} N) \cdot \Theta(\sqrt{\log N})} \ll O(\sqrt{N})$ words.

- lookup table for Lemma 4.4. For each level ℓ and each possible number of keys $n(u)$ in a node u , we need a table of $O(|\mathcal{X}| \cdot r \cdot \max K(x))$ words, where \mathcal{X} is the set formed by the array of number of keys in children of u , i.e., $\mathcal{X} := \{(n(v_1), n(v_2), \dots, n(v_B)) : \sum_{i=1}^B n(v_i) = n(u)\}$, whose size is at most $O(\bar{n}_1^B)$; we also have $\max K(x) \leq (2r)^B$. Taking summation over all possible $n(u)$'s and all h levels, the total size is $O(h \cdot \bar{n}_1 \cdot \bar{n}_1^B \cdot r \cdot (2r)^B) = (\log N)^{O(\log^{0.3} N) \cdot \Theta(\sqrt{\log N})} \ll O(\sqrt{N}) \ll O(\sqrt{N})$ words.

- Tables for quickly computing each $M(U_\ell, n(u))$, $K(U_\ell, n(v_i))$, and $M_{\text{fix}} = M_{\text{fix}}(U_\ell, n(u))$. They occupy $O(h\bar{n}_1) \ll O(\sqrt{N})$ words.

All lookup tables mentioned above can be pre-computed efficiently and will not become the time bottleneck. Now, summing up the space usage of all parts, the data structure occupies a space of $\log \binom{U}{N} + O(R)$ bits.

4.5 Query, Insertion, and Deletion

All operations on the data structure begin by simulating the hashing process on the input key x . Specifically, we first apply the first hash function h on x , obtaining a triple $(u, x_{\text{id}}, x_{\text{quot}})$, which means x is hashed to a root node u with the ID-quotient pair $(x_{\text{id}}, x_{\text{quot}})$. Then, by applying the second hash function, x is further hashed to a (level- $8h$) leaf node v , with ID-quotient pair $(y_{\text{id}}, x_{\text{quot}})$ in v . These nodes indicate a path from the root to the leaf, which we will walk along in all types of operations.

Query algorithm and nested adapters. Recall that every node u maintains a sub-data structure storing keys hashed into u 's subtree, which is stored in a VM and a spillover $k = k^*$. The VM consists of two parts: m_{fix} and m^* , where m_{fix} comes from adapters that concatenate sub-VMs from the children, and m^* includes the information (spillovers and numbers of keys) of the children. The latter part m^* has at most $O(w)$ bits and is contained in the rightmost $O(1)$ words in u 's VM. Throughout this section, we call m^* the *metadata region* of u 's VM.

Before we visit the node u , we need the knowledge of its spillover k and number of keys $n(u)$. Then, we immediately read the metadata m^* on u 's VM, which takes $O(1)$ word-accesses. According to Lemma 4.4, we can decode (k^*, m^*) in constant time to recover the number of keys in each child, $n(v_1), n(v_2), \dots, n(v_B)$, their spillovers k_1, k_2, \dots, k_B , and the rightmost bits of the concatenated memory, m_{rem} . Consequently, we get vectors $(L_{\text{word}}^{[i]})_{i \in [B]}$ and $(L_{\text{bit}}^{[i]})_{i \in [B]}$, the sizes of both tapes of WVMs from children, which are fully determined by \mathbf{n} . Next, the query algorithm recurses into a child of u along the path, and repeat the above process until it reaches the leaf. Finally, it makes a query to the minimap on the leaf to get the answer.

The above description has assumed direct access to the VM of any given node. However, only the VM of the root node is directly stored into the “physical memory”, while the VM of other nodes are connected to it via several nested adapters (and random-swap structures (Lemma 4.3) which stores VMs into WVMs). To access a word in node u 's VM, we first translate it into access requests to the parent's VM, then to the grandparent, and continue until the root is reached. Formally, we use the following subroutine (Algorithm 2) to access a word in the VM of an arbitrary node. We denote by (v, i) the i -th word in the VM of node v . As reading the tail of the VM has the same interface as reading a word, we do not distinguish them and regard the tail as the last word.

When we read a word (v, i) , Line 12 does not produce extra word-accesses, because we have already gained the knowledge of m_{rem} when we visit u (we always need to visit the parent u before we can visit v). On the other hand, when we write to some word (v, i) (which is not required for queries, but we will need it for updates), it might seem that Line 12 would need to write to m_{rem} of u ; what we actually do here is to delay the writing request to m_{rem} until the end of the entire update operation, at which point we will update all m_{rem} bottom-up along the path we visited.

Due to the random swap lemma (Lemma 4.3), accessing (v, i) may cause more than 1 word-accesses to the VM on the parent of u . Specifically, assuming v is in level ℓ , the expected number of word-accesses to its parent u is at most

$$1 + O(\min(1, w^2/M(U_\ell, n(v)))) = 1 + O(\min(1, w/n(v))) \leq 1 + c \min\left(1, B^{1-8h+\ell}\right)$$

Algorithm 2: Accessing Virtual Memory Words

```

1 Function Access( $v, i$ ): ▷ Access the  $i$ -th word in  $v$ 's VM
2   if  $v$  is the root then
3     Directly access the  $i$ -th word
4     return
5    $u \leftarrow v$ 's parent
6   Translate the access request  $(v, i)$  into access requests to  $v$ 's WVM according to the
   random swap lemma (Lemma 4.3)
7   Translate each WVM access request into an access request to  $u$ 's concatenated memory
    $m_{\text{cat}} = m_{\text{fix}} \parallel m_{\text{rem}}$  via the adapter
8   foreach access request to the  $j$ -th word in  $u$ 's concatenated memory do
9     if any part of word  $j$  belongs to  $m_{\text{fix}}$  then
10      Access( $u, j$ )
11     if any part of word  $j$  belongs to  $m_{\text{rem}}$  then
12      Access the corresponding bits in  $m_{\text{rem}}$  which has been decoded via Lemma 4.4

```

for a fixed constant $c > 0$ (note that $n(v) = \Theta(n_\ell) = \Omega(B^{8h-\ell+1})$ and $w = \Theta(B^2)$). As the random seeds for random swapping are independent for different levels, and

$$\prod_{\ell=1}^{8h} \left(1 + c \min\left(1, B^{1-8h+\ell}\right)\right) \leq (1+c)^2 \cdot \prod_{i=1}^{\infty} \left(1 + \frac{c}{B^i}\right) = O(1),$$

we can see that every word-access in node v will cause at most $O(1)$ word-accesses to the root in expectation, and it takes $O(\ell) = O(h)$ time to translate the word-access requests from v to the root. That means we need $O(h)$ expected time to access a word in the VM of any given node. Moreover, the query algorithm initiates $O(1)$ word-accesses at every node it visits (mainly for reading m^*), so the total running time for the query algorithm is $O(h^2)$.

Insertion and deletion algorithms. For insertions and deletions, we first follow the same procedure as the query algorithm, walking from the root down to the leaf that the key was hashed to, recovering the numbers of keys $n(u)$ and spillovers for all the nodes along the path. Next, we insert/delete the key in the minimap on the leaf, which takes $O(\log \bar{n}_{8h}) = O(\log \log N)$ word-accesses to the VM of the leaf.

However, when we insert/delete a key in the leaf, every node u on the path changes its number of keys $n(u)$, and therefore changes the space usage of u 's VM, i.e., $M(U_\ell, n(u))$. Additional time is spent to adjust the VM sizes. Due to the symmetry, we only analyze the insertions below. We will show later that the difference of $M(U_\ell, n(u))$ before and after the insertion is $w \pm O(1)$, which can be expressed as $O(1)$ allocation/release requests to u 's VM, each allocating/releasing exactly w bits or a single bit. According to Lemma 4.3, we can further translate these requests to $O(1)$ allocation/release requests (followed by $O(1)$ word or bit accesses) to the WVM, which are further translated to $O(1)$ allocation/release requests to the two adapters connecting u 's WVM with its parent. They take $O(\log \max(n_\ell, 2w)) = O((8h - \ell) \cdot \log \log N)$ word-accesses to the VM on the parent of u , each can be completed within $O(h)$ time as shown above. Taking summation over all h nodes on the path, the time complexity for an insertion is $O(h^3 \log \log N)$.

After adjusting the VM sizes, for all nodes u on the path from bottom to top, we redo all encoding steps introduced in Section 4.3, and update all changed words. They may include:

- Step 4 involves compressing $O(1)$ words of information into a spillover representation. Its resulting memory bits are directly stored in the rightmost words of u 's VM, namely m^* . We initiate $O(1)$ word-accesses to rewrite all of them.
- Changing the number of keys $n(u)$ of node u may cause M_{fix} to change since it depends on $n(u)$. M_{fix} will change by at most $O(w)$ (bits) since it only differs from $M(U_\ell, n(u))$ by $O(w)$, while the latter only changes by $O(w)$ during an update. The change of M_{fix} requires us to update the rightmost $O(1)$ words in m_{fix} , incurring no more than $O(1)$ word-accesses at each level.

The time spent on these word-updates is less than the time spent for allocations, which is $O(h^3 \log \log N)$ as shown above. Therefore, the (expected) time complexity for an insertion or a deletion is $O(h^3 \log \log N)$.

It remains to show that the difference of VM size before and after the insertion is equal to $w \pm O(1)$ bits, i.e.,

$$w - O(1) \leq M(U_\ell, n(u) + 1) - M(U_\ell, n(u)) \leq w + O(1). \quad (5)$$

Actually, by the induction hypothesis, we have

$$\begin{aligned} \log \binom{U_\ell}{n(u)} - 1 &< M(U_\ell, n(u)) + \log K(U_\ell, n(u)) \leq \log \binom{U_\ell}{n(u)} + 1, \\ \log \binom{U_\ell}{n(u) + 1} - 1 &< M(U_\ell, n(u) + 1) + \log K(U_\ell, n(u) + 1) \leq \log \binom{U_\ell}{n(u) + 1} + 1. \end{aligned}$$

As $K(U_\ell, n(u)), K(U_\ell, n(u) + 1) \in (r, 2r]$, we can see that $|\log K(U_\ell, n(u)) - \log K(U_\ell, n(u) + 1)| < 1$. Moreover, by Claim 4.1 and $w + 5 \leq \log \frac{U_{sh}}{n_{sh}} < w + 6$, we have

$$w + 2 \leq \log \binom{U_\ell}{n(u) + 1} - \log \binom{U_\ell}{n(u)} \leq w + 7.$$

Combining these inequalities, we get

$$\begin{aligned} &M(U_\ell, n(u) + 1) - M(U_\ell, n(u)) \\ &\geq \left(\log \binom{U_\ell}{n(u) + 1} - 1 \right) - \left(\log \binom{U_\ell}{n(u)} + 1 \right) - |\log K(U_\ell, n(u)) - \log K(U_\ell, n(u) + 1)| \\ &\geq \log \binom{U_\ell}{n(u) + 1} - \log \binom{U_\ell}{n(u)} - 3 \geq w - 1, \end{aligned}$$

and similarly, $M(U_\ell, n(u) + 1) - M(U_\ell, n(u)) \leq w + 10$. Thus, (5) holds.

