# NEARLY OPTIMAL STATIC LAS VEGAS SUCCINCT DICTIONARY*

HUACHENG YU†

**Abstract.** For positive integers $U$, $n$ and $\sigma$, given a set $S$ of $n$ (distinct) keys from key space $[U]$, each associated with a value from $[\sigma]$, the *static dictionary* problem asks to preprocess these (key, value) pairs into a data structure, supporting value-retrieval queries: for any given $x \in [U]$, $\mathbf{valRet}(x)$ must return the value associated with $x$ if $x \in S$, or return $\perp$ if $x \notin S$. The special case where $\sigma = 1$ is called the *membership* problem. The "textbook" solution is to use a hash table, which occupies linear space and answers each query in constant time. On the other hand, the minimum possible space to encode all (key, value) pairs is only $\mathbf{OPT} := \lceil \lg_2 \binom{U}{n} + n \lg_2 \sigma \rceil$ bits, which could be much smaller than a hash table.

In this paper, we design a randomized dictionary data structure using

$$\mathbf{OPT} + \operatorname{poly} \lg n + O(\lg^{(\ell)} U)$$

bits of space, and it has *expected constant* query time, assuming the query algorithm can access an external lookup table of size $n^\epsilon$ for any constant $\ell$ and $\epsilon$. The lookup table depends only on $U$, $n$ and $\sigma$, and not the input. Previously, even for membership queries and $U \le n^{O(1)}$, the best known data structure with constant query time requires $\mathbf{OPT} + n/\operatorname{poly} \lg n$ bits of space (Pagh [21] and Pătraşcu [23]); the best known using $\mathbf{OPT} + n^{1-\epsilon}$ space has query time $O(\lg n)$. Our new data structure answers open questions by Pătraşcu and Thorup [23, 30].

We also present a scheme that compresses a sequence $X \in [\sigma]^n$ to its zeroth order (empirical) entropy up to $\sigma \cdot \operatorname{poly} \lg n$ extra bits, supporting decoding each $X_i$ in $O(\lg \sigma)$ expected time.

**Key words.** succinct data structure, dictionary, perfect hashing

**AMS subject classifications.** 68P05

*Preliminary version appeared in STOC'20.

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

**1. Introduction.** Given $n$ (key, value) pairs $\{(x_i, v_i)\}_{i=1,\ldots,n}$ with distinct keys $x_i \in [U]$ and (possibly duplicated) values $v_i \in [\sigma]$,[1] the *static dictionary* problem asks to preprocess them into a data structure, supporting value-retrieval queries

- $\mathbf{valRet}(x)$: return $v_i$ if $x = x_i$, and return $\perp$ if $x \neq x_1, \ldots, x_n$.

When $\sigma = 1$, it is called the *membership* problem, i.e., preprocessing a set $S$ of $n$ keys into a data structure, supporting queries of form "is $x \in S$?"[2]

Dictionaries are very fundamental data structures, which have been extensively studied in theory [4, 5, 6, 8, 9, 10, 11, 18, 19, 21, 22, 29, 33]. They are also one of the most basic data structures in practice, included in standard libraries for most of the popular programming languages.[3]

One "textbook" implementation of a dictionary is to store a hash table: use a hash function to map all keys to $m = O(n)$ buckets, and store each (key, value) pair in the corresponding bucket. Simple hash functions (e.g. $(kx \bmod p) \bmod m$ for prime $p \geq U$ and random $k \in \{1, \ldots, p-1\}$) have low collision probabilities, and resolving collisions by chaining leads to a dictionary data structure with *expected* constant query time. Using perfect hashing (e.g. [11]), one can further improve the query time to *worst-case* constant. However, such data structures use at least $n \lg U + n \lg \sigma$ bits of space, even just to write down all (key, value) pairs in the buckets, while the information theoretical space lower bound for this problem is only[4]

$$\mathbf{OPT} := \lg \binom{U}{n} + n \lg \sigma$$

bits, which is about $n \lg n$ bits less than $n \lg U + n \lg \sigma$ (note that $\lg \binom{U}{n} = n \lg(U/n) + O(n)$).

It turns out that it is possible to not *explicitly* store all pairs, and beat $n \lg U + n \lg \sigma$ bits. For *membership* queries ($\sigma = 1$), the previously best known data structure by Pagh [21], as later improved by Pătraşcu [23], uses $\mathbf{OPT} + O(n/\mathrm{poly} \lg n + \lg \lg U)$ bits of space, and answers queries in constant time. Such data structures that use $\mathbf{OPT} + o(\mathbf{OPT})$ bits of space are called the *succinct data structures* [15]; the extra space is called the *redundancy*. This data structure also gives a smooth tradeoff between time and space: for query time $O(t)$, it uses space

$$\mathbf{OPT} + n/r + O(\lg \lg U),$$

where $r = (\frac{\lg n}{t})^{\Omega(t)}$. To achieve this query time, it is assumed that the query algorithm has access to an external lookup table of size $\min\{n^3, r^6\}$, which depends only on $U$ and $n$, and not the input. In particular, when $U = \mathrm{poly}\ n$, if the redundancy is $n^{0.99}$, the query time becomes $O(\lg n)$. If we want the space to be very close to $\mathbf{OPT}$, the query time is $O(\lg n)$, but the lookup table size becomes about $n^3$ (it may even be larger than the data structure itself). For $\sigma > 1$, only $(\mathbf{OPT} + O(n + \lg \lg U))$-bit data structures were known [21]. While these data structures have deterministic query algorithms (and worst-case query-time guarantee), no better zero-error randomized data structure was known.[5]

**1.1. Our contributions.** In this paper, we show that if we allow randomization, a much smaller redundancy and optimal time can be achieved simultaneously. We design a dictionary data structure with $\mathrm{poly} \lg n + O(\lg \lg U)$ bits of redundancy and expected constant query

[1] $[U]$ denotes the set $\{0, \ldots, U-1\}$.

[2] Note that in some literature, all elements in the key space are called the "keys". In this paper, we only call elements in the input set $S$ "keys", and all other elements in the key space are called the "non-keys".

[3] These dictionary data structures usually also support insertion and deletion of key-value pairs.

[4] Throughout the paper, lg is the binary logarithm.

[5] Monte Carlo algorithms, where the query is allowed to err with a small probability, would have a different optimal space bound. Thus, they are not the focus of this paper.

time, making a step towards the optimal static dictionary. The query algorithm only needs to access the data structure and a small lookup table of size $n^\epsilon$.

THEOREM 1.1. *For any constant $\epsilon > 0$, there is a randomized algorithm that can preprocess $n$ (key, value) pairs into a data structure with*

$$\mathbf{OPT} + \operatorname{poly}\lg n + O(\lg\lg U)$$

*bits, and a lookup table of size $n^\epsilon$ that only depends on $U$, $n$ and $\sigma$, such that for any given query $x \in [U]$, a query algorithm can output $\mathbf{valRet}(x)$ in expected constant time on a random access machine with word size $w \geq c \cdot (\lg U + \lg \sigma)$ for some large constant $c$, by accessing the data structure and the lookup table.*

In fact, the $\lg\lg U$ term can be improved to $\lg\lg\cdots\lg U$ for logarithm iterated any constant number of times. Hence, when $U$ is at most $2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$ with $O(1)$ levels, this term can be removed. In this case, among the $\mathbf{OPT} + \operatorname{poly}\lg n$ bits of the data structure, the first $\operatorname{poly}\lg n$ are the (plain) random bits used by the preprocessing algorithm, and the "actual" data structure only occupies the next (and last) $\mathbf{OPT} + 1$ bits. The expectation of the query time is taken over these random bits, which we assume the worst-case input data and query do not see. Moreover, the query time is $O(1)$ with probability $1 - o(1)$, and is $\operatorname{poly}\lg n$ in worst case.

By storing the lookup table as part of the data structure, Theorem 1.1 implies a data structure with $\mathbf{OPT} + n^\epsilon + O(\lg\lg U)$ bits and expected constant query time, which is still an improvement over the previous best known. In the *cell-probe* model, where we only count how many times the query algorithm accesses the memory and the computation is free, the lookup table is not necessary, because it does not depend on the input and can be computed without accessing the data structure.

In the theorem, we assumed that each (key, value) pair fits in $O(1)$ words, which is necessary to obtain constant query time on random access machines. We will discuss larger keys or values in Section 10.

*Perfect hashing.* In general, a perfect hashing maps $n$ input keys to distinct buckets, and it is called *minimal* if it maps them to exactly $n$ distinct buckets, labeled from 0 to $n-1$. En route to the new dictionary data structure, the key component is a succinct membership data structure, equipped with a *two-sided* minimal perfect hashing, defined as follows.

DEFINITION 1.2. *Given a set $S \subseteq [U]$ of size $n$, a* two-sided minimal perfect hash function with membership *(2-PHM) for $S$ is an* bijection *between $[U]$ and $(\{0\} \times [n]) \cup (\{1\} \times [U-n])$ such that it maps $S$ to $\{0\} \times [n]$. Hence, when restricted to $S$, it can be viewed as a bijection $h$ between $S$ and $[n]$; when restricted to $\overline{S}$, it can be viewed as a bijection $\overline{h}$ between $\overline{S}$ and $[U-n]$.*

That is, we want to perfectly hash all keys, as well as all non-keys.

THEOREM 1.3 (informal). *For any constant $\epsilon > 0$, there is a randomized algorithm that preprocesses a set $S \subseteq [U]$ of size $n$ into a data structure with*

$$\lg \binom{U}{n} + \operatorname{poly}\lg n + O(\lg\lg U)$$

*bits, and a lookup table of size $n^\epsilon$ that depends only on $n$ and $U$. Moreover, the data structure determines $2\text{-hq} : [U] \to (\{0\} \times [n]) \cup (\{1\} \times [U-n])$, a 2-PHM for $S$, such that for any $x$, the query algorithm outputs $2\text{-hq}(x)$ in expected constant time on a random access machine with word size $w \geq c \cdot \lg U$ for some sufficiently large constant $c$, by accessing the data structure and the lookup table.*

See Section 9 for the formal statement. Note that the optimal space bound for 2-PHM is the same as storing the set $S$. This is because if we have stored the set $S$, then one trivial way to define a 2-PHM is to map the $i$-th largest element in $S$ to $(0, i)$, and map the $i$-th largest element in $[U] \setminus S$ to $(1, i)$.

*Locally decodable arithmetic codes.* We also show that the above perfect hashing data structure can be applied to obtain a version of locally decodable arithmetic codes with a better space [23]. This problem asks to compress a sequence $X = (x_1, \ldots, x_n) \in \Sigma^n$ for some (small) alphabet set $\Sigma$, such that each symbol $x_i$ can be recovered efficiently from the compression. We should think of a sequence $X$ that is sampled from some low entropy source, and the encoding should take much less than $n \lg |\Sigma|$ bits. The size of such a data structure should match the zeroth order entropy of $X$, i.e., if each symbol in the sequence has entropy $H$ (marginally), then the encoding has length $\sim n \cdot H$. Pătraşcu [23] gave a data structure whose size is

$$\sum_{a \in \Sigma} f_a \lg \frac{n}{f_a} + O(|\Sigma| \lg n) + n/\left(\frac{\lg n}{t}\right)^t + \tilde{O}(n^{3/4}),$$

where $f_a$ is the number of occurrences of $a$. It supports single-element access in $O(t)$ time on a word RAM. Note that when each symbol $x_i$ is sampled independently from a source of entropy $H$, then the empirical entropy $\sum_{a \in \Sigma} f_a \lg \frac{n}{f_a}$ is approximately $n \cdot H$ with high probability.

THEOREM 1.4. *For any constant $\epsilon > 0$, there is a randomized algorithm that preprocesses a sequence $(x_1, \ldots, x_n) \in \Sigma^n$ into a data structure with*

$$\lg \left(\frac{n!}{f_{a_1}! f_{a_2}! \cdots}\right) + |\Sigma| \cdot \operatorname{poly} \lg n$$

*bits, where $f_a$ is the number of occurrences of $a$. For any index $i$, the query algorithm recovers $x_i$ in $O(\lg |\Sigma|)$ time in expectation on a word RAM with word-size $w \geq c \cdot \lg n$ for some large constant $c$, assuming it has access to an external lookup table of size $n^\epsilon$.*

Note that the first term in the space is the minimum possible space to store a sequence with frequencies $(f_a)_{a \in \Sigma}$, which is at most $\sum_{a \in \Sigma} f_a \lg \frac{n}{f_a}$.

**1.2. Related work.** The perfect hashing scheme [11] by Fredman, Komlós and Szemerédi maps $[U]$ to $[n + o(n)]$ such that for any given set $S$ of size $n$, there is a hash function $h$ that maps all elements in $S$ to different buckets (i.e., no hash collision) such that $h(x)$ can be evaluated in constant time. This hash function takes $O(n\sqrt{\lg n} + \lg \lg U)$ bits to store. It was later improved to $O(n + \lg \lg U)$ bits by Schmidt and Siegel [28], and to $n \lg e + \lg \lg U + o(n + \lg \lg U)$ bits by Hagerup and Tholey [14]. By storing the (key, value) pair in the corresponding bucket, the perfect hashing scheme solves the dictionary problem with $O(n)$ *words* of space and constant query time. Fiat, Naor, Schmidt and Siegel [9] showed that only $O(\lg n + \lg \lg U)$ extra bits are needed to store *both* the hashing function and the table, obtaining space of $n\lceil \lg U \rceil + n\lceil \lg \sigma \rceil + O(\lg n + \lg \lg U)$. Fiat and Naor [8] further removed the $O(\lg n)$ term, as well as the $O(\lg \lg U)$ term when $U$ is not too large.

The first dictionary data structure that achieves nearly optimal space is due to Brodnik and Munro [4]. Their data structure uses $\mathbf{OPT} + O(\mathbf{OPT}/\lg \lg \lg U)$ bits, and it has constant query time. Pagh [21] reduced the dictionary problem to the *rank* problem (see below, also Section 5.1 for definition of the rank problem), and designed a data structure for membership queries using $\mathbf{OPT} + O(n \lg^2 \lg n / \lg n + \lg \lg U)$ bits for $n < U/\lg \lg U$, and $\mathbf{OPT} + O(U \lg \lg U / \lg U)$ for $n \geq U/\lg \lg U$. Pagh's dictionary uses rank data structures

as subroutines. By improving the rank data structures, Pătraşcu [23] improved the bound to $\mathbf{OPT} + n/\mathrm{poly}\lg n + O(\lg\lg U)$, as we mentioned earlier.

It is worth mentioning that when $n = U$, i.e., when the input is a sequence of values $v_1, \ldots, v_U \in [\sigma]$, Dodis, Pătraşcu and Thorup [7] designed a data structure using optimal space. Their data structure uses a lookup table of $\mathrm{poly}\lg n$ size. We make this lookup table more explicit (see Lemma 5.1 in Section 8) as an application of our new technique.

No non-trivial lower bounds on the query time are known without restrictions on the data structure or model. Fich and Miltersen [10] and Miltersen [18] proved $\Omega(\lg n)$ and $\Omega(\lg\lg n)$ lower bounds in the RAM model with restricted operations. Buhrman, Miltersen, Radhakrishnan and Venkatesh [5] proved that in the *bit-probe* model (where the word size $w$ is 1), any data structure using $O(\mathbf{OPT})$ space must have query time at least $O(\lg \frac{U}{n})$. Viola [31] proved a lower bound for the case where $U = 3n$, that any bit-probe data structure with query time $q$ must use space $\mathbf{OPT} + n/2^{O(q)} - \log n$.

Raman, Raman and Rao [27] considered the *indexable dictionary* problem, which generalizes membership. Given a set $S$ of $n$ keys, it supports rank and select queries: $\texttt{rank}_S(x)$ returns $\perp$ if $x \notin S$, and returns $i$ if $x$ is the $i$-th smallest in $S$; $\texttt{select}_S(i)$ returns the $i$-th smallest element in $S$. They obtained a data structure using $\mathbf{OPT} + o(n) + O(\lg\lg U)$ bits and constant query time. Grossi, Orlandi, Raman and Rao [13] studied the *fully indexable dictionary* problem. It generalizes the indexable dictionary problem to let $\texttt{rank}_S(x)$ return the number of elements in $S$ that are at most $x$ (also for $x \notin S$). They obtained a data structure using space $\mathbf{OPT} + O\left(n^{1+\delta} + U^{\epsilon} \cdot n^{1-s\epsilon}\right)$ with query time $O(s/\delta + 1/\epsilon)$. In fact, this problem is much harder. It was observed in [26] that `rank` queries can be reduced from *colored predecessor search*, which has a query time lower bound of $\Omega(\lg\lg n)$ even when the space is $O(n \lg U)$ [24, 25] (not to say the succinct regime). When $U > n^2$, the problem requires $n^{1+\epsilon}$ space to get constant query time (when the word-size is $\lg n$), even only supporting `rank` queries.

Makhdoumi, Huang, Médard, Polyanskiy [17] studied locally decodable source coding. They consider $X$ that consists of i.i.d samples from a source of entropy $H$, and would like to encode $X$ such that each $X_i$ can be recovered efficiently. However, the main focus is non-adaptive bit-probe query algorithms. That is, the query algorithm has to decide which $t$ *bits* of the encoding to access based only on the queried index $i$. They also studied the lossy case, where the encoding is equipped with the error correcting ability.

**1.3. Technical contributions.** We make two technical contributions to succinct data structures: We summarize the "spillover representation", introduced by Pătraşcu [23], to define binary strings with *fractional lengths* and build a "toolkit" of black-box operations; we study the "opposite" of data structures, called the *data interpretation*. We believe they will have more applications to other problems in succinct data structures.

**1.3.1. Strings with fractional lengths.** A data structure is simply a bit string, and its length (or size) is the number of bits. Under standard notions, an $s$-bit string is only well-defined for integer $s$. Here, we show how to define strings when $s$ is non-integer.[6] We will see why this notion is useful later (or see [23]).

Let $(M, K)$ be a pair such that $M \in \{0,1\}^m$ is a bit string, and $K \in [R]$ is an integer. These pairs are viewed as "binary strings" of length $s = m + \lg_2 R$. When $R$ is a power of two, this matches the standard notion of length, as we could simply write $K$ in its binary

[6] The terminology "fractional" may seem to imply that $s$ has to be expressible as fractions, i.e., rational numbers. However, this is not the case.

representation using $\lg R$ bits and append it to $M$. As we increase $R$, such a pair could potentially represent more information. Only when $R$ is increased by a factor of two, does the pair correspond to a string with one more bit. That is, by restricting $R \in [2^\kappa, 2^{\kappa+1})$ for some fixed parameter $\kappa$, we essentially "insert" $2^\kappa - 1$ valid lengths between adjacent integers. It makes the measure of space more fine-grained. In this paper, $R$ is always set to $2^{\Theta(w)}$, i.e. $\kappa = \Theta(w)$ (recall that $w$ is the word size). Thus, $K$ is an $O(w)$-bit integer, and the algorithms are able to do arithmetic operations on $K$ in constant time.

We summarize a few black-box operations on fractional-length strings. The two major ones are *concatenation* and *fusion*.

*Concatenation.* Given $B$ (fractional-length) strings $\mathcal{S}_1, \ldots, \mathcal{S}_B$ of lengths $s_1, \ldots, s_B$, we show that they can be "concatenated" into one string of length $s \approx s_1 + \cdots + s_B$. Moreover, we prove that given access to $\mathcal{S}$, each $\mathcal{S}_i$ can be decoded efficiently. We emphasize that decoding an $\mathcal{S}_i = (M_i, K_i)$ *does not* mean reconstructing the entire string. Instead, the decoding algorithm only recovers $K_i$, and *finds where* $M_i$ is located within the long string (where $M_i$ is guaranteed to be a consecutive substring). Thus, the decoding algorithm can be *very efficient*. Nevertheless, after decoding, $\mathcal{S}_i$ can still be accessed as if it was stored independently.

Concatenation is useful when the data structure has multiple parts. We could simply construct each part separately and then concatenate them. It also demonstrates, to some extent, why fractional lengths are useful and necessary. If we only use integral-length strings, then each $\mathcal{S}_i$ will have length (at least) $\lceil s_i \rceil$. The length of the concatenated string becomes $\lceil s_1 \rceil + \cdots + \lceil s_B \rceil$, which could be $B - 1$ bits longer than $\lceil s_1 + \cdots + s_B \rceil$.

*Fusion.* The other major operation is to *fuse* an integer into a string. That is, we first fix lengths $s_1, \ldots, s_C$. Then the fusion operation maps a pair $(i, \mathcal{S}_i)$, for $i \in [C]$ and $\mathcal{S}_i$ of length $s_i$, to a single fractional-length string $\mathcal{S}$. We show that $\mathcal{S}$ can have length $s \approx \lg(2^{s_1} + \cdots + 2^{s_C})$. This is the best possible length, since there are $2^{s_1} + \cdots + 2^{s_C}$ different pairs $(i, \mathcal{S}_i)$ in total. Furthermore, we show that given access to $\mathcal{S}$, we can efficiently decode the original input pair $(i, \mathcal{S}_i)$.

The fusion operation is useful when we study different cases of the input, and construct a data structure for each case separately. Suppose we can partition the set of all possible inputs $\mathcal{I}$ into $C$ subsets $\mathcal{I}_1, \ldots, \mathcal{I}_C$, such that we can apply a possibly different construction for each $\mathcal{I}_i$ that produces a data structure of size $\approx \lg |\mathcal{I}_i|$. Then by fusing the index of the subset that contains the input into the data structure, we obtain a data structure for all inputs using space $\approx \lg |\mathcal{I}|$.

By including a few other operations, we build a "toolkit" for operating on fractional-length strings. The view of fractional-length strings makes the "spillover representation" of Pătraşcu [23] more explicit. The original data structure of [23] needs huge lookup tables to store truth tables for $O(w)$-bit word operations. The new view assigns semantic meanings to those operations, so that a major part can be efficiently computed *without* lookup tables. This is the main reason why we can reduce the lookup table size.

**1.3.2. Data interpretation.** For a data structure problem, we preprocess a combinatorial object into a binary string. Then this string is stored in memory, which is divided into $w$-bit words. In each time step, a query algorithm may access a memory word (i.e. a $w$-bit substring), or do local computation. Finally, it computes some function of the input object. The concept of *data interpretation* is to perform the above procedure in the opposite direction. Given a binary string, we preprocess it into a combinatorial object. In each time step, a query algorithm may query an oracle for some function of the object, or do local computation. Finally, it reconstructs a $w$-bit substring of the input string.

It might not be obvious at this moment why it is beneficial to convert a string (data structure) back to a set, but it turns out to be a key subroutine in our data structure. Since this paper concerns data structures with space almost matching the information theoretical lower bound, we will also make data interpretation space-efficient. We design a data interpretation algorithm which preprocesses an input string of (fractional) length $\approx \lg \binom{V}{m}$ into a set $S \subseteq [V]$ of size $m$, such that assuming there is a `rank` oracle for $S$ ($\texttt{rank}_S(x)$ returns the number of elements in $S$ that are at most $x$), any designated $w$ consecutive bits of the input string can be reconstructed in $\operatorname{poly}\lg V$ time (see Section 8.2).

**2. Overview.** In this section, we give an overview over our new data structure. For simplicity, we will focus only on the membership queries, and assume $U = \operatorname{poly} n$, and there is no divisibility problem. These assumptions will be removed in the later sections. In this case, all previous solutions use hash functions in their main construction, to map the keys into buckets. Our data structure is conceptually different: Instead of random hash functions, we consider *random inputs*. While our data structure works for worst-case inputs, let us first think of the input set being $n$ uniformly random (distinct) keys. Then with high probability, the input already has the properties we wanted from a random hash function, e.g., by dividing the key space into buckets in some fixed way, we have the sizes of buckets roughly balanced, etc. We first construct a data structure just for those "random-looking" inputs. On the other hand, with low probability, the input may look "non-typical," e.g., some bucket may have size much larger than average. However, "with low probability" means that only a small fraction of all possible inputs have these non-typical features. Suppose the total number of such inputs is, say $\frac{1}{n^2} \cdot 2^{\mathbf{OPT}}$, then only $\mathbf{OPT} - 2\lg n$ bits are needed for the optimal encoding. This suggests that we can afford to spend more extra bits on them. Suppose we use $\mathbf{OPT} - \lg n$ bits ($\lg n$ extra bits) to encode these non-typical inputs, it is still negligible overall — among all $O(2^{\mathbf{OPT}})$ possible data structures (memory states), such an encoding only wastes $\approx 2^{\mathbf{OPT}} \cdot \frac{1}{n}$ memory states. Another useful way to view it is that if we use $r$ extra bits for such rare cases, then those $r$ bits "start" at the $(\mathbf{OPT} - 2\lg n)$-th bit, rather than the $\mathbf{OPT}$-th bit. The more non-typical the input is, the more extra bits we can afford to spend. Finally, we will use the fusion operation to fuse all cases together. Similar strategies for constructing succinct data structures, where we consider random inputs and/or non-typical inputs, have been used in [3, 32, 34]. In the following, we give an overview for the "random-looking" case, the formal argument can be found in Section 7.3. Then in Section 7.4, we show that we can afford to apply known constructions with larger redundancy on the "non-typical parts" of the "non-typical" sets, according to the above argument.

**2.1. Random inputs.** We partition the universe into $n/\lg^4 n$ blocks of size $V$. Then for a "random-looking" input set $S$, *every* block contains $\lg^4 n \pm \lg^3 n$ keys.[7] As we will see in Section 8.1, for $\operatorname{poly}\lg n = \operatorname{poly} w$ keys, we can construct a rank data structure with only $O(1/U)$ extra bits, such that given the number of keys, a query algorithm answers `rank` queries in constant time. In particular, it supports membership queries (e.g., by asking $\texttt{rank}_S(x)$ and $\texttt{rank}_S(x-1)$). The high-level idea is to construct a rank data structure for each block, then *concatenate* them. In order to answer a query in block $i$, we need to

- recover the number of keys in block $i$ (as the rank data structure assumes this number is known), and
- approximate the total length of data structures for the first $i-1$ blocks (to decode the $i$-th data structure).

That is, besides the $n/\lg^4 n$ rank data structures, we need to store their lengths such that any prefix sum can be approximated. Unfortunately, any data structure supporting prefix

[7] $a \pm b$ denotes a number in the range $[a-b, a+b]$.

sum queries cannot simultaneously have "low" query time and "small" space, due to a lower bound of Pătraşcu and Viola [26]. The underlying issue in this approach is that the data structure for each block has a variable length (the length depends on the number of keys in the block, which varies based on the input). In order to locate the $i$-th data structure from the concatenated string, computing a prefix sum on a sequence of variables seems inevitable. The Pătraşcu-Viola lower bound even prevents us from supporting prefix sums *implicitly*. That is, not only separately storing a prefix sum data structure for the lengths requires "high" query time or "large" space, there is also no "clever" way to jointly store the lengths together with the data structures for the blocks. Hence, this "variable-length encoding" issue is the primary problem we need to tackle for "random-looking" inputs.

To this end, observe that although the number of keys in each block is not fixed, its deviation is actually small compared to the number, i.e., the number of keys cannot be too different for different inputs. Then the main idea is to construct *two* data structures for each block, consisting of

- *a main data structure*, which stores "most of the information" about the block, and importantly, has a fixed length (independent of the number of keys), and
- *an auxiliary data structure*, which stores all "remaining information" about the block (and unavoidably has variable length).

Furthermore, we wish that with high probability, a given query can be answered by only accessing the main data structure (in constant time) *without knowing the number of keys*. If this is possible, then to construct the final data structure, we

- concatenate all main data structures,
- concatenate the auxiliary data structures, and store them together with a prefix sum structure,
- finally concatenate the two.

Now, since all main data structures have fixed lengths, each one can be decoded in constant time without a prefix sum structure (the total length of the first $i-1$ data structures is simply $i-1$ times the length of a single one). Then to answer a query in block $i$, we first decode the $i$-th main data structure, and query it in constant time. With high probability, the answer to the query is already found, and we are done. Otherwise, we decode the $i$-th auxiliary data structure by querying the prefix sum structure, and query the data structures to find the answer. This may take a longer time, but if the probability that we have to decode the auxiliary data structure is sufficiently low, then the expected query time is still constant.

Next, we describe an approach to construct such two data structures for a block, which uses more space than what we aim for, but exhibits the main idea. For each block of size $V$, we pick $\lg^4 n - \lg^3 n$ *random* keys in the block to store in the main data structure. We also pick $V - (\lg^4 n + \lg^3 n)$ *random non-keys* (i.e. the elements in the key space but not in the input set), and store them in the main data structure. This is always possible because there are at least $\lg^4 n - \lg^3 n$ and at most $\lg^4 n + \lg^3 n$ keys in each block for "random-looking" inputs. Hence, only $2\lg^3 n$ elements are "unknown" from the main data structure. Then we show that such a separation of the block into $\lg^4 n - \lg^3 n$ keys, $V - (\lg^4 n + \lg^3 n)$ non-keys and $2\lg^3 n$ unknowns can be jointly stored using the near-optimal $\approx \lg \binom{V}{\lg^4 n - \lg^3 n, 2\lg^3 n}$ bits (this is an easy application of the rank data structures).[8] Its size is independent of the actual input. Then in the auxiliary data structure, we store the remaining information about the block, i.e., among the unknowns, which ones are the keys. For a block with $m$ keys, it takes $\approx \lg \binom{2\lg^3 n}{m - (\lg^4 n - \lg^3 n)}$ bits. Then for each query, the answer can be found in the main data structure with probability at least $1 - O(1/\lg n)$. Only when the main data structure

[8] $\binom{n}{k_1, k_2} = n!/(k_1! k_2! (n - k_1 - k_2)!)$.

returns "unknown" does the query algorithm need to access the auxiliary data structure.

The above construction has all the desired properties, except that it uses too much space. The inherent reason is that it implicitly stores the *randomness* used in deciding which keys and non-keys to store in the main data structure. If we sum up the sizes of the main and auxiliary data structures,

$$\begin{aligned}&\lg\binom{V}{\lg^4 n-\lg^3 n, 2\lg^3 n}+\lg\binom{2\lg^3 n}{m-(\lg^4 n-\lg^3 n)}\\ =\ &\lg\binom{V}{m}+\lg\binom{m}{\lg^4 n-\lg^3 n}+\lg\binom{V-m}{V-(\lg^4 n+\lg^3 n)}.\end{aligned}$$

Unsurprisingly, the number of extra bits $\lg\binom{m}{\lg^4 n-\lg^3 n}+\lg\binom{V-m}{V-(\lg^4 n+\lg^3 n)}$ is exactly how much is needed to decide which keys and non-keys to store in the main data structure. These "random bits" are not part of the input, and implicitly storing them causes a large amount of redundancy.

However, when the inputs are uniform, we do not really need any external randomness to decide the two subsets, since the entire data structure is close to a random string. This suggests that for each block, we should treat the data structure from other part of the inputs as the "randomness". That is, we turn the necessity of implicitly storing the random bits into an opportunity to store other information. This is where we use *data interpretation*. We convert existing data structures back to subsets of certain sizes, which correspond to the keys and non-keys in the main data structure (note that the main purpose of using data interpretation is to save space). We present more details in the next subsection.

**2.2. Using data interpretation.** To implement this idea, we will have to slightly modify the construction. Now, the universe is partitioned into *pairs of blocks*. Each pair consists of a *primary* block and a *secondary* block, such that for a "random-looking" input, the primary block contains $\lg^{2c} n \pm \lg^{c+1} n$ keys, and the secondary block contains $\Theta(\lg^{c+1} n)$ keys, for some constant $c$, where the secondary block plays the role of the "randomness".[9] Fix a block pair, let $V$ be the size of the primary block, $m$ be the number of keys in the primary block, $V_{\text{sec}}$ be the size of the secondary block, and $m_{\text{sec}}$ be the number of keys in the secondary block. The goal is to construct two data structures using $\approx \lg\binom{V}{m}+\lg\binom{V_{\text{sec}}}{m_{\text{sec}}}$ bits in total.

We first construct a rank data structure for the secondary block using $\approx \lg\binom{V_{\text{sec}}}{m_{\text{sec}}}$ bits. We then *split* this data structure into three substrings of lengths approximately $\lg\binom{m}{\lg^{2c} n-\lg^{c+1} n}$, $\lg\binom{V-m}{V-(\lg^{2c} n+\lg^{c+1} n)}$ and the remaining bits (we show that such a split can also be done for fractional-length strings). Then $m_{\text{sec}} = \Theta(\lg^{c+1} n)$ guarantees that there are enough bits and such split is possible. Next, we apply a data interpretation algorithm to interpret the first string of length $\lg\binom{m}{\lg^{2c} n-\lg^{c+1} n}$ as a set of size $\lg^{2c} n - \lg^{c+1} n$ over a universe of size $m$, indicating which of the $m$ keys in the primary block should be stored in the main data structure. We also interpret the second string as a subset indicating which of the $V - (\lg^{2c} n + \lg^{c+1} n)$ non-keys should be stored in the main data structure. Moreover, we show that the data interpretation algorithm guarantees that any consecutive $w$ bits of the original string can be recovered in $\lg^{O(1)} n$ time, assuming there is a `rank` oracle of the set generated from the interpretation. Therefore, there is no need to store the first two strings, as they can be implicitly accessed efficiently.

The main data structure is the same as what we stated in the previous subsection: storing $\lg^{2c} n-\lg^{c+1} n$ keys, $V-(\lg^{2c} n+\lg^{c+1} n)$ non-keys and $2\lg^{c+1} n$ "unknowns", supporting `rank` queries in constant time. The auxiliary data structure now consists of two parts:

[9] The parameters used here are slightly different from the formal proof for simplicity of notations.

- among the $2\lg^{c+1} n$ unknowns, which $m - (\lg^{2c} n - \lg^{c+1} n)$ are keys, and
- the third substring from above.

One may verify that the sizes of the two data structures is what we claimed. This leads to our main technical lemma.

LEMMA 2.1 (main technical lemma, informal). *For $V \le \operatorname{poly} n$, given $S \subseteq [V]$ of size $m$ and $S_{\text{sec}} \subseteq [V_{\text{sec}}]$ of size $m_{\text{sec}}$, we can construct a main data structure $\mathcal{D}_{\text{main}}$ of size*

$$\approx \lg \binom{V}{\lg^{2c} n - \lg^{c+1} n, 2\lg^{c+1} n}$$

*and an auxiliary data structure $\mathcal{D}_{\text{aux}}$ of size*

$$\approx \lg \binom{V}{m} + \lg \binom{V_{\text{sec}}}{m_{\text{sec}}} - \lg \binom{V}{\lg^{2c} n - \lg^{c+1} n, 2\lg^{c+1} n},$$

*such that*

- *any given query "$x \stackrel{?}{\in} S$" can be answered in constant time by accessing only $\mathcal{D}_{\text{main}}$ with probability $1 - O(\lg^{-c+1} n)$;*
- *any given query "$x \stackrel{?}{\in} S$" or "$x \stackrel{?}{\in} S_{\text{sec}}$" can be answered in $\operatorname{poly}\lg n$ time by accessing both $\mathcal{D}_{\text{main}}$ and $\mathcal{D}_{\text{aux}}$ in worst case.*

See Lemma 7.1 for the formal statement. Then, the final data structure will be the concatenation of all main and auxiliary data structures, in a similar way to what we stated in the previous subsection. The auxiliary data structure needs to be decoded only when the main data structure returns "unknown" or the query lands in a secondary block. By randomly shifting the universe, we bound the probability of needing the auxiliary data structure by $O(\lg^{-c+1} n)$. By setting $c$ to be a sufficiently large constant, the expected query time is constant.

**2.3. Organization.** In Section 3, we define notations and the model of computation. In Section 4, we formally define fractional-length strings, and present the operations on them. In Section 5, we present applications of fractional-length strings. In Section 6, we show how to construct the succinct dictionary and locally decodable arithmetic codes using 2-PHM. In Section 7, we design the data structure for the case where $U = \operatorname{poly} n$ using the main technical lemma. Then we prove the main technical lemma in Section 8, and generalize to all $n$ and $U$ in Section 9.

## 3. Preliminaries and Notation.

**3.1. Random access machine.** A random access machine (RAM) [12] has a memory divided into $w$-bit *words*, where $w$ is called the *word-size*. Typically, we assume the number of words in the memory is at most $2^w$, and that the words are indexed by the elements of $[2^w]$. In each time step, an algorithm may load one memory word to one of its $O(1)$ CPU registers, write the content of a CPU register to one memory word, or compute (limited) word operations on the CPU registers.

The standard word operations are the four basic arithmetic operations (addition, subtraction, multiplication and division) on $w$-bit integers, bit-wise operators (AND, OR, XOR), and comparison. In this paper, we also assume that the machine supports *floating-point* numbers. We use two registers $a$ and $b$ to represent the number $a \cdot 2^b$ such that $1 \le a < 2$ and $b$ is an integer. The arithmetic operations extend to these numbers as well (possibly with rounding errors). This is without loss of generality, as they can be simulated using the standard operations. Finally, we assume it is possible to compute $2^x$ up to a $1 \pm 2^{-w}$ multiplicative error

for $|x| \le 2^{O(w)}$, and $\lg_2 x$ up to an additive error of $\pm 2^{-w}$ for $|\lg_2 x| \le 2^{O(w)}$. We further assume that the error can be arbitrary but has to be deterministic, i.e., for any given $x$, $2^x$ and $\lg_2 x$ always evaluate to the same result within the desired range. By expanding into the Taylor series, the two can be computed in $O(w)$ time using only arithmetic operations, which is already sufficient for our application.

To compute $2^x$, we first compute $x_0 = \lfloor x \rfloor$ and $x_1 = x - x_0$. Since $x_0$ is an integer, and $1 \le 2^{x_1} < 2$, it suffices to approximate $2^{x_1}$. Since $0 \le x_1 < 1$, by Taylor expansion, we have

$$2^{x_1} = e^{x_1 \ln 2} = \sum_{i \ge 0} \frac{1}{i!} \cdot (x_1 \ln 2)^i,$$

where $\ln 2 < 1$ is a constant that can be stored. Truncating the sum at $i = w$ results in an additive error of at most $O(1/w!)$, and the sum can be computed using $O(w)$ arithmetic operations. The rounding error from the arithmetic operations can be made to at most $2^{-2w}$. Finally, observe that $2^{x_1} \in [1, 2)$, the multiplicative error is at most $1 \pm 2^{-w}$.

To compute $\lg_2 x$, where $x = a \cdot 2^b$, note that $\lg_2 x = b + \lg_2 a$. Letting $t = (a-1)/a$, we have $a = 1/(1-t)$, and $0 \le t < 1/2$. By Taylor expansion, we have

$$\lg_2 a = (\lg e) \cdot \ln(1/(1-t)) = (\lg e) \sum_{i \ge 1} \frac{t^i}{i}.$$

By truncating the sum at $i = w$, the truncation error is at most $O(2^{-w}/w)$, since $t < 1/2$. Similarly, the sum can be computed in $O(w)$ arithmetic operations, and the rounding error from them can be made to at most $2^{-2w}$. Therefore, the total additive error is at most $2^{-w}$.

**3.2. Notation.** In this paper, for integers $X$ and $Y > 0$, let $X \operatorname{div} Y$ denote $\lfloor X/Y \rfloor$, $X \bmod Y$ denote $X - Y \cdot (X \operatorname{div} Y)$, and let $[Y]$ denote the set $\{0, 1, \ldots, Y-1\}$. For integer $a$ and set $S$, let $a + S$ denote the set $\{a + x : x \in S\}$. Let $\operatorname{frac}(\alpha)$ denote $\alpha - \lfloor \alpha \rfloor$. Throughout the paper, $\lg x$ is the binary logarithm $\lg_2 x$, $\tilde{O}(f) = f \cdot \operatorname{poly} \lg f$.[10] $a \pm b$ denotes a number in the range $[a-b, a+b]$. Let $\binom{n}{k_1, k_2} := n!/(k_1! k_2! (n - k_1 - k_2)!)$.

Let

$$\mathbf{OPT}_{V,m} := \lg \binom{V}{m}$$

be the information theoretical optimal space when storing a set of $m$ keys over key space of size $V$.

**3.3. Useful equations and inequalities.** Stirling's formula states that

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + O(1/n)).$$

Let $P, Q$ be two distributions over $\mathcal{U}$, then Pinsker's inequality states that

$$\|P - Q\|_1 \le \sqrt{2 D_{\mathrm{KL}}(P \,\|\, Q)},$$

where $\|P - Q\|_1 = \sum_{x \in \mathcal{U}} |P(x) - Q(x)|$ is the total variation distance ($\ell_1$ distance), and $D_{\mathrm{KL}}(P \,\|\, Q) = \sum_{x \in \mathcal{U}} P(x) \lg \frac{P(x)}{Q(x)}$ is the Kullback–Leibler divergence [16] (or the KL-divergence). In particular, when $Q$ is the uniform distribution over $\mathcal{U}$, we have

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_{x \in \mathcal{U}} P(x) \lg P(x) + \lg |\mathcal{U}| = H(Q) - H(P),$$

where $H(P) = \sum_{x \in \mathcal{U}} P(x) \lg \frac{1}{P(x)}$ is the entropy.

