Succinct Filters for Sets of Unknown Sizes

Mingmou Liu
State Key Laboratory for Novel Software Technology, Nanjing University, China
http://tcs.nju.edu.cn/files/people/mingmou
liu.mingmou@smail.nju.edu.cn

Yitong Yin
State Key Laboratory for Novel Software Technology, Nanjing University, China
http://tcs.nju.edu.cn/yinyt/
yinyt@nju.edu.cn

Huacheng Yu
Princeton University, NJ, USA
https://www.cs.princeton.edu/~hy2/
yuhch123@gmail.com

Abstract

The membership problem asks to maintain a set $S \subseteq [u]$, supporting insertions and *membership queries*, i.e., testing if a given element is in the set. A data structure that computes exact answers is called a *dictionary*. When a (small) false positive rate ϵ is allowed, the data structure is called a *filter*.

The space usages of the standard dictionaries or filters usually depend on the upper bound on the size of S, while the actual set can be much smaller.

Pagh, Segev and Wieder [28] were the first to study filters with varying space usage based on the *current* $|S|$. They showed in order to match the space with the current set size $n = |S|$, any filter data structure must use $(1 - o(1))n(\log(1/\epsilon) + (1 - O(\epsilon)) \log \log n)$ bits, in contrast to the well-known lower bound of $N \log(1/\epsilon)$ bits, where N is an upper bound on $|S|$. They also presented a data structure with almost optimal space of $(1 + o(1))n(\log(1/\epsilon) + O(\log \log n))$ bits provided that $n > u^{0.001}$, with expected amortized constant insertion time and worst-case constant lookup time.

In this work, we present a filter data structure with improvements in two aspects:

- it has constant worst-case time for all insertions and lookups with high probability;
- it uses space $(1 + o(1))n(\log(1/\epsilon) + \log \log n)$ bits when $n > u^{0.001}$, achieving optimal leading constant for all $\epsilon = o(1)$.

We also present a dictionary that uses $(1 + o(1))n \log(u/n)$ bits of space, matching the optimal space in terms of the current size, and performs all operations in constant time with high probability.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

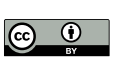
Keywords and phrases Bloom filters, Data structures, Approximate set membership, Dictionaries

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.79

Category Track A: Algorithms, Complexity and Games

Related Version A full version of the paper is available at http://arxiv.org/abs/2004.12465.

Funding *Mingmou Liu*: Part of the research was done when Mingmou Liu was visiting the Princeton University. Mingmou Liu is supported by National Key R&D Program of China 2018YFB1003202 and NSFC under Grant Nos. 61722207 and 61672275.
Yitong Yin: Yitong Yin is supported by National Key R&D Program of China 2018YFB1003202 and NSFC under Grant Nos. 61722207 and 61672275.



1 Introduction

Membership data structures are fundamental subroutines in many applications, including databases [9], content delivery network for web caching [24], image processing [17], scanning for viruses [14], etc. The data structure maintains a set of keys from a key space $[u]$,[1] supporting the following two basic operations:

- insert(x): insert x into the set;
- lookup(x): return `YES` if x is in the set, and `NO` otherwise.

When false positive errors are allowed, such a data structure usually is referred as a *filter*. That is, a filter with false positive rate ϵ may answer `YES` with probability ϵ when x is not in the set (but it still needs to always answer `YES` when x is in the set).

In the standard implementations, a initialization procedure receives the key space size u and a *capacity* N, i.e., an upper bound on the number of keys that can simultaneously exist in the database. Then it allocates sufficient space for the data structure, e.g., a hash table consisting of $\Theta(N)$ buckets. Thereafter, the memory usage is always staying at the maximum, as much space as N keys would take. It introduces inefficiency in the space, when only few keys have been inserted so far. On the other hand, it could also happen that only a rough estimation of the maximum size is known (e.g. [16, 1, 22]). Therefore, to avoid overflowing, one has to set the capacity conservatively. The capacity parameter given to the initialization procedure may be much more than the actual need. To avoid such space losses, a viable approach is to dynamically allocate space such that at any time, the data structure occupies space depending only on the *current* database size (rather than the maximum possible).

For exact membership data structures, it turns out that such promise is not too hard to obtain if one is willing to sacrifice an extra constant factor in space and accept amortization: When the current database has n keys, we set the capacity to $2n$; after n more keys are inserted, we construct a new data structure with capacity equal to $4n$ and transfer the whole database over. The amortized cost to transfer the database is $O(1)$ per insertion. Raman and Rao [29] showed that the extra constant factor in space is avoidable, they designed a *succinct*[2] membership data structure using space $(1+o(1))\log\binom{u}{n}$,[3] where n is the *current* database size, supporting insertions in expected amortized constant time, and lookup queries in worst-case constant time.

For filters, the situation is more complicated. The optimal space to store at most N keys while supporting approximate membership queries with false positive rate ϵ is $N\log 1/\epsilon$ [8, 23] (Pagh, Pagh and Rao [27] achieved $(1+o(1))N\log 1/\epsilon$ bits). However, the above trick to reduce the space may not work in general. This is because the filter data structures do not store perfect information about the database, and therefore, it is non-trivial to transfer to the new data structure with capacity $4n$, as one might not be able to recover the whole database from the previous data structure. In fact, Pagh, Segev and Wieder [28] showed an information theoretical space lower bound of $(1-o(1))n(\log 1/\epsilon + (1-O(\epsilon))\log\log n)$ bits, regardless of the insertion and query times. That is, one has to pay extra $\approx \log\log n$ bits per key in order to match the space with the current database size. They also proposed a data structure with a nearly matching space of $(1+o(1))n\log 1/\epsilon + O(n\log\log n)$ bits when $n > u^{0.001}$, while supporting insertions in expected amortized constant time and lookup

[1] Throughout the paper, $[u]$ stands for the set $\{0,\ldots,u-1\}$.
[2] A succinct data structure uses space equal to the information theoretical minimum plus an asymptotically smaller term called *redundancy*.
[3] All logarithms are base 2.

queries in worst-case constant time. When ϵ is at least $1/\text{poly}\log n$, the extra $\log\log n$ bits per key is dominating. It was proposed as an open problem in [28] whether one can make the $\log\log n$ term succinct as well, i.e., to pin down its leading constant.

On the other hand, an amortized performance guarantee is highly undesirable in many applications. For instances, IP address lookups in the context of router hardware [7, 19], and timing attacks in cryptography [21, 20, 26, 25]. When the database size is always close to the capacity (or when the space is not a concern), it was known how to support all operations in worst-case constant time [13, 3] with high probability. That is, except for a probability of $1/\text{poly}\, n$, the data structure handles *every* operation in a sequence of length $\text{poly}\, n$ in constant time.[4] However, it was not known how to obtain such a guarantee when the space is succinct with respect to the current database size, i.e., $(1+o(1))\log\binom{u}{n}$. For filters, Pagh et al. [28] showed it is possible to get worst-case constant time with high probability, at the price of a constant factor more space $O(n\log 1/\epsilon + n\log\log n)$. They asked if there is a data structure which enjoys the succinct space usage and the worst-case constant time with high probability simultaneously.

1.1 Main Results

In this paper, we design a new dynamic filter data structure that answers both questions. Our data structure has both worst-case constant time with high probability and is succinct in space in terms of the current database size.

▶ **Theorem 1** (Dynamic filter – informal). *There is a data structure for approximate membership with false positive rate ϵ that uses space $(1+o(1))n(\log(1/\epsilon)+\log\log n)$ bits, where $n > u^{0.001}$ is the current number of keys in the database, such that every insertion and lookup takes constant time in the worst case with high probability.*

We also present a dictionary data structure with the space depending on the current n. A dictionary is a generalization of membership data structures, it maintains a set of key-value pairs, supporting

- insert(x, y): insert a key-value pair (x, y) for $x \in [u]$ and v-bit y ;
- lookup(x): if $\exists(x, y)$ in the database, output y; otherwise output `NO`.

By setting $v = 0$, the lookup query simply tests if x is in the database.