5 Dictionary via Multi-Level Hashing

In this section, our focus shifts to dynamic dictionaries that exhibit larger redundancy and smaller update time compared to those discussed in the previous section. Up to Section 5.5, we will design dynamic dictionaries that can store N distinct keys in the universe $[U]$ with $O(R)$ bits redundancy,

$O(\log^* N)$ query time, and $O(\log^* N + \log \frac{N}{R})$ update time, for any $\frac{N}{\log^{0.1} N} \leq R \leq \frac{N}{2^{\log^* N}}$; the query time is improved to $O(1)$ in the worst case in Section 5.6. The upper limit of R already corresponds to the optimal update time $O(\log^* N)$ within this range, so considering larger values of R is unnecessary. It is important to note that when N is sufficiently large, the value of $r = N/R$ exceeds any constant value.

As before, we make the assumption that $U = N^{1+\alpha}$ for some constant $\alpha > 0$, and $w = \Omega(\log N)$. We also assume that the dictionary always contains either N keys or $N - 1$ keys, without loss of generality.

Algorithm Overview. Let $n_1 = \Theta(\log^9 N)$ be a parameter. We hash keys to $B_0 = N/n_1$ buckets which we call the *level-1 buckets*. The expected number of keys hashed into a specific level-1 bucket u equals n_1 .

For all keys that are mapped to bucket u , we further hash them to smaller *level-2 buckets*; the keys in the same level-2 bucket are further hashed to smaller level-3 buckets, and so on. Such procedure is described by a forest: the roots are level-1 buckets and every internal node u at level ℓ has B_ℓ children at level $\ell + 1$. All keys that were hashed into u will further be hashed into a random child of u . We use n_ℓ to denote the expected number of keys hashed into a level- ℓ node: it is clear that $n_\ell = n_{\ell-1}/B_{\ell-1}$. We set $n_\ell = \Theta((\log^{(\ell)} N)^9)$ for internal levels. (Note that the step of hashing all N keys into level-1 buckets is not a part of the tree; we have different technical details for this stage.)

During the hashing process, we carefully guarantee that the tree is not “too unbalanced”. We define $\Delta_\ell = n_\ell^{2/3}$ and hope the number of keys n in a level- ℓ node is always in $[n_\ell - \Delta_\ell, n_\ell + \Delta_\ell]$. If after we hash all keys in u to its children, some child violates such rule, then we immediately pick another hash function for u and rebuild the subtree rooted at u . If some root node already violates this rule, we simply rebuild the whole dictionary. The cost of rehashing is low on average. Let $r = N/R$ and $\log^{(k)} N \leq r < \log^{(k-1)} N$. The tree terminates at a level k , consisting of leaf nodes, each with expected number of keys $n_k = \Theta(r^9)$. The number of keys is small enough so that a minimap is capable of storing keys in a leaf. The minimap has $O(1)$ query time, $O(\log r)$ insertion/deletion time, and almost no redundancy. Minimaps are the time bottleneck of our algorithm.

However, the number of keys hashed into a leaf is not fixed; it changes with time. This is the main challenge of the algorithm: if we use variable-size data structures for leaf nodes, then it is hard to concatenate them efficiently and succinctly.⁹ Our solution here is to carefully make each node fixed-size, and assign a fixed piece of memory for each node, as follows:

- For each leaf node, it contains at least $\underline{n}_k := \lceil n_k - \Delta_k \rceil$ keys. We arbitrarily pick \underline{n}_k keys that were hashed into this leaf and use a fixed-size minimap to store them. For the remaining keys that were not stored in the leaf, we send them back to the parent node.
- Before we process for an inner node u at level ℓ , each child v of u already stored $\underline{n}_{\ell+1}$ keys in its subtree, and all unstored keys are sent to u . Recall that the number of keys hashed into the subtree of u is at least \underline{n}_ℓ , hence the number of keys sent back to u is at least $\underline{n}_\ell - B_\ell \cdot \underline{n}_{\ell+1}$. We arbitrarily pick this number of keys, store them at node u with a fixed-size hash table, then send other keys to the parent of u .
- Such procedure ends when all unstored keys are sent back to the level-1 bucket, i.e., the root. All keys here are stored in a hash table with enough capacity $\bar{n}_1 := \lfloor n_1 + \Delta_1 \rfloor$. Some of its

⁹Using *adapters* here is also too slow.

capacity is wasted since the number of keys is always below the capacity.

Below, we add implementation details to complete the construction. We will first introduce the design when everything is ideal, then construct the hash functions that we use, and lastly handle rare bad events via rehashing.

Tree Parameters. Before we start, we briefly specify the tree parameters. For technical convenience, we let B_ℓ ($\ell \in [0, k - 1]$) be always a power of two. We further require $n_\ell := n_{\ell-1}/B_{\ell-1} \in [(\log^{(\ell)} U)^9, 2(\log^{(\ell)} U)^9]$ for $\ell \in [1, k - 1]$; require $n_k \in [r^9, 2r^9]$. These requirements uniquely determine the branching factor B_ℓ for all levels. Also note that $r \leq \log^{0.1} N$ is guaranteed, meaning that $n_2 \leq 2 \log^{0.9} N$.

n_ℓ is the expected number of keys hashed into a level- ℓ node u . We only consider the ideal case where the number of keys n is between $[\underline{n}_\ell, \bar{n}_\ell] = [\lceil n_\ell - \Delta_\ell \rceil, \lfloor n_\ell + \Delta_\ell \rfloor]$. If not, we say node u *overflows* ($n > \bar{n}_\ell$) or *underflows* ($n < \underline{n}_\ell$). Once overflow or underflow happens, we immediately rebuild some subtree so that it no longer occurs.

5.1 Hashing with ID-Quotient Separation

Similar to the previous section, during the hashing steps, we regard keys as ID-quotient pairs. Besides the merit mentioned before that the short ID enables us to use *minimap* for each leaf node, it also allows short pointers to the stored keys. To access a stored key, only the ID of that key needs to be remembered, which saves space.

Specifically, the keys are represented as ID-quotient pairs since they were hashed into tree roots. In a level- ℓ node u , each key is represented by an element in $[2^{I_\ell}] \times [Q_\ell]$, where $I_\ell := \lceil 4 \log n_\ell \rceil$ represents the length of ID and Q_ℓ represents the range of the quotient.

As the keys received at node u will be hashed into the child nodes, the ID length will become shorter, i.e., of length $I_{\ell+1}$; the rest bits will be moved to the quotient. Formally, we label the children of u with integers $v \in [B_\ell]$. Assume some key $x = (x_{\text{id}}, x_{\text{quot}})$ arrives at u . The procedure of hashing x into a child v can be described by a bijective hash function from x_{id} to $(v, y_{\text{id}}, x_{\text{rem}})$, where $v \in [B_\ell]$ is the child index that x will enter, $y_{\text{id}} \in [2^{I_{\ell+1}}]$ is the ID part of the next-level representation of x , and x_{rem} is the remaining bits that will be merged into the quotient. Then, we concatenate x_{rem} with x_{quot} , obtaining $y_{\text{quot}} := (x_{\text{rem}} \parallel x_{\text{quot}}) \in [Q_{\ell+1}]$ as the next-level quotient. Finally, the pair $(y_{\text{id}}, y_{\text{quot}})$ will be passed to the child v .

Similar to the previous section, the hash function $h : [2^{I_\ell}] \rightarrow [B_\ell] \times [2^{I_{\ell+1}}] \times [Q_{\ell+1}/Q_\ell]$ used above needs two properties. First, it should evenly distribute all keys to the children, preventing overflowing or underflowing of any child. Second, for each child $v \in [B_\ell]$, all keys assigned to that child should get different y_{id} . The formal construction of the hash function families are deferred to Section 5.4.

Above we introduced the hashing process at a tree node that distributes the keys to its child nodes. The process of hashing N initial keys to B_0 level-1 buckets (tree roots) is similar. We first round up U to a multiple of $B_0 \cdot 2^{I_1}$, then use a hash function $h : [U] \rightarrow [B_0] \times [2^{I_1}] \times [Q_1]$ to map each key $x \in U$ to $(u, y_{\text{id}}, y_{\text{quot}})$. Then $(y_{\text{id}}, y_{\text{quot}})$ is passed to the u -th level-1 bucket.

5.2 Storage Structure

Leaf Nodes. Recall that a leaf node u is responsible for storing exactly \underline{n}_k keys among all received keys. The selection of keys to store is arbitrary. Let the set of selected keys be S . The main structure inside a leaf node is a minimap storing S .

Each key $x \in S$ to store is already represented as $(x_{\text{id}}, x_{\text{quot}}) \in [2^{I_k}] \times [Q_k]$. To store S , we regard x_{id} as “keys” and x_{quot} as “values”, then use the minimap to store these key-value pairs. The set of ID $S_{\text{id}} = \{x_{\text{id}} : x \in S\}$ can be encoded within

$$\log \binom{2^{I_k}}{\underline{n}_k} \leq \log \binom{2^8 \cdot r^{36}}{2r^9} \leq 2r^9 \log(2^8 \cdot r^{36}) = O(\log^{0.9} N \cdot \log \log N) \leq O(w)$$

bits, where the last inequality holds because $w = \Omega(\log N)$. Thus the ID set S_{id} fits in constant words and satisfies the prerequisite of minimap. Applying Lemma 3.7 with key universe $[2^{I_k}]$, value universe $[Q_k]$, and redundancy parameter $r = 1$, we get a minimap storing the set S .

With minimap, one can query a key in $O(1)$ time, and insert/delete a key in $O(\log r)$ time. The redundancy for the minimap is $O(1)$ bits which comes from the following two parts: (1) Lemma 3.7 itself introduces $O(1)$ bits of redundancy; (2) as we want to store the minimap in a fixed-length consecutive space in the memory without using spillover representation, the rounding of spillover introduce 1 bit of redundancy. Other unstored keys will be returned to the parent node and finally stored at some ancestor.

Inner Nodes. Let u be a level- ℓ inner node. The number of received keys is guaranteed to be in $[\underline{n}_\ell, \bar{n}_\ell]$. After further hashing these keys into child nodes, each child node stores $\underline{n}_{\ell+1}$ keys in its subtree, and the remaining keys are sent back to u . Clearly, there are at least $m := \underline{n}_\ell - B_\ell \underline{n}_{\ell+1}$ returned keys. The main task of node u is to store m arbitrarily selected keys that were returned to u , so that the subtree of u stores exactly \underline{n}_ℓ keys in total.

To finish the task, we maintain a hash table of fixed capacity m , called *storage table*. Recall that each key is represented as $(x_{\text{id}}, x_{\text{quot}})$ at this level. The storage table takes x_{id} as its key and x_{quot} as its value. We choose the hash table implementation from [BFK⁺22] with $O(m \log \log 2^{I_\ell}) = O(m \log I_\ell)$ redundancy and $O(1)$ query/update time. Note that the storage table is always full and takes a fixed space, so we put it at a static location in the memory.