[10] Note that in some literature, $\tilde{O}(f)$ hides log factors in any parameter of the problem studied, e.g., $\lg^3 n = \tilde{O}(1)$, which is *not* the case for this paper.

**4. Fractional-length Strings.** In this section, we formally define binary strings with fractional lengths using the spillover representation of [23], and present block-box operations.

**4.1. Definition of fractional-length strings.** Throughout the paper, let $\kappa$ be the *fineness parameter*, which characterizes the gaps between adjacent valid lengths and determines the space loss when doing the operations. It is an integer parameter that is specified by the algorithm designer and will be hardwired into the preprocessing and query algorithms. In the following, we will see that each operation introduces a redundancy of $O(2^{-\kappa})$ bits; and on the other hand, the algorithms will have to perform arithmetic operations on $\kappa$-bit integers. In our data structure construction, $\kappa$ will be set to $\Theta(\lg U) = O(w)$, so that each operation introduces negligible redundancy, and $\kappa$-bit arithmetic operations can still be performed in constant time.

DEFINITION 4.1 (fractional-length strings). *For integers $m, R$ such that either $m = 0$ and $R \in [1, 2^\kappa)$, or $m > 0$ and $R \in [2^\kappa, 2^{\kappa+1})$, $\{0,1\}^m \times [R]$ is the set of all* fractional-length strings *of dimension $(m, R)$, denoted by $\mathbf{FL}(m, R)$.*

*For $\mathcal{S} = (M, K) \in \mathbf{FL}(m, R)$, we say $\mathcal{S}$ is a* binary string of length $s = m + \lg R$*, and*

$$\mathcal{S}[i] := \begin{cases} M[i] & i \in [m], \\ K & i = m. \end{cases}$$

*Let $|\mathcal{S}|$ denote the length of $\mathcal{S}$. Let $\mathcal{S}[i, j]$ denote the sequence (substring) $(\mathcal{S}[i], \ldots, \mathcal{S}[j])$. Let $\mathrm{range}[K] := R$ be the size of the range of $K$.*

*For any real number $s > 0$, $s$ is a* valid fractional length *if there exists such $m$ and $R$ such that $s = m + \lg R$.*

Note that any valid fractional length $s$ uniquely determines $(m, R)$, since when $s < \kappa$, we have

$$m = 0 \quad \text{and} \quad R = 2^s,$$

when $s \geq \kappa$,

$$m = \lfloor s - \kappa \rfloor \quad \text{and} \quad R = 2^{s-m}.$$

We will also use $\mathbf{FL}(s)$ to denote $\mathbf{FL}(m, R)$.

*Remark* 4.2. Note the following facts about fractional-length strings:

- When $s$ is an integer, by writing $K$ in its binary representation, a binary string of length $s$ from Definition 4.1 is a standard binary string of $s$ bits;
- A uniformly random string of length $s$ has entropy exactly $s$;
- The length of a string may be an irrational number, but it can always be efficiently encoded, e.g., by encoding $|M|$ and $\mathrm{range}[K]$;

Since the word size is $\Omega(\kappa)$, any $O(\kappa)$ consecutive bits of a string can be retrieved using $O(1)$ memory accesses, which suggests how a fractional-length string is accessed. Formally, we define an access as follows.

DEFINITION 4.3 (access). *Let $\mathcal{S}$ be a (fractional-length) string, an* access *to $\mathcal{S}$ is to retrieve $\mathcal{S}[i, j]$ for $j - i \leq O(\kappa)$.*

When $j < |M|$, an access is to retrieve $j - i + 1$ bits of $M$. When $j = |M|$, it is to retrieve $j - i$ bits and the integer $K$.

In the following, we show how to operate on fractional-length strings. First, one can always convert a fractional-length string with a shorter length to one with a longer length. Therefore, when we say an algorithm outputs a string of length at most $s$, we can always

assume without loss of generality that its length is exactly the largest valid fractional length that is at most $s$.

DEFINITION 4.4. *An algorithm* decodes *a string $\mathcal{S}_1 = (M_1, K_1)$ from $\mathcal{S}_2 = (M_2, K_2)$, where $M_1$ is a (consecutive) substring of $M_2$, if the algorithm computes the value of $K_1$ and the start location of $M_1$.*

After decoding $\mathcal{S}_1$, any further access can be performed as if $\mathcal{S}_1$ is stored explicitly, by only directly accessing $\mathcal{S}_2$.

PROPOSITION 4.5. *Let $s_1, s_2$ be two valid fractional lengths such that $s_1 \le s_2$. Then given a string $\mathcal{S}_1 = (M_1, K_1)$ of length $s_1$, it can be converted to a string $\mathcal{S}_2 = (M_2, K_2)$ of length $s_2$ such that $M_1$ is a substring of $M_2$. Moreover, given $s_1, s_2$, $\mathcal{S}_1$ can be decoded using $O(1)$ time and one access to $\mathcal{S}_2$.*

*Proof.* Let $s_1 = m_1 + \lg R_1$ and $s_2 = m_2 + \lg R_2$ such that $R_1, R_2 \in [2^\kappa, 2^{\kappa+1})$. If $m_1 = m_2$, we must have $R_1 \le R_2$, by setting $\mathcal{S}_2 := \mathcal{S}_1$, $\mathcal{S}_1$ can be decoded trivially.

If $m_1 < m_2$, then we append the lowest bit of $K_1$ to $M_1$: Let

$$M_2 := M_1 \circ 0^{m_2 - m_1 - 1} \circ (K_1 \bmod 2)$$

and

$$K_2 := K_1 \operatorname{div} 2.$$

Thus, $|M_2| = m_2$, and $K_2 \le R_1/2 \le 2^\kappa \le R_2$. $K_1$ can be decoded by accessing $\mathcal{S}_2[m_2 - 1, m_2]$. ∎

**4.2. Concatenation.** Now we study the *concatenation* of fractional-length strings.

DEFINITION 4.6. *An algorithm* concatenates *$B$ strings $\mathcal{S}_1 = (M_1, K_1) \in \mathbf{FL}(s_1)$, ..., $\mathcal{S}_B = (M_B, K_B) \in \mathbf{FL}(s_B)$ into one single string $\mathcal{S} \in \mathbf{FL}(s)$, if each $M_i$ is a substring of $M$, and each $\mathcal{S}_i$ can be decoded from $\mathcal{S}$.*

We begin by proving there is an algorithm that concatenates two strings.

PROPOSITION 4.7. *Let $s_1, s_2 \ge 0$ be valid fractional lengths. Given two strings $\mathcal{S}_1 \in \mathbf{FL}(s_1)$ and $\mathcal{S}_2 \in \mathbf{FL}(s_2)$, they can be concatenated into one string $\mathcal{S}$ of length at most $s_1 + s_2 + 2^{-\kappa+2}$. Moreover, given the values of $s_1$ and $s_2$, both $\mathcal{S}_1$ and $\mathcal{S}_2$ can be decoded using constant time and one access to $\mathcal{S}$.*

*Proof.* Let $\mathcal{S}_1 = (M_1, K_1)$ and $\mathcal{S}_2 = (M_2, K_2)$. To concatenate two strings, let us first combine $K_1$ and $K_2$ into a single integer $K' \in [\operatorname{range}[K_1] \cdot \operatorname{range}[K_2]]$:

$$K' := K_1 \cdot \operatorname{range}[K_2] + K_2.$$

If $s_1 + s_2 < \kappa + 1$, we simply let $K = K'$, and let $M$ be the empty string. Then $\lg(\operatorname{range}[K]) = s_1 + s_2 < \kappa + 1$, $(M, K)$ is the concatenation.

Next, we assume $s_1 + s_2 \ge \kappa + 1$. In this case, in the final string $\mathcal{S} = (M, K)$, $M$ will be the concatenation of $M_1$, $M_2$ and the lowest bits of $K'$. More specifically, let $|M|$ be $\lfloor s_1 + s_2 + 2^{-\kappa+2} \rfloor - \kappa$, and let $\operatorname{range}[K]$ be $\lfloor 2^{\kappa + \operatorname{frac}(s_1+s_2+2^{-\kappa+2})} \rfloor$.[11] It is easy to verify that $|M| + \lg(\operatorname{range}[K]) \le s_1 + s_2 + 2^{-\kappa+2}$ and $\operatorname{range}[K] \in [2^\kappa, 2^{\kappa+1})$.

We set

$$M := M_1 \circ M_2 \circ (K' \bmod 2^{|M| - |M_1| - |M_2|})_2,$$

where $(x)_2$ is the binary representation of $x$, and

$$K := K' \operatorname{div} 2^{|M| - |M_1| - |M_2|}.$$

[11] Recall that $\operatorname{frac}(x) = x - \lfloor x \rfloor$.

To see why $K$ is at most $\mathrm{range}[K]-1$, we have

$$\begin{aligned}
K &\le (\mathrm{range}[K_1]\cdot\mathrm{range}[K_2]-1) \operatorname{div} 2^{|M|-|M_1|-|M_2|}\\
&= \left(2^{s_1-|M_1|+s_2-|M_2|}-1\right) \operatorname{div} 2^{|M|-|M_1|-|M_2|}\\
&\le 2^{s_1+s_2-|M|}\\
&\le (\mathrm{range}[K]+1)\cdot 2^{-2^{-\kappa+2}}\\
&\le \mathrm{range}[K]-1,
\end{aligned}$$

where the last inequality uses the fact that $2^{-2^{-\kappa+2}} \le 1-2^{-\kappa+1}$ and $\mathrm{range}[K]\ge 2^\kappa$.

To decode $\mathcal{S}_1$ and $\mathcal{S}_2$, observe that $\mathcal{S}[|M_1|+|M_2|,|M|]$ encodes exactly $K'$. We access $\mathcal{S}$ to retrieve its value, and compute $K_1$ and $K_2$ using $K_1 := K' \operatorname{div} \mathrm{range}[K_2]$ and $K_2 := K' \bmod \mathrm{range}[K_2]$. By our construction, $M_1$ is $M[0,|M_1|-1]$ and $M_2$ is $M[|M_1|,|M_1|+|M_2|-1]$. Hence, we decode $\mathcal{S}_1$ and $\mathcal{S}_2$ in constant time and one access to $\mathcal{S}$. ∎

Using ideas similar to [7], we show that a sequence of strings can be concatenated, as long as one can provide upper bounds $\tilde{T}_i$ of $s_1+\cdots+s_i$ such that $\tilde{T}_i-\tilde{T}_{i-1}\ge s_i+2^{-\kappa+2}$.

PROPOSITION 4.8. *Let $s_1,\ldots,s_B \ge \kappa$ be valid fractional lengths. Suppose there are numbers $\tilde{T}_0,\ldots,\tilde{T}_B$ such that*

- *$\tilde{T}_0=0$ and $\tilde{T}_i-\tilde{T}_{i-1}\ge s_i+2^{-\kappa+2}$;*
- *each $\tilde{T}_i$ is of the form $\tilde{T}_i=\tilde{m}_i+\lg\tilde{R}_i$, where $\tilde{R}_i\in[2^\kappa,2^{\kappa+1})$ and $\tilde{m}_i\ge 0$ are integers;*
- *for any given $i$, $\tilde{T}_i$ can be computed in $O(t)$ time.*

*Then given $B$ strings $\mathcal{S}_1,\ldots,\mathcal{S}_B$, where $\mathcal{S}_i\in\mathbf{FL}(s_i)$, they can be concatenated into one string $\mathcal{S}$ of length $\tilde{T}_B$. Moreover, given any $i$ and $s_i$, $\mathcal{S}_i$ can be decoded using $O(t)$ time and three accesses to $\mathcal{S}$.*

*Proof.* Let $\tilde{s}_i$ be the largest valid fractional length such that $\tilde{s}_i\le\tilde{T}_i-\tilde{T}_{i-1}-2^{-\kappa+2}$. Note that $\tilde{s}_i\ge s_i$ and $\tilde{s}_i$ can be computed in constant time given $\tilde{T}_i$ and $\tilde{T}_{i-1}$. We first apply Proposition 4.5 to convert every $\mathcal{S}_i$ to a string of length $\tilde{s}_i$. By Proposition 4.5, it suffices to decode this new string, and then apply one access to decode $\mathcal{S}_i$. Hence, for simplicity of notations, we simply assume $s_i=\tilde{s}_i$ below.

Without loss of generality, we assume $B$ is odd, since otherwise, we could first apply the following argument to the first $B-1$ strings, then apply Proposition 4.7 to concatenate the outcome with the last string $\mathcal{S}_B$.

Let $\mathcal{S}_i=(M_i,K_i)$ for $i=1,\ldots,B$. To concatenate all strings, we break $\mathcal{S}_2,\ldots,\mathcal{S}_B$ into $(B-1)/2$ pairs, where the $j$-th pair consists of $\mathcal{S}_{2j}$ and $\mathcal{S}_{2j+1}$. We start with the first string $\mathcal{S}_1$, and append the pairs one by one. More specifically, let $\mathcal{S}^{(0)}:=\mathcal{S}_1$. Suppose we have concatenated $\mathcal{S}_1$ and the first $j-1$ pairs into $\mathcal{S}^{(j-1)}=(M^{(j-1)},K^{(j-1)})$, such that $|M^{(j-1)}|=\tilde{m}_{2j-1}$ and $\mathrm{range}[K^{(j-1)}]=\tilde{R}_{2j-1}$. In particular, it has length $\tilde{T}_{2j-1}$. Now, we show how to "append" $\mathcal{S}_{2j}$ and $\mathcal{S}_{2j+1}$ to it.

To this end, we combine $K_{2j}$ and $K_{2j+1}$ into a single integer $L_j$,

$$L_j := K_{2j}\cdot\mathrm{range}[K_{2j+1}]+K_{2j+1}.$$

Thus, $\mathrm{range}[L_j]=\mathrm{range}[K_{2j}]\cdot\mathrm{range}[K_{2j+1}]$, and we have $\mathrm{range}[L_j]\in[2^{2\kappa},2^{2\kappa+2})$. Then, we re-break $L_j$ into a pair $(X_j,Y_j)$, such that the product of $\mathrm{range}[X_j]$ and $\mathrm{range}[K^{(j-1)}]$ is

close to a power of two: we set

$$\mathrm{range}[X_j] := \left\lfloor \frac{2^{\lfloor \tilde{T}_{2j+1} \rfloor - \lfloor \tilde{T}_{2j-1} \rfloor - |M_{2j}| - |M_{2j+1}|}}{\mathrm{range}[K^{(j-1)}]} \right\rfloor,$$

and

$$\mathrm{range}[Y_j] := \left\lceil \frac{\mathrm{range}[L_j]}{\mathrm{range}[X_j]} \right\rceil.$$

Note that

$$2\kappa \leq \lfloor \tilde{T}_{2j+1} \rfloor - \lfloor \tilde{T}_{2j-1} \rfloor - |M_{2j}| - |M_{2j+1}| \leq 2\kappa + 4.$$

To break $L_j$ into such a pair, we let $Y_j := L_j \operatorname{div} \mathrm{range}[X_j]$ and $X_j := L_j \bmod \mathrm{range}[X_j]$. Next, we combine $K^{(j-1)}$ and $X_j$ into an integer $Z_j < 2^{\lfloor \tilde{T}_{2j+1} \rfloor - \lfloor \tilde{T}_{2j-1} \rfloor - |M_{2j}| - |M_{2j+1}|}$: let $Z_j := K^{(j-1)} \cdot \mathrm{range}[X_j] + X_j$.

Finally, we let $\mathcal{S}^{(j)} := (M^{(j)}, K^{(j)})$, where

$$M^{(j)} := M^{(j-1)} \circ (Z_j)_2 \circ M_{2j} \circ M_{2j+1},$$

and

$$K^{(j)} := Y_j.$$

The length of $M^{(j)}$ is

$$\begin{aligned} |M^{(j)}| &= |M^{(j-1)}| + (\lfloor \tilde{T}_{2j+1} \rfloor - \lfloor \tilde{T}_{2j-1} \rfloor - |M_{2j}| - |M_{2j+1}|) + |M_{2j}| + |M_{2j+1}| \\ &= \lfloor \tilde{T}_{2j+1} \rfloor - \kappa \\ &= \tilde{m}_{2j+1}. \end{aligned}$$

The range of $K^{(j)}$ has size

$$\begin{aligned} \mathrm{range}[K^{(j)}] &< \frac{\mathrm{range}[L_j]}{\mathrm{range}[X_j]} + 1 \\ &\leq \frac{\mathrm{range}[K_{2j}] \cdot \mathrm{range}[K_{2j+1}]}{\frac{2^{\lfloor \tilde{T}_{2j+1} \rfloor - \lfloor \tilde{T}_{2j-1} \rfloor - |M_{2j}| - |M_{2j+1}|}}{\mathrm{range}[K^{(j-1)}]} - 1} + 1 \\ &\leq \frac{\mathrm{range}[K_{2j}] \cdot \mathrm{range}[K_{2j+1}] \cdot \mathrm{range}[K^{(j-1)}]}{2^{\lfloor \tilde{T}_{2j+1} \rfloor - \lfloor \tilde{T}_{2j-1} \rfloor - |M_{2j}| - |M_{2j+1}|}} \cdot (1 - 2^{-\kappa+1})^{-1} + 1 \\ &= 2^{s_{2j} + s_{2j+1} + \tilde{T}_{2j-1} - |M^{(j)}|} \cdot (1 - 2^{-\kappa+1})^{-1} + 1 \\ &\leq 2^{\kappa + \mathrm{frac}(\tilde{T}_{2j+1}) - 2^{\kappa+3}} \cdot (1 - 2^{-\kappa+1})^{-1} + 1 \\ &\leq 2^{\kappa + \mathrm{frac}(\tilde{T}_{2j+1})} \\ &= \tilde{R}_{2j+1}. \end{aligned}$$

Thus, $\mathcal{S}^{(j)}$ has length $\tilde{T}_{2j+1}$, and hence, the final string $\mathcal{S} := \mathcal{S}^{((B-1)/2)}$ has length $\tilde{T}_B$.

Next, we show that each $\mathcal{S}_i$ can be decoded in $O(t)$ time and two accesses to $\mathcal{S}$. If $i = 1$, we compute $\tilde{T}_1, \tilde{T}_2, \tilde{T}_3$ in $O(t)$ time, and then use them to compute $\tilde{s}_1, \tilde{s}_2, \tilde{s}_3$ (which we assumed are equal to $s_1, s_2, s_3$ respectively). Then, $\mathrm{range}[Z_1]$, $\mathrm{range}[X_1]$ and $|M_1|$ can be computed according to their definitions. Thus, $M_1 = M[0, |M_1| - 1]$, and $Z_1$ is stored in $M$ immediately after $M_1$. By making one access to $\mathcal{S}$, we recover the value of $Z_1$, and hence,

$K_1$ can be computed using $K_1 = Z_1 \operatorname{div} \operatorname{range}[X_1]$. This decodes $\mathcal{S}_1$ in $O(t)$ time and one access to $\mathcal{S}$.

If $i > 1$, let $j = \lfloor i/2 \rfloor$, i.e., $\mathcal{S}_i$ is in the $j$-th pair. We first compute $\tilde{T}_{2j-1}$, $\tilde{T}_{2j}$ and $\tilde{T}_{2j+1}$ in $O(t)$ time, then compute $s_{2j}$ and $s_{2j+1}$ from them. They together determine $\operatorname{range}[Z_j]$, $\operatorname{range}[X_j]$ and $|M^{(j-1)}|$, as well as the starting location of $M_i$. Thus, $Z_j$ can be recovered with one access to $\mathcal{S}$. $X_j$ can be computed using $X_j = Z_j \bmod \operatorname{range}[X_j]$. Similarly, we then recover $Z_{j+1}$, and $Y_j$ can be computed using $Y_j = K^{(j)} = Z_{j+1} \operatorname{div} \operatorname{range}[X_{j+1}]$ (if $\mathcal{S}_i$ is in the last pair, $Y_j$ is simply $K$ in the final string). This recovers both $X_j$ and $Y_j$. Next, we recover $L_j$ using $L_j = Y_j \cdot \operatorname{range}[X_j] + X_j$, and compute $K_{2j}$ and $K_{2j+1}$ using $K_{2j} = L_j \operatorname{div} \operatorname{range}[K_{2j+1}]$ and $K_{2j+1} = L_j \bmod \operatorname{range}[K_{2j+1}]$. In particular, it recovers the value of $K_i$, and hence, it decodes $\mathcal{S}_i$. ∎

If we can efficiently approximate every $s_1 + \cdots + s_i$, then the strings can be concatenated.

PROPOSITION 4.9. *Let $s_1, \ldots, s_B \geq \kappa$ be valid fractional lengths. Suppose for any given $i$, $s_1 + \cdots + s_i$ can be approximated (deterministically) in $O(t)$ time with an additive error of at most $2^{-\kappa}$. Then given $B$ strings $\mathcal{S}_1, \ldots, \mathcal{S}_B$, where $\mathcal{S}_i \in \mathbf{FL}(s_i)$, they can be concatenated into one string $\mathcal{S}$ of length at most*

$$s_1 + \cdots + s_B + (B-1) \cdot 2^{-\kappa+4}.$$

*Moreover, given any $i$ and $s_i$, $\mathcal{S}_i$ can be decoded using $O(t)$ time and three accesses to $\mathcal{S}$.*

*Proof.* Suppose we can compute $\tilde{S}_i = s_1 + \cdots + s_i \pm 2^{-\kappa}$. We set $\tilde{m}_i = \lfloor \tilde{S}_i + i \cdot 2^{-\kappa+3} \rfloor - \kappa$, $\tilde{R}_i = \lfloor 2^{\tilde{S}_i + i \cdot 2^{-\kappa+3} - \tilde{m}_i} \rfloor$ and $\tilde{T}_i = \tilde{m}_i + \lg \tilde{R}_i$. Then $\tilde{T}_i \leq \tilde{S}_i + i \cdot 2^{-\kappa+3}$, and by the fact that $\lg(1-\varepsilon) \geq -2\varepsilon$ for small $\varepsilon \in (0, 1/2)$, $\tilde{T}_i > \tilde{S}_i + i \cdot 2^{-\kappa+3} - 2^{-\kappa+1}$. Therefore,

$$\begin{aligned} \tilde{T}_i - \tilde{T}_{i-1} &\geq \tilde{S}_i - \tilde{S}_{i-1} + 2^{-\kappa+3} - 2^{-\kappa+1} \\ &\geq s_i + 2^{-\kappa+2}. \end{aligned}$$

Finally, by Proposition 4.8, the size of the data structure is at most $\tilde{T}_B \leq \tilde{S}_B + B \cdot 2^{-\kappa+3} \leq s_1 + \cdots + s_B + (B-1) \cdot 2^{-\kappa+4}$. ∎

In particular, by storing approximations of all $B$ prefix sums in a lookup table of size $O(B)$ words, each $\mathcal{S}_i$ can be decoded in $O(1)$ time. Note that this lookup table *does not* depend on the $B$ strings.

PROPOSITION 4.10. *Let $s_1, \ldots, s_B \geq 0$ be valid fractional lengths. Given $B$ strings $\mathcal{S}_1, \ldots, \mathcal{S}_B$, where $\mathcal{S}_i \in \mathbf{FL}(s_i)$, they can be concatenated into one string $\mathcal{S}$ of length at most*

$$s_1 + \cdots + s_B + (B-1)2^{-\kappa+4}.$$

*Moreover, there is a lookup table of size $O(B)$ words that depends only on $s_1, \ldots, s_B$ such that assuming we can make random accesses to the lookup table, each $\mathcal{S}_i$ can be decoded using constant time and three accesses to $\mathcal{S}$.*

*Proof.* If all $s_i \geq \kappa$, the proposition is an immediate corollary of Proposition 4.9, as we could simply store the approximations of all $B$ prefix sums as well as all $s_i$. For general $s_i \geq 0$, we group the strings so that each group has length at least $\kappa$.

Let $\mathcal{S}_i = (M_i, K_i)$ for $i = 1, \ldots, B$. We greedily divide all strings into groups: Pick the first $i_1$ such that $s_1 + \cdots + s_{i_1} \geq \kappa$, then pick the first $i_2$ such that $s_{i_1+1} + \cdots + s_{i_2} \geq \kappa$, etc. Then each group has total length at least $\kappa$, possibly except for the last group. We store in the lookup table which group each string belongs to, and the values of $i_1, i_2, \ldots$ Then consider a

group consisting of $\mathcal{S}_a, \ldots, \mathcal{S}_b$. We must have $s_a + \cdots + s_{b-1} < \kappa$, which means that they can be combined into one single integer smaller than $\prod_{i=a}^{b-1} \operatorname{range}[K_i] < 2^\kappa$, e.g.,

$$K := \sum_{i=a}^{b-1} K_i \cdot \prod_{j=a}^{i-1} \operatorname{range}[K_j].$$

If we store $\prod_{j=a}^{i-1} \operatorname{range}[K_j]$ and $\operatorname{range}[K_i]$ in the lookup table for each $i$ in the group, then $K_i$ can be recovered from $K$ using

$$K_i = (K \operatorname{div} \prod_{j=a}^{i-1} \operatorname{range}[K_j]) \bmod \operatorname{range}[K_i].$$

This concatenates all strings in the group except the last one. We then apply Proposition 4.7 to concatenate the last string in the group to it. Then we apply Proposition 4.9 to concatenate the strings obtained from each group (except for the last group), using the lookup table. Finally, we concatenate the string obtained from the last group to it.

Concatenating strings in each group loses at most $2^{-\kappa+2}$ bits due to Proposition 4.7. The length of the final string is at most $s_1 + \cdots + s_B + (B-1)2^{-\kappa+4}$. The lookup table has size $O(B)$ words. ∎

**4.3. Fusion.** Next, we consider the *fusion* operation.

DEFINITION 4.11. *Let $(i, \mathcal{S}_i) = (i, (M_i, K_i))$ be a pair in $(\{1\} \times \mathbf{FL}(s_1)) \cup (\{2\} \times \mathbf{FL}(s_2)) \cup \cdots \cup (\{C\} \times \mathbf{FL}(s_C))$. An algorithm* fuses *$i$ into $\mathcal{S}_i$ and obtains a single fractional-length string $\mathcal{S} = (M, K)$, if $M_i$ is a substring of $M$, $i$ can be recovered from $\mathcal{S}$, and $\mathcal{S}_i$ can be decoded from $\mathcal{S}$.*

Equivalently, this is to jointly store the pair $(i, \mathcal{S}_i)$ in one data structure. In the following, we show that the fusion operation can be done with nearly optimal output length such that the input can be decoded efficiently, as long as one can provide upper bounds $\tilde{T}_i$ on $\lg(2^{s_1} + \cdots + 2^{s_i})$ such that $2^{\tilde{T}_i} - 2^{\tilde{T}_{i-1}} \geq 2^{s_i}$.

PROPOSITION 4.12. *Let $s_1, \ldots, s_C \geq 0$ be valid fractional lengths. Suppose there are numbers $\tilde{T}_1, \ldots, \tilde{T}_C$ such that*

- *$2^{\tilde{T}_1} \geq 2^{s_1}$, and $2^{\tilde{T}_i} - 2^{\tilde{T}_{i-1}} \geq 2^{s_i}$ for $i = 2, \ldots, C$;*
- *each $\tilde{T}_i$ is of the form $\tilde{T}_i = \tilde{m} + \lg \tilde{R}_i$, where $\tilde{m}, \tilde{R}_i$ are integers (note that $\tilde{m}$ does not depend on $i$);*
- *$\tilde{T}_C$ is a valid length, i.e., $\tilde{m} = 0$ and $\tilde{R}_C \in [1, 2^\kappa)$, or $\tilde{m} > 0$ and $\tilde{R}_C \in [2^\kappa, 2^{\kappa+1})$;*
- *for any given $K$, the largest $i \leq C$ such that $\tilde{R}_i \leq K$ can be computed in $O(t)$ time.*

*Then given $i \in \{1, \ldots, C\}$ and $\mathcal{S}_i \in \mathbf{FL}(s_i)$, $i$ can be fused into $\mathcal{S}_i$ resulting in a string $\mathcal{S}$ of length $\tilde{T}_C$. Moreover, we can recover the value of $i$ and* decode *$\mathcal{S}_i$ using $O(t)$ time and two accesses to $\mathcal{S}$.*

*Proof.* Let $\mathcal{S}_i = (M_i, K_i)$. Clearly, we have $s_i \leq \tilde{T}_C$ for all $i$, and hence, $|M_i| \leq \tilde{m}$. We first extend $|M_i|$ to obtain $M$, which has length $\tilde{m}$, by appending the least significant bits of $K_i$ to it. That is, let

$$M := M_i \circ (K_i \bmod 2^{\tilde{m}-|M_i|})_2.$$

Next, we encode the remaining information of $(i, \mathcal{S}_i)$ in $K$, i.e., encode $i$ and the top bits of $K_i$:

$$K := \tilde{R}_{i-1} + (K_i \operatorname{div} 2^{\tilde{m}-|M_i|}),$$

where $\tilde{R}_0$ is assumed to be 0. Let the final output $\mathcal{S} := (M, K)$. Note that we have

$$\begin{aligned}\tilde{R}_{i-1} + (\mathrm{range}[K_i] - 1) \operatorname{div} 2^{\tilde{m}-|M_i|} &< \tilde{R}_{i-1} + \mathrm{range}[K_i] \cdot 2^{|M_i|-\tilde{m}} \\ &= \tilde{R}_{i-1} + 2^{s_i-\tilde{m}} \\ &= 2^{-\tilde{m}}(2^{\tilde{T}_{i-1}} + 2^{s_i}) \\ &\leq 2^{\tilde{T}_i-\tilde{m}} \\ &= \tilde{R}_i.\end{aligned}$$

That is, the value of $K$ determines both $i$ and $K_i \operatorname{div} 2^{\tilde{m}-|M_i|}$, and $\mathrm{range}[K]$ is at most $\tilde{R}_C$. Thus, $\mathcal{S}$ is a string of length $\tilde{T}_C$.

To decode $i$ and $\mathcal{S}_i$, we first access $\mathcal{S}$ to retrieve $K$. Then we compute the largest $i \leq C$ such that $\tilde{R}_i \leq K$ in $O(t)$ time. By the argument above, it recovers the value of $i$ and determines

$$(K_i \operatorname{div} 2^{\tilde{m}-|M_i|}) = K - \tilde{R}_i.$$

To decode $\mathcal{S}_i$, observe that $M_i = M[0, |M_i|-1]$, and $M[|M_i|, \tilde{m}-1]$ stores the value of $K_i \bmod 2^{\tilde{m}-|M_i|}$. If $\tilde{m} - |M_i| \leq \kappa + 1$, we retrieve its value using one access, and together with $K_i \operatorname{div} 2^{\tilde{m}-|M_i|}$, it determines $K_i$. Otherwise, since $K_i < 2^{\kappa+1}$, its value is entirely stored in $M$ (in its binary representation). We simply make one access to retrieve it. In both cases, we recover the value of $i$ and decode $\mathcal{S}_i$ in $O(t)$ time and two accesses to $\mathcal{S}$. ∎

Now if one can approximate every $2^{s_1} + \cdots + 2^{s_i}$, then the fusion operation can be done.

PROPOSITION 4.13. *Let $s_1, \ldots, s_C \geq 0$ be valid fractional lengths, where $C \leq 2^{\kappa/2}$. Suppose for any given $i$, $2^{s_1} + \cdots + 2^{s_i}$ can be approximated (deterministically) in $O(t)$ time with an additive error of at most $(2^{s_1} + \cdots + 2^{s_C}) \cdot 2^{-\kappa-3}$. Then given $i \in \{1, \ldots, C\}$ and $\mathcal{S}_i \in \mathbf{FL}(s_i)$, $i$ can be fused into $\mathcal{S}_i$ to obtain $\mathcal{S}$ of length at most*

$$\lg(2^{s_1} + \cdots + 2^{s_C}) + C \cdot 2^{-\kappa+4}.$$

*Moreover, we can recover the value of $i$ and decode $\mathcal{S}_i$ using $O(t \lg C)$ time and two accesses to $\mathcal{S}$.*

Note that the error term is only required to be bounded by a multiple of $2^{s_1} + \cdots + 2^{s_C}$, for every $i$. In particular, it is possible that for some (small) $i$, the error term dominates the value, making the assumption easy to satisfy (for that $i$).

*Proof.* We compute $\tilde{S}_i$ such that

$$\left|\tilde{S}_i - (2^{s_1} + \cdots + 2^{s_i})\right| \leq (2^{s_1} + \cdots + 2^{s_C}) \cdot 2^{-\kappa-3}.$$

If $2^{s_1} + \cdots + 2^{s_C} < 2^{\kappa}$, then the error term $(2^{s_1} + \cdots + 2^{s_C}) \cdot 2^{-\kappa-3} < 1/8$. However, each $2^{s_1} + \cdots + 2^{s_i}$ must be an integer by definition. $\tilde{S}_i$ rounded to the nearest integer is the *accurate* value of $2^{s_1} + \cdots + 2^{s_i}$. To apply Proposition 4.12, we simply set $\tilde{m} := 0$, $\tilde{R}_i := \lfloor \tilde{S}_i + 1/2 \rfloor$ for $i = 1, \ldots, C$ and $\tilde{T}_i = \tilde{m} + \lg \tilde{R}_i$. It is easy to verify that $\tilde{R}_i - \tilde{R}_{i-1} \geq 2^{s_i}$; $\tilde{T}_C$ is a valid length. For any given $K$, by doing a binary search, the largest $i$ such that $\tilde{R}_i \leq K$ can be found in $O(t \lg C)$ time. Thus, by Proposition 4.12, the pair $(i, \mathcal{S}_i)$ can be stored using space

$$\tilde{T}_C = \lg(2^{s_1} + \cdots + 2^{s_C}),$$

and allowing $O(t \lg C)$ time for decoding.

Next, we consider the case where $2^{s_1} + \cdots + 2^{s_C} \geq 2^\kappa$. To apply Proposition 4.12, we let $\tilde{T}_C$ be the largest valid length smaller than $\lg \tilde{S}_C + C \cdot 2^{-\kappa+3}$. That is, we set

$$\tilde{m} := \lfloor \lg \tilde{S}_C + C \cdot 2^{-\kappa+3} \rfloor - \kappa.$$

Then

$$\tilde{R}_C := \lfloor \tilde{S}_C \cdot 2^{C \cdot 2^{-\kappa+3}} \cdot 2^{-\tilde{m}} \rfloor,$$

and $\tilde{T}_C = \tilde{m} + \lg \tilde{R}_C$. For $i < C$, we let

$$\tilde{R}_i := \lfloor \tilde{S}_i \cdot 2^{-\tilde{m}} \rfloor + 2i,$$

and $\tilde{T}_i = \tilde{m} + \lg \tilde{R}_i$.

To apply Proposition 4.12, we need to verify $2^{\tilde{T}_1} \geq 2^{s_1}$, and $2^{\tilde{T}_i} - 2^{\tilde{T}_{i-1}} \geq 2^{s_i}$ for $i > 1$. For $\tilde{T}_1$, we have

$$\begin{aligned} 2^{\tilde{T}_1} &\geq 2^{\tilde{m}} \cdot (\tilde{S}_1 \cdot 2^{-\tilde{m}} + 1) \\ &\geq 2^{s_1} + 2^{\tilde{m}} - (2^{s_1} + \cdots + 2^{s_C}) \cdot 2^{-\kappa-3}. \end{aligned}$$

On the other hand, $\tilde{S}_C = (2^{s_1} + \cdots + 2^{s_C}) \cdot (1 \pm 2^{-\kappa-3})$, i.e., $2^{s_1} + \cdots + 2^{s_C} = \tilde{S}_C \cdot (1 \pm 2^{-\kappa-3})^{-1}$. We have

$$2^{\tilde{m}} - (2^{s_1} + \cdots + 2^{s_C}) \cdot 2^{-\kappa-3} \geq 2^{\tilde{m}} - \tilde{S}_C \cdot 2^{-\kappa-2} \geq 0.$$

Thus, $2^{\tilde{T}_1} \geq 2^{s_1}$. Similarly, for $1 < i < C$, we have

$$\begin{aligned} 2^{\tilde{T}_i} - 2^{\tilde{T}_{i-1}} &= 2^{\tilde{m}} \cdot \left( \tilde{R}_i - \tilde{R}_{i-1} \right) \\ &\geq 2^{\tilde{m}} \cdot (\tilde{S}_i \cdot 2^{-\tilde{m}} - \tilde{S}_{i-1} \cdot 2^{-\tilde{m}} + 1) \\ &\geq 2^{s_i} + 2^{\tilde{m}} - (2^{s_1} + \cdots + 2^{s_C}) \cdot 2^{-\kappa-3} \\ &\geq 2^{s_i}. \end{aligned}$$

For $i = C$, it suffices to show $\lfloor \tilde{S}_C \cdot 2^{-\tilde{m}} \rfloor + 2C \leq \tilde{R}_C$. Indeed, we have

$$\begin{aligned} \tilde{R}_C - (\lfloor \tilde{S}_C \cdot 2^{-\tilde{m}} \rfloor + 2C) &\geq \tilde{S}_C \cdot 2^{C \cdot 2^{-\kappa+3}} \cdot 2^{-\tilde{m}} - 1 - \tilde{S}_C \cdot 2^{-\tilde{m}} - 2C \\ &= \tilde{S}_C \cdot 2^{-\tilde{m}} \cdot (2^{C \cdot 2^{-\kappa+3}} - 1) - 2C - 1 \\ &\geq \tilde{S}_C \cdot 2^{-\tilde{m}} \cdot C \cdot 2^{-\kappa+3} \cdot \ln 2 - 2C - 1. \end{aligned}$$

Since $\tilde{m} + \kappa \leq \lg \tilde{S}_C + 1/2$, it is at least

$$2^{2.5} \cdot C \cdot \ln 2 - 2C - 1 \geq 0.$$

Since each $\tilde{R}_i$ can be computed in $O(t)$ time, by doing a binary search, for any given $K$, we can find the largest $i$ such that $\tilde{R}_i \leq K$ in $O(t \lg C)$ time. By Proposition 4.12, we obtain a data structure of size

$$\tilde{T}_C \leq \lg \tilde{S}_C + C \cdot 2^{-\kappa+3} \leq \lg(2^{s_1} + \cdots + 2^{s_C}) + C \cdot 2^{-\kappa+4}.$$

This proves the proposition. ∎

The decoding algorithm can take constant time if we use a lookup table of $O(C)$ words. Again, the lookup table does not depend on the string.

PROPOSITION 4.14. *Let $s_1, \dots, s_C \geq 0$ be valid fractional lengths. Given an integer $i \in \{1, \dots, C\}$ and a string $\mathcal{S}_i \in \mathbf{FL}(s_i)$, $i$ can be fused into $\mathcal{S}_i$ resulting in a string $\mathcal{S}$ of length at most*

$$\lg(2^{s_1} + \cdots + 2^{s_C}) + C \cdot 2^{-\kappa+2}.$$

*Moreover, there is a lookup table of size $O(C)$ words that depends only on $s_1, \dots, s_C$ such that assuming we can make random accesses to the lookup table, the value of $i$ can be recovered and $\mathcal{S}_i$ can be decoded using constant time and two accesses to $\mathcal{S}$.*

*Proof.* Without loss of generality, assume $s_1 \leq \cdots \leq s_C$, since otherwise, we simply sort $s_1, \dots, s_C$ and store the permutation in the lookup table.

To apply Proposition 4.12, if $2^{s_1} + \cdots + 2^{s_C} \leq 2^\kappa$, we set

$$\tilde{m} := 0,$$

$$\tilde{R}_i = 2^{s_1} + \cdots + 2^{s_i}$$

and $\tilde{T}_i = \tilde{m} + \lg \tilde{R}_i$. Otherwise, if $2^{s_1} + \cdots + 2^{s_C} > 2^\kappa$, we set

$$\tilde{m} := \left\lfloor \lg(2^{s_1} + \cdots + 2^{s_C}) + C \cdot 2^{-\kappa+2} \right\rfloor - \kappa,$$

for $i < C$, let

$$\tilde{R}_i := \lceil 2^{s_1 - \tilde{m}} \rceil + \cdots + \lceil 2^{s_i - \tilde{m}} \rceil,$$

and

$$\tilde{R}_C := \max\left\{ \lceil 2^{s_1 - \tilde{m}} \rceil + \cdots + \lceil 2^{s_C - \tilde{m}} \rceil, 2^\kappa \right\}.$$

Finally, let $\tilde{T}_i = \tilde{m} + \lg \tilde{R}_i$. Clearly, in both cases, we have $2^{\tilde{T}_1} \geq 2^{s_1}$, $2^{\tilde{T}_i} - 2^{\tilde{T}_{i-1}} \geq 2^{\tilde{m}} \cdot 2^{s_i - \tilde{m}} = 2^{s_i}$. Also, we have $\tilde{T}_C \leq \lg(2^{s_1} + \cdots + 2^{s_C}) + C \cdot 2^{-\kappa+2}$. This is because

$$\begin{aligned}
\tilde{R}_C &< \max\{(2^{s_1} + \cdots + 2^{s_C}) \cdot 2^{-\tilde{m}} + C, 2^\kappa\} \\
&= \max\{2^{\kappa + \mathrm{frac}(\lg(2^{s_1} + \cdots + 2^{s_C}) + C2^{-\kappa+2}) - C2^{-\kappa+2}} + C, 2^\kappa\} \\
&\leq \max\{2^{\kappa + \mathrm{frac}(\lg(2^{s_1} + \cdots + 2^{s_C}) + C2^{-\kappa+2})} \cdot (1 - C \cdot 2^{-\kappa+1}) + C, 2^\kappa\} \\
&\leq 2^{\kappa + \mathrm{frac}(\lg(2^{s_1} + \cdots + 2^{s_C}) + C2^{-\kappa+2})}.
\end{aligned}$$

Thus, $\tilde{T}_C = \tilde{m} + \lg \tilde{R}_C \leq \lg(2^{s_1} + \cdots + 2^{s_C}) + C \cdot 2^{-\kappa+2}$. Also, in particular, $\tilde{R}_C < 2^{\kappa+1}$.