▶ **Theorem 2** (Dynamic dictionary – informal). *There is a dictionary data structure that uses space $(1+o(1))n(\log(u/n)+v+O(\log\log\log u))$ bits, where $n > u^{0.001}$ is the current number of key-value pairs in the database, such that every insertion and lookup takes constant time in the worst case with high probability.*

1.2 Related Work

Membership with Constant Time Worst-Case Guarantee. The FKS perfect hashing [15] stores a set of n *fixed* (i.e., static) keys using $O(n)$ space, supporting membership queries in worst-case constant time. Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert and Tarjan [12] introduced an extension of the FKS hashing, which is the first dynamic membership data structure with worst-case constant query time and the expected amortized constant insertion time. Later, Dietzfelbinger and Meyer auf der Heide [13] improved

[4] This is stronger guarantee than expected constant time, since when the unlikely event happened, one could simply rebuild the data structure in linear time. The expected time is still a constant.

the insertion time to worst-case constant, with an overall failure probability of $1/\text{poly}\, n$. Demaine, Meyer auf der Heide, Pagh and Pătraşcu [11] improved the space to $O(n \log(u/n))$ bits of space. Arbitman, Naor and Segev [2] proved that a de-amortized version of cuckoo hashing [19] has constant operation time in the worst case with high probability.

On the other hand, filters can be reduced to dictionaries with a hash function $h : [u] \to [n/\epsilon]$, and thus, all the dictionaries imply similar upper bounds for filters [8].

Succinct Membership. Raman and Rao [29] presented the first succinct dictionary with constant time operations, while the insertion time is amortized. Arbitman, Naor and Segev [3] refined the schema of [2], suggested a succinct dictionary with worst case operation time with high probability.

By using the reduction from [8] and the succinct dictionary from [29], Pagh, Pagh and Rao [27] provided a succinct filter with constant time, while the insertion time is amortized due to [29]. Bender, Farach-Colton, Goswami, Johnson, McCauley and Singh [5] suggested a succinct *adaptive filter*[5] with constant time operation in the worst case with high probability.

Membership for Sets of Unknown Sizes. The data structure of Raman and Rao [29] can be implemented such that the size of the data structure always depends on the "current n". Pagh, Segev and Wieder [28] were the first to study dynamic filters in this setting from a foundational perspective. As we mentioned above, they proved an information-theoretical space lower bound of $(1 - o(1))n(\log(1/\epsilon) + (1 - O(\epsilon)) \log \log n)$ bits for filter, and presented a filter data structure using $n(\log(1/\epsilon) + O(\log \log n))$ bits of space with constant operation time when $n > u^{0.001}$. Indeed, the insertion time is expected amortized, since the succinct dictionary of Raman and Rao is applied as a black box (it was not clear if any succinct dictionary with worst-case operational time can be generalized to this setting).

Very recently, Bercea and Even [6] proposed a succinct membership data structure for maintaining *dictionaries* and *random multisets* with constant operation time. While their data structure is originally designed for the case where an upper bound N on the keys is given (and the space usage is allowed to depend on N), we note that it is possible to extend their solution and reduce the space to depend only on the current n. However, their data structure assumes free randomness, and straightforward extension results in an additive $\Omega(n \log \log u)$ term in space. The redundancy makes their data structure space-inefficient for filters, since the space lower bound is $(1 - o(1))n(\log(1/\epsilon) + (1 - O(\epsilon)) \log \log n)$.

1.3 Previous Construction

As we mentioned earlier, for dynamic membership data structures, if we are willing to pay an extra constant factor in space, one way to match the space with the "current" n is to set the capacity to be $2n$. When the data structure is full after another n insertions, we double the capacity, and transfer the database to the new data structure. However, the standard way to construct an efficient filter is to hash $[u]$ to $[n/\epsilon]$ (where ϵ is the false positive rate) and store all n hash values in a membership data structure, which takes $O(n \log 1/\epsilon)$ bits of space. As we insert more keys and increase the capacity to $4n$, the range of the hash value needs to increase as well. Unfortunately, it cannot be done, because the original keys are not stored, and we have lost the information in order to save space (this is exactly the point of a filter). On the other hand, we could choose to keep the previous

[5] In an adaptive filter, for a negative query x, the false positive event is independent of previous queries.

data structure(s), and only insert the future keys to the new data structure. For each query, if it appears in any of the (at most $\log n$) data structures, we output `YES`. By setting the false positive rate for the data structure with capacity 2^i to $O(\epsilon/i^2)$, the overall false positive rate is at most $\epsilon \cdot \sum_i O(1/i^2) \leq \epsilon$ by union bound. The total space usage becomes roughly $n \log(\log^2 n/\epsilon) = n(\log 1/\epsilon + O(\log \log n))$.

To avoid querying all $\log n$ filters for each query, the previous solution by Pagh et al. [28] uses a *single* global hash function h that maps $[u]$ to $\log(u/\epsilon)$-bit strings for all $\log n$ filters. For a key x in the i-th data structure (with capacity 2^i), one simply takes the first $i + \log 1/\epsilon + 2 \log i$ bits of $h(x)$ as its hash value. Then querying the i-th data structure on y is to check whether the $(i + \log 1/\epsilon + 2 \log i)$-bit prefix of $h(y)$ exists. Since all filters use the same hash function, the overall task is to check whether *some* prefix of $h(y)$ appears in the database, which now consists of strings of various lengths. Note that there are very few short strings in the database, the previous solution extends all short strings to length $\log(n/\epsilon)$ by duplicating the string and appending all possible suffixes, e.g., a string of length $\log(n/\epsilon) - c$ is duplicated into 2^c strings by appending all possible c-bit suffixes. Then all strings are stored in one single dictionary (longer strings are stored according to their first $\log(n/\epsilon)$ bits), and the query becomes to check if the $\log(n/\epsilon)$-bit prefix of $h(y)$ is in the dictionary, which is solved by invoking Raman and Rao [29]. One may verify that duplicating the short strings does not significantly increase the total space, and comparing only the $\log(n/\epsilon)$-bit prefix of a longer string does not increase the false positive rate by much.

1.4 Our Techniques

Our new construction follows a similar strategy, but the "prefix matching" problem is solved differently. Given a collection of $2^{i-1} < n \leq 2^i$ strings of various lengths, we would like to construct a data structure such that given any query $h(y)$, we will be able to quickly decide if any prefix of $h(y)$ appears in the database. The first observation is that the short strings in the database can be easily handled. In fact, all strings shorter than i bits can be stored in a "truth table" of size $2^i = O(n)$. That is, we simply store for all i-bit strings, whether any of its prefix appears in the database. For a query $h(y)$, by checking the corresponding entry of its i-bit prefix, one immediately resolves all short strings. On the other hand, for strings longer than $\log m$ bits, we propose a new (exact) membership data structure, and show that it in fact, automatically solves prefix matching when all strings are long. Before describing its high-level construction in Section 1.4.1, let us first see what it can do and how it is applied to our filter construction.

When the capacity is set to m, the membership data structure stores $n \leq m$ keys from $[u]$ using space $n(\log(u/m) + O(\log \log \log u)) + O(m)$ bits, supporting insertion and membership query in worst-case constant time with high probability. When applying to prefix matching, it stores n strings of length at most ℓ (and more than $\log m$) using $n(\log(2^\ell/m) + O(\log \log \ell)) + O(m)$ bits. Using this data structure with the capacity set to $m = 2^i$, we are able to store the database succinctly when $m/2 < n \leq m$. As we insert more keys to the database, the capacity needs to increase. Another advantage of our membership data structure is that the data can be transferred from the old data structure with capacity m to a new one with capacity $2m$ in $O(m)$ time. More importantly, the transfer algorithm runs almost "in-place", and the data structure remains "queryable" in the middle of the execution. That is, one does not need to keep both data structures in full, at any time the total memory usage is still $n(\log(2^\ell/n) + O(\log \log \ell)) + O(m)$, and the data structure can be queried. Therefore, as n is increasing from $m/2$ to m, we gradually build a new data structure with capacity $2m$. Every time a key is inserted, the background data-transfer

algorithm is run for constant steps. By the time n reaches m, we will have already transferred everything to the new data structure, and will be ready to build the next one with capacity $4m$. Overall, the data structure is going to have $\log n$ stages, the i-th stage handles the $(2^{i-1}+1)$-th to the 2^i-th insertion. In each stage, the database size is doubled, and the data structure also gradually doubles its capacity. This guarantees that the total space is succinct with respect to the current database size, and every operation is handled in constant time with high probability.