There will always be unstored keys; we again return them to the parent. The only exception is the root nodes. At root nodes, we slightly change the configuration, letting $m = \bar{n}_1 - B_1 \underline{n}_2$. Then, as long as the root node does not overflow, all keys can be successfully stored in the storage table. In this case, a portion of capacity is wasted, but the produced redundancy is still acceptable.

Waiting Lists. The above construction is already enough to encode the set of keys. However, to support quick deletion, we need one more auxiliary structure. Let u be a level- ℓ node where $\ell \geq 2$. After storing a fixed number of keys in the storage table, other unstored keys are returned to the parent of u . Imagine that we need to remove some key from u 's storage table. To make the storage table full again, we need to pull down some key x from u 's ancestors and then insert x to u 's storage table. *Waiting list* is designed for quickly finding such x .

For every key $x = (x_{\text{id}}, x_{\text{quot}})$ that is not stored in the subtree of u , i.e., has been sent back to u 's parent, we store its current-level ID x_{id} and last-level ID $x_{\text{id}}^{[\ell-1]}$ in the waiting list. Its tail part is not stored here. The waiting list has a fixed capacity $\bar{n}_\ell - \underline{n}_\ell = \Theta(\Delta_\ell)$. As long as node u does not overflow, this capacity is enough. (The whole waiting list is auxiliary redundant information, so underutilizing its capacity is not a problem.) When we need to find a key x stored at ancestors of u , we just pick an arbitrary $x_{\text{id}}^{[\ell-1]}$ from the waiting list of u , then query x at u 's parent. This is doable because $x_{\text{id}}^{[\ell-1]}$ serves as the unique identifier of x at u 's parent.

To implement the waiting list, we straightforwardly list all $(x_{\text{id}}, x_{\text{id}}^{[\ell-1]})$ as bit strings in the memory, with a 1-bit separation between adjacent two terms which marks the end point of the list.

The waiting list can hold at most $\bar{n}_\ell - \underline{n}_\ell \leq 2\Delta_\ell$ keys, so the space usage is at most

$$2\Delta_\ell(1 + I_\ell + I_{\ell-1}) \leq 6\Delta_2 \cdot I_1 = 6n_2^{2/3} \cdot \lceil 4 \log n_1 \rceil < O(\log N) = O(w),$$

where the second inequality holds because $n_2 \leq \log^{0.9} N$. Hence, the whole waiting list can fit in $O(1)$ words, meaning that all operations on the waiting list can be done in $O(1)$ time via bit operations.

All leaf nodes and inner nodes except the roots maintain their own waiting lists. Root nodes do not need waiting lists because their storage tables are large enough to contain all keys.

Redundancy Analysis. As introduced above, the dictionary is composed of a range of fixed-size structures:

- a minimap for each leaf node;
- a storage table for each non-leaf node;
- a waiting list for each non-root node.

(The space usage of hash functions are not counted here.) Next, we carefully list the redundant information in the dictionary:

- Each minimap produces $O(1)$ redundancy. Multiplied by the number of leaf nodes N/n_k , there is a redundancy of $O(N/n_k) = O(N/r^9)$.
- A waiting list in level ℓ occupies $O(\Delta_\ell \cdot I_{\ell-1}) = O(n_\ell^{2/3} \log n_{\ell-1}) \leq O(n_\ell^{2/3} \cdot n_\ell^{1/9}) = O(n_\ell^{7/9})$ bits of memory. These bits are all redundant. Multiplied by the number of level- ℓ nodes N/n_ℓ , it is a redundancy of $O(N/n_\ell^{2/9})$ bits. It reaches maximum when $\ell = k$, i.e., at the leaf nodes, where the total redundancy is $O(N/r^2)$.
- According to the hash table implementation, a storage table in level ℓ produces $O(m \log \log 2^{I_\ell}) = O(m \log \log n_\ell)$ redundancy, where $m = \underline{n}_\ell - B_\ell \underline{n}_{\ell+1}$ for non-root level ℓ and $m = \bar{n}_1 - B_1 \underline{n}_2$ for the root level. As each child can only contribute $2\Delta_{\ell+1}$ unstored keys, $m = O(B_\ell \Delta_{\ell+1})$. The total redundancy over all level- ℓ nodes is

$$\frac{N}{n_\ell} \cdot O(B_\ell \Delta_{\ell+1} \log \log n_\ell) \leq O\left(\frac{N}{n_{\ell+1}^{1/3}} \log n_{\ell+1}\right).$$

This reaches maximum when $\ell + 1 = k$, i.e., at the level above leaves. The corresponding redundancy is $O(N/r^3 \cdot \log r)$.

- Each root's storage table wastes $O(\Delta_1)$ capacity; wasting one capacity results in $O(\log U)$ redundancy. The total redundancy of this part is $O(N/n_1) \cdot O(\Delta_1 \log U) = O(N/\log^2 N)$.
- Key selection. When we select n_ℓ keys to store at u , "which keys are selected" is an implicit piece of redundant information that was stored in the dictionary. Its entropy is at most $\log \binom{\bar{n}_\ell}{\underline{n}_\ell} = O(\Delta_\ell \log n_\ell)$. It is dominated by the redundancy of the waiting list on the same node.
- Lookup table used by minimap. Recall that we use minimap to maintain $n' := \underline{n}_k = O(r^9) = O(\log^{0.9} N)$ key-value pairs from a key universe $U' := 2^{I_k} = \text{poly log } N$, allowing $O(1)$ bits redundancy. By Lemma 3.7, the lookup table size for this minimap is $O(U' n' \binom{U'}{n'}) \cdot (2n')^{n'} = (\text{poly log } N)^{O(\log^{0.9} N)} \ll O(\sqrt{N})$ words.

The largest redundancy appears at the second item: the waiting lists of leaf nodes. Summing up the redundancy from all categories, the overall redundancy of the dictionary is $O(N/r^2) \leq O(R)$ which meets the requirement.

5.3 Query, Insertion, and Deletion

Provided the storage structure of the dictionary, it is straightforward to perform operations efficiently, including queries, insertions, and deletions.

In the beginning of each operation, we need to simulate the hashing process on the input key x , trace all the nodes which x was hashed to, and obtain a path from a root node to a leaf node. During this process, we also record the ID-quotient pairs $(x_{\text{id}}^{[\ell]}, x_{\text{quot}}^{[\ell]})$ in each node along the path. The next step varies depending on the operation:

- Query. For the level- ℓ node on the path, we query $x_{\text{id}}^{[\ell]}$ in its storage table (if it is an inner node) or minimap (if it is a leaf). If at any node the query result is “exist”, and the tail part in the query result equals $x_{\text{quot}}^{[\ell]}$, then we return “exist”.¹⁰ Otherwise, as x is not stored in any node along the path, we report that x is not in the dictionary.
- Insertion. Since all storage tables and minimaps are full except for the root, we insert the key directly into the root’s storage table. We also add x to the waiting lists of all non-root nodes along the path (strictly speaking, add $(x_{\text{id}}^{[\ell]}, x_{\text{id}}^{[\ell-1]})$ to the waiting list of level ℓ).

The deletion procedure is shown in Algorithm 3. Assume x was stored at node u before the deletion. The motivation is to try to delete x from the storage table or minimap of u . To make it full again, we access the waiting list to find another key y that was stored at an ancestor of u . Then we recursively delete y from the ancestor while inserting y into u ’s storage table or minimap. We also carefully maintain the waiting lists along this procedure. See Algorithm 3 for details.

Note that when we read an entry y from the waiting list, we do not know the quotient of y , which is required to insert y to the current node u . The solution is that we first recursively remove y from the ancestors and simultaneously learn its quotient (Line 8).

All above operations have expected running time $O(\log^* N + \log r) = O(\log^* N + \log(N/R))$. First, the tree has $O(\log^* N)$ levels, while updating a storage table or a waiting list takes $O(1)$ expected time. Also, every node on the path will only be visited constant times, so the time on operating inner nodes is $O(\log^* N)$. Second, the minimap on leaf nodes takes $O(\log r)$ update time; $O(1)$ minimap updates are required in every single dictionary operation. Combining them together leads to the desired time complexity. Furthermore, query operations only take $O(\log^* N)$ time because they do not need minimap updates (minimap queries are done in constant time). The query time will be improved to $O(1)$ in Section 5.6.

5.4 Construct the Hash Function

So far, we have introduced the basic construction of our algorithm except for one detail – the hash function h that assigns child indices and next-level IDs for the keys. For each level ℓ , we will specify a function family \mathcal{H}_ℓ , and sample a function h from \mathcal{H}_ℓ uniformly at random for each level- ℓ node u . Recall that we have the following two requirements for h :

- (1) h distributes all keys received by u evenly among its children so that the number of keys assigned to any child v is between $[\underline{n}_{\ell+1}, \bar{n}_{\ell+1}]$.

¹⁰If $x_{\text{id}}^{[\ell]}$ is found but the tail part is not $x_{\text{quot}}^{[\ell]}$, we immediately know x is not in the dictionary, by the uniqueness of the ID.

Algorithm 3: Deletion

```

1 Function DeleteAndGetQuotient( $u, x_{\text{id}}^{[\ell]}$ ):            $\triangleright u$  is the level- $\ell$  node along the path.
2   if  $u$  is the root then
3     Delete  $x_{\text{id}}^{[\ell]}$  from the storage table while getting its quotient  $x_{\text{quot}}^{[\ell]}$ .
4     return  $x_{\text{quot}}^{[\ell]}$ 
5   else if  $x$  is stored in the storage table or minimap of  $u$  then
6     Delete  $x$  from the storage table or minimap while getting its quotient  $x_{\text{quot}}^{[\ell]}$ .
7     Delete an arbitrary key  $y$  from the waiting list of  $u$  while obtaining  $y_{\text{id}}^{[\ell-1]}$ .
8      $y_{\text{quot}}^{[\ell-1]} \leftarrow \text{DeleteAndGetQuotient}(\text{parent}(u), y_{\text{id}}^{[\ell-1]})$ 
9     Insert  $y$  to the storage table or minimap of  $u$ .
10    return  $x_{\text{quot}}^{[\ell]}$ .
11  else                                            $\triangleright x$  is in the waiting list of  $u$ .
12    Delete  $x$  from the waiting list while getting  $x_{\text{id}}^{[\ell-1]}$ .
13     $x_{\text{quot}}^{[\ell-1]} \leftarrow \text{DeleteAndGetQuotient}(\text{parent}(u), x_{\text{id}}^{[\ell-1]})$ 
14    return  $x_{\text{quot}}^{[\ell]}$  obtained from  $x_{\text{quot}}^{[\ell-1]}$  and  $x_{\text{id}}^{[\ell-1]}$ 
15 Let  $u$  be the leaf node along the path.
16 DeleteAndGetQuotient( $u, x_{\text{id}}^{[k]}$ )

```

(2) For a fixed child v of u , all keys assigned to v have different level- $(\ell + 1)$ IDs.