Then, we need to show that for any given $K$, the largest $i$ such that $\tilde{R}_i \leq K$ can be found in constant time. To this end, we store a *predecessor search* data structure for the set $\{\tilde{R}_1, \dots, \tilde{R}_C\}$. Note that the set of integers $\{\tilde{R}_1, \dots, \tilde{R}_C\}$ has *monotone gaps*. That is, the difference between adjacent numbers is non-decreasing. Pătraşcu (Claim 7 in [23]) showed that for such sets, there is a predecessor search data structure using linear space and constant query time, i.e., there is an $O(C)$-sized data structure such that given an integer $K$, the query algorithm can answer in constant time the largest value in the set that is at most $K$. This data structure is stored in the lookup table (it only depends on $s_1, \dots, s_C$, but not the input string). To compute the index $i$ rather than $\tilde{R}_i$, we simply store another hash table using perfect hashing in the lookup table. Hence, the lookup table has size $O(C)$ words.

The premises of Proposition 4.12 are all satisfied. The size of $\mathcal{S}$ is $\tilde{T}_C \leq \lg(2^{s_1} + \cdots + 2^{s_C}) + C \cdot 2^{-\kappa+2}$, and $i$ and $\mathcal{S}_i$ can be decoded in constant time. This proves the proposition. □

**4.4. Double-ended strings.** Fractional-length strings have "one of its ends" encoded using an integer. For technical reasons, we also need the following notion of *double-ended* fractional-length strings.

DEFINITION 4.15 (double-ended strings). *For integers $R_h, m, R_t$ such that $m \geq 0$ and $R_h, R_t \in [2^\kappa, 2^{\kappa+1})$, $[R_h] \times \{0,1\}^m \times [R_t]$ is the set of all* double-ended fractional-length strings *of dimension $(R_h, m, R_t)$, denoted by $\mathbf{2}\text{-}\mathbf{FL}(R_h, m, R_t)$.*

*For a double-ended fractional-length string $\mathcal{S} = (K_h, M, K_t) \in \mathbf{2}\text{-}\mathbf{FL}(R_h, m, R_t)$, let*

$$\mathcal{S}[i] := \begin{cases} K_h & i = -1, \\ M[i] & i \in [m], \\ K_t & i = m. \end{cases}$$

*The length of $\mathcal{S}$, denoted by $|\mathcal{S}|$, is defined to be $m + \lg R_h + \lg R_t$. Let $\mathcal{S}[i,j]$ denote the substring $(\mathcal{S}[i], \ldots, \mathcal{S}[j])$. Let $\mathrm{range}[K_h] := R_h, \mathrm{range}[K_t] := R_t$ be the sizes of ranges of $K_h$ and $K_t$ respectively.*

*Remark* 4.16. Note the following facts:

- Unlike the (single-ended) fraction-length strings, the length of a double-ended string does not necessarily determine $\mathrm{range}[K_h]$, $\mathrm{range}[K_t]$, or even $|M|$;
- For $s \geq 2\kappa$, any $(M, K) \in \mathbf{FL}(s)$ can be viewed as a double-ended string by taking the first $\kappa$ bits of $M$ as $K_h$ and letting $K_t$ be $K$;
- For simplicity, in this paper, we do not define double-ended strings with length shorter than $2\kappa$.

Double-ended strings are accessed in the same way as usual fractional-length strings.

DEFINITION 4.17 (access). *Let $\mathcal{S}$ be a double-ended string, an* access *to $\mathcal{S}$ is to retrieve $\mathcal{S}[i,j]$ for $j - i \leq O(\kappa)$.*

Prefixes and suffixes of a double-ended string are defined in the natural way, as follows.

DEFINITION 4.18 (prefix/suffix). *Let $\mathcal{S} = (K_h, M, K_t)$ be a double-ended string. Then $\mathcal{S}[-1, j]$ is a* prefix *of $\mathcal{S}$ for any $j \leq |M|$, $\mathcal{S}[i, |M|]$ is a* suffix *of $\mathcal{S}$ for any $i \geq -1$.*

Using double-ended strings, it is possible to split a double-ended fractional-length string into two strings.

DEFINITION 4.19. *An algorithm* splits *a double-ended fractional-length string $\mathcal{S} = (K_h, M, K_t)$ into two strings $\mathcal{S}_1 = (K_{1,h}, M_1, K_{1,t})$ and $\mathcal{S}_2 = (K_{2,h}, M_2, K_{2,t})$, if*

- *$(K_{1,h}, M_1)$ is a prefix of $\mathcal{S}$,*
- *$(M_2, K_{2,t})$ is a suffix of $\mathcal{S}$, and*
- *$K_{1,t}, K_{2,h}$ together determine $M\,[|M_1|, |M| - |M_2| - 1]$, i.e., the remaining bits of $M$.*

PROPOSITION 4.20. *Let $s = \lg R_h + m + \lg R_t$. Given a double-ended string $\mathcal{S} = (K_h, M, K_t) \in \mathbf{2}\text{-}\mathbf{FL}(R_h, m, R_t)$ and $s_1, s_2 \geq 3\kappa$ such that $s \leq s_1 + s_2 - 2^{-\kappa+2}$, there is an algorithm that splits $\mathcal{S}$ into two strings $\mathcal{S}_1 = (K_{1,h}, M_1, K_{1,t})$ and $\mathcal{S}_2 = (K_{2,h}, M_2, K_{2,t})$ of lengths at most $s_1$ and $s_2$ respectively. Moreover, $\mathrm{range}[K_{i,h}]$, $\mathrm{range}[K_{i,t}]$ and $|M_i|$ can be computed in $O(1)$ time given $R_h$, $m$, $R_t$ and $s_1, s_2$, for $i = 1, 2$; $M\,[|M_1|, |M| - |M_2| - 1]$ can be computed in $O(1)$ time given $K_{1,t}, K_{2,h}$.*

*Remark* 4.21. Proposition 4.20 guarantees that each access to $\mathcal{S}$ can be implemented using at most two accesses to $\mathcal{S}_1$ and $\mathcal{S}_2$. Moreover, accessing a (short) prefix of $\mathcal{S}$ requires only accessing the prefix of $\mathcal{S}_1$ of the same length. Likewise, accessing a suffix of $\mathcal{S}$ requires only accessing the suffix of $\mathcal{S}_2$ of the same length.

Note that split is not the inversion function of concatenation, i.e., if we split $\mathcal{S}$ into $\mathcal{S}_1$ and $\mathcal{S}_2$, then $\mathcal{S}_1$ and $\mathcal{S}_2$ do not concatenate into $\mathcal{S}$ using Proposition 4.7 (the easiest way to see it is that its length becomes strictly longer).

*Proof.* We first calculate the length of $M_1$ and $M_2$, let $|M_1| := \lfloor s_1 - \lg R_h \rfloor - \kappa$ and $|M_2| := \lfloor s_2 - \lg R_t \rfloor - \kappa$. Then let

$$(K_{h,1}, M_1) := \mathcal{S}[-1, |M_1| - 1]$$

be a prefix, and

$$(M_2, K_{t,2}) := \mathcal{S}[m - |M_2|, m]$$

be a suffix. The remaining task is to divide the middle $m - |M_1| - |M_2|$ bits of $M$ into $K_{t,1}$ and $K_{h,2}$.

To this end, we represent the middle bits as an integer $L$ in the range $[2^{m-|M_1|-|M_2|}]$. The sizes of ranges of $K_{t,1}$ and $K_{h,2}$ can be calculated using

$$\mathrm{range}[K_{t,1}] = \lfloor 2^{s_1 - \lg(R_h) - |M_1|} \rfloor$$

and

$$\mathrm{range}[K_{h,2}] = \lfloor 2^{s_2 - \lg(R_t) - |M_2|} \rfloor.$$

Then let $K_{t,1} := L \bmod \mathrm{range}[K_{t,1}]$ and $K_{h,2} := L \operatorname{div} \mathrm{range}[K_{t,1}]$. Clearly, $K_{t,1} \in [\mathrm{range}[K_{t,1}]]$. It suffices to show that $K_{h,2}$ is in its range:

$$\begin{aligned}
K_{h,2} &< \frac{2^{m-|M_1|-|M_2|}}{2^{s_1 - \lg R_h - |M_1|} - 1} \\
&= \frac{2^{m-|M_2|-s_1+\lg R_h}}{1 - 2^{-s_1+\lg R_h+|M_1|}} \\
&\le \frac{2^{s-|M_2|-s_1-\lg R_t}}{1 - 2^{-\kappa}} \\
&\le \frac{2^{s_2-|M_2|-\lg R_t - 2^{-\kappa+2}}}{1 - 2^{-\kappa}} \\
&< (\mathrm{range}[K_{h,2}] + 1) \cdot \frac{2^{-2^{-\kappa+2}}}{1 - 2^{-\kappa}} \\
&\le \mathrm{range}[K_{h,2}] \cdot \frac{(1 + 2^{-\kappa})(1 - 2^{-\kappa+1})}{1 - 2^{-\kappa}} \\
&< \mathrm{range}[K_{h,2}].
\end{aligned}$$

Thus, $\mathcal{S}_1$ has at most $s_1$ bits and $\mathcal{S}_2$ has at most $s_2$ bits. This proves the proposition. ∎

Finally, we study the extraction operation.

DEFINITION 4.22. *Let $s_1, \ldots, s_C \ge 0$. An algorithm* extracts *an integer $i$ from a double-ended fractional-length string $\mathcal{S}$ with respect to lengths $s_1, \ldots, s_C$ and obtains a pair $(i, \mathcal{S}_i)$, if*

- $i \in \{1, \ldots, C\}$,
- *$\mathcal{S}_i = (K_{i,h}, M_i, K_{i,t})$ has length at most $s_i$,*
- *$(M_i, K_{i,t})$ is a suffix of $\mathcal{S}$, and*
- *$\mathcal{S}[-1, m - |M_i| - 1]$ (i.e., the rest of $\mathcal{S}$) can be recovered given $i$ and access to $\mathcal{S}_i$.*

PROPOSITION 4.23. *Let $s_1, \dots, s_C \geq 0$, $R_h, R_t \in [2^\kappa, 2^{\kappa+1})$ and $m \geq \kappa$, let $s = m + \lg R_h + \lg R_t$, and $s \leq \lg(2^{s_1} + \cdots + 2^{s_C}) - C \cdot 2^{-\kappa+2}$. Given a double-ended string $\mathcal{S} = (K_h, M, K_t) \in \mathbf{2\text{-}FL}(R_h, m, R_t)$, there is an algorithm that extracts $i$ from $\mathcal{S}$ with respect to $s_1, \dots, s_C$ and obtains $(i, \mathcal{S}_i)$ for $\mathcal{S}_i = (K_{i,h}, M_i, K_{i,t})$. Moreover, there is a lookup table of size $O(C)$ words that depends only on $s_1, \dots, s_C$ such that assuming we can make random access to the lookup table, $\mathcal{S}[-1, m - |M_i| - 1]$ can be recovered in constant time given $i$ and access to $\mathcal{S}_i$. $\mathrm{range}[K_{i,h}]$, $\mathrm{range}[K_{i,t}]$ and $|M_i|$ does not depend on $\mathcal{S}$, and can be stored in the lookup table.*

*Remark* 4.24. We can safely omit any $i$ with $|M| - |M_i| > \kappa + 1$, since removing such $s_i$ from the list $(s_1, \dots, s_C)$ (and decreasing $C$ by one) could only increase the upper bound on $s$, $\lg(2^{s_1} + \cdots + 2^{s_C}) - C \cdot 2^{-\kappa+2}$. That is, the extraction algorithm may never generate a pair with this $i$. Therefore, we may assume that $\mathcal{S}[-1, |M| - |M_i| - 1]$ has length at most $O(\kappa)$, taking constant time to output.

*Proof.* We first set $K_{i,t} := K_t$. Then, the task becomes to encode $(K_h, M)$ using $(i, (K_{i,h}, M_i))$. Next, we show how to determine $i$. To this end, we divide the range of $K_h$ into $C$ disjoint intervals $\{[l_i, r_i)\}_{i=1,\dots,C}$, such that the $i$-th interval has size at most

$$\lfloor 2^{s_i - m - \lg R_t} \rfloor.$$

Such division is possible, because

$$\begin{aligned}
\sum_{i=1}^{C} \lfloor 2^{s_i - m - \lg R_t} \rfloor &> \sum_{i=1}^{C} 2^{s_i - m - \lg R_t} - C \\
&\geq 2^{-m - \lg R_t} \cdot 2^{s + C \cdot 2^{-\kappa+2}} - C \\
&\geq 2^{s - m - \lg R_t} \cdot (2^{C \cdot 2^{-\kappa+2}} - C \cdot 2^{-\kappa}) \\
&\geq R_h \cdot (1 + C \cdot 2^{-\kappa+1} - C \cdot 2^{-\kappa}) \\
&\geq R_h.
\end{aligned}$$

Fix one such division, e.g., the $i$-th interval is from

$$l_i := \lfloor 2^{s_1 - m - \lg R_t} \rfloor + \cdots + \lfloor 2^{s_{i-1} - m - \lg R_t} \rfloor$$

to

$$r_i := \lfloor 2^{s_1 - m - \lg R_t} \rfloor + \cdots + \lfloor 2^{s_i - m - \lg R_t} \rfloor$$

excluding the right endpoint. We store all endpoints $l_i, r_i$ in the lookup table, taking $O(C)$ space.

Now, find $i$ such that $K_h \in [l_i, r_i)$. Then compute $|M_i| = \lfloor s_i - \lg R_t \rfloor - \kappa$, and let

$$M_i := M[m - |M_i|, m - 1].$$

Finally, we view the first $m - |M_i|$ bits of $M$ as a nonnegative integer $Z \in [2^{m - |M_i|}]$ and let

$$K_{i,h} := 2^{m - |M_i|} \cdot (K_h - l_i) + Z.$$

Observe that $K_{i,h} < \lfloor 2^{\kappa + \mathrm{frac}(s_i - \lg R_t)} \rfloor$, because

$$\begin{aligned}
K_{i,h} &< 2^{m - |M_i|} \cdot (r_i - l_i) \\
&\leq 2^{m - (\lfloor s_i - \lg R_t \rfloor - \kappa)} \cdot 2^{s_i - m - \lg R_t}
\end{aligned}$$

$$= 2^{\kappa+\mathrm{frac}(s_i-\lg R_t)}.$$

Thus, the length of $\mathcal{S}_i = (K_{i,h}, M_i, K_{i,t})$ is at most

$$\lg(\mathrm{range}[K_{i,h}]) + |M_i| + \lg(\mathrm{range}[K_{i,t}]) \le s_i.$$

We also store the sizes of $\mathcal{S}_i$ for every $i$ in the lookup table.

It is clear that $(M_i, K_{i,t})$ is a suffix of $\mathcal{S}$. Given $i$ and $K_{i,h}$, we retrieve $l_i$ and $M_i$ from the lookup table. Then $\mathcal{S}[-1] = K_h$ can be recovered using

$$K_h = l_i + K_{i,h} \operatorname{div} 2^{m-|M_i|}.$$

Also, $Z$ can be recovered using

$$Z = K_{i,h} \bmod 2^{m-|M_i|},$$

which determines $\mathcal{S}[0, m-|M_i|-1]$. This proves the proposition. □

**5. Applications of Fractional-length Strings.**

**5.1. Overview of Pătraşcu's rank data structure.** In this subsection, we summarize how Pătraşcu's rank data structure [23] works, which has important ideas to be used in our data structure. We will "rephrase" this data structure using fractional-length strings, which is a non-trivial simplification.

Given a set $S \subseteq [U]$ of size $n$, the rank data structure preprocesses it into $\approx \lg \binom{U}{n}$ bits, such that for any given query $x$, the number of elements in $S$ that are at most $x$ can be computed in $O(\lg U)$ time (recall that $U$ and $n$ are given ahead of time, hence, both the preprocessing algorithm and the query algorithm know their values). For simplicity, we assume that the query algorithm is allowed to access lookup tables that depend only on $U$ and $n$. The idea is to recursively construct data structures for smaller universes, and then merge them using concatenation and fusion. Suppose $S$ has $i$ elements in $[U/2]$, the first half of the universe, and it has $n-i$ elements in the second half. We first recursively construct (fractional-length) data structures for both halves, using space $\approx \lg \binom{U/2}{i}$ and $\approx \lg \binom{U/2}{n-i}$ respectively. Next, we concatenate two data structures using Proposition 4.10, and obtain one single data structure $\mathcal{S}_i$, which has length $\approx \lg\left(\binom{U/2}{i}\binom{U/2}{n-i}\right)$. Note that the data structure $\mathcal{S}_i$ encodes an input set $S$ with exactly $i$ elements in the first half (and $n-i$ in the second half), *assuming* the value of $i$ is known. Finally, we fuse $i$ into $\mathcal{S}_i$, and obtain one single data structure $\mathcal{S}$. Proposition 4.14 guarantees that $\mathcal{S}$ has length roughly

$$\lg\left(\sum_{i=0}^{n} 2^{|\mathcal{S}_i|}\right) \approx \lg\left(\sum_{i=0}^{n} \binom{U/2}{i}\binom{U/2}{n-i}\right) = \lg\binom{U}{n}$$

bits.

This recursion terminates at sets of size $n = 0$ or $n = U$, in which case there is nothing to store (again $n$ does not need encoding, so it is clear which case we are in). The propositions guarantee that both concatenation and fusion are implemented such that each operation only causes an overhead of no more than $O(1/U^2)$ bits, by setting $\kappa = 3\lg U$. Therefore, the overall space is no more than $\lg \binom{U}{n} + O(1/U)$. For the final (fractional-length) data structure $(M, K)$, we simply write $K$ in its binary representation and append it to $M$. This gives us an integral-length data structure using at most $\lceil \lg \binom{U}{n} \rceil + 1$ bits.

It is then straightforward to answer a `rank` query on this data structure. Given a query $x$, we first recover the value of $i$, and decode $\mathcal{S}_i$ (again, decoding $\mathcal{S}_i$ does not mean reconstructing it). Then we further decode $\mathcal{S}_i$ into the two data structures for the two halves. This can be done in constant time using a lookup table. Next, if $x < U/2$, we recurse into the first half. If $x \geq U/2$, we recurse into the second half (and add $i$ to the final answer). Since each time $U$ decreases by a factor of two, the query time is $O(\lg U)$.

In [23], it is also shown that when $U$ is small, we can do a $B$-way divide-and-conquer, as long as $B \lg U \leq O(w)$ (recall that $w$ is the word size). Therefore when $U \leq w^{O(1)}$, we can afford to set $B = w^{1/2}$ and have only constant depth of recursion (rather than $O(\lg U)$). This gives us a rank data structure with constant query time for small $U$. As we will see in Section 8.1, we show that it is possible to further improve this result, and we design a constant-query-time data structure when only $n$ is bounded by $w^{O(1)}$ (and $U$ could be still as large as $2^{\Theta(w)}$). This will be the starting point of our new data structure.

**5.2. Overview of data interpretation.** Now, we give a high-level description on designing a data interpretation algorithm, i.e., converting a (fractional-length) string to a set. The idea is similar to the rank data structure described in the previous subsection, with all steps done in the opposite direction.

Given a string $\mathcal{D}$ of length $\approx \lg \binom{V}{m}$, to interpret it as a set of size $m$, we first *extract* an integer $i$ from $\mathcal{D}$ such that $i \in \{0, \dots, m\}$ and $\mathcal{D}_i$ has length $s_i \approx \lg \binom{V/2}{i}\binom{V/2}{m-i}$. Then we *split* $\mathcal{D}_i$ of length $s_i$ into two substrings $\mathcal{D}_a$ and $\mathcal{D}_b$ of lengths $\approx \lg \binom{V/2}{i}$ and $\approx \lg \binom{V/2}{m-i}$ respectively. The integer $i$ will represent the number of keys in the first half of the universe, and $m - i$ is the number of keys in the second half. We recursively construct sets $S_a, S_b \subseteq [V/2]$ from $\mathcal{D}_a$ and $\mathcal{D}_b$ of sizes $i$ and $m-i$ respectively. Then the final set $S$ is $S_a \cup (V/2 + S_b)$.

To access $w$ consecutive bits of $\mathcal{D}$ given a $\texttt{rank}_S(\cdot)$ oracle, we first ask the oracle $\texttt{rank}_S(V/2)$, i.e., the number of keys in the first half. This determines the value of $i$, and hence the lengths of $\mathcal{D}_a$ and $\mathcal{D}_b$, which in turn determines whether the $w$ consecutive bits are entirely in $\mathcal{D}_a$, or entirely in $\mathcal{D}_b$, or split across the two substrings. If it is entirely contained in one substring, we simply recurse into the corresponding half of the universe. On the other hand, it is possible to show that splitting across the two substrings does not happen more than once, and when it happens, we recurse into both halves. The recursion has depth $O(\lg V)$, and so is the query time. The formal argument can be found in Section 8.2.

**5.3. Storing a sequence.** The following lemma by Dodis, Pătraşcu and Thorup [7] shows that a sequence in $[\sigma]^n$ can be stored using almost optimal space such that each symbol can be retrieved in constant time. Their construction requires a lookup table of $\Theta(\lg n)$ words. Here, we show that using fractional-length strings, the lookup table can be made more explicit, i.e., knowing $\sigma^i$ for all $i = O(w/\lg \sigma)$. When $\sigma = 2^{\Theta(w)}$, it completely removes the lookup table.

LEMMA 5.1 (see also [7]). *Fix integer $\kappa = O(w)$. There is an algorithm that preprocesses a given sequence $(x_1, \dots, x_n) \in [\sigma]^n$ for $\sigma \leq 2^\kappa$ into a data structure of length at most $n \lg \sigma + (n-1)2^{-\kappa+5}$, such that given any $i$, $x_i$ can be retrieved in constant time, assuming the query algorithm knows $\sigma^i$ for all $i \leq 2\kappa/\lg \sigma$.*

*Proof.* Let $b = \lceil 2\kappa/\lg \sigma \rceil$. We partition the sequence into $n/b$ chunks of $b$ symbols each, then combine each chunk into one single character in $[\sigma^b]$ (if $n$ is not a multiple of $b$, then the last group will have between $b+1$ and $2b-1$ symbols). Since $\sigma^b = 2^{O(\kappa)} = 2^{O(w)}$, each $x_i$ can be decoded in constant time given the character and $\sigma^i$. Then compute $m = \lfloor \lg \sigma^b \rfloor - \kappa$ and $R = \lceil \sigma^b \cdot 2^{-m} \rceil$, and view each character in $[\sigma^b]$ as a data structure of size $m + \lg R$. Note that $m + \lg R - b \lg \sigma \leq \lg(\sigma^b + 2^m) - b \lg \sigma \leq \lg(1 + 2^{-\kappa}) \leq 2^{-\kappa+1}$. Then we apply Proposition 4.9 to concatenate all $n/b$ data structures. Since $m$ and $R$ can both be computed

in constant time, $m + \lg R$ can be approximated in constant time, hence Proposition 4.9 guarantees that there is a data structure of size

$$(m + \lg R) \cdot (n/b) + (n/b - 1) \cdot 2^{-\kappa+4} \leq n \lg \sigma + (n-1) \cdot 2^{-\kappa+5},$$

supporting symbol retrieval in constant time. This proves the lemma. ∎

**6. Reductions to Perfect Hashing.** In this section, we show how to design a succinct dictionary and compress a low entropy sequence with local decodability using the 2-PHM data structure in Theorem 1.3.

**6.1. Succinct dictionary.** Assuming Theorem 1.3, we can prove Theorem 1.1 using Lemma 5.1.

*Proof of Theorem 1.1.* To store a set of $n$ key-value pairs for keys in $[U]$ and values in $[\sigma]$, we first apply Theorem 1.3 on the set of keys. It produces a data structure of size $\lg \binom{U}{n} + \operatorname{poly} \lg n + O(\lg \lg U)$ bits, which determines a 2-PHM 2-hq. In particular, 2-hq restricted to the $n$ keys can be viewed as a bijection $h$ between the keys and $[n]$. Next, we apply Lemma 5.1 to store the values. Specifically, we construct the sequence $(v_1, \ldots, v_n)$ such that if $(x, u)$ is an input key-value pair, then $v_{h(x)+1} = u$. This sequence can be stored in space $n \lg \sigma + O(1)$ by Lemma 5.1. Hence, the total space of the data structure is

$$\lg \binom{U}{n} + n \lg \sigma + \operatorname{poly} \lg n + O(\lg \lg U) = \mathbf{OPT} + \operatorname{poly} \lg n + O(\lg \lg U),$$

as claimed in the theorem statement.

To answer a query $\mathbf{valRet}(x)$, we first query the perfect hashing data structure. If $x$ is not a key, we return $\perp$. Otherwise, we retrieve and return the $(h(x)+1)$-th value in the sequence. The total query time is constant in expectation. This proves the theorem. ∎

**6.2. Compression to zeroth order entropy with local decodability.** Next, we prove Theorem 1.4 assuming Theorem 1.3.

*Proof of Theorem 1.4.* Given a sequence $(x_1, \ldots, x_n) \in \Sigma^n$ such that each $\sigma \in \Sigma$ appears $f_\sigma$ times, we construct a data structure *recursively on* $\Sigma$. We first arbitrarily partition $\Sigma$ into $\Sigma_1 \cup \Sigma_2$ such that $|\Sigma_1| = \lfloor |\Sigma|/2 \rfloor$ and $|\Sigma_2| = \lceil |\Sigma|/2 \rceil$. For any set $\Gamma \subseteq \Sigma$, define $S_\Gamma := \{i \in [n] : x_i \in \Gamma\}$. Then we apply Theorem 1.3 to construct a 2-PHM for $S_{\Sigma_1}$, which uses space

$$\lg \binom{n}{|S_{\Sigma_1}|} + \operatorname{poly} \lg n = \lg \left( \frac{n!}{|S_{\Sigma_1}|! \cdot |S_{\Sigma_2}|!} \right) + \operatorname{poly} \lg n,$$

and defines a bijection $h$ that maps all coordinates in $S_{\Sigma_1}$ to $[|S_{\Sigma_1}|]$, and a bijection $\overline{h}$ that maps all $S_{\Sigma_2}$ to $[|S_{\Sigma_2}|]$. We recursively construct a data structure for $\Sigma_1$ over $h(S_{\Sigma_1})$, and a data structure for $\Sigma_2$ over $\overline{h}(|S_{\Sigma_2}|)$.

In general, each node in the recursion tree corresponds to a subset $\Gamma$ of the alphabet such that $\Gamma_1$ and $\Gamma_2$ are the subsets corresponding to the left and the right child respectively. In this node, we store

- the size of subset of its left child $|\Gamma_1|$,
- the perfect hashing data structure for $S_{\Gamma_1}$,
- two pointers to the data structures in its left and its right children.

For $\Gamma$ of size one, we store nothing. Thus, we obtain a final data structure of size

$$\lg \left( \frac{n!}{f_{\sigma_1}! f_{\sigma_2}! \cdots} \right) + |\Sigma| \cdot \operatorname{poly} \lg n.$$

To answer a query $i$, we first retrieve the size of $\Sigma_1$ and query if $i \in S_{\Sigma_1}$. If $i \in S_{\Sigma_1}$, we go to the left child and recursively query $h(i)$. If $i \notin S_{\Sigma_1}$, we go to the right child and recursively query $\overline{h}(i)$. Finally, when the current subset $|\Gamma| = 1$, we return the only element in $\Gamma$. Since in each level of the recursion, the perfect hashing data structure takes constant query time in expectation, and the size of $\Gamma$ reduces by a factor two, the total query time is $O(\lg |\Sigma|)$ in expectation. This proves the theorem. ∎

**7. Perfect Hashing for Medium-Sized Sets.** In this section, we present the 2-PHM data structure when the number of keys $n$ is neither too large nor too small, focusing on the case where $n \geq U^{1/12}$ and $n \leq U - U^{1/12}$. Generalizing to all $n$ involves fewer new ideas, and we defer the proof of the main theorem to Section 9.

**7.1. Block pairs.** As we mentioned in Section 2.2, to construct a 2-PHM for input set $S$, we partition the universe $[U]$ into pairs of blocks. For each pair, we construct a *main* data structure and an *auxiliary* data structure, such that the main data structure contains "most" of the information in the block and has fixed length, and the auxiliary data structure stores the remaining information (which unavoidably has variable length). Finally, we concatenate all data structures for all blocks.

Below is our main technical lemma, which constructs such two (fractional-length) data structures for a pair of blocks of sizes $V$ and $V_{\text{sec}}$. Roughly speaking, given a set $S$ of size between $\kappa^{2c-3} + \kappa^c/3$ and $\kappa^{2c-3} + 2\kappa^c/3$ in the block $[V]$, and a set $S_{\text{sec}}$ of size $O(\kappa^{c+1})$ in the second block $V + [V_{\text{sec}}]$, we can preprocess them into two data structures $\mathcal{D}_{\text{main}}$ and $\mathcal{D}_{\text{aux}}$ using a randomized algorithm such that the two data structures together define a 2-PHM 2-hq for $S \cup S_{\text{sec}}$. The total size of the two data structures is very close to the optimal space:

$$|\mathcal{D}_{\text{main}}| + |\mathcal{D}_{\text{aux}}| \approx \lg \binom{V}{|S|} + \lg \binom{V_{\text{sec}}}{|S_{\text{sec}}|},$$

while $|\mathcal{D}_{\text{main}}|$ does not depend on the actual set sizes $|S|, |S_{\text{sec}}|$. Moreover, a query algorithm can access only $\mathcal{D}_{\text{main}}$ and output 2-hq$(x)$ in $O(1)$ time without knowing $|S|, |S_{\text{sec}}|$, for "most" queries $x \in [V]$; otherwise, it outputs "unknown". All "unknown" queries in $[V]$ and the queries in $V + [V_{\text{sec}}]$ can be answered in $\kappa^{O(1)}$ time by accessing both data structures.

LEMMA 7.1 (main technical lemma). *Let $\kappa$ be the fineness parameter for fractional-length strings, and $c$ be a constant positive integer. Let $V \in [2\kappa^{2c-3}, 2^{\kappa/2}]$ and $V_{\text{sec}} \geq 4\kappa^{c+1}$. For any constant $\epsilon > 0$, there is a preprocessing algorithm* `perfHashBlk`*, query algorithms* $\texttt{qalgBlk}_{\text{main}}$, `qalgBlk` *and lookup tables* $\texttt{tableBlk}_{V,V_{\text{sec}}}$ *of size $\tilde{O}(2^{\epsilon\kappa})$. Given*

- *a set $S \subseteq [V]$ such that $m := |S| \in [\kappa^{2c-3} + \kappa^c/3, \kappa^{2c-3} + 2\kappa^c/3]$,*
- *a set $S_{\text{sec}} \subseteq V + [V_{\text{sec}}]$ and $m_{\text{sec}} := |S_{\text{sec}}| \in [\kappa^{c+1}, 3\kappa^{c+1}]$,*
- *a random string $\mathcal{R}$ of $\kappa^{c+1}$ bits,*

`perfHashBlk` *preprocesses $S$ and $S_{\text{sec}}$ into a pair of two (fractional-length) data structures $\mathcal{D}_{\text{main}}$ and $\mathcal{D}_{\text{aux}}$, such that*

*(i) $\mathcal{D}_{\text{main}}$ has length at most*

$$\lg \binom{V}{\kappa^{2c-3}, \kappa^c} + \kappa^{2c-3} \cdot 2^{-\kappa/2+1};$$

*(ii) $\mathcal{D}_{\text{aux}}$ has length at most*

$$\lg \binom{V}{m} + \lg \binom{V_{\text{sec}}}{m_{\text{sec}}} - \lg \binom{V}{\kappa^{2c-3}, \kappa^c} + \kappa^{c+1} 2^{-\kappa/2+2};$$

*(iii) $\mathcal{D}_{\text{main}}$ and $\mathcal{D}_{\text{aux}}$ together define a* bijection $h$ *between*

$$S \cup S_{\text{sec}} \quad \textit{and} \quad [m + m_{\text{sec}}],$$

*and a* bijection $\overline{h}$ *between*

$$[V + V_{\text{sec}}] \setminus (S \cup S_{\text{sec}}) \quad \textit{and} \quad [(V + V_{\text{sec}}) - (m + m_{\text{sec}})],$$

*such that* $h(S) \supset [\kappa^{2c-3}]$ *and* $\overline{h}([V] \setminus S) \supset [V - \kappa^{2c-3} - \kappa^c]$*; let*

$$\text{2-hq}(x) := \begin{cases} (0, h(x)) & \text{if } x \in S \cup S_{\text{sec}}, \\ (1, \overline{h}(x)) & \text{if } x \notin S \cup S_{\text{sec}}; \end{cases}$$

*(iv) given any* $x \in [V]$, $\texttt{qalgBlk}_{\text{main}}(V, x)$ *outputs* $\text{2-hq}(x)$ *when* $x \in S$ *and* $h(x) \in [\kappa^{2c-3}]$*, or when* $x \notin S$ *and* $\overline{h}(x) \in [V - \kappa^{2c-3} - \kappa^c]$*, otherwise it outputs "unknown"; moreover, it only accesses* $\mathcal{D}_{\text{main}}$*,* $\mathcal{R}$ *and the lookup table* $\texttt{tableBlk}_{V,V_{\text{sec}}}$*, and it runs in constant time in the worst case;*

*(v) for any* $x \in [V]$*, the probability that* $\texttt{qalgBlk}_{\text{main}}(V, x)$ *outputs "unknown" is at most* $O(\kappa^{-c+3})$ *over the randomness of* $\mathcal{R}$*;*

*(vi) given any* $x \in [V + V_{\text{sec}}]$*,* $\texttt{qalgBlk}(V, m, V_{\text{sec}}, m_{\text{sec}}, x)$ *computes* $\text{2-hq}(x)$*; it accesses* $\mathcal{D}_{\text{main}}$*,* $\mathcal{D}_{\text{aux}}$*,* $\mathcal{R}$ *and the lookup table* $\texttt{tableBlk}_{V,V_{\text{sec}}}$*, and it runs in* $O(\kappa^4)$ *time.*

*Remark* 7.2. The size of $\mathcal{D}_{\text{main}}$ is close to the optimum, as $\texttt{qalgBlk}_{\text{main}}$ has to identify a set of $\kappa^{2c-3}$ keys and $V - \kappa^{2c-3} - \kappa^c$ non-keys by only accessing $\mathcal{D}_{\text{main}}$, which takes exactly $\lg \binom{V}{\kappa^{2c-3}, V-\kappa^{2c-3}-\kappa^c} = \lg \binom{V}{\kappa^{2c-3}, \kappa^c}$ bits.

The proof of the lemma is deferred to Section 8.

**7.2. Basic setup.** The rest of this entire section is devoted to the construction of the 2-PHM for $U^{1/12} \le n \le U - U^{1/12}$, assuming Lemma 7.1. Formally, we will prove the following theorem (recall that $\mathbf{OPT}_{V,m} = \lg \binom{V}{m}$ is the information theoretical optimal space for storing $m$ keys over $[V]$).

THEOREM 7.3. *For any constant* $\epsilon > 0$ *and constant integer* $c > 0$*, there is a preprocessing algorithm* `perfHash`*, a query algorithm* `qAlg` *and lookup tables* $\texttt{table}_{U,n}$ *of size* $n^\epsilon$*, such that given*

- *a set* $S$ *of* $n$ *keys over the key space* $[U]$*, where* $n \ge U^{1/12}$ *and* $n \le U - U^{1/12}$*,*
- *a uniformly random string* $\mathcal{R}$ *of length* $O(\lg^{c+1} n)$*,*

`perfHash` *preprocesses* $S$ *into a data structure* $\mathcal{D}$ *of (worst-case) length*

$$\lceil \mathbf{OPT}_{U,n} + U^{-1} \rceil,$$

*such that* $\mathcal{D}$ *defines* 2-hq*, a 2-PHM for* $S$*. Given access to* $\mathcal{D}$*,* $\mathcal{R}$ *and* $\texttt{table}_{U,n}$*, for any key* $x \in [U]$*,* $\texttt{qAlg}(U, n, x)$ *outputs* $\text{2-hq}(x)$ *on a RAM with word size* $w = \Omega(\lg U)$*, in time*

- $O(1)$ *with probability* $1 - O(\lg^{-c+4} U)$ *and*
- $O(\lg^7 U)$ *in worst case,*

*where the probability is taken over the randomness in* $\mathcal{R}$*. In particular, the query time is constant in expectation and with high probability by setting* $c = 11$*.*

*Remark* 7.4. When $n < U^{1/12}$, we could use a hash function to map the keys to $n^2$ buckets with no collisions. We could apply this theorem with the new key space being all buckets, and the keys being the non-empty buckets. By further storing for each non-empty bucket, the key within it (using Lemma 5.1), it extends the membership query to $n < U^{1/12}$, using $O(\lg n + \lg \lg U)$ extra bits. We will see a more generic approach in Section 9 (which works for perfect hashing and improves the $\lg \lg U$ term).

Without loss of generality we may assume $n \le U/2$, since otherwise we could simply take the complement of $S$ (note that this assumption can only be made for the problem of

Theorem 7.3, but not for the general dictionary problem). Let $\kappa := \lceil 4 \lg U \rceil$ be the fineness parameter, and $c$ be a (large) constant positive integer to be specified later. We partition the universe $[U]$ into pairs of blocks. Each block pair consists of a larger *primary* block containing roughly $\kappa^{2c-3} + \kappa^c/2$ keys, and a smaller *secondary* block containing roughly $2\kappa^{c+1}$ keys. Formally, let

$$V_{\text{pri}} := \left\lfloor \frac{(\kappa^{2c-3} + \kappa^c/2)U}{n} \right\rfloor,$$

$$V_{\text{sec}} := \left\lfloor \frac{2\kappa^{c+1}U}{n} \right\rfloor$$

and $V_{\text{bl}} = V_{\text{pri}} + V_{\text{sec}}$. Every primary block has size $V_{\text{pri}}$ and every secondary block has size $V_{\text{sec}}$. Every block pair has size $V_{\text{bl}}$. For simplicity, let us first consider the case where $U$ is a multiple of $V_{\text{bl}}$, and $U = V_{\text{bl}} \cdot N_{\text{bl}}$. We will show how to handle general $U$ later.

Thus, we partition $U$ into $N_{\text{bl}}$ block pairs in the natural way, where the $i$-th primary block

$$\mathcal{B}^i_{\text{pri}} := \{x \in [U] : (i-1)V_{\text{bl}} \le x < V_{\text{pri}} + (i-1)V_{\text{bl}}\}$$

and the $i$-th secondary block

$$\mathcal{B}^i_{\text{sec}} := \{x \in [U] : V_{\text{pri}} + (i-1)V_{\text{bl}} \le x < iV_{\text{bl}}\}.$$

We call the $i$-th block pair *good*, if the numbers of keys in the primary and secondary blocks are close to the average:

$$|S \cap \mathcal{B}^i_{\text{pri}}| \in [\kappa^{2c-3} + \kappa^c/3, \kappa^{2c-3} + 2\kappa^c/3],$$

and

$$|S \cap \mathcal{B}^i_{\text{sec}}| \in [\kappa^{c+1}, 3\kappa^{c+1}].$$

The pair is *bad* if the number of keys in at least one of the two blocks is outside the prescribed range.

In Section 7.3, we show that we can construct a data structure for inputs $S$ with no bad blocks. The goal is to design a data structure using space close to $\mathbf{OPT}_{N_{\text{bl}}V_{\text{bl}},n}$. Then in Section 7.4, we handle inputs with at least one bad block. Finally, we put together the two cases and handle general $U$ (not necessarily a multiple of $V_{\text{bl}}$) in Section 7.5.