Finally, to pin down the leading constant in the extra $O(\log\log n)$ bits, we show that for the n-th inserted key x for $2^{i-1} < n \le 2^i$, storing the $(i + \log(i/\epsilon) + \log\log\log u)$-bit prefix of $h(x)$ balances the false positive rate and the space. Since our new membership data structure only introduces an extra $\approx \log\log i \approx \log\log\log n$ bits of space per key, it is not hard to verify that the total space of our construction is $(1 + o(1))n(\log(1/\epsilon) + \log\log n)$.

1.4.1 Membership Data Structure

In the following, let us briefly describe how our new membership data structure works. The data structure works in the *extendable array* model, as the previous solution by Raman and Rao. See Section 2.2.2 or [29] for more details.

Our main technique contribution is the idea of *data block*. Without the data blocks, our data structure degenerates into a variant of the one proposed in [6]. Instead of a redundancy of $O(n\log\log\log u)$ bits, the degeneration contributes a redundancy of $O(n\log\log u)$ bits, which makes the data structure space-inefficienct for filters as we discussed early.

For simplicity, let us for now assume that we have free randomness, and the first step is to randomly permute the universe. Thus, we may assume that at any time, the database is a uniformly random set (of certain size). We divide the universe into $m/\log u$ *buckets*, e.g., according to the top $\log(m/\log u)$ bits of the key. Then with high probability, every bucket will have $O(\log u)$ keys. We will then dynamic allocate space for each bucket. Note that given that a key is a bucket b, we automatically know that its top $\log(m/\log u)$ bits is "b". Therefore, within each bucket, we may view the keys have lengths only $\log u - \log(m/\log u) = \log((u\log u)/m)$, or equivalently, the universe size being $(u\log u)/m$ (recall that the goal is to store each key using $\approx \log(u/m)$ bits on average).

To store the keys in a bucket, we further divide it into *data blocks* consisting of $O(\log u/\log\log u)$ keys each, based on the time of insertion. That is, the first $O(\log u/\log\log u)$ keys inserted to this bucket will form the first data block, the next $O(\log u/\log\log u)$ keys will be the second data block, etc. Since each data block has few enough keys, they can be stored using a *static* constructions (supporting only queries) using nearly optimal space of $\approx \log\binom{(u\log u)/m}{O(\log u/\log\log u)}$, which is $\log((u\log\log u)/m) = \log(u/m) + \log\log\log u$ bits per key, or a dynamic constructions use $\log(u/m) + O(\log\log u)$ bits per key. The latest data block, which we always insert the new key into, is maintained using the dynamic construction. When it becomes full, we allocate a new data block, and at the same time, we run a in-place *reorganization* algorithm in the background. The reorganization algorithm runs in $O(\log u/\log\log u)$ time, and convert the dynamic construction into the static construction, which uses less space. For each insertion in the future, the reorganization algorithm is run for constant steps, thus, it finishes before the next data block becomes full. Finally, for each bucket, we maintain an *adaptive prefixes* structure [4, 5] to navigate the query to the relevant data block. Roughly speaking, when all $O(\log u)$ keys in the bucket are random, most keys will have a unique prefix of length $\log\log u$. In fact, Bender et al. [4, 5] showed that for every keys, the shortest prefix that is unique in the bucket can be implicitly maintained in constant

time, and the total space for all $O(\log u)$ keys is $O(\log u)$ bits with high probability.[6] We further store for each such unique prefix, which data block contains the corresponding key. It costs $O(\log\log\log u)$ bits per key. Given a query, the adaptive prefix structure is able to locate the prefix that matches the query in constant time, which navigates the query algorithm to the (only) relevant data block. We present the details in Section 4.

2 Preliminaries

2.1 String Notations

Let $\{0,1\}^{\leq \ell} \triangleq \bigcup_{0\leq i\leq \ell}\{0,1\}^i$ and $\{0,1\}^* \triangleq \bigcup_{i\geq 0}\{0,1\}^i$. Given a string $x \in \{0,1\}^\ell$, we use $|x| = \ell$ to denote its length. We denote by $a \circ b$ the concatenation of two strings $a, b \in \{0,1\}^*$. We denote the concatenation of k ones or zeros by 1^k or 0^k, respectively.

For $x, y \in \{0,1\}^*$, we use $x \sqsubseteq y$ (or $y \sqsupseteq x$) to denote that y is a *prefix* of x, formally:

$$x \sqsubseteq y \iff x = y \circ a \text{ for some } a \in \{0,1\}^*. \tag{1}$$

Note that our notation is unconventional: we use $x \sqsubseteq y$ for y prefixing x, to reflect that the Hamming cube identified by x is contained by the Hamming cube for its prefix y.

For two strings x, y such that $|x| \leq |y|$, to compare x and y in lexicographical order, we compare $x\circ \perp^{|y|-|x|}$ and y in lexicographical order, where $\perp$ is a special symbol which is smaller than any other symbol.

Recall that an injection (code) on strings is a *prefix-free code* if no codeword is a prefix of another codeword.

▷ Claim 3. There is a prefix-free code $\texttt{PFC} : \{0,1\}^{\leq \ell} \to \{0,1\}^{\ell+1}$ for strings of length $\leq \ell$.

Proof. Given any $x \in \{0,1\}^{\leq \ell}$, the codeword $\texttt{PFC}(x)$ is $1^{\ell-|x|} \circ 0 \circ x$. ◁

2.2 Computational Models

2.2.1 Random Access Machine

Throughout the paper, we use w to denote the word size: each memory word is a Boolean string of w bits. We assume that the total number of memory words is at most 2^w, and each memory word has an unique address from $[2^w]$, so that any pointer fits in one memory word. We also assume CPU has constant number of registers of size w, and any datapoint fits in constant number of words (i.e. $w = \Omega(v + \log u)$). During each CPU clock tick, CPU may load one memory word to one of its register, write the content of some register to some memory word, or execute the basic operations on the registers. Specifically, the basic operations include four arithmetic operations (addition, subtraction, multiplication, and division), bitwise operations (AND, OR, NOT, XOR, shifting), and comparison.

2.2.2 Memory Models

We use a memory access model known as the *extendable arrays* [29] to model the dynamic space usage.

The extendable array is one of the most fundamental data structures in practice. It is implemented by the standard libraries of most popular programming languages, such as `std::vector` in `C++`, `ArrayList` in `java` and `list` in `python`.

[6] The $O(\log u)$-bit representation is implicit.

▶ **Definition 4** (Extendable arrays). *An* extendable array *of length n maintains a sequence of n fixed-sized elements, each assigned a unique address from $[n]$, such that the following operations are supported:*

- `access`(i)*: access the element with address i;*
- `grow`*: increment the length n, creating an arbitrary element with address $n+1$;*
- `shrink`*: decrement the length n, remove the element with address n.*

A collection of extendable arrays *supports*

- `create`(r)*: create an empty extendable array with element of size r and return its name;*
- `destroy`(A)*: destroy the empty extendable array A;*
- `access`(A,i), `grow`(A), `shrink`(A)*: apply the corresponding operations on array A.*

Each of above operations takes constant time. The space overhead of an extendable array is $O(w)+nr$, where w,n,r are the word size, the length of the array, and the element size respectively. Indeed, the space overhead of a collection of extendable arrays is $O(|\mathcal{A}|w)+\sum_{A\in\mathcal{A}} n_A r_A$, where $\mathcal{A}$, n_A and r_A are the set of extendable arrays, the length of array A, and the element size of array A respectively.

We also consider the following *allocate-free* model.

▶ **Definition 5** (Allocate and free). *In the* allocate-free *model, there are two built-in procedures:*

- `allocate`(b)*: return a pointer to a block of b consecutive memory words which is uninitialized;*
- `free`(p)*: free the block of consecutive memory words which is pointed by p and have been initialized to 0s.*

Each of above operations takes constant time. The total space overhead is $O(|\mathcal{A}|w)+\sum_{A\in\mathcal{A}} n_A w$, where $\mathcal{A}$ is set of all memory blocks and n_A is the length of memory block A.