We require these two properties to hold with high probability. If any of them is violated, we immediately resample¹¹ a new h . Besides, we also need h to have small space and evaluation time so that it will not be the bottleneck of our dictionary.

Challenges. Before we give the concrete construction of our hash function, let us first see why picking a uniform random permutation as h does not work. Formally, we first pick a uniform random permutation $h : [2^{\ell}] \rightarrow [B_{\ell}] \times [2^{\ell+1}] \times [Q_{\ell+1}/Q_{\ell}]$; as long as any of the two requirements does not meet, we repeatedly pick another one at random. The problem is that Property (2) above has too low probability to be satisfied. For every child v of u , there are about $n_{\ell+1}$ keys assigned to it. When every key is assigned a random ID in a universe $2^{\ell+1} \approx n_{\ell+1}^4$, the probability of existing an ID collision at v is about $\Theta(1/n_{\ell+1}^2)$. However, we have $B_{\ell} = n_{\ell}/n_{\ell+1} \gg n_{\ell+1}^2$ children, so there is a child violating Property (2) with high probability.

In this example, when some child v of u violates the uniqueness of IDs, we resample the whole function h while ignoring the success of other children. This is indeed unnecessary – by carefully construct the function family \mathcal{H}_{ℓ} , we are able to only reassign level- $(\ell + 1)$ IDs in v while keeping the function values in other children, which is the motivation of our two-stage construction below.

Construction. We let the hash function h be a composition of two subfunctions h^{ch} and h_v^{id} . h^{ch} , sampled from family $\mathcal{H}_{\ell}^{\text{ch}}$ at random, extracts several bits from the input key's x_{id} to be the child index $v \in [B_{\ell}]$; then, for each child v , an individual bijection h_v^{id} (sampled from family $\mathcal{H}_{\ell}^{\text{id}}$)

¹¹The new function may depend on the previous one; it is not necessarily uniformly random from \mathcal{H}_{ℓ} .

extracts several bits again to be the ID inside that child. Formally,

$$\begin{aligned} h^{\text{ch}} : [2^{I_\ell}] &\rightarrow [B_\ell] \times [2^{I_\ell}/B_\ell] && \text{maps } x_{\text{id}} \mapsto (v, y_{\text{prim}}), \\ \forall v \in [B_\ell], \quad h_v^{\text{id}} : [2^{I_\ell}/B_\ell] &\rightarrow [2^{I_{\ell+1}}] \times [2^{I_\ell - I_{\ell+1}}/B_\ell] && \text{maps } y_{\text{prim}} \mapsto (y_{\text{id}}, x_{\text{rem}}). \end{aligned}$$

Here, the intermediate variable y_{prim} is called the *primitive ID*. Similar to y_{id} , y_{prim} is unique among all keys mapped to the child v , but it is much longer. (Its uniqueness comes from the fact that h^{ch} is a bijection.) Note that different children v have different h_v^{id} , which allows us to resample for a single child when an ID collision occur. After decomposing h into two subfunctions, Property (1) becomes a requirement of $h^{\text{ch}} \in \mathcal{H}_\ell^{\text{ch}}$; Property (2) becomes a requirement of $h_v^{\text{id}} \in \mathcal{H}_\ell^{\text{id}}$.

The existence of function families $\mathcal{H}_\ell^{\text{ch}}$, $\mathcal{H}_\ell^{\text{id}}$ is summarized as the following lemmas.

Lemma 5.1. *Let s, L be integers where $1 \leq s \leq (1 - \Omega(1))L$. There is a hash function family \mathcal{H} in which every member $h \in \mathcal{H}$ is a bijection $h : \{0, 1\}^L \rightarrow \{0, 1\}^L$, satisfying:*

- (a) *For any $h \in \mathcal{H}$ and any input $x \in \{0, 1\}^L$, $h(x)$ and $h^{-1}(x)$ can be evaluated in $O(1)$ time.*
- (b) *It takes $O(2^{\varepsilon L})$ bits to store an $h \in \mathcal{H}$, where ε is a constant given in advance which does not depend on s and L .*
- (c) *For $n \geq 2^s \cdot s^4$ different inputs x_1, \dots, x_n , if we divide $h(x_1), \dots, h(x_n)$ into equivalent classes according to the first s bits of $h(x_i)$, then with probability $\geq 1 - \frac{1}{4n^2}$, the number of elements in any equivalent class is between $\left[\frac{n}{2^s} - \frac{1}{5} \left(\frac{n}{2^s} \right)^{2/3}, \frac{n}{2^s} + \frac{1}{5} \left(\frac{n}{2^s} \right)^{2/3} \right]$.*

We defer the proof of Lemma 5.1 to Appendix B and first see how we use it to construct $\mathcal{H}_\ell^{\text{ch}}$. In fact, taking the parameters $\varepsilon = 1/100$, $L = I_\ell = \lceil 4 \log n_\ell \rceil$, and $s = \log B_\ell$ (the premise $s \leq (1 - \Omega(1))L$ holds as $2s \leq 2 \log n_\ell < 4 \log n_\ell \leq L$), we get a function family that supports $O(1)$ time evaluation, and takes space $O(2^{\frac{1}{100}I_\ell}) \leq O(2^{\frac{1}{100}\lceil 4 \log n_\ell \rceil}) \ll O(n_\ell^{2/3})$. Recall that a level- ℓ node maintains a waiting list of space $\Theta(n_\ell^{7/9})$, the space cost of storing the hash function $h^{\text{ch}} \in \mathcal{H}_\ell^{\text{ch}}$ is not the bottleneck.¹²

Substituting $n \in [\underline{n}_\ell, \bar{n}_\ell]$ in Property (c) (the premise $n \geq 2^s \cdot s^4$ holds as $n/2^s = \Theta(n_{\ell+1}) \geq \Omega(\log^9 n_\ell) \gg s^4$), the interval given in (c) becomes

$$\left[\frac{n}{B_\ell} - \frac{1}{5} \left(\frac{n}{B_\ell} \right)^{2/3}, \frac{n}{B_\ell} + \frac{1}{5} \left(\frac{n}{B_\ell} \right)^{2/3} \right].$$

It is easy to verify that this interval is contained¹³ in $[\underline{n}_{\ell+1}, \bar{n}_{\ell+1}]$. For example, the upper limit

$$\begin{aligned} \frac{n}{B_\ell} + \frac{1}{5} \left(\frac{n}{B_\ell} \right)^{2/3} &\leq \bar{n}_\ell \cdot \frac{n_{\ell+1}}{n_\ell} + \frac{1}{5} \left(\bar{n}_\ell \cdot \frac{n_{\ell+1}}{n_\ell} \right)^{2/3} \\ &\leq \left(1 + \frac{\Delta_\ell}{n_\ell} \right) n_{\ell+1} + \frac{1}{5} \left(\left(1 + \frac{\Delta_\ell}{n_\ell} \right) n_{\ell+1} \right)^{2/3} \\ &\leq n_{\ell+1} + \frac{n_{\ell+1}}{n_\ell^{1/3}} + \frac{1}{4} n_{\ell+1}^{2/3} \\ &\leq n_{\ell+1} + n_{\ell+1}^{2/3}, \end{aligned}$$

¹²Although the root nodes do not maintain waiting lists, one can still verify that the space cost of storing h^{ch} is dominated by the redundancy of the node's storage table.

¹³More strictly, the integers in this interval are contained in $[\underline{n}_{\ell+1}, \bar{n}_{\ell+1}]$. It is enough because we are bounding the number of elements in an equivalent class, which is always an integer.

whose integral part equals $\bar{n}_{\ell+1}$. Here, the third inequality holds because $(1 + \Delta_\ell/n_\ell)^{2/3} \leq 5/4$ as long as n_ℓ is not too small; the last inequality holds because $n_\ell^{1/3} \geq (3n_{\ell+1})^{1/3} > \frac{4}{3}n_{\ell+1}^{1/3}$. It is similar for the lower limit.

From (c), the probability of any child overflowing or underflowing is at most $1/4n^2 \leq 1/n_\ell^2$ (we have $n \geq n_\ell/2$ as n_ℓ is larger than a constant).

In summary, by applying Lemma 5.1 with proper parameters, we get the function family $\mathcal{H}_\ell^{\text{ch}}$ such that with probability at least $1 - 1/n_\ell^2$, no child of u overflows or underflows. Moreover, any function $h^{\text{ch}} \in \mathcal{H}_\ell^{\text{ch}}$ and its inverse can be evaluated in constant time; its storage space is also not the bottleneck of our algorithm.

Compared to $\mathcal{H}_\ell^{\text{ch}}$, $\mathcal{H}_\ell^{\text{id}}$ has a stricter space constraint, because each child v of u needs to store a copy of h_v^{id} independently. Correspondingly, its independence constraint is less strict.¹⁴ Similar to Lemma 5.1, we will prove the following lemma in Appendix B, which gives the construction of $\mathcal{H}_\ell^{\text{id}}$:

Lemma 5.2. *Let s, L be integers where $0 < s < L$. There is a hash function family \mathcal{H} in which every member $h \in \mathcal{H}$ is a bijection $h : \{0, 1\}^L \rightarrow \{0, 1\}^L$, satisfying:*

- (a) *For any $h \in \mathcal{H}$ and any input $x \in \{0, 1\}^L$, $h(x)$ and $h^{-1}(x)$ can be evaluated in constant time.*
- (b) *It takes $O(L)$ bits to store an $h \in \mathcal{H}$.*
- (c) *For any integer n and n fixed distinct inputs x_1, \dots, x_n , with probability at least $1 - n^2/2^s$, the first s bits of $h(x_1), \dots, h(x_n)$ are pairwise distinct.*

Applying Lemma 5.2 with $L = I_\ell - \log B_\ell$ and $s = I_{\ell+1}$, we get a function family $\mathcal{H}_\ell^{\text{id}}$ which supports constant time evaluation and $O(I_\ell)$ storage space. We first check that the space usage of h_v^{id} does not become the bottleneck of our dictionary: For a level- ℓ node u and its child v , we use $O(I_\ell)$ bits of space to store h_v^{id} at node v , while v already has a larger redundant waiting list of space $\Omega(I_\ell \cdot \Delta_{\ell+1})$.

In Property (c), let n be any integer in $[n_{\ell+1}, \bar{n}_{\ell+1}]$, then the probability of having a collision is at most

$$\frac{n^2}{2^{I_{\ell+1}}} \leq \frac{\bar{n}_{\ell+1}^2}{n_{\ell+1}^4} \leq \frac{2}{n_{\ell+1}^2},$$

where the last inequality holds as long as $n_{\ell+1}$ is larger than a constant. Hence, for level- ℓ node u and its child v , such hash function h_v^{id} is able to guarantee the uniqueness of the level- $(\ell + 1)$ IDs in v with probability $1 - 2/n_{\ell+1}^2$, which meets our requirement. So far, by combining $h^{\text{ch}} \in \mathcal{H}_\ell^{\text{ch}}$ and $h_v^{\text{id}} \in \mathcal{H}_\ell^{\text{id}}$, we conclude the construction of the hash function h .