**7.3. No bad block pair.** We prove the following lemma in this subsection.

LEMMA 7.5. *If $U$ is a multiple of $V_{\text{bl}}$, then there is a data structure with the guarantees as in Theorem 7.3, for all sets $S$ with no bad block pair. Moreover, the size of the data structure is*

$$\mathbf{OPT}_{U,n} + n \cdot 2^{-\kappa/2+2}.$$

*Proof.* As a preparatory step in the preprocessing (i.e. the construction of the data structure), we take the last $O(\lg U)$ bits of the random bits $\mathcal{R}$, interpret them as a (random) number $\Delta \in [U]$, and shift the entire universe according to $\Delta$, i.e., $x \mapsto (x + \Delta) \bmod U$. This shift is applied to input $S$, and will be applied to the queries too (which guarantees that the query is in a primary block with good probability).

The preprocessing algorithm is based on recursion. The following preprocessing algorithm `dict_prep_rec` preprocesses $S$ restricted to the $i$-th to $j$-th blocks $\mathcal{B}^i_{\text{pri}}, \mathcal{B}^i_{\text{sec}}, \ldots, \mathcal{B}^j_{\text{pri}}$, $\mathcal{B}^j_{\text{sec}}$, and outputs $j - i + 2$ data structures $\mathcal{D}^i_{\text{main}}, \ldots, \mathcal{D}^j_{\text{main}}$ and $\mathcal{D}_{\text{aux}}$. We will inductively prove upper bounds on the sizes of the data structures: the length of each $\mathcal{D}^i_{\text{main}}$ is at most

$$\texttt{SIZE}_{\text{main}} := \lg \binom{V_{\text{pri}}}{\kappa^{2c-3}, \kappa^c} + \kappa^{2c-3} \cdot 2^{-\kappa/2+1}, \tag{7.1}$$

and the length of $\mathcal{D}_{\text{aux}}$ generated from $i, \ldots, j$-th block pair is at most

$$\mathbf{OPT}_{(j-i+1)V_{\text{bl}},m} - (j-i+1)\texttt{SIZE}_{\text{main}} + (m-1)2^{-\kappa/2+2}, \tag{7.2}$$

where $m$ is the number of keys in blocks $i$ to $j$. In the base case with only one block pair, we simply apply Lemma 7.1.

**preprocessing algorithm** $\texttt{dict\_prep\_rec}(i, j, m, S, \mathcal{R})$:

1. if $i = j$
2. let $S_{\text{pri}} \subseteq S$ be the set of keys in the $i$-th primary block
3. let $S_{\text{sec}} \subseteq S$ be the set of keys in the $i$-th secondary block
4. $m_{\text{pri}} := |S_{\text{pri}}|$ and $m_{\text{sec}} := |S_{\text{sec}}|$
5. $(\mathcal{D}^i_{\text{main}}, \mathcal{D}'_{\text{aux}}) := \texttt{perfHashBlk}(V_{\text{pri}}, m_{\text{pri}}, V_{\text{sec}}, m_{\text{sec}}, S_{\text{pri}}, S_{\text{sec}}, \mathcal{R})$ (from Lemma 7.1)
6. apply Proposition 4.14 to fuse $m_{\text{pri}}$ into $\mathcal{D}'_{\text{aux}}$, and obtain $\mathcal{D}_{\text{aux}}$
7. return $(\mathcal{D}^i_{\text{main}}, \mathcal{D}_{\text{aux}})$

(to be cont'd)

CLAIM 7.6. *If* $i = j$, $|\mathcal{D}^i_{\text{main}}| \le \texttt{SIZE}_{\text{main}}$ *and* $|\mathcal{D}_{\text{aux}}| \le \mathbf{OPT}_{V_{\text{bl}},m} - \texttt{SIZE}_{\text{main}} + (m-1) \cdot 2^{-\kappa/2+2}$.

To prove the claim, note that the premises of Lemma 7.1 are satisfied: since $2n \le U$, $V_{\text{pri}} \ge 2\kappa^{2c-3}$ and $V_{\text{pri}} \le U \le 2^{\kappa/2}$; $V_{\text{sec}} \ge 4\kappa^{c+1}$; by assumption, every primary block has between $\kappa^{2c-3} + \kappa^c/3$ and $\kappa^{2c-3} + 2\kappa^c/3$ keys, and every secondary block has between $\kappa^{c+1}$ and $3\kappa^{c+1}$ keys. Therefore, by Lemma 7.1, the size of $\mathcal{D}^i_{\text{main}}$ is at most $\texttt{SIZE}_{\text{main}}$, and the size of $\mathcal{D}'_{\text{aux}}$ is at most

$$\lg \binom{V_{\text{pri}}}{m_{\text{pri}}} + \lg \binom{V_{\text{sec}}}{m - m_{\text{pri}}} - \texttt{SIZE}_{\text{main}} + \kappa^{2c-3} \cdot 2^{-\kappa/2+2}.$$

By fusing the value of $m_{\text{pri}} \in \{0, \ldots, m\}$ into the data structure, the size of $\mathcal{D}_{\text{aux}}$ is at most

$$\mathbf{OPT}_{V_{\text{bl}},m} - \texttt{SIZE}_{\text{main}} + (m-1) \cdot 2^{-\kappa/2+2},$$

due to the fact that $\sum_{m_{\text{pri}}} \lg \left( \binom{V_{\text{pri}}}{m_{\text{pri}}} \binom{V_{\text{sec}}}{m-m_{\text{pri}}} \right) = \lg \binom{V_{\text{bl}}}{m} = \mathbf{OPT}_{V_{\text{bl}},m}$ and $m \ge \kappa^{2c-3} + \kappa^{c+1}$. $\mathcal{D}^i_{\text{main}}$ and $\mathcal{D}_{\text{aux}}$ both have sizes as claimed in (7.1) and (7.2). Also, note that we give the same random string $\mathcal{R}$ to *all* block pairs. Thus, the total number of random bits needed is $\kappa^{c+1}$ by Lemma 7.1.

Next, when $i < j$, the set of block pairs is split evenly, the algorithm is applied to both parts recursively, and for the result, the lists of the main structures are simply written one after the other, the auxiliary structures are merged into one (by concatenation).

8. $k := \lfloor (i+j)/2 \rfloor$
9. let $m_1$ be the number of keys in the $i$-th, $\ldots$, $k$-th block pair
10. let $m_2$ be the number of keys in the $(k+1)$-th, $\ldots$, $j$-th block pair
11. recurse on the two halves:
    $(\mathcal{D}^i_{\text{main}}, \ldots, \mathcal{D}^k_{\text{main}}, \mathcal{D}_{\text{aux},1}) := \texttt{dict\_prep\_rec}(i, k, m_1, S, \mathcal{R})$
    $(\mathcal{D}^{k+1}_{\text{main}}, \ldots, \mathcal{D}^j_{\text{main}}, \mathcal{D}_{\text{aux},2}) := \texttt{dict\_prep\_rec}(k+1, j, m_2, S, \mathcal{R})$
12. apply Proposition 4.9 to concatenate $\mathcal{D}_{\text{aux},1}$ and $\mathcal{D}_{\text{aux},2}$, and obtain $\mathcal{D}'_{\text{aux}}$
13. apply Proposition 4.13 to fuse the value of $m_1$ into $\mathcal{D}'_{\text{aux}}$ for $m_1 \in \{0, \ldots, m\}$, and obtain $\mathcal{D}_{\text{aux}}$
14. return $(\mathcal{D}^i_{\text{main}}, \ldots, \mathcal{D}^j_{\text{main}}, \mathcal{D}_{\text{aux}})$

CLAIM 7.7. *We have* $|\mathcal{D}^i_{\text{main}}| \le \texttt{SIZE}_{\text{main}}$ *for all* $i$, *and*

$$|\mathcal{D}_{\text{aux}}| \le \mathbf{OPT}_{(j-i+1)V_{\text{bl}},m} - \texttt{SIZE}_{\text{main}} + (j-i+1)(m-1) \cdot 2^{-\kappa/2+2}.$$

We have already shown that each $|\mathcal{D}^i_{\text{main}}| \le \texttt{SIZE}_{\text{main}}$ above. To prove the bound on $|\mathcal{D}_{\text{aux}}|$, by inductive hypothesis, we know that $\mathcal{D}_{\text{aux},1}$ has size at most

$$\mathbf{OPT}_{(k-i+1)V_{\text{bl}},m_1} - (k-i+1)\texttt{SIZE}_{\text{main}} + (m_1-1)2^{-\kappa/2+2}$$

and $\mathcal{D}_{\text{aux},2}$ has size at most

$$\mathbf{OPT}_{(j-k)V_{\text{bl}},m_2} - (j-k)\texttt{SIZE}_{\text{main}} + (m_2-1)2^{-\kappa/2+2}.$$

To apply Proposition 4.9 in line 12, we need to approximate the data structure sizes. The following claim implies that the premises can be satisfied.

CLAIM 7.8. *Both* $s_1 := \mathbf{OPT}_{(k-i+1)V_{\text{bl}},m_1} - (k-i+1)\texttt{SIZE}_{\text{main}} + (m_1-1)2^{-\kappa/2+2}$ *and* $s_2 := \mathbf{OPT}_{(j-k)V_{\text{bl}},m_2} - (j-k)\texttt{SIZE}_{\text{main}} + (m_2-1)2^{-\kappa/2+2}$ *can be approximated with an additive error of at most* $2^{-\kappa}$ *in* $O(1)$ *time.*

Assuming Claim 7.8, Proposition 4.9 concatenates $\mathcal{D}_{\text{aux},1}$ and $\mathcal{D}_{\text{aux},2}$ into a data structure $\mathcal{D}'_{\text{aux}}$ of length at most

$$\begin{aligned} s'_{\text{aux},m_1} := {} & \mathbf{OPT}_{(k-i+1)V_{\text{bl}},m_1} + \mathbf{OPT}_{(j-k)V_{\text{bl}},m-m_1} \\ & - (j-i+1)\texttt{SIZE}_{\text{main}} + (m-2)2^{-\kappa/2+2} + 2^{-\kappa+4}. \end{aligned}$$

The following claim implies that the premises of Proposition 4.13 from line 13 can be satisfied, because $-(j-i+1)\texttt{SIZE}_{\text{main}} + (m-2)2^{-\kappa/2+2} + 2^{-\kappa+4}$ does not depend on $m_1$, and can be computed efficiently.

CLAIM 7.9. *For any* $V_1, V_2, m \ge 0$*, and* $0 \le l \le m$*,*

$$\sum_{i=0}^{l} 2^{\mathbf{OPT}_{V_1,i}+\mathbf{OPT}_{V_2,m-i}}$$

*can be approximated up to an additive error of at most* $2^{-\kappa-3} \cdot \sum_{i=0}^{m} 2^{\mathbf{OPT}_{V_1,i}+\mathbf{OPT}_{V_2,m-i}}$ *in* $O(\kappa^5)$ *time.*

The proofs of both claims are deferred to Appendix A. Assuming Claim 7.9, Proposition 4.13 fuses $m_1$ into $\mathcal{D}'_{\text{aux}}$, and obtains $\mathcal{D}_{\text{aux}}$ of length at most

$$\begin{aligned} & \lg\left(\sum_{m_1=0}^{m} 2^{s'_{\text{aux},m_1}}\right) + (m+1)\cdot 2^{-\kappa+4} \\ \le {} & \lg\left(\sum_{m_1=0}^{m} 2^{\mathbf{OPT}_{(k-i+1)V_{\text{bl}},m_1}+\mathbf{OPT}_{(j-k)V_{\text{bl}},m-m_1}}\right) \\ & - (j-i+1)\texttt{SIZE}_{\text{main}} + (m-2)2^{-\kappa/2+2} + 2^{-\kappa+4} + (m+1)\cdot 2^{-\kappa+4} \\ \le {} & \mathbf{OPT}_{(j-i+1)V_{\text{bl}},m} - (j-i+1)\texttt{SIZE}_{\text{main}} + (m-1)2^{-\kappa/2+2}. \end{aligned}$$

This proves Claim 7.7.

Thus, by induction, `dict_prep_rec` outputs $\mathcal{D}^1_{\text{main}}, \ldots, \mathcal{D}^{N_{\text{bl}}}_{\text{main}}$ of length $\texttt{SIZE}_{\text{main}}$ and $\mathcal{D}_{\text{aux}}$ of length

$$\mathbf{OPT}_{N_{\text{bl}}V_{\text{bl}},n} - N_{\text{bl}} \cdot \texttt{SIZE}_{\text{main}} + (n-1)2^{-\kappa/2+2}.$$

Finally, we apply Proposition 4.9 again to concatenate all $N_{\text{bl}}+1$ data structures. By storing approximations of sizes of $\mathcal{D}^i_{\text{main}}$ and $\mathcal{D}_{\text{aux}}$ in the lookup table, we obtain a data structure of length at most

$$N_{\text{bl}} \cdot \texttt{SIZE}_{\text{main}} + (\mathbf{OPT}_{N_{\text{bl}}V_{\text{bl}},n} - N_{\text{bl}} \cdot \texttt{SIZE}_{\text{main}} + (n-1)2^{-\kappa/2+2}) + N_{\text{bl}} \cdot 2^{-\kappa+3}$$

$\leq \mathbf{OPT}_{N_{\mathrm{bl}}V_{\mathrm{bl}},n} + n \cdot 2^{-\kappa/2+2}$.

This proves the space bound in Lemma 7.5. Next, we define 2-hq and show that it can be evaluated in constant time in expectation.

*2-PHM.* Let $h_i$ and $\overline{h}_i$ be the bijections obtained by Lemma 7.1 for blocks $\mathcal{B}^i_{\mathrm{pri}}$ and $\mathcal{B}^i_{\mathrm{sec}}$. We define the bijections $h$ and $\overline{h}$ as follows:

- for key $x \in S \cap (\mathcal{B}^i_{\mathrm{pri}} \cup \mathcal{B}^i_{\mathrm{sec}})$, if $h_i(x) < \kappa^{2c-3}$, let $h(x) := (i-1)\kappa^{2c-3} + h_i(x)$, otherwise, let $h(x) := (N_{\mathrm{bl}} - i) \cdot \kappa^{2c-3} + \sum_{j<i} |(\mathcal{B}^j_{\mathrm{pri}} \cup \mathcal{B}^j_{\mathrm{sec}}) \cap S| + h_i(x)$;
- for non-key $x \notin S$, if $\overline{h}_i(x) < V_{\mathrm{pri}} - \kappa^{2c-3} - \kappa^c$, let $\overline{h}(x) := (i-1)(V_{\mathrm{pri}} - \kappa^{2c-3} - \kappa^c) + \overline{h}_i(x)$, otherwise, let $\overline{h}(x) := (N_{\mathrm{bl}}-i)\cdot(V_{\mathrm{pri}}-\kappa^{2c-3}-\kappa^c)+\sum_{j<i} |(\mathcal{B}^j_{\mathrm{pri}}\cup\mathcal{B}^j_{\mathrm{sec}})\setminus S|+\overline{h}_i(x)$.

Essentially, the smallest hash values will be those with $h_i(x) < \kappa^{2c-3}$ or $\overline{h}_i(x) < V_{\mathrm{pri}} - \kappa^{2c-3} - \kappa^c$, ordered according to $i$ and $h_i(x)$ or $\overline{h}_i(x)$. Then the rest take larger values ordered according to $i$ and $h_i(x)$ or $\overline{h}_i(x)$ (note that $h(x)$ and $\overline{h}(x)$ depend on the number of keys in the first $i-1$ block pairs, which will be recovered by the query algorithm from $\mathcal{D}_{\mathrm{aux}}$). By definition, they are both bijections. Finally, let

$$\text{2-hq}(x) := \begin{cases} (0, h(x)) & \text{if } x \in S, \\ (1, \overline{h}(x)) & \text{if } x \notin S. \end{cases}$$

*Lookup tables.* We store the following information in the lookup table.

**lookup table** `tbl`:
1. $\texttt{tableBlk}_{V_{\mathrm{pri}},V_{\mathrm{sec}}}$ from Lemma 7.1
2. the lookup table for line 6 from Proposition 4.14 for all valid values $m_{\mathrm{pri}}$ and $m_{\mathrm{sec}}$
3. approximated value of $\texttt{SIZE}_{\mathrm{main}}$ and the (final) size of $\mathcal{D}_{\mathrm{aux}}$, up to $O(\kappa)$ bits of precision

By Lemma 7.1, the lookup table size is $2^{\epsilon\kappa}$. Since $\kappa = O(\lg U)$ and $n \geq U^{1/12}$, by readjusting the constant $\epsilon$, the lookup table size is at most $n^\epsilon$.

*Query algorithm.* Now, we show how to answer 2-hq queries. Given a query $x \in [U]$, we first shift it according to $\Delta$, as we did at preprocessing, $x \mapsto (x + \Delta) \bmod U$. If $x$ is in a primary block, we query the corresponding main data structure. If the main data structure does not return the answer, or $x$ is not in a primary block, we recursively decode the corresponding auxiliary data structure, and run `qalgBlk`.

**query algorithm** $\texttt{qalgG}(U, n, x)$:
1. if $x$ is in the $i$-th primary block
2. apply Proposition 4.9 to decode $\mathcal{D}^i_{\mathrm{main}}$
3. if $(b, v) := \mathcal{D}^i_{\mathrm{main}}.\texttt{qalgBlk}_{\mathrm{main}}(V_{\mathrm{pri}}, x) \neq$ "unknown" (from Lemma 7.1)
4. if $b = 1$, return $(1, (i-1)\kappa^{2c-3} + v)$
5. if $b = 0$, return $(0, (i-1)(V_{\mathrm{pri}} - \kappa^{2c-3} - \kappa^c) + v)$
6. decode $\mathcal{D}_{\mathrm{aux}}$ and return $\mathcal{D}_{\mathrm{aux}}.\texttt{qalg\_rec}(1, N_{\mathrm{bl}}, 0, n, x)$

Since $V_{\mathrm{pri}}/V_{\mathrm{sec}} = O(\kappa^{c-4})$ and we randomly shifted the universe, $x$ is in a primary block with probability $1 - O(\kappa^{-c+4})$. Also, by Lemma 7.1, $\texttt{qalgBlk}_{\mathrm{main}}$ runs in constant time. It returns "unknown" with probability at most $O(\kappa^{-c+3})$ for a uniformly random $\mathcal{R}$, and returns 2-hq$(x)$ otherwise. Therefore, the probability that `qalgG` terminates before reaching the last line is $1 - O(\kappa^{-c+4})$. Since $\kappa = \Theta(\lg U)$, it computes 2-hq$(x)$ in constant time with probability $1 - O(\lg^{-c+4} U)$.

Next, we show how to implement $\texttt{qalg\_rec}(i, j, s, m, x)$, which takes as parameters

- $(i, j)$: a range of blocks,
- $s$: the total number of keys before block $i$,
- $m$, the total number of keys in blocks $i$ to $j$, and
- $x$, the element being queried.

We will prove that its worst-case running time is $O(\lg^7 U)$.

**query algorithm** `qalg_rec`$(i, j, s, m, x)$:

1. if $i = j$
2. apply Proposition 4.14 to decode $m_{\text{pri}}$ and $\mathcal{D}'_{\text{aux}}$
3. $(b, v) := (\mathcal{D}^i_{\text{main}}, \mathcal{D}'_{\text{aux}})$.`qalgBlk`$(V_{\text{pri}}, m_{\text{pri}}, V_{\text{sec}}, m - m_{\text{pri}}, x - (i-1)(V_{\text{pri}} + V_{\text{sec}}))$

(from Lemma 7.1)

4. if $b = 1$, return $(1, (N_{\text{bl}} - i) \cdot \kappa^{2c-3} + s + v)$
5. if $b = 0$, return $(0, (N_{\text{bl}} - i) \cdot (V_{\text{pri}} - \kappa^{2c-3} - \kappa^c) + ((i-1) \cdot V_{\text{bl}} - s) + v)$

(to be cont'd)

In the base case with only one block, we simply decode the value of $m_{\text{pri}}$ as well as the corresponding $\mathcal{D}'_{\text{aux}}$ from the $i$-th block pair. By running `qalgBlk` from Lemma 7.1 to query within the block pair, we compute 2-hq$(x)$ according to its definition in $O(\kappa^4)$ time.

6. $k := \lfloor (i + j)/2 \rfloor$
7. apply Proposition 4.13 and Claim 7.9 to decode $m_1$ and $\mathcal{D}'_{\text{aux}}$
8. apply Proposition 4.9 and Claim 7.8 to decode $\mathcal{D}_{\text{aux},1}$ and $\mathcal{D}_{\text{aux},2}$
9. if $x$ is in $i$-th, ..., $k$-th block pair
10. return $\mathcal{D}_{\text{aux},1}$.`qalg_rec`$(i, k, s, m_1, x)$
11. else
12. return $\mathcal{D}_{\text{aux},2}$.`qalg_rec`$(k + 1, j, s + m_1, m - m_1, x)$

In general, we decode $m_1$, the number of elements in the first half of the blocks. Then we decode the data structures for the two halves. Depending on where the query is, we recurse into one of the two data structures. Proposition 4.9, Proposition 4.13, Claim 7.8 and Claim 7.9 guarantee that the decoding takes $O(\kappa^6)$ time. The recursion has at most $O(\lg n) \le \kappa$ levels. Thus, the total running time of `qalg_rec` is at most $O(\kappa^7)$. This proves the claim on the query time, and hence, it proves Lemma 7.5. ∎

**7.4. At least one bad block pair.** Now, let us show how to handle sets with at least one bad block. We will show that the space usage for such sets is $\mathbf{OPT}_{U,n} - \Omega(\kappa^3)$.

LEMMA 7.10. *If $U$ is a multiple of $V_{\text{bl}}$, then there is a data structure with guarantees as in Theorem 7.3, for all sets with at least one bad block pair. Moreover, the size of the data structure is at most*

$$\mathbf{OPT}_{U,n} - \Omega(\kappa^3).$$

Note that storing all such sets using less than $\mathbf{OPT}_{U,n}$ space is possible, because by a Chernoff bound, at most $2^{-\Omega(\kappa^3)}$ fraction of the sets have at least one bad block pair. The optimal space for this case is only $\mathbf{OPT}_{U,n} - \Omega(\kappa^3)$.

*Proof.* The first $\lceil \lg N_{\text{bl}} \rceil$ bits are used to encode the number of bad block pairs $N_{\text{bad}}$. It turns out that the fraction of input sets with $N_{\text{bad}}$ bad pairs is $2^{-\Omega(\kappa^3 N_{\text{bad}})}$, as we mentioned in Section 2.1. By the argument there, we can afford to use $O(\kappa^3 N_{\text{bad}})$ extra bits.

The idea is to construct a mapping which maps all good block pairs to the first $N_{\text{bl}} - N_{\text{bad}}$ pairs, construct a data structure using the above algorithm for good blocks, and finally handle the bad pairs separately.

To construct such a mapping, observe that the following two numbers are equal:
(a) the number of *good* pairs among the last $N_{\text{bad}}$ pairs, and
(b) the number of *bad* pairs among the first $N_{\text{bl}} - N_{\text{bad}}$ pairs.
Hence, in the mapping, we map all the good pairs among the last $N_{\text{bad}}$ to all bad pairs among the first $N_{\text{bl}} - N_{\text{bad}}$. The good pairs in the first $N_{\text{bl}} - N_{\text{bad}}$ pairs will be mapped to themselves. To store such a mapping, we spend $O(N_{\text{bad}} \cdot \lg N_{\text{bl}})$ bits to store a hash table of all the bad pairs using the FKS hashing. Then we spend $O(N_{\text{bad}} \cdot \lg N_{\text{bl}})$ bits to store for each pair in the last $N_{\text{bad}}$ pairs, whether it is a good pair and if it is, which bad pair it will be mapped to. The mapping takes $O(N_{\text{bad}} \cdot \lg N_{\text{bl}})$ bits to store in total (which is much smaller than $\kappa^3 N_{\text{bad}}$). It takes constant time to evaluate.

This mapping maps all good pairs to the first $N_{\mathrm{bl}} - N_{\mathrm{bad}}$ pairs. Then we apply preprocessing algorithm `dict_prep_rec` from Lemma 7.5 for good pairs to construct a data structure using

$$\mathbf{OPT}_{(N_{\mathrm{bl}}-N_{\mathrm{bad}})V_{\mathrm{bl}},n-n_{\mathrm{bad}}} + (n - n_{\mathrm{bad}}) \cdot 2^{-\kappa/2+2} \leq \left\lceil \mathbf{OPT}_{(N_{\mathrm{bl}}-N_{\mathrm{bad}})V_{\mathrm{bl}},n-n_{\mathrm{bad}}} \right\rceil + 1$$

bits, where $n_{\mathrm{bad}}$ is the number of keys in the bad block pairs.

Next, the preprocessing algorithm constructs data structures for the bad pairs. Consider a bad pair with $m_{\mathrm{pri}}$ keys in the primary block and $m_{\mathrm{sec}}$ keys in the secondary block. Thus, either $m_{\mathrm{pri}} \notin [\kappa^{2c-3} + \kappa^c/3, \kappa^{2c-3} + 2\kappa^c/3]$, or $m_{\mathrm{sec}} \notin [\kappa^{c+1}, 3\kappa^{c+1}]$. We construct two separate data structures, one for the primary block and one for secondary block (note that it might be the case that the number of keys in the primary block is within the above range, but the block pair is bad due to the secondary block, or vice versa, we still construct two separate data structures for *both* of them using the following argument). It turns out that if the number of keys in the block is at most $\kappa^{O(1)}$, then there is a data structure using only $O(1)$ extra bits, answering queries in constant time.

LEMMA 7.11. *Let $c$ be any constant positive integer and $\epsilon$ be any positive constant. There is a preprocessing algorithm* `perfHashS`*, query algorithm* `qalgS` *and lookup tables* $\texttt{tableS}_{V,m}$ *of sizes $\tilde{O}(2^{\epsilon\kappa})$, such that for any $V \leq 2^{\kappa/2}$ and $m \leq \kappa^c$, given a set $S \subset [V]$ of $m$ keys,* `perfHashS` *preprocesses $S$ into a data structure of size at most*

$$\mathbf{OPT}_{V,m} + (m-1) \cdot 2^{-\kappa/2+1},$$

*such that it defines a bijection $h$ between $S$ and $[m]$ and a bijection $\overline{h}$ between $[V] \setminus S$ and $[V-m]$. Given any $x \in [V]$,* `qalgS` *answers* 2-hq$(x)$ *in constant time, by accessing the data structure and* $\texttt{tableS}_{V,m}$.

In particular, the size is at most $\mathbf{OPT}_{V,m} + O(1)$. The lemma is an immediate corollary of Lemma 8.1 in Section 8.1 (see Remark 8.2).

On the other hand, sets that have a block with more than $\kappa^{3c}$ keys are even more rare. By a Chernoff bound, we can estimate that the fraction of sets with at least one block with $m > \kappa^{3c}$ keys is at most $2^{-\Omega(m \lg m)}$. This suggests that for every (bad) block with $m > \kappa^{3c}$ keys, we can afford to spend $O(m \lg m)$ extra bits. A simple modification to [21] gives such a data structure.

LEMMA 7.12. *Given a set $S \subset [V]$ of $m$ keys, there is a data structure of size*

$$\mathbf{OPT}_{V,m} + O(m + \lg\lg V),$$

*such that it defines a bijection $h$ between $S$ and $[m]$ and a bijection $\overline{h}$ between $[V] \setminus S$ and $[V-m]$. It supports* 2-hq *queries in constant time.*

Note that $\lg\lg V \leq \lg \kappa$, and $m \geq \kappa^{3c}$. The number of extra bits is simply $O(m)$. We prove this lemma in Appendix B.

For each bad block pair, we write down the two numbers $m_{\mathrm{pri}}$ and $m_{\mathrm{sec}}$ using $O(\lg n)$ bits. Then if $m_{\mathrm{pri}} \leq \kappa^{3c}$, we apply Lemma 7.11, and obtain a data structure with

$$\mathbf{OPT}_{V_{\mathrm{pri}},m_{\mathrm{pri}}} + O(1)$$

bits. If $m_{\mathrm{pri}} > \kappa^{3c}$, we apply Lemma 7.12, and obtain a data structure with

$$\mathbf{OPT}_{V_{\mathrm{pri}},m_{\mathrm{pri}}} + O(m_{\mathrm{pri}})$$

bits. Likewise for the secondary block, we obtain a data structure with

$$\mathbf{OPT}_{V_{\mathrm{sec}},m_{\mathrm{sec}}} + O(1)$$

bits if $m_{\mathrm{sec}} \le \kappa^{3c}$, and

$$\mathbf{OPT}_{V_{\mathrm{sec}},m_{\mathrm{sec}}} + O(m_{\mathrm{sec}})$$

bits if $m_{\mathrm{sec}} > \kappa^{3c}$.

Finally, we concatenate all data structures for bad pairs (which all have integer lengths). For each bad pair, we further store a pointer pointing to its corresponding data structure, as well as the total number of keys in all *bad* blocks prior to it (which helps us compute the hash values).

*Space usage.* Let us first bound the space usage for the data structures for a bad block pair. Consider the $i$-th bad block pair, suppose it has $m_{\mathrm{pri},i}$ keys in the primary block and $m_{\mathrm{sec},i}$ keys in the secondary block. Then note that we have

$$\lg \binom{V_{\mathrm{pri}}}{m_{\mathrm{pri},i}} = \lg \frac{V_{\mathrm{pri}}!}{m_{\mathrm{pri},i}!(V_{\mathrm{pri}} - m_{\mathrm{pri},i})!}$$

which by Stirling's formula, is at most

$$\begin{aligned} &\le V_{\mathrm{pri}} \lg \frac{V_{\mathrm{pri}}}{e} - m_{\mathrm{pri},i} \lg \frac{m_{\mathrm{pri},i}}{e} - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg \frac{V_{\mathrm{pri}} - m_{\mathrm{pri},i}}{e} + O(\lg V_{\mathrm{pri}}) \\ &= V_{\mathrm{pri}} \lg V_{\mathrm{pri}} - m_{\mathrm{pri},i} \lg m_{\mathrm{pri},i} - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) + O(\lg V_{\mathrm{pri}}). \end{aligned} \tag{7.3}$$

In the following, we are going to compare (7.3) with

$$V_{\mathrm{pri}} \lg V_{\mathrm{pri}} - m_{\mathrm{pri},i} \lg \overline{m_{\mathrm{pri}}} - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg (V_{\mathrm{pri}} - \overline{m_{\mathrm{pri}}}) + O(\lg V_{\mathrm{pri}}), \tag{7.4}$$

where $\overline{m_{\mathrm{pri}}} = \kappa^{2c-3} + \kappa^c/2$ is the average number of keys in a primary block. First observe that $f(x) = m \lg x + (V-m)\lg(V-x)$ achieves its maximum at $x = m$, thus, (7.3) $\le$ (7.4). On the other hand, if $m_{\mathrm{pri},i} \notin [\kappa^{2c-3} + \kappa^c/3, \kappa^{2c-3} + 2\kappa^c/3]$, i.e., $m_{\mathrm{pri},i}$ is far from $\overline{m_{\mathrm{pri}}}$, (7.3) is even smaller.

CLAIM 7.13. *We have* (7.4) $\ge$ (7.3). *Moreover, if* $m_{\mathrm{pri},i} \notin [\kappa^{2c-3} + \kappa^c/3, \kappa^{2c-3} + 2\kappa^c/3]$,

$$(7.4) - (7.3) \begin{cases} = \Omega(\kappa^3) & m_{\mathrm{pri},i} \le \kappa^{3c}, \\ \ge m_{\mathrm{pri},i} \lg \kappa & m_{\mathrm{pri},i} > \kappa^{3c}. \end{cases}$$

*Proof.*

$$\begin{aligned} &(7.4) - (7.3) \\ &= -m_{\mathrm{pri},i} \lg \frac{\overline{m_{\mathrm{pri}}}}{m_{\mathrm{pri},i}} - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg \frac{V_{\mathrm{pri}} - \overline{m_{\mathrm{pri}}}}{V_{\mathrm{pri}} - m_{\mathrm{pri},i}} \\ &= -m_{\mathrm{pri},i} \lg \left(1 + \frac{\overline{m_{\mathrm{pri}}} - m_{\mathrm{pri},i}}{m_{\mathrm{pri},i}}\right) - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg \left(1 + \frac{m_{\mathrm{pri},i} - \overline{m_{\mathrm{pri}}}}{V_{\mathrm{pri}} - m_{\mathrm{pri},i}}\right). \end{aligned}$$

By the facts that $\ln(1+x) \le x$ for $x > -1$, $\ln(1+x) \le x - \frac{1}{4}x^2$ for $|x| \le 1$, and $\ln(1+x) \le 3x/4$ for $x > 1$, when $m_{\mathrm{pri},i} \le \overline{m_{\mathrm{pri}}}/2$, we have

$$\begin{aligned} &(7.4) - (7.3) \\ &= -m_{\mathrm{pri},i} \lg \left(1 + \frac{\overline{m_{\mathrm{pri}}} - m_{\mathrm{pri},i}}{m_{\mathrm{pri},i}}\right) - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg \left(1 + \frac{m_{\mathrm{pri},i} - \overline{m_{\mathrm{pri}}}}{V_{\mathrm{pri}} - m_{\mathrm{pri},i}}\right) \end{aligned}$$

$$
\begin{aligned}
&\geq -m_{\mathrm{pri},i} \cdot \frac{3}{4} \cdot \frac{\overline{m_{\mathrm{pri}}} - m_{\mathrm{pri},i}}{m_{\mathrm{pri},i}} \lg e - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \cdot \frac{m_{\mathrm{pri},i} - \overline{m_{\mathrm{pri}}}}{V_{\mathrm{pri}} - m_{\mathrm{pri},i}} \lg e \\
&= \frac{\lg e}{4} (\overline{m_{\mathrm{pri}}} - m_{\mathrm{pri},i}) \\
&= \Omega(\kappa^{2c-3}).
\end{aligned}
$$

If $m_{\mathrm{pri},i} \geq \overline{m_{\mathrm{pri}}}/2$ and $m_{\mathrm{pri},i} \leq \kappa^{3c}$, we have

$$
\begin{aligned}
&(7.4) - (7.3) \\
&= -m_{\mathrm{pri},i} \lg \left(1 + \frac{\overline{m_{\mathrm{pri}}} - m_{\mathrm{pri},i}}{m_{\mathrm{pri},i}}\right) - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg \left(1 + \frac{m_{\mathrm{pri},i} - \overline{m_{\mathrm{pri}}}}{V_{\mathrm{pri}} - m_{\mathrm{pri},i}}\right) \\
&\geq -m_{\mathrm{pri},i} \left(\frac{\overline{m_{\mathrm{pri}}} - m_{\mathrm{pri},i}}{m_{\mathrm{pri},i}} - \frac{1}{4} \left(\frac{\overline{m_{\mathrm{pri}}} - m_{\mathrm{pri},i}}{m_{\mathrm{pri},i}}\right)^2\right) \lg e \\
&\quad - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \cdot \frac{m_{\mathrm{pri},i} - \overline{m_{\mathrm{pri}}}}{V_{\mathrm{pri}} - m_{\mathrm{pri},i}} \lg e \\
&= \frac{\lg e}{4} \cdot \frac{(m_{\mathrm{pri},i} - \overline{m_{\mathrm{pri}}})^2}{m_{\mathrm{pri},i}} \\
&= \Omega(\kappa^3),
\end{aligned}
$$

where the last inequality uses the fact that $m_{\mathrm{pri},i} \notin [\kappa^{2c-3} + \kappa^c/3, \kappa^{2c-3} + 2\kappa^c/3]$.

If $m_{\mathrm{pri},i} \geq \kappa^{3c}$, we have

$$
\begin{aligned}
&(7.4) - (7.3) \\
&= -m_{\mathrm{pri},i} \lg \left(1 + \frac{\overline{m_{\mathrm{pri}}} - m_{\mathrm{pri},i}}{m_{\mathrm{pri},i}}\right) - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg \left(1 + \frac{m_{\mathrm{pri},i} - \overline{m_{\mathrm{pri}}}}{V_{\mathrm{pri}} - m_{\mathrm{pri},i}}\right) \\
&\geq -m_{\mathrm{pri},i} \lg \frac{\overline{m_{\mathrm{pri}}}}{m_{\mathrm{pri},i}} - (m_{\mathrm{pri},i} - \overline{m_{\mathrm{pri}}}) \lg e \\
&\geq m_{\mathrm{pri},i} \lg \frac{m_{\mathrm{pri},i}}{e \cdot \overline{m_{\mathrm{pri}}}} \\
&\geq m_{\mathrm{pri},i} \lg \kappa.
\end{aligned}
$$

Combining the three cases, we obtain the claim. ∎

Since $V_{\mathrm{pri}} = U \cdot \frac{\overline{m_{\mathrm{pri}}}}{n} \pm O(1)$, we have

$$(7.4) = V_{\mathrm{pri}} \lg U - m_{\mathrm{pri},i} \lg n - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg(U - n) + O(\lg V_{\mathrm{pri}}).$$

Thus, Claim 7.13 implies that if $m_{\mathrm{pri},i} \notin [\kappa^{2c-3} + \kappa^c/3, \kappa^{2c-3} + 2\kappa^c/3]$, i.e., the primary block is bad, then the data structure size for the primary block is at most

$$V_{\mathrm{pri}} \lg U - m_{\mathrm{pri},i} \lg n - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg(U - n) - \Omega(\kappa^3)$$

(since $O(m_{\mathrm{pri},i}) \ll m_{\mathrm{pri},i} \lg \kappa$ and $O(1) \ll \kappa^3$), and otherwise, it is at most

$$V_{\mathrm{pri}} \lg U - m_{\mathrm{pri},i} \lg n - (V_{\mathrm{pri}} - m_{\mathrm{pri},i}) \lg(U - n) + O(\lg V_{\mathrm{pri}}).$$

By applying the same argument to the secondary blocks, we conclude that if $m_{\mathrm{sec},i} \notin [\kappa^{c+1}, 3\kappa^{c+1}]$, i.e., the secondary block is bad, then the data structure size for the secondary block is at most

$$V_{\mathrm{sec}} \lg U - m_{\mathrm{sec},i} \lg n + (V_{\mathrm{sec}} - m_{\mathrm{sec},i}) \lg(U - n) - \Omega(\kappa^3),$$

and otherwise, it is at most

$$V_{\mathrm{sec}} \lg U - m_{\mathrm{sec},i} \lg n + (V_{\mathrm{sec}} - m_{\mathrm{sec},i}) \lg(U-n) + O(\lg V_{\mathrm{pri}}).$$

Summing up the two bounds, and using the fact that at least one of the primary and secondary block is bad, the data structure size for the $i$-th bad block pair is at most

$$V_{\mathrm{bl}} \lg U - (m_{\mathrm{pri},i} + m_{\mathrm{sec},i}) \lg n - (V_{\mathrm{bl}} - m_{\mathrm{pri},i} - m_{\mathrm{sec},i}) \lg(U-n) - \Omega(\kappa^3),$$

since $\kappa^3 \gg \lg V_{\mathrm{pri}}$.

Now we sum up the sizes for all $N_{\mathrm{bad}}$ bad blocks, which in total contain $n_{\mathrm{bad}}$ keys. The total size is at most

$$N_{\mathrm{bad}} \cdot V_{\mathrm{bl}} \lg U - n_{\mathrm{bad}} \lg n - (N_{\mathrm{bad}} \cdot V_{\mathrm{bl}} - n_{\mathrm{bad}}) \lg(U-n) - \Omega(N_{\mathrm{bad}} \cdot \kappa^3).$$

Therefore, summing up the sizes of all parts of the data structure: the values of $N_{\mathrm{bad}}$ and $n_{\mathrm{bad}}$, the mapping from all good pairs to $[N_{\mathrm{bl}} - N_{\mathrm{bad}}]$, the data structure for the good blocks, all data structures for the bad blocks and the pointers to them, the total size is at most

$$\begin{aligned}
&\mathbf{OPT}_{(N_{\mathrm{bl}}-N_{\mathrm{bad}})V_{\mathrm{bl}},n-n_{\mathrm{bad}}} + O(N_{\mathrm{bad}} \lg N_{\mathrm{bl}}) + N_{\mathrm{bad}} \cdot V_{\mathrm{bl}} \lg U \\
&\quad - n_{\mathrm{bad}} \lg n - (N_{\mathrm{bad}} \cdot V_{\mathrm{bl}} - n_{\mathrm{bad}}) \lg(U-n) - \Omega(N_{\mathrm{bad}} \cdot \kappa^3) \\
&= (N_{\mathrm{bl}} - N_{\mathrm{bad}}) V_{\mathrm{bl}} \lg((N_{\mathrm{bl}} - N_{\mathrm{bad}})V_{\mathrm{bl}}) - (n - n_{\mathrm{bad}}) \lg(n - n_{\mathrm{bad}}) \\
&\quad - ((N_{\mathrm{bl}} - N_{\mathrm{bad}})V_{\mathrm{bl}} - (n - n_{\mathrm{bad}})) \lg((N_{\mathrm{bl}} - N_{\mathrm{bad}})V_{\mathrm{bl}} - (n - n_{\mathrm{bad}})) \\
&\quad + N_{\mathrm{bad}} \cdot V_{\mathrm{bl}} \lg U - n_{\mathrm{bad}} \lg n - (N_{\mathrm{bad}} \cdot V_{\mathrm{bl}} - n_{\mathrm{bad}}) \lg(U-n) - \Omega(N_{\mathrm{bad}} \cdot \kappa^3),
\end{aligned}$$

which by Claim 7.13 and the fact that $N_{\mathrm{bad}} \geq 1$, is at most

$$\begin{aligned}
&= (N_{\mathrm{bl}} - N_{\mathrm{bad}}) V_{\mathrm{bl}} \lg U - (n - n_{\mathrm{bad}}) \lg n - ((N_{\mathrm{bl}} - N_{\mathrm{bad}})V_{\mathrm{bl}} - (n - n_{\mathrm{bad}})) \lg(U-n) \\
&\quad + N_{\mathrm{bad}} \cdot V_{\mathrm{bl}} \lg U - n_{\mathrm{bad}} \lg n - (N_{\mathrm{bad}} \cdot V_{\mathrm{bl}} - n_{\mathrm{bad}}) \lg(U-n) - \Omega(\kappa^3), \\
&= U \lg U - n \lg n - (U-n) \lg(U-n) - \Omega(\kappa^3) \\
&\leq \lg \binom{U}{n} - \Omega(\kappa^3) \\
&= \mathbf{OPT}_{U,n} - \Omega(\kappa^3),
\end{aligned}$$

as we claimed.

