Algorithmica (2023) 85:1786–1804
https://doi.org/10.1007/s00453-022-01057-0

Dynamic Dictionaries for Multisets and Counting Filters with Constant Time Operations

Ioana O. Bercea[1] · **Guy Even**[1]

Received: 30 August 2021 / Accepted: 15 October 2022 / Published online: 5 December 2022

Abstract

We resolve the open problem posed by Arbitman, Naor, and Segev [FOCS 2010] of designing a dynamic dictionary for multisets in the following setting: (1) The dictionary supports multiplicity queries and allows insertions and deletions to the multiset. (2) The dictionary is designed to support multisets of cardinality at most n (i.e., including multiplicities). (3) The space required for the dictionary is $(1 + o(1)) \cdot n \log \frac{u}{n} + \Theta(n)$ bits, where u denotes the cardinality of the universe of the elements. This space is $1 + o(1)$ times the information-theoretic lower bound for static dictionaries over multisets of cardinality n if $u = \omega(n)$. (4) All operations are completed in constant time in the worst case with high probability in the word RAM model. A direct consequence of our construction is the first dynamic counting filter (i.e., a dynamic data structure that supports approximate multiplicity queries with a one-sided error) that, with high probability, supports operations in constant time and requires space that is $1 + o(1)$ times the information-theoretic lower bound for filters plus $O(n)$ bits. The main technical component of our solution is based on efficiently storing variable-length bounded binary counters and its analysis via weighted balls-into-bins experiments in which the weight of a ball is logarithmic in its multiplicity.

Keywords Data structures · Dictionaries · Membership · Approximate membership · Multisets

This research was supported by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel and the United States National Science Foundation (NSF).

An extended abstract of this paper appeared in WADS 2021.

✉ Ioana O. Bercea
ioana@cs.umd.edu

Guy Even
guy@eng.tau.ac.il

[1] Tel Aviv University, Tel Aviv, Israel

1 Introduction

We consider the dynamic dictionary problem for multisets. The special case of dictionaries for sets (i.e., multiplicities are ignored) is a fundamental problem in data structures and has been well studied [2, 11, 27, 32]. In the case of multisets, elements can have arbitrary (adversarial) multiplicities and we are given an upper bound n on the total cardinality of the multiset (i.e., including multiplicities) at any point in time. The goal is to design a data structure that supports multiplicity queries (i.e., how many times does x appear in the multiset?) and allows insertions and deletions to the multiset (i.e., the dynamic setting).

A related problem is that of supporting *approximate* membership and multiplicity queries. Approximate set membership queries allow for one-sided errors in the form of false positives: given an error parameter $\varepsilon > 0$, the probability of returning a "yes" on an element not in the set is at most ε. Such data structures are known as *filters*. For multisets, the corresponding data structure is known as a *counting* filter (or a *spectral* filter). A counting filter returns a count that is at least the multiplicity of the element in the multiset and overcounts with probability bounded by ε. Counting filters have received significant attention over the years due to their applicability in practice [6, 9, 17]. One of the main applications of dictionaries for multisets is in designing dynamic filters and counting filters [2]. This application is based on Carter et al. [8] who showed that by hashing each element into a random fingerprint, one can reduce a counting filter to a dictionary for multisets by storing the fingerprints in the dictionary.

The lower bound on the space required for storing a dictionary follows from a simple counting argument (i.e., information theoretic lower bound). Namely, the space of a dictionary for multisets of cardinality n is at least $\log\left(\frac{u+n-1}{n}\right) = n\log(u/n) + \Theta(n)$ bits, where u is the size of the universe.[1,2,3] In the case of filters, the lower bound is at least $n\log(1/\varepsilon) + \Theta(n)$ bits [25]. A data structure is *succinct* if the total number of bits it requires is $(1 + o(1)) \cdot \mathcal{B}$, where $\mathcal{B}$ denotes the lower bound on the space and the $o(1)$ term converges to zero as n tends to infinity. A data structure is *space-efficient* if it is succinct up to an additive $O(n)$ term in space.

For the design of both dictionaries and filters, the performance measures of interest are the space the data structure takes and the time it takes to perform the operations. The first goal is to design data structures for dictionaries over multisets that are space-efficient with high probability.[4] Our dictionary and counting filter are space-efficient for all ranges of parameters and succinct if the lower bound on the space satisfies $\mathcal{B} = \omega(n)$. Indeed, this is the case in a dictionary if $u = \omega(n)$ and in a filter if $\varepsilon = o(1)$.

[1] To see why this is the case, consider a $u \times (n+1)$ grid and all the shortest paths that go from the leftmost bottom vertex to the rightmost top vertex. Such a path consists of $u + n - 1$ edges and can be completely described by its n horizontal edges, where each horizontal edge corresponds to one occurence of an element of the universe in the input set.

[2] All logarithms are base 2 unless otherwise stated. $\ln x$ is used to denote the natural logarithm.

[3] This equality holds when u is significantly larger than n.

[4] By with high probability (whp), we mean with probability at least $1 - 1/n^{\Omega(1)}$. The constant in the exponent can be controlled by the designer and only affects the $o(1)$ term in the space of the dictionary or the filter.

The second goal is to support queries, insertions, and deletions in constant time in the word RAM model. The constant time guarantees should be in the worst case with high probability (see [1, 2, 7, 23] for a discussion on the shortcomings of expected or amortized performance in practical scenarios). We assume that each memory access can read/write a word of $w = \log u$ contiguous bits.

The current best known dynamic dictionary for multisets was designed by Pagh, Pagh, and Rao [27] based on the dictionary for sets of Raman and Rao [32]. The dictionary is space-efficient and supports membership queries in constant time in the worst case. Insertions and deletions take amortized expected constant time and multiplicity queries take $O(\log n)$ in the worst case. In the case of sets, the state-of-the-art dynamic dictionary of Arbitman, Naor, and Segev [2] achieves the "best of both worlds": it is succinct and supports all operations in constant time whp. Arbitman et al. [2] pose the open problem of whether a similar result can be achieved for multisets.

1.1 The Challenge with Multisets

Recently, progress on the multiset problem was achieved by Bercea and Even [4] who designed a constant-time dynamic space-efficient dictionary for *random* multisets. In a random multiset, each element is sampled independently and uniformly at random from the universe (with repetitions). Multiplicities of elements in the dictionary in [4] are handled by storing duplicates. Namely, an element x with multiplicity $m(x)$ has $m(x)$ duplicate copies in the dictionary. The analysis employs ball-into-bins experiments in which the weight of a ball is linear in its multiplicity (more precisely, $\log(u/n) \cdot m(x)$). Their analysis breaks if the multiset is arbitrary (i.e., not random).[5]

To see this, consider an adversary that chooses the input, in particular, the vector of elements and their multiplicities. Suppose we want to design a dictionary for multisets that is space-efficient and supports operations in constant time with respect to such an adversary. The challenge in designing such a data structure is that the dictionary must "compress" the input (to achieve space-efficiency) while requiring only constant time per operation. If the data structure uses long counters (expecting high multiplicities), then the adversary can choose small multiplicities, causing the data structure to waste space. If the data structure uses short counters, then the adversary can choose high multiplicities, which the data structure may accommodate by storing several duplicates whose counters are added to maintain the overall multiplicity of the element. This, however, may require too many duplicates to be stored, causing load balancing problems (that impede constant time operations).