Hash functions in the top level. Although the above discussion is all about hash functions on the tree nodes, the bijective hash function that we use to hash initial keys into level-1 buckets (i.e., tree roots) are essentially the same. If U is a power of two, then we just regard the whole key $x \in [U]$ as an ID (while the quotient is empty), and apply Lemmas 5.1 and 5.2 as above to construct corresponding hash functions. If U is not a power of two, we only need to slightly generalize the construction in Lemmas 5.1 and 5.2. The details are deferred to Appendix B.

¹⁴We only need 2-wise independence for h_v^{id} . See the appendix for details.

5.5 Rehashing

When the hash function at some node does not meet the requirements, we need to resample a hash function and then reconstruct part of the data structure. This procedure is called *rehashing*. In this subsection, we show that the rehashing frequency is low enough so that the expected time cost of rehashing will not become the bottleneck of our algorithm.

We start by introducing the behavior of rehashing. Denote by h_u^{ch} the hash function $h^{\text{ch}} \in \mathcal{H}_\ell^{\text{ch}}$ for node u , and recall that $h_v^{\text{id}} \in \mathcal{H}_\ell^{\text{id}}$ is stored at the child v . They distribute the keys received by u to the children and assign IDs of level $\ell + 1$ to these keys. When h_u^{ch} lets a child overflow or underflow, we resample h_u^{ch} uniformly at random from $\mathcal{H}_\ell^{\text{ch}}$, and reconstruct the whole subtree rooted at u . For any child v , when h_v^{id} leads to an ID collision in v , we resample h_v^{id} randomly from $\mathcal{H}_\ell^{\text{id}}$, and then reconstruct the subtree rooted at v .

Notice the similarity between h_u^{ch} and h_u^{id} – any failure among them leads to the reconstruction of the same subtree rooted at u ; also, they have similar failure probabilities ($1/n_\ell^2$ vs $2/n_\ell^2$). For simplicity, we consider these two subfunctions as a whole: let $g_u = (h_u^{\text{ch}}, h_u^{\text{id}}) \in \mathcal{G}_\ell = \mathcal{H}_\ell^{\text{ch}} \times \mathcal{H}_{\ell-1}^{\text{id}}$ be the function stored at u . Note that this grouping is different from previous subsections (where h_u^{ch} is grouped with the children’s h_v^{id}). When one of $h_u^{\text{ch}}, h_u^{\text{id}}$ fails to satisfy the requirement, we say g_u *fails*, in which case we resample the whole g_u instead of only the failed part. From the previous subsection, we know that a random g_u from \mathcal{G}_ℓ fails with probability at most $3/n_\ell^2$ for a fixed set of input keys.

The recursive process of reconstructing a subtree is described below in Algorithm 4.

Algorithm 4: Rehashing

```

1 Function Rehash( $u$ ):
2   Sample  $g_u \in \mathcal{G}_\ell$  uniformly at random.
3   if  $g_u$  fails then
4     Rehash( $u$ )
5   return
6   Distribute the keys to children according to  $g_u$ .
7   foreach child  $v$  of  $u$  do
8     Rehash( $v$ )
9   Fill the storage table or minimap at  $u$ .
10  Fill the waiting list of  $u$  if  $u$  is not the root.

```

Rehashing the subtree rooted at a level- ℓ node u takes $O(n_\ell(k - \ell + 1)) \leq O(n_\ell \log^* n_\ell)$ time: the function **Rehash** takes $O(n_{\ell'})$ expected time processing every level- ℓ' node; summing over all descendants of u leads to this time complexity. It is possible that the resampled g_u still fails, in which case we immediately rehash u once more.

When the hash function g fails at an ancestor p of u , the **Rehash**(u) is also called in the recursion. In this case, we say p *causes* the rehashing. However, we still count the time cost of this rehashing on every descendant of p . Once **Rehash**(u) is called, we count $O(n_\ell)$ time cost on node u , regardless whether u causes the rehashing or not. Then, the total time usage of rehashing equals the sum of time costs over all nodes. Based on this idea, we prove the following lemma:

Lemma 5.3. *Assume there are M operations (insertion/deletion) after the initialization of the dictionary with capacity N . The expected time of rehashing in this procedure is at most $O((N + M) \log^* N)$.*

Proof. Let u be a level- ℓ node. We consider every maximal consecutive subsequence A of operations such that no ancestor of u is rehashed. We also ignore the operations in A where the key to insert or delete is not hashed into u . Throughout A , u may be rehashed multiple times (which are all caused by u), but the input of g_u (i.e., the primitive ID x_{prim} of each key x) is unchanged except for the keys being deleted or inserted. Below, we first derive an upper bound for such a maximal subsequence, then sum over all nodes u and maximal subsequences. When doing such analysis, we fix the hash functions g and the rehashing history of u 's ancestors, and only consider the randomness of sampling g_u when rehashing.

Claim 5.4. *Assume u is a level- ℓ node and A is a maximal consecutive subsequence of operations where u 's ancestors are not rehashed, where at the t -th operation in A there is a rehashing caused by u . Then, with probability at least $1/2$ under the randomness of resampled g_u , there will not be rehashing caused by u in the next $\lceil n_\ell^2/10 \rceil$ operations in A .*

Proof. We define S_i as the set of primitive IDs of the keys hashed into u after the $(t+i)$ -th operation in A , for $0 \leq i \leq \lceil n_\ell^2/10 \rceil$. For a random $g_u \in \mathcal{G}_\ell$, the probability that g_u fails on S_i is at most $3/n_\ell^2$, provided $|S_i| \in [\underline{n}_\ell, \bar{n}_\ell]$, which is guaranteed since there is no rehashing at u 's ancestors. Applying the union bound over $i = 0, 1, \dots, \lceil n_\ell^2/10 \rceil$, we know that $S_0, \dots, S_{\lceil n_\ell^2/10 \rceil}$ do not let g_u fail with probability at least $1 - \left(\frac{n_\ell^2}{10} + 1\right) \left(\frac{3}{n_\ell^2}\right) \geq \frac{1}{2}$. In the latter case, u will not be rehashed in the next $\lceil n_\ell^2/10 \rceil$ operations. \square

As a direct implication of Claim 5.4, if the subsequence A consists of T operations, then the expected number of rehashings of u is $O(T/n_\ell^2 + 1)$. The term $O(1)$ refers to the inevitable rehashing of u at the beginning of A , which is caused by an ancestor of u .

Next, we take the summation of the time costs over all nodes. Denote by a_ℓ the total number of rehashings of level- ℓ nodes. We write the recurrence

$$a_\ell \leq a_{\ell-1} \cdot B_{\ell-1} + \frac{M}{n_\ell^2}, \quad (\ell > 1)$$

where the first term refers to the number of rehashings that are recursively called from the parents; the second term is the summation of $O(T/n_\ell^2)$ terms, as the sum of T of all level- ℓ nodes equals M since they form a partition of the operation sequence. (The notation O is omitted for cleanliness.) Similarly, the initial value is $a_1 \leq \frac{N}{n_1} + \frac{M}{n_1^2}$. Solving this recurrence, we have

$$a_\ell \leq \frac{N}{n_\ell} + \frac{M}{n_\ell} \sum_{i=1}^{\ell} \frac{1}{n_i} \leq \frac{N}{n_\ell} + \frac{M}{n_\ell}.$$

Each rehashing of level ℓ takes $O(n_\ell)$ time, so the total time usage of all rehashings is

$$\sum_{\ell=1}^k a_\ell \cdot n_\ell \leq k(N + M) \leq (N + M) \log^* N. \quad \square$$

5.6 Achieving Constant-Time Queries

The dictionary we described so far has $\Theta(\log^* N)$ query time, which is due to simulating the hashing process on the input key x along the path of length $O(\log^* N)$ from a root to a leaf. We can improve it as follows: After only $O(1)$ hashing steps at the beginning, the IDs of the keys become very short ($\ll \log \log N$), and the memory usage to store the set of IDs is far less than a word. If we rearrange

the order of information in the memory such that the information of ID set occurs consecutively, it is possible to use lookup tables to complete the remaining steps of the query. This will improve the query time to $O(1)$ in the worst case.

Specifically, each level- ℓ node u ($\ell \geq 3$) is responsible for storing a set of $\text{poly log}^{(\ell)} N \ll \log \log N$ IDs from a universe $2^{I_\ell} \ll \log \log N$, and an associated quotient of $O(w)$ bits for each ID. Our first observation is that we can store the ID and quotient parts separately. On leaf nodes, we used minimaps to store the information, which consists of the ID set and a fixed number of quotient slots. Even if we store these two parts in different regions of the memory, the minimap still works.¹⁵ For internal nodes, we used succinct hash tables from [BFK⁺22] to store a set of ID-quotient pairs. Their hash tables also store quotients in fixed slots, and thus work even if we store the metadata (including the ID set) separately from the quotient slots. We also store the hash functions at a node as part of the metadata.

Then, we store the metadata at all nodes in the depth-first pre-order in the memory, so that the metadata within a subtree are arranged consecutively. For the level- ℓ node u ($\ell \geq 3$), the total space usage of the metadata for all its descendants does not exceed $\text{poly log log } N \ll \log^{0.1} N$, which fits in a machine word so that we can access in $O(1)$ word-accesses. Furthermore, these metadata are enough for us to perform the remaining steps of the query, locating the quotient slot we need to access (if there is one). A lookup table of $2^{O(\log^{0.1} N)} = N^{o(1)}$ words is sufficient to support it in worst-case constant time.

In sum, by rearranging the order of storing information in the memory, and with the help of a lookup table, we can improve the query time of the dictionary to constant in the worst case. This covers the regime of $R \geq N/\log^{0.1} N$ in Theorem 1.1.

References

- [ANS09] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP): Part I*, pages 107–118, 2009.
- [ANS10] 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.
- [BCF⁺23] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. Iceberg hashing: Optimizing many hash-table criteria at once. Preprint arXiv:2109.04548 [cs.DS], 2023.
- [BFK⁺22] 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 Symposium on Theory of Computing (STOC)*, pages 1284–1297, June 2022.
- [BKP⁺22] Aaron Berger, William Kuszmaul, Adam Polak, Jonathan Tidor, and Nicole Wein. Memoryless worker-task assignment with polylogarithmic switching cost. In *Proc. 49th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 229, pages 19:1–19:19, 2022.

¹⁵Technically, we extracted a spillover from every quotient before storing it in the memory, which is encoded and stored along with the key set (in the former part).