*Hash functions.* For all $x$ in the good blocks, we simply use their hash value according to Lemma 7.5, for which $h$ takes values in $[n - n_{\mathrm{bad}}]$ and $\overline{h}$ takes values in $[V_{\mathrm{bl}} \cdot (N_{\mathrm{bl}} - N_{\mathrm{bad}}) - (n - n_{\mathrm{bad}})]$. For $x$ in the $i$-th bad pair with hash value $v$, let $s_i$ be the total number of keys in first $i-1$ bad pairs (which is explicitly stored in the data structure), then if $x \in S$, we set $h(x) := n - n_{\mathrm{bad}} + s_i + v$; if $x \notin S$, we set $\overline{h}(x) := V_{\mathrm{bl}} \cdot (N_{\mathrm{bl}} - N_{\mathrm{bad}} + (i-1)) - (n - n_{\mathrm{bad}} + s_i) + v$.

That is, all elements in good blocks take the smallest values, and elements in bad blocks take the rest according to the order of the blocks. By definition, they are both bijections. Similarly, let

$$\text{2-hq}(x) := \begin{cases} (0, h(x)) & \text{if } x \in S, \\ (1, \overline{h}(x)) & \text{if } x \notin S. \end{cases}$$

*Lookup tables.* We include the lookup table from the data structure for no bad blocks, as well as all tables $\texttt{tableS}_{V,m}$ from Lemma 7.11 for $V = V_{\mathrm{pri}}$ or $V = V_{\mathrm{sec}}$, and $1 \leq m \leq \kappa^{3c}$. The total lookup table size is $\tilde{O}(2^{\epsilon\kappa})$. It is at most $n^{\epsilon}$ by readjusting the constant $\epsilon$.

*Query algorithm.* Given a query $x$, suppose $x$ is in the $i$-th block pair. We first query the hash table to check if it is one of the bad pairs. If the block pair is bad, we follow the pointer and query the data structure for the primary block or the secondary block depending on which block $x$ is in. Its hash value can be computed according to the definition. It takes constant time.

If the block pair is good, we spend constant time to find out where the $i$-th block pair is mapped to, in the first $N_{\mathrm{bl}} - N_{\mathrm{bad}}$ pairs. Then we run `qalgG` for good blocks, which takes constant time in expectation. This completes the proof of Lemma 7.10. □

**7.5. Complete data structure for medium size sets.** Now, we are ready to prove Theorem 7.3.

*Proof of Theorem 7.3.* Consider the following preprocessing algorithm for general $U$ and $U^{1/12} \le n \le U/2$ ($U$ not necessarily a multiple of $V_{\mathrm{bl}}$). We first construct a data structure for the block pairs, and fuse the two cases (with or without bad blocks) together.

**preprocessing algorithm** `perfHash`$(U, n, S, \mathcal{R})$:
1. compute $V_{\mathrm{pri}}$, $V_{\mathrm{sec}}$ and $\kappa$
2. compute $N_{\mathrm{bl}} := U \operatorname{div} V_{\mathrm{bl}}$ and $V := U \bmod V_{\mathrm{bl}}$
3. divide the universe $[U]$ into $N_{\mathrm{bl}}$ block pairs and a last block of size $V$
4. if all $N_{\mathrm{bl}}$ block pairs are good
5. set $i := 0$, apply Lemma 7.5 on the $N_{\mathrm{bl}}$ block pairs, and obtain a data structure $\mathcal{D}_0$
6. else
7. set $i := 1$, apply Lemma 7.10 on the $N_{\mathrm{bl}}$ block pairs, and obtain a data structure $\mathcal{D}_1$
8. apply Proposition 4.14 to fuse $i$ into $\mathcal{D}_i$, and obtain a data structure $\mathcal{D}_{\mathrm{bl}}$

(to be cont'd)

Suppose there are $n_{\mathrm{bl}}$ keys in the first $N_{\mathrm{bl}}$ block pairs. By Lemma 7.5, $\mathcal{D}_0$ has length at most

$$\mathbf{OPT}_{N_{\mathrm{bl}}V_{\mathrm{bl}},n_{\mathrm{bl}}} + n_{\mathrm{bl}} \cdot 2^{-\kappa/2+2}.$$

By Lemma 7.10, $\mathcal{D}_1$ has length at most

$$\mathbf{OPT}_{N_{\mathrm{bl}}V_{\mathrm{bl}},n_{\mathrm{bl}}} - \Omega(\kappa^3).$$

Thus, by Proposition 4.14, $\mathcal{D}_{\mathrm{bl}}$ has length at most

$$\begin{aligned}
&\mathbf{OPT}_{N_{\mathrm{bl}}V_{\mathrm{bl}},n_{\mathrm{bl}}} + \lg(2^{n_{\mathrm{bl}}\cdot 2^{-\kappa/2+2}} + 2^{-\Omega(\kappa^3)}) + 2^{-\kappa+2} \\
\le\ &\mathbf{OPT}_{N_{\mathrm{bl}}V_{\mathrm{bl}},n_{\mathrm{bl}}} + \lg(1 + n_{\mathrm{bl}} \cdot 2^{-\kappa/2+2} + 2^{-\Omega(\kappa^3)}) + 2^{-\kappa+2} \\
\le\ &\mathbf{OPT}_{N_{\mathrm{bl}}V_{\mathrm{bl}},n_{\mathrm{bl}}} + n_{\mathrm{bl}} \cdot 2^{-\kappa/2+3} \\
=\ &\mathbf{OPT}_{U-V,n_{\mathrm{bl}}} + n_{\mathrm{bl}} \cdot 2^{-\kappa/2+3}.
\end{aligned}$$

Then we construct a separate data structure for the last block using Lemma 7.11 or Lemma 7.12 based on the number of keys in it.

9. if $n - n_{\mathrm{bl}} \le \kappa^{3c}$
10. construct $\mathcal{D}_{\mathrm{last}}$ for the last block using Lemma 7.11
11. apply Proposition 4.10 to concatenate $\mathcal{D}_{\mathrm{bl}}$ and $\mathcal{D}_{\mathrm{last}}$, and obtain $\mathcal{D}'$
12. let $n'_{\mathrm{bl}} = n_{\mathrm{bl}}$
13. else
14. construct $\mathcal{D}_{\mathrm{last}}$ for the last block using Lemma 7.12
15. spend $\lceil \lg n \rceil$ bits to store $n_{\mathrm{bl}}$,
16. round both $\mathcal{D}_{\mathrm{bl}}$ and $\mathcal{D}_{\mathrm{last}}$ to integral lengths and concatenate them
17. spend $\lceil \lg n \rceil$ bits to store a pointer to $\mathcal{D}_{\mathrm{last}}$, let the resulting data structure be $\mathcal{D}'$
18. let $n'_{\mathrm{bl}} = n - \kappa^{3c} - 1$
19. apply Proposition 4.14 to fuse the value of $n'_{\mathrm{bl}}$ into $\mathcal{D}'$ for $n'_{\mathrm{bl}} \in [n - \kappa^{3c} - 1, n]$, and obtain $\mathcal{D}$
20. return $\mathcal{D}$

We do not fuse the whole value of $n_{\mathrm{bl}}$ into $\mathcal{D}'$, as its range is large and Proposition 4.14 requires a large lookup table to do this. Instead, we only fuse its value if $n_{\mathrm{bl}} \geq n - \kappa^{3c}$, and otherwise only indicate that it is smaller than $n - \kappa^{3c}$ (by setting $n'_{\mathrm{bl}}$ to $n - \kappa^{3c} - 1$). This is fine because we spend extra $\lg n$ bits to explicitly store its value in this case. We will show in the following that the total space is close to the optimum.

There are $n - n_{\mathrm{bl}}$ keys in the last block. If $n - n_{\mathrm{bl}} \leq \kappa^{3c}$, $\mathcal{D}_{\mathrm{last}}$ has size at most

$$\mathbf{OPT}_{V,n-n_{\mathrm{bl}}} + (n - n_{\mathrm{bl}} - 1) \cdot 2^{-\kappa/2+1}.$$

In this case, the length of $\mathcal{D}'$ is at most

$$\begin{aligned}
&\mathbf{OPT}_{U-V,n_{\mathrm{bl}}} + \mathbf{OPT}_{V,n-n_{\mathrm{bl}}} + n_{\mathrm{bl}} \cdot 2^{-\kappa/2+3} + (n - n_{\mathrm{bl}} - 1) \cdot 2^{-\kappa/2+1} + 2^{-\kappa+2} \\
\leq &\mathbf{OPT}_{U-V,n_{\mathrm{bl}}} + \mathbf{OPT}_{V,n-n_{\mathrm{bl}}} + n \cdot 2^{-\kappa/2+3} \\
(7.5)\quad = &\lg \binom{U-V}{n_{\mathrm{bl}}} \binom{V}{n-n_{\mathrm{bl}}} + n \cdot 2^{-\kappa/2+3}.
\end{aligned}$$

If $n - n_{\mathrm{bl}} > \kappa^{3c}$, $\mathcal{D}_{\mathrm{last}}$ has size at most

$$\mathbf{OPT}_{V,n-n_{\mathrm{bl}}} + O(n - n_{\mathrm{bl}} + \lg n).$$

The length of $\mathcal{D}'$ is at most

$$\lg \binom{U-V}{n_{\mathrm{bl}}} \binom{V}{n-n_{\mathrm{bl}}} + O(n - n_{\mathrm{bl}} + \lg n). \tag{7.6}$$

By Stirling's formula, the first term is

$$\begin{aligned}
&\lg \binom{U-V}{n_{\mathrm{bl}}} \binom{V}{n-n_{\mathrm{bl}}} \\
&= \lg \frac{(U-V)!V!}{n_{\mathrm{bl}}!(U-V-n_{\mathrm{bl}})!(n-n_{\mathrm{bl}})!(V-n+n_{\mathrm{bl}})!} \\
&\leq (U-V)\lg(U-V) - n_{\mathrm{bl}} \lg n_{\mathrm{bl}} - (U-V-n_{\mathrm{bl}})\lg(U-V-n_{\mathrm{bl}}) \\
&\quad + V \lg V - (n-n_{\mathrm{bl}})\lg(n-n_{\mathrm{bl}}) - (V-n+n_{\mathrm{bl}})\lg(V-n+n_{\mathrm{bl}}) + O(\lg U).
\end{aligned}$$

By the fact that $f(x) = m \lg x + (V-m)\lg(V-x)$ is maximized at $x = m$, it is at most

$$\begin{aligned}
&(U-V)\lg(U-V) - n_{\mathrm{bl}} \lg \frac{(U-V)n}{U} - (U-V-n_{\mathrm{bl}})\lg(U-V-\frac{(U-V)n}{U}) \\
&\quad + V \lg V - (n-n_{\mathrm{bl}})\lg(n-n_{\mathrm{bl}}) - (V-n+n_{\mathrm{bl}})\lg(V-n+n_{\mathrm{bl}}) + O(\lg U) \\
&= (U-V)\lg U - n_{\mathrm{bl}} \lg n - (U-V-n_{\mathrm{bl}})\lg(U-n) \\
&\quad + V \lg V - (n-n_{\mathrm{bl}}) \lg \frac{Vn}{U} - (V-n+n_{\mathrm{bl}}) \left(V - \frac{Vn}{U}\right) \\
&\quad - (n-n_{\mathrm{bl}}) \lg \frac{U(n-n_{\mathrm{bl}})}{Vn} - (V-n+n_{\mathrm{bl}}) \lg \frac{V-n+n_{\mathrm{bl}}}{V-\frac{Vn}{U}} + O(\lg U) \\
&= U \lg U - n \lg n - (U-n)\lg(U-n) \\
&\quad - (n-n_{\mathrm{bl}}) \lg \frac{U(n-n_{\mathrm{bl}})}{Vn} + (V-n+n_{\mathrm{bl}}) \lg \left(1 + \frac{n-n_{\mathrm{bl}}-\frac{Vn}{U}}{V-n+n_{\mathrm{bl}}}\right) + O(\lg U).
\end{aligned}$$

By Stirling's formula and the fact that $\ln(1+x) \le x$, it is at most

$$\lg \binom{U}{n} - (n - n_{\mathrm{bl}}) \lg \frac{U(n - n_{\mathrm{bl}})}{Vn} + (n - n_{\mathrm{bl}}) \lg e + O(\lg U)$$

which by the fact that $n - n_{\mathrm{bl}} > \kappa^{3c}$, is

$$\le \lg \binom{U}{n} - (n - n_{\mathrm{bl}}) \lg \frac{U\kappa^{3c}}{eVn} + O(\lg U)$$

which by the fact that $U/(eVn) \ge \kappa^{-2c}$ and $c \ge 1$, is

$$\le \lg \binom{U}{n} - (n - n_{\mathrm{bl}}) \lg \kappa + O(\lg U).$$

Therefore, when $n - n_{\mathrm{bl}} > \kappa^{3c}$, the size of $\mathcal{D}'$ is at most

$$\begin{aligned}(7.6) &\le \lg \binom{U}{n} - \Omega((n - n_{\mathrm{bl}}) \lg \kappa) + O(\lg U + \lg n) \\ &\le \mathbf{OPT}_{U,n} - \Omega(\kappa^3).\end{aligned}$$

Finally together with Equation (7.5), by Proposition 4.14, the size of $\mathcal{D}$ is at most

$$\begin{aligned}&\lg \left( 2^{\mathbf{OPT}_{U,n} - \Omega(\kappa^3)} + \sum_{n_{\mathrm{bl}} = n - \kappa^{3c}}^{n} \binom{U - V}{n_{\mathrm{bl}}} \binom{V}{n - n_{\mathrm{bl}}} \cdot 2^{n \cdot 2^{-\kappa/2+3}} \right) + n \cdot 2^{-\kappa+2} \\ \le &\lg \left( 2^{\mathbf{OPT}_{U,n} - \Omega(\kappa^3)} + \sum_{n_{\mathrm{bl}} = 0}^{n} \binom{U - V}{n_{\mathrm{bl}}} \binom{V}{n - n_{\mathrm{bl}}} \cdot 2^{n \cdot 2^{-\kappa/2+3}} \right) + n \cdot 2^{-\kappa+2} \\ = &\lg \left( 2^{\mathbf{OPT}_{U,n} - \Omega(\kappa^3)} + \binom{U}{n} \cdot 2^{n \cdot 2^{-\kappa/2+3}} \right) + n \cdot 2^{-\kappa+2} \\ = &\mathbf{OPT}_{U,n} + \lg \left( 2^{-\Omega(\kappa^3)} + 2^{n \cdot 2^{-\kappa/2+3}} \right) + n \cdot 2^{-\kappa+2} \\ \le &\mathbf{OPT}_{U,n} + \lg \left( 2^{-\Omega(\kappa^3)} + 1 + n \cdot 2^{-\kappa/2+3} \right) + n \cdot 2^{-\kappa+2} \\ \le &\mathbf{OPT}_{U,n} + 2^{-\Omega(\kappa^3)} + n \cdot 2^{-\kappa/2+4} + n \cdot 2^{-\kappa+2} \\ \le &\mathbf{OPT}_{U,n} + U^{-1}.\end{aligned}$$

This proves that `perfHash` outputs a data structure with $U^{-1}$ bits of redundancy. Next, we describe the 2-PHM that the data structure defines.

*Hash functions.* For all $x$ in the first $N_{\mathrm{bl}}$ block pairs, we simply use their hash values defined by $\mathcal{D}_{\mathrm{bl}}$ (from Lemma 7.5 or Lemma 7.10), for which, $h$ takes values from $[n_{\mathrm{bl}}]$ and $\overline{h}$ takes values from $[V_{\mathrm{bl}} \cdot N_{\mathrm{bl}} - n_{\mathrm{bl}}]$. For all $x$ in the last block, let $v$ be its hash value defined by $\mathcal{D}_{\mathrm{last}}$. If $x \in S$, let $h(x) := n_{\mathrm{bl}} + v$; if $x \notin S$, let $\overline{h}(x) := V_{\mathrm{bl}} \cdot N_{\mathrm{bl}} - n_{\mathrm{bl}} + v$. By definition, $h$ and $\overline{h}$ are both bijections.

*Lookup table.* We include the lookup tables in Section 7.3 and in Section 7.4, which both have size $n^{\epsilon}$. Then we include the lookup tables needed by Proposition 4.10 and Proposition 4.14 in line 8, 11 and 19. The total size is $n^{\epsilon}$.

*Query algorithm.* Given a query $x$, we decode all the components, and query the part based on the value of $x$.

**query algorithm** $\texttt{qAlg}(x)$:
1. compute $V_{\mathrm{pri}}, V_{\mathrm{sec}}, N_{\mathrm{bl}}$ and $V$
2. apply Proposition 4.14 to recover $n'_{\mathrm{bl}}$ and decode $\mathcal{D}'$ from $\mathcal{D}$
3. if $n'_{\mathrm{bl}} > n - \kappa^{3c} - 1$
4. let $n_{\mathrm{bl}} := n'_{\mathrm{bl}}$
5. apply Proposition 4.10 to decode $\mathcal{D}_{\mathrm{bl}}$ and $\mathcal{D}_{\mathrm{last}}$
6. else
7. recover $n_{\mathrm{bl}}$ from $\mathcal{D}'$
8. decode $\mathcal{D}_{\mathrm{bl}}$ and $\mathcal{D}_{\mathrm{last}}$
9. if $x \geq U - V$
10. query $x$ in $\mathcal{D}_{\mathrm{last}}$, and obtain $(b, v)$
11. if $b = 1$, then return $(1, n_{\mathrm{bl}} + v)$
12. if $b = 0$, then return $(0, V_{\mathrm{bl}} \cdot N_{\mathrm{bl}} - n_{\mathrm{bl}} + v)$
13. else
14. apply Proposition 4.14 to recover $i$ and decode $\mathcal{D}_i$ from $\mathcal{D}_{\mathrm{bl}}$
15. query $x$ in $\mathcal{D}_b$ using the corresponding query algorithm, and return the outcome

All subroutines run in expected constant time, the overall expected query time is constant. This completes the proof of Theorem 7.3. ∎

**8. Data Structure Pair for One Block Pair.** In this section, we prove our main technical lemma (Lemma 7.1), which constructs a pair of data structures for a pair of blocks. In Section 8.1, we construct a succinct rank data structure for sets of size $\kappa^{O(1)}$ with constant query time. Then we study data interpretation and show that a string can be represented as a set in Section 8.2. Finally, we prove Lemma 7.1 in Section 8.3.

**8.1. Improved rank data structure for small sets.** We first improve the rank data structure of Pătraşcu [23]. We show that if a block has $\kappa^c$ keys, then there is a rank data structure with constant query time and negligible extra space. Recall that the `rank` problem asks to preprocess a set $S$ of $m$ keys into a data structure, supporting

- $\texttt{rank}_S(x)$: return the number of keys that are at most $x$.

In particular, by computing both $\texttt{rank}_S(x)$ and $\texttt{rank}_S(x-1)$, one can decide if $x \in S$. Formally, we will prove the following lemma.

LEMMA 8.1. *Let $c$ be any constant positive integer and $\epsilon$ be any positive constant. There is a preprocessing algorithm* `prepRank`*, query algorithm* `qAlgRank` *and lookup tables* $\texttt{tableRank}_{V,m}$ *of sizes $\tilde{O}(2^{\epsilon\kappa})$, such that for any integers $V \leq 2^{\kappa/2}, m \leq \kappa^c$, given a set $S \subset [V]$ of size $m$,* $\texttt{prepRank}(V, m, S)$ *outputs a data structure $\mathcal{D}$ of length*

$$\lg \binom{V}{m} + (m-1) \cdot 2^{-\kappa/2}.$$

*Given $x \in [V]$,* $\texttt{qAlgRank}(V, m, x)$ *computes* $\texttt{rank}_S(x)$ *in constant time, by accessing $\mathcal{D}$ and* $\texttt{tableRank}_{V,m}$*. The algorithms run on a random access machine with word size $w = \Omega(\kappa)$.*

*Remark* 8.2. Note that a rank data structure easily defines hash functions that map the keys to $[m]$, and the non-keys to $[V-m]$: For each key $x$, we set $h(x) := \texttt{rank}(x) - 1$; for each non-key $x$, we set $\overline{h}(x) := x - \texttt{rank}(x) - 1$. Hence, Lemma 7.11 is an immediate corollary.

To prove the lemma, we first show that the fusion trees [12] can be implemented succinctly. This gives us a data structure for small sets with a sublinear, although large, redundancy.

LEMMA 8.3. *Let $c$ be any constant positive integer and $\epsilon$ be any positive constant. There is a preprocessing algorithm* `prepRankL`*, a query algorithm* `qAlgRankL` *and lookup tables*

$\texttt{tableRankL}_{V,m}$ *of sizes* $2^{\epsilon\kappa}$ *such that for any integers* $V, m$ *such that* $V \leq 2^\kappa$ *and* $m \leq \kappa^c$*, given a set* $S \subseteq [V]$ *of size* $m$*,* `prepRankL` *preprocesses it into a data structure using*

$$\lg \binom{V}{m} + \frac{1}{8} m \lg \kappa$$

*bits of space. Given any* $x \in [V]$*,* `qAlgRankL` *compute* $\texttt{rank}_S(x)$ *in constant time, by accessing the data structure and* $\texttt{tableRankL}_{V,m}$*. The algorithms run on a random access machine with word size* $w = \Omega(\kappa)$*.*

Since the main ideas are similar, we may omit the proof of a few claims in the construction, and refer the readers to the original fusion trees for details ([12]).

*Proof.* (sketch) Let $S = \{y_1, \ldots, y_m\}$ and $y_1 < y_2 < \cdots < y_m$. Let us first show how to construct such a data structure using

$$m\lceil \lg V \rceil + m \lceil \lg \kappa \rceil$$

bits when $m \leq \kappa^{1/4}$. We view each $y_i$ as a $\lceil \lg V \rceil$-bit binary string, and consider the first bit where $y_i$ and $y_{i+1}$ differ, for every $i = 1, \ldots, m-1$. Let $W$ be this set of bits, i.e., $j \in W$ if and only there exists some $i$ such that $j$-th bit is the first bit where $y_i$ and $y_{i+1}$ differ. Then $|W| \leq m - 1$. Similar to fusion trees, let $\texttt{sketch}(y)$ be $y$ restricted to $W$. Observe that we must have $\texttt{sketch}(y_1) < \texttt{sketch}(y_2) < \cdots < \texttt{sketch}(y_m)$.

The data structure first stores $W$ using $m\lceil \lg \kappa \rceil$ bits. Then it stores $\texttt{sketch}(y_1)$, ..., $\texttt{sketch}(y_m)$. Finally, the data structure stores the remaining bits of each $y_i$, for $i = 1, \ldots, m$ and from the top bits to the low bits. It is clear that the data structure occupies $m\lceil \lg V \rceil + m\lceil \lg \kappa \rceil$ bits of space.

To answer a query $x \in [V]$, `qAlgRankL` first breaks $x$ into $\texttt{sketch}(x)$ and the remaining bits. That is, it generates two strings: $x$ restricted to $W$ (a $|W|$-bit string), and the remaining bits (a $(\lceil \lg V \rceil - |W|)$-bit string). It can be done in constant time using a lookup table of size $2^{O(\epsilon\kappa)}$, e.g., we divide the bits of $x$ into chunks of length $\epsilon\kappa$, and store in $\texttt{tableRankL}_{V,m}$ for each chunk, every possible set $W$ and every possible assignment to the bits of $x$ in the chunk, their contribution to $\texttt{sketch}(x)$ and the remaining bits (note that there are only $2^{o(\kappa)}$ different sets $W$). Summing over all chunks gives us $\texttt{sketch}(x)$ and the remaining bits. The query algorithm then finds the unique $i$ such that $\texttt{sketch}(y_i) \leq \texttt{sketch}(x) < \texttt{sketch}(y_{i+1})$. This can be done by using a lookup table of size at most $2^{(m+1)|W|} \leq 2^{\kappa^{1/2}}$, since $(\texttt{sketch}(y_1), \ldots, \texttt{sketch}(y_m))$ has only $m|W|$ bits, and $\texttt{sketch}(x)$ has $|W|$ bits. However, we might not necessarily have $y_i \leq x < y_{i+1}$, but similar to the arguments in fusion trees, $x$ has the longest common prefix (LCP) with either $y_i$ or $y_{i+1}$ (among all $y \in S$). `qAlgRankL` next computes the LCP between $x$ and $y_i$ and the LCP between $x$ and $y_{i+1}$. Both can be done in constant time, since to compute the LCP between $x$ and $y_i$, it suffices to compute the LCP between $\texttt{sketch}(x)$ and $\texttt{sketch}(y_i)$ and the LCP between their remaining bits. Suppose $x$ and $y_{i^*}$ have a longer LCP ($i^* = i$ or $i+1$). If $x = y_{i^*}$, then $\texttt{rank}_S(x) = i^*$. Otherwise, let their common prefix be $x'$. If $x > y_{i^*}$, then let $j$ be the unique index such that $\texttt{sketch}(y_j) \leq \texttt{sketch}(x'111\cdots11) < \texttt{sketch}(y_{j+1})$. The argument from fusion trees shows that we must have $y_j < x < y_{j+1}$, i.e., $\texttt{rank}_S(x) = j$. Likewise, if $x < y_{i^*}$, then let $j$ be the unique index such that $\texttt{sketch}(y_j) < \texttt{sketch}(x'000\cdots00) \leq \texttt{sketch}(y_{j+1})$. We must have $y_j < x < y_{j+1}$. By computing the value of $j$ using the lookup table again, we find the number of elements in $S$ that is at most $x$. Note that this data structure also allows us to retrieve each $y_i$ in constant time.

Next, we show that the above data structure generalizes to any $m \le \kappa^c$, and uses space

$$m(\lg V + (c+3)\lg\kappa) \le \lg \binom{V}{m} + (2c+3)m\lg\kappa.$$

When $m > \kappa^{1/4}$, let $B = \lfloor \kappa^{1/4} \rfloor$, we take $B$ evenly spaced elements from $S$, i.e., $y_{\lceil im/B \rceil}$ for $i = 1, \ldots, B$. Denote the set of these $B$ elements by $S' = \{y'_1, \ldots, y'_B\}$, where $y'_i = y_{\lceil im/B \rceil}$. We apply the above data structure to $S'$, using space

$$B\lceil \lg V \rceil + B\lceil \lg \kappa \rceil < B(\lg V + \lg\kappa + 2).$$

We recurse on all $B$ subsets between elements in $S'$, where the $i$-th subset has $\lceil im/B \rceil - \lceil (i-1)m/B \rceil - 1$ elements. Then the final data structure stores

- the data structure for $S'$;
- $B$ data structures for all subsets between elements in $S'$;
- an array of $B$ pointers, pointing to the starting locations of the above $B$ data structures.

We assign $(c + 3/2)\lg\kappa$ bits to each pointer.

Suppose for each subset, we are able to (recursively) construct a data structure using

$$(\lceil im/B \rceil - \lceil (i-1)m/B \rceil - 1)(\lg V + (c+3)\lg\kappa)$$

bits of space. The total space usage is

$$\begin{aligned} &B(\lg V + \lg\kappa + 2) + (m-B)(\lg V + (c+3)\lg\kappa) + B(c+3/2)\lg\kappa \\ &\quad \le m(\lg V + (c+3)\lg\kappa). \end{aligned}$$

On the other hand, assigning $(c + 3/2)\lg\kappa$ bits to each pointer is sufficient, because

$$\lg\left(m(\lg V + (c+3)\lg\kappa)\right) \le \lg\left(m\kappa + (c+3)m\lg\kappa\right) \le (c+1)\lg\kappa + 1.$$

To answer query $x$, we first query the data structure for $S'$, and find the $i$ such that $y'_i \le x < y'_{i+1}$. Then we recurse into the $i$-th subset. The query time is constant, because the size of the set reduces by a factor of $B = \Theta(\kappa^{1/4})$ each time. Note that for any given $i$, this data structure can also return $y_i$ in constant time.

Finally, we show that the redundancy $(2c+3)m\lg\kappa$ can be reduced to $\frac{1}{8}m\lg\kappa$. To this end, let $S'$ be the subset of $S$ with gap $16(2c+3)$, i.e., $S' = \{y'_1, y'_2, \ldots\}$ such that $y'_i = y_{16(2c+3)\cdot i}$. Then $|S'| = \lfloor \frac{m}{16(2c+3)} \rfloor$. We construct a data structure for $S'$ using space

$$|S'|(\lg V + (c+3)\lg\kappa).$$

Naturally, $S'$ partitions $S$ into chunks of $16(2c+3) - 1$ elements. We simply write them down using

$$(16(2c+3) - 1)\lceil \lg(y'_{i+1} - y'_i - 1) \rceil$$

bits for chunk $i$. The final data structure consists of

1. the data structure for $S'$,
2. all other elements in $S$ encoded as above,
3. $|S'| + 1$ pointers to each chunk.

We assign $\lceil (c+3/2)\lg\kappa \rceil$ bits to each pointer. By the concavity of $\lg x$, the total space usage is

$$\begin{aligned}
&|S'|(\lg V + (c+3)\lg\kappa) + \sum_i (16(2c+3)-1)\lceil \lg(y'_{i+1} - y'_i - 1)\rceil \\
&\qquad + (|S'|+1)\lceil (c+3/2)\lg\kappa\rceil \\
&\le |S'|\lg\frac{V}{m} + |S'|(3c+5)\lg\kappa + \sum_i (16(2c+3)-1)\lg\frac{V}{|S'|+1} + m \\
&\le |S'|\lg\frac{V}{m} + \frac{(3c+5)m}{16(2c+3)}\lg\kappa + \sum_i (16(2c+3)-1)\lg\frac{V}{m} + O(m) \\
&\le m\lg\frac{V}{m} + \frac{(3c+5)m}{16(2c+3)}\lg\kappa + O(m) \\
&\le \lg\binom{V}{m} + \frac{m}{8}\lg\kappa.
\end{aligned}$$

To answer query $x$, we first query the data structure for $S'$, and find $i$ such that $y'_i \le x < y'_{i+1}$. Then we go over the $16(2c+3)$ elements between $y'_i$ and $y'_{i+1}$, and compare each of them with $x$. This finishes the proof of Lemma 8.3. □

Next, we show that if the sets are very small ($m \le O(\kappa/\lg\kappa)$), then there is a data structure with constant query time and negligible extra bits.

LEMMA 8.4. *Let $c \ge 2, \epsilon$ be two positive constants. There is a preprocessing algorithm* `prepRankS`*, a query algorithm* `qAlgRankS` *and lookup tables* $\texttt{tableRankS}_{V,m}$ *of sizes $O(2^{\epsilon\kappa})$, such that for any integers $V \le 2^\kappa$ and $m \le c\cdot\kappa/\lg\kappa$, such that given a set $S \subset [V]$ of size $m$,* `prepRankS` *preprocesses $S$ into a data structure using $\lg\binom{V}{m} + 2^{-\kappa/2}$ bits of space. Given any $x \in [V]$,* `qAlgRankS` *computes* $\texttt{rank}_S(x)$ *in constant time by accessing the data structure and* $\texttt{tableRankS}_{V,m}$.

*Proof.* Consider the binary trie over $\{0,\dots,V\}$.[12] Every element in $\{0,\dots,V\}$ corresponds to a root-to-leaf path. Consider all paths corresponding to an element in $S \cup \{V\}$ ($V$ is included for technical reasons). Their union forms a subtree $T(S)$ of the binary trie with $m+1$ leaves. In the following, we construct a data structure assuming the *topological structure* of $T(S)$ is known, then apply Proposition 4.14 to fuse the topological structure into the data structure.

Roughly speaking, the *topological structure* of a subtree $T$ is the tree $T$ without specifying for each node with only one child, whether it is a left or a right child (see Figure 1a). Formally, it is defined by partitioning the set of such subtrees into equivalence classes, modulo the `flip` operation. Let $v$ be a node in $T$ with only a left [resp. right] child, let $\texttt{flip}(v,T)$ be $T$ relocating $v$'s entire left [resp. right] subtree to its right [resp. left] child. We say two trees $T \sim T'$ if there is a (finite) sequence of `flip` operations that modifies $T$ to $T'$. It is easy to verify that $\sim$ is an equivalence relation, hence it partitions the set of all $T$ into equivalence classes.

We call an edge in $T(S)$ *a shared edge* if it has more than one leaf in its subtree. Equivalently, a shared edge is shared between at least two root-to-leaf paths. Note that if an edge is shared, then all edges on the path from root to it are shared. It turns out that the *number of shared edges* in $T(S)$ is an important parameter, which is also invariant under `flip`. Thus, for each equivalence class $\mathcal{T}$, all $T \in \mathcal{T}$ have the same number of shared edges (see Figure 1b).

[12] We write every integer in the set as a $\lceil\lg(V+1)\rceil$-bit string, then construct a trie over these $V+1$ binary strings. Note that $S$ is a subset of $\{0,\dots,V-1\}$, while the trie has $V+1$ leaves.

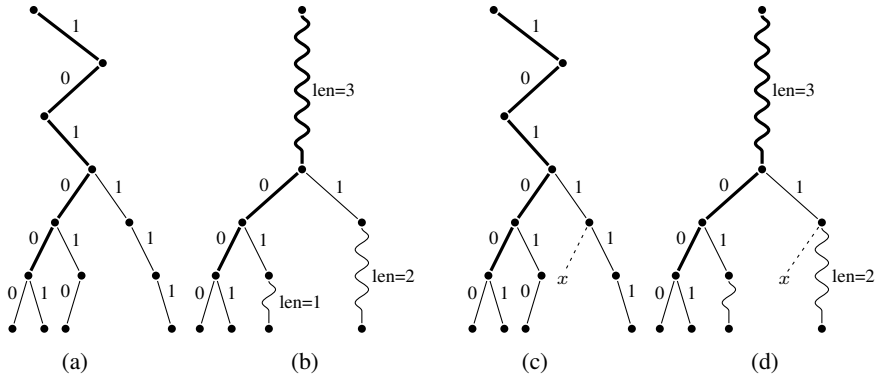


Fig. 1: (b) is the *topological structure* of (a), by getting rid of the information that for each single child, whether it is a left or a right child. The thick edges are *shared*. Query $x$ branches off the tree along the dotted edge.

Intuitively, for a typical set $S$, the corresponding $\mathcal{T} \ni T(S)$ should have most of its degree-two nodes close to the root, i.e, it should have very *few* shared edges. Indeed, if we sample a uniformly random $S$, the number of shared edges is at most $O(\kappa)$ with probability at least $1 - 2^{-\Omega(\kappa)}$. As we will see below, on the inputs with few shared edges, it is relatively easy to construct data structures and answer queries. However, for the rare inputs with more than $\Omega(\kappa)$ shared edges, we can afford to use a different construction with a larger redundancy. Since they are sufficiently rare, the overall redundancy turns out to be small.

*Few shared edges.* Let us fix an equivalence class $\mathcal{T}$, assume $\mathcal{T}$ is known and consider all inputs $S$ such that $T(S) \in \mathcal{T}$. Furthermore, assume the trees in $\mathcal{T}$ have at most $(2c+1)\kappa$ shared edges. For each such $\mathcal{T}$, we are going to construct a lookup table $\texttt{tableRankS}_{V,m,\mathcal{T}}$, and preprocess $S$ into a data structure using about $\lg|\mathcal{T}|$ bits such that if the query algorithm is given access to this particular lookup table (specific for $\mathcal{T}$), it answers `rank` queries in constant time.

Since the tree $T(S)$ uniquely determines $S$, to construct the data structure for $S$, it suffices to encode for each edge in $T(S)$ that connects a single child and its parent, whether the child is left or right. The preprocessing algorithm constructs $T(S)$, then goes through all such edges in a *fixed* order, and uses one bit to indicate whether the corresponding edge in $T(S)$ connects to a left child or a right child. To facilitate the queries (which we will describe in the next paragraph), all shared edges are encoded first in the *depth-first-search* order, followed by all other edges encoded in the *depth-first-search* order. This ensures that

1. if a shared edge $e_1$ is on the path from root to shared edge $e_2$, then $e_1$ is encoded before $e_2$;
2. for each $y_i$, its non-shared edges (which is a suffix in the root-to-leaf path) are consecutive in the data structure.

Note that this encoding is a *one-to-one* mapping: Every $S$ such that $T(S) \in \mathcal{T}$ is encoded to a different string; Every string has a corresponding $S$ with $T(S) \in \mathcal{T}$ encoded to it. Thus, the algorithm constructs a data structure using exactly

$$\lg|\{S : T(S) \in \mathcal{T}\}|$$

bits of space.

Let $S = \{y_1, \ldots, y_m\}$ such that $y_1 < y_2 < \cdots < y_m$, and let $y_0 = -1$ and $y_m = V$.

Given a query $x \in \{0, \ldots, V-1\}$, the goal is to compute $i$ such that $y_i \le x < y_{i+1}$. Let us consider the process of walking down the tree $T(S)$ following the bits in $x$. That is, we write $x$ also as a $\lceil \lg(V+1) \rceil$-bit string, and walk down the tree from the root: if the current bit in $x$ is 0, we follow the left child, otherwise we follow the right child. The process stops when either the current node in $T(S)$ does not have a left (or right) child to follow, or we have reached a leaf. The location where it stops determines the answer to the query, in the same way for *all* $T \in \mathcal{T}$. See Figure 1c and 1d for a concrete example. Note that in the example, $x$ branches off the tree from left, which may not be possible at the same location for all $T \in \mathcal{T}$, as some $T$ may have a left child there. But *given* that $x$ branches off the tree at this location from left, all $T(S) \in \mathcal{T}$ must have the same answer to $\texttt{rank}_S(x)$. Thus, we store in $\texttt{tableRankS}_{V,m,\mathcal{T}}$, for all nodes $v$ in the tree, $\text{ans}_{v,0}$ and $\text{ans}_{v,1}$, the answer to the query when the process branches off the tree from $v$ due to the lack of its left child (i.e., from left), and the answer when it branches off from $v$ due to the lack of its right child (i.e., from right) respectively. It takes $O(\kappa^2)$ words, since $m \le \kappa$.

Now the task is reduced to efficiently simulating this walk. To this end, the query algorithm needs to compare the bits in $x$ with the corresponding bits of $T(S)$, which are stored in the data structure. It turns out that the difficult part is to compare $x$ with the shared edges, which are stored in the first (at most) $(2c+1)\kappa$ bits. The first step is to simulate the walk, and check if $x$ branches off $T(S)$ at a shared edge. We create lookup tables of size $2^{\epsilon\kappa}$ to compare $\epsilon\kappa$ bits at once. For now, let us focus on the first $\epsilon\kappa$ bits $x_{\le \epsilon\kappa}$. These bits determine for all the degree-two nodes in the first $\epsilon\kappa$ levels, which child $x$ follows (note we have fixed $\mathcal{T}$). Thus, it determines for all other bits, which bits in the data structure they should compare with. In the lookup table, we store for each of the $2^{\epsilon\kappa}$ possible values,

- a $(2c+1)\kappa$-bit string, which permutes $x_{\le \epsilon\kappa}$ to the same location as the bits they are comparing with;
- a $(2c+1)\kappa$-bit string, indicating for each shared edge in the data structure, whether they are being compared.