We resolve this dilemma by identifying a threshold of $\log^3 n$ between low and high multiplicities. Low multiplicities are encoded using variable length counters. The analysis deals with ball-into-bins experiments in which the weight of a ball is *logarithmic* in its multiplicity (more precisely, $\log(u/n) + O(\log(m(x)))$). Each variable-length counter requires on average a constant number of bits. High multiplicities are encoded by $\log n$-bit counters, but there are only $O(n/\log^3 n)$ high multiplicity elements.

[5] For example, storing n copies of the same element would lead to almost all the elements being stored in the second level spare, causing the spare to overflow.

1.2 Results

In the following theorem, *overflow* refers to the event that the space allocated in advance for the dictionary does not suffice. Such an event occurs if the random hash function fails to "balance loads".

Theorem 1 (dynamic multiset dictionary) *There exists a dynamic dictionary that maintains dynamic multisets of cardinality at most n from the universe* $\mathcal{U} = \{0,1\}^{\log_2 u}$ *with the following guarantees: (1) For every polynomial in n sequence of operations (multiplicity query, insertion, deletion), the dictionary does not overflow whp. (2) If the dictionary does not overflow, then every operation can be completed in constant time. (3) The required space is* $(1 + o(1)) \cdot n \log(u/n) + O(n)$ *bits.*

Our dictionary construction considers a natural separation into the *sparse* and *dense* case based on the size of the universe relative to n. The sparse case, defined when $\log(u/n) = \omega(\log\log n)$, enables us to store additional $\Theta(\log\log n)$ bits per element without sacrificing space-efficiency. However, the encoding of the elements is longer, so fewer encodings can be packed in a word. In this case, we propose a dictionary for multisets that is based on dynamic dictionaries that support both membership queries and satellite data (i.e., it stores (key, value) pairs where the key is the element and the value is its satellite data). We use two separate dictionaries: (1) One dictionary is used for the elements with multiplicity at most $\log^3 n$ (in which the satellite data is the multiplicity that is encoded using $O(\log\log n)$ bits). (2) The second dictionary is used for the elements with multiplicity at least $\log^3 n$ (in which the satellite data is the multiplicity that is encoded using $\log n$ bits). This construction is described in Sect. 3.

The dictionary for the *dense* case deals with the case in which $\log(u/n) = O(\log\log n)$.[6] Following [4], we hash distinct elements into a first level that consists of small space-efficient "bin dictionaries" of fixed capacity. The first level only stores elements of multiplicity strictly smaller than $\log^3 n$, just like in the sparse case. However, we employ variable-length counters to encode multiplicities and store them in a separate structure called a "counter dictionary". We allocate one counter dictionary for each bin dictionary. The space (i.e., number of bits) of the counter dictionary is linear in the capacity of the associated bin dictionary (i.e., the maximum number of elements that it can store). Namely, we spend a constant number of bits on average to encode the multiplicity of each element in the first level.

Elements that do not fit in the first level are stored in a secondary data structure called the *spare*. We prove that whp, the number of elements stored in the spare is $O(n/\log^3 n)$. Hence, even if a $\log n$-bit counter is attached to each element in the spare, then the spare still requires $o(n)$ bits. To bound the number of elements that are stored in the spare, we cast the process of hashing counters into counter dictionaries as a weighted balls-into-bins experiment in which balls have logarithmic weights (see Sect. 4.5).

As a corollary of Theorem 1, we obtain a counting filter with the following guarantees.[7]

[6] The dense case is especially relevant in practical approximate membership (filter) settings in which $u/n = 1/\varepsilon$ due to the reduction of Carter et al. [8].

[7] Note that we allow ε to be as small as n/u (below this threshold, we can simply use a dictionary instead).

Corollary 1 (dynamic counting filter) *There exists a dynamic counting filter for multisets of cardinality at most n from a universe $\mathcal{U} = \{0,1\}^u$ such that the following hold: (1) For every polynomial in n sequence of operations (multiplicity query, insertion, deletion), the filter does not overflow whp. (2) If the filter does not overflow, then every operation can be completed in constant time. (3) The required space is $(1 + o(1)) \cdot \log(1/\varepsilon) \cdot n + O(n)$ bits. (4) For every multiplicity query, the probability of overcounting is bounded by ε.*

We note that our filter is guaranteed to be succinct when $\varepsilon = o(1)$. The problem of designing succinct dynamic filters for constant ε remains an interesting open problem, even in the case of sets [4]. In this latter case, the lower bound of Lovett and Porat [25] requires that we use at least $C(\varepsilon) \cdot n \log(1/\varepsilon)$ bits, where the constant $C(\varepsilon) > 1$ depends only on ε.

1.3 Related Work

The dictionary for multisets of Pagh et al. [27] is space-efficient and supports membership queries in constant time in the worst case. Insertions and deletions take amortized expected constant time and multiplicity queries take $O(\log c)$ for a multiplicity of c. Multiplicities are represented "implicitly" by a binary counter whose operations (query, increment, decrement) are simulated as queries and updates to dictionaries on sets.[8] Increments and decrements to the counter take $O(1)$ bit probes (and hence $O(1)$ dictionary operations) but decoding the multiplicity takes $O(\log n)$ time in the worst case. We are not aware of any other dictionary constructions for multisets.[9]

Dynamic dictionaries for sets have been extensively studied [1, 2, 10–12, 14, 19, 30, 32]. The dynamic dictionary for sets of Arbitman et al. [2] is succinct and supports operations in constant time whp. In [2], they pose the problem of designing a dynamic dictionary for multisets as an open question.

In terms of counting filters, several constructions do not come with worst case guarantees for storing arbitrary multisets [6, 17]. The only previous counting filter with worst case guarantees we are aware of is the Spectral Bloom filter of Cohen and Matias [9] (with over 500 citations in Google Scholar). The construction is a generalization of the Bloom filter and hence requires $\Theta(\log(1/\varepsilon))$ memory accesses per operation. The space usage is similar to that of a Bloom filter and depends on the sum of logs of multiplicities. Consequently, when the multiset is a set, the required space is $1.44 \cdot \log(1/\varepsilon) \cdot n + \Theta(n)$.

1.4 Paper Organization

Preliminaries are in Sect. 2. The construction for the sparse case can be found in Sect. 3 and the one for the dense case is described and analyzed in Sect. 4. Section 5 describes

[8] To be more exact, for each bit of the counter, the construction in Pagh et al. [27] allocates a dictionary on sets such that the value of the bit can be retrieved by performing a lookup in the dictionary. Updating a bit of the counter is done by inserting or deleting elements in the associated dictionary.

[9] Data structures for predecessor and successor queries such as [31] can support multisets but they do not meet the required performance guarantees for multiplicity queries.

how our analysis in the dense case works without the assumption of access to truly random hash functions. Possible implementations of the dictionary from Sect. 4 are discussed in Sect. 6. Finally, Corollary 1 is proved in Sect. 7.