- [DKM⁺94] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.
- [DM90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 6–19, 1990.
- [DMPP06] Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Proc. 7th Latin American Conference on Theoretical Informatics (LATIN)*, pages 349–361, 2006.
- [FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, June 1984.
- [FPSS03] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. In *Proc. 20th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 271–282, 2003.
- [KLL⁺97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. 29th ACM Symposium on the Theory of Computing (STOC)*, pages 654–663, 1997.
- [Knu63] Don Knuth. Notes on “open” addressing, 1963. Available at <https://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf>.
- [Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
- [Kus23] William Kuszmaul. Strongly history-independent storage allocation: New upper and lower bounds. In *Proc. 64th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2023.
- [LLYZ23a] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. Dynamic “succincter”. In *Proc. 64th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2023.
- [LLYZ23b] 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)*, 2023.
- [LYY20] Mingmou Liu, Yitong Yin, and Huacheng Yu. Succinct filters for sets of unknown sizes. In *Proc. 47th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 168, pages 79:1–79:19, 2020.
- [Pag01] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.
- [Păt08] Mihai Pătraşcu. Succincter. In *Proc. 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.
- [PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

- [RR03] Rajeev Raman and Satti Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th International Conference on Automata, Languages and Programming (ICALP)*, pages 357–368, 2003.
- [SSS95] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.
- [Tho13] Mikkel Thorup. Simple tabulation, fast expanders, double tabulation, and high independence. In *Proc. 54th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 90–99, 2013.
- [Yu20] Huacheng Yu. Nearly optimal static Las Vegas succinct dictionary. In *Proc. 52nd ACM-SIGACT Symposium on Theory of Computing (STOC)*, pages 1389–1401, 2020.

A Proof of Random Swap Lemma

In this section, we provide the proof of the following random swap lemma introduced in Section 4:

Lemma 4.3 (Restated). *We can store a word-dominant VM containing M bits ($2w < M < Nw$) into a WVM with the same total number of bits $wL_{\text{word}} + L_{\text{bit}} = M$, while any given operation to the VM can be transformed into several operations to the WVM within constant expected additional computation time. Moreover:*

- (a) *An access to the VM (either a complete word or the tail) is transformed into $1+O(\min(1, w^2/M))$ accesses to the WVM in expectation.*
- (b) *Allocating/releasing a word or a bit in the VM can be transformed into $O(1)$ allocation/release operations followed by $O(1)$ accesses to the WVM (both in expectation).*
- (c) *The transformation from VM to WVM does not introduce additional space usage, except that we need to access an $O(\log^2 N)$ -bit random seed which can be shared between multiple instances. L_{word} and L_{bit} are fully determined by M and the seed.*

Proof. For the given VM of M bits, we use $L := \lfloor M/w \rfloor$ to denote the number of words in it, $T := M - wL$ to denote the number of bits in its tail, and m to denote its whole memory string. Below we illustrate how to store this VM with memory string m into a WVM.

As mentioned in Section 4.2, we swap the tail with a random word to amortize the slow access to the tail. The first step is to cut the tail into T individual bits and store them into the bit tape of the WVM. When the tail is “too short” and the number of words is not “very few” (explained later), we additionally cut the last word into w bits as well. Next, we swap the contents cut into bits with one or two preceding words at a random location. Finally, the bit tape contains the new individual bits (after swap), and we store the rest of the words into the word tape. See Algorithm 5.

Note that the algorithm exhibits different behaviors depending on the length of m . There are three cases:

- The *Few-Word Case*: the number of words $L \leq L_r$.
- The *Short-Tail Case*: the length of the tail $T \leq a$ while $L > L_r$, which is the only case we additionally cut the last word into individual bits.

Algorithm 5: Storing VM into WVM

```

1 Function Store( $m$ ): ▷ Store a VM with memory string  $m$  into a WVM
2   Sample  $L_r$  from  $[\frac{w}{20}, \frac{w}{11})$  uniformly at random
3   foreach  $i \in [\log N]$  do
4     Sample  $L_i$  from  $[2^i, 2^{i+1})$  uniformly at random
5     Sample  $s_i$  from  $[1, L_i]$  uniformly at random
6   Sample  $a$  from  $[0, w - 2]$  uniformly at random
7    $k \leftarrow$  the unique index such that  $L_k < L \leq L_{k+1}$ 
8   if  $L \leq L_r$  or  $T > a$  then
9     Swap the tail of  $m$  with the first  $T$  bits of the  $s_k$ -th word of  $m$ 
10     $m_{\text{temp}} \leftarrow$  the resulting memory string after swap
11    return a WVM with its word tape storing all  $L$  words of  $m_{\text{temp}}$  and its bit tape
12    storing the tail of  $m_{\text{temp}}$ 
13  else
14    Swap the last word of  $m$  with the  $s_k$ -th word of  $m$ 
15    Swap the tail of  $m$  with the first  $T$  bits of the  $((s_k + 1) \bmod L_k)$ -th word of  $m$ 
16     $m_{\text{temp}} \leftarrow$  the resulting memory string after swapp
17    return a WVM with it word tape storing the first  $L - 1$  words of  $m_{\text{temp}}$  and the bit
18    tape storing the last word plus the tail of  $m_{\text{temp}}$ 

```

- The *Long-Tail Case*: $T > a$ while $L > L_r$.

The random numbers sampled on Lines 2 to 6 serve as the random seed of the whole lemma, which does not change with variations in m . One can check that the random seed has at most $O(\log^2 N)$ bits, and $L_{\text{word}}, L_{\text{bit}}$ are fully determined by M and the seed, i.e., Condition (c) in Lemma 4.3 holds.

Based on this algorithm, we can transform the access and allocation/release operations in VM to those in WVM. As we know which words to swap (from M and the seed), it is easy to transform an access to m into several accesses to the WVM without additional time consumption (i.e., the time taken is linear in the number of transformed operations). Similarly, an allocation or release in the VM can be transformed to a series of necessary operations in the WVM without additional time.

First, note that for any given operation in the VM, the number of transformed operations to the *word tape* is at most $O(1)$, and specially, is exactly 1 for a VM access. Therefore, it suffices to bound the expected number of operations to the *bit tape*, which does not exceed the length of the bit tape L_{bit} times the probability of accessing the bit tape. We have $L_{\text{bit}} = O(\min(L, w))$, because

- In the Few-Word Case, the number of bits equals $L_{\text{bit}} = T \leq 10L < w$ because the VM is word-dominant, and thus $T = O(\min(L, w))$.
- Otherwise, $\min(L, w) = \Theta(w)$, and the number of bits $L_{\text{bit}} \leq T + w \leq 2w = O(\min(L, w))$.

Once the bit tape is accessed (except for the $O(1)$ inevitable operations when we allocate or release 1 bit on the VM), we pretend there are $\Theta(\min(L, w))$ bit-accesses (plus $\Theta(\min(L, w))$ allocations/releases if it was a VM allocation/release), as a clear upper bound of the real cost. Next, it only remains to bound the probability of accessing the bit tape.

Access operations. The access to the bit tape happens only when we access the s_k -th original word, or the $((s_k + 1) \bmod L_k)$ -th word in the Short-Tail Case. As s_k is uniformly sampled from $[1, L_k]$, the probability to access the bit tape is at most $2/L_k \leq 8/L_{k+1} \leq 8/L$. Hence, the expected number of induced accesses to the bit tape is at most $O(\min(L, w)) \cdot (8/L) = O(\min(1, w/L)) = O(\min(1, w^2/M))$ in expectation. Adding the 1 transformed access to the word tape implies Lemma 4.3 (a).

Allocating/releasing a word. By symmetry, we only consider the case of allocation. According to Algorithm 5, the bit tape stores the substring of m that starts from the first bit of the s_k -th word and has length L_{bit} . Hence, the content of the bit tape changes only in the following two situations:

- The number of words L crosses over the threshold L_r and switches from the Few-Word Case to the Short-Tail Case. In this situation, L_{bit} will increase by w . As L_r is uniformly sampled from $[w/20, w/11]$, the probability of this case is at most $O(1/w)$.
- The number of words L crosses over the threshold L_{k+1} and changes the index k , thus s_k is replaced by s_{k+1} . As L_{k+1} is uniformly sampled from $[2^{k+1}, 2^{k+2}]$, the probability of this case is at most $1/2^{k+1} = O(1/L)$.

Combining them together, the probability of changing the bit tape is at most $O(\max(1/w, 1/L))$, hence the expected number of operations in the bit tape is at most $O(\max(1/w, 1/L) \cdot \min(L, w)) = O(1)$.

Allocating/releasing a bit. In most of the cases during the allocation, $L_{\text{bit}} = T$ or $T + w$ only increases by 1, so we only need to (1) allocate 1 bit on the bit tape, and (2) write into the newly allocated bit the swapped content (the $(T + 1)$ -th bit of some preceding word), which takes $O(1)$ bit-tape operations. (The contents in the original L_{bit} bits remain unchanged, so we do not need to access them.) Other additional bit-tape operations will only happen in the following situations:

- The number of words L crosses over the threshold L_r and switches from the Few-Word Case to the Short-Tail Case. As argued above, its probability is at most $O(1/w)$.
- The number of words L crosses over the threshold L_{k+1} and changes the index k . As argued above, the probability is at most $O(1/L)$.
- The number of bits in the tail T crosses over the threshold a and switches from the Short-Tail Case to the Long-Tail Case, thus decreases L_{bit} by $w - 1$. As a is uniformly sampled from $[0, w - 2]$, the probability of this case is at most $O(1/w)$.

Notice that when we allocate 1 bit upon $T = w - 1$, changing T to 0, no additional action is required: The initial state cannot be the Few-Word Case, as there are at most $T \leq 10L \leq 10L_r < w - 1$ bits in the tail; since $a \in [0, w - 2]$, this allocation turns the Long-Tail Case to the Short-Tail Case, thus L_{bit} only increases by one, which falls back to the “common case” mentioned above.

Combining them together, the probability of additional bit-tape operations is at most $O(\max(1/w, 1/L))$. Hence, the expected number of transformed operations is still at most $O(\max(1/w, 1/L) \cdot \min(L, w)) = O(1)$. \square

B Hash Permutation Families

In this section, we prove Lemmas 5.1 and 5.2. They assert existence of hash permutation families with low time and space usage as well as good concentration guarantees. The main technique used here is “Feistel permutation”:

Definition B.1 (Feistel permutation). *For any $x \in \{0, 1\}^L$ and $1 \leq s < L$, we denote by x_L, x_R the first s bits and the last $L - s$ bits of x , respectively, written $x = x_L \parallel x_R$. Fixing a function $f : \{0, 1\}^{L-s} \rightarrow \{0, 1\}^s$, its Feistel permutation $\pi_f : \{0, 1\}^L \rightarrow \{0, 1\}^L$ is defined as*

$$\pi_f : x \mapsto (x_L \oplus f(x_R)) \parallel x_R.$$

It is easy to verify that π_f is indeed a permutation over $\{0, 1\}^L$, and $\pi_f(\pi_f(x)) = x$.

In order to construct hash permutation family \mathcal{H} , we only need to construct a family \mathcal{F} of hash functions, which induces a family of Feistel permutations $\mathcal{H} = \{\pi_f : f \in \mathcal{F}\}$. For Lemma 5.1, we use the following hash function family.

Claim B.2 ([Tho13]). *For any fixed constant c , there is a hash function family \mathcal{F} , in which every function*

$$f : \{0, 1\}^{L-s} \rightarrow \{0, 1\}^s$$

can be evaluated in $O(c)$ time and can be stored in $O(2^{(L-s)/c} \cdot L)$ bits. Moreover, \mathcal{F} is $\Omega(2^{(L-s)/c^2})$ -wise independent.