With these two strings, the query algorithm is able to compare $x_{\le \epsilon\kappa}$ with the first $\epsilon\kappa$ levels of $T(S)$. If they do not match, we could find the first edge where they differ (since edges are encoded in the DFS order), which is the location where $x$ branches off $T(S)$. If they all equal, we proceed and compare the next $\epsilon\kappa$ bits. Note that we may start the next chunk of the walk from different nodes depending on the value of $x_{\le \epsilon\kappa}$, and we will need a different lookup for each starting location. However, $\mathcal{T}$ can have at most $m$ nodes in each level, thus, only $m$ tables are needed for each chunk. We repeat the above process until we find a different bit, or we find out that $x$ matches all shared edges from the root. In the former case, as we argued above, the answer to the query can be found in the lookup table. In the latter case, by the definition of shared edges, we identified one $y_i$ which is the only element in $S$ that matches the prefix of $x$. Thus, it suffices to retrieve the remaining bits of $y_i$, which are stored consecutively in the data structure and take constant retrieval time, and compare $y_i$ with $x$. If $y_i \le x$, then the query algorithm returns $i$, otherwise, it returns $i-1$. The query time is constant.

So far for every $\mathcal{T}$ with at most $(2c+1)\kappa$ shared edges, we have designed a data structure that works for all inputs $S$ such that $S \in \mathcal{T}$ using space $\lg |\{S : T(S) \in \mathcal{T}\}|$ bits, constant query time and lookup table of size $2^{\epsilon\kappa}$. Next, we fuse $\mathcal{T}$ into the data structure and merge all lookup tables, obtaining a single data structure that works for all $S$ such that $T(S)$ has at most $(2c+1)\kappa$ shared edges, which uses lookup table $\texttt{tableRankS}_{V,m,\text{few}}$. To this end, we fix an arbitrary ordering of all such equivalence classes $\mathcal{T}$: $\mathcal{T}_1, \ldots, \mathcal{T}_C$, where $C$ is the number of equivalence classes. Let $s_i = \lg |\{S : T(S) \in \mathcal{T}_i\}|$ be the size of the data structure for $\mathcal{T}_i$. Then, $C \le 2^{2m} \cdot \binom{(2c+1)\kappa+1}{m-1} \le 2^{m \lg(\kappa/m)+O(m)}$. This is because there are at most

$2^{2m}$ rooted binary trees with $m+1$ nodes (corresponding to the degree-two nodes). Each such tree can be extended to a class $\mathcal{T}$ by specifying the distance from each child to its parent (adding the degree-one nodes). However, there are only $(2c+1)\kappa$ shared edges, thus, the sum of distances of all internal edges is at most $(2c+1)\kappa$, and there are $m-1$ internal edges.[13] Hence, there are at most $\binom{\leq(2c+1)\kappa}{m-2} \leq \binom{(2c+1)\kappa+1}{m-1}$ choices. Once the distances on all internal edges are determined, the distance on each edge connecting to a leaf is also fixed, because all leaves are at depth $\lceil \lg(V+1) \rceil$.

Given an input set $S$ such that $T(S)$ has at most $(2c+1)\kappa$ shared edges, the preprocessing algorithm computes $T(S)$ and finds the index $i$ such that $\mathcal{T}_i \ni T(S)$. Then it runs the preprocessing algorithm for class $\mathcal{T}_i$ on $S$, and computes a data structure $\mathcal{D}_i$ of at most $s_i$ bits. Next, we use Proposition 4.14 to store the pair $(i, \mathcal{D}_i)$, using space at most

$$\begin{aligned}
\lg \sum_{i=1}^{C} 2^{s_i} + C \cdot 2^{-\kappa+2} &\leq \lg \left( \sum_{i=1}^{C} |\{S : T(S) \in \mathcal{T}_i\}| \right) + 2^{m \lg(\kappa/m) + O(m) - \kappa + 2} \\
&< \lg \binom{V}{m} + 2^{m \lg(\kappa/m) + O(m) - \kappa + 2} \\
&< \lg \binom{V}{m} + 2^{-\frac{3}{4}\kappa}.
\end{aligned}$$

The lookup table $\texttt{tableRankS}_{V,m,\text{few}}$ is the collection of all tables $\texttt{tableRankS}_{V,m,\mathcal{T}_i}$ for $i = 1, \ldots, C$, as well as the $O(C)$-sized table from Proposition 4.14. Thus, the total size is at most $2^{\epsilon\kappa} \cdot C + O(C) = 2^{(\epsilon + o(1))\kappa}$.

To answer a query $x$, Proposition 4.14 allows us to decode $i$ and $\mathcal{D}_i$ in constant time by using a lookup table of size $O(C)$. Then, we find the corresponding $\texttt{tableRankS}_{V,m,\mathcal{T}_i}$ and run the query algorithm for $\mathcal{T}_i$ on query $x$ and data structure $\mathcal{D}_i$. The query time is constant.

*Many shared edges.* Next, we construct a data structure that works for all $S$ such that $T(S)$ has more than $(2c+1)\kappa$ shared edges, using

$$\lg \binom{V}{m} - \kappa$$

bits of space. Note that this is possible, because there are very few such sets $S$ (a tiny fraction of all $\binom{V}{m}$ sets). We find the largest $k$ such that $T(S_{\leq k})$ has at most $(2c+1)\kappa$ shared edges, where $S_{\leq k} = \{y_1, \ldots, y_k\}$. Note that every element can introduce no more than $\kappa$ shared edges, thus, $T(S_{\leq k})$ has at least $2c\kappa$ shared edges. The data structure stores the (index of) equivalence class $\mathcal{T} \ni T(S_{\leq k})$, then we run the preprocessing algorithm on $S_{\leq k}$. This encodes the first $k$ elements of $S$. For the next $m-k$ elements, we simply apply Lemma 8.3.

More specifically, for $k$ elements, there are at most $2^{k \lg(\kappa/k) + O(k)}$ equivalence classes, as we showed earlier. We construct the data structure as follows:

1. write down the index $k$ using $\lceil \lg m \rceil$ bits;
2. write down the index $i$ such that $\mathcal{T}_i \ni T(S_{\leq k})$ using $\lceil k \lg(\kappa/k) + O(k) \rceil$ bits;
3. run the preprocessing algorithm on $S_{\leq k}$ and obtain a data structure of size
$$\lg |\{S_{\leq k} : T(S_{\leq k}) \in \mathcal{T}_i\}|;$$
4. run `prepRankL` on $\{y_{k+1}, \ldots, y_m\}$ and obtain a data structure of size
$$\lg \binom{V}{m-k} + \frac{1}{8}(m-k) \lg \kappa.$$

[13] An edge is internal if it does not connect to a leaf.

Observe that Step 3 uses at most

$$k\lceil \lg V \rceil - 2c\kappa$$

bits, because for any such $\mathcal{T}_i$,

- by construction, each bit of the data structure stores an input bit, i.e., one of the bits representing $\{y_1, \ldots, y_k\}$;
- each of the $\geq 2c\kappa$ shared edges corresponds to at least two input bits (since given $\mathcal{T}$, these two input bits are always the same);
- each input bit is stored only once.

Therefore, the preprocessing algorithm outputs a data structure using

$$\begin{aligned}
&\lg m + k\lg(\kappa/k) + O(k) + (k\lg V - 2c\kappa) \\
&\quad + \left(\lg \binom{V}{m-k} + \frac{1}{8}(m-k)\lg\kappa\right) + k + 2 \\
&\leq \lg m + k\lg(\kappa/k) + (k\lg V - 2c\kappa) + (m-k)\lg V + \frac{1}{8}m\lg\kappa + O(k) \\
&\leq m\lg V - 2c\kappa + \lg m + k\lg(\kappa/k) + \frac{1}{8}m\lg\kappa + O(k) \\
&\leq \lg \binom{V}{m} + m\lg m - 2c\kappa + \lg m + m\lg(\kappa/m) + \frac{1}{8}m\lg\kappa + O(m) \\
&\leq \lg \binom{V}{m} - 2c\kappa + \frac{9}{8}m\lg\kappa + O(m).
\end{aligned}$$

By the fact that $m \leq c\kappa/\lg\kappa$ and $c \geq 2$, it is at most

$$\lg \binom{V}{m} - \kappa.$$

The lookup table includes $\texttt{tableRankS}_{V,k,\text{few}}$ for all $k \leq m$, and has $2^{(\epsilon+o(1))\kappa}$ size.

To answer query $x$, the query algorithm reads $k$ and $i$. Then it runs the query algorithm for $\mathcal{T}_i$ for query $x$ on the data structure for $S_{\leq k}$, as well as `qAlgRankL` for $x$ on the data structure for $\{y_{k+1}, \ldots, y_m\}$. Both algorithms run in constant time. The answer to the query is simply the sum of the two answers.

*Combining the two cases.* Finally, we combine the two cases using Proposition 4.14, and construct a data structure that works for all sets $S$. Given set $S$, `prepRankS` computes $T(S)$ and the number of shared edges. If it has no more than $(2c+1)\kappa$ shared edges, it sets $b := 1$, runs the preprocessing algorithm for "many shared edges" and obtains a data structure $\mathcal{D}_1$. Otherwise, it sets $b := 2$, runs the preprocessing algorithm for "few shared edges" and obtains a data structure $\mathcal{D}_2$. At last, it applies Proposition 4.14 to store the pair $(b, \mathcal{D}_b)$. The space usage is

$$\begin{aligned}
&\lg\left(\binom{V}{m} \cdot 2^{2^{-\frac{3}{4}\kappa}} + \binom{V}{m} \cdot 2^{-\kappa}\right) + 2^{-\kappa+2} \\
&\leq \lg \binom{V}{m} + 2^{-\frac{3}{4}\kappa} + \lg(1 + 2^{-\kappa-2^{-\frac{3}{4}\kappa}}) + 2^{-\kappa+2} \\
&\leq \lg \binom{V}{m} + 2^{-\frac{1}{2}\kappa}.
\end{aligned}$$

To answer query $x$, we simply decode $b$ and $\mathcal{D}_b$ using Proposition 4.14, and use the corresponding query algorithm based on $b$.

The lookup table $\texttt{tableRankS}_{V,m}$ also includes all $\texttt{tableRankS}_{V,k}$ for $k \le m$, which has size $2^{O(\epsilon\kappa)}$. This proves Lemma 8.4. □

Finally, we are ready to prove Lemma 8.1, which constructs a rank data structure for $m \le \kappa^c$.

*Proof of Lemma 8.1.* The data structure construction is based on recursion. As the base case, if $m \le 16\kappa/\lg\kappa$, we simply use the data structure from Lemma 8.4, and the statement holds. Otherwise for $m > 16\kappa/\lg\kappa$, we divide $V$ into $B$ blocks of equal size, for $B = \lceil\kappa^{1/2}\rceil$. For a typical set $S$, we would expect each block to contain roughly $m/B$ elements. If it indeed happens, the size of $S$ would be reduced by a factor of $B$. Hence, we will reach the base case in a constant number of rounds. On the other hand, input sets $S$ which have at least one block with significantly more than $m/B$ elements are very rare. If such blocks occur, we are going to apply Lemma 8.3 on them. Although Lemma 8.3 introduces a large redundancy, such cases occur sufficiently rarely, so that the overall redundancy is still small.

More specifically, we partition the input set $S$ into $B$ subsets $S_1, \ldots, S_B$ such that $S_i$ contains all elements of $S$ between $\lceil(i-1)V/B\rceil$ and $\lceil iV/B\rceil - 1$. Let $V_i := \lceil iV/B\rceil - \lceil(i-1)V/B\rceil$ be the size of the $i$-th block. By definition, $|S_1| + \cdots + |S_B| = m$ and $V_1 + \cdots + V_B = V$. We construct a data structure for each $S_i$, over a universe of size $V_i$. Then we apply Proposition 4.10 to concatenate the $B$ data structures *given* the sizes of $S_1, \ldots, S_B$. Finally, we apply Proposition 4.14 to union all combinations of sizes. We present the details below.

*Preprocessing algorithm.* Given a set $S$ of size $m$, if $2m \ge V$, we take the complement. Note that the space bound stated in the lemma becomes smaller after taking the complement. It is also easy to derive the answer from the data structure for the complement. Then if $m = 1$, we simply write down the element; if $m \le 16\kappa/\lg\kappa$, we apply Lemma 8.4.

**preprocessing algorithm** $\texttt{prepRank}(V, m, S)$:
1. if $V \le 2m$
2. $\quad m := V - m$ and $S := [V] \setminus S$
3. if $m = 1$
4. $\quad$ return the only element in $S$
5. if $m \le 16\kappa/\lg\kappa$
6. $\quad$ return $\mathcal{D} := \texttt{prepRankS}(V, m, S)$ using Lemma 8.4

(to be cont'd)

If $m > 16\kappa/\lg\kappa$, we divide $[V]$ into $\kappa^{1/4}$ chunks, and construct a data structure for each chunk.

7. $B := \lfloor\kappa^{1/4}\rfloor$
8. compute $S_i := S \cap [(i-1)V/B, iV/B)$ and $m_i := |S_i|$
9. let $V_i := \lceil iV/B\rceil - \lceil(i-1)V/B\rceil$
10. for $i = 1, \ldots, B$
11. $\quad$ if $m_i > \max\{m \cdot \kappa^{-1/4}, 16\kappa/\lg\kappa\}$
12. $\quad\quad$ compute $\mathcal{D}_i := \texttt{prepRankL}(V_i, m_i, S_i)$ using Lemma 8.3
13. $\quad$ else
14. $\quad\quad$ compute $\mathcal{D}_i := \texttt{prepRank}(V_i, m_i, S_i)$ recursively

(to be cont'd)

If the chunk has too many elements, we apply Lemma 8.3 to construct a data structure with larger redundancy. Otherwise, the size of the set at least decreases by a factor of $\kappa^{1/4}$, and we recurse.

Next, we concatenate the data structures for all chunks, and fuse the tuple $(m_1, \ldots, m_B)$ into the data structure.

15. apply Proposition 4.10 to concatenate $\mathcal{D}_1, \ldots, \mathcal{D}_B$, and obtain $\mathcal{D}_{\text{cat}}$
16. let $C := \binom{m+B-1}{B-1}$ be the number of different tuples $(m_1, \ldots, m_B)$ such that $m_i \geq 0$ and $m_1 + \cdots + m_B = m$
17. let $1 \leq j \leq C$ be the index such that the current $(m_1, \ldots, m_B)$ is the $j$-th in the lexicographic order
18. apply Proposition 4.14 to fuse $j$ into $\mathcal{D}_{\text{cat}}$, and obtain $\mathcal{D}$
19. return $\mathcal{D}$

*Space analysis.* We will prove by induction that $\texttt{prepRank}(V, m, S)$ outputs a data structure of size at most

$$\lg \binom{V}{m} + (m-1)2^{-\kappa/2}.$$

The base case when $m \leq 16\kappa/\lg\kappa$ is a direct implication of Lemma 8.4 (or if $m = 1$, the space usage is $\lg V = \lg \binom{V}{1}$). Now, let us consider larger $m$.

To prove the inductive step, let us fix a $B$-tuple $(m_1, \ldots, m_B)$, and consider the size of $\mathcal{D}_{\text{cat}}$ from line 15. By Proposition 4.10, when all $m_i \leq \max\{m \cdot \kappa^{-1/4}, 16\kappa/\lg\kappa\}$, its size is at most

$$s(m_1, \ldots, m_B) := \lg \prod_{i=1}^{B} \binom{V_i}{m_i} + (m-B) \cdot 2^{-\kappa/2} + (B-1)2^{-\kappa+4};$$

otherwise, its size is at most

$$s(m_1, \ldots, m_B) := \lg \prod_{i=1}^{B} \binom{V_i}{m_i} + \sum_{i: m_i > \max\{m \cdot \kappa^{-1/4}, 16\kappa/\lg\kappa\}} \frac{1}{8} m_i \lg\kappa + B. \tag{8.1}$$

It turns out that in the latter case, (8.1) is *significantly* smaller than $\lg \binom{V}{m}$.

CLAIM 8.5. *If at least one* $m_i > \max\{m \cdot \kappa^{-1/4}, 16\kappa/\lg\kappa\}$, *then* $s(m_1, \ldots, m_B)$ *is at most* $\lg \binom{V}{m} - \kappa$.

To focus on the space analysis, we defer the proof of the claim to the end of this subsection. Assuming the claim, by Proposition 4.14, the size of $\mathcal{D}$ from line 18 is at most

$$\lg \left( \sum_{\substack{m_1, \ldots, m_B: \\ \sum_i m_i = m}} 2^{s(m_1, \ldots, m_B)} \right) + C \cdot 2^{-\kappa+2}. \tag{8.2}$$

To bound the sum in the logarithm, we first take the sum only over all tuples such that $m_i \leq \max\{m \cdot \kappa^{-1/4}, 16\kappa/\lg\kappa\}$, the sum is at most

$$\begin{aligned} \sum 2^{s(m_1, \ldots, m_B)} &\leq \sum \prod_{i=1}^{B} \binom{V_i}{m_i} \cdot 2^{(m-B) \cdot 2^{-\kappa/2} + (B-1)2^{-\kappa+4}} \\ &\leq \binom{V}{m} \cdot 2^{(m-B) \cdot 2^{-\kappa/2} + (B-1)2^{-\kappa+4}}. \end{aligned}$$

The second inequality uses the fact that $\sum_{m_1, \ldots, m_B: \sum m_i = m} \prod_{i=1}^{B} \binom{V_i}{m_i} = \binom{\sum_{i=1}^{B} V_i}{m}$, and we are taking this sum over a subset of all such $B$-tuples. By Claim 8.5, $s(m_1, \ldots, m_B) \leq \lg \binom{V}{m} - \kappa$ for all other tuples. Thus, the sum in the logarithm is at most

$$\binom{V}{m} \cdot 2^{(m-B) \cdot 2^{-\kappa/2} + (B-1)2^{-\kappa+2}} + \binom{V}{m} \cdot C \cdot 2^{-\kappa}.$$

Finally, since $C \le m^B$ and $m \le \kappa^c$, (8.2) is at most

$$\begin{aligned}
(8.2) &\le \lg\left(\binom{V}{m}\cdot 2^{(m-B)\cdot 2^{-\kappa/2}+(B-1)2^{-\kappa+4}} + \binom{V}{m}\cdot m^B\cdot 2^{-\kappa}\right) + m^B\cdot 2^{-\kappa+2}\\
&\le \lg\binom{V}{m} + (m-B)2^{-\kappa/2} + (B-1)2^{-\kappa+4} + \lg(1+2^{-\kappa+B\lg m}) + 2^{-\kappa+B\lg m+2}\\
&\le \lg\binom{V}{m} + (m-B)2^{-\kappa/2} + (B-1)2^{-\kappa+4} + 2^{-\kappa+c\kappa^{1/4}\lg\kappa+3}\\
&\le \lg\binom{V}{m} + (m-1)2^{-\kappa/2}.
\end{aligned}$$

By induction, it proves the data structure uses space as claimed.

*Lookup table.* We store the following information in the lookup table.

**lookup table** $\texttt{tableRank}_{V,m}$:
1. if $m \le 16\kappa/\lg\kappa$, include $\texttt{tableRankS}_{V,m}$ from Lemma 8.4
2. the value of $C = \binom{m+B-1}{B-1}$
3. for $j = 1,\dots,C$
4. the $j$-th $B$-tuple $(m_1,\dots,m_B)$ in lexicographic order
5. for $i = 1,\dots,B$
6. $m_1+\cdots+m_i$
7. lookup table for Proposition 4.10 in line 15, for all possible $B$-tuples $(m_1,\dots,m_B)$
8. lookup table for Proposition 4.14 in line 18
9. include all tables $\texttt{tableRank}_{V',m'}$ and $\texttt{tableRankL}_{V',m'}$ for $V' = \lfloor V/B^i\rfloor$ or $\lceil V/B^i\rceil$ for $i \ge 1$ and $m' \le m$

Since $C = \binom{m+B-1}{B-1} \le 2^{o(\kappa)}$, line 2 to 8 all have size $2^{o(\kappa)}$. Finally, we are only including $\kappa^{O(1)}$ other tables in line 1 and 6, each taking at most $\tilde{O}(2^{\epsilon\kappa})$ bits by Lemma 8.3 and 8.4. The total size of $\texttt{tableRank}_{V,m}$ is $\tilde{O}(2^{\epsilon\kappa})$.

*Query algorithm.* Given a query $x$, if $V \le 2m$, we retreat the data structures as storing the complement of $S$, and use the fact that $\texttt{rank}_S(x) = x+1-\texttt{rank}_{[V]\setminus S}(x)$. Then if $m = 1$, we simply compare it with $x$. If $m \le 16\kappa/\lg\kappa$, we invoke the query algorithm from Lemma 8.4.

**query algorithm** $\texttt{qAlgRank}(V,m,x)$:
1. if $V \le 2m$
2. $m := V - m$
3. in the following, when about to return answer $r$, return $x+1-r$
4. if $m = 1$
5. retrieve the element, compare it with $x$, and return 0 or 1
6. if $m \le 16\kappa/\lg\kappa$,
7. return $\texttt{qAlgRankS}(V,m,x)$ (from Lemma 8.4)

(to be cont'd)

If $m > 16\kappa/\lg\kappa$, we decode $j$, which encodes the tuple $(m_1,\dots,m_B)$ and $\mathcal{D}_{\text{cat}}$. Then if $x$ is in the $i$-th chunk, we decode $m_i$ and the corresponding $\mathcal{D}_i$.

8. apply Proposition 4.14 to decode $j$ and $\mathcal{D}_{\text{cat}}$
9. let $i$ be the chunk that contains $x$
10. apply Proposition 4.10 to decode $\mathcal{D}_i$
11. retrieve $m_1+\cdots+m_{i-1}$ and $m_i$ for $j$-th tuple from the lookup table

(to be cont'd)

Then depending on the value of $m_i$, we invoke the query algorithm from Lemma 8.3 or recurse.

12. if $m_i > \max\{m\cdot\kappa^{-1/4}, 16\kappa/\lg\kappa\}$
13. return $(m_1+\cdots+m_{i-1})+\mathcal{D}_i.\texttt{qAlgRankL}(V_i,m_i,x-\lceil(i-1)V/B\rceil)$ (from Lemma 8.3)
14. else
15. return $(m_1+\cdots+m_{i-1})+\mathcal{D}_i.\texttt{qAlgRank}(V_i,m_i,x-\lceil(i-1)V/B\rceil)$

The query algorithm recurses only when $m_i \le m\cdot\kappa^{-1/4}$. In all other cases, the query

is answered in constant time. On the other hand, $m \leq \kappa^c$. The level of recursion must be bounded by a constant. Thus, the data structure has constant query time, proving Lemma 8.1.□

In order to complete the proof, we now prove the remaining claim.

*Proof of Claim 8.5.* To prove the claim, let us first compare the first term with $\lg \binom{V}{m}$. We have

$$\begin{aligned}
&\lg \binom{V}{m} - \lg \prod_{i=1}^{B} \binom{V_i}{m_i} \\
&= \lg \frac{V!}{m!(V-m)!} + \sum_{i=1}^{B} \lg \frac{m_i!(V_i-m_i)!}{V_i!},
\end{aligned}$$

which, by Stirling's formula, is

$$\begin{aligned}
&\geq \lg \frac{\sqrt{2\pi V}(V/e)^V}{\sqrt{2\pi m}(m/e)^m \cdot \sqrt{2\pi(V-m)}((V-m)/e)^{V-m}} \\
&\qquad + \sum_{i=1}^{B} \lg \frac{\sqrt{2\pi m_i}(m_i/e)^{m_i} \cdot \sqrt{2\pi(V_i-m_i)}((V_i-m_i)/e)^{V_i-m_i}}{\sqrt{2\pi V_i}(V_i/e)^{V_i}} - O(B) \\
&\geq \lg \frac{V^V}{m^m \cdot (V-m)^{V-m}} - \lg V + \sum_{i=1}^{B} \lg \frac{m_i^{m_i} \cdot (V_i-m_i)^{V_i-m_i}}{V_i^{V_i}} - O(B) \\
&= \sum_{i=1}^{B} \left( V_i \lg \frac{V}{V_i} - m_i \lg \frac{m}{m_i} - (V_i-m_i) \lg \frac{V-m}{V_i-m_i} \right) - O(B) - \lg V,
\end{aligned}$$

which by the fact that $f(\varepsilon) = \varepsilon \log 1/\varepsilon$ is concave and hence $V \cdot f(\frac{V_i}{V}) \geq m \cdot f(\frac{m_i}{m}) + (V - m) \cdot f(\frac{V_i - m_i}{V-m})$ (i.e., all summands are nonnegative), is

$$\begin{aligned}
(8.3) \quad &\geq \sum_{i: m_i > \max\{m \cdot \kappa^{-1/4}, 16\kappa/\lg\kappa\}} \left( V_i \lg \frac{V}{V_i} - m_i \lg \frac{m}{m_i} - (V_i - m_i) \lg \frac{V-m}{V_i - m_i} \right) \\
&\qquad - O(B) - \lg V.
\end{aligned}$$

For each term in this sum, we have

$$V_i \lg \frac{V}{V_i} = V_i \lg B - V_i \lg \left( 1 + \frac{V_i - V/B}{V/B} \right) \geq V_i \lg B - O(1),$$

since $|V_i - V/B| \leq 1$; and

$$\begin{aligned}
(V_i - m_i) \lg \frac{V-m}{V_i - m_i} &= (V_i - m_i) \left( \lg B + \lg \left( 1 + \frac{m_i - m/B + (V/B - V_i)}{V_i - m_i} \right) \right) \\
&\leq (V_i - m_i) \lg B + (V_i - m_i) \cdot \frac{m_i - m/B + 1}{V_i - m_i} \cdot \lg e \\
&\leq (V_i - m_i) \lg B + 2m_i.
\end{aligned}$$

Plugging it into (8.3), we have

$$\lg \binom{V}{m} - \lg \prod_{i=1}^{B} \binom{V_i}{m_i}$$

$$
\begin{aligned}
&\geq \sum_{i:m_i>\max\{m\cdot\kappa^{-1/4},16\kappa/\lg\kappa\}} \left(V_i \lg B - m_i \lg \frac{m}{m_i} - (V_i - m_i)\lg B - 2m_i\right) \\
&\qquad - O(B) - \lg V \\
&= \sum_{i:m_i>\max\{m\cdot\kappa^{-1/4},16\kappa/\lg\kappa\}} m_i\left(\lg\frac{Bm_i}{m} - 2\right) - O(B) - \lg V \\
&\geq \sum_{i:m_i>\max\{m\cdot\kappa^{-1/4},16\kappa/\lg\kappa\}} m_i\left(\frac{1}{4}\lg\kappa - 2\right) - O(B) - \lg V.
\end{aligned}
$$

Therefore, we have

$$
\begin{aligned}
(8.1) &\leq \lg\binom{V}{m} - \sum_{i:m_i>\max\{m\cdot\kappa^{-1/4},16\kappa/\lg\kappa\}} m_i\left(\frac{1}{4}\lg\kappa - 2\right) + O(B) + \lg V \\
&\quad + \sum_{i:m_i>\max\{m\cdot\kappa^{-1/4},16\kappa/\lg\kappa\}} \frac{1}{8} m_i \lg\kappa \\
&\leq \lg\binom{V}{m} - \sum_{i:m_i>\max\{m\cdot\kappa^{-1/4},16\kappa/\lg\kappa\}} m_i\left(\frac{1}{8}\lg\kappa - 2\right) + O(B) + \lg V \\
&\leq \lg\binom{V}{m} - \kappa.
\end{aligned}
$$

The last inequality is due to the fact that there is at least one $m_i$ that is larger than $\max\{m\cdot\kappa^{-1/4}, 16\kappa/\lg\kappa\}$ (in particular, $m_i > 16\kappa/\lg\kappa$), $B = \Theta(\kappa^{1/2})$ and $\lg V \leq \kappa/2$. □

**8.2. Data interpretation.** Now, we show how to represent a string as a set which allows us to locally decode the string given access to a `rank` oracle of the resulting set. Formally, we will prove the following lemma.

LEMMA 8.6. *Let $c \geq 2$ be a constant positive integer. There is a preprocessing algorithm* `prepIntoSet`*, a query algorithm* `request` *and lookup tables* $\texttt{tableInt}_{V,m}$ *of sizes $O(\kappa^{c+2})$, such that for any integers $V$ and $m$ where $V \leq 2^{\kappa/2}$ and $m \leq \kappa^c$, given a (double-ended) string $\mathcal{D} = (K_h, M, K_t)$ of length*

$$
s \leq \lg\binom{V}{m} - m(V-1)2^{-\kappa+2},
$$

$\texttt{prepIntoSet}(V, m, \mathcal{D})$ *outputs a set $S \subseteq [V]$ of size $m$ with the following properties: For any $-1 \leq a_1 \leq a_2 \leq |M|$ and $a_2 < a_1 + \kappa$,*

$$
\texttt{request}(0, V, m, \text{range}[K_h], |M|, \text{range}[K_t], a_1, a_2, 0)
$$

*computes $\mathcal{D}[a_1, a_2]$ in $O(\kappa^4)$ time using $O(\kappa^2)$* `rank` *queries to $S$, assuming it can make random accesses to the lookup table* $\texttt{tableInt}_{V,m}$.

*Proof.* To construct set $S$ from the input string $\mathcal{D}$, the main idea is to apply Proposition 4.23 and 4.20, and then recurse on the two halves of $[V]$. Roughly speaking, we extract an integer $m_1 \in [m+1]$ from $\mathcal{D}$ using Proposition 4.23, which encodes the number of elements in the first half of $[V]$. Then we split $\mathcal{D}$ into two data structures $\mathcal{D}_1$ and $\mathcal{D}_2$ such that the length of $\mathcal{D}_1$ is approximately $\lg\binom{V/2}{m_1}$ and the length of $\mathcal{D}_2$ is approximately $\lg\binom{V/2}{m-m_1}$. Then the set $S$ is recursively constructed in the two halves. When the $m$ gets sufficiently

small and $\mathcal{D}$ has length $O(\kappa^2)$, we continue the recursion without applying the two propositions about fraction-length strings. Instead, we take the whole string as an integer smaller than $2^{O(\kappa^2)}$, and use the integer to encode a set (also decode the whole integer at the decoding time).

*Transforming an integer to and from a set.* We first show given an integer $Z \le \binom{V}{m}$, how to turn it into a subset of $[V]$ of size $m$, such that $Z$ can be recovered using `rank` queries.

**encoding algorithm** $\texttt{encSet}(V, m, Z)$:
1. if $m = 0$, return $\emptyset$
2. if $m = V$, return $[V]$
3. $V_1 := \lfloor V/2 \rfloor$ and $V_2 := \lceil V/2 \rceil$
4. compute the largest $0 \le j \le m$ such that $Z \ge \sum_{i=0}^{j-1} \binom{V_1}{i}\binom{V_2}{m-i}$
5. $Y := Z - \sum_{i=0}^{j-1} \binom{V_1}{i}\binom{V_2}{m-i}$
6. $Z_1 := Y \operatorname{div} \binom{V_2}{m-j}$ and $Z_2 := Y \bmod \binom{V_2}{m-j}$
7. return $\texttt{encSet}(V_1, j, Z_1) \cup (\texttt{encSet}(V_2, m-j, Z_2) + V_1)$

To construct the set, the algorithm is a standard recursion. All possible sets are listed in increasing order of the number of elements in $[V_1]$. We compute this number, and then recurse into the two halves. $Z$ can be recovered by the following algorithm, assuming the set generated is in the universe $X + [V]$. For technical reasons that we will encounter later, sometimes we may only have access to the *complement* of the set. The bit $b$ indicates whether we should take the complement.

**decoding algorithm** $\texttt{decSet}(X, V, m, b)$:
1. if $m = 0$ or $m = V$, return 0
2. $V_1 := \lfloor V/2 \rfloor$ and $V_2 := \lceil V/2 \rceil$
3. $j := \texttt{rank}_S(X + V_1 - 1) - \texttt{rank}_S(X - 1)$
4. if $b$, then $j := V_1 - j$
5. $Z_1 := \texttt{decSet}(X, V_1, j, b)$ and $Z_2 := \texttt{decSet}(X + V_1, V_2, m - j, b)$
6. return $Z := \sum_{i=0}^{j-1} \binom{V_1}{i}\binom{V_2}{m-i} + Z_1 \cdot \binom{V_2}{m-j} + Z_2$

We will store the sums $\sum_{i=0}^{j-1} \binom{V_1}{i}\binom{V_2}{m-i}$ and the binomial coefficients $\binom{V_2}{m-j}$ in the lookup table. Since $V$ decreases by a factor of two during the recursion, the depth of the recursion is at most $\lg V$. The size $m$ is split into $j$ and $m - j$, and the algorithm terminates when $m = 0$, hence, there are at most $m$ branches in total at each level of the recursion. The size of the recursion tree is $O(m \lg V)$. Thus, we have the following claim.

CLAIM 8.7. `decSet` *uses* $O(m \lg V)$ *arithmetic operations on* $O(\lg \binom{V}{m})$*-bit integers, as well as* $O(m \lg V)$ `rank` *queries.*

*Preprocessing into a set.* Given a string $\mathcal{D} = (K_h, M, K_t)$ of length at most $\lg \binom{V}{m} - m(V-1)2^{-\kappa+2}$, we preprocess it into a set $S \subseteq [V]$ of size $m$ using two algorithms `prepIntoSet` and `prepIntoTwo`, which are mutually recursive.

**preprocessing algorithm** $\texttt{prepIntoSet}(V, m, \mathcal{D} = (K_h, M, K_t))$:
1. if $2m > V$
2. return $[V] \setminus \texttt{prepIntoSet}(V, V - m, \mathcal{D})$
3. if $m \le 24\kappa$
4. rewrite $\mathcal{D}$ as an nonnegative integer $Z < \text{range}[K_h] \cdot \text{range}[K_t] \cdot 2^{|M|}$
5. return $\texttt{encSet}(V, m, Z)$
6. if $|\mathcal{D}| \le 24\kappa$
7. return $\texttt{prepIntoSet}(48\kappa, 24\kappa, \mathcal{D}) \cup \{V - (m - 24\kappa), \ldots, V - 1\}$

(to be cont'd)

If $m$ is larger than $V/2$, we simply work on the complement. If $m$ is $O(\kappa)$, we view the entire string $\mathcal{D}$ as an integer, and call `encSet`. If the string is too short while $m$ (and $V$) are still large, we decrease $m$ and $V$, and reduce it to the $m = O(\kappa)$ case. Note that $\binom{V}{m}$ may be at most $2^{O(\kappa^2)}$, $Z$ occupies $O(\kappa)$ words (as $\kappa = \Theta(w)$).

Otherwise, we extract an integer $j$ from $\mathcal{D}$.

8. $V_1 := \lfloor V/2 \rfloor$ and $V_2 := \lceil V/2 \rceil$
9. compute $s_j := \lg \binom{V_1}{j} + \lg \binom{V_2}{m-j} - m(V-2)2^{-\kappa+2}$
10. apply Proposition 4.23 for $j \in \{\lfloor m/3 \rfloor + 1, \ldots, 2\lfloor m/3 \rfloor\}$ and $C = \lfloor m/3 \rfloor$, extract $j$ from $\mathcal{D}$ and obtain a pair $(j, \mathcal{D}_j)$ such that $\mathcal{D}_j$ has length at most $s_j$
11. let $(S_1, S_2) :=$ `prepIntoTwo`$(V_1, V_2, j, m-j, \mathcal{D}_j)$
12. return $S_1 \cup (S_2 + V_1)$

`prepIntoTwo` transforms $\mathcal{D}_j$ into two sets of sizes $j$ and $m-j$ over the two halves of the universe (see below). Then we return their union. Proposition 4.23 requires that the length of $\mathcal{D}$ is at least $3\kappa + 2$ and at most $\lg(\sum_j 2^{s_j}) - (C-1)2^{-\kappa+2}$. This is true, because on one hand, the length of $\mathcal{D}$ is at least $24\kappa$; on the other hand,

$$
\begin{aligned}
&2^{s_1} + \cdots + 2^{s_C} \\
&= \sum_{j=\lfloor m/3 \rfloor + 1}^{2\lfloor m/3 \rfloor} \binom{V_1}{j} \cdot \binom{V_2}{m-j} \cdot 2^{-m(V-2)2^{-\kappa+2}} \\
&\geq 2^{-m(V-2)2^{-\kappa+2}} \cdot \left( \binom{V}{m} - \frac{2m}{3} \binom{V_1}{\lfloor m/3 \rfloor} \binom{V_2}{\lceil 2m/3 \rceil} \right) \\
&= 2^{-m(V-2)2^{-\kappa+2}} \cdot \left( \binom{V}{m} - \frac{2m}{3} \binom{V_1}{\lfloor m/2 \rfloor} \binom{V_2}{\lceil m/2 \rceil} \right. \\
&\qquad \left. \cdot \prod_{j=\lfloor m/3 \rfloor + 1}^{\lfloor m/2 \rfloor} \frac{j(V_2 - m + j)}{(V_1 - j + 1)(m - j + 1)} \right) \\
&\geq 2^{-m(V-2)2^{-\kappa+2}} \cdot \binom{V}{m} \cdot \left( 1 - \frac{2m}{3} \cdot \prod_{j=\lfloor m/3 \rfloor + 1}^{\lfloor m/2 \rfloor} \frac{j}{m - j + 1} \right) \\
&\geq 2^{-m(V-2)2^{-\kappa+2}} \cdot \binom{V}{m} \cdot \left( 1 - \frac{2m}{3} \cdot e^{-\sum_{j=\lfloor m/3 \rfloor + 1}^{\lfloor m/2 \rfloor} \frac{m-2j+1}{m-j+1}} \right) \\
&\geq 2^{-m(V-2)2^{-\kappa+2}} \cdot \binom{V}{m} \cdot \left( 1 - \frac{2m}{3} \cdot e^{-m/24} \right).
\end{aligned}
$$

Therefore, by the fact that $m \geq 24\kappa$, we have

$$
\begin{aligned}
\lg(2^{s_1} + \cdots + 2^{s_C}) &\geq \lg \binom{V}{m} - m(V-2)2^{-\kappa+2} - m2^{-\kappa} \\
&\geq s + m2^{-\kappa+2} - m2^{-\kappa} \\
&\geq s + (C-1) \cdot 2^{-\kappa+2}.
\end{aligned}
$$

Thus, the premises of Proposition 4.23 are also satisfied. Next, we describe `prepIntoTwo`.

**procedure** `prepIntoTwo`$(V_1, V_2, m_1, m_2, \mathcal{D})$:

1. let $s_1 := \lg \binom{V_1}{m_1} - m_1(V_1 - 1)2^{-\kappa+2}$ and $s_2 := \lg \binom{V_2}{m_2} - m_2(V_2 - 1)2^{-\kappa+2}$
2. apply Proposition 4.20, split $\mathcal{D}$ into $\mathcal{D}_1$ and $\mathcal{D}_2$ of lengths at most $s_1$ and $s_2$ respectively
3. let $S_1 :=$ `prepIntoSet`$(V_1, m_1, \mathcal{D}_1)$ and $S_2 :=$ `prepIntoSet`$(V_2, m_2, \mathcal{D}_2)$
4. return $(S_1, S_2)$

Proposition 4.20 requires that the length of $\mathcal{D}$ is at most $s_1 + s_2 - 2^{-\kappa+2}$ (and at least $3\kappa$), and $s_1, s_2 \geq 3\kappa$. It is easy to verify the former. For the latter, because $m_1 + m_2 \leq V/2$, $m_2/2 \leq m_1 \leq 2m_2$ and $m_1 + m_2 > 24\kappa$, and in particular, we have $V_1 \geq 24\kappa$ and $m_1 \in [V_1/3, 2V_1/3]$. Hence, we have

$$
\binom{V_1}{m_1} \geq 3^{8\kappa},
$$

and it implies $s_1 \geq 8\kappa$. Similarly, we also have $s_2 \geq 8\kappa$.

*Lookup table.* We store the following lookup table.

**lookup table** $\texttt{tableInt}_{V,m}$:
1. if $m \leq 24\kappa$
2. $\quad \sum_{i=0}^{j-1} \binom{V_1}{i}\binom{V_2}{m-i}$ for $j = 0, \ldots, m$
3. $\quad \binom{V_2}{j}$ for all $j = 0, \ldots, m$
4. else
5. $\quad$ lookup table from Proposition 4.23 for line 10 of `prepIntoSet`
6. include all tables $\texttt{tableInt}_{V',m'}$ for $V' = \lfloor V/2^i \rfloor$ or $V' = \lceil V/2^i \rceil$ for $i \geq 1$, and $0 \leq m' \leq m$

The lookup table $\texttt{tableInt}_{V,m}$ itself has size at most $O(\kappa)$ words for $m > 24\kappa$ and $O(\kappa^2)$ words for $m \leq 24\kappa$. Including the smaller tables, its total size is at most $O(\kappa^2 m + \kappa^4) \leq O(\kappa^{c+2})$ words for $m \leq \kappa^c$ and $c \geq 2$.

*Accessing string $\mathcal{D}$ by* `rank` *queries to $S$.* Suppose $S$ is the set generated from a string $\mathcal{D}$ using the above preprocessing algorithm. In the following, we show how to access $\mathcal{D}[a_1, a_2]$ for $a_2 - a_1 < \kappa$, assuming `rank` queries can be computed efficiently on $S$. We will have two procedures `request` and `reqFromTwo`, which are mutually recursive. Assuming the set $S$ restricted to $X + [V]$ (with $m$ elements in this range) is generated from a string $\mathcal{D} = (K_h, M, K_t)$, $\texttt{request}(X, V, m, \text{range}[K_h], |M|, \text{range}[K_t], a_1, a_2, b)$ recovers $\mathcal{D}[a_1, a_2]$, where $b$ indicates if we take the complement of $S$.