2 Preliminaries

For $k > 0$, let $[k]$ denote the set $\{0, \ldots, \lceil k \rceil - 1\}$. Let $\mathcal{U} \triangleq [u]$ denote the universe of all possible elements. We often abuse notation, and regard elements in $[u]$ as binary strings of length $\log u$. For a string $a \in \{0,1\}^*$, let $|a|$ denote the length of a in bits.

Definition 1 *(multiset)* A *multiset* $\mathcal{M}$ over $\mathcal{U}$ is a function $\mathcal{M} : \mathcal{U} \to \mathbb{N}$. We refer to $\mathcal{M}(x)$ as the *multiplicity* of x.The *cardinality* of a multiset $\mathcal{M}$ is denoted by $|\mathcal{M}|$ and defined by $|\mathcal{M}| \triangleq \sum_{x \in \mathcal{U}} \mathcal{M}(x)$.

The *support* of the multiset is denoted by $\sigma(\mathcal{M})$ and is defined by $\sigma(\mathcal{M}) \triangleq \{x \mid \mathcal{M}(x) > 0\}$.

Operations over Dynamic Multisets We consider the following operations: $\mathsf{insert}(x)$, $\mathsf{delete}(x)$, and $\mathsf{count}(x)$. Let $\mathcal{M}_t$ denote the multiset after t operations. A *dynamic multiset* $\{\mathcal{M}_t\}_t$ is specified by a sequence $\left\{\mathsf{op}_t\right\}_{t \geq 1}$ of as follows.[10]

$$\mathcal{M}_t(x) \triangleq \begin{cases} 0 & \text{if } t = 0 \\ \mathcal{M}_{t-1}(x) + 1 & \text{if } \mathsf{op}_t = \mathsf{insert}(x) \\ \mathcal{M}_{t-1}(x) - 1 & \text{if } \mathsf{op}_t = \mathsf{delete}(x) \\ \mathcal{M}_{t-1}(x) & \text{otherwise.} \end{cases}$$

We say that a dynamic multiset $\{\mathcal{M}_t\}_t$ has *cardinality* at most n if $|\mathcal{M}_t| \leq n$, for every t.

Dynamic Dictionary for Multisets A *dynamic dictionary for multisets* maintains a dynamic multiset $\{\mathcal{M}_t\}_t$. The response to $\mathsf{count}(x)$ is simply $\mathcal{M}_t(x)$.

Dynamic Counting Filter A *dynamic counting filter* maintains a dynamic multiset $\{\mathcal{M}_t\}_t$ and is parameterized by an error parameter $\varepsilon \in (0,1)$. Let out_t denote the response to a $\mathsf{count}(x_t)$ at time t. We require that the output out_t satisfy the following conditions:

$$\mathsf{out}_t \geq \mathcal{M}_t(x_t) \tag{1}$$

$$\Pr\left[\mathsf{out}_t > \mathcal{M}_t(x_t)\right] \leq \varepsilon\ . \tag{2}$$

Namely, out_t is an approximation of $\mathcal{M}_t(x_t)$ with a one-sided error.

Definition 2 *(overcounting)* Let Err_t denote the event that $\mathsf{op}_t = \mathsf{count}(x_t)$, and $\mathsf{out}_t > \mathcal{M}_t(x_t)$.

[10] We require that $\mathsf{op}_t = \mathsf{delete}(x_t)$ only if $\mathcal{M}_{t-1}(x_t) > 0$.

Note that overcounting generalizes false positive events in filters over sets. Indeed, a false positive event occurs in a filter for sets if $\mathcal{M}_t(x_t) = 0$ and $\mathsf{out}_t > 0$.[11]

2.1 The Model

2.1.1 Memory Access Model

We assume that the data structures are implemented in the word RAM model in which every access to the memory accesses a word. Let w denote the memory word length in bits. We assume that $w = \log u$. As is standard in this model, we assume that in constant time, the following operations can be performed on a word in constant time: read/write, addition, subtraction, multiplication, division, shifting, and bitwise operations (AND, OR, XOR). In one of the implementations we propose (the Elias-Fano encoding from Sect. 6), we also employ rank and select operations on words, which can also be performed in constant time [21].

2.1.2 Oblivious Adversary

Our data structures are designed to work against an oblivious adversary, that is, the input sequence of the adversary is independent of the random bits used for the construction of the data structure. Such independence occurs, for example, if the adversary fixes the input sequence before the data structure randomly chooses the hash functions.

2.1.3 Probability of Overflow

We prove that overflow occurs with probability at most $1/\mathsf{poly}(n)$ and that one can control the degree of the polynomial (the degree of the polynomial only affects the $o(1)$ term in the size bound). The probability of an overflow depends only on the random choices that the dictionary makes.

2.1.4 Hash Functions

Our dictionary employs similar succinct hash functions as in Arbitman et al. [2] which have a small representation and can be evaluated in constant time. For simplicity, we first analyze the data structure assuming fully random hash functions (Sect. 4.5). In Sect. 5 , we prove that the same arguments hold when we use succinct hash functions and that the techniques in [2] used for sets can also be employed for multisets. The counting filter reduction additionally employs pairwise independent hash functions.

3 Dictionary for Multisets via Key-Value Dictionaries (Sparse Case)

In this section, we show how to design a multiset dictionary based on a key-value dictionary on sets that supports attaching satellite data per element. Such a key-value

[11] The probability space is induced only by the random choices (i.e., choice of hash functions) that the filter makes. Note also that if $\mathsf{op}_t = \mathsf{op}_{t'} = \mathsf{count}(x)$, then the events Err_t and $\mathrm{Err}_{t'}$ need not be independent.

dictionary with satellite data supports the operations: query, insert, delete, retrieve, and update. A retrieve operation for x returns the satellite data of x. An update operation for x with new satellite data d stores d as the new satellite data of x. Loosely speaking, we use the satellite data to store a counter with $\Theta(\log\log n)$ bits. Hence, a succinct multiset dictionary is obtained from a succinct key-value dictionary for sets only if $\log(u/n) = \omega(\log\log n)$.

Let $\mathsf{Dict}(\mathcal{U}, n, r)$ denote a dynamic key-value dictionary for sets of cardinality at most n over a universe $\mathcal{U}$, where r bits of satellite data are attached to each element. One can design $\mathsf{Dict}(\mathcal{U}, n, r)$ from $\mathsf{Dict}(\mathcal{U}', n, 0)$, where $\mathcal{U}' \triangleq \mathcal{U} \times [2^r]$ if the first component of an element is a key. Namely, we require that the dataset $\mathcal{D}'(t) \subset \mathcal{U} \times [2^r]$ does not contain two elements (x_1, d_1) and (x_2, d_2) such that $x_1 = x_2$. An implementation of $\mathsf{Dict}(\mathcal{U}, n, r)$ (for $r = O(\log n)$) with constant time per operation can be obtained from the dictionary of Arbitman et al. [2] (see also [3]). The space of such an implementation is $(1 + o(1)) \cdot (\log(u/n) + r) \cdot n + O(n)$.