We discuss the space usages of our data structures in allocate-free model in Section 7.

To avoid the pointer being too expensive in the dynamic memory models, we assume $w=\Theta(\log u)$.

2.3 Random Functions

▶ **Definition 6** (k-wise independent random function). *A random function $h:[u]\to[r]$ is called* k-wise independent *if for any distinct $x_1,\cdots,x_k\in[u]$, and any $y_1,\cdots,y_k\in[r]$,*

$$\Pr_h\left[\bigwedge_{i\le k} h(x_i)=y_i\right]=1/r^k.$$

▶ **Theorem 7** ([31, 10]). *Let $[u]$ be a universe, $w=\Omega(\log u)$, $c_1>0$, $r=\mathrm{poly}(u)$, and $k=u^{o(1)}$. There exists a data structure for a random function $h:[u]\to[r]$ such that*

- *with probability $\ge 1-1/u$, the data structure is constructed successfully;*
- *upon successful construction of the data structure, h is k-wise independent;*
- *the data structure uses space u^{c_1} bits;*
- *for each $x\in[u]$, $h(x)$ is evaluated in $\tilde{O}(1/c_1)$ time in the worst case in the RAM model.*

▶ **Theorem 8** (Chernoff bound with limited independence [30]). *Let $X_1,\cdots,X_n$ be arbitrary k-wise independent boolean random variables with $\Pr[X_i=1]=p$ for any $i\in[n]$. Let $X\triangleq\sum_i X_i$, $\mu\triangleq\mathbb{E}[X]=np$, then for any $\delta>0$, it holds that*

$$\Pr[\,X\ge(1+\delta)\mu\,]\le\exp(-\mu\delta^2/2),$$

as long as $k\ge\lceil\frac{\mu\delta}{1-p}\rceil$.

2.4 Adaptive Prefixes

Given a sequence $S = (x_1, x_2, \ldots)$ of strings, let $\alpha_m(S) = \{\alpha_m(x_1), \alpha_m(x_2), \ldots\}$ be a collection of prefixes, such that for every $x_i \in S$, the $\alpha_m(x_i)$ is the shortest prefix, of length at least m, of the binary representation of x_i, such that $\alpha_m(x_i)$ prefixes no other $x_j \in S$. Note that for any string y, there is at most one $x \in S$ such that $\alpha_m(x) \sqsupseteq y$ as long as $\alpha_m(S)$ exists. In particular, $\alpha_m(S)$ does not exist if there are $i \neq j$ such that $x_i = x_j$.

The prefixes are stored in lexicographical order, thus we refer k-th prefix as the prefix with rank k in lexicographical order.

▶ **Theorem 9** (Refined from [4, 5]). *Let $c_0, c_1 > 1$ be two constants where $c_0 > c_1$. For a random sequence $S = (x_1, \cdots)$ of strings drawn from $(\{0,1\}^{c_0 \log u})^{\leq c_3 \log u}$ uniformly at random with replacement, with probability at least $1 - u^{-c_1}$, the prefix collection $\alpha_{\log\log u}(S)$ exists and can be represented with at most $c_2 \log u$ bits, where $c_2 > 0$ is determined by c_1, c_3. Furthermore, the following operations are supported in constant time:*

- insert(y): *update the representation by inserting a new string $y \in \{0,1\}^{c_0 \log u}$ to S, when there is at most one $x \in S$ such that $\alpha_{\log\log u}(x) \sqsupseteq y$;*
- lookup(y): *given any query $y \in \{0,1\}^{c_0 \log u}$, return the rank of the only $z \in \alpha_{\log\log u}(S)$ that prefixes y, and return `NO` if there does not exist such a z;*
- lowerbound(y): *given any query $y \in \{0,1\}^{\log\log u}$, return the lowest rank of all $z \in \alpha_{\log\log u}(S)$ that $z \sqsubseteq y$, and return 0 if there is no $z \sqsubseteq y$ in the collection;*

For completeness, a proof is provided in the full version of this paper.

3 Data Structures for Sets of Unknown Sizes

In this section, we present our filter and dictionary data structures for sets of unknown sizes.

3.1 The Succinct Dynamic Filters

The following theorem is a formal restatement of Theorem 1.

▶ **Theorem 10** (Dynamic filter – formal). *Let $0 < \epsilon < 1$, $[u]$ the data universe, and $\delta = u^{-C}$, where $C > 1$ is an arbitrary constant. Assume the word size $w = \Theta(\log u)$. There exists a data structure for approximate membership for subsets of unknown sizes of $[u]$, such that*

1. *for any $n = \omega(\log u)$ and $n < u$, the data structure uses $n(\log(1/\epsilon) + \log\log n + O(\log\log\log u))$ bits of space after insertions of any n key, and extra u^c precomputed bits that are independent of the input, where $0 < c < 1$ is an arbitrary small constant;*
2. *each insertion and membership query takes $O(1)$ time in the worst case;*
3. *after each insertion, a failure may be reported by the data structure with some probability, and for any sequence of insertions, the probability that a failure is ever reported is at most δ, where the probability is taken over the precomputed random bits;*
4. *conditioned on no failure, each membership query is answered with false positive rate at most ϵ.*

As we mentioned in the introduction, our data structure has $\log n$ stages when handling n insertions. The i-th stage is from the insertion of the $(2^{i-1}+1)$-th key to the 2^i-th key – the database size doubles after each stage.

The main strategy is to reduce the problem of (approximate) membership to (exact) *prefix matching*. More formally, in the *prefix matching* problem, we would like to maintain a set of binary strings $\{s_1, s_2, \ldots\}$ of possibly different lengths, supporting

- insert(s): add string s to the set;
- query(y): decide of any string s in the set is a prefix of y.

To this end, our filter first applies a *global* hash function h such that $h : [u] \to [u^{c_2}]$ is $(c_1 \log u)$-wise independent according to Theorem 7, where $c_1 > 0$ is a constant to be fixed later, and c_2 is a sufficiently large constant (which in fact, is the c_0 in Theorem 9). To insert a key x in stage i, we calculate its hash value $h(x)$, and then insert the ℓ_i-bit prefix of $h(x)$, for some parameter ℓ_i. To answer a membership query y, we simply calculate $h(y)$ and search if any prefix of $h(y)$ is in the database. If no prefix of $h(y)$ is in the database, we output `NO`; otherwise, we output `YES`. It is easy to see that this strategy will never output any false negatives. On the other hand, by union bound, if the query y is not in the set, the probability that the query algorithm outputs `YES` is at most

$$\sum_{i=1}^{\log u} 2^i \cdot 2^{-\ell_i},$$

since h is $(c_1 \log u)$-wise independent (in particular, it is pairwise independent), then the probability that the ℓ_i-bit prefix of $h(y)$ matches with the prefix of the hash value $h(x)$ of key x is $2^{-\ell_i}$. Hence, by setting

$$\ell_i \triangleq i + \log(1/\epsilon) + \log i + \log\log\log u + 2, \tag{2}$$

the false positive rate is at most

$$\sum_{i=1}^{\log u} 2^i \cdot 2^{-i-\log(1/\epsilon)-\log i-\log\log\log u-2} = \epsilon \cdot \sum_{i=1}^{\log u} \frac{1}{4i \log\log u} < \epsilon.$$

We use $\mathcal{D}_{c_1 \log u}$ to denote the distribution of the random insertion sequence $y_1, y_2, \ldots, y_n$ for prefix matching constructed above. Formally, $\mathcal{D}_{c_1 \log u}$ is the distribution of a sequence of random strings $y_1, y_2, \ldots, y_n$ obtained from $(c_1 \log u)$-wise independent sequence $z_1, z_2, \ldots, z_n \in [u^{c_2}]$ by truncating: $\forall 1 \le j \le n$, $y_j = (z_j)_{\le \ell_i}$, where $i = \lceil \log j \rceil$.