**accessing algorithm** $\texttt{request}(X, V, m, \text{range}[K_h], |M|, \text{range}[K_t], a_1, a_2, b)$:
1. if $2m > V$
2. $\quad m := V - m$ and $b := \neg b$
3. if $m \leq 24\kappa$
4. $\quad Z := \texttt{decSet}(x, V, m, b)$
5. $\quad$ rewrite $Z$ as a string $\mathcal{D} = (K_h, M, K_t)$
6. $\quad$ return $\mathcal{D}[a_1, a_2]$
7. if $\lg(\text{range}[K_h]) + |M| + \lg(\text{range}[K_t]) \leq 24\kappa$
8. $\quad$ return $\texttt{request}(X, 48\kappa, 24\kappa, \text{range}[K_h], |M|, \text{range}[K_t], a_1, a_2, b)$

(to be cont'd)

If $S$ has small size, we recover the whole data structure using `decSet`. If $\mathcal{D}$ is too short, we reduce $m$ and $V$.

9. $V_1 := \lfloor V/2 \rfloor$, $V_2 := \lceil V/2 \rceil$
10. ask `rank` queries and compute $j := \texttt{rank}_S(X + V_1 - 1) - \texttt{rank}_S(X - 1)$
11. if $b$, then $j := V_1 - j$
12. find the size of $\mathcal{D}_j = (K_{j,h}, M_j, K_{j,t})$ in the lookup table

(to be cont'd)

We compute $j$, the integer extracted from $\mathcal{D}$, which encodes the number of elements in the first half. We recover the size of $\mathcal{D}_j$, and use the fact that $(M_j, K_{j,t})$ is a suffix of $\mathcal{D}$ (by Proposition 4.23) to recurse.

13. if $a_1 \geq |M| - |M_j|$
14. $\quad$ return $\texttt{reqFromTwo}(X, V_1, V_2, j, m - j, \text{range}[K_{j,h}], |M_j|, \text{range}[K_{j,t}], a_1 - (|M| - |M_j|), a_2 - (|M| - |M_j|), b)$
15. else
16. $\quad$ recover $\mathcal{D}_j[-1, a_2 - (|M| - |M_j|)] :=$ $\texttt{reqFromTwo}(X, V_1, V_2, j, m - j, \text{range}[K_{j,h}], |M_j|, \text{range}[K_{j,t}], -1, a_2 - (|M| - |M_j|), b)$
17. $\quad$ compute $\mathcal{D}[a_1, a_2]$ using Proposition 4.23

`reqFromTwo` recovers the requested substring of $\mathcal{D}$ assuming `rank` queries to the set generated from `prepIntoTwo`. Since $(M_j, K_{j,t})$ is a suffix of $\mathcal{D}$, if $\mathcal{D}[a_1, a_2]$ is entirely contained in this range, we simply recursive on $\mathcal{D}_j$. Otherwise, Proposition 4.23 guarantees that the remaining bits can be recovered from $j$ and $K_{j,h}$. Note that in either case, the difference $a_2 - a_1$ does not increase.

Next, we describe `reqFromTwo`.

**procedure** `reqFromTwo`$(X, V_1, V_2, m_1, m_2, \text{range}[K_{j,h}], |M_j|, \text{range}[K_{j,t}], a_1, a_2, b)$:

1. compute the sizes of $\mathcal{D}_1 = (K_{1,h}, M_1, K_{1,t})$ and $\mathcal{D}_2 = (K_{2,h}, M_2, K_{2,t})$, which $\mathcal{D}$ is split into

(to be cont'd)

Suppose $S$ restricted to $[X, X + V_1)$ and $[X + V_1, X + V_1 + V_2)$ is generated from $\mathcal{D}$ using `prepIntoTwo`. Then by Proposition 4.20, $(K_{1,h}, M_1)$ is a prefix of $\mathcal{D}$ and $(M_2, K_{2,t})$ is a suffix.

2. if $a_1 \geq |M| - |M_2|$
3. return `request`$(X + V_1, V_2, m_2, \text{range}[K_{2,h}], |M_2|, \text{range}[K_{2,t}], a_1 - (|M| - |M_2|), a_2 - (|M| - |M_2|), b)$
4. if $a_2 < |M_1|$
5. return `request`$(X, V_1, m_1, \text{range}[K_{1,h}], |M_1|, \text{range}[K_{1,t}], a_1, a_2, b)$

(to be cont'd)

If the requested bits $\mathcal{D}[a_1, a_2]$ are entirely contained in $\mathcal{D}_1$ or $\mathcal{D}_2$, we simply recurse on the corresponding substring. In this case, the difference $a_2 - a_1$ does not change either.

6. recover $\mathcal{D}_1[a_1, |M_1|] :=$ `request`$(X, V_1, m_1, \text{range}[K_{1,h}], |M_1|, \text{range}[K_{1,t}], a_1, |M_1|, b)$
7. recover $\mathcal{D}_2[-1, a_2 - (|M| - |M_2|)] :=$ `request`$(X + V_1, V_2, m_2, \text{range}[K_{2,h}], |M_2|, \text{range}[K_{2,t}], -1, a_2 - (|M| - |M_2|), b)$
8. reconstruct $\mathcal{D}[a_1, a_2]$ using Proposition 4.20

Finally, if the requested bits $\mathcal{D}[a_1, a_2]$ split across both substrings, then we make two recursive calls.

*Accessing time.* First observe that `request` has at most $O(\lg m)$ levels of recursion before we call `decSet`. This is because each time $m$ is reduced at least by a factor of $1/3$ by the preprocessing algorithm. The only place where the algorithm makes more than one recursive calls is line 6 and line 7 in `reqFromTwo`. In all other cases, the algorithm makes at most one recursive call with the same (or smaller) difference $a_2 - a_1$. Moreover, we claim that those two lines can only be executed at most once throughout the whole recursion.

CLAIM 8.8. *Line 6 and line 7 in* `reqFromTwo` *can at most be executed once throughout the whole recursion.*

To see this, observe that when these two lines are executed, the two recursive calls will both request either a prefix or a suffix of the substring. Also, as we observed above, the difference $a_2 - a_1$ never increases throughout the recursion. The recursive call that requests a prefix will have $a_1 = -1$ and $a_2 < \kappa - 1$. Thereafter, any subsequence recursive calls in this branch will have $a_1 = -1$ and $a_2 < \kappa - 1$. Since Proposition 4.20 always generates two strings of length at least $3\kappa$, line 4 in `reqFromTwo` is always true (as $|M_1| \geq \kappa - 1$). Line 6 and line 7 will hence not be executed in this branch. The recursive branch that requests a suffix is similar, in which line 2 in `reqFromTwo` is alway true. Hence, the claim holds.

Claim 8.8 implies that the whole recursion tree has at most $O(\lg m)$ nodes, and at most two leaves. In each node, the algorithm spends constant time, and makes two `rank` queries. In each leaf, the algorithm makes one call to `decSet`. As we argued earlier, the integer $Z \leq \lg \binom{V}{m}$ has at most $O(\kappa^2)$ bits (and $O(\kappa)$ words). Since $m \leq O(\kappa)$ when `decSet` is called, by Claim 8.7, each `decSet` takes $O(\kappa^4)$ time ($O(\kappa)$-word numbers take $O(\kappa^2)$ time to multiply or divide), and makes $O(\kappa^2)$ `rank` queries. Combining the above facts, we conclude that `request` runs in $O(\kappa^4)$ time, and it makes at most $O(\kappa^2)$ `rank` queries. This completes the proof of Lemma 8.6. ∎

**8.3. Proof of Lemma 7.1.** Finally, we are ready to prove Lemma 7.1.

To construct the two data structures, we first apply Lemma 8.1 to construct a rank data structure for the secondary block $S_{\text{sec}} \subseteq V + [V_{\text{sec}}]$. Denote this data structure by $\mathcal{D}_{\text{sec}}$. Then we pick a set of $\kappa^{2c-3}$ keys from $S$, as well as $V - \kappa^{2c-3} - \kappa^c$ non-keys from $[V] \setminus S$, and

construct a rank data structure for them, which will be the main data structure. The remaining $\kappa^c$ elements in $[V]$ will correspond to the "unknowns." We pick the two sets based on the bits in $\mathcal{D}_{\text{sec}}$. That is, we apply Proposition 4.20 and split $\mathcal{D}_{\text{sec}}$ into a string of length $\approx \lg \binom{m}{\kappa^{2c-3}}$, a string of length $\approx \lg \binom{V-m}{V-\kappa^{2c-3}-\kappa^c}$ and the remaining bits. Then we apply Lemma 8.6 to interpret the first string as a subset of $S$ and the second string as a subset of $[V] \setminus S$. The final auxiliary data structure consists of the remaining bits of $\mathcal{D}_{\text{sec}}$, as well as a data structure for the "unknowns." Then Lemma 8.6 guarantees that even if the original string is not stored explicitly as part of the final data structure, it can still be accessed implicitly assuming rank queries to the sets, which the data structures support naturally.

*Proof of Lemma 7.1.* We begin by presenting the preprocessing algorithm.

*Preprocessing algorithm.* In the preprocessing algorithm, we first construct a rank data structure for the secondary block, and split it into three substrings.

**Preprocessing algorithm** $\texttt{perfHashBlk}(V, m, V_{\text{sec}}, m_{\text{sec}}, S, S_{\text{sec}}, \mathcal{R})$:

1. run $\mathcal{D}_{\text{sec}} := \texttt{prepRank}(V_{\text{sec}}, m_{\text{sec}}, S_{\text{sec}})$ (from Lemma 8.1) to construct a rank data structure for $S_{\text{sec}}$ using
$$s_{\text{sec}} \le \lg \binom{V_{\text{sec}}}{m_{\text{sec}}} + (m_{\text{sec}} - 1) \cdot 2^{-\kappa/2}$$
bits
2. apply Proposition 4.20 twice to split $\mathcal{D}_{\text{sec}}$ into
   - $\mathcal{D}_{\text{sec},1}$: length $\le \lg \binom{m}{m-\kappa^{2c-3}} - \kappa^c(m-1)2^{-\kappa+2}$
   - $\mathcal{D}_{\text{sec},2}$: length $\le \lg \binom{V-m}{\kappa^{2c-3}+\kappa^c-m} - \kappa^c(V-m-1)2^{-\kappa+2}$
   - $\mathcal{D}_{\text{sec},3}$: length $\le s_{\text{sec}} - \lg \binom{m}{m-\kappa^{2c-3}} - \lg \binom{V-m}{\kappa^{2c-3}+\kappa^c-m} + \kappa^c V 2^{-\kappa+2}$

(to be cont'd)

Note that $m - \kappa^{2c-3} \ge \kappa^c/3$ and $\kappa^{2c-3} + \kappa^c - m \ge \kappa^c/3$, therefore, both $\mathcal{D}_{\text{sec},1}$ and $\mathcal{D}_{\text{sec},2}$ have length at least $4\kappa$. For $\mathcal{D}_{\text{sec},3}$, we have $s_{\text{sec}} \ge \lg \binom{V_{\text{sec}}}{m_{\text{sec}}} \ge \kappa^{c+1}$, and thus,

$$\begin{aligned}
&s_{\text{sec}} - \lg \binom{m}{m-\kappa^{2c-3}} - \lg \binom{V-m}{\kappa^{2c-3}+\kappa^c-m} + \kappa^c V 2^{-\kappa+2} \\
&\ge \kappa^{c+1} - (m-\kappa^{2c-3}) \lg V - (\kappa^{2c-3}+\kappa^c-m) \lg V \\
&\ge \kappa^{c+1} - \kappa^c \lg V \\
&\ge 4\kappa.
\end{aligned}$$

The premises of Proposition 4.20 are satisfied. In order to store two *random* subsets in the main data structure, we "XOR" $\mathcal{D}_{\text{sec},1}$ and $\mathcal{D}_{\text{sec},2}$ with the random string $\mathcal{R}$.[14]

3. compute $\mathcal{D}_{\text{sec},1} \oplus \mathcal{R}$ and $\mathcal{D}_{\text{sec},2} \oplus \mathcal{R}$

(to be cont'd)

For double-ended string $\mathcal{D} = (K_h, M, K_t)$, $\mathcal{D} \oplus \mathcal{R}$ is defined as follows: compute the bitwise XOR of $M$ and $\mathcal{R}[1, |M|]$, treat $\mathcal{R}[|M|+1, |M|+2\kappa]$ and $\mathcal{R}[|M|+2\kappa+1, |M|+4\kappa]$ as two $2\kappa$-bit integers, and compute $(K_h + \mathcal{R}[|M|+1, |M|+2\kappa]) \bmod \text{range}[K_h]$ and $(K_t + \mathcal{R}[|M|+2\kappa+1, |M|+4\kappa]) \bmod \text{range}[K_t]$; $\mathcal{D} \oplus \mathcal{R}$ is the double-ended string (with the same length as $\mathcal{D}$), formed by the outcomes. In particular, since $\text{range}[K_h]$ and $\text{range}[K_t]$ are both smaller than $2^{\kappa+1} \ll 2^{2\kappa}$, when $\mathcal{R}$ is uniformly random, $\mathcal{D} \oplus \mathcal{R}$ is very close to uniform. We have the following claim by standard information theory.

CLAIM 8.9. *For any fixed $\mathcal{D} = (K_h, M, K_t)$ and uniformly random $\mathcal{R}$, we have*

$$H(\mathcal{D} \oplus \mathcal{R}) \ge (\lg(\text{range}[K_h]) + |M| + \lg(\text{range}[K_t]))(1 - 2^{-\kappa+2})$$

[14] We wish to "XOR" $\mathcal{D}_{\text{sec},1}$ and $\mathcal{D}_{\text{sec},2}$ with $\mathcal{R}$, because on one hand, we want to ensure that the subsets stored in the main data structure are random, and on the other hand, we want to ensure that which subsets are stored also encode information about the input (hence, we cannot simply use $\mathcal{R}$).

$$\geq (|\mathcal{D}| - 2^{-\kappa+2})(1 - 2^{-\kappa+2}).$$

*Proof.* When $\mathcal{R}$ is uniformly random, by construction, the bitwise XOR of $M$ and $\mathcal{R}[1, |M|]$ has entropy $|M|$. $(K_h + \mathcal{R}[|M| + 1, |M| + 2\kappa])$ can take $2^{2\kappa}$ different values, each with $2^{-2\kappa}$ probability. Hence, at most $\lceil 2^{2\kappa}/\mathrm{range}[K_h] \rceil$ values can be congruent modulo $\mathrm{range}[K_h]$. The maximum probability of any value $(K_h + \mathcal{R}[|M| + 1, |M| + 2\kappa]) \bmod \mathrm{range}[K_h]$ can take is at most

$$2^{-2\kappa} \cdot \lceil 2^{2\kappa}/\mathrm{range}[K_h] \rceil \leq 1/\mathrm{range}[K_h] + 2^{-2\kappa}.$$

Since $\mathrm{range}[K_h] \leq 2^{\kappa+1}$, this is at most $(1 + 2^{-\kappa+1})/\mathrm{range}[K_h]$. In particular,

$$\begin{aligned}
&H((K_h + \mathcal{R}[|M| + 1, |M| + 2\kappa]) \bmod \mathrm{range}[K_h]) \\
&\geq \lg(\mathrm{range}[K_h]) - \lg(1 + 2^{-\kappa+1}) \\
&\geq \lg(\mathrm{range}[K_h]) - 2^{-\kappa+2} \\
&\geq (1 - 2^{-\kappa+2}) \lg(\mathrm{range}[K_h]).
\end{aligned}$$

Similarly, we have

$$H((K_t + \mathcal{R}[|M| + 2\kappa + 1, |M| + 4\kappa]) \bmod \mathrm{range}[K_t]) \geq (1 - 2^{-\kappa+2}) \lg(\mathrm{range}[K_t]).$$

Since the three components are independent, we prove the claim by summing up the bounds on their entropy. ∎

Also, $K_h, K_t$ and any $O(\kappa)$ consecutive bits of $M$ can be computed in constant time, given random access to $\mathcal{D} \oplus \mathcal{R}$ and $\mathcal{R}$. Next, we interpret the $\mathcal{D}_{\mathrm{sec},1} \oplus \mathcal{R}$ and $\mathcal{D}_{\mathrm{sec},2} \oplus \mathcal{R}$ as two subsets using Lemma 8.6.

4. run $S_1 := \texttt{prepIntoSet}(m, m - \kappa^{2c-3}, \mathcal{D}_{\mathrm{sec},1} \oplus \mathcal{R})$ (from Lemma 8.6) to interpret $\mathcal{D}_{\mathrm{sec},1} \oplus \mathcal{R}$ as a set $S_1 \subseteq [m]$ of size $m - \kappa^{2c-3}$
   run $S_2 := \texttt{prepIntoSet}(V - m, \kappa^{2c-3} + \kappa^c - m, \mathcal{D}_{\mathrm{sec},2} \oplus \mathcal{R})$ to interpret $\mathcal{D}_{\mathrm{sec},2} \oplus \mathcal{R}$ as a set $S_2 \subseteq [V - m]$ of size $\kappa^{2c-3} + \kappa^c - m$
5. compute $S_{\mathrm{unk}} \subseteq S$ according to $S_1$
   compute $\overline{S}_{\mathrm{unk}} \subseteq [V] \setminus S$ according to $S_2$

(to be cont'd)

More specifically, for each $i \in [m]$, $S_{\mathrm{unk}}$ contains the $(i+1)$-th smallest element in $S$ if and only if $i \in S_1$. Similarly, $\overline{S}_{\mathrm{unk}}$ contains the $(i+1)$-th smallest element in $[V] \setminus S$ if and only if $i \in S_2$. They are the keys and non-keys that are *not* to be stored in the main data structure, i.e., the "unknowns."

Then, we compute the main data structure $\mathcal{D}_{\mathrm{main}}$.

6. apply Proposition 4.10 to concatenate the following two data structures, and obtain $\mathcal{D}_{\mathrm{main}}$:
   - $\mathcal{D}_{\mathrm{main},1} := \texttt{prepRank}(V, \kappa^{2c-3}, S \setminus S_{\mathrm{unk}})$ (from Lemma 8.1), a rank data structure
   - $\mathcal{D}_{\mathrm{main},2} := \texttt{prepRank}(V - \kappa^{2c-3}, \kappa^c, \text{"}S_{\mathrm{unk}} \cup \overline{S}_{\mathrm{unk}}\text{"})$ a rank data structure for $S_{\mathrm{unk}} \cup \overline{S}_{\mathrm{unk}}$ over $[V] \setminus (S \setminus S_{\mathrm{unk}})$ (see below)

(to be cont'd)

For $\mathcal{D}_{\mathrm{main},2}$, before running `prepRank`, we first remove all $\kappa^{2c-3}$ elements in $S \setminus S_{\mathrm{unk}}$ from both the universe $[V]$ and $S_{\mathrm{unk}} \cup \overline{S}_{\mathrm{unk}}$, and keep the order of the remaining elements. Thus, the new universe becomes $[V - \kappa^{2c-3}]$. In the other words, $\mathcal{D}_{\mathrm{main},2}$ supports queries of form "return # of elements in $S_{\mathrm{unk}} \cup \overline{S}_{\mathrm{unk}}$ that are no larger than $i$-th smallest element in $[V] \setminus (S \setminus S_{\mathrm{unk}})$".

Finally, we compute the auxiliary data structure $\mathcal{D}_{\mathrm{aux}}$.

7. apply Proposition 4.10 to concatenate the following two data structures and obtain $\mathcal{D}_{\mathrm{aux}}$:
   - $\mathcal{D}_{\mathrm{aux},1} := \texttt{prepRank}(\kappa^c, m - \kappa^{2c-3}, \text{"}S_{\mathrm{unk}}\text{"})$, a rank data structure for $S_{\mathrm{unk}}$ over $S_{\mathrm{unk}} \cup \overline{S}_{\mathrm{unk}}$
   - $\mathcal{D}_{\mathrm{aux},2} := \mathcal{D}_{\mathrm{sec},3}$

Similarly, $\mathcal{D}_{\mathrm{aux},1}$ supports queries of form "return # of elements in $S_{\mathrm{unk}}$ that are no larger than the $i$-th smallest element in $S_{\mathrm{unk}} \cup \overline{S}_{\mathrm{unk}}$."

*Space analysis.* Next, we analyze the length of $\mathcal{D}_{\mathrm{main}}$ and $\mathcal{D}_{\mathrm{aux}}$. $\mathcal{D}_{\mathrm{main}}$ is the concatenation of $\mathcal{D}_{\mathrm{main},1}$ and $\mathcal{D}_{\mathrm{main},2}$. For $\mathcal{D}_{\mathrm{main},1}$, its length is at most

$$\lg \binom{V}{\kappa^{2c-3}} + (\kappa^{2c-3} - 1)2^{-\kappa/2}$$

by Lemma 8.1. For $\mathcal{D}_{\mathrm{main},2}$, its length is at most

$$\lg \binom{V - \kappa^{2c-3}}{\kappa^c} + (\kappa^c - 1)2^{-\kappa/2}.$$

By Proposition 4.10, the length of $\mathcal{D}_{\mathrm{main}}$ is at most

$$\begin{aligned}
&\lg \binom{V}{\kappa^{2c-3}} + (\kappa^{2c-3} - 1)2^{-\kappa/2} + \lg \binom{V - \kappa^{2c-3}}{\kappa^c} + (\kappa^c - 1)2^{-\kappa/2} + 2^{-\kappa+4} \\
&\le \lg \binom{V}{\kappa^{2c-3}, \kappa^c} + \kappa^{2c-3}2^{-\kappa/2+1}.
\end{aligned}$$

$\mathcal{D}_{\mathrm{aux}}$ is the concatenation of $\mathcal{D}_{\mathrm{aux},1}$ and $\mathcal{D}_{\mathrm{aux},2}$. For $\mathcal{D}_{\mathrm{aux},1}$, its length is at most

$$\lg \binom{\kappa^c}{m - \kappa^{2c-3}} + (m - \kappa^{2c-3} - 1)2^{-\kappa/2}.$$

For $\mathcal{D}_{\mathrm{aux},2}$, which is $\mathcal{D}_{\mathrm{sec},3}$, its length is at most

$$\begin{aligned}
&s_{\mathrm{sec}} - \lg \binom{m}{m - \kappa^{2c-3}} - \lg \binom{V - m}{\kappa^{2c-3} + \kappa^c - m} + \kappa^c V 2^{-\kappa+2} \\
&\le \lg \binom{V_{\mathrm{sec}}}{m_{\mathrm{sec}}} - \lg \binom{m}{m - \kappa^{2c-3}} - \lg \binom{V - m}{\kappa^{2c-3} + \kappa^c - m} + 3\kappa^{c+1}2^{-\kappa/2},
\end{aligned}$$

since $V \le 2^{\kappa/2}$ and $m_{\mathrm{sec}} \le 3\kappa^{c+1}$. Summing up the lengths and by Proposition 4.10, the length of $\mathcal{D}_{\mathrm{aux}}$ is at most

$$\begin{aligned}
&\lg \binom{\kappa^c}{m - \kappa^{2c-3}} + (m - \kappa^{2c-3} - 1)2^{-\kappa/2} + \lg \binom{V_{\mathrm{sec}}}{m_{\mathrm{sec}}} \\
&\quad - \lg \binom{m}{m - \kappa^{2c-3}} - \lg \binom{V - m}{\kappa^{2c-3} + \kappa^c - m} + 3\kappa^{c+1}2^{-\kappa/2} + 2^{-\kappa+4} \\
&\le \lg \frac{\kappa^c! \kappa^{2c-3}! (V - \kappa^{2c-3} - \kappa^c)!}{m!(V - m)!} + \lg \binom{V_{\mathrm{sec}}}{m_{\mathrm{sec}}} + (m - \kappa^{2c-3} + 3\kappa^{c+1})2^{-\kappa/2} \\
&\le \lg \binom{V}{m} + \lg \binom{V_{\mathrm{sec}}}{m_{\mathrm{sec}}} - \lg \binom{V}{\kappa^{2c-3}, \kappa^c} + \kappa^{c+1}2^{-\kappa/2+2},
\end{aligned}$$

as we claimed. This proves item (i) and (ii) in the statement.

*Hash functions.* For $x \in S \cup S_{\mathrm{sec}}$, we define $h(x)$ as follows.

- For $x \in S \setminus S_{\mathrm{unk}}$, let $h(x) := \mathtt{rank}_{S \setminus S_{\mathrm{unk}}}(x) - 1$; they are mapped to $[\kappa^{2c-3}]$.
- For $x \in S_{\mathrm{unk}}$, let $h(x) := \kappa^{2c-3} + \mathtt{rank}_{S_{\mathrm{unk}}}(x) - 1$; they are mapped to $\{\kappa^{2c-3}, \ldots, m - 1\}$.
- For $x \in S_{\mathrm{sec}}$, let $h(x) := m + \mathtt{rank}_{S_{\mathrm{sec}}}(x) - 1$; they are mapped to $\{m, \ldots, m + m_{\mathrm{sec}} - 1\}$.

Similarly, for $x \notin S \cup S_{\text{sec}}$, we define $\overline{h}$ as follows.

- For $x \in ([V] \setminus S) \setminus \overline{S}_{\text{unk}}$, let $\overline{h}(x) := \texttt{rank}_{[V]\setminus S\setminus \overline{S}_{\text{unk}}}(x) - 1$; they are mapped to $[V - \kappa^{2c-3} - \kappa^c]$.
- For $x \in \overline{S}_{\text{unk}}$, let $\overline{h}(x) := V - \kappa^{2c-3} - \kappa^c + \texttt{rank}_{\overline{S}_{\text{unk}}}(x) - 1$; they are mapped to $\{V - \kappa^{2c-3} - \kappa^c, \ldots, V - m - 1\}$.
- For $x \in \{V, \ldots, V+V_{\text{sec}}-1\}\setminus S_{\text{sec}}$, let $\overline{h}(x) := V-m+\texttt{rank}_{(V+[V_{\text{sec}}])\setminus S_{\text{sec}}}(x)-1$; they are mapped to $\{V - m, \ldots, V + V_{\text{sec}} - m - m_{\text{sec}} - 1\}$.

Overall, $h$ is a bijection between $S \cup S_{\text{sec}}$ and $[m + m_{\text{sec}}]$, and $\overline{h}$ is a bijection between $[V+V_{\text{sec}}]\setminus(S\cup S_{\text{sec}})$ and $[V+V_{\text{sec}}-m-m_{\text{sec}}]$. Moreover, $h(S) \supset [\kappa^{2c-3}]$ and $\overline{h}([V]\setminus S) \supset [V - \kappa^{2c-3} - \kappa^c]$. This proves item (iii) in the statement.

*Lookup table.* We store the following information in the lookup table.

**lookup table** $\texttt{tableBlk}_{V,V_{\text{sec}}}$:

1. lookup table for line 6 from Proposition 4.10
2. $\texttt{tableRank}_{V,\kappa^{2c-3}}$, $\texttt{tableRank}_{V-\kappa^{2c-3},\kappa^c}$ from Lemma 8.1
3. for *all* $m \in [\kappa^{2c-3} + \kappa^c/3, \kappa^{2c-3} + 2\kappa^c/3]$ and $m_{\text{sec}} \in [\kappa^{c+1}, 2\kappa^{c+1}]$
   - lookup tables for line 7 from Proposition 4.10
   - $\texttt{tableRank}_{\kappa^c,m-\kappa^{2c-3}}$ and $\texttt{tableRank}_{V_{\text{sec}},m_{\text{sec}}}$ from Lemma 8.1
   - $\texttt{tableInt}_{m,m-\kappa^{2c-3}}$ and $\texttt{tableInt}_{V-m,\kappa^{2c-3}+\kappa^c-m}$ from Lemma 8.6

Each $\texttt{tableRank}$ has size $\tilde{O}(2^{\epsilon\kappa})$ and each $\texttt{tableInt}$ has size $O(\kappa^{c+2})$. The total size of $\texttt{tableBlk}_{V,V_{\text{sec}}}$ is $\tilde{O}(2^{\epsilon\kappa})$.

*The main query algorithm.* We show how to answer each query $x$ in constant time with high probability, by querying only the main data structure (and without knowing $m$). We begin by decoding the two data structures $\mathcal{D}_{\text{main},1}$ and $\mathcal{D}_{\text{main},2}$ from $\mathcal{D}_{\text{main}}$, and query $\mathcal{D}_{\text{main},1}$.

**query algorithm** $\texttt{qalgBlk}_{\text{main}}(V, x)$:

1. decode $\mathcal{D}_{\text{main},1}$ and $\mathcal{D}_{\text{main},2}$ from $\mathcal{D}_{\text{main}}$ using Proposition 4.10
2. $x_r := \mathcal{D}_{\text{main},1}.\texttt{qAlgRank}(V, \kappa^{2c-3}, x)$ (from Lemma 8.1)
3. if $x_r > \mathcal{D}_{\text{main}.1}.\texttt{qAlgRank}(V, \kappa^{2c-3}, x - 1)$
4. return $(1, x_r - 1)$

(to be cont'd)

$x_r$ is the number of elements in $S \setminus S_{\text{unk}}$ that are at most $x$. Line 3 checks if $x \in S \setminus S_{\text{unk}}$. If it is, then $x$ is the $x_r$-th element in $S \setminus S_{\text{unk}}$, and we return its hash value according by the definition of $h$.

5. $x_{\text{unk}} := \mathcal{D}_{\text{main},2}.\texttt{qAlgRank}(V - \kappa^{2c-3}, \kappa^c, x - x_r)$
6. if $x_{\text{unk}} > \mathcal{D}_{\text{main},2}.\texttt{qAlgRank}(V - \kappa^{2c-3}, \kappa^c, x - x_r - 1)$
7. return "unknown"
8. return $(0, x - x_r - x_{\text{unk}})$

If $x \notin S \setminus S_{\text{unk}}$, we query $\mathcal{D}_{\text{main},2}$ to check if $x \in S_{\text{unk}} \cup \overline{S}_{\text{unk}}$ in line 6. Note that $x$ is the $(x - x_r + 1)$-th element in $[V] \setminus (S \setminus S_{\text{unk}})$. If it is, we return "unknown". Otherwise, we know that $x \notin S$, and it is the $(x - x_r - x_{\text{unk}} + 1)$-th element in $[V] \setminus (S \cup \overline{S}_{\text{unk}})$, we return its $\overline{h}$-value. Since $\texttt{qAlgRank}$ has constant query time, $\texttt{qalgBlk}$ also runs in constant time. Clearly, $\texttt{qalgBlk}$ outputs 2-hq$(x)$ when $x \in S$ and $h(x) \in [\kappa^{2c-3}]$, or $x \notin S$ and $\overline{h}(x) \in [V - \kappa^{2c-3} - \kappa^c]$, and otherwise it outputs "unknown". This proves item (iv) in the statement.

Next, we show that the probability that the query algorithm outputs "unknown" is small. To this end, let us fix the input data $S, S_{\text{sec}}$ and query $x$, and let $\mathcal{R}$ be uniformly random. We will show that $S_{\text{unk}}$ is close to a uniformly random subset of $S$ of size $m - \kappa^{2c-3}$, and $\overline{S}_{\text{unk}}$ is close to a uniformly random subset of $[V] \setminus S$ of size $\kappa^{2c-3} + \kappa^c - m$. By Claim 8.9, we have $H(\mathcal{D}_{\text{sec},1} \oplus \mathcal{R}) \geq (|\mathcal{D}_{\text{sec},1}| - 2^{-\kappa+2})(1 - 2^{-\kappa+2})$. Since the division operation in Proposition 4.20 is an injection, we have

$$|\mathcal{D}_{\text{sec},1}| \geq s_{\text{sec}} - |\mathcal{D}_{\text{sec},2}| - |\mathcal{D}_{\text{sec},3}|$$

$$\geq \lg \binom{m}{m-\kappa^{2c-3}} - \kappa^c V 2^{-\kappa+3}.$$

Therefore, $H(\mathcal{D}_{\mathrm{sec},1} \oplus \mathcal{R}) \geq \lg \binom{m}{m-\kappa^{2c-3}} - \kappa^c 2^{-\kappa/2+4}$. Furthermore, since the mapping $\texttt{prepIntoSet}(m, m-\kappa^{2c-3}, \cdot)$ is an injection, we have $H(S_1) \geq \lg \binom{m}{m-\kappa^{2c-3}} - \kappa^c 2^{-\kappa/2+4}$, which in turn implies that

$$H(S_{\mathrm{unk}}) \geq \lg \binom{m}{m-\kappa^{2c-3}} - \kappa^c 2^{-\kappa/2+4},$$

for any fixed $S$ and $S_{\mathrm{sec}}$. By Pinsker's inequality (see Section 3.3), the $\ell_1$ distance between $S_{\mathrm{unk}}$ and a uniformly random subset $\mathcal{U}$ of $S$ of size $m - \kappa^{2c-3}$ is

$$\|S_{\mathrm{unk}} - \mathcal{U}\|_1 \leq O(\sqrt{D_{\mathrm{KL}}(S_{\mathrm{unk}} \,\|\, \mathcal{U})})$$

which by the fact that $\mathcal{U}$ is the uniform distribution, is

$$\begin{aligned} &= O(\sqrt{H(\mathcal{U}) - H(S_{\mathrm{unk}})}) \\ &\leq O(\kappa^{c/2} 2^{-\kappa/4}). \end{aligned}$$

In particular, it implies that for any fixed $x \in S$, the probability that $x \in S_{\mathrm{unk}}$ is at most

$$\frac{m - \kappa^{2c-3}}{m} + O(\kappa^{c/2} 2^{-\kappa/4}) \leq O(\kappa^{-c+3}).$$

By applying the same argument to $\mathcal{D}_{\mathrm{sec},2}$, $S_2$ and $\overline{S}_{\mathrm{unk}}$, we conclude that for any fixed $x \notin S$, the probability that $x \in \overline{S}_{\mathrm{unk}}$ is at most

$$\frac{\kappa^{2c-3} + \kappa^c - m}{V - m} + O(\kappa^{c/2} 2^{-\kappa/4}) \leq O(\kappa^{-c+3}).$$

This proves item (v) in the statement.

*The general query algorithm.* Finally, we describe the query algorithm for all $x \in [V + V_{\mathrm{sec}}]$. We use two different algorithms for $x \in [V]$ and $x \in \{V, \ldots, V + V_{\mathrm{sec}} - 1\}$. We begin by the $x \in [V]$ case ($x$ is in the primary block).

**query algorithm** $\texttt{qalgBlk}(V, m, V_{\mathrm{sec}}, m_{\mathrm{sec}}, x)$:
1. (if $x < V$)
2. $(b, v) := \texttt{qalgBlk}_{\mathrm{main}}(V, x)$
3. if $(b, v)$ is not "unknown"
4. return $(b, v)$
5. let $x_{\mathrm{unk}} := \texttt{rank}_{S_{\mathrm{unk}} \cup \overline{S}_{\mathrm{unk}}}(x)$ (already computed in $\texttt{qalgBlk}_{\mathrm{main}}(V, x)$)

(to be cont'd)

When $\texttt{qalgBlk}_{\mathrm{main}}$ returns "unknown", $x$ is the $x_{\mathrm{unk}}$-th element in $S_{\mathrm{unk}} \cup \overline{S}_{\mathrm{unk}}$. Next, we query $\mathcal{D}_{\mathrm{aux},1}$ to find out whether $x \in S_{\mathrm{unk}}$ or $x \in \overline{S}_{\mathrm{unk}}$ and its rank in the corresponding set. Then we return its $h$ or $\overline{h}$ value according to the definition.

6. apply Proposition 4.10 on $\mathcal{D}_{\mathrm{aux}}$ to decode $\mathcal{D}_{\mathrm{aux},1}$
7. $x_{\mathrm{unk},r} := \mathcal{D}_{\mathrm{aux},1}.\texttt{qAlgRank}(\kappa^c, m - \kappa^{2c-3}, x_{\mathrm{unk}} - 1)$
8. if $x_{\mathrm{unk},r} > \mathcal{D}_{\mathrm{aux},1}.\texttt{qAlgRank}(\kappa^c, m - \kappa^{2c-3}, x_{\mathrm{unk}} - 2)$
9. return $\kappa^{2c-3} + x_{\mathrm{unk},r} - 1$
10. else
11. return $V - \kappa^{2c-3} - \kappa^c + (x_{\mathrm{unk}} - x_{\mathrm{unk},r}) - 1$

Similarly to $\texttt{qalgBlk}_{\mathrm{main}}$, we check if the $x_{\mathrm{unk}}$-th element is in $S_{\mathrm{unk}}$. if it is, then it is the $x_{\mathrm{unk},r}$-th element in $S_{\mathrm{unk}}$. Otherwise, it is the $(x_{\mathrm{unk}} - x_{\mathrm{unk},r})$-th element in $\overline{S}_{\mathrm{unk}}$. In this case ($x \in [V]$), the query algorithm runs in constant time.

Next, we show how to handle $x \in \{V, \ldots, V + V_{\mathrm{sec}} - 1\}$. To this end, let us first assume that we can make random access to $\mathcal{D}_{\mathrm{sec}}$.

12. (if $x \geq V$)
13. apply Proposition 4.10 on $\mathcal{D}_{\text{aux}}$ to decode $\mathcal{D}_{\text{aux},2}$
14. $x_r := \mathcal{D}_{\text{sec}}.\texttt{qAlgRank}(V_{\text{sec}}, m_{\text{sec}}, x - V)$ (from Lemma 8.1)
15. if $x_r > \mathcal{D}_{\text{sec}}.\texttt{qAlgRank}(V_{\text{sec}}, m_{\text{sec}}, x - V - 1)$
16. return $m + x_r - 1$
17. else
18. return $V - m + (x - V - x_r)$

If we had access to $\mathcal{D}_{\text{sec}}$, then the query algorithm would be similar to the previous cases, and it would run in constant time. However, $\mathcal{D}_{\text{sec}}$ is not stored in the data structure explicitly. In the following, we show how `qalgBlk` accesses $\mathcal{D}_{\text{sec}}$ from its implicit representation.

More specifically, `qalgBlk` only needs to access $\mathcal{D}_{\text{sec}}$ when it runs the query algorithm `qAlgRank` on $\mathcal{D}_{\text{sec}}$. By Lemma 8.1, `qAlgRank` runs on a RAM with word-size $\Theta(\kappa)$, i.e., it may request $\Theta(\kappa)$ consecutive bits of the data structure $\mathcal{D}_{\text{sec}}$ during its runtime. To implement such access requests, we first apply Proposition 4.20 to reduce each access to $O(1)$ accesses to $\mathcal{D}_{\text{sec},1}$, $\mathcal{D}_{\text{sec},2}$ and $\mathcal{D}_{\text{sec},3}$. $\mathcal{D}_{\text{sec},3}$ is stored as $\mathcal{D}_{\text{aux},2}$, which has been decoded. Each access to it can be implemented in constant time. For $\mathcal{D}_{\text{sec},1}$, $\mathcal{D}_{\text{sec},1} \oplus \mathcal{R}$ is interpreted as a set $S_1 \subseteq [m]$ of size $m - \kappa^{2c-3}$. Lemma 8.6 guarantees that each access to $\mathcal{D}_{\text{sec},1} \oplus \mathcal{R}$ can be implemented in $O(\kappa^4)$ time and $O(\kappa^2)$ `rank` queries to $S_1$, which by the previous argument, implies that each access to $\mathcal{D}_{\text{sec},1}$ can also be implemented in the same time and number of `rank` queries.

On the other hand, the way the preprocessing algorithm "encodes" $S_1$ guarantees that $\texttt{rank}_{S_1}(k)$ queries can be implemented efficiently. To see this, recall that $S_{\text{unk}} \subset S$ is determined according to $S_1$. $\texttt{rank}_{S_1}(k)$ is exactly the number of elements in $S_{\text{unk}}$ that are no larger than the $(k+1)$-th smallest element in $S$. We first do a binary search to find the $(k+1)$-th smallest element in $S$.