Based on this hash function family, we prove Lemma 5.1.

Lemma 5.1 (Restated). *Let s, L be integers where $1 \leq s \leq (1 - \Omega(1))L$. There is a hash function family \mathcal{H} in which every member $h \in \mathcal{H}$ is a bijection $h : \{0, 1\}^L \rightarrow \{0, 1\}^L$, satisfying:*

- (a) *For any $h \in \mathcal{H}$ and any input $x \in \{0, 1\}^L$, $h(x)$ and $h^{-1}(x)$ can be evaluated in $O(1)$ time.*
- (b) *It takes $O(2^{\varepsilon L})$ bits to store an $h \in \mathcal{H}$, where ε is a constant given in advance which does not depend on s and L .*
- (c) *For $n \geq 2^s \cdot s^4$ different inputs x_1, \dots, x_n , if we divide $h(x_1), \dots, h(x_n)$ into equivalent classes according to the first s bits of $h(x_i)$, then with probability $\geq 1 - \frac{1}{4n^2}$, the number of elements in any equivalent class is between $\left[\frac{n}{2^s} - \frac{1}{5} \left(\frac{n}{2^s} \right)^{2/3}, \frac{n}{2^s} + \frac{1}{5} \left(\frac{n}{2^s} \right)^{2/3} \right]$.*

Proof of Lemma 5.1. Applying Claim B.2 with $c = 2/\varepsilon$ gives a hash function family \mathcal{F} ; let $\mathcal{H} = \{\pi_f : f \in \mathcal{F}\}$ be the induced family of Feistel permutations. It is clear that any $h \in \mathcal{H}$ can be evaluated in constant time (as well as its inverse $h^{-1} = h$), and can be stored within $O(2^{(L-s)\varepsilon/2} \cdot L) = O(2^{\varepsilon L})$ bits.

It remains to check (c). For convenience, we use $\hat{h}(x)$ to represent the first s bits of $h(x)$. By the definition of Feistel permutations, we know $\hat{h}(x) = x_L \oplus f(x_R)$ (assuming $x = x_L \parallel x_R$). We make the following claim:

Claim B.3. *Assume $n = \Omega(2^s \cdot s^4)$. We arbitrarily fix distinct strings $x_1, \dots, x_n \in \{0, 1\}^L$, $y \in \{0, 1\}^s$, and sample $h \in \mathcal{H}$ uniformly at random. Let random variable $X := |\hat{h}^{-1}(y)|$ denote the number of $i \in [n]$ with $\hat{h}(x_i) = y$, then*

$$\Pr_{h \in \mathcal{H}} \left[\left| X - \frac{n}{2^s} \right| \geq \frac{1}{5} \left(\frac{n}{2^s} \right)^{2/3} \right] \leq \frac{1}{4n^3}.$$

We first assume the claim holds. By the union bound, the probability that there is an equivalent class whose number of elements lies outside $\left[\frac{n}{2^s} - \frac{1}{5}\left(\frac{n}{2^s}\right)^{2/3}, \frac{n}{2^s} + \frac{1}{5}\left(\frac{n}{2^s}\right)^{2/3}\right]$ is at most

$$\Pr_{h \in \mathcal{H}} \left[\exists y \in \{0, 1\}^s : |\hat{h}^{-1}(y)| \notin \left[\frac{n}{2^s} - \frac{1}{5}\left(\frac{n}{2^s}\right)^{2/3}, \frac{n}{2^s} + \frac{1}{5}\left(\frac{n}{2^s}\right)^{2/3} \right] \right] \leq \frac{1}{4n^3} \cdot 2^s \leq \frac{1}{4n^2},$$

which implies Property (c). Now it only remains to prove this claim.

Assume $x_i = x_{L,i} \parallel x_{R,i}$ where $x_{L,i}$ is the first s bits of x_i . We divide x_1, \dots, x_n into groups according to the right part $x_{R,i}$. Formally, we let $\{z_1, \dots, z_m\} = \{x_{R,1}, \dots, x_{R,n}\}$ be the set of occurred x_R 's, and let $I_j := \{i \in [n] : x_{R,i} = z_j\}$.

For $i_1 \neq i_2 \in I_j$, we know $\hat{h}(x_{i_1}) \neq \hat{h}(x_{i_2})$. It is because $x_{R,i_1} = x_{R,i_2} = z_j$, thus $\hat{h}(x_{i_1}) = x_{L,i_1} \oplus f(z_j) \neq x_{L,i_2} \oplus f(z_j) = \hat{h}(x_{i_2})$. Therefore, for every $j \in [m]$, there is at most one $i \in I_j$ with $\hat{h}(x_i) = y$. Let $X_j := \mathbb{1}[\exists i \in I_j : \hat{h}(x_i) = y]$. Then we have

- $X_j = \mathbb{1}[\exists i \in I_j : x_{L,i} \oplus f(z_j) = y] = \mathbb{1}[f(z_j) \in \{x_{L,i} \oplus y : i \in I_j\}]$ is solely determined by $f(z_j)$. For a random $f \in \mathcal{F}$, its expectation is $\mathbb{E}[X_j] = \frac{|I_j|}{2^s}$.
- $X := |\hat{h}^{-1}(y)| = |\{i \in [n] : \hat{h}(x_i) = y\}| = X_1 + \dots + X_m$. Its expectation equals $\mathbb{E}[X] = \sum_{j=1}^m \frac{|I_j|}{2^s} = \frac{n}{2^s}$.
- Let $k := 10 \ln n$, then any k -subset of $\{X_1, \dots, X_m\}$ is independent. This is because \mathcal{F} is $\Omega(2^{(L-s)\varepsilon^2/4})$ -wise independent, and

$$k := 10 \ln n \leq (10 \ln 2)L = 2^{o(L)} \leq o(2^{(L-s)\varepsilon^2/4}).$$

We then apply the Chernoff bound with limited independence on $X = X_1 + \dots + X_m$, stated below:

Theorem B.4 ([SSS95, Theorem 5]). *If X is the sum of k -wise independent r.v.'s, each of which is confined to the interval $[0, 1]$, with $\mu = \mathbb{E}[X]$. For $\delta \leq 1$, if $k \leq \lfloor \delta^2 \mu e^{-1/3} \rfloor$, then $\Pr[|X - \mu| \geq \delta \mu] \leq e^{-\lfloor k/2 \rfloor}$.*

Now let $\delta = \frac{1}{5}\left(\frac{n}{2^s}\right)^{-1/3}$ and apply Theorem B.4. It is easy to check that

$$\delta^2 \mu e^{-1/3} = \frac{1}{25e^{1/3}} \left(\frac{n}{2^s}\right)^{1/3} \geq \frac{1}{25(2e)^{1/3}} \log^{4/3} n \gg k = 10 \ln n,$$

where the first inequality holds since $\frac{n}{\log^4 n} \geq \frac{2^s \cdot s^4}{(s+4 \log s)^4} \geq \frac{1}{2} \cdot 2^s$ (for sufficiently large s), which is equivalent to $\frac{n}{2^s} \geq \frac{1}{2} \cdot \log^4 n$. Theorem B.4 gives

$$\Pr \left[\left| X - \frac{n}{2^s} \right| \geq \frac{1}{5} \left(\frac{n}{2^s}\right)^{2/3} \right] \leq e^{-\lfloor k/2 \rfloor} \leq e^{-5 \ln n + 1} \ll \frac{1}{4n^3}.$$

This proves Claim B.3 and further implies Lemma 5.1. \square

Lemma 5.2 can be proved similarly using Feistel permutations.

Lemma 5.2 (Restated). *Let s, L be integers where $0 < s < L$. There is a hash function family \mathcal{H} in which every member $h \in \mathcal{H}$ is a bijection $h : \{0, 1\}^L \rightarrow \{0, 1\}^L$, satisfying:*

- For any $h \in \mathcal{H}$ and any input $x \in \{0, 1\}^L$, $h(x)$ and $h^{-1}(x)$ can be evaluated in constant time.

(b) It takes $O(L)$ bits to store an $h \in \mathcal{H}$.

(c) For any integer n and n fixed distinct inputs x_1, \dots, x_n , with probability at least $1 - n^2/2^s$, the first s bits of $h(x_1), \dots, h(x_n)$ are pairwise distinct.

Proof of Lemma 5.2. It is well-known that there exists a 2-independent hash function family \mathcal{F} , in which every function $f : \{0, 1\}^{L-s} \rightarrow \{0, 1\}^s$ can be stored within $O(L)$ bits and evaluated in constant time (e.g., polynomial hashing over a finite field \mathbb{F}_{2^L}). For distinct inputs $x_1 \neq x_2 \in \{0, 1\}^{L-s}$ and outputs $y_1, y_2 \in \{0, 1\}^s$, $\Pr[f(x_1) = y_1 \wedge f(x_2) = y_2] = 2^{-2s}$.

We then verify that \mathcal{F} 's Feistel permutation family $\mathcal{H} := \{\pi_f : f \in \mathcal{F}\}$ satisfies the requirements. First, $h \in \mathcal{H}$ (which equals its inverse) can be evaluated in $O(1)$ time, and can be stored using $O(L)$ bits of space. Next, let $h \in \mathcal{H}$ be chosen uniformly at random, and denote by $\hat{h}(x)$ the first s bits of $h(x)$. For any two distinct inputs $x_1 \neq x_2 \in \{0, 1\}^{L-s}$, the probability of $\hat{h}(x_1) = \hat{h}(x_2)$ is at most 2^{-s} , because:

- Assume $x_1 = x_{L,1} \parallel x_{R,1}$ and $x_2 = x_{L,2} \parallel x_{R,2}$. If $x_{R,1} = x_{R,2}$, we always have $\hat{h}(x_1) = x_{L,1} \oplus f(x_{R,1}) \neq x_{L,2} \oplus f(x_{R,2}) = \hat{h}(x_2)$.
- Otherwise, $f(x_{R,1})$ and $f(x_{R,2})$ are independent r.v.'s following the uniform distribution over $\{0, 1\}^s$. Therefore, $\Pr[\hat{h}(x_1) = \hat{h}(x_2)] = \Pr[f(x_{R,1}) \oplus x_{L,1} = f(x_{R,2}) \oplus x_{L,2}] = 2^{-s}$.

Combining these two cases tells that $\Pr[\hat{h}(x_1) = \hat{h}(x_2)] \leq 2^{-s}$ for all $x_1 \neq x_2$. Taking the union bound over all pairs among x_1, \dots, x_n gives Property (c). \square

Generalizations for the top level. In Sections 4 and 5, when hashing all N keys into N/n_1 buckets, the input domain of the hash function might not be a power of two, in which case we use the following generalization of Lemma 5.1. It just replaces the rightmost $L - s$ bits of the domain $(\{0, 1\}^{L-s})$ with a more general set $[Q]$.