▶ **Lemma 11** (Prefix matching). *Let $\delta = u^{-C}$, where $C > 1$ is an arbitrary constant. There exist a constant c_1 and a deterministic data structure for prefix matching such that*

1. *for any $n = \omega(\log u)$ and $n < u$, the data structure uses $n(\ell_{\lceil \log n \rceil} - \log n + O(\log\log\log u))$ bits of space after n insertions, and extra u^c precomputed bits, where $0 < c < 1$ is an arbitrary small constant;*
2. *each insertion and query takes $O(1)$ time in the worst case;*
3. *after each insertion, a failure may be reported by the data structure, and for a random sequence of insertions drawn from $\mathcal{D}_{c_1 \log u}$, the probability that a failure is ever reported is at most δ, where the probability is taken over $\mathcal{D}_{c_1 \log u}$;*
4. *every query is answered correctly if no "fail" is reported.*

We present the construction in Section 4. Using this prefix matching data structure, the space usage of the filter is

- $n(\ell_{\lceil \log n \rceil} - \log n + O(\log\log\log u)) = n(\log(1/\epsilon) + \log\log n + O(\log\log\log u))$ bits,
- and u^c bits for storing h by Theorem 7 and for the precomputed lookup tables described in the appendix of the full version of this paper, both independent of the operation sequence.

Each insertion and query can be handled in constant time given the data structure does not fail. This proves Theorem 10.

3.2 The Succinct Dynamic Dictionaries

The data structure for prefix matching also works well as a dictionary data structure for the insertions with keys are sampled uniformly at random. A worst-case instance can be converted into a random instance by a random permutation $\pi : [u] \to [u]$. Assuming an idealized $(c_1 \log u)$-wise independent random permutation whose representation and evaluation are efficient, the data structure for prefix matching in Lemma 11 can be immediately turned to a dictionary. However, the construction of k-wise independent random permutation with low space and time costs is a longstanding open problem [18].

We show that our data structure can solve the dictionary problem in the worst case unconditionally, at the expense of extra u^c bits of space for storing random bits which are independent of the input.

▶ **Theorem 12** (Dynamic dictionary – formal). *Let $[u] \times \{0,1\}^v$ be the data universe, and $\delta = u^{-C}$, where $C > 1$ is an arbitrary constant. Assume the word size $w = \Theta(v + \log u)$. There exists a data structure for dictionary for sets of unknown sizes of key-value pairs from $[u] \times \{0,1\}^v$, such that*

1. *for any $n = \omega(\log u)$ and $n < u$, the data structure uses $n(\log(u/n) + v + O(\log\log\log u))$ bits of space after insertions of any n key-value pairs, and extra u^c precomputed bits that are independent of the input, where $0 < c < 1$ is an arbitrary small constant;*
2. *each insertion and query takes $O(1)$ time in the worst case;*
3. *after each insertion, a failure may be reported by the data structure with some probability, and for any sequence of insertions, the probability that a failure is ever reported is at most δ, where the probability is taken over the precomputed random bits;*
4. *conditioned on no failure, each query is answered correctly.*

The details of the data structure are postponed to Secion 6.

4 Prefix Matching Upper Bound

In this section, we prove Lemma 11.

Recall the distribution $\mathcal{D}_{c_1 \log u}$ of random insertion sequence $y_1, y_2, \ldots, y_n$ assumed in Lemma 11. Given an insertion sequence $\bar{y} = (y_1, y_2, \ldots, y_n) \sim \mathcal{D}_{c_1 \log u}$, we define the *core set* $B(\bar{y}) \triangleq \{x \in \bar{y} : \forall x' \in \bar{y}, x = x' \vee x' \not\sqsupseteq x\}$, and its subset $B^{(a,b]} \triangleq \{x \in B : |x| \in (a,b]\}$ for any $a < b$. Let $\mathcal{D}^{(a,b]}_{c_1 \log u}$ denote the distribution of $B^{(a,b]}$. We say that a random sequence Y of strings is drawn from $\mathcal{D}^{(a,b]}_{c_1 \log u}$ if it can be obtained by permuting the random core set $B^{(a,b]}$.

We show that Lemma 11 is true as long as there exist a family of deterministic data structures for prefix matching with known capacity m. An instance of the data structure $D = D(m, \ell)$ is parameterized by capacity $m < u$, and string length upper bound $\ell \geq \log m$. The data structure uses u^c bits extra space whose contents are precomputed lookup tables, and supports following functionalities with good guarantees:

- $\mathsf{initialize}(D)$ and $\mathsf{destroy}(D)$: subroutines for initializing and destroying D respectively. The data structure is successfully initialized (or destroyed) after invoking $\mathsf{initialize}(D)$ ($\mathsf{destroy}(D)$) consecutively for $O(m)$ times. When successfully initialized, D uses space $O(m)$ bits. The $\mathsf{initialize}(D)$'s are invoked before all other subroutines and $\mathsf{destroy}(D)$'s are invoked after all other subroutines.
- $\mathsf{insert}(D, x)$: insert string x to D, where $\log m < |x| \leq \ell$. After n insertions, D uses at most $n(\ell - \log m + 2\log\log\log u) + O(m)$ bits. Each insertion may cause D to fail. A failure ever occurs for a random insertion sequence Y with probability at most u^{-2C}, as long as Y is drawn from $\mathcal{D}^{(\log m, \ell]}_{c_1 \log u}$, where c_1 is suitably determined by constant C.

- query(D, x): return one bit to indicate whether there exists a prefix of x in D. The correct answer is always returned as long as D has not failed.
- decrement(D): try to delete an arbitrary string y in D and return the y if y is deleted. An invoking may delete nothing and hence nothing is returned, but it guarantees that the total number of such empty invoking is at most m. Each invoking that successfully deletes a string frees space $\ell - \log m$ bits. The decrement(D)'s are invoked after all insertions.

▷ Claim 13. Given the deterministic data structures supporting above functionalities in constant time in the worst case, Lemma 11 is true.

Proof. We use an auxiliary structure called *truth table* to deal with short strings. A truth table T_i is a bitmap (i.e. array of bits) of length 2^i and supports the required functionalities in the worst cases:

- T_i is initialized to the all-0 string 0^{2^i}, where each invoking of initialize(T_i) extends T_i by one 0 until T_i is of length 2^i, and each invoking of destroy(T_i) shrinks T_i by one bit until T_i is fully destroyed;
- to insert x where $|x| = i$, we set $T_i[x+1] \leftarrow 1$;[7]
- to query x where $|x| = i$, we return `YES` if $T_i[x+1] = 1$ and return `NO` if otherwise;
- to decrement T_i, we maintain a j that traverses from 1 to 2^i, and at each time set $T_i[j] \leftarrow 0$, return $j-1$ if $T_i[j] = 1$, and increment j by 1.

Initially, the prefix matching data structure required by Lemma 11 consists of T_0, T_1 and $D_0 = D(1, \ell_0), D_1 = D(2, \ell_1)$ respectively with capacities $1, 2$, and string lengths ℓ_0, ℓ_1, where ℓ_i is defined in Eq(2).

To insert x, which is the n-th insertion, we set $i \leftarrow \lceil \log n \rceil$, invoke insert$(D_i, x)$ if there is no prefix of x has been inserted. Then we execute the following procedure for 10 times to maintain our data structure:

1. If T_{i-1} is non-empty, we decrement it by invoking decrement(T_{i-1}). If a y is returned, we insert it into T_i by invoking insert$(T_i, y \circ 0)$ and insert$(T_i, y \circ 1)$.
2. If D_{i-1} is non-empty, we invoke decrement(D_{i-1}). If a y is returned, we insert it into D_i by invoking insert(D_i, y) when $|y| > i$ and insert y into T_i by invoking insert(T_i, y) otherwise.
3. If T_{i-1} (or D_{i-1}) is empty but not destroyed yet, we invoke destroy(T_{i-1}) (or destroy(D_{i-1})).
4. If T_{i-1} (or D_{i-1}) has been destroyed, we invoke initialize(T_{i+1}) (or initialize(D_{i+1}) for $D_{i+1} = D(2^{i+1}, \ell_{i+1})$ with capacity 2^{i+1} and string length upper bound ℓ_{i+1}), where ℓ_i is defined in Eq(2).

A failure is reported whenever a failure is reported during insertion to D_i.

Clearly, for any integer $n \in [2^i, 2^{i+1})$, after n insertions, all inserted strings are stored in either D_{i-1}, T_{i-1} or D_i, T_i. By the time n reaches 2^{i+1}, D_{i+1}, T_{i+1} have been initialized, all inserted strings are stored in D_i, T_i, and D_{i-1}, T_{i-1} have been destroyed.