Let $\mathsf{MS\text{-}Dict}(\mathcal{U}, n)$ denote a dynamic dictionary for multisets over $\mathcal{U}$ of cardinality at most n. We propose a reduction that employs two key-value dictionaries. The space for these dictionaries is allocated up front before the first element is inserted. (Hence, overflow of $\mathsf{MS\text{-}Dict}(n)$ occurs if one of these dictionaries overflows.)

Observation 1 *One can implement* $\mathsf{MS\text{-}Dict}(\mathcal{U}, n)$ *using two dynamic key-value dictionaries:* $D_1 = \mathsf{Dict}(\mathcal{U}, n, 3\log\log n)$ *and* $D_2 = \mathsf{Dict}(\mathcal{U}, n/(\log^3 n), \log n)$. *Each operation over* $\mathsf{MS\text{-}Dict}$ *can be performed using a constant number of operations over* D_1 *and* D_2.

Proof (sketch) An element is *light* if its multiplicity is at most $\log^3 n$, otherwise it is *heavy*. Dictionary D_1 is used for storing the light elements, whereas dictionary D_2 is used for storing the heavy elements. The satellite data in both dictionaries is a binary counter of the multiplicity. Counters in D_1 are $3\log\log n$ bits long, whereas counters in D_2 are $\log n$ bits long. □

Lemma 1 *If* $\log(u/n) = \omega(\log\log n)$, *then there exists a dynamic multiset dictionary that is succinct and supports operations in constant time in the worst case whp.*

Proof The implementation suggested in Obs. 1 employs two dictionaries D_1 and D_2 (each with satellite data). The space of D_1 is $(1+o(1)) \cdot ((\log(u/n) + 3\log\log n) \cdot n + O(n)$. The space of D_2 is $(1+o(1)) \cdot ((\log((u\log^3 n)/n) + \log n) \cdot \frac{n}{\log^3 n} + O(\frac{n}{\log^3 n}) = o(\log(u/n) \cdot n)$. Hence, the space of the multiset dictionary $\mathsf{MS\text{-}Dict}(n)$ is: $(1+o(1)) \cdot ((\log(u/n) + 3\log\log n) \cdot n + O(n)$. In the sparse case $\log(u/n) = \omega(\log\log n)$. The lower bound on the space per element is $\log(u/n)$ bits, and hence the obtained $\mathsf{MS\text{-}Dict}(n)$ is succinct. □

This completes the proof of Theorem 1 for the sparse case.

Remark An alternative solution stores the multiplicities in an array separately from a dictionary that stores the support of the multiset. Let s denote the cardinality of the support of the multiset. Let $h : \mathcal{U} \to [s + o(s)]$ be a dynamic perfect hashing that requires $\Theta(s\log\log s)$ bits and supports operations in constant time (such as the one in [11]). Store the (variable-length) binary counter for x at index $h(x)$ in the array.

The array can be implemented in space that is linear in the total length of the counters and supports query and update operations in constant time [5].

4 Dictionary for Multisets (Dense Case)

In this section, we prove Theorem 1 for the case in which $\log(u/n) = O(\log\log n)$, which we call the *dense* case. We refer to this dictionary construction as the *MS-Dictionary* (Multiset Dictionary) in the dense case.

The MS-Dictionary construction follows the same general structure as in [2, 4, 11]. Specifically, it consists of two levels of dictionaries. The first level is designed to store a $(1 - o(1))$ fraction of the elements (Sect. 4.3). An element is stored in the first level provided that its multiplicity is at most $\log^3 n$ and there is enough capacity. Otherwise, the element is stored in the second level, which is called the *spare* (Sect. 4.4).

The first level of the MS-Dictionary consists of m *bin dictionaries* $\{\mathsf{BD}_i\}_{i\in[m]}$ together with m *counter dictionaries* $\{\mathsf{CD}_i\}_{i\in[m]}$. Each bin dictionary stores at most $n_B = (1+\delta)B$ distinct elements, where $\delta = o(1)$ and $B \triangleq n/m$ denotes the mean occupancy of a bin dictionary (see Sect. 4.1 for further parametrizations). We say that a bin dictionary is *full* if it stores n_B distinct elements. Elements are assigned to bin dictionaries via a hash function (see Sect. 4.2). If, upon insertion of a new element x, the bin dictionary is full, then the insertion is forwarded to the spare.

Each counter dictionary stores variable-length binary counters that encode the multiplicities of elements in the associated bin dictionary. The counter dictionaries require that the maximum multiplicity is polylogarithmic. In addition, the sum of the logarithms of the multiplicities is $O(B)$, stated formally as follows.

Definition 3 *(full counter dictionary)* Consider a bin dictionary BD_i and its corresponding counter dictionary CD_i. We say that the counter dictionary CD_i is *full* if $\sum_{x\in\mathsf{BD}_i} \lceil\log(\mathcal{M}(x)+1)\rceil > 12B$.

If upon an insertion of a new counter or an increment of an already existing counter, the counter dictionary is full or becomes full, then the element and its multiplicity are moved to the spare.

Overall, we maintain the following invariant on which elements are stored in the spare.

Invariant 2 *An element x such that $\mathcal{M}_t(x) > 0$ is stored in the spare at time t if: (1) $\mathcal{M}_t(x) \geq \log^3 n$, or (2) the bin dictionary corresponding to x is full, or (3) the counter dictionary corresponding to x is full.*

We emphasize that an element x cannot further stay in the spare if it does not satisfy Invariant 2. Namely, if the justification for storing x in the spare does not hold anymore, then x has to be transferred to the first level of the dictionary. This transfer may be performed in a "lazy" fashion. Namely, instead of searching for elements in the spare that should be transferred to the first level, the transfer takes place when we "stumble" upon them while trying to insert an element. Please refer to Sect. 4.4 for further details on implementation.

We denote the upper bound on the cardinality of the support of the multiset stored in the spare by n_S. We say that the spare *overflows* when more than n_S elements are

Table 1 Setting of parameters in the MS-Dictionary in the dense case (i.e., $\log(u/n) = O(\log \log n)$)

Parameter	Value	Meaning
u		Cardinality of the universe $\mathcal{U}$
n		Maximum cardinality of the multiset $\mathcal{M}(t)$
B	$\triangleq \frac{\log n}{\log(u/n)}$	Average number of elements per bin
m	$\triangleq \frac{n}{B}$	Number of bins
δ	$\triangleq \Theta(\frac{\log \log n}{\sqrt{B}})$	Over-provisioning fraction per bin
n_B	$\triangleq (1+\delta) \cdot B$	Maximum number of distinct elements stored in a bin
n_S	$\triangleq \frac{3n}{\log^3 n}$	Maximum number of distinct elements stored in the spare

stored in it. In Sect. 4.5, we show that this does not happen whp over a polynomial sequence of insertions.

4.1 Parametrization

The choice of parameters in the design of the MS-Dictionary for the dense case is summarized in Table 1.