**implementing `rank` queries on** $S_1$ $\texttt{rank\_S}_1(k)$:
1. decode $\mathcal{D}_{\text{main},1}, \mathcal{D}_{\text{main},2}$ and $\mathcal{D}_{\text{aux},1}$
2. binary search for $(k+1)$-th element $x^*$ in $S$: given $x \in [V]$,
   (i) $x_r := \mathcal{D}_{\text{main},1}.\texttt{qAlgRank}(V, \kappa^{2c-3}, x)$
   (ii) $x_{\text{unk}} := \mathcal{D}_{\text{main},2}.\texttt{qAlgRank}(V - \kappa^{2c-3}, \kappa^c, x - x_r)$
   (iii) $\texttt{rank}_S(x) := x_r + \mathcal{D}_{\text{aux},1}.\texttt{qAlgRank}(\kappa^c, m - \kappa^{2c-3}, x_{\text{unk}} - 1)$

$x_r$ is the number of elements in $S \setminus S_{\text{unk}}$ that are at most $x$. $x_{\text{unk}}$ is the number of elements in $S_{\text{unk}} \cup \overline{S}_{\text{unk}}$ that are at most $x$. $\mathcal{D}_{\text{aux},1}.\texttt{qAlgRank}(\kappa^c, m - \kappa^{2c-3}, x_{\text{unk}} - 1)$ computes the number of elements in $S_{\text{unk}}$ that are at most $x$. By summing up $x_r$ and $\mathcal{D}_{\text{aux},1}.\texttt{qAlgRank}(\kappa^c, m - \kappa^{2c-3}, x_{\text{unk}})$, we compute $\texttt{rank}_S(x)$, the number of elements in $S$ that are at most $x$, in constant time. Being able to compute $\texttt{rank}_S(x)$ for any given $x$ allows us to binary search for the $(k+1)$-th smallest element $x^*$ in $S$ in $O(\lg V) = O(\kappa)$ time, which then allows us to compute $\texttt{rank}_{S_1}(k)$.

3. $x_r^* := \mathcal{D}_{\text{main},1}.\texttt{qAlgRank}(V, \kappa^{2c-3}, x^*)$
4. return $\texttt{rank}_{S_1}(k) := k - x_r^* + 1$

This shows that $\texttt{rank}_{S_1}(k)$ can be computed in $O(\kappa)$ time, and thus, each access to $\mathcal{D}_{\text{sec},1}$ can be implemented in $O(\kappa^4 + \kappa \cdot \kappa^2) = O(\kappa^4)$ time.

Similarly, each access to $\mathcal{D}_{\text{sec},2}$ can be implemented in $O(\kappa^4)$ time: Lemma 8.6 reduces it to $O(\kappa^2)$ `rank` queries to $S_2$ and $O(\kappa^4)$ processing time; For $\texttt{rank}_{S_2}(k)$, we do binary search to find the $(k+1)$-th element in $[V] \setminus S$; By querying $\mathcal{D}_{\text{main},1}$, $\mathcal{D}_{\text{main},2}$ and $\mathcal{D}_{\text{aux},1}$, we compute $\texttt{rank}_{S_2}(k)$.

Overall, the above algorithms allow us to access $\mathcal{D}_{\text{sec},1}$, $\mathcal{D}_{\text{sec},2}$ and $\mathcal{D}_{\text{sec},3}$ in $O(\kappa^4)$ time, which in turn, allows us to access $\mathcal{D}_{\text{sec}}$ in $O(\kappa^4)$. Thus, `qalgBlk` runs in $O(\kappa^4)$ time. This proves item (vi) in the statement, and completes the proof of Lemma 7.1. □

**9. Perfect Hashing for Sets of Any Size.** In this section, we generalize the data structure from Section 7 to arbitrary universe sizes $U$ and set sizes $n$, proving our main theorem

(this is the formal version of Theorem 1.3).

THEOREM 9.1 (main theorem). *For any constant $\epsilon > 0$, there is a preprocessing algorithm* `perfHash`*, a query algorithm* `qAlg` *and lookup tables* $\texttt{table}_{U,n}$ *of size $n^\epsilon$, such that given*

- *a set $S$ of $n$ keys over the key space $[U]$,*
- *a uniformly random string $\mathcal{R}$ of length $O(\lg^{12} n)$,*

`perfHash` *preprocesses $S$ into a data structure $\mathcal{D}$ of (worst-case) length*

$$\mathbf{OPT}_{U,n} + O(\lg \lg U),$$

*such that $\mathcal{D}$ defines* 2-hq*, a 2-PHM for $S$. Given access to $\mathcal{D}$, $\mathcal{R}$ and $\texttt{table}_{U,n}$, for any key $x \in [U]$, $\texttt{qAlg}(U, n, x)$ outputs* 2-hq$(x)$ *on a RAM with word-size $w = \Omega(\lg U)$, in time*

- *$O(1)$ with probability $1 - O(\lg^{-7} U)$ and*
- *$O(\lg^7 U)$ in worst case,*

*where the probability is taken over the randomness in $\mathcal{R}$. In particular, the query time is constant in expectation and with high probability.*

Recall that 2-hq$(x)$ returns a pair $(b, v)$ such that $b$ indicates whether $x \in S$, and $v = h(x)$ when $x \in S$ and $v = \overline{h}(x)$ when $x \notin S$.

*Proof.* As in the proof of Theorem 7.3, we assume $2n \leq U$ (otherwise, we take the complement of $S$). Then if $n \geq U^{1/12}$, Theorem 7.3 already gives the desired result. From now on, we assume $n < U^{1/12}$.

We partition $[U]$ into $n^{12}$ blocks. A typical set $S$ has all the keys in different blocks. In this case, we may view the universe size being only $n^{12}$, and apply Theorem 7.3. On the other hand, only roughly $1/n^{11}$-fraction of the inputs have at least one pair of keys in the same block. Hence, the optimal space to store those inputs is $\mathbf{OPT} - 11 \lg n$, which suggests that we can afford $10 \lg n$ extra bits.

*No collision in blocks and last block empty.* More specifically, given $S$ such that $n = |S| < U^{1/12}$, let $V := \lceil U \cdot n^{-12} \rceil$ be the block size. We partition $U$ into blocks: $(U \operatorname{div} V)$ blocks of size $V$ and one last block of size $(U \bmod V)$. Let us first only consider inputs that have at most one key in every block and no key in the last block. We apply Theorem 7.3 on the universe of all blocks. That is, let the universe size $U_{\text{new}} = U \operatorname{div} V$, number of keys $n_{\text{new}} = n$. We construct the new set $S_{\text{new}}$ such that $i \in S_{\text{new}}$ if and only if block $i$ contains a key $x \in S$. By Theorem 7.3, we construct a data structure of size

$$\lg \binom{U_{\text{new}}}{n_{\text{new}}} + 1/U_{\text{new}} = \lg \binom{U \operatorname{div} V}{n} + 1/(U \operatorname{div} V),$$

which defines hash functions $h_{\text{new}}$ and $\overline{h}_{\text{new}}$. Besides this data structure, we also apply Lemma 5.1 to store for each $i \in S_{\text{new}}$, the key $x$ within block $i$, according to $h_{\text{new}}(i)$. That is, we store $x - (i-1)V$ in coordinate $h_{\text{new}}(i)$. Hence, this part takes

$$n \lg V + (n-1)2^{-\kappa+5}$$

bits. Then we apply Proposition 4.10 to concatenate the two data structures. The total space is at most

$$\begin{aligned} &\lg \binom{U \operatorname{div} V}{n} + 1/(U \operatorname{div} V) + n \lg V + n \cdot 2^{-\kappa+5} \\ \leq\ &\lg \frac{V^n \prod_{i=0}^{n-1} (U \operatorname{div} V - i)}{n!} + O(1/n) \end{aligned}$$

$$
\begin{aligned}
&\le \lg \frac{\prod_{i=0}^{n-1}(U-iV)}{n!} + O(1/n) \\
&\le \lg \frac{\prod_{i=0}^{n-1}(U-i)}{n!} + O(1/n) \\
&= \lg \binom{U}{n} + O(1/n).
\end{aligned}
$$

To define the hash functions in this case, for each $x \in S$, which is in block $i$, we simply let $h(x) := h_{\text{new}}(i)$, the hash value of the block. For $x \notin S$ in block $i$,

- if $i \in S_{\text{new}}$, let $x^*$ be the key in block $i$,
  - if $x < x^*$, we let $\overline{h}(x) := (V-1) \cdot h_{\text{new}}(i) + (x - (i-1)V)$,
  - if $x > x^*$, we let $\overline{h}(x) := (V-1) \cdot h_{\text{new}}(i) + (x - (i-1)V - 1)$,
- if $i \notin S_{\text{new}}$, we let $\overline{h}(x) := (V-1)n + V \cdot \overline{h}_{\text{new}}(i) + (x - (i-1)V)$.
- if $x$ is in the last block, we let $\overline{h}(x) := (U \operatorname{div} V) \cdot V - n + (x - (U \operatorname{div} V) \cdot V)$

That is, we order all non-keys in block $i$ for $i \in S_{\text{new}}$ first, in the increase order of $(h_{\text{new}}(i), x)$; then we order all non-keys not in the last block, in the increasing order of $(\overline{h}_{\text{new}}(i), x)$; finally we order all non-keys in the last block.

To answer a query $x$ in block $i$, we first query if $i \in S_{\text{new}}$. If $i \notin S_{\text{new}}$, then we know $x$ is not a key, calculate $\overline{h}(x)$ by its definition, and return. Otherwise, we query the $(h_{\text{new}}(i)+1)$-th value in the second data structure, using Lemma 5.1, to retrieve the key in block $i$. If $x$ happens to be this key, we return $(1, h_{\text{new}}(i))$. Otherwise, $x \notin S$, and $\overline{h}(x)$ can be calculated by its definition. Finally, for queries $x$ in the last block, $x$ is not a key, and we calculate $\overline{h}(x)$ according to its definition.

*Exist collision in blocks or keys in last block.* Next, we consider the case where at least one block contains more than one key, or the last block contains at least one key. We spend the first $3\lceil \lg n \rceil$ bits to store

- $N$, the number of blocks with at least two keys (blocks with collisions, or simply *collision blocks*),
- $n_{\text{coll}}$, the total number of keys in all collision blocks,
- $n_{\text{last}}$, the number of keys in the last block.

Next, we apply Lemma 7.12, and construct a membership data structure for $N$ collision blocks using

$$\lg \binom{U \operatorname{div} V}{N} + O(N)$$

bits, which defines a bijection $h_{\text{coll}}$ between all collision blocks and $[N]$, and a bijection $\overline{h}_{\text{coll}}$ between all other blocks (except for the last block) and $[(U \operatorname{div} V) - N]$.

The final data structure has three *more* components:

1. store all keys in $N$ collision blocks using Lemma 7.12, where each element $x$ in block $i$ is stored as $V \cdot h_{\text{coll}}(i) + (x - (i-1)V)$, which uses at most

   $$\lg \binom{NV}{n_{\text{coll}}} + O(n_{\text{coll}} + \lg\lg V)$$

   bits;
2. store all other $(U \operatorname{div} V) - N$ blocks using the data structure for no collisions, where each element $x$ in block $i$ is stored as $V \cdot \overline{h}_{\text{coll}}(i) + (x - (i-1)V)$, which uses at most

   $$\lg \binom{(U \operatorname{div} V)V - NV}{n - n_{\text{coll}} - n_{\text{last}}} + 1$$

   bits;

3. store the last block using Lemma 7.12, which uses

$$\lg \binom{U \bmod V}{n_{\text{last}}} + O(n_{\text{last}} + \lg\lg V)$$

bits.

Summing up the sizes of these three data structures, we get

$$\lg \binom{NV}{n_{\text{coll}}} \binom{(U \operatorname{div} V)V - NV}{n - n_{\text{coll}} - n_{\text{last}}} \binom{U \bmod V}{n_{\text{last}}} + O(n_{\text{coll}} + n_{\text{last}} + \lg\lg V).$$

By the fact that $\binom{n}{k} \le (en/k)^k$ and $n_{\text{coll}} \ge 2N$, the first term is at most

$$\begin{aligned}
&n_{\text{coll}} \lg \frac{NV}{n_{\text{coll}}} + (n - n_{\text{coll}} - n_{\text{last}}) \lg \frac{(U \operatorname{div} V)V - NV}{n - n_{\text{coll}} - n_{\text{last}}} + n_{\text{last}} \lg \frac{U \bmod V}{n_{\text{last}}} + n \lg e \\
&\le n_{\text{coll}} \lg \frac{V}{2} + (n - n_{\text{coll}} - n_{\text{last}}) \lg \frac{U}{n - n_{\text{coll}} - n_{\text{last}}} + n_{\text{last}} \lg V + n \lg e \\
&\le n \lg \frac{eU}{n} + n_{\text{coll}} \lg \frac{nV}{2U} + (n - n_{\text{coll}} - n_{\text{last}}) \lg \frac{n}{n - n_{\text{coll}} - n_{\text{last}}} + n_{\text{last}} \lg \frac{nV}{U}
\end{aligned}$$

which by the fact that $V \le 2U \cdot n^{-12}$, is at most

$$\begin{aligned}
&\le n \lg \frac{eU}{n} + n_{\text{coll}} \lg n^{-11} + (n - n_{\text{coll}} - n_{\text{last}}) \lg \left(1 + \frac{n_{\text{coll}} + n_{\text{last}}}{n - n_{\text{coll}} - n_{\text{last}}}\right) \\
&\qquad + n_{\text{last}} \lg(2n^{-11}) \\
&\le n \lg \frac{eU}{n} + n_{\text{coll}} \lg n^{-11} + (n_{\text{coll}} + n_{\text{last}}) \lg e + n_{\text{last}} \lg(2n^{-11}) \\
&\le n \lg \frac{eU}{n} - (n_{\text{coll}} + n_{\text{last}})(11 \lg n - O(1)).
\end{aligned}$$

On the other hand, by Stirling's formula,

$$\begin{aligned}
\lg \binom{U}{n} &= \lg \frac{U!}{n!(U-n)!} \\
&\ge \lg \frac{\sqrt{U} U^U}{\sqrt{n} n^n \cdot \sqrt{U-n} (U-n)^{U-n}} - O(1) \\
&\ge n \lg \frac{U}{n} + (U-n) \lg \frac{U}{U-n} - \frac{1}{2} \lg n - O(1)
\end{aligned}$$

which by the fact that $\ln(1+x) \ge x - x^2/2$ for $x \ge 0$, is at least

$$\begin{aligned}
&n \lg \frac{U}{n} + (U-n) \left(\frac{n}{U-n} - \frac{n^2}{2(U-n)^2}\right) \lg e - \frac{1}{2} \lg n - O(1) \\
&= n \lg \frac{eU}{n} - \frac{n^2 \lg e}{2(U-n)} - \frac{1}{2} \lg n - O(1)
\end{aligned}$$

which by the fact that $n^2 \ll U$, is

$$\ge n \lg \frac{eU}{n} - \frac{1}{2} \lg n - O(1).$$

Thus, the total size of the data structure when $N \geq 1$ is at most

$$\begin{aligned}
&\lg \binom{U}{n} + \frac{1}{2}\lg n - (n_{\text{coll}} + n_{\text{last}})(11\lg n - O(1)) \\
&\quad + 3\lg n + \lg \binom{U \operatorname{div} V}{N} + O(N + \lg\lg U) \\
\leq\ &\lg \binom{U}{n} + \frac{1}{2}\lg n - (n_{\text{coll}} + n_{\text{last}})(11\lg n - O(1)) \\
&\quad + 3\lg n + 12N\lg n + O(N + \lg\lg U)
\end{aligned}$$

which by the fact that $n_{\text{coll}} \geq 2N$, is at most

$$\begin{aligned}
&\leq \lg \binom{U}{n} + \frac{1}{2}\lg n - (n_{\text{coll}} + n_{\text{last}})(5\lg n - O(1)) + 3\lg n + O(\lg\lg U) \\
&= \mathbf{OPT}_{U,n} - \lg n + O(\lg\lg U).
\end{aligned}$$

In this case, the hash functions are defined as follows. For both $h$ and $\overline{h}$, we first order all elements in the $N$ collision blocks according to their hash values from component 1, which are mapped to $[n_{\text{coll}}]$ and $[N \cdot V - n_{\text{coll}}]$ respectively. Then we order all elements in the $(U \operatorname{div} V) - N$ non-collision blocks according to their hash values from component 2, which are mapped to

$$\{n_{\text{coll}}, \ldots, n - n_{\text{last}} - 1\}$$

and

$$\{N \cdot V - n_{\text{coll}}, \ldots, (U \operatorname{div} V) \cdot V - (n - n_{\text{last}}) - 1\}$$

respectively. Finally, we order all elements in the last block according to their hash values from component 3, which are mapped to

$$\{n - n_{\text{last}}, \ldots, n - 1\}$$

and

$$\{(U \operatorname{div} V) \cdot V - (n - n_{\text{last}}), \ldots, U - n - 1\}$$

respectively.

To answer a query $x$ in block $i$, we retrieve $N$, $n_{\text{coll}}$ and $n_{\text{last}}$, and query if $i$ is a collision block and $h(i)$ (or $\overline{h}(i)$). If $i$ is a collision block, we query component 1; if $i$ is not a collision block, we query component 2; if $i$ is the last block, we query component 3. In any case, the hash value of $x$ can be computed according to its definition in constant time.

Finally, we apply Proposition 4.14 to combine the two cases, by fusing a bit indicating whether there is any collision block. The final data structure has space bounded by

$$\begin{aligned}
&\lg\left(2^{\mathbf{OPT}_{U,n} + O(1/n)} + 2^{\mathbf{OPT}_{U,n} - \lg n + O(\lg\lg U)}\right) + 2^{-\kappa+2} \\
&= \mathbf{OPT}_{U,n} + \lg(2^{O(1/n)} + (\lg^{O(1)} U)/n) + 2^{-\kappa+2} \\
&\leq \mathbf{OPT}_{U,n} + O(\lg\lg U).
\end{aligned}$$

The query algorithm is straightforward. To answer a query $x$, we apply Proposition 4.14 to decode the data structure, and the bit indicating whether there is any collision block or any element in the last block. Then we apply the corresponding query algorithm as described above. This finishes the proof of Theorem 9.1. □

*Remark* 9.2. When the $O(\lg\lg U)$ term is at most $0.5\lg n$, the above data structure uses $\mathbf{OPT} + o(1)$ bits. To improve the $O(\lg\lg U)$ term when $U$ is large, we partition the universe into $\lg^{10} U$ blocks, and check if any block has at least two keys. In this case, the fraction of inputs with some block with at least two keys is only $1/\lg^{10} U$ fraction. Therefore, we can afford to "waste" about $10\lg\lg U$ bits, which dominates the $O(\lg\lg U)$ term. This strategy reduces the problem to storing $n$ non-empty blocks among a total of $\lg^{O(1)} U$ blocks, i.e., the universe size is reduced from $U$ to $\lg^{O(1)} U$. Thus, repeatedly applying it improves the $O(\lg\lg U)$ term to $O(\lg\lg\cdots\lg U)$ for logarithm iterated for an arbitrary (constant) number of times.

**10. Discussions and Open Problems.** In this paper, we assumed that the word size $w$ is at least $\Omega(\lg U + \lg\sigma)$, i.e., each (key, value) pair fits in $O(1)$ words. When either the key or the value is larger than $\Theta(w)$ bits, it would take super-constant time to just read the query or write the output on a RAM. The best query time one can hope for is $O((\lg U + \lg\sigma)/w)$.

When $\lg\sigma \gg w$, the only place being affected is Lemma 5.1, where we need to retrieve values longer than one word. Our data structure naturally supports such long answers in optimal time. When $\lg U \gg w$, a similar strategy to Section 9 applies. We view the first $O(w)$ bits of an element in $[U]$ as its "hash value". If it turns out that all keys have different "hash values", it suffices to add the remaining bits of the key into its value. Otherwise, if multiple keys share the same prefix, then we will be able to save $O(w)$ bits for every extra key with the same prefix.

Our dictionary data structure supports each query in constant expected time. A major open question is to design a *deterministic* succinct dictionary with similar bounds, or to prove this is impossible.

OPEN PROBLEM 10.1. *Is there a deterministic dictionary that uses* $\mathbf{OPT} + \mathrm{poly}\lg n + O(\lg\lg U)$ *bits of space and can answer queries in constant time in worst case?*

Our approach crucially relies on sampling a small set of keys to be the "hard queries". There is always a small portion of the data stored using the rank data structure of Pătraşcu, which takes $O(\lg n)$ time to decode. "Derandomizing" this data structure seems to require a completely different strategy. On the other hand, proving lower bounds may also be challenging, as the common strategy of "designing a hard distribution and proving average-case lower bound" is doomed to fail. For any fixed input distribution, we could always fix and hardwire the random bits in the data structure, thus, our data structure uses only $\mathbf{OPT} + 1$ bits of space.

Our data structure only supports value-retrieval queries on a *fixed* set of (key, value) pairs, i.e., it solves the *static* dictionary problem. The *dynamic* dictionary problem further requires the data structure to support (key, value) insertions and deletions. The state-of-the-art dynamic dictionary uses $\mathbf{OPT} + O(\frac{n\lg(U/n)\lg\lg n}{\lg^{1/3} n})$ bits of space and takes constant time for updates and queries with high probability [1, 2].[15] Another open question is whether we can obtain dynamic dictionaries with significantly smaller redundancy, even with expected constant time.

OPEN PROBLEM 10.2. *Is there a dynamic dictionary that uses* $\mathbf{OPT} + n/\mathrm{poly}\lg n + O(\lg\lg U)$ *bits of space and supports (key, value) insertions, deletions and value-retrieval queries in constant time in expectation?*

It seems non-trivial to extend our data structure to such updates to the data, even with good

[15] The data structures in [1, 2] fail with at most $1/\mathrm{poly}\, n$ probability over any sequence of $\mathrm{poly}\, n$ operations. As long as it does not fail, the operational time is constant. Therefore, by rebuilding the data structure entirely when it fails, the expected time also becomes constant.

amortized expected time. On the other hand, it turns out that our data structure has $\tilde{O}(n)$ preprocessing time, using a hash table to store a "buffer" of size $n^{1-\epsilon}$ and using the technique of *global-rebuilding* [20], one can get $\tilde{O}(n^{1-\epsilon})$ redundancy, $n^{\epsilon}$ update time and expected constant query time. It is also possible to update a (key, value) pair to a new value in our data structure, in $O(1)$ expected time.

The dependence on $U$ in the redundancy is intriguing. In the RAM model, the dependence is very slow-growing, but still super constant. We believe it is not necessary, but it is unclear how to remove this extra small term. On the other hand, note that in the cell-probe model, it can actually be entirely removed (even for very large $U$). This is because when $U$ is large enough so that $\lg \lg U$ becomes unignorable, we could simply apply Lemma 7.11. This strategy does not work on a word RAM, since it requires a large lookup table, which can only be hardwired in a cell-probe data structure.

OPEN PROBLEM 10.3. *Is there a dictionary that uses* $\mathbf{OPT} + \operatorname{poly} \lg n$ *bits of space on a word RAM and can answer queries in constant time in expectation?*

Finally, most of the lookup tables in the data structure in Theorem 9.1 have size $n^{o(1)}$. It turns out that the only two parts that require $n^{\epsilon}$ for some constant $\epsilon > 0$ are Lemma 8.3 and Claim 7.8. In Lemma 8.3, the lookup table is used to aid the following operation: given two $w$-bit strings $a$, $b$, output $c$ such that $c_i = a_j$ where $j$ is the index of the $i$-th "1" in $b$. In Claim 7.8, the lookup table is used to aid the approximation of $k!$ for a given integer $k$.

OPEN PROBLEM 10.4. *Is there a dictionary that uses* $\mathbf{OPT} + \operatorname{poly} \lg n + O(\lg \lg U)$ *bits of space on a word RAM, can answer queries in constant time in expectation, and requires a lookup table of only* $n^{o(1)}$ *size?*

**Acknowledgment.** The author would like to thank anonymous reviewers for helpful comments.

REFERENCES

[1] Y. ARBITMAN, M. NAOR, AND G. SEGEV, *Backyard cuckoo hashing: Constant worst-case operations with a succinct representation*, in 51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA, IEEE Computer Society, 2010, pp. 787–796.

[2] I. O. BERCEA AND G. EVEN, *A dynamic space-efficient filter with constant time operations*, in 17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands, S. Albers, ed., vol. 162 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 11:1–11:17.

[3] K. BRINGMANN AND K. G. LARSEN, *Succinct sampling from discrete distributions*, in Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013, 2013, pp. 775–782.

[4] A. BRODNIK AND J. I. MUNRO, *Membership in constant time and almost-minimum space*, SIAM J. Comput., 28 (1999), pp. 1627–1640.

[5] H. BUHRMAN, P. B. MILTERSEN, J. RADHAKRISHNAN, AND S. VENKATESH, *Are bitvectors optimal?*, SIAM J. Comput., 31 (2002), pp. 1723–1744.

[6] L. CARTER AND M. N. WEGMAN, *Universal classes of hash functions*, J. Comput. Syst. Sci., 18 (1979), pp. 143–154.

[7] Y. DODIS, M. PĂTRAŞCU, AND M. THORUP, *Changing base without losing space*, in Proc. 42nd ACM Symposium on Theory of Computing (STOC), 2010, pp. 593–602.

[8] A. FIAT AND M. NAOR, *Implicit O(1) probe search*, SIAM J. Comput., 22 (1993), pp. 1–10.

[9] A. FIAT, M. NAOR, J. P. SCHMIDT, AND A. SIEGEL, *Nonoblivious hashing*, J. ACM, 39 (1992), pp. 764–782.

[10] F. E. FICH AND P. B. MILTERSEN, *Tables should be sorted (on random access machines)*, in Algorithms and Data Structures, 4th International Workshop, WADS '95, Kingston, Ontario, Canada, August 16-18, 1995, Proceedings, 1995, pp. 482–493.

[11] M. L. FREDMAN, J. KOMLÓS, AND E. SZEMERÉDI, *Storing a sparse table with O(1) worst case access time*, J. ACM, 31 (1984), pp. 538–544.

[12] M. L. FREDMAN AND D. E. WILLARD, *Surpassing the information theoretic bound with fusion trees*, J. Comput. Syst. Sci., 47 (1993), pp. 424–436.
[13] R. GROSSI, A. ORLANDI, R. RAMAN, AND S. S. RAO, *More haste, less waste: Lowering the redundancy in fully indexable dictionaries*, in 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings, 2009, pp. 517–528.
[14] T. HAGERUP AND T. THOLEY, *Efficient minimal perfect hashing in nearly minimal space*, in STACS 2001, 18th Annual Symposium on Theoretical Aspects of Computer Science, Dresden, Germany, February 15-17, 2001, Proceedings, A. Ferreira and H. Reichel, eds., vol. 2010 of Lecture Notes in Computer Science, Springer, 2001, pp. 317–326.
[15] G. JACOBSON, *Space-efficient static trees and graphs*, in 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989, 1989, pp. 549–554.
[16] S. KULLBACK, *Information Theory and Statistics*, Wiley, New York, 1959.
[17] A. MAKHDOUMI, S. HUANG, M. MÉDARD, AND Y. POLYANSKIY, *On locally decodable source coding*, in 2015 IEEE International Conference on Communications (ICC), 2015, pp. 4394–4399.
[18] P. B. MILTERSEN, *Lower bounds for static dictionaries on rams with bit operations but no multiplication*, in Automata, Languages and Programming, 23rd International Colloquium, ICALP96, Paderborn, Germany, 8-12 July 1996, Proceedings, 1996, pp. 442–453.
[19] P. B. MILTERSEN, N. NISAN, S. SAFRA, AND A. WIGDERSON, *On data structures and asymmetric communication complexity*, J. Comput. Syst. Sci., 57 (1998), pp. 37–49.
[20] M. H. OVERMARS, *The Design of Dynamic Data Structures*, Lecture Notes in Economic and Mathematical Systems, Springer-Verlag, 1983.
[21] R. PAGH, *Low redundancy in static dictionaries with constant query time*, SIAM J. Comput., 31 (2001), pp. 353–363.
[22] R. PAGH, *On the cell probe complexity of membership and perfect hashing*, in Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece, 2001, pp. 425–432.
[23] M. PĂTRAŞCU, *Succincter*, in Proc. 49th IEEE Symposium on Foundations of Computer Science (FOCS), 2008, pp. 305–313.
[24] M. PĂTRAŞCU AND M. THORUP, *Time-space trade-offs for predecessor search*, in Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006, 2006, pp. 232–240.
[25] M. PĂTRAŞCU AND M. THORUP, *Randomization does not help searching predecessors*, in Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007, 2007, pp. 555–564.
[26] M. PĂTRAŞCU AND E. VIOLA, *Cell-probe lower bounds for succinct partial sums*, in Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010, 2010, pp. 117–122.
[27] R. RAMAN, V. RAMAN, AND S. RAO SATTI, *Succinct indexable dictionaries with applications to encoding* k*-ary trees, prefix sums and multisets*, ACM Trans. Algorithms, 3 (2007), p. 43.
[28] J. P. SCHMIDT AND A. SIEGEL, *The spatial complexity of oblivious k-probe hash functions*, SIAM J. Comput., 19 (1990), pp. 775–786.
[29] R. E. TARJAN AND A. C. YAO, *Storing a sparse table*, Commun. ACM, 22 (1979), pp. 606–611.
[30] M. THORUP, *Mihai Pătraşcu: Obituary and open problems*, Bulletin of the EATCS, 109 (2013), pp. 7–13, http://eatcs.org/beatcs/index.php/beatcs/article/view/22.
[31] E. VIOLA, *Bit-probe lower bounds for succinct data structures*, SIAM J. Comput., 41 (2012), pp. 1593–1604.
[32] E. VIOLA, O. WEINSTEIN, AND H. YU, *How to store a random walk*, CoRR, abs/1907.10874 (2019), http://arxiv.org/abs/1907.10874.
[33] A. C. YAO, *Should tables be sorted?*, J. ACM, 28 (1981), pp. 615–628.
[34] H. YU, *Optimal succinct rank data structure via approximate nonnegative tensor decomposition*, in Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019., 2019, pp. 955–966.

**Appendix A. Approximating Binomial Coefficients.** In this section, Claim 7.8 and Claim 7.9 from Section 7.3 are proved.

CLAIM 7.8 (RESTATED). *Both* $s_1 := \mathbf{OPT}_{(k-i+1)V_{\mathrm{bl}},m_1} - (k-i+1)\mathtt{SIZE}_{\mathrm{main}} + (m_1-1)2^{-\kappa/2+2}$ *and* $s_2 := \mathbf{OPT}_{(j-k)V_{\mathrm{bl}},m_2} - (j-k)\mathtt{SIZE}_{\mathrm{main}} + (m_2-1)2^{-\kappa/2+2}$ *can be approximated with an additive error of at most* $2^{-\kappa}$ *in* $O(1)$ *time.*

*Proof.* (sketch) To approximate $s_1$ and $s_2$, we can store an approximation of $\mathtt{SIZE}_{\mathrm{main}}$ with up to $O(\kappa)$ bits of precision in the lookup table. The task reduces to approximate the

two $\mathbf{OPT}$ terms. Recall that

$$\mathbf{OPT}_{V,m} = \lg \binom{V}{m}.$$

In the following, we show that for any given $V, m \leq 2^\kappa$, it is possible to approximate $\lg \binom{V}{m}$ in $O(1)$ time.

$\lg \binom{V}{m}$ can be expanded to $\lg V! - \lg m! - \lg(V-m)!$. We approximate each term separately. By Stirling's formula,

$$\ln k! = k \ln \left(\frac{k}{e}\right) + \frac{1}{2} \ln 2\pi n + \sum_{i=2}^{d} \frac{(-1)^i B_i}{i(i-1)k^{i-1}} + O(k^{-d}),$$

where $B_i$ is the $i$-th Bernoulli number, and $d \geq 2$. For any constant $\epsilon > 0$, by setting $d = \Omega(1/\epsilon)$, the above approximation gives an error of $2^{-\Omega(\kappa)}$ for any $k \geq 2^{\epsilon\kappa}$. We store the Bernoulli numbers in the lookup table, and the formula can be evaluated in constant time. On the other hand, for all $k < 2^{\epsilon\kappa}$, we simply store an approximation of $\lg k!$ in a global lookup table, taking $2^{\epsilon\kappa}$ size. Finally, by approximating $\lg V!$, $\lg m!$ and $\lg(V-m)!$ independently with additive error $2^{-2\kappa-2}$, we obtain an estimation of $\lg \binom{V}{m}$ with additive error smaller than $2^{-2\kappa}$. ∎

CLAIM 7.9 (RESTATED). *For any $V_1, V_2, m \geq 0$, and $0 \leq l \leq m$,*

$$\sum_{i=0}^{l} 2^{\mathbf{OPT}_{V_1,i} + \mathbf{OPT}_{V_2,m-i}}$$

*can be approximated up to an additive error of at most $2^{-\kappa-3} \cdot \sum_{i=0}^{m} 2^{\mathbf{OPT}_{V_1,i} + \mathbf{OPT}_{V_2,m-i}}$ in $O(\kappa^5)$ time.*

*Proof.* (sketch) The goal is to approximate

$$\sum_{i=0}^{l} \binom{V_1}{i} \binom{V_2}{m-i}$$

up to additive error of $2^{-\kappa-3} \cdot \binom{V_1+V_2}{m}$, because

$$2^{\mathbf{OPT}_{V_1,i} + \mathbf{OPT}_{V_2,m-i}} = \binom{V_1}{i} \binom{V_2}{m-i}.$$

To this end, we shall use the following lemma from [34] to approximate binomial coefficients.

LEMMA A.1 ([34]). *For any large integers $V$, $d$ and $0 < a \leq V/2$, such that $d \leq c \cdot a$, there is a polynomial $P_{V,d}$ of degree $d$, such that*

$$\binom{V}{a+x} \leq \binom{V}{a} \cdot \left(\frac{V-a}{a}\right)^x \cdot P_{V,d}(x) \leq \binom{V}{a+x} \cdot (1 + 2^{-\sqrt{d}+8}),$$

*for all integers $x \in [0, c \cdot \sqrt{a}]$, a (small) universal constant $c > 0$. Moreover, given $V$ and $d$, the coefficients of $P_{V,d}$ can be computed in $O(d^{1.5})$ time.*

This lemma allows us to approximate $\sum_{l=a}^{b} \binom{V_1}{l}\binom{V_2}{m-l}$ where $b - a \leq c \cdot \sqrt{a}$, up to a *multiplicative* error of $1 \pm 2^{-2\kappa}$ in $O(\kappa^4)$ time: it reduces approximating the sum to computing $\sum_l \alpha^l \cdot P_1(l) P_2(l)$ for two degree-$O(\kappa^2)$ polynomials $P_1, P_2$.

Let $\overline{m} = \frac{V_1}{V_1+V_2} \cdot m$. For $l < \overline{m} - 2\sqrt{\overline{m} \cdot \kappa}$, we return 0 as the approximation; For $\overline{m} - 2\sqrt{\overline{m} \cdot \kappa} \le l \le \overline{m} + 2\sqrt{\overline{m} \cdot \kappa}$, we divide the range into chunks of size $O(\sqrt{\overline{m}})$, apply Lemma A.1 to approximate $\sum_l \binom{V_1}{l}\binom{V_2}{m-l}$ for each chunk in $O(\kappa^4)$ time, and return the sum; For $l > \overline{m} + 2\sqrt{\overline{m} \cdot \kappa}$, we return (an approximation of) $\binom{V_1+V_2}{m}$ as the estimation. It is not hard to verify that in all cases we return an approximation with desired error. The details are omitted. □

**Appendix B. Dictionary with Linear Redundancy.** In this section, we show a proof sketch of Lemma 7.12, and present a dictionary data structure that uses a linear number of extra bits. Recall that $\mathbf{OPT}_{V,m} := \lg \binom{V}{m}$. For membership queries only, Pagh [21] already obtained a better data structure. The data structure in this section is a generalization of Pagh's static dictionary.

LEMMA 7.12 (RESTATED). *Given a set $S \subset [V]$ of $m$ keys, there is a data structure of size*

$$\mathbf{OPT}_{V,m} + O(m + \lg\lg V),$$

*such that it defines a bijection $h$ between $S$ and $[m]$ and a bijection $\overline{h}$ between $[V] \setminus S$ and $[V - m]$. It supports* 2-hq *queries in constant time.*

*Proof.* We are going to use Pagh's static dictionary as a subroutine. For this reason, let us first give an overview of this data structure. The data structure uses a minimal perfect hashing of Schmidt and Siegel [28]. The hashing has three levels. In the first level, each key $x$ is mapped to $h_{k,p}(x) = (kx \bmod p) \bmod m^2$ *with no collisions*, for a prime $p = \Theta(m^2 \lg V)$ and $k \in [p]$. A random pair $(k, p)$ works with constant probability, and it takes $O(\lg m + \lg\lg V)$ bits to encode the function. This level effectively reduces the universe size from $V$ to $m^2$. Each key $x \in S$ is then represented by a pair $(x^{(1)}, x^{(2)})$ where $x^{(1)} \in [m^2]$ is the hash value, and $x^{(2)} = (x \operatorname{div} p) \cdot \lceil p/m^2 \rceil + (kx \bmod p) \operatorname{div} m^2$ (called the quotient function in [21]). Then $x^{(2)} \le O(V/m^2)$ and $(x^{(1)}, x^{(2)})$ uniquely determines $x$.

In the second level, we apply another hash function from the same family on $x^{(1)}$, $h_{k',p'}(x^{(1)}) = (k'x^{(1)} \bmod p') \bmod m$ to map $x^{(1)}$ to $m$ buckets. This time, we have $p' = \Theta(m^2)$ and $k' \in [p']$. Let $A_i$ be the number of keys mapped to bucket $i$. The hashing guarantees that for a random pair $(k', p')$, the expectation of each $A_i^2$ is bounded by $O(1)$. Similarly, we can represent $x^{(1)}$ further as a pair such that the first component is the hash value in $[m]$, and the second component is the quotient function value, which is at most $O(m)$.

The third level hashing then hashes all keys in the same bucket to different integers. It is applied on $x^{(1)}$: $g_{k_i,p_i}(x^{(1)}) = (k_i x^{(1)} \bmod p_i) \bmod A_i^2$, for $p_i = \Theta(m^2)$ and $k_i \in [p_i]$ such that all keys in the bucket are mapped to different integers. It turns out that a random pair $(k_i, p_i)$ works with constant probability.

The data structure stores the following for the hash functions:

1. the top-level hash functions $(k, p)$ and $(k', p')$,
2. $(k_1, p_1), (k_2, p_2), \ldots$, a list of $O(\lg m)$ (random) choices for the third-level hash functions,
3. for each bucket $i$, the index $\pi_i$ of the first hash function in the list that works.

It turns out that it is possible to use only $O(m)$ *bits* to store the indices $\pi_i$. This is because each second-level hash function works with constant probability, the entropy of each $\pi_i$ is a constant. We can use the Huffman coding for each $\pi_i$ to achieve constant bits per index (which turns out to be the unary representation of $\pi_i$).

These hash functions map all $m$ input keys to $O(m)$ buckets with *no* collisions. By storing a rank data structure (e.g., [23]) among the $O(m)$ buckets using $O(m)$ bits of space, we further map all the non-empty buckets to $[m]$. Finally, we store for each bucket, the

*quotient values* of the input key mapped to it. Hence, it takes $\lg(V/m^2) + \lg m + O(1) = \lg(V/m) + O(1)$ bits to encode each key. Thus, the total space is $m \lg(V/m) + O(m + \lg\lg V) = \lg \binom{V}{m} + O(m + \lg\lg V)$ bits.

This data structure supports membership queries, and naturally defines a bijection $h$ between $S$ and $[m]$, namely $h(x)$ simply being the bucket $x$ is mapped to. To generalize the data structure and define an efficiently computable bijection $\overline{h}$ between $[V] \setminus S$ and $[V - m]$, we apply an approach similar to Section 7.4. To this end, we first store the number of keys $m'$ in $[V - m]$. This is also the number of non-keys in $\{V - m, \ldots, V - 1\}$. We are going to store a mapping that maps all $m'$ non-keys in $\{V - m, \ldots, V - 1\}$ to all $m'$ keys in $[V - m]$.

We then store the above data structure for all keys in $[V - m]$, using

$$m' \lg((V - m)/m') + O(m' + \lg\lg V) \leq \lg \binom{V}{m} + O(m + \lg\lg V)$$

bits, which defines a bijection $h'$ between $S \cap [V - m]$ and $[m']$. Note that this data structure also allows us to "randomly access" all keys. That is, given an index $i \in [m']$, it returns a key $x_i$, such that $\{x_1, \ldots, x_{m'}\}$ is the set of all $m'$ keys in $[V - m]$. Then, we store a rank data structure for $\{V - m, \ldots, V - 1\}$, such that given an $x \in \{V - m, \ldots, V - 1\}$, the query algorithm returns if $x$ is a key, as well as its rank over the set of keys (or non-keys). Hence, it maps all keys in $\{V - m, \ldots, V - 1\}$ to $[m - m']$ and all non-keys to $[m']$. The total space is $\mathbf{OPT}_{V,m} + O(m + \lg\lg V)$.

For each $x \in S$, we define $h(x)$ as follows.

- if $x < V - m$, let $h(x) := h'(x)$;
- if $x \geq V - m$, let $h(x)$ be $m' - 1$ plus the rank of $x$ in $S \cap \{V - m, \ldots, V - 1\}$.

For $x \notin S$, we define $\overline{h}(x)$ as follows.

- if $x < V - m$, let $h(x) := x$;
- if $x \geq V - m$, suppose the rank of $x$ in $\{V - m, \ldots, V - 1\} \setminus S$ is $i$, then let $h(x) := x_i$.

Having stored the above data structures, $h(x)$ or $\overline{h}(x)$ can be computed in constant time. □