Consider the insertion sequence for a fixed D_i. Observe that the strings inserted into D_i must be in the core set $B^{(i,\ell_i]}(\bar{y})$. Therefore the insertion sequence is drawn from $\mathcal{D}^{(i,\ell_i]}_{c_1 \log u}$, which means that insertions to each D_i ever failed with probability at most δ. By union bound, a failure is ever reported with probability at most $\sum_{i=1}^{\log u} u^{-2C} \le u^{-C} = \delta$.

Overall, the data structure uses at most $n(\ell_{\lceil \log n \rceil} - \log n + O(\log\log\log u)) \le n(\log(1/\epsilon) + \log\log n + O(\log\log\log u))$ bits after n insertions, besides the u^c precomputed bits.

[7] For A, a list or array of items, we let $A[i]$ denote the i-th item of A.

Suppose n strings has been inserted, let $i \leftarrow \lceil \log n \rceil$. To query x, we invoke $\mathsf{query}(D_{i-1}, x)$, $\mathsf{query}(D_i, x)$, $\mathsf{query}(T_{i-1}, x_{\le i-1})$, $\mathsf{query}(T_i, x_{\le i})$ simultaneously, and return `YES` if any one of the invokings returns `YES`.

Obviously each insertion and query takes constant time in the worst case, and it is easy to check that every query is correctly answered as long as no failure is reported. ◁

5 Succinct Prefix Matching with Known Capacity

We now describe the data structures required by Claim 13. The pseudocodes are given in the appendix of the full version of this paper.

The data structure consists of a main table and $m/\log u$ subtables.

We partition each binary string x into four consecutive parts: $st(x), hd(x), hs(x), rt(x)$ of lengths $\log(m/\log u), \log(\log u/\log\log u), \log\log\log u, |x| - \log m$ respectively. Roughly speaking, a datapoint x will be distributed into a subtable according to $st(x)$, then be put into a data block of size $\log u/\log\log u$ according to the order it is inserted, therefore we can save $|st(x)| + |hd(x)| - O(1)$ bits for each datapoint by properly encoding.

Main Table. The main table consists of $m/\log u$ entries, each of which contains a pointer to a subtable. Each insertion/query x is distributed into an entry of the main table addressed by $st(x)$. Recall the word size $w = \Theta(\log u)$. The main table uses $mw/\log u = O(m)$ bits.

Recall that $\bar{y} = (y_1, y_2, \ldots, y_n) \sim \mathcal{D}_{c_1 \log u}$ is transformed from a $(c_1 \log u)$-wise independent sequence $Z = (z_1, z_2, \cdots, z_n)$ by truncating. The insertion sequence Y is drawn from $\mathcal{D}^{(\log m, \ell]}_{c_1 \log u}$ by permuting $B^{(\log m, \ell]}$, the restriction of the core set $B(\bar{y})$ to the strings whose lengths ranges within $(\log m, \ell]$.

Let Y_i, Z_i denote the subsequences of Y, Z which contain all the strings that have prefix i, respectively. By definitions, $|Y_i| \le |Z_i|$. Recall that Z are $(c_1 \log u)$-wise independent. Due to Theorem 8, the load of entry i exceeds $c_3 \log u$ with probability

$$\Pr[\,|Y_i| \ge c_3 \log u\,] \le \Pr[\,|Z_i| \ge c_3 \log u\,] \le \exp(-(c_3-1)^2 \log u/2), \tag{3}$$

as long as $c_1 \ge \lceil 2(c_3-1)^2 \rceil$. Therefore the max-load of entries of the main table is upper bounded by $c_3 \log u$ with probability at least $1 - (m/\log u)\exp(-(c_3-1)^2 \log u/2)$. The data structure reports failure if any entry of the main table overflows. In the rest of the proof, we fairly assume $|Y_i| \le |Z_i| \le c_3 \log u$ for all i.

Observe that a datapoint x can be identified with $hd(x) \circ hs(x) \circ rt(x)$ if the entry $i = st(x)$ it is distributed into is fixed. Therefore we let Y'_i, Z'_i denote the subsequences generated from Y_i, Z_i by discarding the left-most $\log(m/\log u)$ bits.

Subtable. Each subtable i consists of the following parts to be specified later:

- a collection of fingerprints $\alpha_{\log\log u}(Y'_i)$ and its indicator list I_i;
- an (extendable) array of navigators N_i;
- an (extendable) array of data blocks A_i;
- two buffers, $B_{i,u}, B_{i,r}$;
- constant many other local variables.

All the datapoints are stored in array A_i. Given a datapoint x, it is easy to see that the addresses of the entries which contains the information of x is high correlated with the order it is inserted, since any insertion takes constant time in the worst case. Hence we take the fingerprints $\alpha_{\log\log u}(Y'_i)$, indicators I_i, navigators N_i, and a tricky way to encode a data block as clues to locate the entries which maintain x. Recall that new insertions is put into

the latest data block using a dynamic construction, and we reorganize the full dynamic data block into a static construction. We use buffer $B_{i,u}$ to maintain the dynamic block, and use buffer $B_{i,r}$ to "de-amortize" the reorganization.

At first consider a static version of our data structure. In the static version, the buffers and the indicator list are unnecessary. Let $n_i \triangleq |Y_i|$.

Fingerprints. The collection of fingerprints $\alpha_{\log\log u}(Y_i')$ is obtained by applying Theorem 9 on Y_i' with guarantee $c_1 \geq c_3$. Note that Z_i' are mutually independent as long as $c_1 \geq c_3$. Due to Theorem 9, there exists a constant $c'' > 0$ such that a fingerprint collection $\alpha_{\log\log u}(Z_i')$ for Z_i' can be represented in $c'' \log u$ bits with probability $1 - u^{-c_5}$.

We show that there exists a injective function $P : [|Y_i'|] \to [|Z_i'|]$ such that $\forall j \in [|Y_i'|], Y_i'[j] \sqsupseteq Z_i'[P(j)]$. Due to the injective function P and the guarantee that Y is prefix-free, the fingerprint collection of Y_i' can be represented with the same space and probability guarantees as above.

We define $P : [|Y_i|] \to [|Z_i|]$ by $P(j) \triangleq \min\{k \in [|Z_i|] : Y_i[j] \sqsupseteq Z_i[k]\}$. By the definition, for any $y \in Y_i$, there is $z \in Z$ such that $y \sqsupseteq z$. Recall that all the strings in Y_i has prefix i. Hence for any $y \in Y_i, z \in Z$ such that $y \sqsupseteq z$, it holds that $i \sqsupseteq z$, i.e. $z \in Z_i$. Thus for any $j \in [|Y_i|], \{k \in [n] : Y_i[j] \sqsupseteq Z_i[k]\} \neq \emptyset$. Therefore P is well-defined. On the other hand, for distinct $j, l \in [|Y_i|]$, $\{k \in [n] : Y_i[j] \sqsupseteq Z_i[k]\}$ and $\{k \in [n] : Y_i[l] \sqsupseteq Z_i[k]\}$ are disjoint, since Y_i is prefix-free and there is no such z that x, y prefix z simultaneously for distinct $x, y \in Y_i$. Therefore P is injective. Recall that Y_i', Z_i' are generated by removing the prefix i from the strings in Y_i, Z_i: $\forall j, Y_i[j] = i \circ Y_i'[j], Z_i[j] = i \circ Z_i'[j]$. Therefore P works for Y_i', Z_i' too, i.e. $\forall j \in [|Y_i'|], Y_i'[j] \sqsupseteq Z_i'[P(j)]$.

A failure is reported if any fingerprint collection can not be represented within $c'' \log u$ bits, which occurs with probability at most u^{-c_5}.

The fingerprints are sorted lexicographically, so that by the j-th fingerprint we mean the j-th in lexicographical order. For simplicity, we write $\alpha_i \triangleq \alpha_{\log\log u}(Y_i')$.

A failure is reported if there are more than $c_4 \log u / \log\log u$ datapoints share identical $hd(x)$ and $hs(x)$, which occurs with probability at most

$$\binom{c_3 \log u}{c_4 \log u / \log\log u} \left(\frac{1}{\log u}\right)^{c_4 \log u / \log\log u} < u^{-(c_4 - 0.01)}. \tag{4}$$

The fingerprints cost $O(\log u)$ bits per subtable if no failures.