4.2 Hash Functions

We employ a permutation $\pi : \mathcal{U} \to \mathcal{U}$. We define $h^b : \mathcal{U} \to [m]$ to be the leftmost $\log m$ bits of the binary representation of $\pi(x)$ and by $h^r : \mathcal{U} \to [u/m]$ to be the remaining $\log(u/m)$ bits of $\pi(x)$. An element x is hashed to the bin dictionary of index $h^b(x)$. Hence storing x in the first level of the dictionary amounts to storing $h^r(x)$ in BD_i, where $i = h^b(x)$, and storing $\mathcal{M}_t(x)$ in CD_i. (This reduction in the universe size is often called "quotienting" [11, 24, 27, 28]).

The overflow analysis in Sect. 4.5 assumes that π is a perfect random permutation (i.e., chosen uniformly at random from the set of all permutations on $\mathcal{U}$). In Sec 5, we discuss how one can replace this assumption with the succinct hash functions of Arbitman et al. [2].

4.3 The First Level of the Multiset Dictionary

The first level of the MS-Dictionary consists of bin dictionaries and counter dictionaries. We review their functionality here and refer the reader to Sect. 6 for details on implementation.

4.3.1 Bin Dictionaries

Each bin dictionary(BD) is a deterministic dictionary for sets of cardinality at most n_B that supports queries, insertions, and deletions. Each bin dictionary can be imple-

mented using global lookup tables [2] or Elias-Fano encoding [4]. Implementation via global lookup tables is succinct, whereas the Elias-Fano encoding requires $2+\log(u/n)$ bits per element, and is succinct only if $\log(u/n) = \omega(1)$. Moreover, each BD fits in a constant number of words and performs queries, insertions and, deletions in constant time in the worst case.

4.3.2 Counter Dictionaries

Let $(x_1, \ldots x_\ell)$ denote the sequence of (distinct) elements stored in BD_i. Let $\mathcal{M}(x_i)$ denote the multiplicity of x_i. The counter dictionary CD_i stores the sequence of multiplicities $(\mathcal{M}(x_1), \ldots, \mathcal{M}(x_\ell))$. Namely, the order of the element multiplicities stored in CD_i is the same order in which the corresponding elements are stored in BD_i. Multiplicities in CD_i are encoded using variable-length counters. We employ a trivial 2-bit alphabet to encode 0, 1 and "end-of-counter" symbols for encoding the multiplicities. Consider a counter that stores the value c. We refer to $\lceil \log_2(c+1) \rceil$ as the *length* of the counter. However, the *encoding* of the counter is $2(1 + \lceil \log_2 c \rceil)$ bits long. The contents of CD_i is simply a concatenation of the encoding of the counters. We allocate $2(12B + n_B) = O(B)$ bits per CD.[12] Alternatively, one can also employ global lookup tables in which every operation to the counter dictionary specifies the index of the multiplicity it is applied to. Please refer to Sect. 6 for further details.

4.3.3 Operations to the Counter Dictionaries

The CD supports the operations of multiplicity query, increment, and decrement. These operations are carried out naturally in constant time because each CD_i fits in $O(1)$ words. We note that an increment to x may cause the CD to be full, in which case x is deleted from the bin dictionary and is inserted into the spare together with its updated counter. Similarly, a decrement may zero the counter, in which case x is deleted from the bin dictionary (and hence its multiplicity is also deleted from the counter dictionary).

4.4 The Spare

Since the multiplicity of every element in the spare is at most n, the multiplicity can be represented by a $\log n$-bit counter. We can thus implement the spare using a dynamic dictionary $\mathsf{Dict}(\mathcal{U}, n_S, \log n)$. An additional requirement from that spare is that it supports moving elements back to the first level if their insertion no longer violates Invariant 2.

For this purpose, we propose to employ the dictionary of Arbitman et al. [1] that is a de-amortized construction of the cuckoo hash table of Pagh and Rodler [29]. Namely, each element is assigned two locations in an array. If upon insertion, both locations are occupied, then space for the new element is made by "relocating" an element occupying one of the two locations. Long chains of relocations are "postponed" by employing a

[12] Note, however, that we define a CD to be full if the sum of counter lengths is $12B$ (even if we did not use all its space). The justification for this choice of constants is to simplify the analysis.

queue of pending insertions. Thus, each operation is guaranteed to perform in constant time in the worst case. The space that the dictionary occupies is $O(n_S(\log(u/n) + \log n)) + O(n_S) = o(n)$.

The dynamic dictionary in [1] is used as a spare in the incremental filter in [2]. We use it in a similar manner to maintain Invariant 2 in a "lazy" fashion. Namely, if an element x residing in the spare is no longer in violation of Invariant 2 (for instance, due to a deletion in the bin dictionary), we do not immediately move x from the spare back to its bin dictionary. Instead, we "delay" such an operation until x is examined during a chain of relocations. Specifically, during an insertion to the spare, for each relocated element, one checks if this element is still in violation of Invariant 2. If it is not, then it is deleted from the spare and inserted into the first level. This increases the time of operations only by a constant and does not affect the overflow probability of the spare.

4.5 Overflow Analysis

The event of an overflow occurs if more than n_S distinct elements are stored in the spare. In this section, we prove that overflow does not occur whp when we employ perfectly random hash functions.

Invariant 2 reduces the dynamic setting to the incremental setting in the sense that the number of elements in the spare at time t depends only on $\mathcal{D}(t)$ (and not on the complete history).[13] The overflow analysis proceeds by proving that, for every t, the spare does not overflow whp. By applying a union bound, we conclude that overflow does not occur whp over a polynomial number of operations in the dynamic setting.

Recall that each component of the first level of the dictionary has capacity parameters: each bin dictionary has an upper bound of $n_B = (1+\delta)B$ on the number of distinct elements it stores and each counter dictionary has an upper bound of $12B$ on the total length of the counters it stores. Additionally, the first level only stores elements whose multiplicity is strictly smaller than $\log^3 n$. According to Invariant 2, if the insertion of some element x exceeds these bounds, then x is moved to the spare.

We bound the number of elements that go to the spare due to failing one of the conditions of Invariant 2 separately. The number of elements whose multiplicity is at least $\log^3 n$ is at most $n/\log^3 n$. The number of distinct elements that are stored in the spare because their bin dictionary is full is at most $n/\log^3 n$ whp. The proof of this bound can be derived by modifying the proof of Lemma 2 (see also [2]). We focus on the number of distinct elements whose counter dictionary is full.

Lemma 2 *The number of distinct elements whose corresponding CD is full is at most $n/\log^3 n$ whp.*

Proof Recall that there are $m = n/B$ counter dictionaries and that each CD stores the multiplicities of at most $n_B = (1+\delta)B$ distinct elements of multiplicity strictly smaller than $\log^3 n$. In a full CD, the sum of the counter lengths reaches $12B$. We start by bounding the probability that the total length of the counters in a CD is at least $12B$.

[13] Note that the fact that we maintain Invariant 2 in a "lazy" fashion does not affect this analysis. If an insertion to the spare fails due to a non-spare element residing in it, we move the non-spare element to the first level. Thus, the temporary presence of non-spare elements does not affect the performance of the spare.