Corollary B.5. *Let s, T be integers where $1 \leq s \leq (1 - \Omega(1))(s + \log T)$. There is a hash function family \mathcal{H} in which every member $h \in \mathcal{H}$ is a permutation over $\{0, 1\}^s \times [T]$, satisfying:*

- For any $h \in \mathcal{H}$ and any input $x \in \{0, 1\}^s \times [T]$, $h(x)$ and $h^{-1}(x)$ can be evaluated in $O(1)$ time.
- It takes $O((2^s \cdot T)^\epsilon)$ bits to store an $h \in \mathcal{H}$, where ϵ is a pre-given constant which does not depend on s and T .
- For $n \geq 2^s \cdot s^4$ different inputs x_1, \dots, x_n , if we divide $h(x_1), \dots, h(x_n)$ into equivalent classes according to the first s bits (i.e., the $\{0, 1\}^s$ part) of $h(x_i)$, then with probability $\geq 1 - \frac{1}{4n^2}$, the number of elements in any equivalent class is between $\left[\frac{n}{2^s} - \frac{1}{5} \left(\frac{n}{2^s}\right)^{2/3}, \frac{n}{2^s} + \frac{1}{5} \left(\frac{n}{2^s}\right)^{2/3}\right]$.

The proof is almost the same as Lemma 5.1. For every $x = (x_L, x_R) \in \{0, 1\}^s \times [T]$, we treat $x_L \in \{0, 1\}^s$ as the left part and $x_R \in [T]$ as the right part, and define *Feistel permutations* similarly to Definition B.1: For any function $f : [T] \rightarrow \{0, 1\}^s$, its Feistel permutation is $\pi_f : x \mapsto (x_L \oplus f(x_R), x_R)$. Then, we take a function family \mathcal{F} from $[T]$ to $\{0, 1\}^s$ according to Claim B.2, and apply the same argument as in the proof of Lemma 5.1 to prove the corollary.

A similar corollary of Lemma 5.2 holds as well:

Corollary B.6. *Let s, T be positive integers. There is a hash function family \mathcal{H} in which every member $h \in \mathcal{H}$ is a permutation over $\{0, 1\}^s \times [T]$, satisfying:*

- (a) For any $h \in \mathcal{H}$ and any input $x \in \{0, 1\}^s \times [T]$, $h(x)$ and $h^{-1}(x)$ can be evaluated in constant time.
- (b) It takes $O(s + \log T)$ bits to store an $h \in \mathcal{H}$.
- (c) For any integer n and n fixed distinct inputs x_1, \dots, x_n , with probability at least $1 - n^2/2^s$, the first s bits (i.e., the $\{0, 1\}^s$ part) of $h(x_1), \dots, h(x_n)$ are pairwise distinct.

C Hashing for Section 4

The construction for hash functions and the strategy of rehashing for Section 4 is similar to that in Sections 5.4 and 5.5. We introduce them in this appendix. All notations here are the same as in Section 4 unless otherwise specified.

Recall that, at the top level of the data structure, we used two hash functions $h^{(1)} : [U] \rightarrow [N/n_1] \times [2^{I_1}] \times [Q]$ and $h_u^{(2)} : [2^{I_1}] \rightarrow [B^{8h-1}] \times [2^{I_{sh}}]$ to map every key x into a leaf node. Here, $h_u^{(2)}$ can be different for different root nodes, and we store N/n_1 independent instances of it, while for $h^{(1)}$ we only have one instance. Both hash functions need to prevent *overflows* and *underflows* with high probability, while $h^{(1)}$ also needs to prevent *ID collisions*, which means the same ID is assigned to two keys x_1, x_2 that are hashed to the same level-1 bucket. If either of these conditions fail, we resample the failed hash functions and reconstruct part of the data structure.

Similar to Section 5.4, $h^{(1)}$ is composed of two (bijective) subfunctions: $h^{\text{ch}} : [U] \rightarrow [N/n_1] \times [2^{I_1}Q]$ and $h_u^{\text{id}} : [2^{I_1}Q] \rightarrow [2^{I_1}] \times [Q]$, where the former extracts the bucket index that the key is hashed to, and the latter extracts the key's ID within that bucket. The second subfunction h_u^{id} can be different for different level-1 buckets. h^{ch} and h_u^{id} are sampled uniformly at random from the permutation families given by Corollary B.5 and Corollary B.6 respectively, restated below, with $(s, T, \varepsilon) = (\log(N/n_1), 2^{I_1}Q, \frac{1}{100(1+\alpha)})$ and $(s, T) = (I_1, Q)$ respectively.

Corollary B.5 (Restated). *Let s, T be integers where $1 \leq s \leq (1 - \Omega(1))(s + \log T)$. There is a hash function family \mathcal{H} in which every member $h \in \mathcal{H}$ is a permutation over $\{0, 1\}^s \times [T]$, satisfying:*

- (a) For any $h \in \mathcal{H}$ and any input $x \in \{0, 1\}^s \times [T]$, $h(x)$ and $h^{-1}(x)$ can be evaluated in $O(1)$ time.
- (b) It takes $O((2^s \cdot T)^\varepsilon)$ bits to store an $h \in \mathcal{H}$, where ε is a pre-given constant which does not depend on s and T .
- (c) For $n \geq 2^s \cdot s^4$ different inputs x_1, \dots, x_n , if we divide $h(x_1), \dots, h(x_n)$ into equivalent classes according to the first s bits (i.e., the $\{0, 1\}^s$ part) of $h(x_i)$, then with probability $\geq 1 - \frac{1}{4n^2}$, the number of elements in any equivalent class is between $\left[\frac{n}{2^s} - \frac{1}{5} \left(\frac{n}{2^s} \right)^{2/3}, \frac{n}{2^s} + \frac{1}{5} \left(\frac{n}{2^s} \right)^{2/3} \right]$.

Corollary B.6 (Restated). *Let s, T be positive integers. There is a hash function family \mathcal{H} in which every member $h \in \mathcal{H}$ is a permutation over $\{0, 1\}^s \times [T]$, satisfying:*

- (a) For any $h \in \mathcal{H}$ and any input $x \in \{0, 1\}^s \times [T]$, $h(x)$ and $h^{-1}(x)$ can be evaluated in constant time.
- (b) It takes $O(s + \log T)$ bits to store an $h \in \mathcal{H}$.
- (c) For any integer n and n fixed distinct inputs x_1, \dots, x_n , with probability at least $1 - n^2/2^s$, the first s bits (i.e., the $\{0, 1\}^s$ part) of $h(x_1), \dots, h(x_n)$ are pairwise distinct.

Property (c) in Corollary B.5 tells that the probability of existing an overflowing or underflowing level-1 bucket does not exceed $\frac{1}{4N^2}$. Once overflows or underflows occur, we immediately pick different hash functions and reconstruct the whole dictionary, using at most $O(Nh)$ time.

Property (c) in Corollary B.6 tells that the probability of ID collisions within any given level-1 bucket does not exceed $n_1^2/2^{I_1} \leq n_1^{-2}$. Once ID collisions occur at level-1 bucket u , we resample h_u^{id} (but not h^{ch}) and rebuild the adapter tree below u , which takes $O(n_1h)$ time.

The construction of $h_u^{(2)}$ is simpler – we directly sample $h_u^{(2)}$ according to Corollary B.5 with $s = (8h - 1) \log B$, $T = 2^{I_{8h}}$, and $\varepsilon = 1/100$. The probability of overflows or underflows occurring is at most $\frac{1}{4n_1^2}$, and when they occur, we reconstruct the adapter tree below u using $O(n_1h)$ time.

The additional space spent to store these hash functions is limited: we need to store one h^{ch} , using $O(U^{1/100(1+\alpha)}) = O(N^{1/100})$ bits, and N/n_1 instances of h_u^{id} and $h_u^{(2)}$, using $O(\frac{N}{n_1} \cdot (I_1 + \log Q + 2^{I_1/100})) = O(\frac{N}{n_1} \cdot n_1^{1/25}) = o(R)$ bits. Moreover, through a similar argument to Section 5.5, we can see that the expected amortized time complexity for rehashing is $O(1)$ per operation.

Achieving strong history-independence using fail modes. The above approach completes our dictionary with $R = n/\log^{O(1)} n$ redundancy and $O(\log \log n)$ operational time. It is almost strongly history-independent except that we resample the hash functions when they fail – the current hash functions carry information about the history. Removing the rehashing step will lead to a strongly history-independent dictionary with the same performance.

Our method is to fix the hash functions a priori, and when a hash function fails, we design a “fail mode” for the corresponding bucket, handling all operations slowly but still correctly. Since the probability of enabling the fail mode is extremely low, the expected time consumption of the fail mode is still $o(1)$. When the failure disappears after several subsequent updates, we reconstruct the memory state for the bucket and return to the “normal mode”. Whether we enable the fail mode only depends on the current key set and the fixed (random) hash function, thus the entire data structure with fail modes is strongly history-independent.

Formally, let u be a level-1 bucket, which is responsible for storing a set of $n(u) = \text{poly log } N$ elements from a universe $U_1 = 2^{I_1} \cdot Q$. If h_u^{id} incurs an ID collision when it tries to assign an ID to each key in bucket u , or $h_u^{(2)}$ incurs an overflow or underflow at a level- $8h$ (leaf) bucket, we say u is in the *fail mode*; otherwise, u is in the *normal mode*. Section 4 already shows how to store the set efficiently for the normal mode; in the fail mode, we present the following lemma.

Lemma C.1 (Fail Mode). *Let U' and n be integers where $\log U' \leq n$. There is a data structure storing n elements from the universe $[U']$ using $\log \binom{U'}{n} + O(\log N)$ bits, while supporting all operations in $O(n^{100})$ time.*

Proof Sketch. We list all n -element sets $S = \{a_1, a_2, \dots, a_n\} \subseteq [U']$ ($a_1 < a_2 < \dots < a_n$) in increasing lexicographic order of the sequences (a_1, \dots, a_n) . For any given set S , assuming it occurs as the k -th set in the list, we define its encoding to be the binary representation of k . Taking into account the $O(\log N)$ -bit overhead for storing n itself, the total encoding length is $\log \binom{U'}{n} + O(\log N)$ bits as desired. It is not difficult to prove that both encoding S into k and decoding S from k takes $O(n^{100})$ time, hence all types of operations can be completed in $O(n^{100})$ time as well. \square

We can assume the fail probability of any level-1 bucket is $O(n_1^{-200})$ instead of $O(n_1^{-2})$ as described above by adjusting the parameters in the analysis accordingly. For any given operation that accesses the level-1 bucket u , with probability at most $O(n_1^{-200})$, u will be in the fail mode,

which costs $O(n_1^{100})$ time to perform the operation due to Lemma C.1. The expected time overhead is thus $o(1)$.

We react similarly when the top-level hash function h^{ch} fails (i.e., when any level-1 bucket overflows or underflows): We use the fail-mode structure to store the whole dictionary, which takes no more than $O(N^{100})$ time per operation. Because this failure only happens with probability $O(N^{-200})$, the expected overhead is negligible.