Navigators. N_i is an array of pointers. For any datapoint, the rank of its fingerprint is synchronized with the index of its navigator. In particular, for the k-th fingerprint in α_i, $N_i[k]$ is the address of the data block which maintains the datapoint with the fingerprint. A data block maintains up to $\log u / \log\log u$ datapoints, thus there are at most $c_3 \log\log u$ data blocks. The navigators cost at most $n_i \log\log\log u + O(n_i)$ bits of space.

Data Blocks. A_i is interpreted as an array of data blocks, with each data block holding up to $\log u / \log\log u$ datapoints.

Consider the following succinct binary representation (called *pocket dictionary* in [6]) of a collection of datapoints $F \in \binom{\{0,1\}^\ell}{m}$: The representation consists of two parts $header(F)$ and $body(F)$. Let $header(x), body(x)$ denote the left-most $\log m$ bits and the right-most $\ell - \log m$ bits of x. Let $n_i' \triangleq |\{x \in F | header(x) = i\}|$, and $F = (x_1, \cdots, x_m)$ sorted lexicographically. Then $header(F) \triangleq 0 \circ 1^{n_0'} \circ 0 \circ 1^{n_1'} \circ 0 \cdots 1^{n_{m-1}'}$ and $body(F) \triangleq body(x_1) \circ body(x_2) \cdots \circ body(x_m)$. It is easy to see that this representation uses $2m + m(\ell - \log m)$ bits of space.

Our static data block is a variant of this representation. Let $(x_1, \cdots, x_{\log u/\log\log u})$ be the sorted list of datapoints maintained by data block j. Data block j consists of a list of headers $(hd(x_1), \cdots, hd(x_{\log u/\log\log u}))$, a list of identities $(hs(x_1), \cdots, hs(x_{\log u/\log\log u}))$ and an array of the rest part of datapoints $(rt(x_1), \cdots, rt(x_{\log u/\log\log u}))$. The header list are represented in the same way as in the pocket dictionary, the identity list is the concatenation $hs(x_1) \circ hs(x_2) \cdots \circ hs(x_{x/\log\log u})$, and the rest part array is the concatenation $\texttt{PFC}(rt(x_1)) \circ \texttt{PFC}(rt(x_2)) \cdots \circ \texttt{PFC}(rt(x_{x/\log\log u}))$, where $\texttt{PFC}(\cdot)$ is the prefix-free code in Claim 3.

Recall that $|hd(x)| = \log(\log u/\log\log u), |hs(x)| = \log\log\log u$. Therefore the data blocks for the subtable cost at most $O(n_i) + n_i(\ell - \log m + \log\log\log u)$ bits of space.

Query in a Static Data Block. Recall that the fingerprints are sorted in lexicographical order, and the indices of the navigators are synchronized with the ranks of corresponding fingerprints. Also note that a navigator costs $\log\log\log u + O(1)$ bits, there are at most $c_4 \log u/\log\log u$ datapoints share $hd(x) \circ hs(x)$ with any query x. By putting everything together, we can retrieve all the navigators of the datapoints which have the same $hd(x) \circ hs(x)$ with query x, and learn a k' such that the unique suspected datapoint is the k'-th datapoint among the datapoints which share the $hd(x) \circ hs(x)$ in its data block. On the other hand, we can retrieve the header list and identity list with constant number of memory accesses. Consequently, we can retrieve the rest part of the suspected datapoint efficiently. See the pseudocode in the appendix of the full version of this paper for more details.

The space usage is upper bounded by $m + (m/\log u) \cdot O(\log u) + n\log\log\log u + O(n) + n(\ell - \log m + \log\log\log u) \leq n(\ell - \log m + 2\log\log\log u) + O(m)$ bits. And the query time is clearly constant.

Insertion. The new insertions will be put into a data block under construction temporarily, and the data block will be reorganized to the static version as long as the data block is full (which means, there are $\log u/\log\log u$ datapoints stored in it). A data block under construction consists of two incomplete lists for headers and identities, and an (extendable) array of the rest parts of datapoints. Note that the space usages of incomplete lists are identical with the ones of the complete lists, it wastes at most $O(\log u)$ bits per dynamic data block.

Reorganizing a data block (i.e. sorting a data block) can be expensive, therefore we finish this work during the procedure that a new data block under construction is being filled. Hence there are two dynamic data blocks, one under construction and one under reorganization, besides the static ones.

Note that the collection of fingerprints α_i can be updated dynamically with small costs. To retrieve the datapoint y with fingerprint $\alpha(y)$, the only things we need are the address of the data block which maintains y and the in-block index of y. (recall that $hd(y)$ and $hs(y)$ are known due to the fingerprint collection.) It is easy to learn the address as long as we know that y is in a dynamic data block, since there are at most two dynamic blocks. We use the buffers to record the in-block index, and use the indicators to inform whether y is in a dynamic data block.

The list of indicators is a string from $\{1, 2, 3\}^{n_i}$. The value of i-th indicator implies which kind of data block the datapoint corresponding to i-th fingerprint is stored in. For a static data block, the query algorithm works in previous way. For a dynamic data block, the address of the data block can be easily learnt with the counter n_i.

The two buffers are arrays of pointers from $[\log u/\log\log u]^{\log u/\log\log u}$, so they fit in constant number of memory cells. In particular, for a indicator $I_i[j]$ which is the k-th indicator has value 2 (or 3), $B_{i,u}[k]$ (or $B_{i,r}[k]$) is the in-block index of the rest part which corresponds to j-th fingerprint.

The new insertion is not a prefix of some preceded insertion due to our guarantees. Therefore, to insert x, we simply append $hs(x), rt(x)$ to the identity list and rest part array, update the header list, fingerprint collection, and indicator list, and insert a proper pointer into $B_{i,u}$, which overall takes constant time. And to query x in a dynamic data block, where x corresponds to a datapoint y stored in the data block, we need to retrieve the address of the data block with counter n_i and the in-block index of y with the buffers, then retrieve $rt(y)$. See the pseudocodes in the appendix of the full version of this paper for more details.

Reorganizing a Data Block. The reorganization procedure starts as long as the data block under construction is full. Informally, the reorganization procedure works as follows:

1. Update the list of indicators and copy $B_{i,r} \leftarrow B_{i,u}$. (We guarantee that the preceded reorganization process has been finished before the buffer $B_{i,u}$ is full)
2. Insert the address of the data block in proper positions of the navigator list.
3. Sort the array of rest parts and the list of identities according to the pointers in the buffer $B_{i,r}$ while keep the buffer updated. Note that the sorting can be done within time cost $O(\log u/\log\log u)$: we enumerate $j \in [\log u/\log\log u]$, find j' such that $B_{i,r}[j'] = j$, swap $hs_j, rt_j, B_{i,r}[j]$ with $hs_{B_{i,r}[j]}, rt_{B_{i,r}[j]}, B_{i,r}[j']$ one by one, where hs_j, rt_j is the j-th item of the corresponding list and array.
4. Update indicator list.

The total time cost is $O(\log u/\log\log u)$, which can be simulated by $\log u/\log\log u$ operations, each costing $O(1)$ time, within a data block. See the appendix of the full version of this paper for more details.

The two dynamic data blocks waste at most $O(\log u)$ bits on the two incomplete lists for headers and identities, therefore we uses at most extra $O(m)$ bits of space.

Setting the Constant Parameters. Our data structure may fail at the load balancing on subtables, constructing the fingerprint collections for subtables, and the load balancing on headers of fingerprint collections. By the union bound, the failure probability is at most

$$\frac{m}{\log u}\left(\exp(-(c_3-1)^2\log u/2) + u^{-c_5}\right) + m2^{-(c_4-0.01)\log u}, \tag{5}$$

when $c_1 \geq \max\{\lceil 2(c_3-1)^2\rceil, c_3\}$. Since $m < u$,the failure probability can be as small as $\delta = u^{-2C}$ if we set the constants c_1, c_3, c_4, c_5 to be sufficiently large.