Formally, consider a multiset $\mathcal{M}$ of cardinality n consisting of s distinct elements $\{x_i\}_{i\in[s]}$ with multiplicities $\{f_i\}_{i\in[s]}$ (note that $\sum_{i\in[s]} f_i = n$). The length of the counter for multiplicity f_i is $w_i \triangleq \lceil \log(f_i+1)\rceil$ (we refer to this quantity as *weight*). For $\beta \in [m]$, let $\mathcal{M}^\beta$ denote the sub-multiset of $\mathcal{M}$ consisting of the elements x_i such that $h^b(x_i) = \beta$. Let C_β denote the event that the weight of $\mathcal{M}^\beta$ is at least $12B$, namely $\sum_{x_i\in\mathcal{M}^\beta} w_i \geq 12B$. We begin by bounding the probability of event C_β occurring.

For $i \in [s]$, define the random variable $X_i \in \{0, w_i\}$, where $X_i = w_i$ if $h^b(x_i) = \beta$ and 0 otherwise. Since the values $\left\{(h^b(x_i), h^r(x_i))\right\}_i$ are sampled at random without replacement (i.e., obtained from a random permutation), the random variables $\{X_i\}_i$ are negatively associated. Let $\mu \triangleq \frac{1}{m}\cdot\sum_{i\in[s]} w_i$ denote the expected weight per CD. Since $w_i \leq \log(2(1+f_i))$, by the concavity of $\log(x)$, we have

$$\mu \leq \frac{s}{m}\log\frac{\sum_{i\in[s]} 2(1+f_i)}{s} \leq \frac{s}{m}\log\left(2+\frac{2n}{s}\right) \leq 2B\ .$$

Since $w_i \leq \log\log^3 n$ (we omit the ceiling to improve readability), by Chernoff's bound (Eq.(8) in [20]):

$$\Pr C_\beta = \Pr\sum_{i\in[s]} X_i \geq 6\cdot 2B \leq 2^{-\frac{12B}{\log\log^3 n}} = 1/(\log n)^{\omega(1)}\ .$$

Let $I(C_\beta)$ denote the indicator variable for event C_β. Then $\mathbb{E}\left[\sum_\beta I(C_\beta)\right] \leq n/(\log n)^{\omega(1)}$. Moreover, the RVs $\left\{I(C_\beta)\right\}_\beta$ are negatively associated (more weight in bin b implies less weight in bin b'). By Chernoff's bound [15, 20]:

$$\Pr\sum_b I(C_\beta) \geq \frac{n}{\log^5 n} \leq O(2^{-n/(\log^5 n)})\ .$$

Whp, a bin is assigned at most $\log^2 n$ elements (recall, the average occupancy of a bin dictionary is $B < \log n$). We conclude that the number of elements that are stored in the spare due to events $\bigcup_b C_\beta$ is at most $n/(\log^3 n)$ whp. □

4.6 Proof of Theorem 1 for the Dense Case

In Lemma 3 (Sect. 6) we show that each bin dictionary can be implemented using $n_B\log(u/n) + \Theta(n_B)$ bits and each CD occupies $\Theta(B)$ bits. Furthermore, every operation can be executed in constant time in the worst case, unless overflow happens. Therefore, the first level of the MS-Dictionary takes $(1+\delta)n\log(u/n) + \Theta(n)$ bits. The spare takes $O(n_S\log(u/n)) = o(n)$ bits, since $n_S = O(n/\log^3 n)$. Therefore, the space the whole dictionary takes is $(1+o(1))\cdot\log(u/n) + \Theta(n)$ bits.

In Lemma 2, we show that overflow happens with low probability over a sequence of polynomial number of insertions. This completes the proof of Theorem 1 for the dense case.

5 Succinct Hash Functions

In this section, we discuss how to replace the assumption of truly random permutations with succinct hash functions (i.e., representation requires $o(n)$ bits) that have constant evaluation time in the RAM model.

We follow the construction in [2], which we describe as follows. Partition the universe into $M = n^{9/10}$ parts using a one-round Feistel permutation (described below) such that the number of elements in each part is at most $n^{1/10} + n^{3/40}$ whp. The permutation uses highly independent hash functions [13, 34]. Apply the dictionary construction separately in each part with an upper bound of $n^{9/10} + n^{3/40}$ on the cardinality of the set. Within each part, the dictionary employs a k-wise δ-dependent permutation. A collection Π of permutations $\pi : \mathcal{U} \to \mathcal{U}$ is k-wise δ-dependent if for any distinct elements $x_1, \ldots, x_k \in \mathcal{U}$, the distribution on $(\pi(x_1), \ldots, \pi(x_k))$ induced by sampling $\pi \in \Pi$ is δ-close in statistical distance to the distribution induced by a truly random permutation. Arbitman et al. [2] show how one can obtain succinct k-wise δ-dependent permutations that can be evaluated in constant time by combining the constructions in [22, 26]. Setting $k = n^{1/10} + n^{3/40}$ and $\delta = 1/n^{\Theta(1)}$ ensures that the bound on the size of the spare holds whp in each part and hence, by union bound, in all parts simultaneously.

To complete the proof, we need to prove that the partitioning is "balanced" whp also with respect to multisets. (Recall, that the cardinality of a multiset equals the sum of multiplicities of the elements in the support of the multiset.) Formally, we prove that the pseudo-random partition induces in each part a multiset of cardinality at most $n^{1/10} + n^{3/40} \log^{3/2} n$ whp. As "heavy" elements of multiplicity at least $\log^3 n$ are stored in the spare, we may assume that multiplicities are less that $\log^3 n$.

We first describe how the partitioning is achieved in [2]. The binary representation of x is partitioned into the leftmost $\log M$ bits, denoted by x_L and the remaining bits, denoted by x_R. A k'-wise independent hash function $f : \{0,1\}^{\log(u/M)} \to \{0,1\}^{\log M}$ is then sampled, with $k' = \lfloor n^{1/20}/(e^{1/3}) \rfloor$. The permutation π is defined as $\pi(x) = (x_L \oplus f(x_R), x_R)$.

Note that this induces a view of the universe as a two-dimensional table with u/M rows (corresponding to each x_R value) and M columns (corresponding to each $x_L \oplus f(x_R)$ value). Indeed, each cell of the table has at most one element (i.e., if $x = (x_L, x_R)$ and $y = (y_L, y_R)$ satisfy $x_L \oplus f(x_R) = y_L \oplus f(y_R)$ and $x_R = y_R$, then $x = y$). We define a *part* of the input multiset as consisting of all the elements of the input multiset that belong to the same column. The index of the part that x is assigned to is $x_L \oplus f(x_R)$. The corresponding part stores x_R.

The following observation follows from [2, Claim 5.4] and the fact that the maximum multiplicity of each element is strictly less than $\log^3 n$.