The initialize, decrement, and destroy subroutines are easy to implement, which are postponed to the appendix of the full version of this paper

6 Unconditional Succinct Dictionary

We show that our data structure can solve the dictionary in the worst case unconditionally. In our data structure for prefix matching, the randomness is used only for:

1. load balancing on the subtables;
2. the representation of the adaptive prefixes;
3. load balancing on the $hd(x) \circ hs(x)$'s.

Note that we should decode $st(x)$ from subtable index i, decode $hd(x) \circ hs(x)$ from bucket index in adaptive prefixes, and decode $hd(x)$ from bucket index in data block. To achieve the identical guarantees with prefix matching, we apply a weaker but strong enough random permutation.

▶ **Definition 14** (Feistel permutation). *Given any $x \in [u]$, let x_L, x_R respectively denote the $\log m$ left-most bits and the $\log(u/m)$ right-most bits of the binary representation of x, so that $x = x_L \circ x_R$. Given any $f : \{0,1\}^{\log(u/m)} \to \{0,1\}^{\log m}$, the* Feistel permutation *$\pi_f : [u] \to [u]$ is defined as*

$$\pi_f(x) = (x_L \oplus f(x_R)) \circ x_R.$$

It is easy to verify that π_f is indeed a permutation. In fact, $\pi_f(\pi_f(x)) = x$ for any x and f.

Our dictionary data structure works with three $(c_1 \log u)$-wise independent hash functions $f : [\frac{u \log u}{m}] \to [m/\log u]$, $g : [u/m] \to [\log u]$, and $h : [u/m] \to [u^{c_2}]$. Given a key $x \in [u]$, we let $st'(x) = \pi_f(x), hd'(x) \circ hs'(x) = \pi_g(hd(x) \circ hs(x) \circ rt(x))$ and $hss(x) = h(rt(x))$. Then we distribute x into subtable $st'(x)$, insert/query $hd'(x) \circ hs'(x) \circ hss(x)$ to the fingerprint collection, and encode $hd'(x) \circ hs'(x) \circ rt(x)$, instead of $hd(x) \circ hs(x) \circ rt(x)$, in its data block.

Consider two datapoints x, x'. If $hd(x) \circ hs(x) \circ rt(x) \neq hd(x') \circ hs(x') \circ rt(x')$, then $st'(x)$ and $st'(x')$ are independent; otherwise $st'(x) \neq st'(x')$. Therefore for any i, c,

$$\Pr\left[\,|\{x \in Y : st'(x) = i\}| \geq c\,\right] \leq \Pr\left[\,|\{x \in Y' : st(x) = i\}| \geq c\,\right], \tag{6}$$

where Y are the insertion sequence, Y' are $(c_1 \log u)$-wise independent random insertion sequence. Hence the load balancing is not worse than the one in the prefix matching case.

Due to Theorem 9 and Eq(6), the fingerprint collection works with the same guarantees. Clearly $hd(x) \circ hs(x) = (hd'(x) \circ hs'(x)) \oplus g(rt(x)), st(x) = st'(x) \oplus f(hd(x) \circ hs(x) \circ rt(x))$, thus the keys can be retrieved precisely. For the values, we store $rt(x)$ and its value together as a tuple in the data block.

7 Upper Bounds in Allocate-Free Model

We mimic the extendable arrays in the allocate-free model. For simplicity, we modify the navigator list from extendable array to an array of length $c_3 \log u$.

Suppose we are dealing with D_i. The main table can be implemented easily since it has fixed length. Let $s = c_3(\log u)(\log(1/\epsilon) + \log i + O(\log\log\log u))$ be the space usage upper bound of any single subtable. For a subtable i, we maintain a pointers array of length $\lceil\sqrt{s/w}\rceil$ to mimic the extendable array. Every pointer in the array points to a memory block of $\lceil\sqrt{sw}\rceil$ bits. Therefore we waste at most

$$(2^i/\log u) \cdot O(w \cdot \sqrt{s/w} + \sqrt{sw}) = O(2^i\sqrt{\log(1/\epsilon) + \log i + \log\log\log u})$$

bits of space. In conclusion, after n insertions our data structure for filters uses at most

$$n(\log(1/\epsilon) + \log\log n + O(\log\log\log u)) + O(n\sqrt{\log(1/\epsilon) + \log\log n + \log\log\log u})$$

bits of space in the allocate-free model.

Similarly, our data structure for dictionaries uses at most

$$n(\log(u/n) + v + O(\log\log\log u)) + O(n\sqrt{\log(u/n) + v + \log\log\log u})$$

bits of space after n insertions in the allocate-free model.

References

1 Paulo Sérgio Almeida, Carlos Baquero, Nuno M Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

2 Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *International Colloquium on Automata, Languages, and Programming*, pages 107–118. Springer, 2009.

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

4 Michael A Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. *arXiv preprint*, 2017. arXiv:1711.01616.

5 Michael A Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 182–193. IEEE, 2018.

6 Ioana O Bercea and Guy Even. Fully-dynamic space-efficient dictionaries and filters with constant number of memory accesses. *arXiv preprint*, 2019. arXiv:1911.05060.

7 Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve ip lookups. In *Proceedings IEEE INFOCOM*, volume 3, pages 1454–1463. IEEE, 2001.

8 Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65, 1978.

9 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

10 Tobias Christiani, Rasmus Pagh, and Mikkel Thorup. From independence to expansion and back again. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 813–820. ACM, 2015.

11 Erik D Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Latin American Symposium on Theoretical Informatics*, pages 349–361. Springer, 2006.

12 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *29th Annual Symposium on Foundations of Computer Science*, pages 524–531. IEEE, 1988.

13 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *International Colloquium on Automata, Languages, and Programming*, pages 6–19. Springer, 1990.

14 O. Erdogan and Pei Cao. Hash-av: fast virus signature scanning by cache-resident filters. In *GLOBECOM '05. IEEE Global Telecommunications Conference, 2005.*, volume 3, pages 6 pp.–, November 2005. doi:10.1109/GLOCOM.2005.1577953.

15 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

16 Deke Guo, Jie Wu, Honghui Chen, and Xueshan Luo. Theory and network applications of dynamic bloom filters. In *Proceedings IEEE INFOCOM*, pages 1–12. IEEE, 2006.

17 Mai Jiang, Chunsheng Zhao, Zaifeng Mo, and Jing Wen. An improved algorithm based on bloom filter and its application in bar code recognition and processing. *EURASIP Journal on Image and Video Processing*, 2018(1):139, December 2018. doi:10.1186/s13640-018-0375-6.

18 Eyal Kaplan, Moni Naor, and Omer Reingold. Derandomized constructions of k-wise (almost) independent permutations. *Algorithmica*, 55(1):113–133, 2009.

19 Adam Kirsch and Michael Mitzenmacher. Using a queue to de-amortize cuckoo hashing in hardware. In *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, volume 75, 2007.

20 Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

21 Richard J Lipton and Jeffrey F Naughton. Clocked adversaries for hashing. *Algorithmica*, 9(3):239–252, 1993.

22 Yi Liu, Xiongzi Ge, David Hung-Chang Du, and Xiaoxia Huang. Par-bf: A parallel partitioned bloom filter for dynamic data sets. *The International Journal of High Performance Computing Applications*, 30(3):259–275, 2016. doi:10.1177/1094342015618452.

23 Shachar Lovett and Ely Porat. A lower bound for dynamic approximate membership data structures. In *IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 797–804, 2010.

24 Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, July 2015. doi:10.1145/2805789.2805800.

25 Moni Naor and Eylon Yogev. Bloom filters in adversarial environments. *ACM Transactions on Algorithms (TALG)*, 15(3):1–30, 2019.

26 Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

27 Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 823–829, 2005.

28 Rasmus Pagh, Gil Segev, and Udi Wieder. How to approximate a set without knowing its size in advance. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 80–89. IEEE, 2013.

29 Rajeev Raman and Satti Srinivasa Rao. Succinct dynamic dictionaries and trees. In *International Colloquium on Automata, Languages, and Programming*, pages 357–368. Springer, 2003.

30 Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff–hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

31 Mikkel Thorup. Simple tabulation, fast expanders, double tabulation, and high independence. In *54th Annual Symposium on Foundations of Computer Science*, pages 90–99. IEEE, 2013.