Observation 2 *The cardinality of every part of the multiset is at most* $n^{1/10} + n^{3/40}\log^{3/2} n$ *whp.*

Proof Fix a part $j \in [M]$ and for each $i \in [u/M]$, let $\mathcal{M}_i$ denote the multiset of all elements x with the x_R value equal to i (i.e., the multisets $\mathcal{M}_i$ consist of all the elements in row i). Each multiset $\mathcal{M}_i$ contributes at most one distinct element to the multiset of part j. Define $X_i \in [\log^3 n]$ to be the random variable that denotes the multiplicity of the element from $\mathcal{M}_i$ that is mapped to part j. Then $\mathbb{E}[X_i] = \frac{1}{M}\sum_{x\in\mathcal{U}} \mathcal{M}_i(x)$. Now define $X = \sum_{i\in[u/m]} X_i$ to be the random variable that denotes the cardinality of the multiset that is mapped into part j. By linearity of expectation, $\mathbb{E}[X] = n/M = n^{1/10}$. The random variables $\{X_i\}_i$ are k'-wise independent, since each variable X_i is determined by a different row in the table (and hence, each $\{X_i\}_i$ depends on a different x_R value). We scale the RVs $\{X_i\}_i$ by $\log^3 n$ and then apply 's bound for k'-wise independent RVs [33] and obtain:

$$\Pr\frac{X}{\log^3 n} \geq \left(1 + \frac{\log^{3/2} n}{n^{1/40}}\right)\cdot\frac{n^{9/10}}{\log^3 n} \leq \exp(-\lfloor k'/2\rfloor) = \exp(-\Omega(n^{1/20}))\ .$$

The claim follows. □

6 Implementation

In this section, we discuss how to implement the bin dictionaries and the counter dictionaries from the first level of the multiset dictionary for the dense case (Sect. 4.3). Recall that each bin dictionary is a space-efficient dictionary for $n_B = (1 + o(1))\log n/\log(u/n)$ elements that executes all operations in constant time in the worst case. Similar bin dictionaries have been employed in the dictionaries for sets of Arbitman et al. [2] and Bercea and Even [4]. We review their implementations here and show that the following holds:

Lemma 3 *The bin dictionaries and the counter dictionaries can be implemented such that:*

1. *Each bin dictionary requires* $n_B \log(u/n) + \Theta(n_B)$ *bits,*
2. *Each counter dictionary requires* $\Theta(B)$ *bits,*
3. *Every operation can be performed in constant time in the worst case.*

For each bin dictionary, the associated counter dictionary stores the variable-length strings representing the multiplicity of the (distinct) elements stored in the bin dictionary. Given an element stored in the bin dictionary, the counter dictionary must be able to retrieve, delete and decrement/increment its multiplicity in constant time in the worst case. To facilitate this, we take advantage of the lexicographic order in which the elements are stored in the bin dictionary. Specifically, we index the variable-length strings in the counter dictionary by the lexicographic order of their associated element in the bin dictionary such that the kth variable-length string represents the multiplicity of the kth element in the corresponding bin dictionary, in lexicographic order. We then store the strings in concatenated form in this order.

Operations to the counter dictionary are then parametrized by the index of the variable length string they refer to. This can be easily achieved since each operation to the counter dictionary is preceded by a query to the bin dictionary in order to verify membership. We thus modify the query operation to the bin dictionary to return the index of the element in the lexicographic order of the elements currently stored in the bin dictionary (if the element is found).

In the following, we present two implementations for bin dictionaries and counter dictionaries that follow the above design idea. Specifically, we show how the global lookup tables of Arbitman et al. [2] and the Elias-Fano encoding [16] employed in [4] can be extended to also work with variable-length strings and hence can be used to store the (variable-length) counters in the counter dictionaries.

6.1 Global Lookup Tables

Arbitman et al. [2] suggest implementing the bin dictionaries using global look up tables. In this implementation, all bin dictionaries employ a common global lookup table per operation. Hence, it is sufficient to show that the size of the tables is $o(n)$. Recall that each bin dictionary stores at most $n_B = (1+\delta)B$ distinct elements from a universe $\mathcal{U}'$ of size $u' = u/(n/B)$. Therefore, the total number of states of a bin dictionary is $s \triangleq \binom{u'}{n_B}$. Insertions and deletions are implemented as functions from $s \times u'$ to s. Namely, given the current state s of the dictionary and an element $x \in \mathcal{U}'$, each function returns an updated state that reflects the corresponding operation to the set. A $\mathsf{query}(x)$ returns an index in $[n_B]$ that is the rank of x according to the lexicographic order of the elements in the set corresponding to state s. The global lookup tables explicitly represent these functions and can be built in advance. Operations are therefore supported in constant time since all inputs and outputs fit in a constant number of words.

Moreover, each table requires at most $s \cdot u' \cdot \log s$ bits. Recall that $\log(u/n) = \omega(1)$ and that we are in the dense case, i.e., $\log(u/n) = O(\log\log n)$, hence $u = O(n \cdot \mathsf{polylog}(n))$. By fine-tuning B and δ, one can show that, $s \leq \sqrt{n}$ and the total number of bits each table takes is $o(n)$.

A similar approach can be employed for implementing the counter dictionaries. Specifically, a state s' of a counter dictionary corresponds to an ordered set of at most n_B variable-length strings such that the total length of the strings does not exceed $12B$ bits. There are thus at most $s' \triangleq 2^{12B} \cdot \binom{12B}{n_B} = 2^{\Theta(B)}$ possible states. The functions corresponding to each operation take as input a state and an index in $[n_B]$ denoting the rank of the variable-length string that the operation is applied to. The output of a query is a variable length string of length at most $12B$ and the output of delete, decrement and increment is another state in $[s']$. Therefore, each table requires at most $s' \cdot n_B \cdot \log s'$ bits. By setting $B = \Theta(\log n/\log(u/n))$, we ensure that $s' = 2^{\Theta(B)} \leq \sqrt{n}$ and have that each table requires $o(n)$ bits.

6.2 Elias-Fano Encoding

In this section, we briefly discuss a variation of the Elias-Fano [16, 18] encoding that appears as the "Exact Membership Tester 2" in Carter et al. [8]. Its usage as a dictionary that fits in a word appears in [3, 4]. A bin dictionary implemented using this encoding is referred to as a "pocket dictionary". The idea is to represent each element in the universe $[u']$ as a pair (q, r), where $q \in [B]$ (the quotient) and $r \in [u'/B]$ (the remainder). A header encodes in unary the number of elements that have the same quotient. The body is the concatenation of remainders. The space required is $B + n_B(1 + \log(u/n))$ bits, which meets the required space bound since $B = O(n_B)$. Similarly, a counter dictionary can be implemented by storing the counters consecutively using an "end-of-string" symbol, i.e., using a 2-bit alphabet to represent 0, 1 and "end-of-string". We use a ternary alphabet for this encoding, which requires at most $\Theta(B)$ bits to encode each CD. The counter dictionary contains at most $12B + n_B$ symbols, and hence fits in $O(B)$ bits.

Both the BDs and the CDs fit in $O(1)$ words. Operations to the BD and the CD require rank and select instructions. In particular for counter dictionaries, operations of locating a counter, incrementing, and decrementing a counter can be executed in constant time. The justification is that: (i) locating a counter amounts to a select operation, and (ii) increment or decrement can be implemented by add/subtract and a shift. See [4] for a discussion of how these operations can be executed in constant time.

7 The Counting Filter

To obtain a counting filter from our dictionary, we employ the reduction of Carter et al. [8]. The reduction is standard, we briefly review it here. Specifically, we use a pairwise independent hash function $h : \mathcal{U} \to [n/\varepsilon]$ to map an element $x \in \mathcal{U}$ to a fingerprint $h(x)$. Every operation on x to the counting filter is then implemented as an operation on $h(x)$ to the dictionary.

Formally, let $\mathcal{M}_h$ denote the multiset over $[n/\varepsilon]$ induced by a multiset $\mathcal{M}$ over $\mathcal{U}$ defined by $\mathcal{M}_h(y) \triangleq \sum_{x:h(x)=y} \mathcal{M}(x)$. We get the following:

Lemma 4 *A multiset dictionary for $\mathcal{M}_h$ constitutes a counting filter in which the probability of an overcount is at most ε.*

Proof Fix an element $x \in \mathcal{U}$. An overcount for x in the counting filter happens if and only if there is a different element $y \neq x$ with $M(y) > 0$ that hashes to the same fingerprint, i.e., $h(x) = h(y)$. Because h was chosen to be pairwise independent, we have that, for a fixed y, $\Pr h(x) = h(y) \leq \varepsilon/n$. By applying a union bound over all possible elements y, we get that the probability of overcounting the multiplicity of an element x is at most ε. □

7.1 Proof of Corollary 1

Operations to the counting filter are implemented as operations to the dictionary, so they take constant time in the worst case unless overflow happens. In terms of space, the dictionary stores a multiset of cardinality at most n from a universe of size n/ε and so it requires $(1 + o(1)) \log(1/\varepsilon)n + O(n)$ bits. Together with Lemma 4, this completes the proof of Corollary 1.

References

1. Arbitman, Y., Naor, M., Segev, G.: De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In: International colloquium on automata, languages and programming pp. 107–118. Springer (2009)
2. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: 2010 IEEE 51st Annual symposium on foundations of computer science, pp. 787–796. IEEE (2010)
3. Bercea, I.O., Even, G.: Fully-dynamic space-efficient dictionaries and filters with constant number of memory accesses. arxiv:1911.05060 (2019)
4. Bercea, I.O., Even, G.: 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, pp. 11:1–11:17 (2020). https://doi.org/10.4230/LIPIcs.SWAT.2020.11
5. Blandford, D.K., Blelloch, G.E.: Compact dictionaries for variable-length keys and data with applications. ACM Trans. Algorith. **4**(2), 1–25 (2008). https://doi.org/10.1145/1361192.1361194
6. Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An improved construction for counting Bloom filters. In: European symposium on algorithms, pp. 684–695. Springer (2006)
7. Broder, A., Mitzenmacher, M.: Using multiple hash functions to improve ip lookups. In: Proceedings IEEE INFOCOM 2001. Conference on computer communications. Twentieth annual joint conference of the IEEE computer and communications society (Cat. No. 01CH37213), vol. 3, pp. 1454–1463. IEEE (2001)
8. Carter, L., Floyd, R., Gill, J., Markowsky, G., Wegman, M.: Exact and approximate membership testers. In: Proceedings of the tenth annual ACM symposium on theory of computing, pp. 59–65. ACM (1978)
9. Cohen, S., Matias, Y.: Spectral Bloom filters. In: Proceedings of the 2003 ACM SIGMOD International conference on Management of data, pp. 241–252 (2003)
10. Dalal, K., Devroye, L., Malalla, E., McLeish, E.: Two-way chaining with reassignment. SIAM J. Comp. **35**(2), 327–340 (2005)
11. Demaine, E.D., auf der Heide, F.M., Pagh, R., Pătraşcu, M.: De dictionariis dynamicis pauco spatio utentibus. In: Latin American symposium on theoretical informatics, pp. 349–361. Springer (2006)
12. Dietzfelbinger, M., auf der Heide, F.M.: A new universal class of hash functions and dynamic hashing in real time. In: International colloquium on automata, languages and programming, pp. 6–19. Springer (1990)
13. Dietzfelbinger, M., Rink, M.: Applications of a splitting trick. In: International colloquium on automata, languages and programming, pp. 354–365. Springer (2009)
14. Dietzfelbinger, M., Weidling, C.: Balanced allocation and dictionaries with tightly packed constant size bins. Theor. Comp. Sci. **380**(1–2), 47–68 (2007)
15. Dubhashi, D., Ranjan, D.: Balls and bins: a study in negative dependence. Rand. Struct. & Algorith. **13**(2), 99–124 (1998)
16. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM (JACM) **21**(2), 246–260 (1974)
17. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw. **8**(3), 281–293 (2000)
18. Fano, R.M.: On the number of bits required to implement an associative memory. memorandum 61. Computer structures group, Project MAC, MIT, Cambridge, Mass. (1971)
19. Fotakis, D., Pagh, R., Sanders, P., Spirakis, P.: Space efficient hash tables with worst case constant access time. Theory Comp. Sys. **38**(2), 229–248 (2005)

20. Hagerup, T., Rüb, C.: A guided tour of chernoff bounds. Inf. Process. Lett. **33**(6), 305–308 (1990)
21. Hagerup, T., Tholey, T.: Efficient minimal perfect hashing in nearly minimal space. In: Annual symposium on theoretical aspects of computer science, pp. 317–326. Springer (2001)
22. Kaplan, E., Naor, M., Reingold, O.: Derandomized constructions of k-wise (almost) independent permutations. Algorithmica **55**(1), 113–133 (2009)
23. Kirsch, A., Mitzenmacher, M.: Using a queue to de-amortize cuckoo hashing in hardware. In: Proceedings of the forty-fifth annual allerton conference on communication, control and computing, vol. 75 (2007)
24. Knuth, D.E.: The art of computer programming, vol. 3: Searching and sorting. Reading MA: Addison-Wisley (1973)
25. Lovett, S., Porat, E.: A lower bound for dynamic approximate membership data structures. In: 2010 IEEE 51st Annual symposium on foundations of computer science, pp. 797–804. IEEE (2010)
26. Naor, M., Reingold, O.: On the construction of pseudorandom permutations: Luby-Rackoff revisited. J. Cryptol. **12**(1), 29–66 (1999)
27. Pagh, A., Pagh, R., Rao, S.S.: An optimal Bloom filter replacement. In: SODA, pp. 823–829. SIAM (2005)
28. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM J. Comp. **31**(2), 353–363 (2001)
29. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: European symposium on algorithms, pp. 121–133. Springer (2001)
30. Panigrahy, R.: Efficient hashing with lookups in two memory accesses. In: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 830–839. Society for industrial and applied mathematics (2005)
31. Pătraşcu, M., Thorup, M.: Dynamic integer sets with optimal rank, select, and predecessor search. In: 2014 IEEE 55th Annual symposium on foundations of computer science, pp. 166–175. IEEE (2014)
32. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: International colloquium on automata, languages and programming, pp. 357–368. Springer (2003)
33. Schmidt, J.P., Siegel, A., Srinivasan, A.: Chernoff-Hoeffding bounds for applications with limited independence. SIAM J. Discr. Math. **8**(2), 223–250 (1995)
34. Siegel, A.: On universal classes of extremely random constant-time hash functions. SIAM J. Comp. **33**(3), 505–543 (2004)